Cyber-Physical Systems
TUM School of Computation, Information and Technology
Technical University of Munich

# ConnecTUM

Development of a cyber-physical system for playing Connect 4

**Georgi Baldzhiev, Attila Berczik, Julien-Alexandre Bertin Klein, Kaviru Kithmal Gunaratne, Andrea Pellegrin, Alexandros Stathakopoulos, and Florian Stupp**

September 1, 2025

**Abstract** — Connect 4 is a popular game including two human players. This can be a problem if one wants to play without having another person, or if a specific difficulty level is desired. This report introduces the development of a Connect 4 playing robot, which can be an opponent for human players. This allows humans to practice their skills. The system includes mechanical design for the construction, computer vision to detect the current game play, a game algorithm to perform the optimal moves and a control system to ensure integration between the components. The project successfully exhibits fully autonomous play with reliable disc movements, which lead to an adjustable win rate of up to 100%.

## 1 Introduction

Connect 4 is a game which is widespread and well-known, not only in terms of popularity among players but also from a computational standpoint regarding solvability. Thus, many online versions are offered, allowing a user to play against a computer. However, practicing one's own Connect 4 skills with a physical game is barely offered. The goal of our project is to develop a cyber-physical system which addresses this gap. To this end, we develop a system capable of playing Connect 4 in the physical environment against a human player. The requirements are therefore: detection of the current game state, the ability to decide on the next move and the ability to physically perform this move. All of these subproblems are solved using a cyber-physical system. This project includes multiple aspects such as: its mechanical construction, the electronic components, the computer vision to detect the current state, as well as the algorithm and the software for the game-play. The system incorporates different knowledge from the lecture on cyber-physical systems and applies it to this real-world project.

This report is structured as follows:

- **Section 2** outlines the overview of the system
- **Section 3** describes the mechanical design
- **Section 4** describes the electronics and control system
- **Section 5** introduces the computer vision aspects
- **Section 6** talks about the interactions between files and the UI developed for the game
- **Section 7** presents the game algorithm
- **Section 8** gives an evaluation of the results discussed in **Section 9**
- **Section 9** provides conclusions

## 2 System overview

During our projects, we developped two successive version of the game. The Version 1 (or v1) served as a prototype to test the software and desing the different 3D pieces. The Version 2 (or v2) aims to be a *plug-and-play* version, closer to a finish product.

Version 1:
The core of our automated game system is a Raspberry Pi, which serves as the central processing unit. It executes the game logic—implemented in Python—and performs real-time computer vision on the video feed from an overhead camera. Movement commands are transmitted over a USB serial link to an ESP32 microcontroller running firmware in C. The ESP32 interprets these commands to drive a stepper motor and two servo motors, enabling precise chip dropping on the game board. The version is complete and fully playable. At the date of the publication, it runs the code available under the release "Version 1 (presentation)" on GitHub. It is also the version showed during the final presentation.

Version 2:
This version is more compact and integrates a battery and a touchscreen, making it playable anywhere and without the need of any external support. The code for the game and computer vision logic is reused from the v1. The ESP32 is replaced by the RaspberryPi and some additional drivers. Transmission of the movement commands to the motors are therefore more direct, the logic being entirely implemented in a Python file on the RaspberryPi. At the time of publication, Version 2 is not playable do to complication in the new motor integration but we are planning on making it available in the coming weeks.

## 3 Mechanical design

This section describes the mechanical components, their design and integration in the system. The hardware is composed of multiple parts, each fulfilling a specific function.
We decided to employ 3D-printing and laser-cutting for the manufacturing of the different parts. The motivation here is to quickly produce a first working prototype. In addition, we aimed for fast iterations over different versions such that alterations could be made easily. Following this strategy, we developed the different critical mechanical parts individually (each going over multiple versions) to account precisely for the requirements of each of them. In addition, we ensured the seamless integration between the parts and other externally sourced components such as servo-motors, stepper motors, chips, etc. Here, it was crucial to consider the required clearances for the different parts and account for them. Since the tolerance of the manufacturing technique also needs to be regarded, the first versions are used to ensure the correct fitting between the parts and the suitability of using 3D-printing and laser-cutting as our way of production.

### 3.1 Base movement

The system needs to be able to insert chips in the correct column at each point during the game; a mechanism for the reachability of all of the different columns needs to be developed. The columns have a uniform width and height forming the game board, as seen in *Figure 1*. This implies that only movement along the axis parallel to the board (as shown) is sufficient.

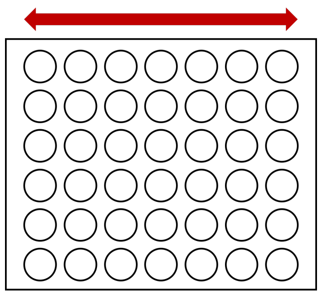


**Figure 1** Direction of movement of base

The tolerance between the diameter of the chips and the size of the column is minimal but sufficient to ensure smooth and predictable operation. Therefore, high precision is required to allow the correct placement of chips. To ensure this precision, we use a T8x8(P2) lead screw and a linear bearing on a shaft to precisely and effectively control the assembly responsible for dropping the chips (described in *subsection 3.2*). The motivation for this design decision is the widespread use of this standard mechanism in CNC and AM.
The hardware used for this axis is among the heaviest elements of the design and therefore has to be mounted to the base preferably low and close to the center of support. The difference in height (shaft ↔ top of the board) is compensated for by the chip dropping assembly base as it is lighter (*see Figure 2*).

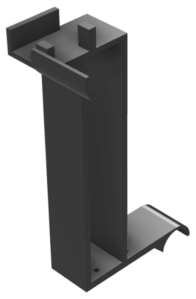


**Figure 2** Design of the base of the dropping mechanism

The base of the dropping mechanism is connected with the leadscrew nut via 4 screws accommodated in its design. This ensures a strong yet compact connection.

## 3.2 Chip insertion

This assembly consists of a base with a mount for a servo on top, which in turn holds the chip to be inserted in the board. To insert the chip in the correct column, it is moved along the linear axis. After positioning, a mechanism is needed to actually place a chip in the board. We decided that this mechanism should only hold one chip at a time and restock it at a magazine after each move. The reason being that moving the entire magazine would have increased the height, weight and reduced the stability of the assembly (note: the magazine must store 21 chips to provide chips for the maximum amount of possible moves).
For the chip insertion mechanism, the following requirements can be formulated:

1. The chip needs to be held securely while the assembly is moving to prevent it from falling out.
2. The mechanism needs to be able to insert the chip in the column with high precision.
3. The construction needs to be able to hold one chip and to be refilled at a magazine.

To account for all of these requirements and still have a compact design, we decided to construct a minimalistic holder for the chip, which is rotated by a servo motor to insert the chip in the column of the game board (*see Figure 3*)

It is mounted onto the arm of a servo motor, which can then turn it by 90 degrees around a specific pivot

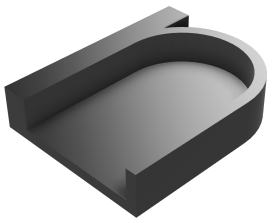


**Figure 3** Holder for the chip

point, such that the chip can fall into the board at the correct position. The walls at the side of the holder ensure that it can not move to one side and so lose its position. *Figure 4* shows the insertion of a chip in the game board.

