Towards a Vision-Based Mobile Manipulator for Autonomous Chess Gameplay

Master of Science in Technology Thesis
University of Turku
Department of Computing, Faculty of Technology
Robotics and Autonomous Systems
2023
Bowen Tan

Supervisors:
Ph.D. (Tech) Jorge Peña Queralta
MS.c. (Tech) Xianjia Yu
Prof. Tomi Westerlund

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing, Faculty of Technology

BOWEN TAN: Towards a Vision-Based Mobile Manipulator for Autonomous Chess Gameplay

Master of Science in Technology Thesis, 58 p.
Robotics and Autonomous Systems
July 2023

With the rise of robotic arms in both industrial and research applications, a growing need is observed for autonomous robotic arm applications. This thesis aims to provide an example case of this need and also to showcase the possibility and limitations of vision-based solutions, specifically in automating chess. The focus is on developing a modular system that is able to autonomously recognize chessboard, detect and manipulate chess pieces. The modular design allows for further exploration into autonomous mobile manipulators. The key components include chessboard recognition using fiducial markers to facilitate accurate chessboard recognition and utilizing image processing techniques like segmentation, absolute difference matching, and perspective warping to analyze and extract meaningful information. By mounting a camera above the chessboard, it enables the detection algorithm to accurately capture and analyze the most important information about the environment to determine the current state of the game. Using this information, human move detection is enabled. Then, a custom protocol is utilized to communicate between the detection algorithm and the chess engine, encapsulating information about the game state changes within the system. The chess engine serves the purpose of game analysis and provides legal moves for the robot manipulator to execute. Manipulation happens through careful motion planning and execution, ensuring the safety of the robot and its environment. Extensive evaluation proves that the system demonstrates high accuracy and success rates for piece manipulation and move detection.

Keywords: Fiducial Markers, Robot Arm, OpenCV, ROS

Contents

List Of Acronyms **1**

1 Introduction **2**

1.1 Significance and Motivation 3

1.2 Related works and contribution 4

1.3 Structure 6

2 Background **7**

2.1 ROS 7

2.1.1 tf2 8

2.1.2 MoveIt 9

2.2 Franka Emika Panda 9

2.2.1 Franka Control Interface 11

2.2.2 PREEMPT_RT 12

2.3 Combining the Franka Emika Panda and ROS 2 13

2.4 Camera and detection algorithms 14

2.4.1 ArUco Markers 14

2.4.2 Detection of ArUco Markers 15

2.5 Chess engines 16

2.5.1 Stockfish 18

3 Hardware and Design **20**

3.1 Hardware . . . 20

3.1.1 Franka Emika Panda Robot Arm . . . 20

3.1.2 Design of an improved gripper . . . 21

3.1.3 Camera system . . . 21

3.1.4 Husky - Unmanned Ground Vehicle . . . 23

3.2 Experimental setup . . . 24

3.2.1 Chessboard . . . 24

3.2.2 Camera mount . . . 25

3.2.3 Robot Arm . . . 26

3.2.4 Unmanned Ground Vehicle . . . 27

3.3 Design . . . 28

3.3.1 Flow of the system . . . 28

4 Implementation **32**

4.1 Calibration process . . . 32

4.1.1 Calibration of the camera . . . 32

4.1.2 Calibration of the chessboard's offset . . . 35

4.2 Image Segmentation . . . 35

4.3 Chess Engine . . . 37

4.4 Robot Movement and Coordinate Transformation . . . 41

4.4.1 Interpreting a custom message . . . 41

4.4.2 Transforming coordinates frames . . . 42

4.4.3 Creating the movement interface . . . 44

5 Results **47**

5.1 Performative evaluation . . . 47

5.2 Comparative evaluation . . . 54

6 Conclusion **56**

6.1 Future work . . . 57

References **59**

List of Figures

2.1 ROS Nodes [15] 8

2.2 The transforms of the Franka Emika Panda 9

2.3 tf2 tree of the Franka Emika Panda 10

2.4 Franka Control Interface diagram [19] 11

2.5 ROS control [20] 12

2.6 Example of ArUco markers [22] 15

2.7 Removing the perspective of AruCo marker [22] 16

2.8 Bit extraction [22] 17

3.1 Images of the Franka Hand and its schematics [28] 22

3.2 3D Model of the attachment and a real-life depiction of the model 23

3.3 The Husky UGV with the Franka Emika Panda Arm 24

3.4 Chessboard with ArUco markers 25

3.5 Camera mount 26

3.6 System functionality pipeline, from retrieving the images to moving the arm. (Icons courtesy of flaticons.com) 28

3.7 Before and after warp transformation and center calculation 30

4.1 Full diagram of the system. Each yellow box represents a node in ROS, each dotted box is an integral part of the application of the node and the dotted lines represent the flow. 33

4.2 Example of a ChArUco board 34

4.3 Image segmentation process 38
4.4 All tf frames visualized in rViz 43
4.5 Hierarchy of interfaces used within the system 44

5.1 Full setup 48
5.2 Correct and incorrect orientation of the marker on the arm 49

List of Tables

5.1 Accuracy results of all ten sets performed 50

5.2 Speed comparison per type move 51

5.3 Move detection accuracy 52

5.4 Comparison between three autonomous chess systems 54

5.5 Comparison between the human move detection 54

List Of Acronyms

AB Alpha-Beta

DDS Data Distribution Service

FCI Franka Control Interface

FPS Frames Per Second

MCTS Monte Carlo tree search

Mbps Megabits per second

NASA National Aeronautics and Space Administration

ROS Robot Operating System

SRMS Shuttle Remote Manipulator System

TCEC Top Chess Engine Championships

UGV Unmanned Ground Vehicle

UWB Ultra Wideband

YOLOv5 You Only Look Once V5

1 Introduction

The usage of robotic arms is widespread and is observed in numeral different fields. They are utilized for pick and place work within a factory to critical exploration missions. Robotic arms offer a unique combination of freedom, complexity, and precision, making them suitable for a wide range of applications. Notable examples are the Shuttle Remote Manipulator System (SRMS) [1], The Mars rovers from The National Aeronautics and Space Administration (NASA) [2], and the Da Vinci Surgical Systems [3]. The precision of a robotic arm in these types of fields is of utmost importance. This is why many of these applications are manually controlled through a control interface managed by a human operator or controlled through predetermined movements. However, with the current rise of vision-based detection algorithms and image-processing capabilities, real-time object recognition has improved significantly and the exploration of vision-based autonomous robotic arm applications has been relatively scarce, while the need for these applications seems to grow. Especially in fields where it is seemingly dangerous for a human to operate, a mobile autonomous manipulator can be utilized to a great extent. This thesis aims to bridge that gap by implementing a system in which the robot arm can autonomously play chess using a vision-based solution. While the implementation of the system is rather niche, it aims to showcase the possibilities and limitations of the current technology within the field. The scope of the thesis is restricted to developing a vision-based solution for autonomous chess gameplay using a non-moving robot manipulator. However, it is important to note that the components and techniques that are used can easily be

extended to incorporate mobility. By adding mobility, it allows the robot manipulator to be transformed into a mobile manipulator, enabling it to navigate and interact with multiple chessboards. Lastly, the modular design of the system lays the groundwork for future exploration into autonomous mobile manipulation, expanding the potential applications beyond chess-playing scenarios.

1.1 Significance and Motivation

The main objective of this paper is to showcase the possibilities and limitations of a vision-based autonomous robot arm system. The combination of using off-the-shelf parts and the software implementation proves the usefulness of this type of system which can then be applied in other fields of work, like autonomous vehicle repair, or healthcare support robots (e.g., RoomieBot) and especially those where autonomy and costs are a big factor because it removes the need for any human operators. Also, the benefit of using off-the-shelf parts makes the system more accessible, scalable, and allows for wider adoption in other fields. This solution provides the underlying building blocks from which other implementations can take inspiration.

One of the main benefits of this implementation is the overall generality of the system. Although the paper discusses one version of object and pose detection using fiducial markers, it is not bound to this implementation and other forms like the usage of the OptiTrack system or ultra-wideband (UWB) solutions can also be explored as easily because the system requires merely the positional and orientation data of several points. Thus, even though the final product is an autonomous vision-based chess-playing robot, the components which form the system can be taken individually and used for other purposes as well.

Finally, the final implementation does not require the need of detecting every individual chess piece separately, thus removing the need of training any sort of neural network

or machine learning algorithm. Which in turn, simplifies the system, reduces computational requirements, and makes the components more reusable and not reliant on any specific training data.

As for me personally, I have a strong interest in automated robots in general, but especially those which are used for applications that are usually considered a human activity, like chess or an automatic vacuum cleaner. Furthermore, while I have experience previously working with computer vision and the Robot Operating System (ROS), the thesis provides a great opportunity for me to learn more about the application of robotic arms, motion planning, image segmentation, the integration of chess engines, and mobile manipulators.

1.2 Related works and contribution

In terms of intelligent robotic arms, many projects surrounding the idea of grasping objects have similar high-level goals, which are the picking and placing of objects to specific locations, whether through predefined joint locations (e.g., automated factory arms) or specified vision-based end goals. Previous works in the area of an intelligent vision-based robotic arm for grasping different-sized objects have been done using the approach of Deep Reinforcement Learning [4] [5]. Where in the former the main finding is the realization of a five-degrees-of-freedom robot arm that uses You Only Look Once v5 (YOLOv5) algorithm to detect and reach three-dimensional positions of specific targets using inverse kinematics. In the latter the authors propose a system that detects robotic grasps from RGB-D data using a two-stage deep learning approach, allowing them to avoid hand-engineering features.

Non-Deep Learning approaches have also been considered in [6] and [7] where the robotic grasp movements are done by object localization, pose estimation, grasp estimation, and color and shape detection respectively. The approach for the Non-Deep Learning

method is to perceive the object through image recognition. The idea of an autonomous robotic arm playing chess has been explored before in [8], [9], [10] and [11]. However, in most of these implementations, some kind of neural network or deep learning algorithm is implemented to either detect the chessboard or the chess pieces. Even though artificial intelligence-based object recognition has improved quite substantially over the past decades, it is heavily reliant on proper training data and a controlled environment. Furthermore, detecting the state of a chess game requires the entire board to be visible, which is most feasible by mounting a camera on top or at an angle from the chess board. In the first case, it is complicated to differentiate the chess pieces from each other as they are all relatively circular from above. In the latter case, a situation can occur where pieces are occluded (partially) by one another, which may result in false game state detection. Lastly, as mentioned before, proper training data is required for artificial intelligence-based approaches. This means that a chess set that has not been trained properly may fail to detect the pieces or game state completely.

There have been projects done where exact chess piece recognition is not used, like in [12], where their approach was to detect the game state through the rules of chess and the natural feature differences between the pieces and chess board. However, in their work, the approach is limited by the specific reference frame which the algorithm calibrates to. In other words, it is likely that the color calibration that is used is tuned for a specific (modified) chess set. In this work, a fiducial marker approach is considered to detect the size of the chess board, where then the game state is assumed using previous frames and the natural progression of a move. Then, a chess engine is utilized to calculate the moves to be made by the computer and lastly, the Franka Emika Panda Arm is used to move the pieces. With this approach, the need for a modified chess set is removed and thus makes the whole solution more scalable. The robot arm is also situated on an unmanned ground vehicle (Husky) with the idea that multiple chess games can be played simultaneously using multiple cameras. Not only that, with the addition of having the robot arm on

the Husky, further works with vision-based autonomous robotic arm movements can be explored where the arm is a mobile entity rather than a stationary one. Summarized, the core contributions of this thesis are as followed:

1. Showcase the possibilities and limitations of vision-based autonomous systems with robotic arms by designing and implementing a chess-playing robot using fiducial markers.
2. Provide working components of the system which can be used for other implementations and further work.
3. Design the system in a way that allows further exploration to be done using different object detection based approaches (e.g., OptiTrack or UWB).

1.3 Structure

- **Chapter 2** introduces the background information on the components used to create this project.
- **Chapter 3** will go over the hardware and setup of the system and how the system functions on a high level, combined with a rationale that describes why certain design choices were made.
- **Chapter 4** focuses on the implementation details of the system and a more technical overview of how the system works.
- **Chapter 5** presents the results of the project, a performative, and comparative analysis.
- and finally, **Chapter 6** will conclude the thesis and give recommendations for further work.

2 Background

This chapter provides background information on the concepts used to implement the system.

2.1 ROS

The Robot Operating System (ROS) is a set of open-source software libraries and tools created for building robot applications. It provides the necessary services for hardware abstraction, low-level control, and package management. Its main advantage comes from the ability for developers to build and reuse code between robotics applications without the need to go through a rewriting process, thus reducing the time needed between development cycles. Knowledge from one robotics area can be applied across all platforms, from drones, to robot arms, to mobile bases [13].

ROS 2 is the successor of the previous ROS 1, going through a complete redesign of the framework, improving upon the shortcomings of the first generation, and adding new packages and backward compatibility [14]. The main difference between the two versions is the changes to the architecture. ROS 1 uses a Master-Slave Architecture while ROS 2 uses Data Distribution Service (DDS), which is designed to provide higher efficiency and reliability, especially for real-time systems.

