Cognitive Interaction with Robots

ChessMate

A Robotic Chess Player

Master in Artificial Intelligence

Fall Term 2023/2024

Authors: Blanca Llauradó Crespo

Benjamí Parellada Calderer

Sonia Rabanaque Rodríguez

Tutor: Dr. Anaís Garrell

Date: March 11, 2024

Contents

Abstract

ChessMate introduces a novel Human-Robot Interaction (HRI) system designed to enhance the traditional game of chess with the sophistication of advanced robotics and intelligent system design. The system incorporates a state-of-the-art Universal Robots UR3e arm to execute chess moves, offering a tangible, interactive experience distinct from virtual platforms. A RealSense D435 camera, strategically positioned above the chessboard, ensures accurate real-time tracking of the game state, while a user-friendly GUI allows for intuitive interaction with the system. ChessMate is versatile, supporting multiple game modes including Human vs. Human, Human vs. Engine, and Engine vs. Engine, appealing to a wide spectrum of chess enthusiasts and technophiles alike. This project delves into the system's architecture, its operational mechanics, and the innovative methodologies employed to create an immersive chess-playing experience. It further explores the system's performance, user satisfaction, and response times through comprehensive testing, offering a detailed analysis of the user experience in comparison to traditional virtual boards. ChessMate's development reflects a confluence of robotics, computer vision, and user-centric design, marking a significant advancement in the realm of interactive gaming and HRI.

1 Introduction

ChessMate represents an innovative Human-Robot Interaction (HRI) system developed to enhance the classical game of chess through advanced robotics and intelligent system design. The project aims to merge the traditional real-life experience of playing chess with the modern capabilities of robotics and computer science, providing an engaging system for players to play against a robotic opponent.

Central to the ChessMate system is the Universal Robots UR3e, a state-of-the-art robotic arm known for its precision and fluidity. It serves as the executor of physical chess moves on a standard-sized chessboard, each square measuring 4×4 cm. The robotic arm's ability to accurately replicate human-like moves on the chessboard transforms the playing experience, making it more dynamic and engaging, when compared to playing on a computer screen.

To capture and analyze the state of the chessboard, a RealSense D435 camera is strategically mounted above. This high-definition camera feeds real-time imagery to the processing unit, ensuring every piece's position and every move made are accurately recognized and reflected in the system's internal game state. This continual feed allows for real-time interaction for the human on the physical chessboard. Alternatively, users can utilize voice commands for a hands-free experience, where spoken moves are interpreted and executed by the robotic arm.

Interaction with ChessMate is facilitated through a user-friendly graphical user interface (GUI), catering to various user preferences and scenarios. The GUI provides a real-time digital mirror of the physical chessboard, allowing users to input moves with simple clicks.

The versatility of ChessMate is further evident in its support for multiple game modes, addressing different user needs and interests:

- **Human vs. Human:** Facilitates a traditional game of chess, where the robotic arm *could* move pieces for both players, enhancing the physical interaction without altering the classic player vs. player format. Alternatively, the system could be used to record moves in a digital format automatically, through the vision system.
- **Human vs. Engine:** Allows a single player to challenge advanced chess engines like Stockfish [1] or Leela Zero [2]. Players can select the engine and adjust its difficulty through the system's GUI for a tailored challenge.
- **Engine vs. Engine:** Enables two computer engines to compete against each other, providing users with the opportunity to observe high-level strategic gameplay and learn from AI decisions.

This report provides a comprehensive examination of the ChessMate system, detailing its components, operational mechanics, and the technologies that enable its functionality. We will explore the system's architecture, discuss its interaction modes, and review the methodologies implemented for game control and robotic movement. The subsequent sections will offer a deep dive into the system's design and the innovative approaches used to create an engaging and interactive chess-playing experience.

1.1 Research Question and Hypotheses

Research Questions:

Primary Research Question:
Do users experience higher levels of enjoyment and engagement when playing chess with a physical, robotic system compared to a virtual board?

Secondary Research Question:
Is age a significant factor in determining user enjoyment and engagement in playing chess with a physical, robotic system versus a virtual board? Unfortunately, due to the lack of diversity in our sample population, we will not be able to rigorously test this question. However, it remains an interesting area for future research.

Hypotheses:

- **Tactile Preference Hypothesis:** The physical, robotic chess system will be particularly favored by users who value tactile sensation and the physical presence of chess pieces. This preference is hypothesized to stem from the tangible interaction and sensory feedback that the robotic system provides.
- **Digital Familiarity vs. Physical Interaction Hypothesis:** Despite a general familiarity and comfort with digital interfaces, users will exhibit a significant preference for the interactive and tangible aspects of the robotic chess system over the virtual board. This hypothesis acknowledges the prevalent use of digital solutions while asserting the unique appeal of physical, interactive systems.
- **Novelty and Engagement Hypothesis:** The novelty and technological sophistication of the robotic system are hypothesized to initially draw user preference. However, repeated use is necessary to determine whether the preference is primarily driven by novelty or if it evolves into genuine, sustained enjoyment and engagement.

These hypotheses aim to explore various facets of user experience, addressing the potential influences of tactile interaction, the balance between digital convenience and physical engagement, and the role of novelty in user preference. By examining these hypotheses, we seek to understand the underlying factors that contribute to user enjoyment and engagement in the context of robotic versus virtual chess systems.

2 System requirements

The main objective of the ChessMate system was to create an interactive human-robot platform specifically designed for playing chess. In particular, the concept centers around enabling individuals to physically interact with the system by playing a game against it. The system is designed to engage with an actual physical chessboard and pieces, allowing for a tactile gaming experience. Moreover, the system is versatile, offering additional modes of play. It allows for traditional human versus human matches, as well as the option for entirely autonomous gameplay, where two chess engines face off against each other. This flexibility enhances the system's appeal, catering to a wide range of chess enthusiasts and technology aficionados.

In all scenarios, our primary aim has been to streamline the interaction between users and the system, ensuring simplicity and clarity to enhance user experience. When it comes to human interaction, it essentially revolves around playing the game, and we have established two types of scenarios for this. In the first scenario, the individual physically makes their move on the chessboard. The system then detects and analyzes the move to formulate its response. In the alternate scenario, the user verbally communicates their desired move to the ChessMate system. Upon receiving the vocal command, the system interprets the instruction and physically executes the move on the board using a robotic arm, effectively translating the user's directive into action.

Regarding system interaction, it primarily involves maneuvering the chess pieces via the robotic arm and utilizing the graphical user interface (GUI). Specifically, we employ the UR3e robot arm

equipped with a gripper to reposition the pieces. This setup ensures the robot reaches every spot on the board while carefully handling the pieces to avoid knocking them over during moves or when placing them back in their respective slots. On the GUI front, we display the most recent move, indicate the current player's turn, and provide vital game status updates, including checks and checkmates, all to keep the user fully informed and engaged with the game's progress.

Although the human-robot interaction system has been a crucial element, the capabilities of the system itself also play a significant role. Specifically, we have equipped ChessMate with the ability to autonomously determine its next move, considering the current state of the game at any given moment.

The designed system relies on various critical components to function properly, and any alteration to these elements would require corresponding system modifications to adapt to new conditions. Specifically, the ChessMate system is optimized for operation in controlled, laboratory-like environments due to the precise safety requirements needed for the robotic arm's operation, which moves pieces on the physical chessboard. Additionally, the system necessitates a camera and/or microphone to capture the user's movements and voice commands. For the camera, environmental factors such as lighting, shadows, and reflective surfaces need careful consideration to ensure accurate detection. Similarly, for the microphone to effectively interpret commands, it is essential that the user's instructions are articulate and the surrounding environment is free from disruptive background noise. These conditions are integral for the system's optimal performance.

Upon completing the development of the ChessMate system in line with our goals, our next step is to evaluate its effectiveness and user satisfaction. For this goal, we conducted an experiment to observe the outcomes of human-robot interactions. The central research question focuses on whether individuals find the system engaging enough to desire repeated use and if they prefer it over traditional electronic applications for chess. Our primary hypothesis is that the physical, robotic chess system ChessMate will be especially preferred by users who value the tactile sensation and the physical presence of the chess pieces. While users might be more accustomed to digital interfaces, they will still show a significant preference for the interactive and physical aspects of the robotic chess system over the virtual board. The novelty and technological aspect of the robotic system will contribute to its preference, but repeated use will confirm if the preference is due to novelty or genuine enjoyment.

For this study, we have identified certain variables to consider. Regarding independent variables or factors, we initially considered both the age of participants and their chess ratings. However, due to challenges in measurement and recruitment, we finally conducted the experiment with a group of 10 people aged between 20 and 30 years old.

On the dependent variable side, we aim to measure several aspects: player satisfaction, system usability, response time, and incidence of errors or malfunctions. Our objective is to analyze how these factors are perceived across the different age groups to gain insights into the system's overall acceptance and areas for improvement. We have gathered these qualitative metrics by conducting surveys after participants play a match against both the ChessMate system and a traditional PC screen.

As a final qualitative assessment, we will record the number of moves executed incorrectly by both systems. For the image recognition component, we'll tally the moves that are accurately recognized versus those that are misinterpreted by the system, saving this data for subsequent analysis. A similar approach will be taken for the audio recognition component. Lastly, we will track and quantify any errors made by the robotic arm, such as inaccurately placing the pieces, to further understand and refine the system's performance.

3 Description

3.1 System Setup Overview

The ChessMate system comprises various hardware components integrated to create a seamless and interactive chess-playing experience. The setup is designed to ensure smooth operation, precise movements, and reliable game management. The following figure illustrates the complete setup of the system.

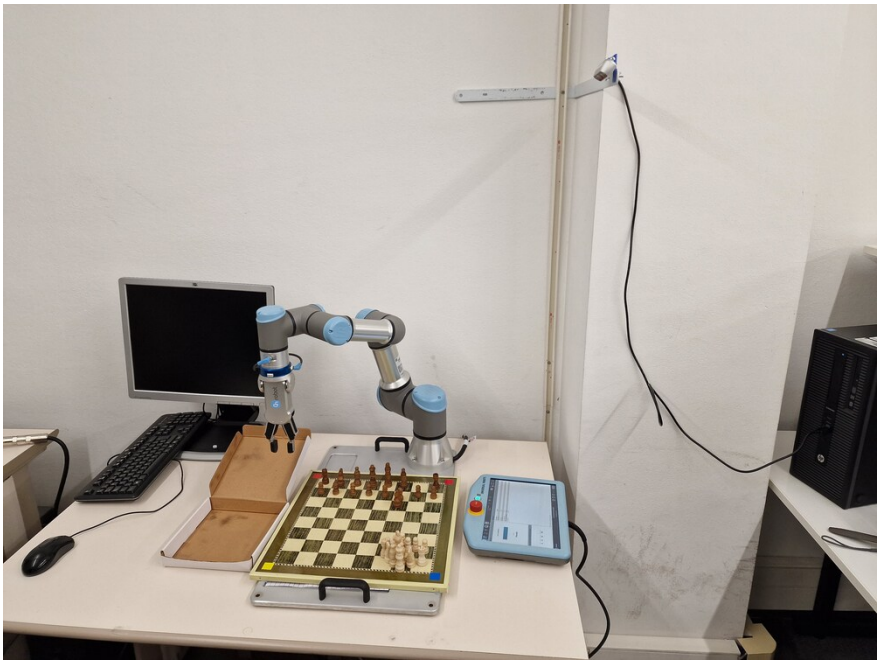


Figure 1: Comprehensive view of the ChessMate system setup.

To replicate or understand the ChessMate system, the following components are integral:

- Universal Robots UR3e: This robotic arm is crucial for the physical movement of chess pieces on the board.
- Chessboard: A standard chessboard where each square measures 4×4 cm, suitable for the robotic arm's precision.
- Chess Pieces: Standard chess pieces that the robotic arm will move during the game.
- Dummy Box: A designated area next to the chessboard where captured chess pieces are placed.
- Bracket: A sturdy 30×50 cm bracket, typically used for mounting microwaves, repurposed here to hold the camera above the chessboard.
- RealSense D435 Camera: This camera captures the state of the chessboard. While we use the RealSense D435 for its quality and reliability, the depth feature isn't utilized, allowing for flexibility in camera choice. The camera should be mounted on the bracket directly above the center of the chessboard, providing a clear and unobstructed view of the entire board. The current setup is at 1 meter height, but we have seen the system work well at 30 cm already. The main issue was that the movement of the robot made the camera wobble, so we had to fix it at a fixed spot. We 3D printed a support to connect the camera to the bracket, allowing it to rest at around a 25° angle from the board.

- USB-A 3.0 to USB-C Cable: Ensures fast and efficient data transfer from the camera to the computer, necessary for high-resolution, real-time video capture.
- Computer: A robust computer with USB A connectors is recommended to handle the data processing and game management. The computer should support USB 3.0 to ensure the camera operates at its full potential, capturing 30 frames per second without lag.
- Audio: It should be noted that there is no microphone visible in the image. This omission is intentional, as the audio capabilities of the system did not achieve sufficient reliability during the development phase to be incorporated in the final testing of the system.

This setup is crucial for the ChessMate system's operation, ensuring that the physical movements, detection, and game strategy components work in harmony.

3.2 Game Control

The system's game control architecture is designed to accommodate different modes of play and input/output methods based on the selected game mode and available resources. The core functionalities are managed by the Controller class, which processes all moves, updates the GUI, and sends messages to the robotic arm as needed. Here are the specifics of game control based on different contexts:

Move Processing: The Controller class is responsible for validating and processing each move. It ensures that all moves are legal within the constraints of the current board configuration. Once a move is verified:

- The GUI is updated to reflect the latest board state.
- If playing against a computer, the system schedules the next move query from the chess engine.
- The Robotic Arm controller, when active and not in test mode, receives instructions for the next move to execute.

Robotic Arm Movement: The robotic arm is engaged under the following conditions:

- The Robotic Arm is connected.
- The chess engine made a move **or** a human move is detected through audio, and it's the human's turn.

Audio and Video Processing: The system's responsiveness to audio and video inputs is dependent on the game mode and the availability of hardware:

- Audio input is activated during *Human vs. Human* or *Human vs. Engine* modes when it's the human's turn, provided a microphone is present.
- Video input is considered in the same modes and conditions as audio, contingent on the availability of a camera.
- The *Engine vs. Engine* mode functions independently of the camera and microphone.
- In the absence of all peripherals, the system still supports all modes of gameplay through the GUI.

Operational Setup: To mitigate the complexity and ensure the safe operation of the Robotic Arm, several parameters are hardcoded within the main file. This approach minimizes the risk of improper system initialization and ensures that all I/O components are correctly configured before gameplay. As such, an operator is required for the initial setup and monitoring, restricting direct user access to these settings. This requirement is in place to avoid accidental misconfigurations and to maintain system integrity and safety.

Chess Engine

To enhance the interactive experience and challenge of the game, our system incorporates two advanced chess engines: Stockfish and Leela Zero. Each engine offers a unique approach to game analysis and strategy, providing users with a diverse range of gameplay experiences.

Stockfish [1] is renowned for its deep classical chess algorithms and immense brute-force computational power. It evaluates millions of positions per second and employs highly optimized algorithms and data structures to analyze future moves. Stockfish's strength lies in its ability to calculate complicated tactical sequences and understand material imbalances very precisely, making it one of the strongest traditional engines available. Its efficiency and depth make it an excellent choice for real-time analysis and providing a challenging opponent to players of all levels.

Leela Zero [2] represents a paradigm shift in chess engine design, utilizing machine learning techniques inspired by DeepMind's AlphaZero [3]. Unlike Stockfish, Leela Zero does not rely on brute force calculation, but instead uses a neural network to evaluate positions and determine the most promising moves. Trained through self-play and reinforcement learning, Leela Zero has developed a highly intuitive and strategic style of play, often resembling human grandmaster thinking patterns. Its positional understanding and long-term strategic planning provide a distinctly different challenge compared to traditional engines.

In our system, users can choose to play against either Stockfish or Leela Zero, each offering a unique chess-playing experience. The engines are integrated into the Controller class, which manages the communication between the GUI, user inputs, and the engine's move recommendations. When playing against an engine, the system considers the current board position, sends it to the selected engine for analysis, and then receives and executes the recommended move. This seamless integration ensures a smooth and challenging gameplay experience, whether the user is practicing, learning, or seeking a formidable chess opponent.