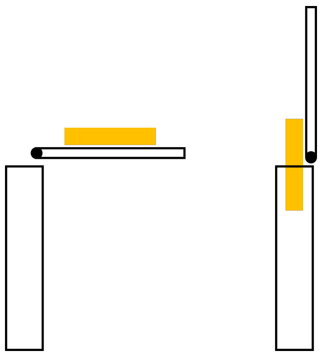


**Figure 4** Process of inserting a chip

The yellow rectangle illustrates here the chip that is supposed to be inserted. While moving the assembly to the correct column, the holder is parallel to the ground (as shown on the left side of the figure). If it reached the correct position, it is rotated around the pivot point (illustrated by the black dot), placing the chip into the column.

## 3.3 Magazine

We described in the *subsection 3.2* the need to use a magazine to store additional chips to ensure a reduced height of the moving part and so increase the stability. The magazine needs to meet the following requirements:

1. Hold 21 chips.
2. Place a chip from the magazine in the chip holder automatically.
3. Ensure that the mechanism is working reliably and the chips do not get stuck.

We worked on different ideas for the magazine. One idea involved a wheel keeping four chips at a time by having four holds for the chips and then dropping them at one position. With the different constructions, we encountered problems with regard to reliability and also with the size. This led to the decision to build a magazine which stacks the chips on top of each other and always pushes the chip at the bottom into the chip holder. The advantage of this construction is that it uses gravity for arranging the chip stack. No springs or other forces are needed to arrange the stack. The chip at the bottom is pushed out of the magazine by a servo, which has an arm. This arm is a bit thinner than a chip, and so pushes the chip out of the stack. *Figure 5* shows the construction. It consists of a tube storing all the chips stacked on top of each other. The ring on the bottom ensures that the chips, which are pushed out by the servo motor, do not move too far but instead fall into the chip holder. The chip holder is moved under the opening of the magazine where the chip is falling out for restocking. The mount on the side holds the servo motor using screws to ensure stability.

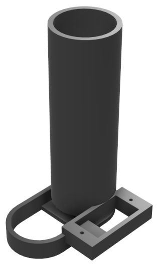


**Figure 5** Construction of the magazine

## 3.4 Mounting and assembly

The components described above, together with additional components such as the game board and a base plate, need to be mounted together. *Figure 6* shows how the different components are positioned relative to each other. This positioning ensures that the dropping mechanism reaches all columns precisely and the magazine is reachable for restocking.

To ensure this positioning, an additional construction is needed to hold the components in their position. For this construction, we decided to use a wooden base plate, allowing flexibility in the actual positioning of the components and also a simple rearrangement.

The lead screw and the linear carriage are mounted to the base plate, ensuring stability and fixing their relative position to each other to ensure a perfect fit.

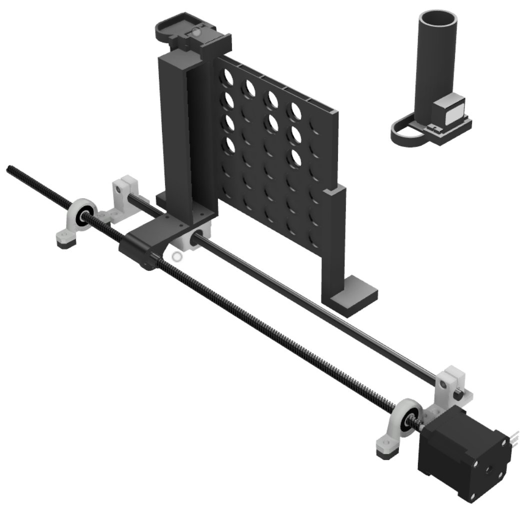


**Figure 6** Rendering of the assembly of the different components

The magazine is mounted on a wooden holder. The holder positions the magazine at the correct height, such that it can restock the chip holder. The game board is also mounted to the ground. The wooden base construction allows to make adjustments in the positions of the components relative to each other. This especially allows a change in the positioning of the game board to the base movement construction. This relation is crucial because it ensures the chip can fall in the correct column.

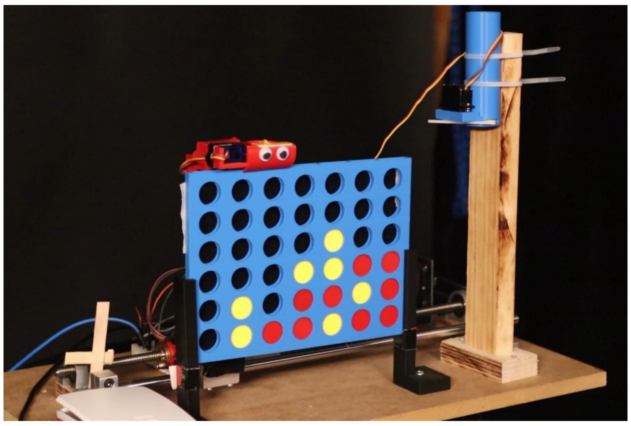


**Figure 7** Image of the completed v1. We can see the RaspberryPi in the white box in the bottom left corner. Behind, we notice the lead screw and the breadboard for the electrical.

## 3.5 Improved construction in version 2

Despite being functional, the first version is not easy to transport and setup. It also offers limited user interactions as it requires an external monitor or an ssh connection to start the game. The objective of the version 2, is to create a more portable version, similar to a real game that can be easily used by anyone.

Therefore the overall design remains the same but was adapted to new requirements such as a screen for the UI, and new components like a battery or a new camera.

One of the requirements was to fully encapsulate all the components inside of a main body. This body is composed of three parts: a bottom box containing most of the electrical components as well as the motor and that integrates the screen in its cover; The main panel with the board and two columns on the left and right that slides in the bottom box and hides the mechanical parts; The top cover holding the magazines.

Furthermore we also added a few modifications to make the game even more compact and sturdy. For example, the shaft and the lead screw for the based movement of the dropping mechanism are now located behind the board and placed on the top of each other, fermly anchered to the left/right columns. The belt connecting the lead screw to the motor is now placed verticaly in the left column. Overall, this construction makes the dropping mechanism significantly smaller and more stable. The unique magazine was also replaced by two shorter version place on each side. Both of them integrates a Time-of-Flight sensor to detect the presence/absence of coins in the magazine. Having two magazines not only contribute to reducing the height of the game but also create new gameplay possibilities. The bot could drop both red and yellow coins allowing for remote control or one magazine can be loaded with special coins offering variations in the gameplay. Additionally, the board is equipped with a new release mechanism for the coins. By pulling on the red stick, the coins will drop at the bottom of the main panel in a drawer that can be removed from the body to easily collect the coins.

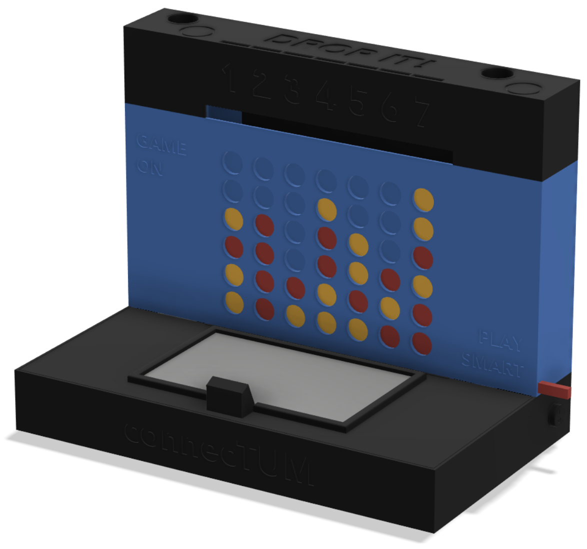


**Figure 8** 3D rendering of Version 2.

# 4 Electrical and control system

This section describes the different electrical and control systems, including the servo motors and the stepper motors. The system includes multiple components which all need to be integrated to ensure a working environment to fulfil each individual function. Therefore, the architecture is discussed.