The main component that is used within ROS 1 and ROS 2 is nodes. A node in ROS has the responsibility of a single, module purpose, for example, a node for controlling the joints of a robot arm, another node for detecting objects. A complete system, therefore,

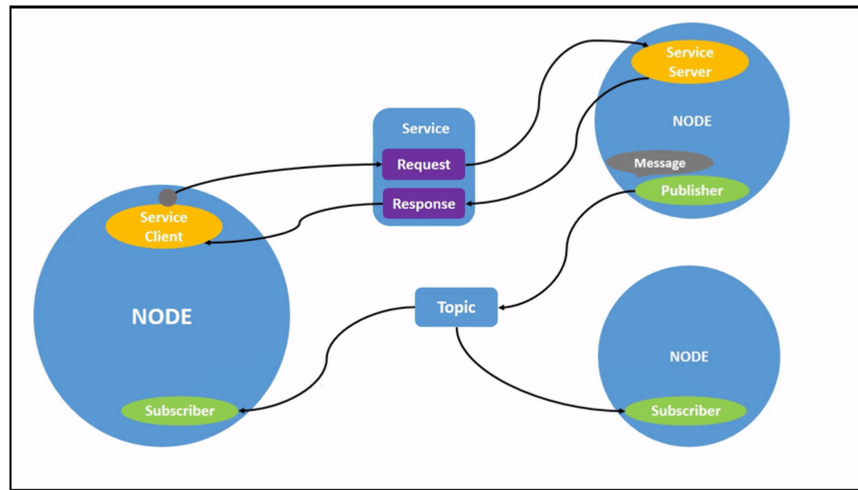


Figure 2.1: ROS Nodes [15]

consists of many nodes working together with one another. In Figure 2.1, a simplified version of the communication path between nodes can be seen [15]. Communication works through exchanging messages through topics. Nodes can subscribe and publish to topics that are labeled with unique identifiers. The benefit of this architecture is that it is highly customizable, nodes can easily be exchanged for others as long as the topics published and subscribed to remain the same. For example, a node responsible for detecting the fiducial markers and publishing its pose is easily exchangeable with a node that uses a different method of pose estimation (e.g., OptiTrack or UWB).

2.1.1 tf2

tf2 [16] is the transform library of ROS2. It enables the user to keep track of multiple coordinate frames over time. It also allows for maintaining the relationship between any multiple coordinate frames, given they are linked together through the same tree structure. For example, the robot arm has its own tree structure describing the relationship between its joints, and the object detection system also has its own tree that showcases the relationship between itself and the objects that are detected. With *tf2*, it is possible to link these two tree coordinate frames which allow for either tree to retrieve information from

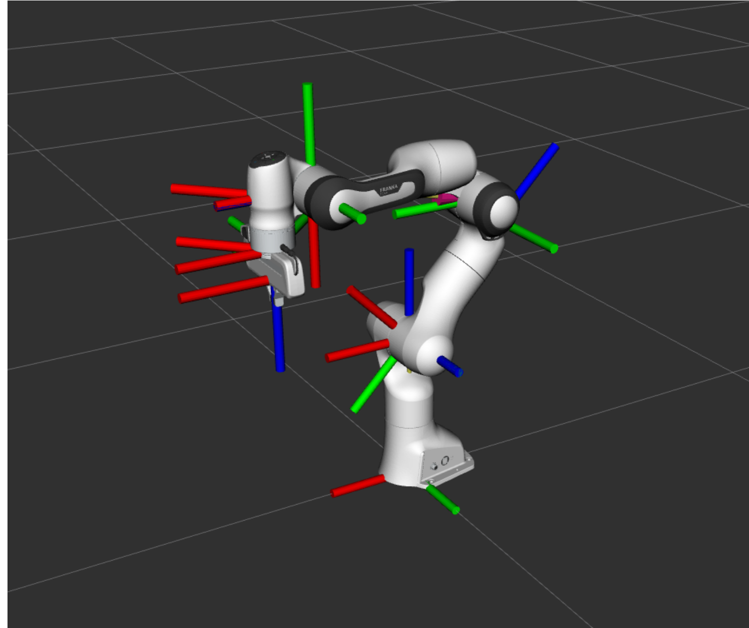


Figure 2.2: The transforms of the Franka Emika Panda

the other tree.

2.1.2 MoveIt

MoveIt [17] is the motion planning framework integrated with ROS. It provides a set of tools and libraries specifically designed for motion planning, control, and manipulation tasks in robotic systems. Furthermore, it offers key features that are useful in controlling the robot arm. For example, collision detection, trajectory planning, and the usage of (inverse) kinematics.

2.2 Franka Emika Panda

The Franka Emika Panda [18], also known as the Franka Emika Research 3, is a robot system that includes an arm and a control. The arm features 7 degree-of-freedom with torque sensors at each joint, is force sensitive, and comes with industrial-grade pose repeatability of +/- 0.1mm accuracy. Additionally, the control features an interface that

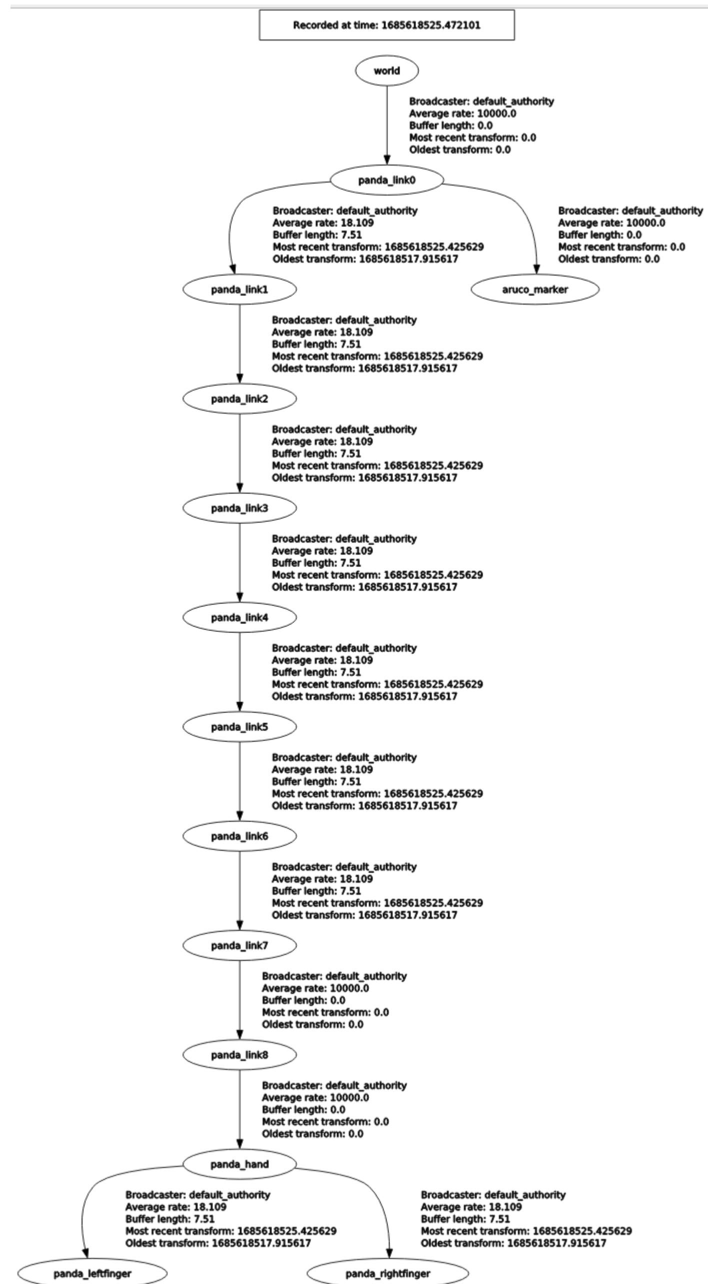


Figure 2.3: tf2 tree of the Franka Emika Panda

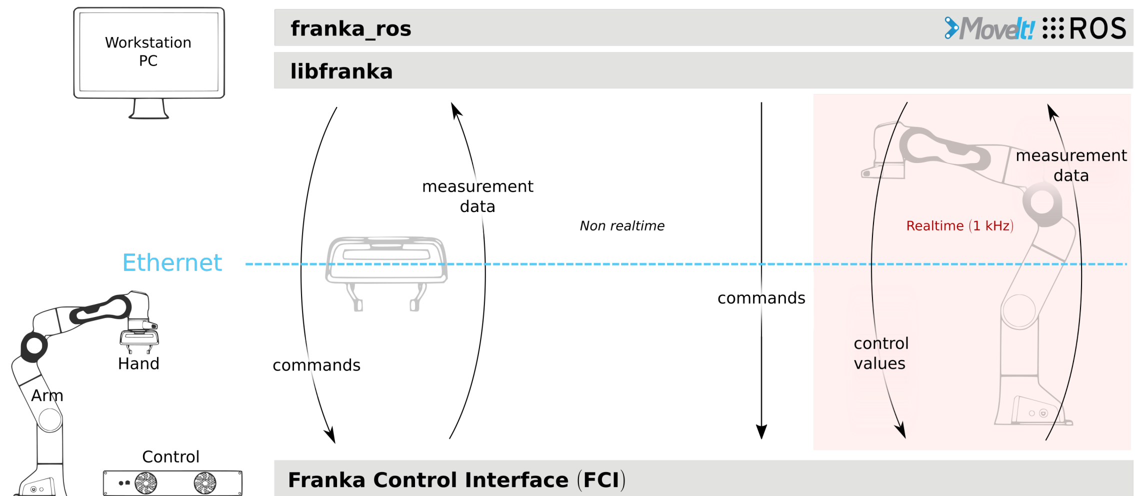


Figure 2.4: Franka Control Interface diagram [19]

offers developers to explore low-level programming and control schemes. Providing the user with its current status and enabling direct torque control at 1 kHz.

2.2.1 Franka Control Interface

The Franka Control Interface (FCI) [19] is an interface designed to allow fast and direct low-level bidirectional connection to the Arm and Hand. It provides the current status of the robot and enables control with an external workstation PC. Using *libfranka*, which is a library developed by the Franka Emika group to control the arm, real-time control values are sent at 1 kHz for interfaces like gravity and friction, joint position, Cartesian pose, joint data, and estimations of externally applied torques and forces, including various collision and contact information. The FCI is used in this project to retrieve the joint positions and robot state from the arm.

In addition to *libfranka*, *franka_ros* can be used to connect Franka Emika robots with the ROS ecosystem including ROS Control. ROS Control is a set of packages designed to take joint state data from a robot's actuator's encodes and enables output control using generic control loop feedback mechanisms[20]. A general diagram of data flow between

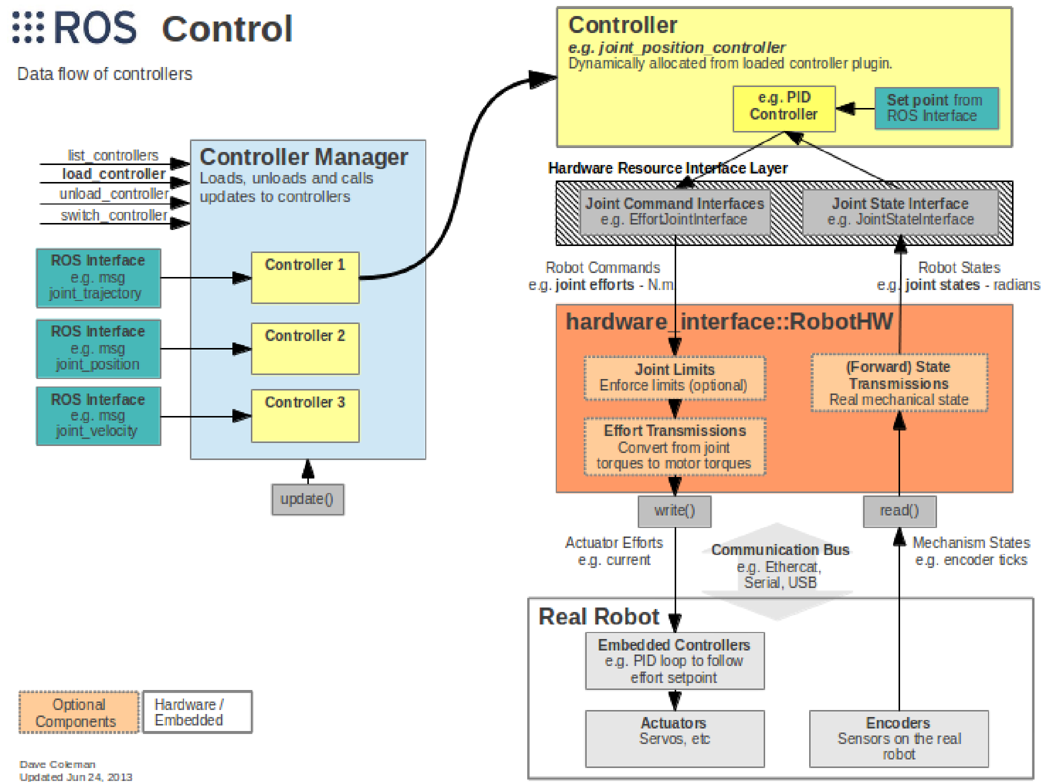


Figure 2.5: ROS control [20]

controllers in ROS Control can be seen in Figure 2.5.