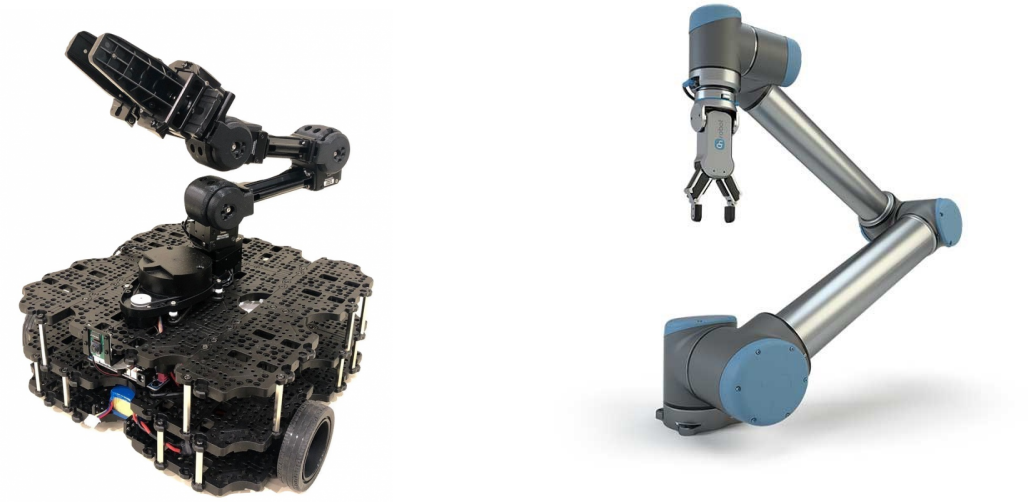
By incorporating both Stockfish and Leela Zero, our system caters to a wide range of player preferences and skill levels. Users can enjoy a more traditional, calculation-heavy approach with Stockfish or explore the nuanced, strategic gameplay offered by Leela Zero. This choice empowers users to tailor their playing experience to their personal style and goals, making the robotic chess player not just a technological marvel, but also a versatile tool for enjoyment and improvement in the game of chess.

3.3 Robotic Arm

The purpose of the Robotic Arm is to perform moves as if one were playing against a human opponent on a physical chessboard. Thus, it should be able to perform any chess move a human does. To ensure precise and safe movement, the robotic arm and its gripper are properly configured to prevent unintended operations while maintaining the integrity of the system.

During the initial phase of the project, the *TurtleBot3 Waffle Pi* (Appendix A) was employed, recognized for its ROS standard platform and powered by the *Logo* programming language. It was developed to surmount the limitations of earlier models while responding effectively to user needs. The *Turtlebot3* is described as a "*small, affordable, programmable, ROS-based mobile robot aimed*

at education, research, hobby, and product prototyping. It seeks to minimize the platform's size and cost without compromising its functionality or quality, all while enhancing its expandability" [4]. It offers the flexibility of customization with modular mechanical parts, enabling applications such as object manipulation when integrated with an *OpenMANIPULATOR*. However, faced with challenges in gripper configuration resulting in unexpected behaviors, the project pivoted to the *Universal Robots UR3e* model. The UR3e excels in performing precise tasks within tight workspaces, significantly boosting the system's reliability and functionality, and benefiting our use case.



(a) TurtleBot3 Waffle Pi [5]. (b) Universal Robots UR3e [6].

Figure 2: Comparison of initial (left) and eventual (right) robotic arms used in the system.

The Universal Robots UR3e features six rotational joints: base, shoulder, elbow, wrist 1, wrist 2, and wrist 3 (Figure 3), each offering a working range of $\pm 360°$, except for wrist 3, which boasts an unlimited range of motion. The first three joints, constituting the robot's primary movement components, achieve a maximum speed of $\pm 180°/s$. In contrast, the remaining wrist joints are capable of speeds up to $\pm 360°/s$, accommodating rapid and precise maneuvers. The 'infinite' rotational capability of wrist 3 is crucial for complex tasks such as automated screw driving, enhancing the robot's versatility and efficiency.

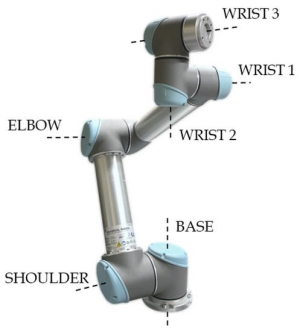


Figure 3: The six rotating joints for the UR3e [7].

This feature allows for the seamless integration of various tools directly onto the robot, streamlining tasks and facilitating easier programming and quicker setup. The gripper, an essential component of the arm, benefits significantly from this design, enabling precise and diverse manipulations.

Programming and control are managed via a 12-inch teach pendant touchscreen, which employs the user-friendly PolyScope graphical user interface. This setup not only simplifies the operation of the robot, but also ensures safety and efficiency, particularly in environments where human-robot interaction is frequent, such as ours.

OnRobot RG2 & RG6

The RG Series Gripper is a dynamic and adaptable collaborative gripper, equipped with an integrated Quick Changer. The RG2 model, designed to manage payloads of up to 2 kg, offers a stroke length of up to 110 mm. On the other hand, the RG6 model is capable of handling heavier payloads up to 6 kg and features a longer stroke length of up to 150 mm, while also including a built-in Quick Changer. Their customizable fingertips add to the gripper's versatility, allowing for modifications to suit different aspects of the production process and thereby enhancing overall robot utilization [8].

The RG2 gripper was chosen for its adaptability and precision, offering a variable stroke length and resistance-sensitive gripping that suits the diverse dimensions of chess pieces. Its efficient operation and flexible grip provide the necessary precision and reliability required for the delicate task at hand.

UR3e Communication

Integrating the Universal Robots UR3e with Python involves a few critical steps. Programming at the Script Level requires creating a client application in Python, which communicates with the robot's URControl via TCP/IP socket connection [9].

- `Connection`: To establish a connection, it's necessary to know the robot's IP address and the specific PORT based on the desired interaction. The e-Series robots have two distinct modes: remote, where URScript commands are accepted, and local, where such commands are not accepted and connections to primary ports are disconnected. For continuous status monitoring in both modes, it's advisable to use ports 30004, 30011, 30012, and 30013 [10]. In this project's context, where remote mode is primarily used, PORT 30002 is strongly recommended by the manual.
- `Script Level`: After establishing a connection, URScript programs or individual commands can be sent as plain text over the socket, ending each line with a newline character (`\n`). The `movej` command is frequently used to direct the arm to a specific joint-space position. This command requires parameters such as `q` (joint positions or pose, where inverse kinematics calculates the corresponding joint positions), the `a` (referring to the joint acceleration of the leading axis in rad/s^2), the `v` (joint speed of the leading axis in rad/s), the `t` (time in seconds) and the `r` (blend radius). Notably, specifying the time (`t`) makes the command ignore the acceleration (`a`) and velocity (`v`) parameters.

  The `get_inverse_kin` command is crucial for calculating the transformation between tool space and joint space, considering the tool pose (`x`) and a list of near joint positions (`qnear`). The detailed functionalities of these commands are further explored in section 3.3.

OnRobot RG2

Extending the control to the OnRobot RG2 gripper, it's expected to behave cohesively with the robotic arm.

- `Connection`: Connection parameters such as IP and PORT are consistent with those used for the arm. However, a reconnection process is required for sending scripts to the gripper, involving closing and reopening the socket connection.
- `Script Level`: Unlike the relatively straightforward `movej` command for the arm, operating the gripper involves transmitting a complete configuration script. The `open` and `close` functionalities are initially set up in the Tech Pendant and then relayed to the gripper via Python, which involves reading and dispatching the script configurations line by line. This approach ensures that the gripper operates smoothly and in sync with the arm's movements, handling the chess pieces with the necessary precision and care.

Waiting time

One of the main issues in sending instructions to the robot involves the handling of transmission and processing. If this duration is too short, the robot may exhibit unexpected behavior due to the insufficient processing time. This can result in accidental misconfigurations, leading to incomplete actions, or worse yet, to potentially dangerous movements for the user.

Therefore, to mitigate these risks, it's critical to adequatly manage the `t` parameter in the `movej` command, which dictates the time allocated for the move. Moreover, implementing a deliberate pause or sleep period before dispatching the gripper's configuration commands is essential. This precaution ensures that the robot has sufficient time to safely and accurately execute the preceding instructions. The sleep duration should ideally match or exceed the specified `t` value, providing a buffer for complete and secure operational performance.

Coordinates, joints & orientations

The e-Series collaborative arms from Universal Robots are engineered to replicate a human arm's versatility and range, consisting of interconnected tubes and joints. PolyScope, the standard software accompanying these cobots, orchestrates the coordination of these joints to maneuver the robot's tool to the desired position and orientation. As depicted in Figure 4, the `X`, `Y`, and `Z` coordinates determine the Tool Center Point location, while `RX`, `RY`, and `RZ` articulate its orientation. Notably, when all coordinate values are set to zero, the Tool Center Point aligns precisely with the tool output flange's center, reflecting the coordinate system's origin.

The robotic arm's movements are also governed by joint limits, which define the maximum and minimum extent of movement for each joint, reflecting the robot's physical and safety constraints. Once a joint reaches its limit, it can move no further [11].

Utilizing inverse kinematics we can obtain the desired end-state, namely the position and orientation of the robot's end effector (hand). This approach simplifies the process by handling the complex calculations required to determine joint movements. It takes into account both Cartesian coordinates for spatial positioning and orientation angles: roll (rotation around the X-axis), pitch (rotation around the Y-axis), and yaw (rotation around the Z-axis).

The method involving direct joint control is centered around defining the specific angles or positions for each of the robot arm's joints. This approach proves advantageous in situations where the trajectory of the end effector is secondary to its final destination, or when increased precision is needed. Consequently, inverse kinematics is essential as it simplifies the arm control.

Thus, for this project, employing joint control is vital to achieve precise and controlled movements on the chessboard. All commands utilize the `get_inverse_kin` function, which translates the `X`, `Y`, and `Z` coordinates, as well as the necessary orientations:

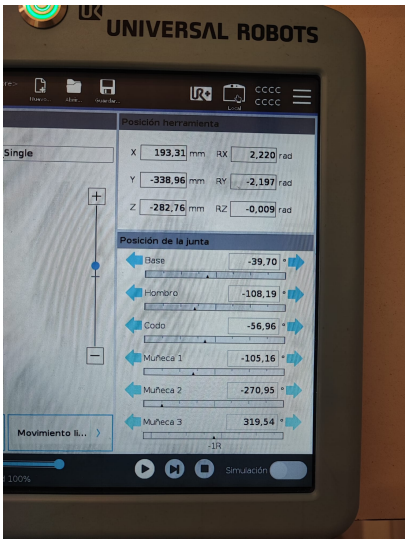


Figure 4: Display of coordinates, orientations, and joint angles on the Tech Pendant.

- `Decode Movement`: Since movements come in as strings, and the chess library uses numbers for each tile, the first task is converting these strings into a numeric format that the library understands.
- `Initial values`: We select the initial position of the arm carefully to avoid interfering with the camera's view. It's crucial to choose coordinates and joint angles that keep the robot's movements out of the camera's field of view.
- `Quadrants`: This aspect ensures that pieces are picked up and placed correctly, preventing them from falling. Instead of a fixed setup for all 64 tiles, we group them into sets of 4. Each group shares the same joint angles and orientations, organized in a matrix as shown in Figure 5:

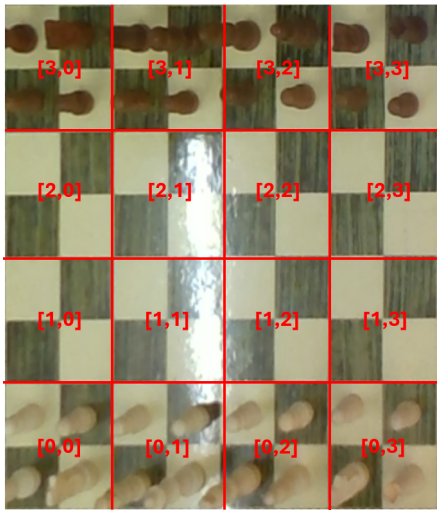


Figure 5: Quadrants created to handle the joints and the orientations.

- `Winning Orientations`: Building on the quadrant system for managing orientations and joint angles, we pay particular attention to the extremes of the board (rows 1 and 8). These edge areas require careful handling during single moves, necessitating specific joint configurations to avoid hazardous movements while picking up or setting down pieces. Misalignments in these delicate zones can result in pieces being placed unevenly or even toppling.

To address these challenges, we've found that maintaining the same orientation from start to finish works well when the destination is close to the starting position. However, when a piece traverses from the first quadrant to the last (`quadrant[0] == 0 to quadrant[0] == 3`), or vice versa, special rules apply. The programmed behavior is as follows, where `source` refers to the initial row in the movement, and `target` to the final one:

- `(source == 0 or source == 3) and (target > 0 and target < 3)`: When a move starts from an edge and ends somewhere in the middle, the piece should maintain its initial orientation throughout the move.
- `(source > 0 and source < 3) and (target == 0 or target == 3)`: Conversely, if the move begins in the middle and concludes at an edge, the piece should adjust to the orientation of the target edge when it's picked up.

Nevertheless, some scenarios are more complex, such as moving a rook from the first to the last row. These instances necessitate the consideration of an additional quadrant for the move, which varies based on the piece's starting row (Figure 6):

- `(source == 0 and target == 3)`: Here, the piece travels from row 1 to row 8, necessitating an orientation change as detailed in the "Extreme Movement" entry of this list.
- `(source == 3 and target == 0)`: Similarly, this move takes the piece from row 8 back to row 1, also requiring an adjustment according to the "Extreme Movement" guidelines.

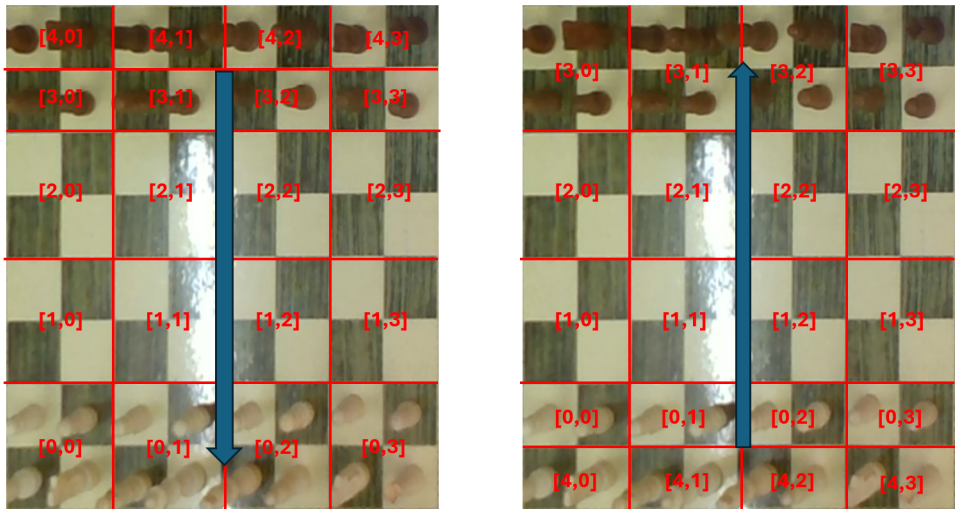


Figure 6: Illustration of what are considered special cases and need the "Extreme Movement" help.

- `Dummy Box`: This is a predefined spot where the robot places the pieces that have been captured during the game. Essentially, it's a holding area for pieces not currently in play. The robot deposits a captured piece here before proceeding to move another piece. In this project, we've conveniently located the dummy box at the robot's initial position to avoid superfluous movements. However, this location can be adjusted according to the specific needs or constraints of the setup.

- `Extreme Movement` The previous configurations cover a wide array of moves and work well most of the time. However, some unique scenarios, such as relocating a rook from the first row directly to the last row, demand special attention. In these instances, the usual piece orientations are impractical as they might lead the piece to be angled too steeply, raising

the risk of it tipping over when released. To address this, we've adopted a strategy of rotating the robot's wrist 3 component by approximately 180 degrees for such extreme moves. This adjustment aims to keep the chess piece as vertical as possible during the transition, significantly reducing the chance of it toppling over upon release.

Behavior

In this section, we detail the various operational modes devised for the robot and the respective modifications in each scenario. It's crucial to understand that robot movements are dictated by the desired positions from the Tool Center Point alongside specified orientations and joints to facilitate smooth moves.

To avoid displacing pieces during transit while ensuring the system's ability to pick up and place the pieces, we have established two primary heights known as z_{up} and z_{down} respectively. Moreover, for handling pawns, which are typically smaller, we've introduced an additional, slightly lower height to enhance the robot's capability in manipulating these particular pieces, thereby preventing any mishandling during pickups or placements.

We identify two main behavior patterns depending on whether the robot intends to move a piece to an unoccupied square or to a square occupied by an opponent's piece:

- In scenarios where the robot needs to move a piece to an empty square, it follows a sequence of steps: It moves above the piece at z_{up}, descends to z_{down}, secures the piece by closing the gripper, lifts the piece by ascending to z_{up}, travels to the target square at a safe height, descends to z_{down}, releases the piece by opening the gripper, ascends back to z_{up}, and finally returns to its initial position.
- When the target square is occupied by an opponent's piece, the robot must first remove the opposing piece. It begins by navigating to the opponent's piece at z_{up}, then descends to z_{down}, secures the opponent's piece by closing the gripper, lifts it to z_{up}, moves to the discard box, releases the piece by opening the gripper, and then follows the standard procedure for moving a piece to an unoccupied square as described in the previous point.

As highlighted in the previous section, defining a robot's movement involves specifying not only the target coordinates but also the configuration of the arm's joints and the gripper's orientation. For each action outlined earlier, we determine the approximate joint positions using the quadrant information. Similarly, we calculate the initial and target orientations. The challenge here is to avoid tilting the piece during transit, requiring us to establish "winning" orientations for both the starting and ending positions to ensure the piece remains stable throughout the move.

