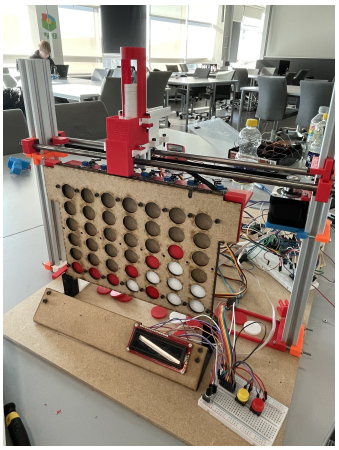
Connect Four Bot



Oskar Wernegreen & Joshua James Calba

29. May, 2026

1 Introduction

The aim for this project is to build a bot that can play Connect Four on a custom-built board against a human. The game rules will follow the general rules of Connect Four as is described by Triangle Lawn Game [1]. In brief, players take turns placing a disc of their color in a game board consisting of seven columns, and the player to first place four discs in a row wins. This report goes through development of the project in detail, focusing on the design choices with regards to the mechanisms, electronics, and software, how these affected results, and what was learned from this.

2 Background

The idea of creating a robot that can play board games is not new. Many examples exist of robots that can play games such as Chess and Tic-Tac-Toe, and even Connect Four, against a human player. Still, we wanted to try our hand at it because it posed an interesting challenge with regards to the precision of mechanisms and sensors, and how these are integrated together in the control of a robot that takes the role of a human in some task. The idea of designing a robot that combines physical control and computer thought to replace a human intrigued us, even if replacing a human for this task is relatively simple.

3 Requirements

Informally, the bot should ”act as a human” in its ability to react and play the game, such that it takes the role of a human adversary. To achieve this, we set the more formal requirements of:

1) The bot must be able to detect moves made by the human, and update its state
2) The bot must choose its next move according to some logic, and the actions of the human.
3) The bot must be able to move and place a game piece into a specific, selected column of the game.
4) The bot must be able to be reset by a human player, such that multiple games can be played.
5) The bot must be able to go either first or second in the game, as decided by the human player.
6) The bot must be able to communicate game state to the player clearly.

As we chose to focus primarily on the physical aspects of the project, note that we set no requirements for planning in the moves of the bot, as this is entirely a software task.

4 DESCRIPTION

4.1 Game board

We chose to make our own Connect Four board instead of using a commercial one, since this gave us greater control, allowing us to more easily mount other parts to it, and cut holes for IR-sensors. The two boards that make up each side of the game are laser-cut from MDF, while every other part, e.g. removable bottom, the vertical dividers between columns of the game, and disc pieces for playing the game, are FDM-3D-printed from PLA.

4.2 Mechanics

The action of mechanically placing a disc, i.e. playing a move of the game, can be split into two distinct actions. The first is releasing a disc from some storage of discs that will be used throughout the game, in such a way that the disc is correctly angled according to the board of the game. The second is ensuring that the disc lands in the correct spot intended by the bot after being released.

4.2.1 Disc dispensing mechanism

To release discs in order for the bot to make a move we constructed a dropper mechanism consisting of a vertical storage of discs, a servo motor with an attached gear (pinion), and a rack. The mechanism functions by turning the rotational motion of the pinion into a linear motion of the rack, such that the end of the rack pushes out the bottom-most disc in the storage, after which the rack pulls back and the next disc falls down, ready to be dispensed. Since the discs are stored horizontally, i.e. on their flat side, the final thing the dropper needs to do is change the orientation of the disc. This is accomplished by making the disc slide down a narrow, square pipe after being dispensed. We chose to store the discs horizontally in spite of this slight extra complication, because it allows storing many more discs in the dropper (21 in total, the max amount you need to play a game of Connect Four). It also makes the storage more stable when the dropper is moving.

Several different mechanisms were considered. The initial idea was to use a double servo horn as the moving part of the dispenser. However, the mechanism could not simply rotate continuously, since this would cause the next disc to be pushed in the opposite direction. Therefore, the horn was redesigned as an obround-shaped plate, as shown in Figure 8. We found this design to be more difficult to prototype and iterate on than the rack-and-pinion approach, and so decided against it.

Another solution that was considered was a linkage mechanism. A linkage consists of rigid parts connected by joints, and converts rotational motion into linear motion, similar to a rack-and-pinion mechanism. However, a linkage mechanism appeared more difficult to design, build, and test reliably within the available time, and has little advantage compared to the rack-and-pinion approach for this application. Therefore, we went with the rack-and-pinion mechanism as our final solution for the dispenser mechanism.