2.2.2 PREEMPT_RT

In order for swift and reliable communication between the Franka Control Interface and the Panda arm to be possible, there are some minimum requirements set for the workstation; one of which is a network card that supports 100BASE-TX, meaning it should be able to carry data traffic at 100Mbps over Ethernet. Because of the 1kHz data frequency, it is also recommended that the workstation is to be configured to minimize latency. For example, by disabling CPU frequency scaling [21].

Another requirement is that the operating system runs Linux with a PREEMPT_RT patched kernel (or a Windows 10 Experimental installation). PREEMPT_RT is a Linux

real-time kernel patch that makes Linux into a real-time system. The main difference between a real-time system versus a non-real-time system is that a real-time system is time-bound and has fixed time constraints, which are necessary for the bridge between the workstation, the FCI, and the robot arm to communicate effectively. The usage of a real-time system however doesn't necessarily correlate to improved performance, it simply aims for reduced response latency instead of optimized throughput.

2.3 Combining the Franka Emika Panda and ROS 2

There are two main ways to control the robot arm, one of which is using the provided library: *libfranka* by Franka Emika. The other one is communicating through ROS 1 or ROS 2 using MoveIt.

To control the robot arm using *libfranka* there are two separate types of commands; non-real-time commands and real-time commands. The non-real-time commands are blocking and are based on the TCP/IP-based protocols. They encapsulate the commands for the gripper and several configuration commands for the arm itself. The real-time commands require a reliable and fast connection from the workstation to the Franka Control Interface.

For this work, the choice is made to control the robot using the MoveIt interface from ROS 2. Using the ROS 2 interface makes it easier and more clear how to communicate between several nodes and setups (e.g., the detection of the chess board happens in a separate node). Furthermore, MoveIt has the benefit of providing built-in motion planning, collision checking, and trajectory execution. All of which are useful in the final implementation of the system. Motion planning is used to determine whether a calculated move is achievable by the robot in its current state, with respect to self-imposed objects in the MoveIt world frame (collision checking). When the planning succeeds, the trajectory for the arm executes automatically and safely. However, relying solely on the MoveIt motion

planning algorithm meant that any loss of robot states in a crucial moment leads to failure in planning and execution. This is also one of the reasons why a reliable and strong connection is required between the workstation and the FCI.

2.4 Camera and detection algorithms

One of the key implementation features of this system is the usage of an RGB camera to detect the position of the chessboard and the arm. The position of the chessboard and the arm in reference to the camera are used together with ROS 2 transforms to make the arm able to move to specific locations on the chessboard. In this work, fiducial markers are used to achieve the detection of the chessboard and the arm. The fiducial markers serve as reference points for the camera and the detection algorithm to determine the position of the chessboard. One benefit of this implementation is that any kind of RGB camera can be used to achieve the same results, given the camera is calibrated correctly and has a high enough resolution (e.g., 1920x1080x30 was used in this project). Calibration is needed because the detection of the fiducial markers relies on a properly calibrated camera to be able to accurately estimate the pose and orientation of the markers by correcting the natural distortions of the camera image. Furthermore, a camera that can not output a high enough resolution may fail to detect the markers entirely due to the lack of image pixel data. Further explanation of the calibration process and detection of the chessboard through fiducial markers can be found in Sections 3.3.1 and 4.1

2.4.1 ArUco Markers

ArUco markers are one of the most popular approaches when it comes to pose estimation through computer vision. A marker is comprised of an inner binary matrix that determines its identifier and an outer black border. The inner binary codification together with a surrounding black border makes the ArUco marker especially robust and relatively easy

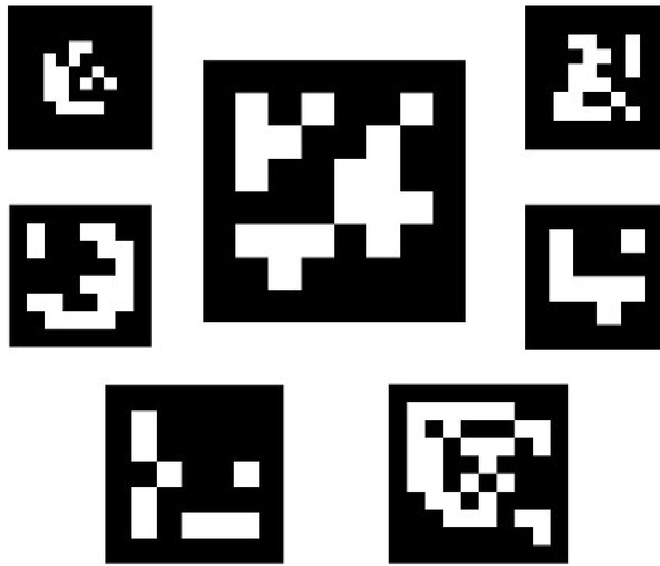


Figure 2.6: Example of ArUco markers [22]

to detect by a camera. This also enables the possibility of applying error detection and correction techniques like perspective transforms, which is one of the techniques used in this project (see Section 3.3.1). One of the main benefits of using binary square fiducial markers like ArUco is that a single marker alone can provide enough correspondences (the four corners of the marker) to obtain the position and orientation [23].

2.4.2 Detection of ArUco Markers

To be able to retrieve the position and orientation of a marker, it first must be detected and segmented from the camera image. This detection process comprises of two distinct steps, the detection of marker candidates and the identification of the candidates [22].

- ***Detection of marker candidates:*** In this step, the image is analyzed to find possible candidates that could be markers. This is done by applying an adaptive threshold to segment the markers from other parts in the frame. Then, contours are extracted from the image and the remaining contours that are convex enough (i.e., roughly represents a square) are considered for identification.

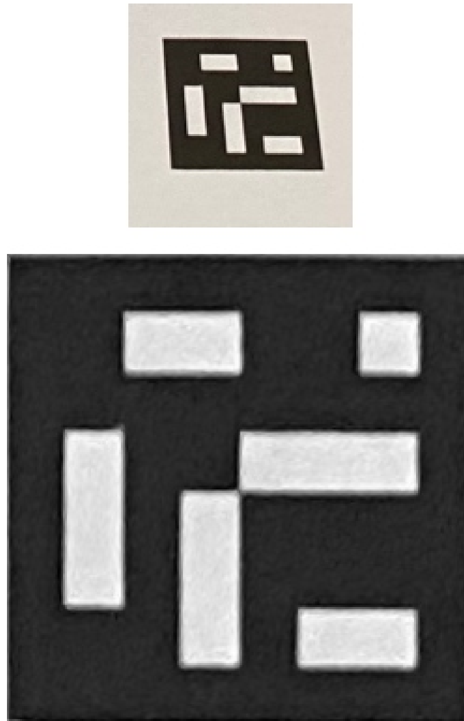


Figure 2.7: Removing the perspective of AruCo marker [22]

- ***Identification of possible markers:*** After possible candidates are found, the inner codifications of the markers are analyzed. This starts with transforming the perspective of the camera image to retrieve the canonical form of the marker. Then, the candidates are segmented with a threshold to separate the white and black bits. Then, these bits are analyzed to determine if they belong to any predefined dictionary assumed by the ArUco library.

2.5 Chess engines

A chess engine [24] is a software program that can play and analyze chess positions. They have been around since the 1900s but it wasn't until 2005 that the first chess engine was able to defeat the best human players. Surpassing even grandmasters, which is the highest possible achievable rank within the chess world. Over time, humans have become

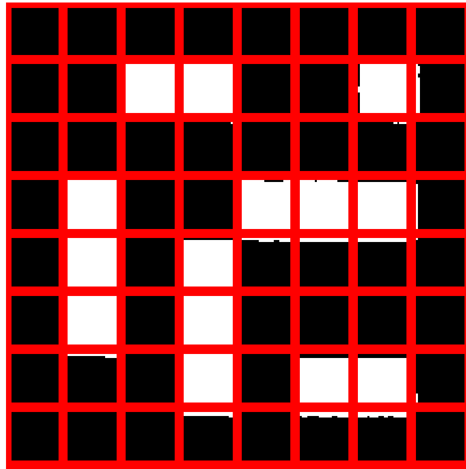


Figure 2.8: Bit extraction [22]

better and better at chess through learning from previous games and collective wisdom. Chess engines have only accelerated this process by providing deeper knowledge and understanding of the game.

A traditional chess engine works by looking at individual positions and evaluating which position is better. A position is evaluated through numerous factors, like material balance, piece activity, pawn structure, king safety, possible moves, and more. Most chess engines display an evaluation number that indicates the advantage of one side over another. (e.g., +1 means a one-point advantage for white, while -1 would mean a one-point advantage for black). It evaluates which position is better by searching and comparing each possible move and its reply and then the reply to that, up to a certain depth. Because the comparisons for every move grow exponentially with the number of possible moves available, traditional chess engines tend to use a strategy called "*smart pruning*". This means the chess engine will deliberately ignore obvious bad moves and their respective replies to reduce the number of moves needed to evaluate.

With the rise of artificial intelligence and machine learning, recent years have also seen the introduction of neural network chess engines [24]. Neural network engines train by learning from previously played games, or by playing games against themselves. It differs from traditional engines in the way how they search for the best possible moves.

In traditional engines, only the best possible moves are evaluated using an alpha-beta (AB) minimax search [24]. In neural network engines, a more common approach is to use the Monte Carlo tree search (MCTS). MCTS works on a principle where the best move is selected based on probable outcomes of many play-outs.

2.5.1 Stockfish

Stockfish [25] is an open-source and widely considered the strongest traditional chess engine available for use, finishing in first place in 8 out of the last 10 seasons of the Top Chess Engine Championships (TCEC) and coming second in the other two. To paint a picture of how strong Stockfish is, the average human player who knows how to play chess is around 800 Elo. Elo is a type of rating measurement system that evaluates how strong a player is, where with each win you gain Elo and with each loss you lose Elo, depending on the strength of your opponent. The world's best player at the time of writing, Magnus Carlsen, is around 2850 Elo, and Stockfish is estimated to be over 3500 Elo. Like any traditional chess engine, its main purpose is to evaluate the position of a chess board given a position. It's mostly used by people who want to analyze either their own games or games played by other people. It assists strongly in understanding the strengths and weaknesses of each position, suggests alternative moves, and provides deeper insights for future moves. In over-the-board chess tournaments, players are encouraged and sometimes even required to record their moves by writing them down in algebraic notation [26]. Algebraic notation is a form of reading and writing chess moves that is universally understood. With the exception of the knight piece, which starts with the letter 'N', each move begins with the first letter of its name in the English language (e.g., rook = R, queen = Q, etc.) followed by the square it moved to (e.g., e4). So for example, the first ten moves of a match could be:

1. Nf3 Nf6 2. c4 g6 3. Nc3 Bg7 4. d4 O-O 5. Bf4 d5 6. Qb3 dxc4 7. Qxc4 c6 8. e4 Nbd7 9. Rd1 Nb6 10. Qc5 Bg4

Naturally, there are more rules and standards when it comes to the algebraic notation but the main takeaway is that this way of notation can be used easily with a chess engine like Stockfish to analyze the game move-by-move. Additionally, games that are played online are usually recorded automatically and can easily be exported and analyzed immediately after a game.

3 Hardware and Design

This chapter goes over the hardware components and experimental setup of the project. It solely covers which components were used, how it was set up, and gives a brief description of the integration of each component. Then, an overview is given of the design of the system. For a more detailed implementation explanation, see Chapter 4.

3.1 Hardware

3.1.1 Franka Emika Panda Robot Arm

As mentioned before in 2.2, the Franka Emika Panda Robot Arm is a 7-degree-of-freedom with torque sensors at each joint. Having an approximate weight of 18 kg, it is possible to handle weights up to 3 kg, which is more than sufficient for this system. Via the *franka_ros2* interface, which is a ROS 2 integration of the *libfranka* library, joint positions and velocities of the arm can be retrieved. Then, using ROS 2's *MoveGroupInterface* [27], which is a part of MoveIt, enables the planning and executing joint and pose trajectories with the arm. The main benefit of using MoveGroupInterface over directly controlling the robot's motors through joint positions and velocity controls is that with each desired move, a plan is created. The plan checks whether the move is reachable and most importantly, safe. For example, moves that violate the constraints of the joints or will lead to a joint colliding with itself or nearby objects will not be allowed to execute.