Once we have identified these parameters, we can construct the movement command for the robotic arm. This command will include the target coordinates, the approximate joint positions from the quadrant data, and the required tool orientation, which may differ slightly from the quadrant's general orientation to accommodate the specific piece and movement.

An important operational detail is the time factor: when instructing the robot to execute a movement, we define a duration for the action (default is 2.5 seconds). To prevent overwhelming the system with rapid, successive instructions, we incorporate a sleep interval in the code. This ensures that each movement is fully executed before the next command is issued, maintaining a smooth and orderly sequence of actions. This careful timing helps to avoid any potential errors or collisions, contributing to the overall reliability and efficiency of the robotic system.

Finally, there are some movements that we consider special and that differ from those explained at the beginning of this section:

- `Castling`: This special move begins by relocating the king to its designated empty square using the established regular behavior. Following this, the robot shifts the rook to its new position adjacent to the king, mirroring the regular move sequence. Thus, castling involves the system performing two consecutive moves to vacant positions within the same turn, without returning to the initial position in between moves.

- `Checkmate`: Here, the piece is initially moved to the intended square, either empty or occupied, using regular movement patterns. Afterward, to symbolize the sweet victory of the robot, the arm picks up the piece from the position where it has just placed it, and uses it to remove the rival king, moving it on top of it. To do the latter, it approaches the king at a mid-level height and then lowers it down (z_{down}) to its new resting position, toppling the king and conclusively marking the checkmate.

3.4 Communication with the Controller

In this section, we'll clarify the operational behavior of the system, focusing on the interplay between the robot and the Python Controller as depicted in the overall schematic (Figure 7). This interaction is critical for initiating and progressing the game seamlessly.

Initially, both the robot and the Controller are set up and activated. A socket connection is established between them to facilitate real-time communication. As the game initiates, the Controller is responsible for sending moves to the robotic arm, which could originate from a processed engine decision or an audio input, as further explained in Section 3.7. Each move undergoes a validation process to ensure legality and to check the occupancy status of the target square. Following the validation and transmission of a move, the chessboard display is updated to reflect the current game status. Afterward, the robot arm will stay on standby awaiting a new move from the Controller, ready to seamlessly continue the game with precision and efficiency.

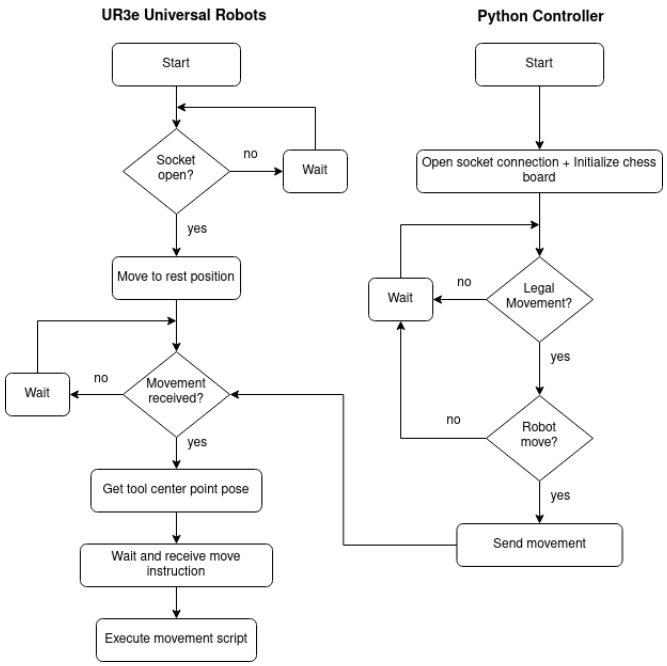


Figure 7: Overall schema detailing the interactions between the robot and the Python Controller.

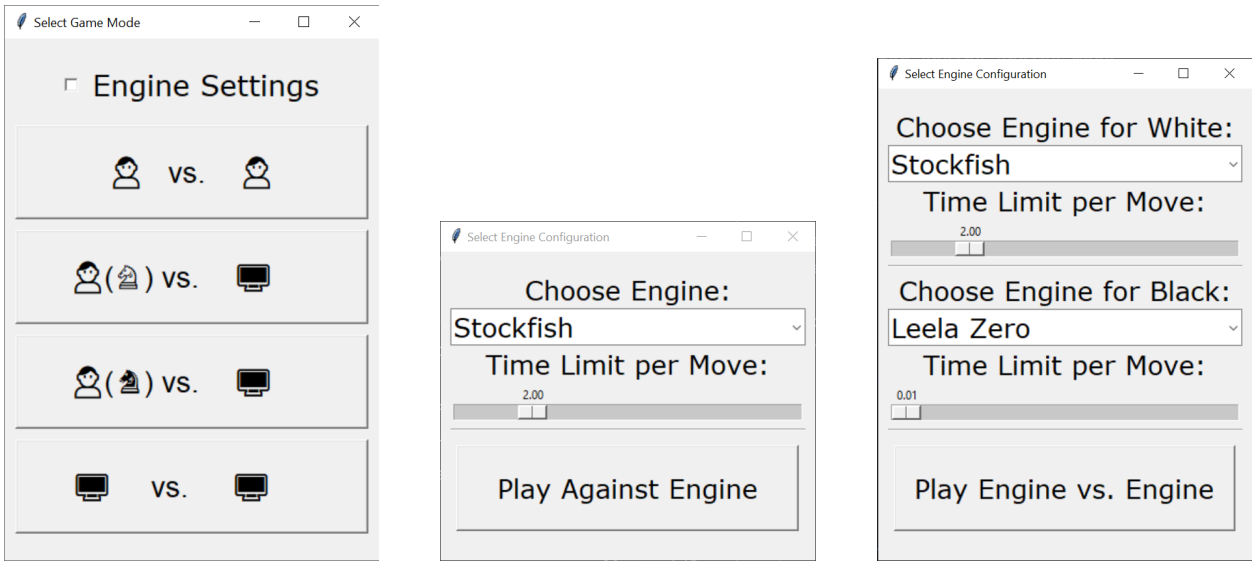
3.5 Graphical User Interface

To facilitate robust communication between the user's input, the robot's actions, and the internal chess logic, the development of a comprehensive Graphical User Interface (GUI) was essential. The GUI serves as an interactive platform ensuring synchronization between the user's moves and the system's internal state, accurately reflecting the physical reality of the chessboard.

The GUI provides an interactive chess-playing experience, enabling users to execute moves, undo past actions, and correct any discrepancies between the perceived and actual state of the game. This feature is particularly crucial in maintaining the integrity of the game flow, especially when integrating human input with automated chess movements.

Game Mode

Upon initialization, users are greeted with a menu offering various gameplay modes, as depicted in Figure 8. Available modes include *Human vs. Human*, *Human (White) vs. Computer*, *Human (Black) vs. Computer*, and *Computer vs. Computer*. The use of universally recognizable emojis in the menu design aims to make the system accessible and understandable to a broad audience, transcending language barriers. When the user opts to play against a machine and chooses *Engine Settings*, a new window appears, prompting the selection of the engine, and a time limit for each turn in seconds. If not, default settings are applied, which are the ones shown in the figure.



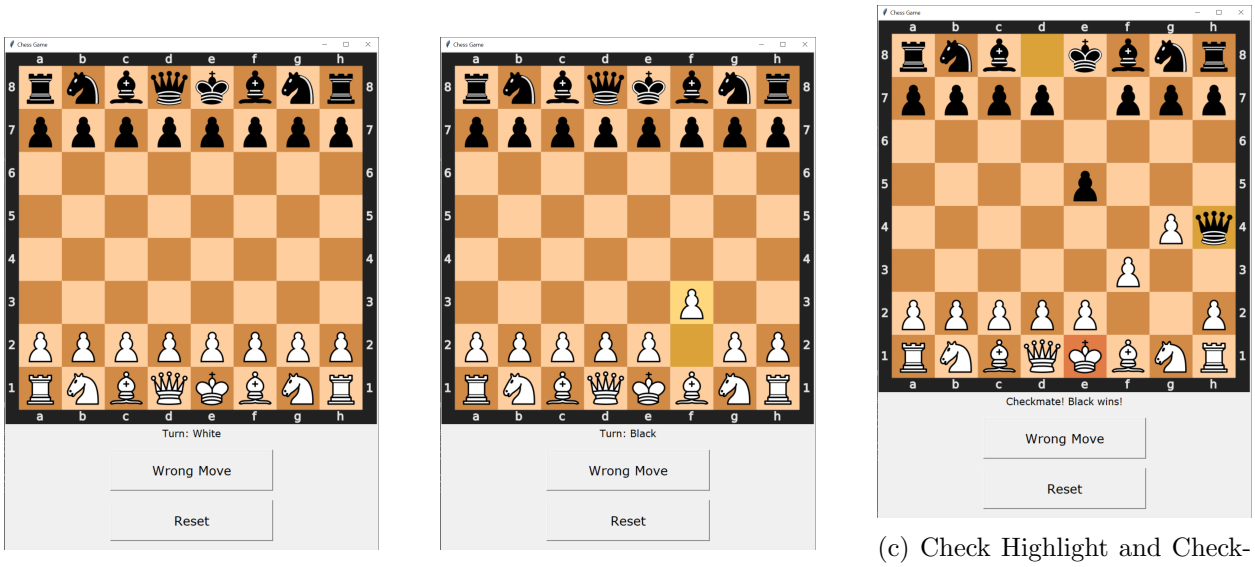
(a) Menu (b) Engine Settings I (c) Engine Settings II

Figure 8: Game mode selection menu with emoji representation for each mode. If an engine opponent has been selected and "Engine Settings" toggled, a second window will open, so the user can choose the game engine to play against. If any of the two *Human vs. Machine* is chosen, window b) appears, while if *Machine vs. Machine* is selected, window c) appears.

Chessboard

Upon selecting a game mode, the primary interface of the GUI is presented, resembling traditional virtual chess platforms, as shown in Figure 9. This interface includes all the standard features expected in a digital chess game, enhancing the user's engagement and strategic planning. It visually denotes each move made by highlighting the last move, whether conducted by the computer or the human player, ensuring clarity and continuity throughout the game.

Additionally, it limits moves to only allow legal-moves, it notifies whoms turn it is and notifies if the position is in check by highlighting the king piece in red.



(a) Initial Setup (b) Last Move Highlight (c) Check Highlight and Checkmate

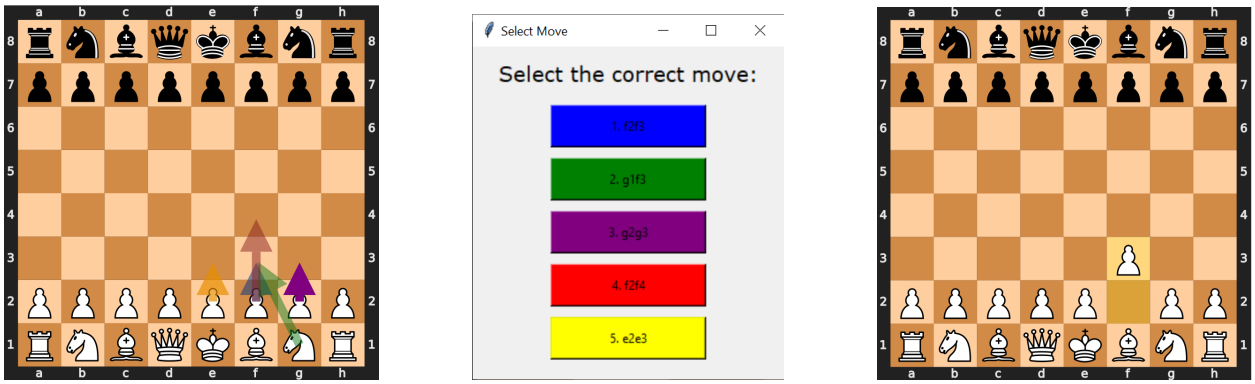
Figure 9: Sequential snapshots of the chessboard: the initial setup, a display highlighting the last move made, and a display highlighting check with a notification that it was checkmate in text.

The GUI the chess system incorporates critical features to enhance user interaction and rectify inaccuracies in real-time. One of the most significant features is the *Wrong Move* button, designed to allow users to reverse their last action. In scenarios where the system is pitted against a computer, activating this button retracts both the computer's and the user's last move. Initially implemented to address inaccuracies in camera-based move detection, the button facilitates a return to a previous state, effectively halting further input from the camera and audio systems until the user rectifies the move directly through the GUI.

This approach, however, was found to be somewhat cumbersome, particularly when playing against an automated opponent on a physical chessboard. It required users to manually reset the opponent's piece before correcting their own move in the GUI. To streamline the experience and reduce ambiguity in move selection, an enhanced feature was introduced: a disambiguation dialog (Figure 10). This interface displays up to five arrows, each representing a potential move derived from the video or audio input, ranked from most to least likely. Users can then select the correct move by choosing the corresponding arrow color from the dialog, thus updating the GUI and the system's internal state. This dialogue only appeared when the difference between the possible moves was small enough to warrant possible ambiguity, otherwise, the system selects the most likely move.

Despite these enhancements, the *Wrong Move* button remains as a critical fallback mechanism, ensuring that players can always revert to a previous state if discrepancies persist. Additionally, the system features a *Reset* button, allowing users to completely restart the game, returning all pieces to their initial positions and refreshing the game state. This functionality ensures that users have full control over the game flow and can initiate a new game at their convenience.

As a pawn reaches the opposite end of the board, the system recognizes the need for promotion. To facilitate this critical game decision, a dialogue interface is presented to the user through the GUI, prompting them to select the desired piece for promotion. This interaction ensures a smooth continuation of the game, mirroring the strategic choices a player would encounter in a traditional chess game. Figure 11 illustrates the pawn promotion dialogue, where users can choose from a queen, rook, bishop, or knight, as one would expect from any other system.



(a) Initial Ambiguity (b) Dialog Interface (c) Final Selection

Figure 10: Sequential snapshots of the disambiguation process.

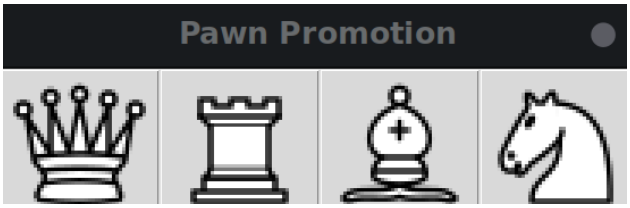


Figure 11: The GUI dialogue for pawn promotion, offering the user a choice of piece to promote to.

Threading

In the quest for a seamless and responsive user experience in ChessMate, we've employed a multi-threaded architecture to handle the concurrent processes of game state management, camera feed analysis, and audio transcription. This approach allows the system to maintain a fluid and interactive GUI, process camera inputs for move detection, and handle voice commands simultaneously without performance bottlenecks or user experience degradation.

Here's how the threading architecture contributes to the robustness and responsiveness of the GUI:

- **Main Thread for GUI**: Due to the nature of Tkinter [12], which is not thread-safe, the GUI runs on the main thread. This ensures that all user interactions and visual updates occur smoothly and without any concurrency issues. The main thread's sole responsibility is to manage user inputs, update game visuals, and provide a responsive interface.
- **Camera Thread**: The camera loop runs on a separate thread, constantly analyzing the physical chessboard's state through the camera feed. When it's a human player's turn, and a new move is detected, it is swiftly communicated to the main thread using a thread-safe command queue. This approach allows the system to continuously monitor the game's physical state without interrupting the user's interaction with the GUI. This system is paused when the robot is doing the move.
- **Transcriber Thread**: This thread is dedicated to continuously listening and transcribing voice commands. It focuses solely on capturing and processing audio data to convert spoken words into text.
- **Audio Loop Thread**: Complementing the transcriber, this thread is responsible for processing the text transcriptions into potential chess moves and communicating them to the

main system. It periodically checks the transcriptions collected by the transcriber thread, interprets them, and, if valid moves are found, forwards them to the main thread via the command queue.

- **Thread Coordination**: The `DetectionController` class coordinates these threads. It initiates and waits for the completion of both camera and audio threads, ensuring that all components of the system are synchronized and that the system can gracefully shut down when needed.

The necessity for two threads in audio processing arises from the continuous nature of audio capture: one thread is dedicated to the persistent recording of audio to ensure no spoken command is missed, while the other interprets the transcribed data into meaningful actions. In contrast, camera processing is less demanding in terms of real-time data capture, as it involves periodically capturing still images from a fixed position, providing all necessary information with intermittent snapshots rather than a continuous stream. This distinction in data capture dynamics necessitates a more complex, dual-threaded approach for robust and efficient audio handling compared to the relatively straightforward process for visual information.

By delegating intensive tasks such as camera feed processing and audio transcription to separate threads, the system ensures that the GUI remains responsive, providing a smooth and engaging user experience. This multithreaded approach harnesses the full potential of the system's hardware, allowing for efficient multitasking and a seamless blend of physical and digital gameplay.