## 4.1 Architecture

While planning the project, we formulated the following requirements for the control unit:

1. sufficient computational power for the optical recognition
2. support (maintained libraries) for the necessary components (stepper driver, servo driver, camera)
3. sufficient electrical connection options for the accompanying components

We concluded that a Raspberry Pi or a similar SBC is the only device to handle our complex workload. For several reasons regarding the control of our servo motors, explained in the following section, we decided to introduce an additional ESP32 micro controller to improve the reliability of our system.

## 4.2 Control of servo-motors

The system relies on two high-precision (Tower Pro Micro Servo 9g) servos to perform distinct but tightly interwoven tasks: one feeds coins into the dropping arm, and the other actuates the release mechanism. Because the installation space is extremely confined, we required sub-millimeter repeatability in both axes of motion. Our first approach—driving the servos directly from the Raspberry Pi's GPIO—introduced unacceptable jitter. We then experimented with an off-the-shelf PWM-servo HAT, but encountered power-supply limitations and sparse documentation. Ultimately, we migrated servo control to an ESP32, eliminating jitter and simplifying power management. On the ESP32, we run a lightweight C firmware that listens for serial commands. The Raspberry Pi remains responsible for high-level logic and communicates with the ESP32 over USB, effectively marrying Python's flexibility with the ESP32's real-time performance.

## 4.3 Control of stepper motor

To ensure the reliable operation of the coin-dropping mechanism, the coin must be deposited with very high positional accuracy so that the chip seats correctly in the board. Achieving this level of precision also requires careful control of the loading force. To meet these requirements, we selected a NEMA 17 bipolar stepper motor to drive the horizontal translation of the drop-arm assembly across the seven columns. NEMA 17 stepper motors are renowned for their fine positional resolution and are widely employed in precision applications such as 3D printers and CNC mills.

All motor control is managed by an ESP32, which receives simple serial commands from the game logic. We encode each column as an integer from 0 to 6, and use 7 to designate the loading bay. To guard against invalid inputs, any number outside the 0–7 range is reduced modulo 7, ensuring the arm always remains within the playable grid. Additionally, two special commands trigger the aforementioned servo actions directly: "8" initiates chip loading, and "9" initiates chip dropping.

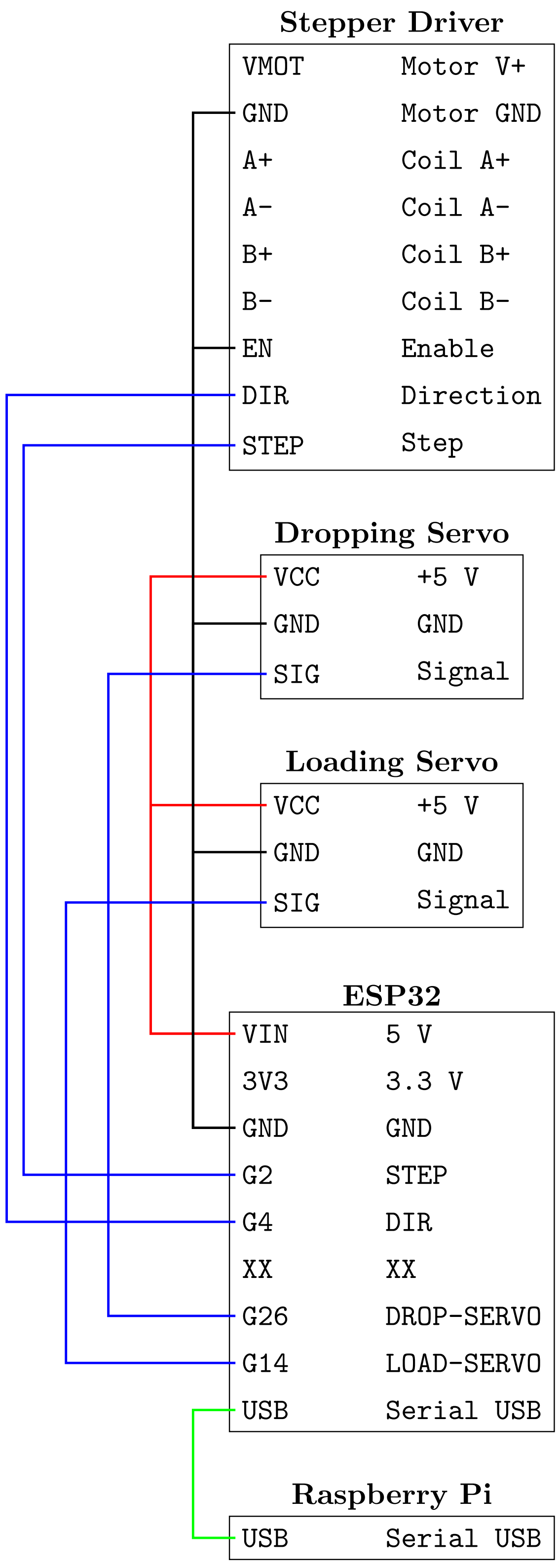


**Figure 9** Wiring diagram.

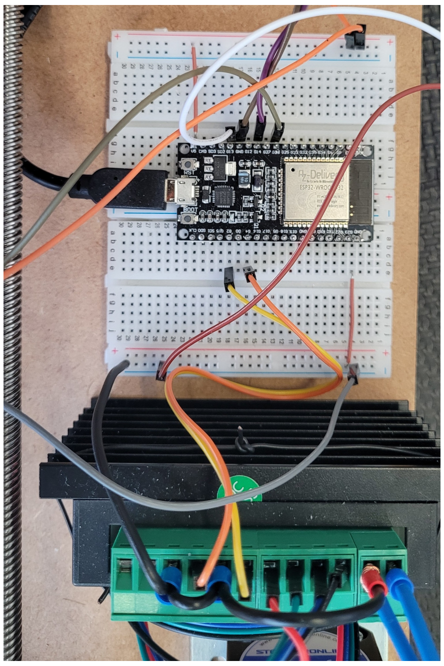


**Figure 10** Image of the wiring from above. On top, the ESP32 and on the bottom the stepper driver.

### 4.4 Changes in the electrical and control system in version 2

One of the main change in the electrical for version 2 is the addition of a 12V battery to be able to play "on the go". To this end, we added a BMS (Battery Management System) to protect the battery as well as buck converter to convert the 12V needed for the stepper motor to the 5V needed for the other components. Finally we also added a PD decoy to charge the battery (or directly provide power) via an external usb-c power supply.

For the control system, we decided to remove the ESP32 and use the RaspberryPi with a PWM driver instead. The driver is powered in 5V by the buck converter and connected to the RaspberryPi and the respective servos and ToF sensors via $I^2C$. Additionally, an I/O Expander is used for the switch logic and is also connected to the Pi via $I^2C$.

The motor was replaced by a less powerful but sufficient and quieter version of the NEMA17. Because the motor is placed in en enclosed environment, it's cooled down by a small fan located next to it.

The biggest addition of the v2, the 7inch touch display is directly connected to the RaspberryPi over HDMI for the video output and over USB for power and touch control.