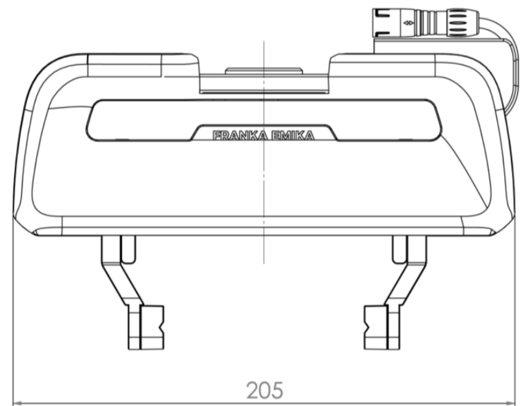
3.1.2 Design of an improved gripper

Normally, the robot arm comes equipped with a Franka Hand (see Figure 3.1). However, due to the irregular widths, heights, and forms of the chess pieces, the usage of this specific end effector is not reliable enough. For example, the tallest piece, the king, has a height of 9 cm while the shortest piece, the pawn, has a height of 4 cm. Because in this implementation no piece detection algorithm is used and the detection of the chessboard is not always accurate (more on this in Section 4.1.2) it means that even a slight inaccuracy in either the detection of the chessboard or the difference in height of the pieces can lead to the arm completely missing the desired piece and hit an adjacent one instead, causing the safety features of the arm to abort the program. Luckily, the end effector can easily be adapted and re-calibrated using the Franka Desk by measuring the new minimum and maximum widths of the gripper. Therefore, A new gripper design is made that serves as an attachment for the Franka Hand. This new design is one where the fingertips are extended and widened to accommodate the previously mentioned limitations. The main benefit of this new attachment is that there is a bigger surface area, so even when the pieces are not completely centered, the new attachment is able to grab the piece. Furthermore, the attachment makes the vertical and horizontal area uniform and flat, which makes grabbing the different-sized and shaped pieces easier. The 3D model and printed-out version of the attachment are depicted in Figure 3.2

3.1.3 Camera system

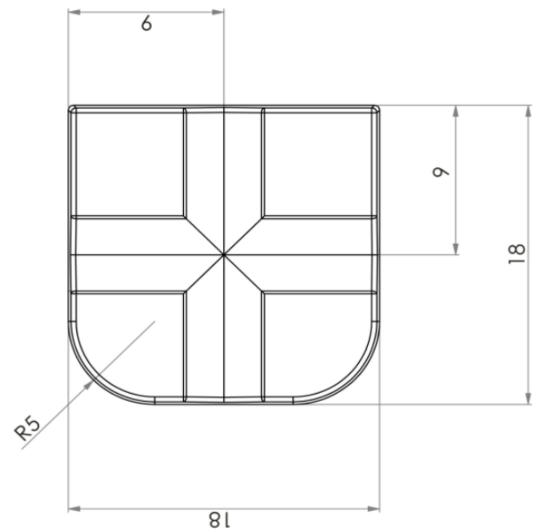
The images retrieved for this project come from a RealSense D435 camera. It is a depth camera using a stereo solution and comes with four imaging sensors. Meaning it uses a left and right imager, and an IR projector to estimate depth with a resolution of 1280x720 up to 90 frames per second (fps). Furthermore, it comes with an RGB sensor that supports a resolution of 1920x1080 up to 30 fps. For this thesis, the usage of the depth module is unnecessary due to the usage of fiducial markers. Therefore, using any RGB camera that

(a) Franka Hand



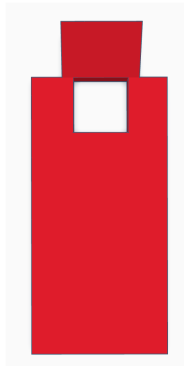
(b) Schematics of the Franka Hand

(c) Fingertip



(d) Schematics of the fingertip

Figure 3.1: Images of the Franka Hand and its schematics [28]



(a) 3D Model of the attachment

(b) Printed out model

Figure 3.2: 3D Model of the attachment and a real-life depiction of the model

supports resolutions equal to or higher than 1920x1080 should suffice. It is not recommended to go lower than this resolution due to the fact that with lower-quality images, it becomes more difficult for the fiducial markers to be detected properly. Thus, it hinders the detection algorithm of the chessboard and the functionality of the robot.

3.1.4 Husky - Unmanned Ground Vehicle

In the experimental setup, the robot arm is situated on a Husky unmanned ground vehicle (UGV). The Husky is a medium-sized, all-terrain, and rugged robot designed for development in robotics. With a weight of around 50kg and a maximum payload of 75kg, it can easily carry the robot arm, which weighs around 18kg. The husky with the robot arm attached can be seen in Figure 3.3.

Figure 3.3: The Husky UGV with the Franka Emika Panda Arm

3.2 Experimental setup

For this project, an experimental setup is created where the robot arm is mounted on the husky, the chessboard is at a height of around the base of the robot arm, and the camera is mounted roughly 2 m above the ground. This setup is depicted in Figure 5.1.

3.2.1 Chessboard

To facilitate accurate and reliable chessboard detection, four ArUco markers are strategically placed at each of the four corners of the chessboard. They form a frame of reference for the camera detection system to be able to create the grid that is needed for getting the coordinates of each of the squares of the chessboard. Each ArUco marker has a size of 4x4 cm with a white border of 0.7 cm. The white border ensures that the inner bi-

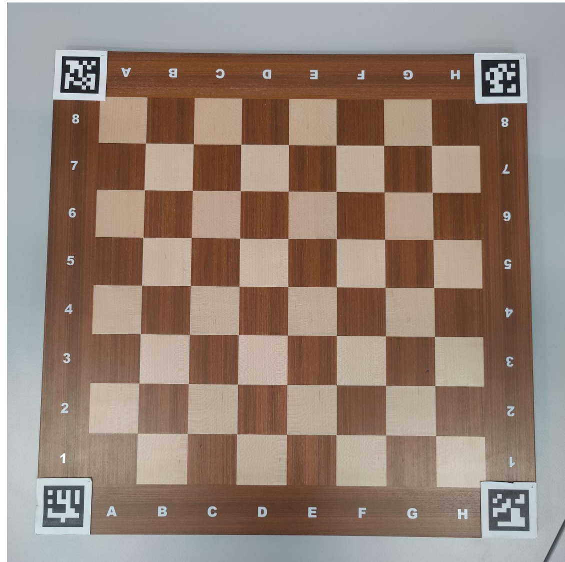


Figure 3.4: Chessboard with ArUco markers

nary codification of the marker is detected more reliably by providing a stark contrast between the black outline of the marker and the inner codification. This, in turn, enables the ArUco detection algorithm to more easily detect and identify the markers, leading to a significantly improved effectiveness of the entire program.

Furthermore, next to the marker on the top-right of the chessboard in the camera frame, another marker is placed. This marker's purpose is for the detection algorithm to determine where to put the captured pieces of the human player. Lastly, the chessboard is situated roughly at the height of the base of the robot arm. The height of the chessboard matching the base is merely coincidental and does not have a significant impact on the functionality of the robot. However, if the chessboard is placed out of reach of the maximum limits of the arm, but still in the frame of the camera, the system will naturally not be able to plan movements to the desired goals and thus fail to move.

3.2.2 Camera mount

As mentioned before, the camera is mounted roughly 2 meters above the ground on a tripod. This decision is made based on the consideration that the robot arm that is mounted

Figure 3.5: Camera mount

on the husky reaches a height of around 1.6 meters from the ground. So mounting the camera above this point ensures that the chessboard and the arm are visible. Furthermore, it guarantees that the marker on the arm (see Figure 3.7(c)) is visible when the arm is in its *Ready* state. The *Ready* state is the state where the robot arm is programmed to move to when waiting for a new command. This configuration allows for minimal obstruction of all the markers needed by the detection algorithm but also maximizes their relative size in the camera frame, making them more easily identifiable.

3.2.3 Robot Arm

The robot arm serves as the manipulator of the system, it moves the chess pieces that are identified by square by the camera detection algorithm. By putting an ArUco marker near the top of the arm, the detection algorithm is able to determine the position of the arm in relation to the chessboard. This relation establishes an important link between the

coordinates obtained from the camera and the internal transform tree of the robot arm. Through this link, the system is able to move the arm to a desired goal identified by the camera. Finally, attached to the end effector is the previously mentioned custom-designed gripper (see Section 3.1.2) to enable a more secure and reliable grip on the pieces.

3.2.4 Unmanned Ground Vehicle

Placing the robot arm on the husky enables an otherwise stationary arm to be mobile. Theoretically, this allows the system to play multiple chess games simultaneously over multiple boards by moving the husky over to the board where another chessboard and camera are set up. This way of playing multiple games at the same time is also known as *simultaneous chess* or a *simultaneous exhibition*, which can be seen performed by masters of the game to showcase their knowledge and memory. Not only that, manning the robot arm on the UGV allows for future research in the field of mobile manipulation. For example, the usage of the mobile manipulator in hazardous environments like a nuclear power plant. These facilities often have dangerous levels of radiation, which pose a significant risk to human workers. By deploying a mobile manipulator, the normally unsafe activities can be performed with minimal risk by the robot.

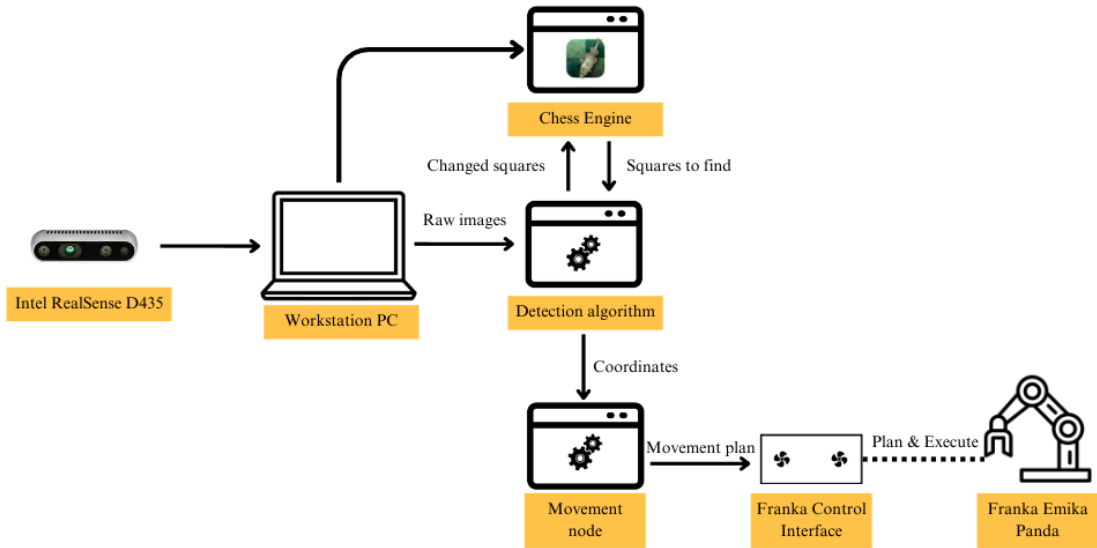


Figure 3.6: System functionality pipeline, from retrieving the images to moving the arm. (Icons courtesy of flaticons.com)

3.3 Design

This section will discuss how everything communicates and works together on a high level. This section is especially important because it explains more generally how the system functions, which is the base that is needed to understand the full implementation details, which can be found in Chapter 4.

3.3.1 Flow of the system

Detection

Figure 3.6 showcases the general flow of the system that enables the system to communicate and function together. The process begins by retrieving the raw images from the camera, which serve as input for the detection algorithm. The detection algorithm uses these images in conjunction with the intrinsic parameters of the camera to accurately identify and estimate the pose of the ArUco markers. The intrinsic parameters, such as

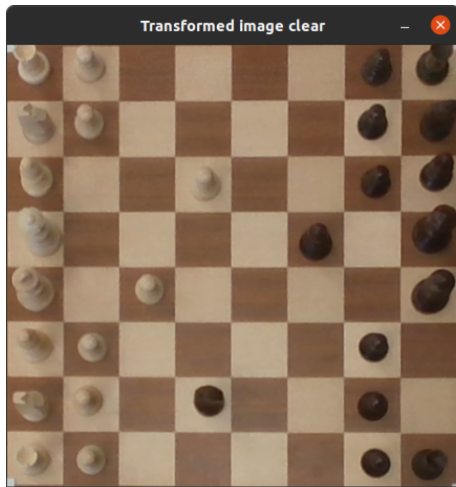
the focal length and the optical center of the camera, are pre-calibrated information about the internal characteristics of the camera. Because each camera can have different focal lengths or optical centers, it is important that the camera is calibrated. Otherwise, the pose estimation of the ArUco markers becomes unreliable.

Once the ArUco markers are detected, the detection algorithm performs a warp transform to convert the image into a 2D representation of the chessboard. This transformation is made possible due to the strategically placed ArUco markers on each of the corners of the chessboard. The warped image provides a top-down view of the chessboard, which facilitates easier analysis. Using this new top-down view, an 8x8 grid is created by dividing the image. Because the warped image covers solely the chessboard, the 8x8 grid can represent each square of the chessboard. Then, to determine the location of the chess pieces, which for this project are assumed to be near the center of each square, the center of each square is calculated. These center coordinates are then re-projected back to the original image and into 3D space, using the intrinsic parameters of the camera.

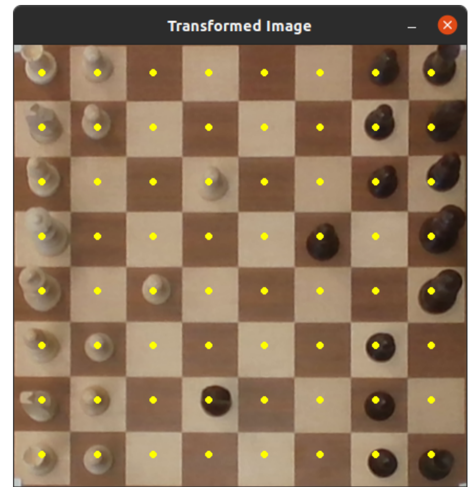
Using these 3D re-projected points and the ArUco marker that is detected on the arm, it is possible to create a link between the camera coordinate frame and the coordinate frame of the arm using *tf2*. Subsequently, this link enables the detection algorithm to transform the coordinates from the camera frame to the robot arm's frame.