Pipeline

The system operation pipeline of ChessMate is a comprehensive framework that coordinates between the human player, the GUI interface, and the robotic arm movements. The pipeline is crucial for ensuring that each component interacts seamlessly, reflecting the real-time state of the game accurately and responsively. Below is an overview of the process accompanying the flowchart in Figure 12.

- GUI Interaction: During a human turn, the system primarily monitors the GUI for any input or move made by the player. This involves checking for mouse clicks or selections within the GUI that represent the player's move.
- Queue Monitoring: Concurrently, the system sporadically checks a shared command queue, which might be filled by the camera or audio threads. These threads are continuously processing the physical game environment and any spoken commands, respectively, and enqueue their results for the main system to process.
- Move Validation: Once a move is detected, either through GUI interaction or from the queue, the system validates it against the current state of the chessboard to ensure it's a legal move.
- Robotic Arm Activation: If it's not the human's turn or the move comes from an audio command (implying a more hands-off approach from the GUI), the system activates the Robotic Arm. The Arm then physically moves the piece on the chessboard, reflecting the move detected from the game's digital state or audio input.
- GUI Update: Regardless of the move's origin, once validated and executed, the system updates the GUI to reflect the new state of the game. This ensures that the visual representation always matches the actual state of the chessboard.
- After executing the move and updating the GUI, the system then proceeds to the next turn, continually looping through this pipeline until the game concludes. The process is designed

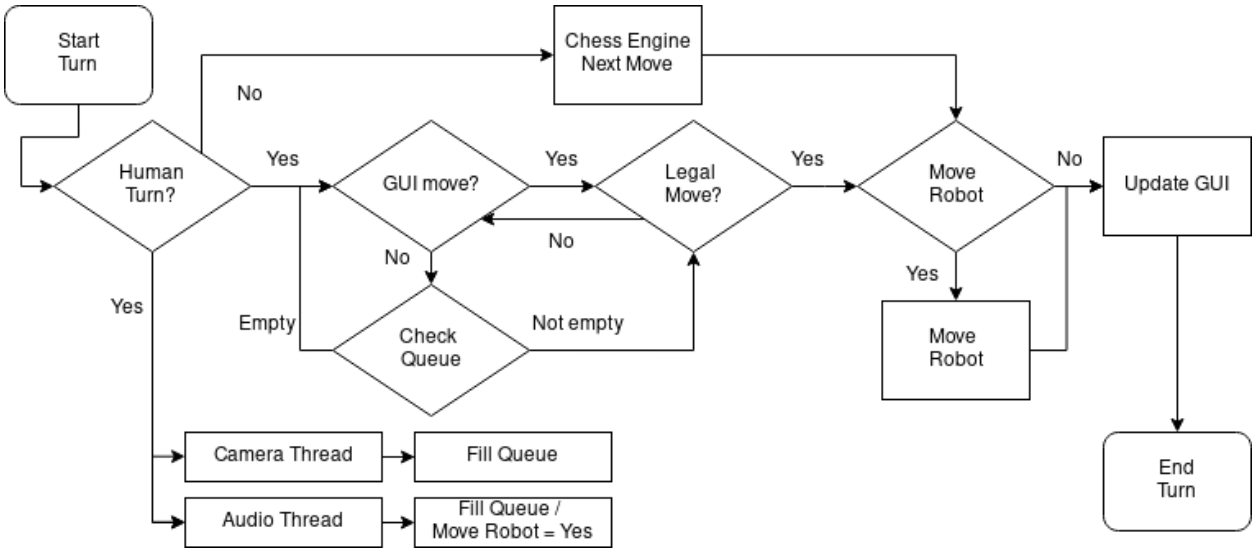


Figure 12: The operational pipeline of ChessMate, illustrating the flow from turn determination to move execution and system update.

to be adaptable, handling various types of input and switching smoothly between human and computer turns.

This pipeline demonstrates the interplay between different modules of the ChessMate system. By coordinating between human input, visual and audio processing, and robotic actions, ChessMate delivers a robust and interactive chess-playing experience.

3.6 Image Processing

This section delves into the image processing techniques applied to recognize and interpret moves on the chessboard. The essence of this process is to accurately detect the current state of the chessboard and any changes due to player moves.

Chessboard Projection

The projection of the chessboard is a critical step in accurately interpreting the current game state from the camera's perspective, which does not have a direct top-down view. Instead, the camera captures the board at an angle due to its placement, as shown in Figure 1. To counteract this and simulate a top-down view, a geometric transformation known as perspective transformation is applied.

The perspective transformation is a fundamental technique in image processing that adjusts the viewpoint from which an image is seen. In our context, it's used to correct the angled view of the chessboard to a view as if the camera were directly above the board. This correction is vital for accurate move detection and game state analysis.

The process involves several steps:

1. Identifying Chessboard Corners: The corners of the chessboard are detected in the camera's image. These points serve as the reference source for the transformation.

2. Defining Destination Points: A set of destination points are defined, representing the corners of a regular chessboard in a top-down view. These are computed from the height-width of

the real physical chessboard, where the viewport will occupy the entire board, as if it were positioned in the center above of the board.

3. Calculating the Transformation Matrix: Using the pairs of corresponding points (original corners in the captured image and the idealized top-down view corners), a 3×3 transformation matrix is computed. This matrix encapsulates the relationship between the positions of points in the camera's view and their desired positions in the top-down view. This is done using OpenCV's `getPerspectiveTransform` [13], which solves the following:

$$\begin{bmatrix} t_i x_i' \\ t_i y_i' \\ t_i \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \\ c_1 & c_2 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$\text{dst}(i) = (x_i', y_i'), \quad \text{src}(i) = (x_i, y_i), \quad i = 0, 1, 2, 3$$

and t_i represents a scaling factor applied to the coordinates as a result of the transformation.

4. Applying the Transformation: The transformation matrix is then applied to the entire image, warping the image so that the angled view of the chessboard appears as a top-down view. The specific function used for this step is `warpPerspective` in OpenCV [13], which applies the matrix to the source image to produce the corrected destination image. This is done as:

$$\text{dst}(x, y) = \text{src}\left(\frac{a_1 x + a_2 y + b_1}{c_1 x + c_2 y + 1}, \frac{a_3 x + a_4 y + b_2}{c_1 x + c_2 y + 1}\right)$$

By these means, the system compensates for the camera's placement and angle, providing a consistent and rectified view of the chessboard for further processing and move detection.

Since the camera and chessboard are stable, steps 1 to 3 need only be performed once at the initial setup. Step 4, however, must be applied in all frames where we want to detect chess movements. An example of the perspective transform can be seen in Figure 13.

This projection is crucial for the subsequent steps in the image processing pipeline, as it ensures the game's state is interpreted correctly, regardless of the camera's physical position relative to the chessboard. With a reliable top-down view of the board, the system can proceed to accurately detect and interpret moves made by the players.

In the early stages of the system development, the camera was positioned lower than its current placement, necessitating a dual-projection approach for accurate image capture and processing. Specifically, the first seven columns of the chessboard were projected using a standard transformation technique. However, due to the camera's lower height, pieces on the last column often appeared cropped or distorted. To mitigate this, an additional projection was applied specifically to the eighth column, incorporating two extra points set further back from the board to extend the field of view and capture the entire column accurately. This dual-projection method significantly improved performance and accuracy for those pieces located at the edges of the board, which are most affected by perspective distortion. Nevertheless, in the final setup, this was not needed.

Corner Detection

In the initial phase of system development, we explored the possibility of automating chessboard detection to create a more resilient system. The concept involved utilizing a pre-trained neural network to identify the chessboard [14] and subsequently transform it into a 2D projection. However, this approach proved to be excessively complex and resource-intensive for our needs. The pipeline, akin to our manual method, aimed to detect and classify each piece on the board. Moreover,

the neural network was trained on a different style of chessboard, which meant it would require significant retraining to adapt to our specific board design.

We also considered classical corner detection methods by using morphological operations, as well as leveraging techniques like SLIC [15] and manually placing color squares on the board to identify the corners. However, variability in lighting conditions and other environmental factors often led to detection failures, undermining the consistency of the projection. Furthermore, we encountered stability issues with automatic detection methods, particularly with a live camera feed. Inconsistencies in chessboard recognition and projection failures led us to reconsider the reliability of such systems. Moreover, these sophisticated methods seemed unnecessary for our purposes.

Ultimately, we settled on a user-assisted approach for corner selection at the beginning of the setup. Given the relative stability of the camera and chessboard placement, this method proved effective and reliable for the duration of a match. Users manually select four corners of the chessboard at the onset, which are then used to create a stable 2D projection for subsequent move detection and processing. An example of the perspective transform can be seen in Figure 13.

(a) Manual corner selection with points in green.

(b) Result of the projection.

Figure 13: Example result of the chessboard projection after the manual corner selection.

Move Detection

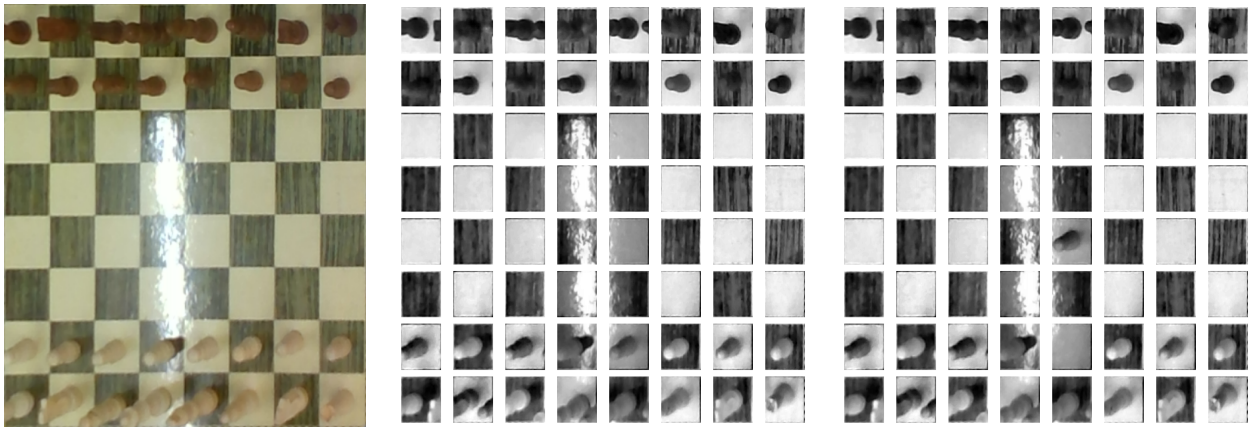
In our quest to accurately detect moves on the chessboard, we've employed several strategies to discern the type and occurrence of each move. Our initial methodology involved utilizing the YOLOv8 [16] object detection framework for identifying individual chess pieces. However, this approach lacked precision, as it required fine-tuning to our chessboard and set, and was further limited by our laboratory computers' hardware, specifically the absence of a dedicated GPU.

Furthering our efforts, we delved into keypoint detection methods, notably utilizing the Scale-Invariant Feature Transform (SIFT) technique [17]. We aimed to distinguish the chess pieces by creating a compact dataset representing different angles of all the different pieces. Unfortunately, this method also proved inadequate. The granularity and similarity of chess pieces, coupled with the limitations of SIFT in handling the low-resolution and noisy conditions of our dataset, led to unsatisfactory results.

Additionally, we experimented with Google MediaPipe segmentation [18] to individually identify

and classify each chess piece. This method aimed to understand the pieces' context and spatial arrangement. Yet, it too faced challenges, primarily due to the low quality of the images, the distance at which the images were captured, and the inherent noise present on the board–particularly from the black squares which are not uniform and possess a distinct wooden texture. These factors significantly impacted the algorithm's ability to accurately discern and segment each piece, leading to unreliable detection.

Ultimately, we opted for a straightforward strategy by initially dividing the transformed chessboard image into individual squares. Given the known dimensions of the chessboard, we precisely segmented each square, as illustrated in Figure 14.



(a) Chessboard perspective. (b) Grid separation of a). (c) Grid separation after a move.

Figure 14: Chessboard grid separated into individual squares before and after a move, illustrating the initial perspective and subsequent grid separation.

To detect changes between successive board states, we applied the Structural Similarity Index Measure (SSIM) [19] to each of the squares. Specifically, we use $1-\text{SSIM}$, where a greater difference signifies a potential change within that square. The implementation uses Python's scikit-image [20] which computes the per-pixel SSIM of two images $\mathbf{A}$ and $\mathbf{B}$ as:

$$\text{SSIM}(x,y) = \frac{(2\mu_{\mathbf{A}}\mu_{\mathbf{A}} + C_1)(2\sigma_{\mathbf{AB}} + C_2)}{(\mu_{\mathbf{A}}^2 + \mu_{\mathbf{B}}^2 + C_1)(\sigma_{\mathbf{A}}^2 + \sigma_{\mathbf{B}}^2 + C_2)}$$

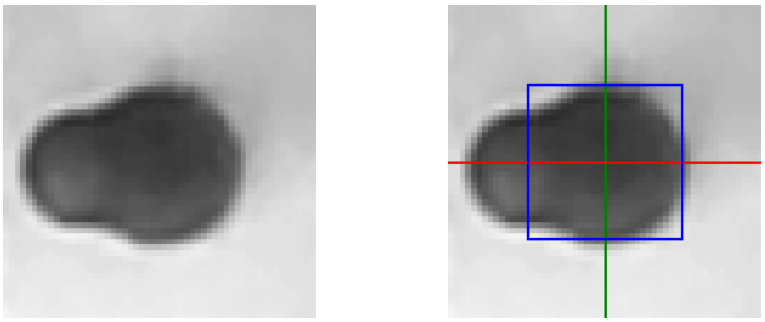
where $\mu_{\mathbf{A}}$ and $\mu_{\mathbf{B}}$ denote the mean of each image, $\sigma_{\mathbf{A}}^2$ and $\sigma_{\mathbf{B}}^2$ the variances, and the $\sigma_{\mathbf{A}}\sigma_{\mathbf{B}}$ the covariance–where all of these are functions of x, y. Constants C_1 and C_2 are included to maintain stability when mean or variance values are close to zero, calculated as: $C_i = (K_i R)^2$, where K_i is a small constant parameter, and R the dynamic range. For scikit-image, the function works per channel, thus, one should convert it to grayscale first[1]. Finally, SSIM returns an average value of all pixels representing the overall structural similarity between the two images.

To speed up the detection process, we only consider squares that correspond to legal moves based on the current state of the game. This reduces the number of squares to examine and increases the accuracy of the detection system. For example, during White's turn, only the squares relevant to White's possible moves are analyzed.

For even more precision, the system employs a cascading comparison approach. Initially, each square is checked against a predefined threshold to identify significant differences. If a square exceeds this threshold, it is further divided into sub-regions: top, bottom, left, right, and center, as seen in Figure 15. A secondary check is then performed on these sub-regions, which we will

[1]This is never specified in the original paper [19, 21], and might be unnecessary, but it is above the requirements of the project to validate the implementation.

call sub-squares, by computing the SSIM difference for each region. If substantial differences are detected in specific sub-regions, these squares are considered candidates for the move.

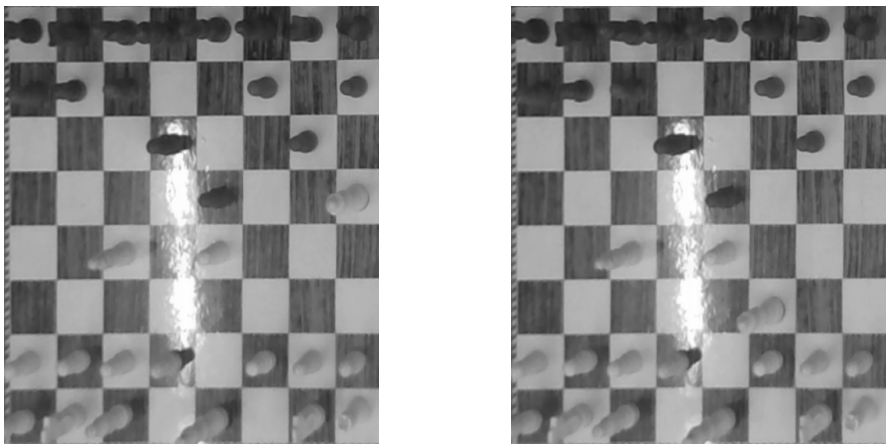


(a) Square (b) Sub-squares

Figure 15: Subdivision of a square into the sub-squares: The green line indicates the division between left and right, the red line between top and bottom, and the blue square marks the center.

The rationale behind subdividing the squares further is twofold. First, chess pieces typically rest towards the center of a square, so a significant difference in the center sub-region strongly indicates a move happened on that specific square. Secondly, due to the camera's perspective and the varying heights of pieces, some pieces may cast shadows or slightly overlap onto adjacent squares. By analyzing differences in specific sub-regions, we can more accurately discern the source of the move, enhancing the system's precision in detecting and interpreting chess moves.

Thus, squares exceeding a predefined SSIM difference threshold undergo further division into five sections for detailed analysis. If the center section manifests a significant change beyond its respective threshold, the square is considered a candidate for a possible move. Moreover, if an apparent change is noted in the right section without a corresponding alteration in the left, the adjacent right square is also shortlisted as a potential candidate, observe Figure 16 for more detail. From these observations, we compile a list of probable candidate squares. Subsequently, we explore all legal move combinations derived from these candidate squares, prioritizing them based on the magnitude of difference detected. In cases where the distinction between the top two move candidates is small, we send up to five of the most likely moves to disambiguate in the GUI.