## 5 Computer vision

To detect the coins and retrieve the moves of the player we used a computer vision based system instead of sensors. This ensure a high level of reliability and prevent cheating moves from the player.

### 5.1 Computer vision setup

The computer vision setup is composed of a camera and a calibration zone on the game board. The image processing is handle by the OpenCV python library. Additionally, we have created a GUI dashboard to efficiently setup and debug the computer vision process. For the camera, we are using a Pi Camera Module 2 (and it's dedicated library [1]).

The calibration zone consists of 2x3 strips of red, yellow and white. We utilize the fact that the calibration zones are fixed on the game board to hard code their position in the program. Despite the camera being fixed, it is completely possible to move it during the setup and the only requirement is to align the physical game board with the guideline provided in the GUI dashboard.

The computer vision process must be run with a camera profile located in the config folder. Each profile is a yaml[2] file containing information about the camera used as well as the mode and options apply during the image processing. These options can be enabled or disabled in the camera profile (or tested in the GUI) to optimize detection performances for different environments and lighting conditions. The path to the camera profile is given as an argument when launching `main.py` or `camera.py`.

### 5.2 Options for the image processing

The color detection mode indicates how the coin's color are detected. It can be set to either `FIX_RANGE` or `DYNAMIC_RANGE`. `FIX_RANGE` (used in v1) relies on predefined HSV color ranges for coin detection, while `DYNAMIC_RANGE` (introduced in v2) automatically computes color ranges from the calibration zone in each frame, making detection more robust to changing lighting conditions. If `FIX_RANGE` is used, two different pairs of color ranges must be defined, one for red and one for yellow. The suffix `_L` and `_U` refers respectively to the lower and upper bound range. The `RED_NOISE_[L|U]` range must be defined if the `RED_NOISE_REDUCTION` is

[1]Pi Camera 2 library documentation
[2]https://en.wikipedia.org/wiki/YAML

enabled, regardless of the color detection mode.

*Note on the HSV format:*
The HSV format (Hue, Saturation, Value) is a color space commonly used in image processing. Hue represents the color type, saturation indicates the intensity of the color, and value corresponds to brightness. HSV is preferred over RGB for color detection because it separates color information from lighting, making it easier to distinguish objects under varying illumination.

The camera options available for pre-processing are:

- **WHITE_BALANCE**: Adjusts the image to correct color balance, compensating for lighting that may cause color shifts.
- **GRAY_WORLD**: Applies gray world normalization, which assumes the average color in the image should be gray, helping to correct color casts.
- **GLOBAL_NORMALIZATION**: Normalizes the image globally, ensuring uniform brightness and contrast across the frame. This is done by matching the histogram of the frame to the histogram of a reference image, so the colors appear more consistent and closer to a golden standard.
- **BLUR**: Applies blurring to the image, which helps reduce high-frequency noise and smooths out small artifacts.
- **RED_NOISE_REDUCTION**: Reduces noise specifically in the red channel by removing the pixels within the range `RED_NOISE_[L|U]` provided in the camera profile. This can improves the detection of red coins in challenging conditions.

All of this options can be tested from the GUI dashboard, although the options enable graphically are not saved in a camera profile.

## 5.3 Computer vision pipeline

In this section, the *grid* refers to intermediate arrays used to compute the *game board*, code object that is equivalent to the real physical game board. The pipeline to analyze each frame and subsequently compute the game board is the following:

1. **Retrieve the input image from the camera and convert it to BGR:**
   OpenCV expects images in the BGR format but our Pi Camera provides us with a classical RGB image. Without converting our input image, OpenCV will interpret the RGB image as BGR and therefore swap the blue and red channel. However, we realized that it is easier to detect blue rather than red (because red is situated at the two ends of the hue spectrum and so requires two different color ranges to be detected as the two ends) so we actually don't convert our input image and do work with the two channels swapped.
2. **Apply selected camera options:**
   This step correspond to the pre-processing of the image, i.e. enhancing the input image to achieve a better coin detection. In our case, this can be boosting the contrast or compensating exposition of a bad lighting for example. Each camera option (detailed above) add a layer to the pre-processing and are apply in the order they appear above.
3. **Compute the dynamic color range:**
   One of the main problem of the `v1`, was that the computer vision pipeline was very dependent to the light conditions. Each major change (between warm and cold light for example) required to reconfigure the color detection parameters manually. To compensate for this, we added this step to dynamically compute these parameters in the v2. For both red and yellow, the program will look at a fixed area in the input image which correspond to our calibration zone on the physical game board. It will then compute the mean of the HSV values for this area and create a color range accordingly by adding a predefined margin to the computed values. The process is similar for applying the white balance option in the pre-processing step.
4. **Create the masks to isolated the pixels of the coins colors:**
   To decide whether a pixel is red or yellow, we use the previously computed HSV ranges for each color. This is done by calling `cv.inRange()` which returns a binary mask (an image with only black or white pixels) indicating for every pixel if its HSV values are within the given range (i.e. of the color we want to detect). If so, the pixel will be white *(logic true)* otherwise black *(logic false)*. After this, we use `cv.morphologyEx()` and `cv.dilate()` to fill any small holes that could

be present in the masks to avoid false negatives in the circle detection step.

5. **Detect the circular shape in the masks and map them to a position in the grid:**
   The detection of the circle is performed using `cv.HoughCircles()` [3], which returns the coordinates and radius of all detected circles in the mask. Each detected circle is then mapped to a grid position (only if the circle is within the game board) by comparing its center coordinates to the position and size of the top-left most and bottom-right most circles. After detection and mapping, the coin's color value is assigned to the corresponding cell in the grid. By accumulating these values across multiple frames, the system computes an averaged game grid, enhancing accuracy and providing a more reliable representation of the current game board. The final result is a 2D array reflecting the current state of the board, which serves as input for the game logic.

The computer vision pipeline is illustrated in Figure 11.

## 5.4 Accuracy and error handling

To improve the robustness of the coin detection and minimize errors due to noise, occlusions or vibrations we implemented a grid accumulator approach in the `grid_accumulator()` method. Instead of relying on a single frame to determine the board state, the program processes multiple consecutive frames and keeps an ongoing count for each cell in the grid. For every frame, detected coins are mapped to their respective grid positions, values in the grid are incremented depending on the coin (-1 for red, 1 for yellow, 0 otherwise). After a predefined number of frame (`max_frame`), the program determines if a coin appears reliably or not by checking for every cells in the grid if its value is higher than the required `success_rate` ($|value| \geq max\_frame \times sucess\_rate$). The sign of the value gives us the color of the coin (negative is red, positive is yellow). Thanks to this method we avoid most of the false positives and unwanted behaviours like flickering coins or coins alternatively detected as red and yellow (-1 and 1 cancels out). However this approach supposed to have access to pure and consistent data from the preceding steps of the pipeline. If the data is not consistent, the grid accumulator will not be able to determine the correct