Chess Engine

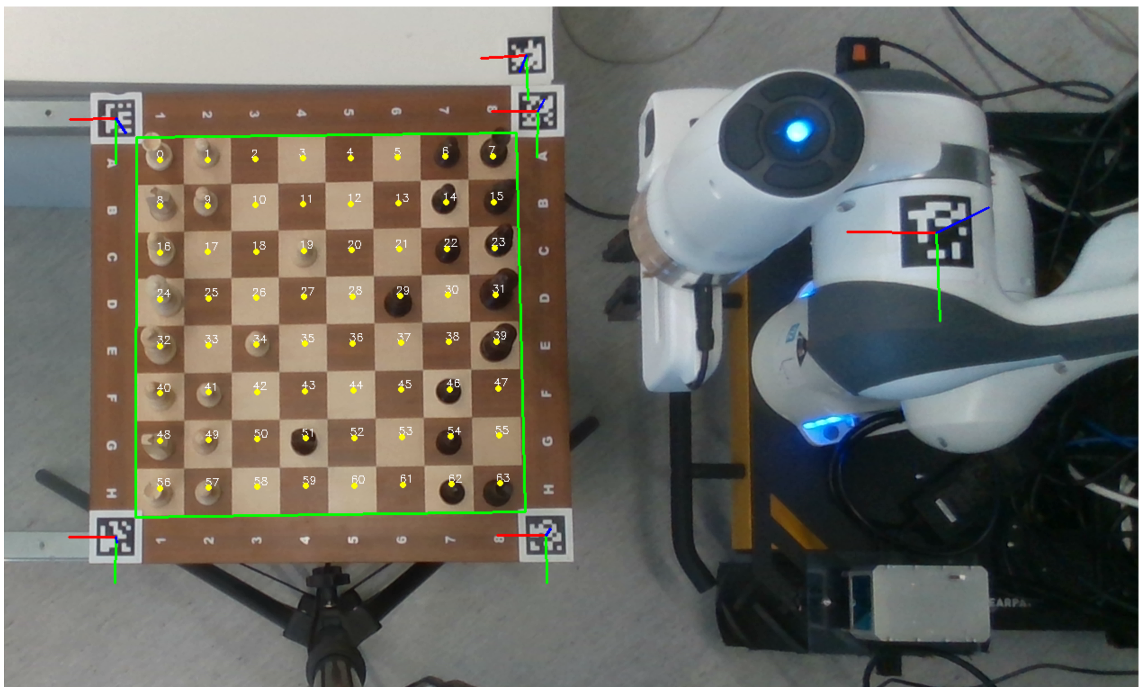
The chess engine plays an important role in the system's flow, especially when it comes to determining the legality of the moves made by the human player and allowing the game to progress. Depending on which player is making the next move, the flow of the system differs slightly. If it's the human player's turn, the chess engine node sends a message to the detection algorithm node, prompting it to capture the current frame of the chessboard and save it. The player is then asked to make a move and input a keystroke to back to the program. Once the move is made, another message is sent. This time, the



(a) Transformed image



(b) Image with the center calculated



(c) Re-projection back to the original image

Figure 3.7: Before and after warp transformation and center calculation

detection algorithm captures the new frame and compares it to the previous frame. The node then analyzes the differences between the frames and determines the squares that have changed. These changes are then remapped back to the chessboard notation and fed back to the chess engine, which will check the legality of the move. Naturally, a legal move will be parsed and update the game state, while an illegal move will be ignored and the user will then be prompted to make a new move.

If it is the robot's turn, the chess engine does not have to send any message to the detection algorithm, as the calculation of the next move is done internally within the chess engine node. Because playing against a computer that is capable of calculating moves up to a very deep depth with unbeatable accuracy is not fun, the design choice is made to lower the overall difficulty of the program so that an average player could still have a chance. However, adjusting the difficulty is very trivial and thus can be changed easily accordingly to the player's skill level.

Making the robot move

Once the next move calculated by the chess engine is completed, a custom message is formed. This custom message encapsulates the type of the move (e.g., a normal move, a capturing move, castling, etc.) followed by the squares that the robot needs to move to. Then, the movement node generates desired arm trajectories for manipulating the given chess squares. If the generation completes, meaning the trajectory is possible and within the limitations of the joints, the robot is allowed to move and execute the trajectory and manipulate the pieces. Once the movement trajectories are completed, the arm returns to a *Ready* state, waiting for a new command.

4 Implementation

This chapter covers a detailed overview of the implementation of the system. Mainly, it provides a more technical view of the components used in the project. Figure 4.1 showcases a fully integrated diagram of the system.

4.1 Calibration process

In this section, two different calibration processes are described: the calibration of the camera sensor and the calibration of the chessboard offset from the base of the robot.

4.1.1 Calibration of the camera

As mentioned before in Section 2.4, camera calibration is a crucial step in ensuring accurate detection of the ArUco markers. The camera's intrinsic parameters and distortion coefficients, which are obtained through the calibration process, remain fixed until changes are made in the camera's optics. This means that the calibration process only has to be performed once. In order to calibrate the camera, a ChArUco board is utilized. A ChArUco board is a printed grid that closely resembles a chessboard but with the white squares populated by ArUco markers. In order to calibrate the camera to retrieve its intrinsic parameters and distortion coefficients, multiple frames of the ChArUco board have to be captured at different angles [29]. Then, the built-in function of OpenCV, *calibrateCamera()* is used with the given images.

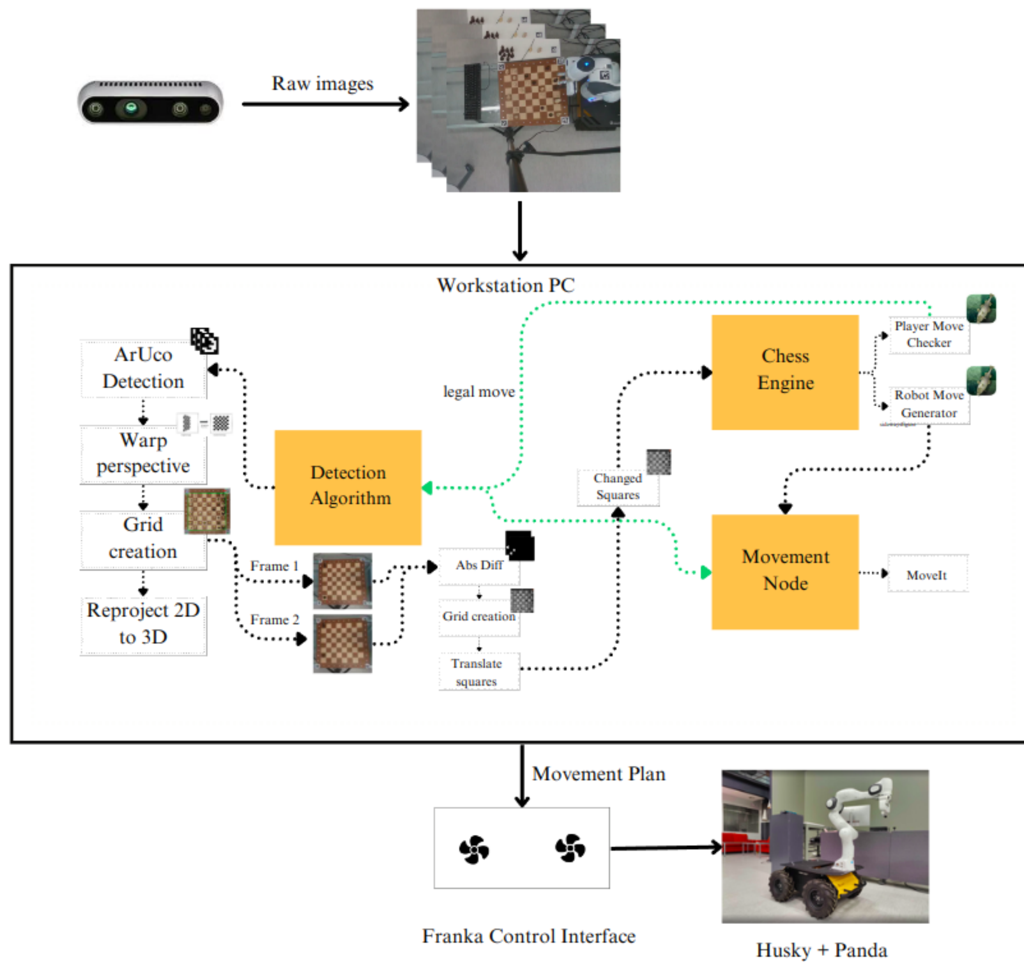


Figure 4.1: Full diagram of the system. Each yellow box represents a node in ROS, each dotted box is an integral part of the application of the node and the dotted lines represent the flow.

Figure 4.2: Example of a ChArUco board

The advantage of using a ChArUco board for camera calibration over a traditional chessboard-like pattern or just ArUco markers is that the ChArUco board allows for partial occlusions. This means that during the calibration process, it is not mandatory for all corners of the markers to be visible at all times.

4.1.2 Calibration of the chessboard's offset

Although the camera is calibrated using the ChArUco board and the pose detection of ArUco markers is relatively accurate, the markers still suffer from ambiguity issues [30]. This means that regardless of calibration, there are some errors in the detection of the markers. While this error margin is relatively small, in the context of this project it is pretty significant. Because the chess pieces are positioned close to each other, these small errors can cause the gripper to miss the piece or hit an adjacent one, causing the program to engage its safety feature and abort the entire program.

To alleviate this problem, at the start of the program, a calibration process is done for all the positions and orientations of the markers. Once the measuring is completed, the median positions and orientation for each marker are saved. Then, the robot arm is tasked to move to each corner of the chessboard. The user can then adjust the position and orientation according to the error observed by the system. Once the chessboard is adjusted, the calibration is complete. This manual calibration of the error and the chessboard's offset allows for the final setup to be more reliable.

4.2 Image Segmentation

In order for the chess engine to be able to respond to the player's moves, accurate and reliable move detection plays a crucial role in the system's functionality. The primary objective of the move detection algorithm is to identify the changes that occur on the chessboard between consecutive frames. To achieve this, image segmentation is utilized.

Image segmentation is a computer vision technique that partitions a digital image into multiple image segments. By partitioning the image into segments, it enables further processing and analysis of the important aspects of the image, like color, contours, depth features, etc.

The approach used in this work compares two consecutive frames captured by the camera and analyzes the absolute difference between them. By analyzing the absolute difference between these two frames, it is possible to identify the areas in which the changes have occurred. Adding to this, because the analyzed frames include solely the chessboard, the same technique described in Section 3.3.1 to divide the grid is used here as well. The combination of analyzing the absolute differences and dividing the image into an 8x8 grid allows the system to determine which chess squares have changed.

Naturally, relying solely on the technique of analyzing the absolute difference between two consecutive frames is not reliable. While it does provide a straightforward measurement of pixel-level changes between frames, it does not account for various factors that may affect the detection accuracy. For example, changes in lighting conditions or the accidental movement of a chess piece that was not part of the final move. Therefore, several checks were put into place to combat these limitations. In chess, a move can lead to the following changes on the chessboard:

1. **Two squares change**: in the case of moving a piece without capturing another piece and a piece capturing another piece.
2. **Three squares change**: in the case where a pawn captures another pawn through en passant.
3. **Four squares change:** in the case where one side castles their king and rook.

Knowing these characteristics, at least two squares and at most four squares can change at any given move. Therefore, any fewer or more changes can be discarded immediately. Furthermore, the movement of a piece causes a relatively large change when

analyzing through absolute differences. Thus, another check is put in place where the size of the detected change in the region of interest is compared to a threshold. If the size does not exceed this threshold, the change is also discarded. Algorithm 1 showcases the implementation of the image segmentation process.