(a) Previous turn position. (b) Current frame position.

Figure 16: Projected board where the queen moved from h7 to f5. Observing the squares, one can notice how some pieces overlap, where a high difference in the right sub-square (and not the left) is likely indicative of the actual move originating from the adjacent right square.

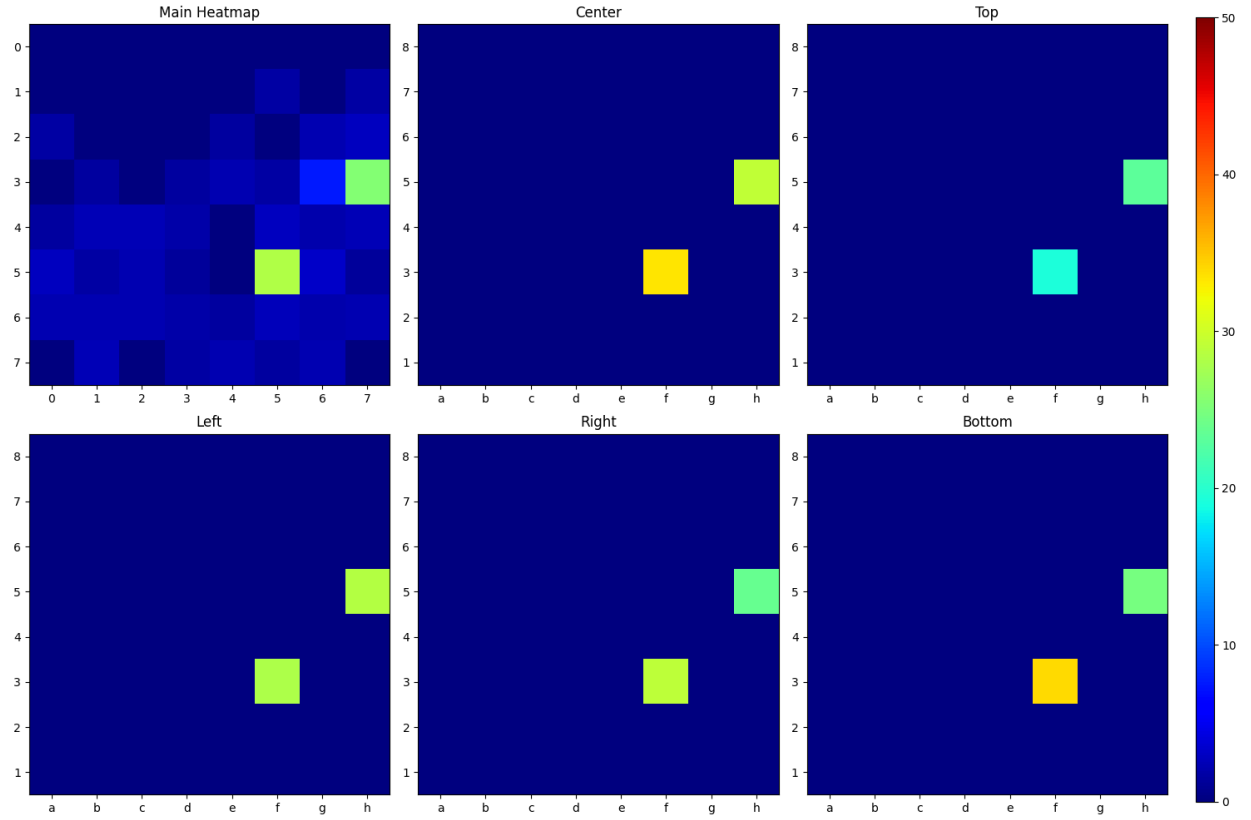


Figure 17: Heatmap representing SSIM differences across possible moves. The colorbar, capped at 50, reflects the SSIM difference, distinguishing between illegal moves (0) and potential legal moves. The cascade is also observable, where only the squares h7 and f5 are above the threshold (result from Figure 16), thus in the sub-squares all others are filtered. In this case, the sub-square detecting logic is not as important as there are no more possible moves.

While this methodology has proven effective, it is essential to acknowledge that a more refined approach could further enhance accuracy and robustness. Ideally, instead of relying on manually setting thresholds and divisions, we would develop a dataset capturing a wide array of move scenarios and their corresponding SSIM differences. Utilizing this data, a machine learning model could then be trained to discern significant changes and predict moves with higher precision. This adaptive approach would allow the system to continuously improve and adjust to various environmental conditions and chessboards.

Baseline

Our system has undergone significant enhancements to bolster its accuracy, reliability, and adaptability, ensuring more consistent performance across a variety of conditions. These improvements include a refined approach to data capture and analysis.

First, to ensure a more accurate and reliable understanding of the game's state, rather than comparing a single sample with its predecessor, our system captures five distinct samples at the end of each turn when the board physical's state is expected to have changed. Following the move processing by the controller (Section 3.2), these images are stored in an array for further analysis, or stored for retrieval in case the "Wrong Move" button was pressed.

Subsequently, two additional images are captured to establish a baseline for each square on the chessboard. These baselines are then compared with the series of five captured samples, allowing

us to calculate various statistical measures such as the median and maximum values. These calculations are instrumental in creating a set of dynamic thresholds, individually tailored to each square, that improve robustness in lighting and other environmental factors.

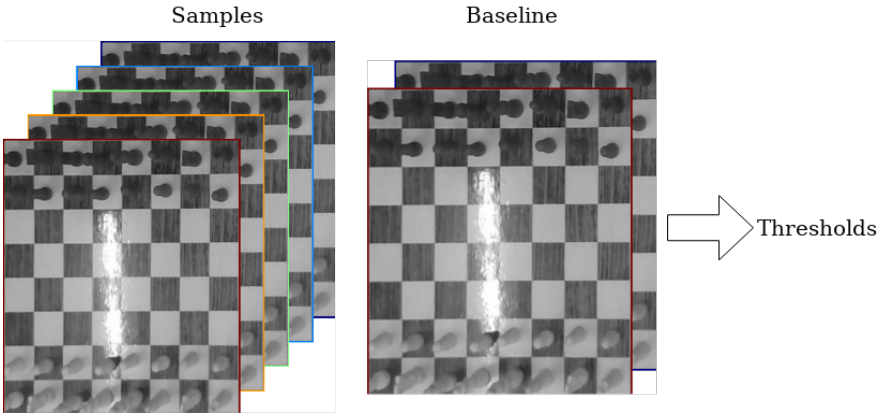


Figure 18: Utilizing the sample and two additional pictures, we compute the necessary thresholds for each board square.

Once we have established the baselines, we can compute the difference in SSIM between a frame and the samples as we did before, where we will take the average of the five SSIM differences and check if it is above the baseline threshold. Specifically, when the average difference exceeds the adaptive threshold–inferred from the maximum estimated baseline value, assuming a Gaussian distribution of noise–we consider it significant. This assumption holds because the differences between samples are typically just random noise, and calculating the standard deviation of these differences allows us to discern normal fluctuations from actual game-related movements. Further validation of this Gaussian assumption is left for future work, but it has proven effective thus far. By leveraging this approach, our system adeptly distinguishes between inconsequential variations and genuine moves or alterations on the chessboard. This method has significantly reduced false positives and has demonstrated robustness against variable lighting conditions.

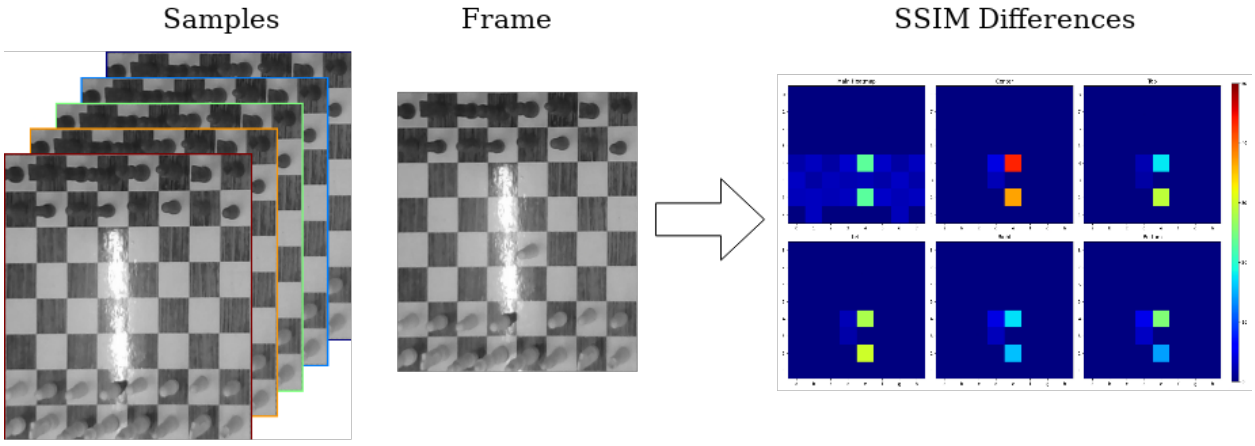


Figure 19: Comparing the standard sample with a single frame allows us to identify variations. The pre-established thresholds enable us to discern and categorize these differences effectively. We can see the pawn move e2e4, and the system correctly identifying it on the heatmaps.

Image Stability

The previous system operates effectively; however, a significant challenge persists: differentiating between a mere hand movement over the board and an actual piece movement. To address this, we've implemented a "debouncing" technique. This involves confirming that a frame closely aligns with the preceding five frames, specifically by ensuring the SSIM exceeds a predefined threshold of 97% similarity. By capturing images rapidly, at a rate of every tenth of a second, and comparing them for near-identical similarity, we can effectively discern transient obstructions, such as a hand moving over the frame, and disregard these anomalies.

However, stationary objects or hands lingering over the camera's view could still confound the system. To mitigate this, we incorporate Google's Mediapipe with Hand Landmarks Detection [18] as a fallback mechanism. This lightweight machine learning model adeptly identifies the presence of a hand in the image frame, prompting the system to postpone further image capture until the obstruction is cleared. Additionally, our approach extends beyond the chessboard to encompass a larger context, ensuring the area is devoid of any movement before capturing the images. This broader view helps in preemptively identifying potential disruptions, as illustrated in Figure 20.

(a) Extended context for potential interferences.

(b) Detection of a hand within the image frame.

Figure 20: Illustrations of hand detection process.

While this system enhances the accuracy and reliability of piece movement detection, it is not infallible. Users should operate the system with care, ensuring minimal interference or deliberate attempts to deceive the mechanism.

Pipeline

The image processing pipeline of ChessMate is a critical component that ensures the accurate detection and interpretation of moves on the physical chessboard. This pipeline operates in a continuous loop, adapting to each turn's unique conditions and movements. Below is an overview of the process accompanying the flowchart in Figure 21.

1. Initial Sampling: At the beginning of a new turn, the system collects 7 frames from the camera feed. Out of these, 5 frames are used for establishing a stable image sample, and 2 are reserved for computing the baseline. The sampling process is sensitive to stability and the absence of obstructions, such as a hand or other objects, in the frame.

2. Baseline Computation: Once a stable set of frames is acquired, the system computes the baseline. This step is critical in setting a reference for detecting changes or movements on the chessboard.

3. Movement Detection: The system then proceeds to compute the SSIM for each square of the chessboard. If the SSIM difference indicates a significant change beyond the previously computed baseline threshold, the system recognizes it as a potential move.

4. Sub-square Analysis: For squares with detected changes, the system further divides them into sub-squares to pinpoint the exact location and nature of the change. This granular approach allows for more accurate detection of piece movement or capture.

5. Move Ordering and Validation: Once potential moves are identified, they are sorted based on the degree of change indicated by the SSIM difference. The system then filters these moves based on their legality and the current game context. The most probable and legal moves are then prepared for output.

6. Output of Moves: The final step involves returning the detected moves to the main system, which then updates the game state accordingly or prompts for disambiguation if necessary.

This pipeline is encapsulated within a dedicated thread, ensuring that the image processing is efficient and doesn't interrupt the overall gameplay experience. It's designed to be both responsive and accurate, providing a reliable link between the physical game and the system's digital understanding.

The entire sampling and analysis process is completed in less than a few seconds, a duration primarily dictated by the need to await for the samples in order to achieve variable lighting conditions to maintain accuracy. Although it's highly likely that reducing the delay between captures could still yield satisfactory results, further fine-tuning was not viable within our project's timeframe.

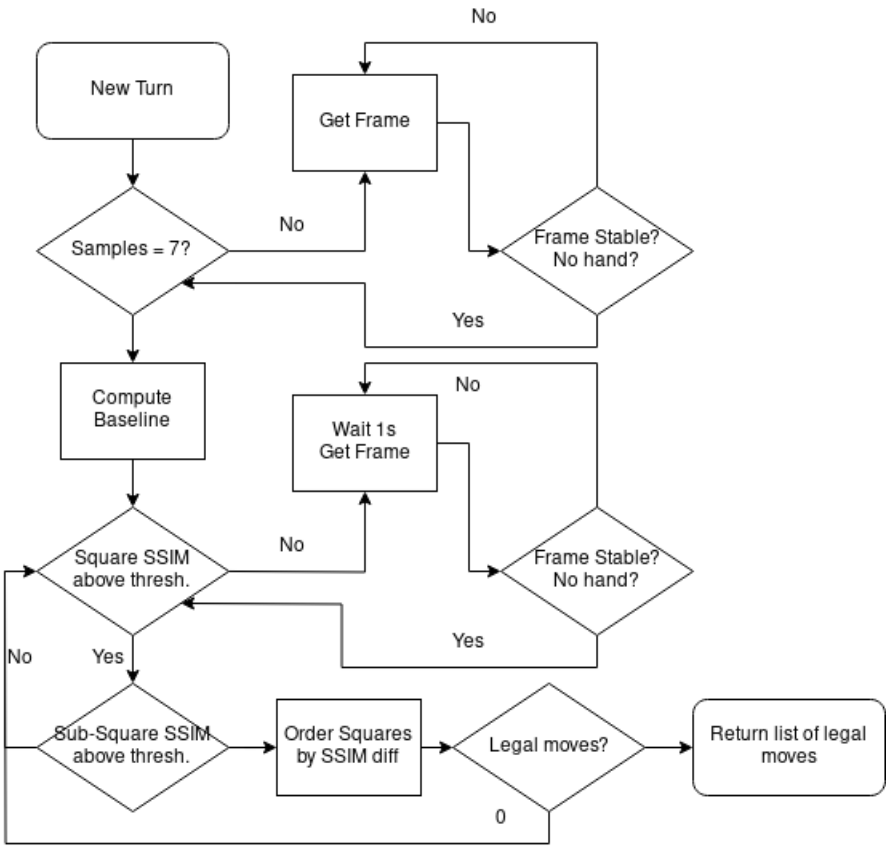


Figure 21: The image processing pipeline of ChessMate, illustrating the sequence from initial sampling to move output.

3.7 Audio Processing

The following section, focused on audio command processing, has been primarily tested and validated within the GUI environment of our chess system. Although not trialed with the Robotic Arm, the underlying logic for translating detected moves into robotic actions is robust and already integrated. Thus, it's expected to function seamlessly once deployed with the robotic component.

The `Transcriber` class is at the heart of our audio processing. It utilizes a continuous loop to listen and transcribe audio data. By setting specific timing thresholds, the system effectively determines the end of a spoken command, distinguishing between complete commands and partial or background noise. This ensures that only intentional moves and commands are captured and processed. This is achieved through the `SpeechRecognition` library [22] from Python.

Once audio data is captured, it's immediately queued for processing. The *OpenAI Whisper* model [23], known for its accuracy and efficiency, is then employed to transcribe the audio data into text, specifically utilizing the Large English model to ensure accuracy and prevent misinterpretation from other languages. These transcribed commands are pushed into a transfer queue, ready to be interpreted as chess moves.

To optimize audio command recognition and ensure the system accurately identifies and processes potential chess moves, we've implemented a keyword-based filtering mechanism. Specifically, the system is configured to listen for the trigger word "move" before it considers the subsequent spoken input as part of a chess move command. This approach significantly reduces false positives and irrelevant input, focusing the system's attention and resources on genuine game-related commands.

For example, if a player says "Move knight to E5," the system first recognizes "move" as the initiation of a command. It then proceeds to process "knight to E5," interpreting it into the corresponding Universal Chess Interface (UCI) format move, such as "f3e5" or "d3e5," depending on the knight's position.

Upon recognizing the command initiation keyword "move," our system employs a two-pronged approach to parse and interpret the transcribed audio input:

1. For straightforward commands such as the system utilizes a regular expression to extract the starting and ending coordinates of the move, effectively converting them into a valid UCI move.

2. For intricate commands or those requiring a deeper understanding of chess terminology, such as "Move E2 to E4," or "Queen captures," our system leverages help from the sophisticated *OpenAI GPT-4* language model [24]. GPT-4 excels in interpreting nuanced natural language and understanding context, thus it can usually parse the move given the current state of the game and the spoken move. This approach significantly enhances the system's ability to process a wide range of verbal chess instructions with greater intelligence and adaptability.