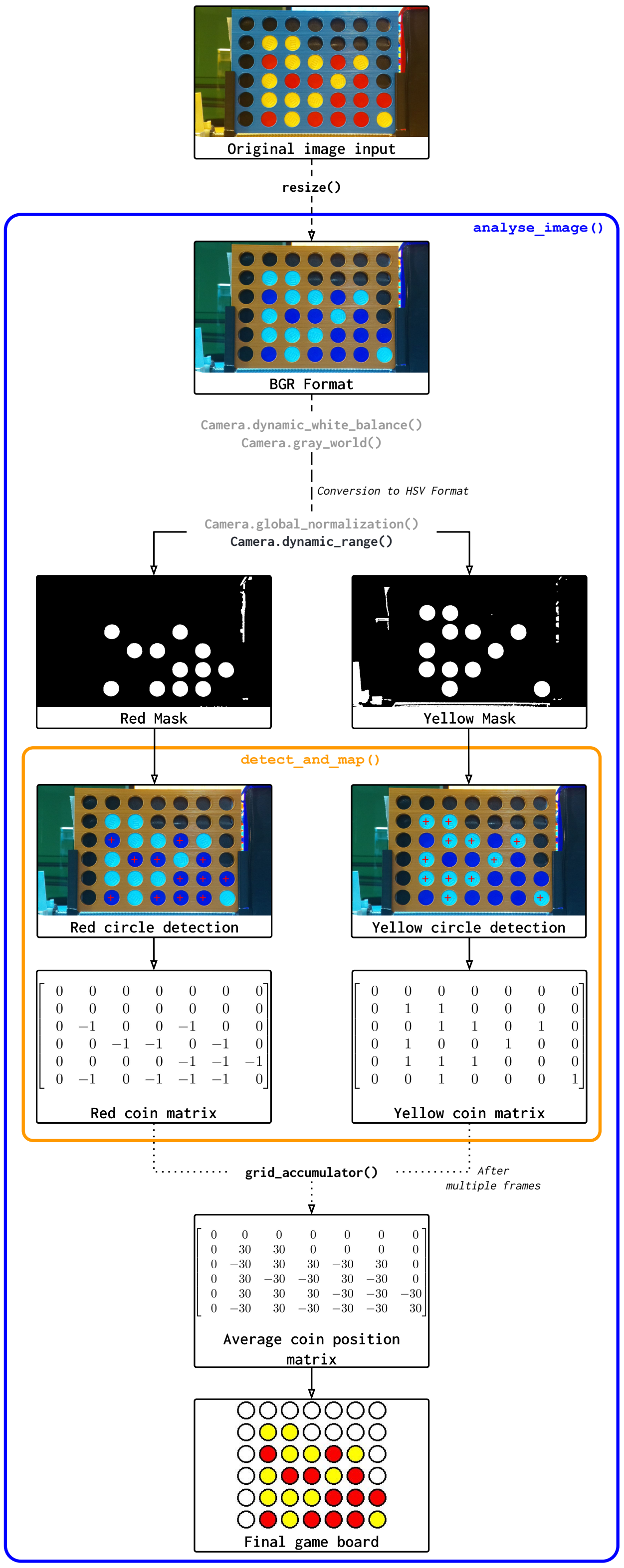


**Figure 11** Pipeline for processing one frame with OpenCV.

[3]cv.HoughCircles() documentation

status of the game board and leads to false negatives in the computed game board.

In addition to this, the main game loop is implemented as such than a new game board can only differs from the one previously stored in only one place. Therefore if the camera cannot detect the physical game board for any reason because it has been moved or completely occulted, the game logic will not be updated and remains stuck in the main loop until it receives a valid game board.

All of this mechanisms not only ensure a high accuracy and reliability of the computer vision system but also prevent the game logic from being updated with faulty data, that could furthermore lead to unexpected behaviours and create a snowball effect in the main loop.

## 6 File Hierarchy and User Interface

### 6.1 File Hierarchy

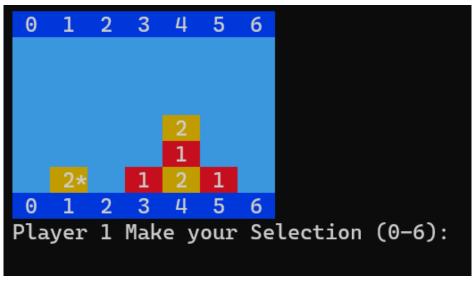
To keep the project organize, the code is logically splitted between different files and folders. The two main files are `main.py` and `camera.py`. The first one is responsible for the game logic and the Connect4 algorithm while the second one handles the camera and the computer vision pipeline.

Both files work and can be launch independently. If a game requires the camera, the main file will start a new process to run it. The infomation needed, like the game grid status read from the camera are accessed through a *shared directory* created by the main file and passed to the camera file at its creation.

In the v2, the api file starts a game on the same principal, it creates a new process for the main file and pass it a *shared directory*. From there the main file behaves exactly like in the v1.

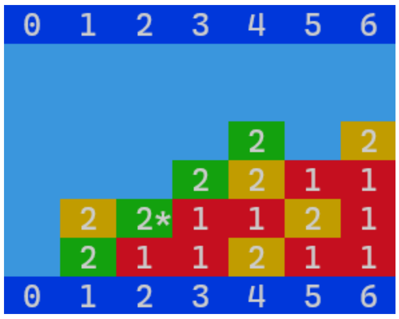
### 6.2 Command Line Interface

`main.py` also serves as a single entry point to start and debug the game. Version 1 doesn't have a proper UI but includes a small terminal interface created as feedback for improving the Connect4 algorithm. By default, the program starts the camera, connects to the motors and prints the grid in the terminal at each step. It is also possible to use some options to disable those features. This can become very handy when debugging or when a part of the setup is not available. The flag `-t` (shorthand for `-no-camera -no-motors`) allows the user to play using only the terminal by inputting a number corresponding to the column where they want to play. Printing the grid in the terminal, also indicate the last coin played (with $\star$ next to the number) and shows the wining combinaison in green at the end of game. With the option `-l` (`-level`), it is possible to select the difficulty for the game between easy, medium, hard and impossible (the default). The flag `-b` or `-bot-first` makes the bot plays the first move instead of the player.



```
 0  1  2  3  4  5  6
                     
                     
                     
             2       
             1       
    2*    1  2  1    
 0  1  2  3  4  5  6
Player 1 Make your Selection (0-6):
```

**Figure 12** Grid and its promt for next move printed in the terminal.



```
 0  1  2  3  4  5  6
                     
                     
             2     2 
          2  2  1  1 
    2  2* 1  1  2  1 
    2  1  1  2  1  1 
 0  1  2  3  4  5  6
```

**Figure 13** Grid printed in the terminal at the end the game.

### 6.3 Graphical User Interface

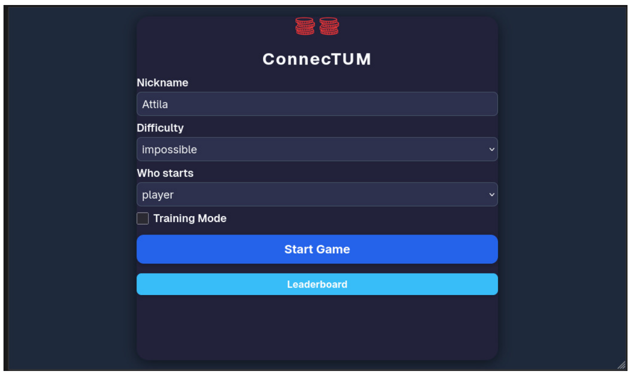
For Version 2, a User Interface was developed to improve the user experience. The main challenge we faced was enabling the user to select their desired gameplay difficulty. Additionally, we wanted to incorporate gamification elements, such as score tracking based on game results, and an educational component to help users learn the game.

The UI was developed in React and runs in a headless browser on a touch display connected to the Raspberry Pi. The backend was also transformed into an API-based structure using FastAPI.