Algorithm 1 Image segmentation algorithm

Require:

Two consecutive RGB images obtained by the camera: $frame1$, $frame2$

Original image with coordinates of the divided grid

if $receivedMessage$ **then**

  Convert $frame1, frame2$ to grayscale

  Blur $frame1$ and $frame2$

  Get the absolute difference between $frame1$ and $frame2$: $absDiff$

  Run a binary threshold on $absDiff$: $threshImage$

  Find contours in $threshImage$

  for $contours = 1, \ldots, N$ **do**

    if $contourSize \geq threshold$ **then**

      Add the corresponding chess square to $changedSquares$

    end if

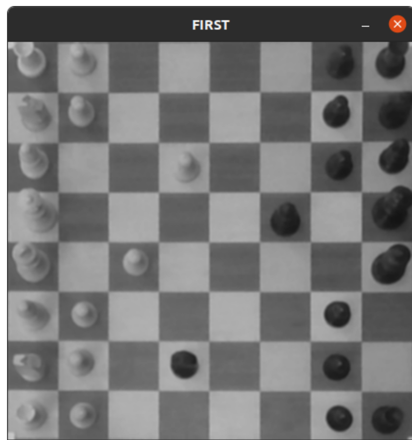
  end for

  Send $changedSquares$ to chess engine

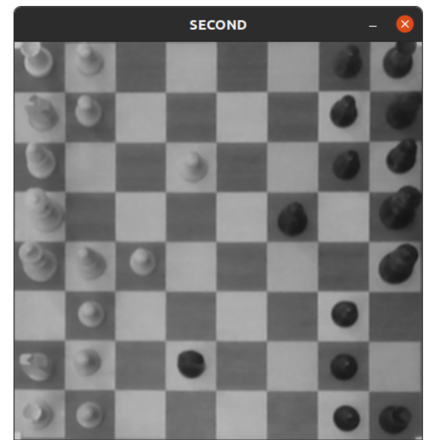
end if

4.3 Chess Engine

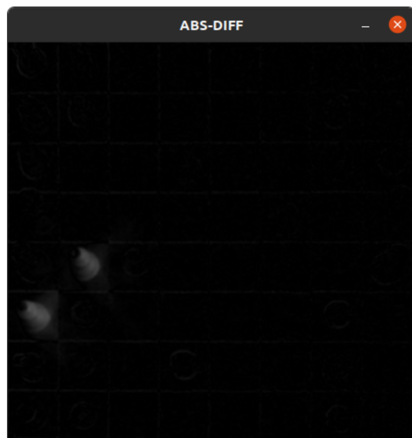
Determining the optimal move in the autonomous chess robot system is done by utilizing an existing chess engine named Stockfish, which has already been covered in detail in Section 2.5.1. To reiterate, Stockfish is an open-source chess engine that uses sophis-



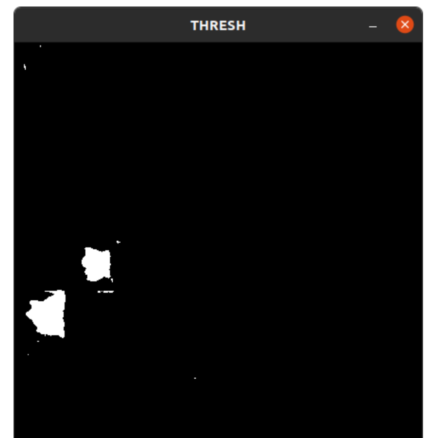
(a) Frame 1



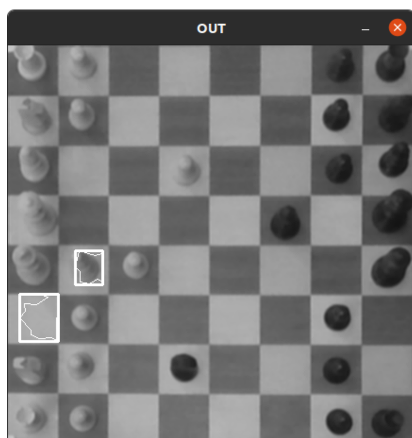
(b) Frame 2



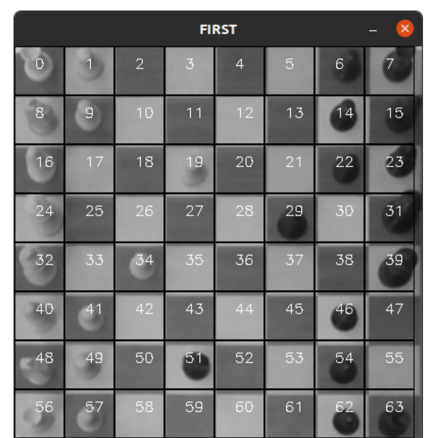
(c) Absolute difference



(d) Threshold



(e) Merged



(f) Grid division

Figure 4.3: Image segmentation process

ticated algorithms to evaluate chess positions and calculates the best position based on chess principles and strategies.

The way Stockfish is integrated into this system is by utilizing its core functionality of calculating optimal moves based on the current state of the board. As mentioned previously in 4.2, the player's move is determined by way of image segmentation, and the chess squares that correspond to the changes seen by the detection algorithm are fed to the chess engine. For example: *"33 35"* (moving a pawn from squares 'e2' to 'e4').

Once the changes are fed to the chess engine, it parses the move by translating the numerical values of the message to their corresponding alphanumeric name. It uses this new alphanumeric value to determine whether the move made by the user is legal. If the move is deemed to be illegal, it prompts the user to undo the move and make a new one. If the move is legal, then the chess engine updates its own current state of the board and generates a list of optimal moves. Then, a randomly chosen move out of the list is parsed into a custom message, depending on what type of move Stockfish chooses, the message is constructed in a different way:

- **Moving a chess piece:***"MOVE: startSquare endSquare*"
- **Capturing a piece:** *"CAPTURE: startSquare endSquare*"
- **En passant:** *"CAPTURE_EP: pawnCaptured pawnStart pawnEnd"*
- **Castling:** *"CASTLE: kingStart kingEnd rookStart rookEnd*

The variable *startSquare* represents the chess square number corresponding to the current position of the moving piece, while *endSquare* denotes the destination square of the piece after the move. In cases where a pawn is involved in capturing, the variables *pawnToCapture*, *pawnToMoveStart*, and *pawnToMoveEnd* are used to indicate the specific pawn being captured, its initial position, and its final position as part of the capturing move, respectively.

For castling moves, the variables *kingStart*, *kingEnd*, *rookStart*, and *rookEnd* are employed. These variables signify the starting and ending positions of the king and rook pieces involved in the castling maneuver.

Using this custom protocol enables easy parsing of the type of move for the detection algorithm. This is necessary because each type of move requires a different set of movements that are instructed to the robot arm.

Algorithm 2 Chess engine algorithm

Require:

Message received with the changed squares

if $receivedMessage$ **then**

  if $message$ = $empty$ **then**

    Prompt user to make a (new) move

  else

    Parse the message into alphanumeric value: $move_input$

    Check the legality of the move $move_input$

    if $move_input$ = legal **then**

      Update chess board state

      Generate a list of optimal moves: top_moves

      Randomly choose a move out of top_moves: $move$

      Construct and publish custom message using $move$

    else

      Prompt user to undo their move and make a new one

    end if

  end if

end if

4.4 Robot Movement and Coordinate Transformation

Moving the robot with respect to the camera's coordinate frame is the final step of the sequence. This section delves into the process of interpreting the message generated by the detection algorithm and performing the necessary transformations to achieve accurate movement of the arm.

4.4.1 Interpreting a custom message

In the previous section, a custom protocol was introduced. This custom protocol is used by the detection algorithm and the movement node to be able to communicate with each other. Different moves in chess require different sequences of movement. Therefore, it is important that the message conveys the type of chess move so that the movement of the arm can be planned and executed correctly.

For example, moving a piece requires just the movement of that specific piece from one square to another, without other interaction of other pieces. Therefore, it is sufficient to encapsulate the message with the starting square of the piece and the destination square. However, in a more complicated move like capturing or castling, the sequence of movements is more complex. In the case of capturing, the correct sequence would be to:

1. Pick up the piece that has to be captured.
2. Move the piece off the board to a discard pile.
3. Pick up the piece that did the capturing from its starting square.
4. Move the piece to captured piece's original square.

and in the case of castling it would be:

1. Pick up the king from its starting square.
2. Move the king to the destination square.

3. Pick up the corresponding rook for the castling move.
4. Move the rook to the opposite side of the king.

Thus, depending on the move type, a specific set of movements have to be executed, and using this custom protocol enables this possibility.

Algorithm 3 Parsing the custom message

Require: Custom message received

if $receivedMessage$ **then**

  if $move_type$ = $CALIB$ **then**

    Start the calibration phase

  else if $move_type$ = $MOVE$ **then**

    Move piece from $origin$ to $destination$

  else if $move_type$ = $CAPTURE$ **then**

    Capture piece: $pieceToMoveSquare$ $pieceToDiscardSquare$ $discardPile$

  else if $move_type$ = $CASTLE$ **then**

    Castle: $kingOrigin$ $kingDestination$ $rookOrigin$ $rookDestination$

  else if $move_type$ = $CAPTURE_EP$ **then**

    Capture: $pieceToMoveSquare$ $pieceToCaptureDestination$ $discardPile$ $pieceToCaptureOrigin$

  end if

end if

4.4.2 Transforming coordinates frames

Before being able to move the robot arm to pick up the chess pieces, it needs to know where to move to in its own coordinate frame. Luckily, the $franka_ros2$ integration keeps track of its own (joint) positions. This means that the robot at all times knows the positions of each of the joints and their relation to each other. However, the positions of

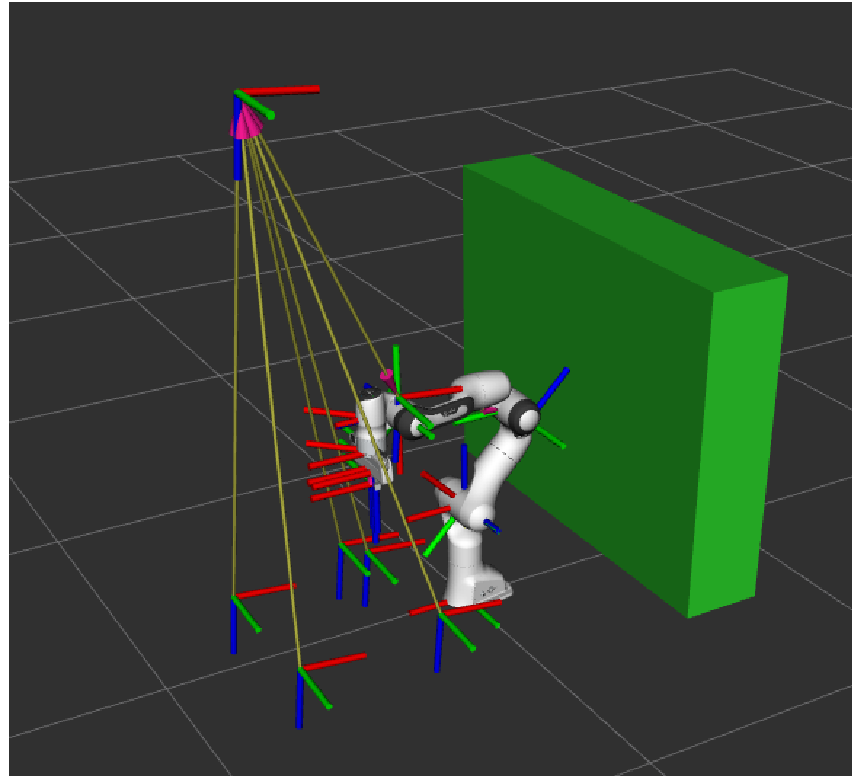


Figure 4.4: All tf frames visualized in rViz

the chessboard's squares retrieved from the detection algorithm do not reside in the robot arm's coordinate frame. Rather, they are part of the camera's coordinate frame. Thus, a link needs to be created between the camera's coordinate frame and the robot's coordinate frame.

In order to achieve this, the usage of another ArUco marker is utilized. By strategically placing the marker in a fixed position next to one of the robot arm's joints, a static transform can be created between the ArUco marker and the joint in the robot arm's coordinate frame (The marker on the arm can be seen in Figure 3.7). Adding to this, because the transform of the ArUco marker is linked to one of the joints, the marker is now also part of the transform tree of the robot arm and therefore also shares a relation with all other joints. This means that the relative position and orientation of the marker can now be retrieved from every joint.

Furthermore, having this marker placed on the arm enables the detection algorithm

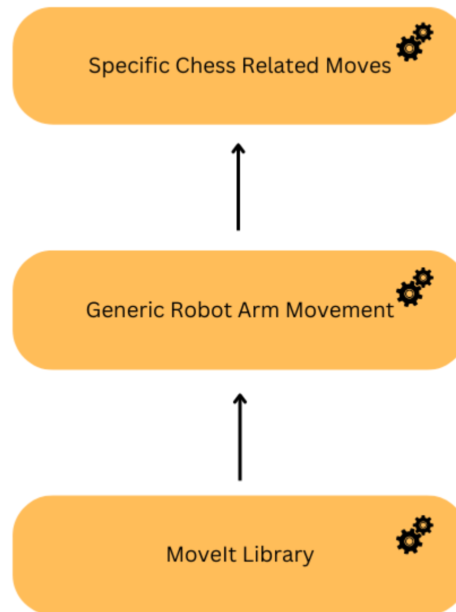


Figure 4.5: Hierarchy of interfaces used within the system

to identify the marker as well, but now in the camera's coordinate frame. Now that the position and orientation of the ArUco marker exist in both coordinate frames, a link between the two frames can be created. This allows for the system to transform positions found in one coordinate frame to the other coordinate frame. For example, the detection algorithm finds the coordinates of the squares that the robot needs to move to, but in its own coordinate frame. Using this transformation link, the system is now also able to tell the robot arm where those positions are in the robot's coordinate frame. Figure 4.4 shows all the tf frames visualized in rViz. Here, each of the joints of the Panda arm is showcased alongside the frames for the ArUco markers and the camera.