   To accomplish this, we use the completions API, which takes a list of messages as input and returns a model-generated message as output. Before querying GPT-4, the current chessboard configuration and the proposed move are parsed and passed along. The system then interprets the model's output, which may involve either one move or a list of potential moves which would then call the disambiguation part of the GUI, as outlined in Section 3.5. The following box contains the prompt's used in order to parse the move.

Chess Move Interpretation with GPT-4

System: You are a chess assistant helping to interpret spoken chess moves into UCI format.
User:

```
I have a transcribed move from a chess player, the board is the following:
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R

I want you to output only the move in UCI format. Take into account that the
transcription is automatic so consider possible misinterpretations, so
consider possible homonyms. Return the most probable move in UCI format.
If there are more than one possibility, return the most probable separated by

commas. Here is the move I want you to transcribe:
Move Knight to F3.
Output:
```

Output:

```
g1f3
```

While the current setup provides a functional balance between direct command interpretation and AI-assisted parsing, it's important to note that our development efforts were more intensively focused on enhancing and optimizing the image processing capabilities of the system. We believed that robust image recognition was crucial for accurately capturing the physical state of the chessboard and thereby improving the overall user experience, particularly in comparison to the audio component. However, further enhancing the sophistication of the speech recognition component remains an avenue for future improvement.

Pipeline

Figure 22 demonstrates the two main components of the audio processing pipeline:

- Sentence Transcriber Thread (Left Side): This part of the pipeline utilizes the Whisper model for continuous audio transcription. It is responsible for listening to the user's voice, capturing the audio data, and transcribing it into text.
- Queue Checker and Move Parser (Right Side): Once the audio is transcribed, the text is placed in a queue. This part of the pipeline continuously checks the queue for new transcriptions, interprets the chess-related commands, and parses them into UCI moves ready for execution.

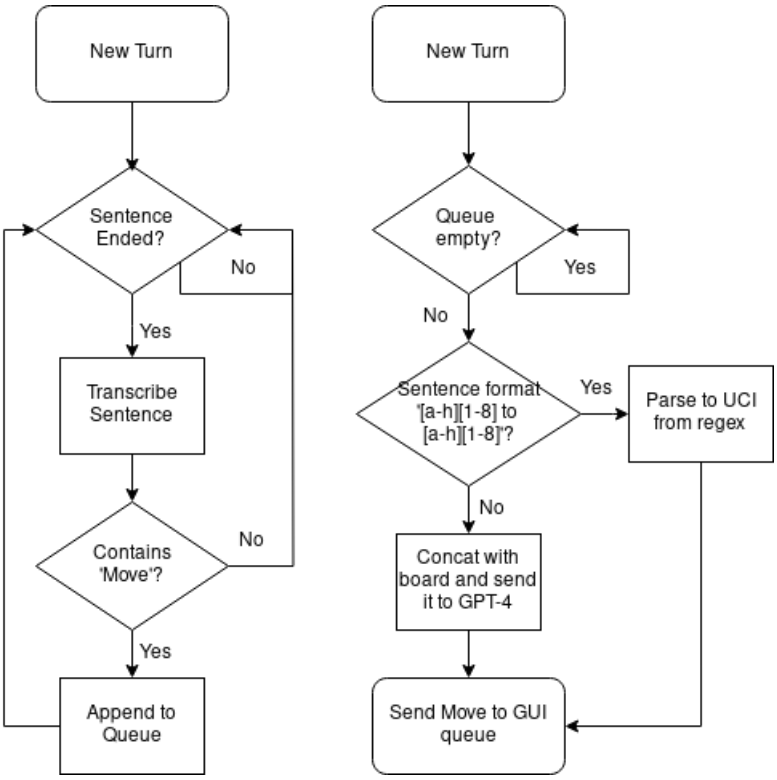


Figure 22: The audio processing pipeline of ChessMate, illustrating the sentence transcriber thread using OpenAI Whisper (left), and the move parser using either regex or GPT-4 (right).

4 Results

To comprehensively evaluate our system, we conducted an in-depth study focusing on various aspects of user experience. This involved inviting participants to engage in chess matches using both our robotic arm system and a conventional GUI chessboard. To ensure no differences in fatigue, the participants were randomized to either start playing on the GUI or on the robotic arm. Following the end of both games, participants were asked to complete a questionnaire aimed at evaluating several key factors of their experience.

The study was conducted with a cohort of 10 individuals aged between 20 and 30 years old. Each participant played against the same Leela Zero chess engine configured with 0.01 seconds of processing time per move, ensuring uniform difficulty across games with both the robotic arm and GUI.

The dependent variables in our study included player satisfaction, system usability, response time, and any errors or issues encountered during gameplay. We delved into specific aspects of these variables to gain a nuanced understanding of the user experience.

To avoid any extraneous factor, we ensured equal conditions to all players. Specifically, all game sessions were held in the FIB laboratory (C5-S203) to maintain consistent conditions, utilizing the same robotic equipment available in the room. Due to scheduling constraints, mainly the holiday season, participants were only able to engage with the system on two designated days, Thursday 21/12 and Friday 22/12, limiting our ability to obtain more data. To ensure uniformity across the sessions, identical instructions were provided to each participant, the details of which are documented in subsection B.1.

The survey administered post-match included a variety of questions related to the previously mentioned dependent variables. Previous to recording data, we obtained a consent form regarding

participation to our analysis. The questionnaire was designed to yield both qualitative insights, through open-ended questions, and quantitative data, utilizing a 5-point Likert scale to gauge agreement levels. Additionally, metrics such as response time and system error rates were recorded independently of the questionnaire to provide objective performance data.

After gathering responses from the surveys, we meticulously analyzed the data to extract insightful information that could guide improvements to our system. This involved interpreting participant feedback and integrating objective measurements that reflect the system's performance independently of user perception.

Due to the limited number of participants and the homogeneity of the demographic that engaged with our system, we acknowledge a lack of a distinct independent variable in our study. The uniformity of the participant group makes it challenging to identify meaningful differences or conduct comparative analyses. Attempting to impose categorical distinctions post-hoc would not provide a fair or statistically sound basis for analysis. Consequently, our capability to perform varied statistical tests is inherently restricted.

In terms of quantitative analysis, our focus will be on documenting and assessing system errors, a metric that is independent of user variations. Specifically, we will report the total number of moves correctly processed by the system and highlight any instances of incorrect processing. This approach provides a clear, objective measure of system performance. Nevertheless, we also ask for the user's experience and if they experienced any possible mistake as a qualitative measure.

For the Likert scale responses gathered, we resorted to non-parametric tests appropriate for ordinal data, avoiding parametric tests due to the diminished statistical power resultant from the small sample size. Without a viable independent variable for grouping or comparison, our analysis is primarily descriptive. We will perform a singular test to assess the overall preference for playing against the robot versus the GUI, providing insight into user enjoyment and system preference without stratifying the data.

Lastly, we delved into the qualitative aspect of our study by thoroughly examining the open-ended responses provided by participants. We selected the most insightful and reflective comments to present a narrative that captures the general attitudes and specific feedback regarding the system. This qualitative analysis is crucial, offering depth and context that numbers alone cannot convey, thereby painting a more comprehensive picture of user interaction and satisfaction with ChessMate. These narratives not only enrich our understanding but also serve as a guide for future improvements and iterations of the system.

4.1 Player satisfaction

In evaluating player satisfaction, our study concentrated on various facets of the participants' experiences and their overall enjoyment of the game. We sought to understand not only if users found the game enjoyable, but also if they would recommend the experience to others. The results can be observed in Figure 23.

The feedback received indicates that most users found their interaction with the robotic system to be a satisfying experience. They reported enjoying the novel aspect of playing chess with a robot. Furthermore, the majority did not find the match to be tedious, although there were instances where some participants reported their engagement decreased at certain points during the game.

Opinions varied significantly concerning the difficulty level of the game. Some users viewed the challenging nature of the system positively, enhancing their experience by providing a tough opponent. Conversely, others felt that the difficulty was too steep, bordering on excessively challenging, which somewhat detracted from their overall enjoyment.

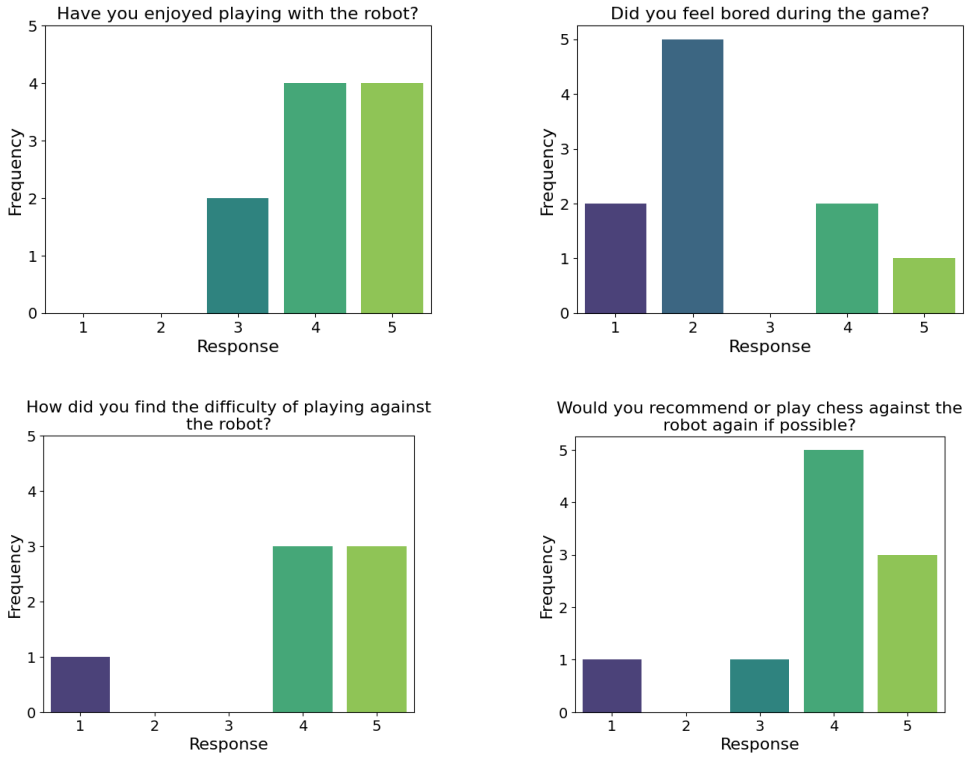


Figure 23: Participant responses on their experience playing with the robot, including enjoyment, boredom, perceived difficulty, and willingness to recommend or replay. Recall that in the survey, the scale was rated as 1 - Strongly Disagree to 5 - Strongly Agree, except for the difficulty, where we allowed a larger range, up to 7.

Despite these variations in experience, when it came to recommending the system to others or considering a repeat play, a predominant portion of participants responded affirmatively. They endorsed the experience as worthwhile and expressed interest in engaging with the robot again, highlighting the positive aspects of their interaction.

The survey also included an open-ended question asking participants to describe their experience with the robot in their own words. Responses were largely positive, with many describing the experience as interesting and novel. Some participants noted that playing against the robot served as a unique way to improve their chess skills. However, there were remarks about the downsides, particularly noting the operational restrictions and waiting times associated with the robotic system, which some felt might limit the appeal of repeated play.

4.2 Usability

This section delves into the usability aspects of our system, focusing on participant feedback related to their interaction and communication with the robotic setup.

A key points in our study was whether participants found the communication with the robot to be clear. The responses indicate a general agreement on this aspect, although the ratings did not reach the highest possible score. When probed about the graphical user interface specifically, users reported that the feedback provided regarding the game's status and the movements made was clear and understandable, garnering more definitive positive responses compared to general communication clarity.

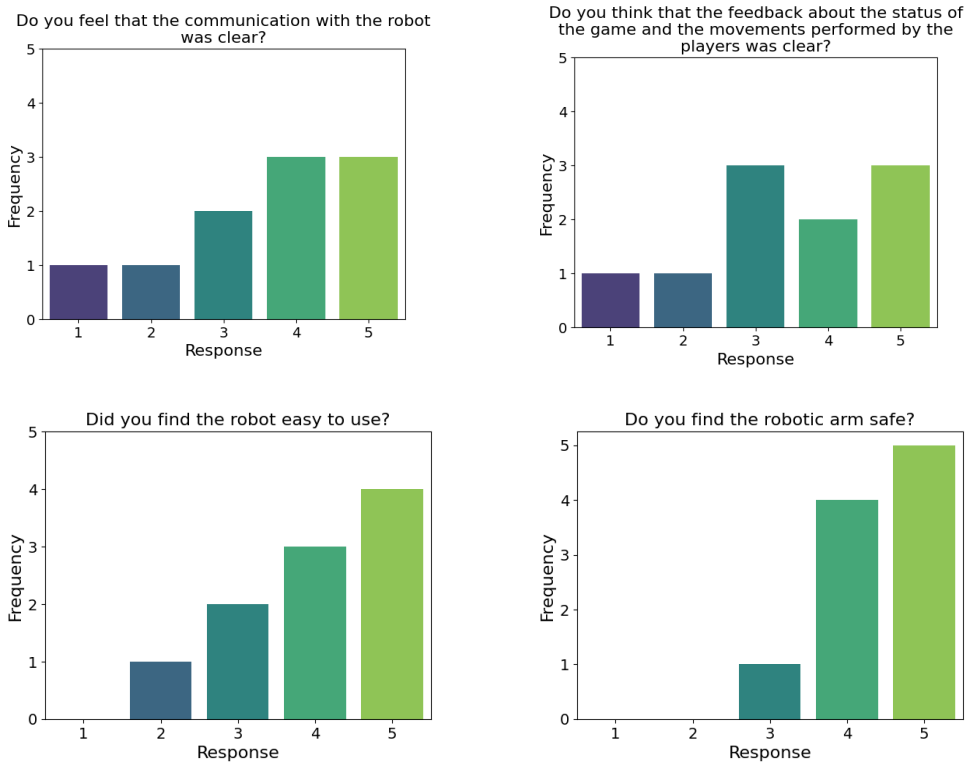


Figure 24: Participant responses for the user interaction with the robotic system, highlighting perceptions of communication clarity, feedback adequacy, ease of use, and safety. These factors are critical in assessing the overall user experience and acceptance of robotic assistance in gameplay.

We also explored the ease of use of the system. The majority of participants found ChessMate to be user-friendly, describing it as straightforward and simple to operate. It's worth noting that these assessments were based solely on interactions with the image recognition system; the usability of voice command features remains untested with users in this study, indicating an area for future research.

The sense of safety while interacting with the robotic arm was another aspect we investigated. Participants generally reported feeling safe and comfortable with the robotic arm, with initial hesitancy subsiding as they became more accustomed to the system's movements and functionality throughout the gameplay.

In terms of areas for improvement, a recurrent point of feedback concerned the clarity in turn transition. Participants expressed that it was sometimes unclear when it was their turn to move, especially as they needed to wait for a certain duration after the robot completed its move. Based on this feedback, some suggested enhancements include an auditory signal or a more conspicuous on-screen notification to more clearly indicate when it's the player's turn to proceed.

4.3 Response time

This section provides a comprehensive evaluation of the system's response time, encompassing both objective measurements and user perceptions. The goal is to assess the efficiency of the system in recognizing and executing movements, as well as understanding how users perceive these timings during their interaction with the system.

Objective Measurements of Response Time

We have conducted a quantitative analysis of the system's performance, focusing particularly on the latency of key operations. This involved measuring the time it takes for the system to recognize a movement made by a person and the time required for the robot to execute a movement, considering variations based on whether the move is to an occupied or a free square.

The table below summarizes the average time and standard deviation for each type of move, including normal moves, castling, captures, and image recognition. These metrics provide a detailed view of the system's performance in various operational scenarios:

Operation	**Average Time (s)**	**Standard Deviation**
Normal Move	22.6	0.95
Castling	43.0	0.82
Capture	38.4	1.28
Image Recognition	6.8	0.93

Table 1: Average operation times and standard deviations for different chess moves and image recognition. The first three correspond to the movement of the robotic arm, from the initial position to finishing in the initial position.

It's important to note that the time associated with image recognition is influenced by await and debounce mechanisms rather than computation alone. Through improvement in the threading section of the code, compared to the lab version, we managed to reduce these response times and enhance overall system performance. Nevertheless, the improved code was not tested in the controlled environment.

User Perception of Response Time

In addition to objective measurements, we sought to understand users' subjective perceptions of the system's response time. A specific survey question aimed to gauge whether users found the waiting time acceptable or overly long. Responses varied, with many expressing initial impatience that tended to diminish as the game progressed. Interestingly, some users noted that they appreciated the extra time to contemplate their next move, particularly in the later stages of the match.