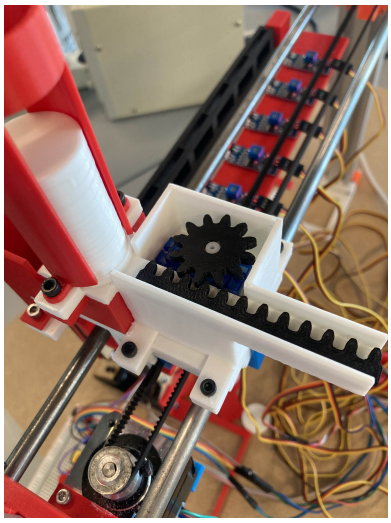


Fig. 1. Rack-and-pinion disc pusher

4.2.2 Carriage guide mechanism

To ensure that the disc is dropped in the correct column of the game board, we placed the described dropper on a carriage, attached to a linear guide consisting of two metal rods placed horizontally, i.e. at the same height.

The carriage is moved to the desired location by a 6 mm G2T belt drive, to which it is attached at the bottom, with the belt clamped down flat by a small 3D-printed part with a single screw. We considered sturdier approaches where the belt is under more friction, e.g. a more complex shape than a flat surface, but informal testing showed that this simpler approach was entirely sufficient. The carriage rests on, and is attached to the guide rails, by way of two linear bearings, one on each rail. This ensures smooth movement across the rails. To actually drive the belt drive we used a stepper motor.

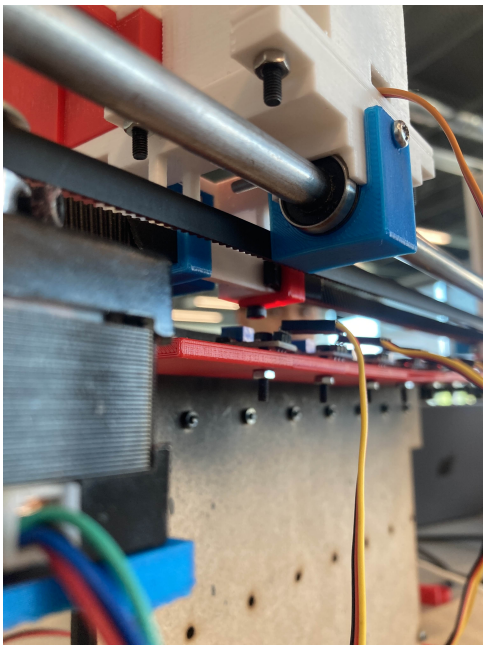


Fig. 2. Carriage attached to belt drive

Both the guide rails and the stepper motor are held up on two v-slot profiles by being attached to a 3D-printed structure clamped to the v-slot profiles, such that the dropper is in line with the top of the game board. The horizontal orientation of the guide rails was chosen because it allows both the stepper motor and the guide rails to be as close as possible to the v-slot profiles holding them up, which makes the whole structure much more stable, while keeping the need for e.g. 3D-printed support structures minimal. Since the v-slot profiles are connected by both being clamped to the same guide rails, and the ends of the v-slot profiles (touching the ground) are mostly flat, this on its own produces a somewhat stable structure. We found it sufficient to simply clamp small 3D-printed "feet" to the bottom of the v-slot profiles to ensure the stability. These feet also connect to the game board itself, such that everything is one connected construction.

In the beginning of the project we considered an entirely different approach, where the dropper would be stationary, dropping coins that would then roll down a slide to the correct column. The slide would have a gear rack at the bottom, and be moved by a gear (pinion) connected to a stepper motor. This approach was discarded, because it would require a slide with a length nearly equal to the width of the game board, and this slide would at times have to stick out almost all the way to one side of the board, making it quite unwieldy. Further, unless we introduced a third actuator to move the dropper up and down (eliminating the gain from having a stationary dropper), the distance the disc would have to drop to the slide would differ depending on the position of the slide, and would introduce some imprecision.

4.3 Electronics

4.3.1 Control and power

The bot uses an Arduino UNO R3 to control the electrical components. The Arduino is powered by USB, supplying 5V DC. A separate 12V DC power supply is used for the higher demands of the stepper motor, and a common ground is kept to prevent floating voltages, especially with regards to the motor driver. An Arduino UNO R3 was sufficient for our project, since we kept our code relatively simple. A more sophisticated approach to the bot's AI might have necessitated a more powerful controller. The pin count of the Arduino is also just about sufficient for our project without having to e.g. use I2C expanders.