4.4.3 Creating the movement interface

In this work, the *MoveGroupInterface* from the *MoveIt* library is used to manipulate the robot arm. With this interface, before execution, the desired movement has to be planned.

The main benefit of this approach is that it disallows movements that are seemingly impossible to achieve by the robot arm. Thus, if the detection algorithm requests the robot arm to move to a position outside of its reach, the planning phase for this movement fails and in turn, does not get executed.

Furthermore, using the *MoveGroupInterface* also allows the imposing of (virtual) objects in the robot's world frame, and also the addition of constraints to specific joints. Adding objects to the robot's world frame can simulate real-life objects that are in similar positions. For instance, in this work, the robot arm is situated on the husky alongside an aluminum frame that houses several sensors and computers. To avoid hitting this frame, the sensors, or the computers, a virtual object can be added. This object is used to tell the *MoveGroupInterface* to avoid planning a collision.

Adding constraints can be used to restrict the movement of a specific joint of the robot arm. This can be useful in situations where it is unwise to move a specific joint past a certain threshold. For example, to avoid hitting objects near that joint.

Building on top of the *MoveGroupInterface*, a generic robot arm movement interface is created. The goal of this interface is to use the generic planning and execution functions of the *MoveGroupInterface* in combination with the robot arm to allow the movement of the arm. This in-between interface is specifically created with the idea to keep it generic and thus further emphasizing the portability and re-usability of the system's individual components. However, as mentioned before, there are four different types of moves within chess. Thus, the decision is made to create another interface that is specific to chess-related moves. This chess-specific interface encapsulates all the different types of movement sequences that are required.

Algorithm 4 showcases the picking and placing of one specific piece. All other types of moves are based on this pick-and-place algorithm. In this algorithm, the origin and destination locations of the piece and the current location of the gripper are required. First, the robot is asked to move about 20 centimeters above the desired piece, followed

by moving the gripper down. The idea of first moving above the piece instead of going in a straight line is to limit the possibility of knocking other pieces over during the trajectory. Then, the piece is grabbed, moved up and to the desired square, moved down, and released. Finally, the robot arm is told to return to the *Ready* position. The *Ready* position is a pre-defined location for the arm to move to once it has completed a movement sequence, allowing for the ArUco marker on the arm to be visible and further movements to be planned.

Algorithm 4 Moving a piece

Require: Poses of origin square, destination square, and gripper: $origin$, $dest$, $gripper$

Ensure: p_1 ... p_n = completed before moving on to $p_n + 1$

p_1: Move above $origin$ using $gripper$ as start

p_2: Move the remainder down

p_3: Grab the piece

p_4: Move up

p_5: Move above $dest$

p_6: Move the remainder down

p_7: Release the piece

p_8: Move to $Ready$ position

5 Results

In this chapter, the results of the experimentation done with the autonomous chess robot system will be presented. The performance of the system in terms of accuracy, speed, and effectiveness of marker and move detection will be discussed. Additionally, the impact of network speed on the overall performance of the system will also be explored. The aim is to showcase the end results of the implementations alongside their limitations. Figure 5.1 shows the final setup of the system and lastly, to demonstrate the functionality of the system, a time-lapsed full chess game was recorded and uploaded [31].

5.1 Performative evaluation

A big factor in whether the autonomous chess system can succeed is the accuracy of the detection algorithm and the movement of the robot arm. As previously mentioned in Section 4.1.2, the ArUco markers have an issue of ambiguity. In this work, it means that the detection of the markers is not always reliable and accurate for the orientation. Because the transformation of the coordinates from the camera's frame to the robot arm's frame is reliant on the accurate detection of the orientation of the marker on the arm, any inaccurate detection during the calibration of the markers can lead to undesired end behavior of the system, where the system assumes the chessboard is at a different location and orientation than it is in reality. Figure 5.2 showcases an example of correct and incorrect orientation.

Furthermore, despite the correct detection and calibration of the orientation of the

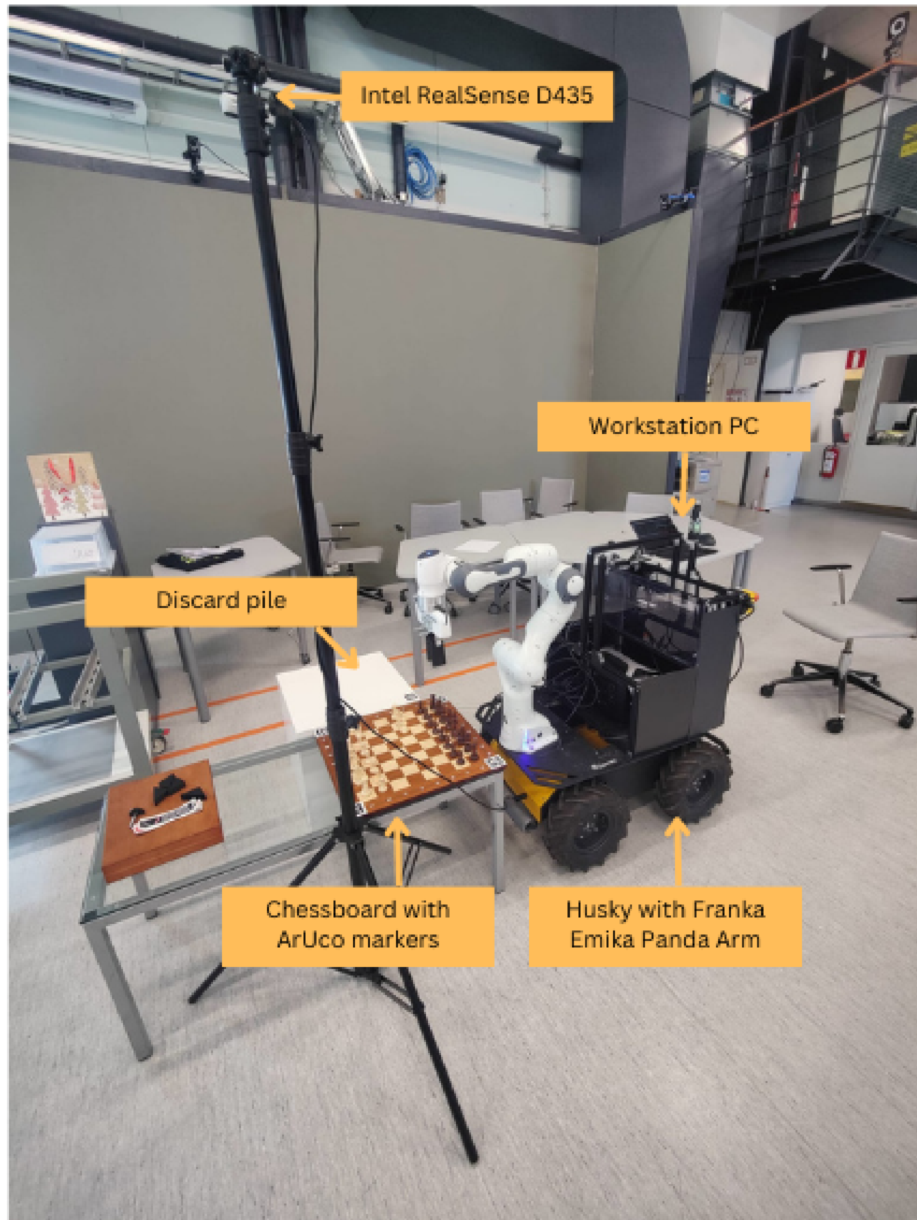
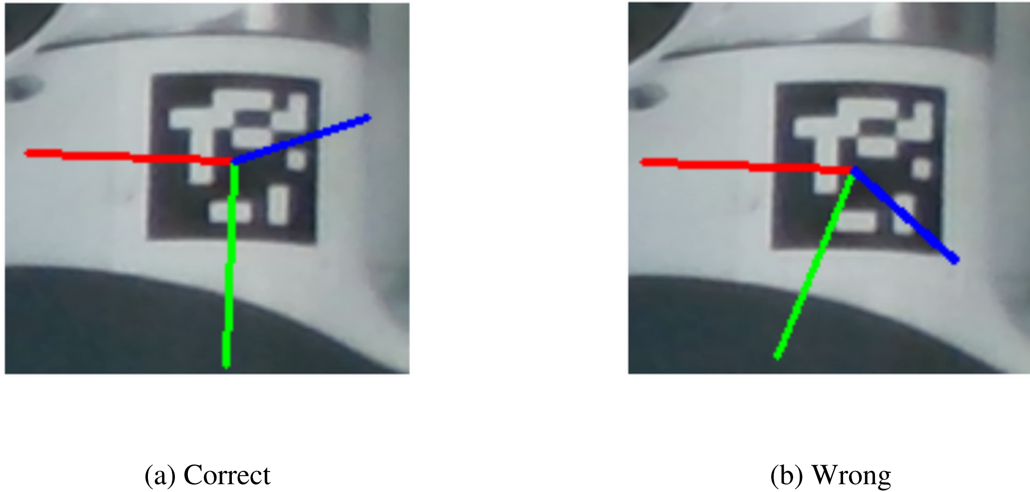


Figure 5.1: Full setup



(a) Correct (b) Wrong

Figure 5.2: Correct and incorrect orientation of the marker on the arm

ArUco markers, the combination of error margins in marker detection, perspective transform, and grid creation within the camera detection algorithm can introduce certain inaccuracies. These inaccuracies exhibit variations in both direction and magnitude, which are specific to each system restart. On average, these inaccuracies range around 0.75 cm $\pm$ 0.25 cm in any direction. Among these inaccuracies, the most problematic scenario occurs when the y-direction of the gripper is affected. This is due to the horizontal space between each chess piece being approximately 3 cm, while the width of the gripper when fully open is around 6.5 cm. The gripper's width is designed to accommodate the pieces comfortably between its two fingers. Consequently, when there is a y-direction inaccuracy of approximately 1 cm, the risk of the 3D-printed gripper's lower part colliding with the adjacent piece significantly increases. This poses a challenge to the accurate and precise manipulation of the chess pieces. However, by utilizing the manual calibration sequence done at the start of each system setup, the accuracy of the system can improve significantly to the point where practically every move that is unhindered by other factors like network issues, performs exceptionally well. This is due to the fact that the calibration sequence saves the detected positions of the markers and keeps them as a static location.

Set	Expected amount of moves	Succeeded	Failed	Success rate (%)	Comments
1	10	2	5	40%	Joint limit reached after 7th move
2	10	0	1	0%	Gripper hit chess piece on 1st move
3	10	2	4	33%	Gripper hit chess piece on 6th move
4	10	1	1	50%	Gripper hit chess piece on 2nd move
5	10	7	3	70%	
6	10	10	0	100%	
7	10	5	1	80%	Robot state lost after 5th move
8	10	7	1	85.7%	Robot state lost after 7th move
9	10	10	0	100%	
10	10	2	1	66%	Movement timed out after 3rd move

Table 5.1: Accuracy results of all ten sets performed

Thus, when the errors are accounted for, the detection of the markers and the movement of the arm are consistently accurate.

Accuracy

To assess the accuracy of the arm movement and emphasize the significance of the calibration sequence, a comparative experiment is conducted involving 10 sets of randomly selected movements repeated 10 separate times on the system. The first 5 sets of movements are performed without manually calibrating the chessboard, while the second set of 5 is executed after a manual calibration process. The results of this experiment, as presented in Table 5.1, highlight the notable differences between the two scenarios. In each set, 10 moves were expected to be completed. However, when the system fails for any reason other than missing the piece (e.g., hitting an adjacent piece or the communication timing out), the set ends and a new set begins.

As anticipated, the system that goes under manual calibration demonstrates better performance compared to the non-calibrated system. However, it is important to note that both systems showcase instances of both success and failure. In the case of the non-calibrated system, despite its overall lack of accuracy in reaching some squares, certain

Type move	**Samples**	**Min** (s)	**Avg** (s)	**Max** (s)
Moving	50	21.99	24.04	27.31
Capturing	50	47.01	47.93	48.96
Capturing (en passant)	50	47.13	47.96	48.58
Castling	50	46.51	46.87	47.19

Table 5.2: Speed comparison per type move

squares display sufficient precision, enabling the robot to execute the intended moves.

Conversely, for the calibrated system, it is observed that instances of failure are primarily attributed to a loss of robot state, where the system believes that a joint was at a different position than it is. This phenomenon is potentially caused by the network card in the workstation or network-related issues, which will be explored in detail in a subsequent section.

Speed

Table 5.2 shows the minimum, average and maximum time taken per type of move. Measurements are taken from 50 samples for each type of move to determine the time required per move. For the experiments, the velocity and acceleration gain for the robot arm are set to 20% of its maximum value. This is done to observe and monitor the movements safely without risking damage to either the pieces, the chessboard, or the robot itself. The decision is deliberately made not to increase the speed of the robot due to the likelihood that the robot produces an error and causes the arm to make an unexpected movement. If the speed is to be increased substantially, it can lead to a dangerous situation where the arm would damage itself or its surroundings.