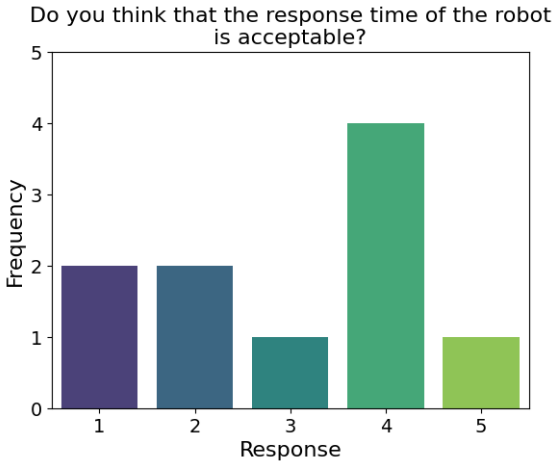


Figure 25: Participant responses for their perception on the response time against the robot.

This mixed feedback indicates that while the objective efficiency of the system is crucial, user tolerance for response times can vary widely and may even change contextually during gameplay.

4.4 System Performance

Similar to the previous case, for this section we have also measured several metrics related to system failures or successes. Specifically, we have analyzed the image and voice recognition system to identify the movement made by the participant, and the motions of the robot. As well as the user's experience with what they perceived to be errors.

Objective Measures of System Performance

In assessing the performance of our system, we meticulously measured several metrics related to its operational success and any failures. Specifically, we analyzed the image and voice recognition capabilities for detecting user movements and the accuracy and reliability of the robotic arm's motions.

Image Recognition: In a controlled home environment, the image recognition system exhibited excellent performance. Out of 161 moves tested, 157 were accurately identified, with the remaining four effectively resolved through our disambiguation system. Conversely, in a laboratory setting, out of 313 moves, 7 required disambiguation intervention. Additionally, users prematurely moved the chess pieces in about 23 instances, necessitating manual input via the GUI. These instances highlight the sensitivity of the system to user behavior and environmental conditions.

Voice Recognition: Our audio recognition system parsed 76 out of 95 spoken legal moves correctly when of the type the simple regular expression could match. However, the GPT-4 parser of complex showed inferior, performance, where only 23 out of 44 were correctly interpreted. Moreover, as the game progressed, the error rate of obtaining the correct move increased drastically. These limitations led us to deprioritize audio-based move input in the current system iteration.

Robotic Arm Performance: The robotic arm executed all 567 moves with high reliability. However, minor issues were noted, such as occasional pieces sticking to the gripper, likely a sign of normal wear and tear.

User Perception of System Performance

Alongside objective metrics, we sought to understand participants' perceptions, particularly regarding any errors or unusual behaviors they might have noticed during gameplay.

Participants generally did not report any significant errors with the system's operation. No instances of the robotic arm moving incorrectly or mishandling pieces were noted by users. However, as already stated, it was observed that participants sometimes moved the chess pieces too soon, leading to situations where the system failed to detect the move. In these instances, manual corrections had to be made using the GUI.

This discrepancy between the system's actual performance and user perception underscores the importance of clear communication and instruction, ensuring that users understand and adhere to the system's operational flow. It also highlights the need for the system to accommodate or respond to premature user actions effectively.

4.5 Evaluation of User Experience: Robotic System vs. Virtual Board

In assessing whether users experience higher levels of enjoyment and engagement with the physical, robotic chess system compared to a virtual board, we employed the Wilcoxon Signed Rank Test as our primary analytical tool. This non-parametric test is particularly suited for paired data, focusing on the magnitude and direction of differences between two related samples.

Our research aimed to explore the following hypotheses under the assumption of a one-sided test:

H_0 : The median difference between satisfaction scores for robot vs. screen is zero (no difference).

H_1 : The median difference is positive, indicating higher satisfaction with the robot, with α set at 0.05.

Participants were asked to rate their satisfaction on a scale from 1 to 5 for both experiences: playing against the robotic system and against a virtual board. These ratings were then used to perform the Wilcoxon Signed Rank Test, examining the differences in satisfaction levels.

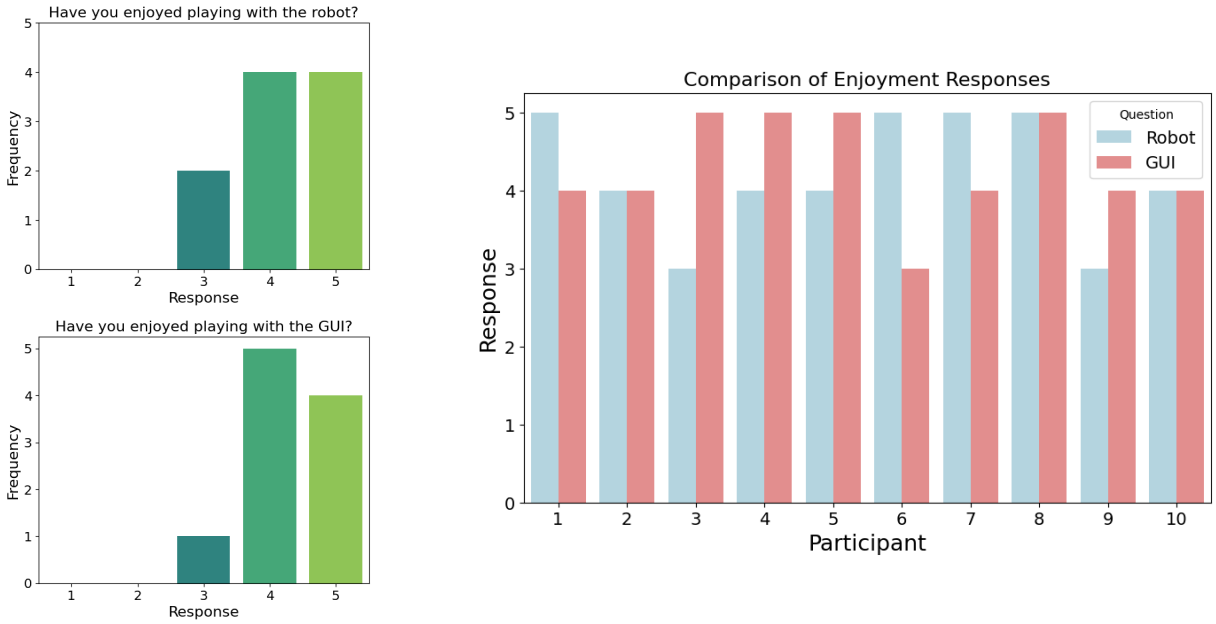


Figure 26: The figure illustrates participant satisfaction ratings for playing with the robotic system (top left) and the virtual board (bottom left), along with a comparative analysis (right).

Using the data collected in Figure 26 and employing the *Wilcoxon Signed Rank Test* through the Scipy [25] Python library, we obtained the test statistic ($W = 12.5$) and corresponding p-value ($p - value = 0.6039$). The results indicated that the p-value exceeded the threshold of 0.05. Consequently, there isn't enough evidence to reject the null hypothesis (H_0), suggesting no significant difference in user satisfaction between playing against the robotic system and the screen.

4.6 Amount of Movements

To explore the potential relationship between user satisfaction during gameplay with the robot and the total number of moves made before the game's conclusion, we employed Pearson's correlation. This analysis aimed to discern if longer or shorter games correlate with higher satisfaction levels.

We utilized the satisfaction ratings with the robot and the total number of moves from the game, as presented in Table 2.

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**
Satisfaction Robot	5	4	3	4	4	5	5	5	3	4
Movements	20	15	17	11	16	19	9	13	7	12

Table 2: User satisfaction and the amount of moves they did when playing against the Robotic Arm before ending the game.

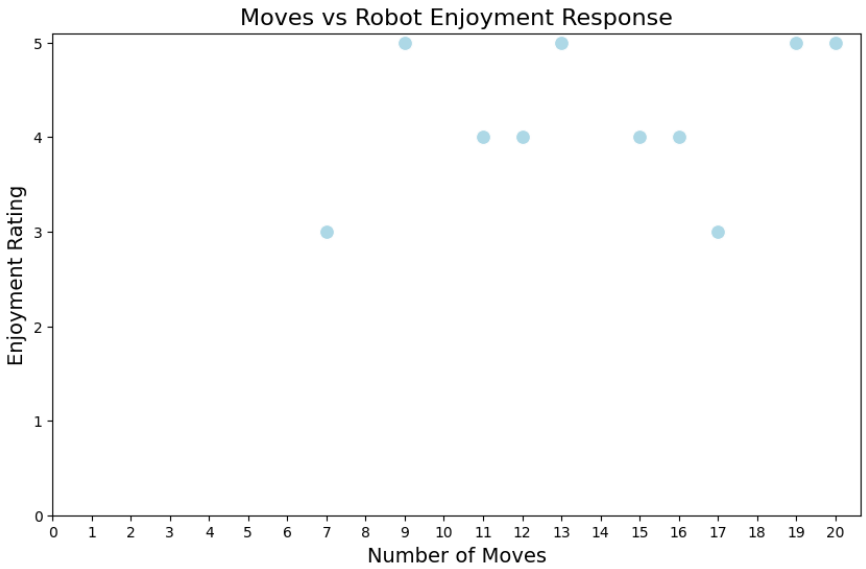


Figure 27: Scatter plot showing the relationship between the number of moves done when playing against the robot.

The results obtained indicate a Pearson correlation coefficient of 0.3046 and a p-value of 0.3921. While the correlation coefficient suggests a low to moderate positive correlation between satisfaction with the robot and the number of moves, the high p-value indicates that this result is not statistically significant. This implies that any observed correlation could likely be due to chance, and there's no definitive evidence to suggest a strong relationship between the length of the game and user satisfaction in this context.

When interpreting the correlation between game length and user satisfaction from our study, several key considerations must be kept in mind. The ordinal nature of Likert scale data can complicate analysis, as treating it as continuous assumes equal intervals between scale points, which may not accurately reflect the data's properties. The small sample size of only 10 participants limits the generalizability and reliability of the results, indicating a need for a larger, more representative sample in future studies. Additionally, various contextual variables, like playing style or game difficulty, can influence both the number of moves and satisfaction, necessitating careful control or consideration of these factors in analysis. Lastly, it's crucial to differentiate between correlation and causation; a significant correlation does not imply one variable causes the other, and understanding the true relationship requires more in-depth, targeted research.

In addition to the primary analysis, we further explored the potential correlations between the number of moves completed in each game and various questionnaire metrics based on the Likert scale, seeking to uncover any meaningful relationships. Despite our efforts to discern patterns or significant connections, the analyses did not reveal any statistically significant or meaningful

correlations between these variables. This suggests that within the scope and scale of our study, the game length does not appear to strongly or consistently impact the various aspects of user experience as measured by the Likert-style questions. This outcome highlights the complexity of user satisfaction and engagement factors in gameplay and indicates the need for further, more nuanced research to understand these dynamics fully.

5 Conclusions

ChessMate emerges as an innovative Human-Robot Interaction (HRI) system, designed to enrich the traditional chess experience with the integration of advanced robotics and intelligent system design. This report has provided a detailed examination of ChessMate, highlighting its ability to offer an engaging, interactive platform for chess players through the use of a state-of-the-art robotic arm and high-definition camera, complemented by a user-friendly GUI. The system's versatility is showcased in its support for multiple game modes, catering to a diverse range of chess enthusiasts and technology aficionados.

Throughout the study, we aimed to understand user experiences, focusing on satisfaction, usability, and system performance. The feedback indicates a generally positive reception, with particular appreciation for the tactile and interactive nature of the robotic system, although some challenges were noted in terms of game difficulty and clarity in turn transitions. Objective measurements provided insight into the system's technical performance, indicating high levels of accuracy and reliability, with areas identified for further enhancement.

Our research also delved into the potential relationship between game length and user satisfaction. Despite some indications of a low to moderate positive correlation, the results were not statistically significant, suggesting the need for further investigation with a larger and more diverse sample. Additionally, the exploration of various questionnaire metrics revealed no meaningful correlations, highlighting the complexity of user satisfaction and the need for nuanced research.

In conclusion, ChessMate stands as a testament to the potential of integrating robotics and AI in traditional games, offering a novel and engaging experience that blurs the lines between physical and virtual gameplay.

6 Future Work

Building upon the foundation laid by ChessMate, the following areas present opportunities for further enhancement and research to advance the capabilities and user experience of this innovative system.

- **Automated Chessboard Detection**: Future versions could explore more robust and automated methods for detecting the chessboard, potentially incorporating machine learning techniques that adapt to different styles and conditions of chessboards.
- **Improved Move Detection**: Enhancing the precision and speed of move detection algorithms, possibly through machine learning or more advanced image processing techniques, could provide faster and more accurate game state updates.
- **Advanced Speech Recognition**: Building a more sophisticated audio command processing system, possibly integrating a more robust logic and following the regular expression method, could provide a more intuitive and error-resistant user experience.

- **Seamless Integration with Physical Moves**: Ensuring the audio processing system works flawlessly with the robotic arm movements, including handling interruptions, confirmations, or repeating misunderstood commands.
- **User Interface Improvements**: Developing a more intuitive and responsive user interface that provides clearer feedback, especially regarding turn status.
- **Expanded User Testing**: Conducting studies with a larger and more diverse user base would provide more data for understanding user satisfaction and system performance across different demographics.
- **Longitudinal Study on Engagement**: Investigating how user engagement changes over time with repeated use could provide insights into the system's lasting appeal and areas for ongoing improvement.
- **Comparative Studies**: Further comparative research between the robotic system and virtual boards or even traditional physical boards could yield deeper insights into user preferences and the unique value proposition of each system.
- **Hardware Improvements**: Upgrading cameras, microphones, or even the robotic arm itself could lead to better performance, accuracy, and user experience.
- **Software Optimization**: Continual refinement of the software, including algorithms for image and audio processing, error handling, and user interaction, would enhance both efficiency and usability.

By addressing these areas, ChessMate can evolve to meet the changing needs and expectations of users, pushing the boundaries of Human-Robot Interaction in gaming and beyond. These advancements would not only improve the system's performance and user satisfaction but also contribute to the broader field of robotics and intelligent systems design.

References

[1] The Stockfish developers (see AUTHORS file). *Stockfish*. Version 1.2.0. 2023. URL: https://stockfishchess.org/.

[2] Pascutto, Gian-Carlo and Linscott, Gary. *Leela Chess Zero*. Version 0.21.0. Mar. 8, 2019. URL: http://lczero.org/.

[3] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[4] Robotis. *Turtlebot3*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[5] Robotis. *OpenManipulator-X*. URL: https://emanual.robotis.com/docs/en/platform/openmanipulator%5C_x/overview/.

[6] Universal Robots. *On robot RG2 Gripper*. URL: https://www.universal-robots.com/fi/plus/products/onrobot/rg2-gripper/.

[7] Andrea Raviola et al. "A Comprehensive Multibody Model of a Collaborative Robot to Support Model-Based Health Management". In: *MDPI-robotics* (2023). URL: https://www.mdpi.com/2218-6581/12/3/71.

[8] Dobot. *On robot RG2 & RG6*. URL: https://www.dobot-robots.com/ecosystem/accessories/592.html?gad_source=1&gclid=Cj0KCQiAkKqsBhC3ARIsAEEjuJiFua7Jge6VOHR0CZodliZbbmYm8UYT7XCcP-p3IOAgvZv2A3YogaAhDDEALw%5C_wcB.

[9] Universal Robots. *The URScript Programming Language G3*. URL: https://s3-eu-west-1.amazonaws.com/ur-support-site/105200/scriptManual%5C_SW3.15.pdf.

[10] Universal Robots Support. *Remote Control Via TCP/IP*. 2022. URL: https://www.universal-robots.com/articles/ur/interface-communication/remote-control-via-tcpip/.

[11] Universal Robots. *Universal Robots e-Series User Manual*. URL: https://s3-eu-west-1.amazonaws.com/ur-support-site/41166/UR3e%5C_User%5C_Manual%5C_en%5C_Global.pdf.

[12] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[13] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[14] David Mallasén Quintana, Alberto Antonio Del Barrio García, and Manuel Prieto Matías. "LiveChess2FEN: A Framework for Classifying Chess Pieces Based on CNNs". In: *arXiv:2012.06858 [cs]* (Dec. 2020). arXiv: 2012.06858. URL: http://arxiv.org/abs/2012.06858.

[15] Radhakrishna Achanta et al. "SLIC superpixels compared to state-of-the-art superpixel methods". In: *IEEE transactions on pattern analysis and machine intelligence* 34.11 (2012), pp. 2274–2282.

[16] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. *YOLO by Ultralytics*. Version 8.0.0. Jan. 2023. URL: https://github.com/ultralytics/ultralytics.

[17] David G Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[18] Google. *Mediapipe: On-device machine learning for everyone*. Accessed: 2023-12-10. 2023. URL: %7Bhttps://developers.google.com/mediapipe/%7D.

[19] Zhou Wang et al. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[20] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453.

[21] Jim Nilsson and Tomas Akenine-Möller. "Understanding ssim". In: *arXiv preprint arXiv:2006.13846* (2020).

[22] Anthony Zhang. *Speech Recognition (Version 3.8)*. [Software]. 2017. URL: %7Bhttps://github.com/Uberi/speech_recognition#readme%7D.

[23] Alec Radford et al. *Robust Speech Recognition via Large-Scale Weak Supervision*. 2022. DOI: 10.48550/ARXIV.2212.04356. URL: https://arxiv.org/abs/2212.04356.