4.3.2 Sensors

In order for the bot to play intelligently against a human adversary, it needed some way to detect moves made by the player. For this we decided to go with an array of seven digital FC-51 IR-sensors aligned with - and hanging off - the top of the game board, such that discs dropped by the human are detected and a digital LOW signal is sent to the Arduino. A digital signal is sufficient, since we only care for whether or not a disc is dropped, and the distance would always be the same regardless. The IR-sensors are powered by the 5V of the Arduino.

Since the material of the game board is somewhat reflective, this provided a challenge for the IR-sensors. It was initially impossible to tune the IR-sensor (through their potentiometers) such that they detected dropping discs but not the background, since the difference in reflectivity between the PLA of the discs and the MDF of the game board is quite small, and the slight difference in distance from the discs dropping in front of the game board does not make up for that. To solve this, we simply increased the difference in reflectivity by spray-painting the top of the game board black, which worked very well.

Another approach to this could have been to increase the reflectivity of the discs by e.g. applying some reflective foil to their sides. However, this would still have necessitated placing the IR-sensors further back to avoid detecting the background, which might have introduced complications with sensors being triggered by coins dropping into adjacent columns.

Initially we considered other ways to detect changes to board state, like putting pressure sensors at the bottom of each column. This might have made for a cleaner-looking design without IR-sensors sticking out on a raised platform at the top of the game, but it would also have made the standard approach to resetting the game (dropping out the bottom) impossible. The logic would also be more complicated, as we would need the signal to be analog in order to detect changes from discs dropping on already-placed discs, and with seven columns - seven sensors, and only six analog pins on our Arduino, this would also have to be changed.

Besides having their moves detected by the IR-sensors, the player interacts with the bot through two button switches placed at the front of the game, and wired to the Arduino digital input with a pull-down resistor to prevent floating voltages. A rotary potentiometer wired to the 5V of the Arduino controls the lighting of the LCD screen.

4.3.3 Motors

For dispensing discs we used a SG90 servo motor, with its control wired into the D11 PWM pin of the Arduino, and powered by the 5V of the Arduino as well. Although it is wired into a PWM pin, we do not make use of the pulse-width modulation. The main reason for selecting this motor was its low weight compared to other motor types. Furthermore, the dispensing mechanism does not require continuous rotation, making a servo motor suitable for this task.

The motor used for our belt drive is a Nema 17 bipolar stepper motor, controlled by the Arduino through an A4988 Pololu driver, with the VMOT of the driver wired to 12V and the VDD wired to the 5V of the Arduino. The Arduino controls the stepper motor through the STEP and DIR pins of the driver. Although mounting a heavy motor like this as high up as is needed was a challenge, this type of motor is necessary to get the required precision and control of the movement of the carriage and attached dropper.

4.3.4 LCD display

The bot uses a HD44780U 16x2 LCD to communicate game state to the player. Although it is wired to the analog pins of the Arduino, it does not make use of these. We are only using four of the eight total data bus pins on the LCD. This is sufficient and it works fine, as it only affects the speed/responsiveness of the LCD display, and this is not critical to the functioning of the bot. Using all eight data bus pins would require expanding the Arduino with an I2C expander, or using a different Arduino. The LCD is powered by the 5V from the Arduino.

5 SOFTWARE

The Arduino is responsible for running code to detect player actions, make the bot act accordingly, and to communicate game state to the player. The mutable part of the program state is first and foremost the program model of the state of the board, represented by a two-dimensional byte-array (6 rows, 7 columns). All gameplay-decisions of the bot are made according to this, and it is updated throughout each game.

At each loop() of the program, the Arduino *always* reads the voltage of each of the wired sensors, i.e. all seven IR-sensors and the two buttons. This input, alongside the existing state, is then used in a large if-else block that determines the entire functioning of the bot.

One button (red in the final prototype) signals that the game should reset. This clears the game board state and resets a boolean flag that shows whether the game is ”fresh”, for use in deciding who goes first. Crucially, the reset button does not physically reset the board, and this has to be done manually by the human player. On pressing the reset button, the program *assumes* that the physical game board is empty. The other button (yellow) signals that the starting player should switch. This has no effect once a game is in session, and only takes effect once the game is reset.

State on the previous voltage reading of each button is kept, such that actions are only taken in response once on a button press, and not on each following loop. This is not too much of a problem for the reset button, as this is essentially idempotent, but for the ”startPlayer”-button it is crucial to avoid confusion.