#### 6.3.1 Start screen

On the start screen, a nickname can be entered, which will be used on the leaderboard, along with the selected level and starting player. There are two magazine icons, linked to the two sensors in Version 2, which check whether the token magazines are empty. The icons turn red if the magazines are empty and green if they are full. The player cannot start a game if a magazine is empty.

A *training mode* is also available for learning the game. After making a move, the bottom of the game screen displays the scores from the "Impossible" mode gameplay, with the highest score representing the most optimal move. We hope this will help users learn the best moves in each situation.


ConnecTUM

Nickname: Attila

Difficulty: impossible

Who starts: player

Training Mode

**Figure 14** On the start screen, a nickname can be entered, the level selected, and the starting player chosen.

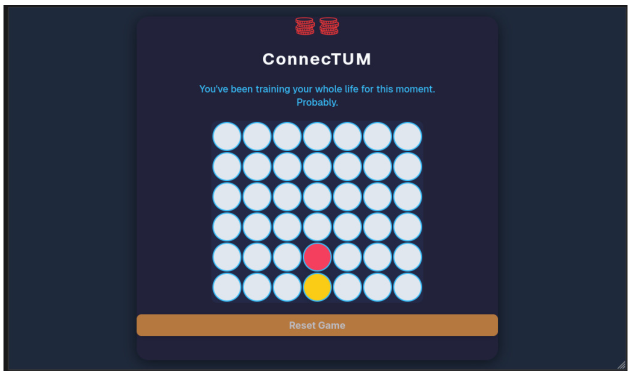
### 6.3.2 Game screen

For normal gameplay, we wanted a playful way to interact with the users. After each player move, the algorithm runs to evaluate the quality of the move. There are three categories: *bad*, *medium*, and *good*. For each category, there are ten humorous sentences, from which one is selected randomly as feedback. Once the bot moves back into position, this sentence disappears, and another random sentence is chosen from a different set to inform the player that it is their turn. On this screen, the gameplay can also be interrupted by the user, returning the application to the start screen.

### 6.3.3 Leaderboard

This screen can be accessed either from the start screen or after the player wins or loses. The system always saves the highest score for a player at each difficulty level. The score calculation rewards players for winning as quickly as possible or for delaying a loss as long as possible. The maximum score is twice the total number of spaces available for placing a token.

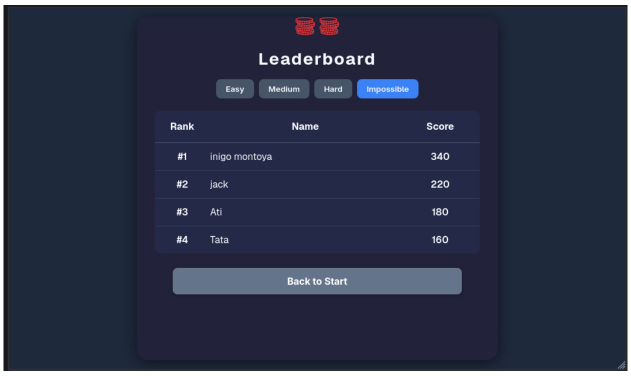
If the player wins, the score is calculated as the maximum score minus the number of tokens played,


ConnecTUM

You've been training your whole life for this moment. Probably.

**Figure 15** Playful messages are shown and the game can be ended.

meaning fewer moves result in a higher score. A draw results in a fixed score of 42, the total number of moves. If the player loses, the score equals the total number of tokens played. In the end, each score is multiplied by 10 to make it more game-like. For each leaderboard, the top ten current players are displayed, along with the current player's best score, regardless of their position.


Leaderboard

| Rank | Name | Score |
|---|---|---|
| #1 | inigo montoya | 340 |
| #2 | jack | 220 |
| #3 | Ati | 180 |
| #4 | Tata | 160 |

**Figure 16** User scores are tracked and displayed.

## 7 Game algorithm

The system, herein referred to as the bot, plays at four levels of difficulty: Easy, Medium, Hard, and Impossible. The first three levels are self-explanatory; the bot plays at increasing levels of competency. At the final difficulty level, the bot plays optimally, making it the strongest possible opponent. Only another player with perfect play can hope to match or beat it. This section details the algorithms utilized by the bot at each difficulty level.

## 7.1 Level Impossible: Optimal Play

Connect 4 has been mathematically solved, meaning that the outcome of a game from any given position can be determined mathematically assuming perfect play from both sides. In 1988, it was shown that the first player can always force a win with optimal strategy [4]. This would mean that a human can never defeat the bot at level "Impossible" if the bot starts first.

In order to model optimal play, we utilize the negamax algorithm along with several optimizations. First, we create a scoring function that assigns values based on the game outcome and the state of the board. Specifically, if the current player has won, the score is equal to the number of empty slots remaining on the board, rewarding faster wins. Conversely, if the player has lost, the score is the negative of the number of empty slots, penalizing faster losses. This approach encourages the algorithm to prefer winning sooner and delaying losing as long as possible.

Now, an optimal player would desire to maximum their score while an optimal opponent would attempt to maximize their own, which is the negative of ours. The optimal player should select the move which gives them the highest score, assuming that their opponent would play the next move such that this score is minimized. Thus, the algorithm proceeds recursively, parsing through the tree of possible board states all the way up to the end of the game. This is visualized in Figure 17.

To avoid reinventing the wheel, we utilize an existing implementation of the negamax algorithm applied to Connect 4 developed by Pascal Pons[5]. It must be noted that the implementation is not quite the algorithm highlighted above: vanilla negamax as described above requires us exhausting the entire search tree, and considering that there are over 4.5 possible board states, this is computationally intractable. Table 1 illustrates how the search space grows as we go down the search tree.

Therefore, several optimizations are utilized. First, a technique called alpha-beta pruning is used, which reduces the number of nodes evaluated in the game tree. Alpha-beta pruning takes advantage of the fact that evaluation along a branch can be terminated prematurely without affecting the final decision using knowledge acquired from previous paths. Figure 18 displays how this works. It must be noted that the branches that are discarded depends strongly on the order of the moves evaluated. This technique is most

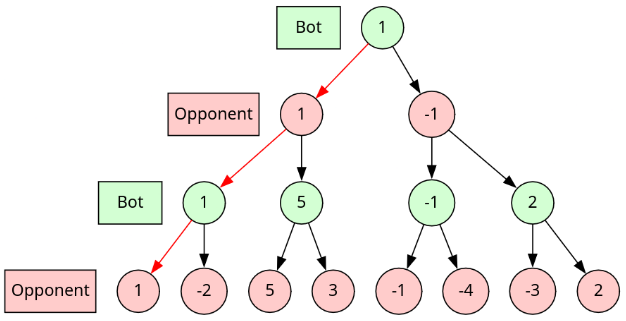


**Figure 17** The Search tree parsed by the minimax algorithm. Each node denotes a board state, and a path down the tree denotes a particular sequence of moves. The bot (green) and its opponent (red) play alternately, with the bot selecting the child that maximizes their score while the opponent selects the child that minimizes it. The numbers on the leaf nodes represent the final score obtained by the bot at the end of the game. The numbers of the parent nodes denote the score of the child node selected by the relevant player. The path with red edges highlights optimal gameplay. *Note: Generated with Graphviz.*

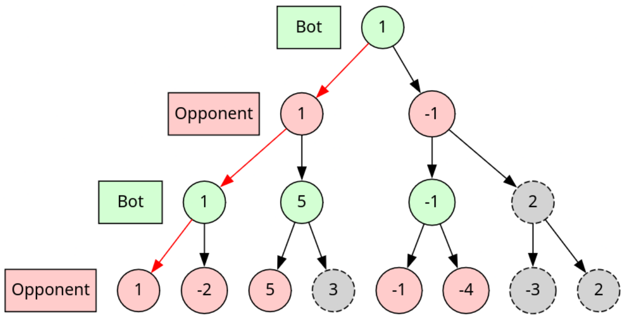


**Figure 18** Alpha-beta pruning on the game tree in Figure 17, assuming moves are evaluated from left to right. The greyed-out nodes are not evaluated. *Note: Generated with Graphviz.*

effective if the nodes are evaluated best choice first. There is of course no way to know this in advance since this is the very thing we are trying to find.