	Moves made	**Correctly identified**	**Incorrectly identified**	**Percentage correct**
Game 1	23	21	2	91.3%
Game 2	22	17	5	77.3%
Game 3	27	24	3	88.9%
Game 4	19	17	2	89.5%
Game 5	29	29	0	100%

Table 5.3: Move detection accuracy

Move detection

Table 5.3 shows the accuracy of the move detection algorithm over 5 full games of chess that are played against the autonomous chess robot. The evaluation process involves comparing the detected moves with the actual move made by the human player. The number of properly detected moves is used as a metric to assess the performance of the move detection module. Notable things are that the incorrectly identified moves were mainly caused by a white piece moving directly to another white square. In some situations, the absolute difference in the comparison between the two frames does not yield a high enough threshold for the algorithm to reliably conclude a move is made. Thus, prompting the user to make a new move. One possible explanation for this phenomenon is that there is a high likelihood that the chess pieces are made from the same material as the chessboard itself. Therefore, in certain lighting conditions where a distinct shadow is not cast from the pieces to the chessboard, or the lighting is very uniform, the top-down view of the camera might not be able to capture a distinct difference between the two frames. Thus, causing the move detection algorithm to believe no move was made.

Importance of connectivity

During the experimentation for the results, a returning issue of the system is that the position of a seemingly random joint is not at the expected location. When this happens,

the *franka_ros2* integration together with MoveIt prevents the arm from completing the planned path for safety reasons. Because the manipulation and moving of a chess piece are all planned out before the actual movement occurs, an error causing one part of the plan to not execute results in the whole plan failing. The assumption is that the network card used in the workstation PC does not fulfill the exact requirements of the system. This is confirmed by running the communication and latency test provided as a benchmark by Franka themselves. In this benchmark, 10000 robot states are requested by the workstation from the robot arm. It evaluates the number of robot states that are lost as well as the minimum, average, and maximum control command success rate [32]. The results of this test on the workstation PC give a warning, stating that the PC might not be suitable for use with the FCI. Running this same benchmark on a different system with a more high-end network card gives no warning. However, even on this workstation, the same reoccurring issue of the joint positions remained. Thus, the causation of this error remains unknown and could be caused by many different aspects like network congestion, connectivity to the network, network cable, problems in the built-in library, or others.

Name of the System	Total manipulations made	Successes	Success Rate
Gambit	786	720	91.6%
WizardChess	80	64	80%
Developed System	696	657	94.4%

Table 5.4: Comparison between three autonomous chess systems

Name of the System	Human move detection	Successes	Success Rate
WizardChess	40	35	87.5%
Developed System	120	108	90.0%

Table 5.5: Comparison between the human move detection

5.2 Comparative evaluation

This section aims to provide a comprehensive analysis and comparison of the results obtained by the system at hand with two other autonomous chess robots. This comparison allows for a deeper understanding of the performance of the system in relation to already existing solutions. The two other chess robots, [33] and [34], selected for this evaluation are chosen based on their similar objectives with move detection and piece manipulation. The results of the evaluation can be seen in Table 5.4 and Table 5.5

Manipulation

In both [33] and [34] a distinction is made between a full move and the manipulations needed. A full move, therefore, consists of multiple manipulations. In all three systems, the manipulation mainly focuses on grasping a piece and dropping a piece at the right place. As shown, the developed system in this thesis scores a slightly higher success rate than the other two systems. The manipulations that are analyzed are based on a calibrated final system, where the errors of the chessboard are accounted for.

Human move detection

Regrettably, the Gambit chess system did not include any data on human move detection and thus, only the WizardChess system is compared with the developed system here. The developed system demonstrates a success rate of 90% over 120 total moves made, while the WizardChess system achieves a success rate of 87.5% over 40 moves. Although slightly lower than the developed system, the results still suggest that the other system's detection algorithm, which is based on a sixty-four magnetic reed switches along an X-Y Cartesian board, is still capable and reliable for capturing the majority of human moves.

However, when comparing the two systems, it is noted that the vision-based detection algorithm used in this system is slightly superior to the other approach. This may be attributed to the robustness of the image segmentation and grid creation algorithms implemented in the system, which enable precise identification and analysis of the chessboard. Having a high detection rate in an autonomous chess robot is crucial for the functionality of the whole system and this approach showcases that a vision-based implementation for human move detection can be reliable and robust.

6 Conclusion

Overall, this thesis has presented the development and implementation of an autonomous chess robot system that utilizes a vision-based detection algorithm in combination with the implementation of robotic arm manipulation using MoveIt. The system successfully integrated and combined image processing, the design of an improved gripper, chess engine integration, and movement control techniques to enable autonomous chess gameplay. To prove its effectiveness, a performative and comparative analysis was done on the system using measurements like accuracy, speed, and success rate of manipulation and human move detection. In the end, the proposed system showcased a slightly higher success rate in both the accuracy and human move detection than the two other autonomous chess systems that were evaluated. To summarize:

- To identify and make the detection algorithm reliable, a camera calibration process was done using a ChArUco board, which contains fiducial markers in the form of a chessboard. The calibration sequence ensured that the intrinsic and extrinsic parameters of the camera used are retrieved which were necessary for an accurate pose detection of the fiducial markers used. These fiducial markers are placed strategically on each of the four corners of the chessboard, on a table next to the chessboard, and on the arm. The fiducial markers on the chessboard served for the detection algorithm to be able to correctly warp the perspective of the camera and to create the grid that was needed for imposing the coordinates required to tell the robot arm how to move. The marker on the table next to the chessboard served as the identifier for

the captured pieces to be placed, and the marker on the arm served as a crucial link between the camera coordinate frame and the robot arm's coordinate frame.

- A new gripper was designed to further enhance the grasping area of the robot's gripper to ensure a more reliable manipulation of the chess pieces.
- A generic movement interface was created for the Franka Emika Panda to enable the movement of the arm. The decision was made to create a generic interface so that the system could be reused in other applications than a chess robot. Specific chess-related moves were then implemented on top of this interface to accommodate the specific complex movements needed for playing chess.
- Stockfish, a chess engine, was utilized to check the legality of the human-made moves and also to generate moves that the chess robot would play in response.
- To ensure the proper translation between coordinates captured by the camera and the movement of the arm, the *tf2* library from ROS2 was used.
- Lastly, experiments were done to prove the effectiveness of the system by measuring its accuracy, speed, and detection rate over numerous tests. The results of these tests were then compared to two other autonomous chess robots.

6.1 Future work

Although this implementation of an autonomous chess robot showcases great promise, there are still several avenues for further exploration and improvement. Firstly, different methods for the chessboard and piece recognition can be considered, like using OptiTrack, UWB, or even machine-learning-based approaches. These technologies may provide more accurate and robust detection in certain scenarios.

Additionally, in this thesis, the exploration of simultaneous chess was not implemented, where multiple chess games are played at the same time. With the robot arm

situated on an unmanned ground vehicle, it is possible to dive deeper into the possibility of adding multiple chessboards, all with different games, over a multitude of different tables, and having the system be able to remember and play all the chess games at once.

Furthermore, the improvement of dynamic detection is something worthwhile to look at. Currently, the system relies on a static calibration at the start of the program and assumes this position for all other moves. Improving the dynamic detection may allow for the capability of real-time changes in the chessboard or camera frame without disturbing the core functionality of the system. This approach was considered at first for this thesis, but due to the ambiguity issues of the fiducial markers, it was deemed too unreliable for this implementation.

Then, due to safety issues and the unreliability of the arm, the maximum velocity and acceleration were only set to 20% of the maximum value of the arm. This resulted in an average of 24.04 to 47.96 seconds per move depending on the type of move. With the improvement of the reliability of the arm, it could be considered to make the robot arm move faster and thus, substantially reduce the time required per move.

Lastly, as mentioned many times in this thesis, the robot arm is situated on an unmanned ground vehicle. With the creation of a generic movement interface for the robot arm, further exploration could be done into mobile manipulators in other fields.

References

[1] C. S. Agency, *About canadarm*. [Online]. Available: https://www.asc-csa.gc.ca/eng/canadarm/about.asp (visited on 03/08/2023).

[2] E. Tunstel, M. Maimone, A. Trebi-Ollennu, J. Yen, R. Petras, and R. Willson, "Mars exploration rover mobility and robotic arm operational performance", in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, Oct. 2005, 1807–1814 Vol. 2. DOI: 10.1109/ICSMC.2005.1571410.

[3] "Davinci surgical systems". (), [Online]. Available: https://www.intuitive.com/en-us/products-and-services/da-vinci/systems (visited on 03/08/2023).

[4] Hiba Sekkat, Smail Tigani, Rachid Saadane, Abdellah Chehri, "Vision-based robotic arm control algorithm using deep reinforcement learning for autonomous objects grasping", 2021.

[5] Ian Lenz, Honglak Lee, Ashutosh Saxena, "Deep learning for detecting robotic grasps", 2015.

[6] G. Du, K. Wang, S. Lian, and K. Zhao, "Vision-based robotic grasping from object localization, object pose estimation to grasp estimation for parallel grippers: A review", *Artificial Intelligence Review*, vol. 54, no. 3, pp. 1677–1734, 2021.

[7] M. Intisar, M. M. Khan, M. R. Islam, and M. Masud, “Computer vision based robotic arm controlled using interactive gui.”, *Intelligent Automation & Soft Computing*, vol. 27, no. 2, 2021.

[8] G. Ranganathan, “An economical robotic armplaying chess using visual servoing”, *Journal of Innovative Image Processing (JIIP)*, vol. 2, no. 03, pp. 141–146, 2020.

[9] C. del Toro, C. Robles-Algarın, and O. Rodrıguez-Álvarez, “Design and construction of a cost-effective didactic robotic arm for playing chess, using an artificial vision system”, *Electronics*, vol. 8, no. 10, p. 1154, 2019.

[10] C. Matuszek, B. Mayton, R. Aimi, *et al.*, “Gambit: An autonomous chess-playing robotic system”, in *2011 IEEE International Conference on Robotics and Automation*, IEEE, 2011, pp. 4291–4297.

[11] G. O. Larregay, F. L. Pinna Gonzalez, L. O. Avila, and O. D. Moran, “Design and implementation of a computer vision system for an autonomous chess-playing robot”, 2018.

[12] D. Urting and Y. Berbers, “Marineblue: A low-cost chess robot.”, in *Robotics and Applications*, 2003, pp. 76–81.

[13] “Configuring environment”. (), [Online]. Available: `https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html` (visited on 01/24/2023).

[14] “What is ros?” (), [Online]. Available: `https://ubuntu.com/robotics/what-is-ros` (visited on 01/26/2023).

[15] “Understanding nodes”. (), [Online]. Available: `bit.ly/3XUyO8C` (visited on 01/27/2023).

[16] “About tf2”. (), [Online]. Available: `https://docs.ros.org/en/foxy/Concepts/About-Tf2.html` (visited on 03/15/2023).

[17] “Moveit concepts”. (2023), [Online]. Available: https://moveit.ros.org/documentation/concepts (visited on 01/26/2023).

[18] “Franka emika panda robot”. (), [Online]. Available: https://www.franka.de/research#tool-set (visited on 03/08/2023).

[19] “Overview - franka control interface (fci) documentation”. (), [Online]. Available: https://frankaemika.github.io/docs/overview.html (visited on 02/01/2023).

[20] “Ros_control”. (), [Online]. Available: https://wiki.ros.org/ros_control (visited on 03/08/2023).

[21] “Disabling cpu frequency scaling”. (), [Online]. Available: bit.ly/3JZoupX (visited on 03/15/2023).

[22] “Tutorial: Aruco detection”. (), [Online]. Available: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html (visited on 02/22/2023).

[23] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marın-Jiménez, “Automatic generation and detection of highly reliable fiducial markers under occlusion”, *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[24] C. Team. “Computer chess engines: A quick guide”. (), [Online]. Available: https://www.chess.com/article/view/computer-chess-engines (visited on 03/16/2023).

[25] “Stockfish chess engine”. (), [Online]. Available: https://stockfishchess.org/ (visited on 03/16/2023).

[26] “Algebraic notation in chess”. (), [Online]. Available: bit.ly/46QVVVq (visited on 01/06/2023).

[27] “Moveit’s movegroupinterface”. (), [Online]. Available: bit.ly/3rpPKaK (visited on 02/03/2023).

[28] "Franka hand manual". (), [Online]. Available: https://download.franka.de/documents/220010_Product%5C%20Manual_Franka%5C%20Hand_1.2_EN.pdf (visited on 06/12/2023).

[29] "Tutorial - aruco calibration". (), [Online]. Available: https://docs.opencv.org/4.x/da/d13/tutorial_aruco_calibration.html (visited on 06/12/2023).

[30] "Aruco ambiguity issue". (), [Online]. Available: https://github.com/opencv/opencv/issues/8813 (visited on 04/13/2023).

[31] "Chess demo". (), [Online]. Available: https://youtu.be/cG6JGErOyXs.

[32] "Troubleshooting - franka control interface". (2017), [Online]. Available: bit.ly/3pMbzke.

[33] S. Sarker, "Wizard chess: An autonomous chess playing robot", in *2015 IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE)*, 2015, pp. 475–478. DOI: 10.1109/WIECON-ECE.2015.7443971.

[34] C. Matuszek, B. Mayton, R. Aimi, *et al.*, "Gambit: An autonomous chess-playing robotic system", in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 4291–4297. DOI: 10.1109/ICRA.2011.5980528.