[24] OpenAI. *GPT-4: Language Model*. Accessed: 2023-12-23. 2023. URL: %7Bhttps://www.openai.com/%7D.

[25] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

Appendices

A TurtleBot3 Waffle Pi - Installation guide

TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping, that can be controlled remotely from a laptop. The TurtleBot3's core technology is SLAM (simultaneous localization and mapping), Navigation and Manipulation, making it suitable for home service robots. Also TurtleBot3 can be used as a mobile manipulator capable of operate with objects by attaching a manipulator like OpenMANIPULATOR, which is compatible with TurtleBot3 Waffle Pi.

This guide is based on the steps specified on the website: https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/, referring to `Foxy`.

Requirements

To make the installation process way more easier please check the following information:

- **Material**: The following material would be required during the different steps carried out in the installation process, so make sure to have access to all of it.
  - MicroSD (more than 8GB). WARNING: Use an empty MicroSD card because all the information will be deleted when the image is burned.
  - MicroSD slot or card reader to burn the recovery image. Unless having Ubuntu partition in your own PC, it is recommended to use an external card reader, and not the slot, to avoid problems in the following steps.
  - HDMI to micro-HDMI (used in the Raspberry Pi) cable.
  - Keyboard and a mouse with USB-A connection.
  - Access to Ubuntu 20.04 system (not a virtual machine one unless you have access to all the native system). This is not a requirement for all the steps but for some of them.
- **Permissions**: Given that the FIB system prevents unidentified addresses from accessing the internal network, it is necessary to request permission so that the robot can connect to the internet. To do so the MAC address should be provided running the following command:

  ```
  $ ipp addr show
  ```

  and using the information displayed in `wlan >> link/ether`.

PC Setup

WARNING: The contents in this chapter corresponds to the **Remote PC** (your desktop or laptop PC) which will control TurtleBot3. Do not apply this instruction to your TurtleBot3.

1. Download and Install Ubuntu on PC

1. Download the proper **Ubuntu 20.04 LTS Desktop** image for your PC from this link.
2. Follow the instructions specified in this link to install Ubuntu on PC.

2. Install ROS 2 on Remote PC

Open the terminal with `Ctrl+Alt+T` and enter below commands one at a time. If you are using a Windows computer, go to the search bar and look for the `wsl` or the `Ubuntu 20.04.6 LTS` options, that will open a command line.

```
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/master/install_ros2_foxy.sh
$ sudo chmod 755 ./install_ros2_foxy.sh
$ bash ./install_ros2_foxy.sh
```

If the above installation fails, they recommend to visit the official ROS2 Foxy installation guide.

3. Install Dependent ROS 2 Packages

1. Open the terminal with `Ctrl+Alt+T` from **Remote PC**. As before, if you are using a Windows computer, go to the search bar and look for the `wsl` or the `Ubuntu 20.04.6 LTS` options, that will open a command line.
2. Install Gazebo11

```
$ sudo apt-get install ros-foxy-gazebo-*
```

3. Install Cartographer

```
$ sudo apt install ros-foxy-cartographer
$ sudo apt install ros-foxy-cartographer-ros
```

4. Install Navigation2

```
$ sudo apt install ros-foxy-navigation2
$ sudo apt install ros-foxy-nav2-bringup
```

4. Install TurtleBot3 Packages

Install TurtleBot3 via Debian Packages.

```
$ source ~/.bashrc
$ sudo apt install ros-foxy-dynamixel-sdk
$ sudo apt install ros-foxy-turtlebot3-msgs
$ sudo apt install ros-foxy-turtlebot3
```

5. Environment configuration

Set the ROS environment for PC.

```
$ echo 'export ROS_DOMAIN_ID=30 #TURTLEBOT3' >> ~/.bashrc
$ source ~/.bashrc
```

SBC Setup

WARNING

- This process may take long time. Please do not use battery while following this section.
- An HDMI monitor and input devices such as a keyboard and a mouse will be required to complete this setup.
- If you are using a Windows OS system either with a Linux Subsystem or a virtual machine environment, the whole SBC Setup process will be way more easier if you do it in the Lab PCs, given that unless having full access to the native Window PC configuration through the subsystem, the process might be tricky otherwise.

1. Prepare microSD Card and Reader

If the PC used do not have a microSD slot, use a microSD card reader to burn the recovery image.

2. Download TurtleBot3 SBC Image

Download `Raspberry Pi 4B (2GB or 4GB)` ROS2 Foxy image file for your hardware and ROS version. Foxy version images are created based on Ubuntu 20.04.

To know the properly file to download, you should check the Raspberry Pi information written in the micro controller. The following image shows the easiest way to read this information.

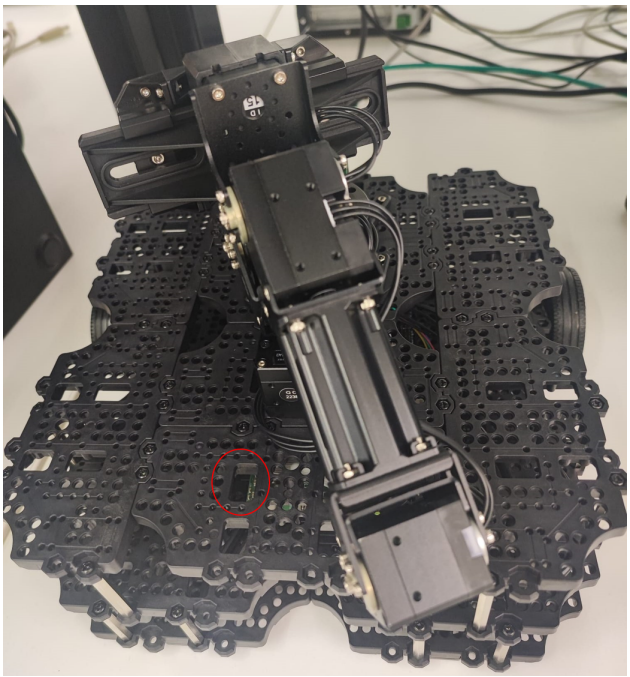


Figure 28: Where to find the version to download written in the Raspberry Pi

3. Unzip the downloaded image file

Extract the `.img` file and save it in the local disk.

4. Burn the image file

For this step, we have used the `Raspberry Pi Imager`, although Linux `Disks` utility can be used (as explained in the manual).

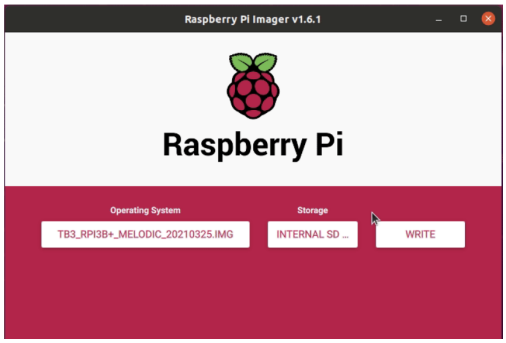


Figure 29: Raspberry Pi Imager

1. Download Raspberry Pi Imager and open it.
2. Click `CHOOSE OS`, go to the `Use custom` option (at the end) and select the extracted `.img` file from local disk (from the previous step).
3. Click `CHOOSE STORAGE` and select the microSD.
4. Click `WRITE` to start burning the image.

5. Resize the Partition

In order to reduce the size of recovery image file and to decrease the time to burn the image onto microSD, the recovery partition is minimized. Please resize the partition to use the unallocated space.

WARNING: Be aware of selecting an incorrect disk or a partition. Partitioning a system disk of your PC may cause a serious system malfunction.

1. Install and open the GParted GUI tool. To do this, open a terminal and write the following:

```
$ sudo apt-get install gparted
$ gparted
```

2. Select microSD card from the menu (mounted location may vary by system). To identify the microSD, we recommend to focus on the capacity of the device and, when selected, check that it has 2 partitions named *system-boot* and *writable*.
3. Right click on the yellow partition.
4. Select `Resize/Move` option.
5. Drag the right edge of the partition to all the way to the right end.
6. Click `Resize/Move` button.
7. Click the green check button at the top, that corresponds to `Apply All Operations`.

6. Configure the WiFi Network Setting

1. Open a terminal window with `Alt+Ctrl+T` and go to the *netplan* directory in the microSD card. Then, start editing the `50-cloud-init.yaml` file with a superuser permission `sudo`.

```
$ cd /media/$USER/writable/etc/netplan
$ sudo nano 50-cloud-init.yaml
```

2. When the editor is opened, replace the `WIFI_SSID` and `WIFI_PASSWORD` with your wifi SSID and password. To obtain this information, please ask the supervisor of the laboratory.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: yes
      dhcp6: yes
      optional: true
  wifis:
    wlan0:
      dhcp4: yes
      dhcp6: yes
      access-points:
        WIFI_SSID:
          password: WIFI_PASSWORD
```

Figure 30: WiFi Network Setting

3. Save the file with `Ctrl+S` and exit with `Ctrl+X`.

Once this is done, you need to boot up the Raspberry Pi following the next steps in the specified order:

1. Connect the HDMI cable of the monitor to the micro-HDMI port of Raspberry Pi and change the input device of the monitor. HDMI cable has to be connected before powering the Raspberry Pi, or else the HDMI port of the Raspberry Pi will be disabled.
2. Connect the keyboard and the mouse to the USB port of Raspberry Pi.
3. Insert the microSD card to Raspberry Pi.
4. Connect the power (either with USB or OpenCR) to turn on the Raspberry Pi.
5. Wait while the screen displays commands related to powering up the Raspberry Pi. When this operation stops and no more lines appear, click enter.
6. Login with ID `ubuntu` and PASSWORD `turtlebot`.

7. ROS2 Network Configuration

In ROS2 DDS communication, `ROS_DOMAIN_ID` must be matched between **Remote PC** and **TurtleBot3** for communication under the same network environment. The default ROS Domain ID for TurtleBot3 is set to `30` in the ***.bashrc*** file.

Please modify the ID to avoid any conflict when there are identical ID in the same network. By default, the number is correct, but we recommend to check it:

```
$ cat .bashrc
```

WARNING: Do not use an identical `ROS_DOMAIN_ID` with others in the same network. It will cause a conflict of communication between users under the same network environment.

8. New LDS-02 Configuration

WARNING: For the following steps consider that the internal Raspberry Pi Ubuntu might use the American keyboard layout configuration. In order to edit this configuration you have to run

```
$ sudo vim /etc/default/keyboard
```

and change the `XKBLAYOUT` line to the name of the keyboard layout we want to use: e.g. `XKBLAYOUT="es"`. After saving and exiting the file, we need to reboot the system to apply the changes.

Follow the instructions below on the SBC (Raspberry Pi) of TurtleBot3.

1. Install the LDS-02 driver and update TurtleBot3 package

```
$ sudo apt update
$ sudo apt install libudev-dev
$ cd ~/turtlebot3_ws/src
$ git clone -b ros2-devel https://github.com/ROBOTIS-GIT/ld08_driver.git
$ cd ~/turtlebot3_ws/src/turtlebot3 && git pull
$ rm -r turtlebot3_cartographer turtlebot3_navigation2
$ cd ~/turtlebot3_ws && colcon build --symlink-install
```

2. Export the LDS_MODEL to the bashrc file. In our case, we have used the `LDS-02`.

```
$ echo 'export LDS_MODEL=LDS-02' >> ~/.bashrc
$ source ~/.bashrc
```

OpenCR Setup

Connect to Raspberry Pi (if not connected yet) and make sure that you are on the root folder (~).

1. OpenCR Setup

1. Install required packages on the Raspberry Pi to upload the OpenCR firmware.

```
$ sudo dpkg --add-architecture armhf
$ sudo apt update
$ sudo apt install libc6:armhf
```

2. Depending on the platform, use either `burger` or `waffle` for the `OPENCR_MODEL` name.

```
$ export OPENCR_PORT=/dev/ttyACM0
$ export OPENCR_MODEL=waffle
$ rm -rf ./opencr_update.tar.bz2
```

3. Download the firmware and loader, then extract the file.

```
$ wget https://github.com/ROBOTIS-GIT/OpenCR-Binaries/raw/master/turtlebot3/ROS2/latest/opencr_update.tar.bz2
$ tar -xvf ./opencr_update.tar.bz2
```

4. Upload firmware to the OpenCR.

```
$ cd ~/opencr_update
$ ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr
```

5. If firmware upload fails, please follow the instructions specified in the manual, in the step 7.

2. OpenCR Test

This process tests the left and right DYNAMIXEL's and the OpenCR board to see whether your robot has been properly assembled.

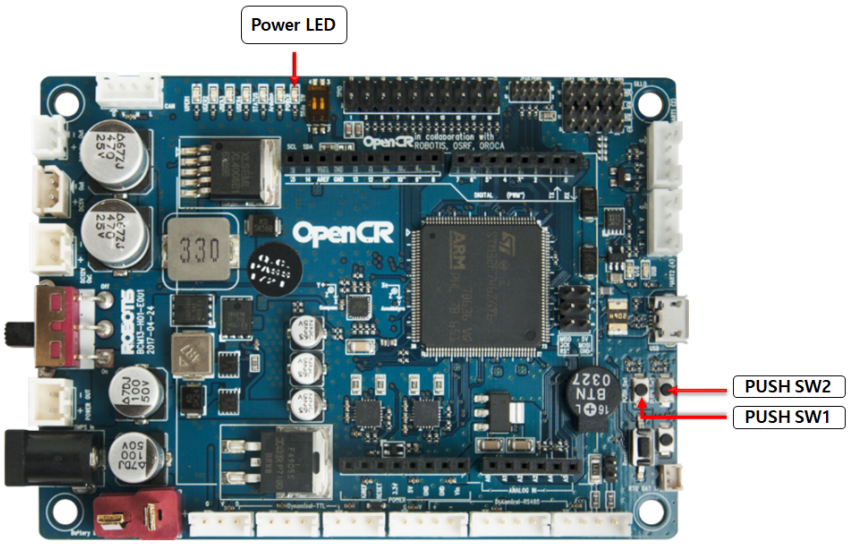


Figure 31: OpenCR Board

1. Switch on the TurtleBot3 and check that the red `Power LED` is turned on.
2. Place the robot on the flat ground in a wide open area. For the test, safety radius of 1 meter (40 inches) is recommended.
3. Press and hold `PUSH SW1` for a few seconds to command the robot to move 30 centimeters (about 12 inches) forward.
4. Press and hold `PUSH SW2` for a few seconds to command the robot to rotate 180 degrees in place.

Bringup

1. Bringup TurtleBot3

1. Connect to the Raspberry Pi and obtain the IP address of the TurtleBot3.

```
$ ifconfig
```

2. Open a new terminal from PC with `Ctrl + Alt + T` and connect to Raspberry Pi with its IP address. The default password is `turtlebot`.

```
$ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
```

3. Bring up basic packages to start TurtleBot3 applications.

```
$ export TURTLEBOT3_MODEL=waffle_pi
$ ros2 launch turtlebot3_bringup robot.launch.py
```

If this is done correctly, you will see a "Run!" message at the end of the terminal.

Basic Operation

1. Teleoperation

WARNING: Teleoperate the robot, and be careful when testing the robot on the table as the robot might fall.

The TurtleBot3 can be teleoperated by various remote controllers: keyboard, RC-100 and PS3 Joystick. We have only use the keyboard option, so we are going to explain only this. If you are interested on the other methods, please refer to the following link.

Keyboard

- **TurtleBot3**: run the *Bringup* before teleoperation (Bringup TurtleBot3).

  1. Connect to Raspberry Pi with its IP address.

  ```
  $ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
  ```

  2. Bring up basic packages to start TurtleBot3 applications.

  ```
  $ ros2 launch turtlebot3_bringup robot.launch.py
  ```

- **Remote PC**

  1. Open a terminal.
  2. Run teleoperation node.

  ```
  $ export TURTLEBOT3_MODEL=waffle_pi
  $ ros2 run turtlebot3_teleop teleop_keyboard
  ```

  If the node is successfully launched, the instructions of how to control the Turtlebot3 will appear in the terminal window.

B Tests

B.1 Procedure

The main objective of this study is to collect data regarding your experience during a chess match against a robot trained to play this game. To do this, you will play a game of chess against the robot. After that, we will send you a survey with several questions about the interaction with the robot, which we expect you to answer as honestly as possible.

Next, we will explain to you how the game will develop and the things you need to keep in mind during the match.

First of all, you have to sit in this chair in front of the robot, trying at all times to keep your hands off the table, except when you have to make your move. So, as you can see in the setup, you will play with the white pieces while the robot will play with the black pieces.

(Wait for the participant to sit down).

Once the game starts, you will see on the screen of the computer to your right an interface with the moves that have been made. This interface is important because it indicates at each moment which player's turn it is by indicating the color of the pieces.

Therefore, until the interface indicates that it is the turn of the white pieces, we ask you not to make any move, as it is still the turn of the black pieces. Once it is indicated that it is your turn, you can make any move you like. However, the system will only recognise legal movements, so when the movement cannot be carried out, the system will not detect it until it is correct.

With all this, we can start the experiment and the game. Good luck and let the best player win!