However, it is possible to use some heuristics from what we know of Connect 4. For example, playing on the center turns out to generally be more advantageous than playing in the corners. Therefore, by evaluating central moves first and corner moves last, we increase the probability of evaluating optimal moves earlier. By employing various such heuristics, we can significantly reduce the parsed tree.

Additionally, it is possible that the same board states get evaluated over and over again. To prevent recalculation, a 'cache' of board states and their scores is kept, with intelligent collision handling.

Even with this highly optimized implementation, it still takes a significant amount of time for the algorithm

[4] V. Allis (1988) *Knowledge-Based Approach of Connect-Four*
[5] https://github.com/PascalPons/connect4

to run in the first few moves as the search tree is still quite large. We solved this problem by pretraining the algorithm on the first few moves; we ran the algorithm on all the possible opening board configurations and stored them in a hash table. Therefore, in the first few moves, the bot can refer to this table to determine the optimal move. This allows the bot to evaluate optimal gameplay in real time.

## 7.2 Easy, Medium, and Hard

While the Impossible level uses a fully deterministic and optimized negamax search for perfect play, the Easy, Medium, and Hard levels utilize probabilistic approaches to simulate varying levels of competency. These three difficulty levels are implemented using Monte Carlo-based algorithms, with complexity and accuracy increasing progressively. Unlike with level Impossible, we developed these levels on our own.

### 7.2.1 Monte Carlo Simulation: Easy and Medium

At the Easy and Medium levels, we employ vanilla Monte Carlo simulations (hereafter MCS). In this approach, at each turn, the bot considers all possible valid moves, and for each one, it simulates a large number of complete games from that point forward. The outcome of each simulation is recorded, and the move with the best average result is selected.

The idea behind MCS is that by running enough random playouts from a given game state, we can obtain a probabilistic estimate of which move is most likely to lead to a win. This relies on the Law of Large Numbers: as the number of simulations increases, the average result converges toward the true expected value of each move.

In Easy mode, the bot runs MCS with 100 total simulations, while Medium runs 1000 total simulations. The lower number of simulations in Easy mode adds significant noise to the distribution, resulting in the bot making frequent blunders, making it suitable for beginners. At level Medium, the bot is significantly smarter. We observe many central plays, which are statistically likelier to generate win conditions.

A notable quirk of MCS is its tendency to favor deceptive strategies that rely on the opponent making unforced errors. For instance, as shown in Figure 20, the bot often selects an opening sequence that sets up a simple tactical trap. If the opponent plays carelessly, the bot can secure a quick win within just a few moves. However, an experienced player will immediately recognize and neutralize the threat, rendering the sequence ineffective.

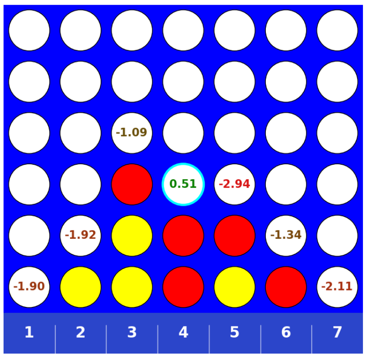


**Figure 19** An MCS-driven bot (yellow) runs random playouts proceeding from playing at each column. A total of 1000 simulations were made, and the numbers for each move depict the total reward gained from that move normalized by the simulation count. A positive reward of 1 was given for each win, while a loss was penalized by -10. Column 4 is deemed the best move as it blocks a mate in 2 for red. Move 3 also blocks the immediate threat, which is why it is ranked second, but it is still a losing move, setting the game up for a loss further down the line. However, all other moves should be weighted equally badly, but here column 5 was calculated to be an especially terrible move. This is a result of sampling error skewing the results through noise.

This stems from a shortcoming of MCS: since it runs random playouts, it treats all branches of the search tree equally. Since the algorithm does not differentiate between plausible and implausible lines of play, it assigns similar weight to all outcomes, even those highly unlikely to occur in real games. As a result, it may overvalue strategies that succeed frequently against naive or random play but fail against thoughtful opposition. (We have migitated this issue to a certain extent by penalizing losses more heavily than we reward wins.)

This flaw becomes especially apparent in positions requiring subtle defense or multi-move planning. The bot might favor a line that appears strong statistically over one that would hold up against a skilled human. Consequently, while vanilla Monte Carlo can produce surprisingly effective moves in simple scenarios, its strategic depth is inherently limited.

### 7.2.2 Hard Mode: Monte Carlo Tree Search

To overcome the above drawbacks, Hard mode uses a more advanced form of simulation: Monte Carlo Tree Search (hereafter MCTS). Unlike vanilla MCS, which treats each simulation independently, MCTS learns from each simulation, building up a search tree that

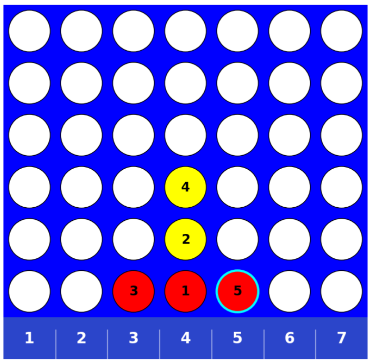


**Figure 20** A common opening by the MCS-driven Medium level bot (red), with pieces numbered by move order. By playing at column 5 (highlighted), red creates a so-called *double check*, where regardless of where yellow plays, red wins in the next turn. However, an experienced opponent would not play at column 4 in their 4th move, and would instead block the threat by playing at 2 or 5, immediately nullifying the threat.

prioritizes promising moves over time. The algorithm selects nodes in the tree using the Upper Confidence Bound for Trees (UCT) formula, which prioritizes moves that have either shown strong results in past simulations (exploitation) or have been less explored and thus might reveal better outcomes if investigated further (exploration). The exploration-exploitable trade-off is balanced with an exploration parameter that we set to $\sqrt{2}$ as advised by the literature.

At this level, the algorithm works significantly more efficiently, enabling us to maintain the same total of 1,000 simulations while still observing sophisticated tactics and strategic depth. To validate the superiority of Monte Carlo Tree Search over vanilla Monte Carlo Simulation, we conducted experiments pitting an MCS-driven bot against an MCTS-driven bot, setting the total simulations parameter to 10,000 for each. The results were striking: MCTS beat MCS over 90% of the time. This clearly demonstrates the substantial advantage of MCTS in exploring the game tree more effectively and making smarter decisions.

### 7.2.3 Fine-tuning

To determine the optimal parameters for each level (e.g. the number of simulations for both MCS and MCTS, the exploration constant, and the final move selection strategy in MCTS), we conducted extensive experimentation. We tested various parameter combinations by having the Easy, Medium, and Hard bots compete against each other. For each pair of opponents, we simulated 1,000 games with one player starting first, and an additional 1,000 games with the other player starting, totaling 2,000 games per matchup. This thorough testing allowed us to identify the parameter settings that produced the most balanced win rates between different levels.

The results of these tests in the final configuration of the levels is depicted in Figure 21. The progression in skill level across the Easy, Medium, and Hard bots is clearly evident. We still see lower levels defeating higher ones now and then, which is a result of the stochastic nature of the algorithms.

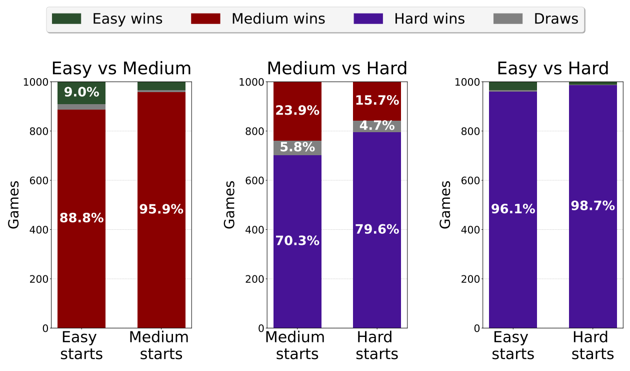


**Figure 21** Each of the 3 levels, Easy, Medium, and Hard, were played against each other. 2,000 games were simulated for each pair of opponents.

## 8 Evaluation and results

For the evaluation of the system's performance especially the following measures should be considered:

1. Accuracy of the physical moves and precision when inserting the chip.
2. The rate at which the detection of the computer vision system is successful.
3. The win rate of the system against a human player

These metrics are chosen since they reflect well the different components on which we worked while developing the system. These metrics also ensure a good game experience for the user if all them are fulfilled with high scores.

For evaluating these metrics we performed several games with a human player against the robot. This allows us to evaluate all of the measures. For a quantitative evaluation we count the total number of moves performed and then count the number of incidents occurred. We map each of the incidents to the quality

| Number of moves played | Number of board states | Cumulative sum |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 7 | 8 |
| 2 | 49 | 57 |
| 3 | 238 | 295 |
| 4 | 1,120 | 1,415 |
| 5 | 4,263 | 5,678 |
| 6 | 16,422 | 22,100 |
| 7 | 54,859 | 76,959 |
| 8 | 184,275 | 261,234 |
| 9 | 558,186 | 819,420 |
| 10 | 1,662,623 | 2,482,043 |
| 11 | 4,568,683 | 7,050,726 |
| 12 | 12,236,101 | 19,286,827 |
| 13 | 30,929,111 | 50,215,938 |
| 14 | 75,437,595 | 125,653,533 |
| 15 | 176,541,259 | 302,194,792 |
| 16 | 394,591,391 | 696,786,183 |
| 17 | 858,218,743 | 1,555,004,926 |
| 18 | 1,763,883,894 | 3,318,888,820 |
| 19 | 3,568,259,802 | 6,887,148,622 |
| 20 | 6,746,155,945 | 13,633,304,567 |
| 21 | 12,673,345,045 | 26,306,649,612 |
| 22 | 22,010,823,988 | 48,317,473,600 |
| 23 | 38,263,228,189 | 86,580,701,789 |
| 24 | 60,830,813,459 | 147,411,515,248 |
| 25 | 97,266,114,959 | 244,677,630,207 |
| 26 | 140,728,569,039 | 385,406,199,246 |
| 27 | 205,289,508,055 | 590,695,707,301 |
| 28 | 268,057,611,944 | 858,753,319,245 |
| 29 | 352,626,845,666 | 1,211,380,164,911 |
| 30 | 410,378,505,447 | 1,621,758,670,358 |
| 31 | 479,206,477,733 | 2,100,965,148,091 |
| 32 | 488,906,447,183 | 2,589,871,595,274 |
| 33 | 496,636,890,702 | 3,086,508,485,976 |
| 34 | 433,471,730,336 | 3,519,980,216,312 |
| 35 | 370,947,887,723 | 3,890,928,104,035 |
| 36 | 266,313,901,222 | 4,157,242,005,257 |
| 37 | 183,615,682,381 | 4,340,857,687,638 |
| 38 | 104,004,465,349 | 4,444,862,152,987 |
| 39 | 55,156,010,773 | 4,500,018,163,760 |
| 40 | 22,695,896,495 | 4,522,714,060,255 |
| 41 | 7,811,825,938 | 4,530,525,886,193 |
| 42 | 1,459,332,899 | **4,531,985,219,092** |

**Table 1** Number of possible board states with number of moves played

measures and so can calculate the percentage of moves causing problems. For the accuracy of the move, we count the move as successful if the coin was placed in the correct column and as false if it was not placed in the correct column or could not enter the column. For the computer vision we count a detection as not successful if the for a move relevant detection does not reflect the actual state of the game board. For the game rate, we count the wins by the system and by the player for the total number of games at the highest difficulty. The reason therefore is, that the other difficulties have the intention to let the player win.

The table 2 shows the results for incidents which occurred. We performed 10 games with a total of 154 moves.

| Type of Measure | Successes | Errors | Success Percentage |
|---|---|---|---|
| Inserting the chip | 152 | 2 | 98.7% |
| Game state detection | 154 | 0 | 100.0% |

**Table 2** Evaluation results of system performance

The results underline the high reliability of the system with an accuracy of 100 % of the computer vision system. The issues with the coin drop were caused by irregularities on the coin surface resulting from production quality. This error can be resolved through appropriate post-processing and a high surface finish.

## 9 Discussion

The evaluation shows that the system performs with high precision and reliability and can so perform its task of playing connect 4 successfully. This is achieved by a cyber-physical system composed of different components including the mechanical design, electronics and system architecture as well as computer vision and the implemented game-algorithm. The implementation shows a successful approach how to implement the game connect-4 in a system that can play the game automatically against a human player. The defined requirements in the beginning of the project are so fulfilled.
The final result shows some trade offs which were made between different factors. The first version shows a clear tendency towards a fast development which is focused on building a working version and less optimized towards a final product. When deciding on the components and especially when choosing the lead screw mechanism we aimed for a high precision to ensure that the chips drop in the correct column. The lead screw mechanism thus is a bit slower in the

movement compared to other possible movements but the tradeoff was made here in favor of the precision to ensure a high reliability of the system.
The system developed by us offers a system that fulfills the requirement of a system being capable to play connect 4. It finds its limitations in the size since the components are not optimized for a small sized version and much space is required. Furthermore since relying on the computer vision to detect the current game state, difficult light conditions such as strong light or also a lack of light in a room can cause problems for the system. We encounter this limitation by adding a calibration mechanism but in some edge cases it can still become a problem. These limitations describe the possible future work on the project which includes a development aiming towards a product which is smaller in size by using custom made components. In addition future mechanisms for the game state detection, such as detections in each column via light sensors to detect each inserted chip can be integrated and developed.

## 10 Conclusion

The development successfully includes the development of a reliable computer vision algorithm for the detection of the game state, the implementation of a game algorithm with a high win rate and the development of a electrical-mechanical system which is able to place chips with high precision in the correct column.
The project worked showed especially that calibration mechanisms (such as for the computer vision color calibration and the position of the stepper motor) increase the reliability of a system. In addition a key learning was made in the architecture of designing a distributed system of different components which communicate via serial connection to ensure an easy and reliable connection between different subsystems.

## List of Figures

## List of Tables