If it is the player’s turn to make a move, then the program will continuously loop until the player makes a move (detected by the IR-sensors), at which point the bot will respond with its own move. In this way, the detection of it being the bot’s turn, and the carrying out of that turn, is all done within a single loop. This makes reasoning about turns easy: at the end of each loop(), it is always either the human player’s turn, or the game is over. This design is guided by the fact that discs dropped by the bot also trigger a LOW voltage signal from the IR-sensors, so it is crucial to avoid any confusion on the part of the bot as to what its response to sensor inputs should be.

We did not spend much time on the algorithm for deciding the moves of the bot. It uses a simple greedy algorithm, described by:

1) Make a game-winning move.
2) Block a game-winning move for the human player.
3) Build on the longest line of discs. If tied, place in the area with the most discs of any color. If still tied, place on the lowest row possible.

Ordered by priority.

The outcome of this algorithm is a column number for the bot to place a disc in. This is immediately carried out. From its initial position at the side of the game board, the carriage and dropper is moved above the correct column by the stepper motor. The number of steps to take is a calibrated constant in the program, at 161 full steps to move one column over. Once this is done, the Arduino signals for the servo motor to activate using the ”Servo.h” library. With the disc now dropped, the stepper motor reverses and takes an equal number of steps. In this way, the dropper returns to its initial ”home” position.

The LCD screen communicates the current status of the game to the player, using the ”LiquidCrystal.h” library. This includes wins, losses, draws, and the turn status, i.e. whether it is ready to accept a move by the player. On button presses, response text is only shown briefly, so that the input is acknowledged, but game information is not obscured.

The delay between loops is kept short with a very minimal delay. This is required, as for the bot to not miss the human dropping a disc, the interval between IR-sensor readings must be shorter than the time it takes for a disc to drop even when moved only by gravity.

6 ANALYSIS

The most important part of the bot, and the most difficult, is its ability to correctly sense the moves of the player, and to move and drop discs of its own accordingly. To analyze its ability to do both, we set up a simple integration test. We ran some simple code, where the bot senses the moves of the player, but instead of deciding what move to make based on the algorithm described previously (since this is entirely a software-problem), it simply ”mirrors” the player, i.e. places a disc in the same column. We did this 100 times, with the moves of the player randomized. See appendix C for the test results.

The test results are very solid, with no mistakes of any kind, although they do not present the full picture. The ability of the bot to carry out moves of its own is more inconsistent than the test data shows. While the servo motor drops the discs nicely, it does not take much for the dropper mechanism to be misaligned with the openings at the top of the game board. If this happens, dropped discs will get stuck, and they will keep doing so until the positioning is fixed manually. If we had e.g. moved or jostled the bot in the course of the test, the performance would likely have

been worse.

Generally, the player-move detection works very well, as is seen in the testing as well. Once the IR-sensors are set correctly, they practically always detect a disc passing by them, and never detect discs dropping in adjacent columns. Further, once set correctly, i.e. with their sensitivities tuned and placed correctly relative to the game board, they will keep working correctly until you very actively interfere with them.

The logic for playing a game from end to end works well and without issues, as do the buttons for resetting and changing starting player. The algorithm for deciding the best move seems passable. We very informally tested it on two people who were not involved in the creation of the bot, and it managed to win against both.

At times, when the floor of the game board is pulled out to reset the game, discs in the two outermost columns can get slightly stuck and need to be pushed down.

7 Discussion

The greatest weakness of the bot is how narrow the margin of error is on dropping discs correctly. While the pointed ends of the vertical dividers between columns help in one direction, the sharp 90-degree angles of the game board mean that discs can easily get caught in the other, if the dropper is even slightly misaligned with the game board. A solution to this could be to change the openings at the top of the board, such that each opening has a funnel-shape in every direction. Another thing that likely contributes to this issue is the relative shakiness of the overall construction. While the game board is connected to the v-slot profiles holding up the linear guide for the dropper, this is only at the base (see appendix B, Figure 3). This makes the position of the linear guide for the dropper relative to the game board not entirely static. Likely, some improvements could be made to the consistency of the disc dropping if the game board was also attached to the v-slot profiles near the top, such that the dropper and the game board cannot swing away from each other.

The bot generally responds very well to user input, and practically never misses a move made by the player, provided it was in a state that was ready to read the move. The communication back to the player through the LCD screen is mostly clear as well, although the screen can be hard to read at times depending on the lighting. Combined, an impatient player can easily, unwittingly, make a move when the bot is not ready. The bot being ready to read the player's move could be made more clear, perhaps simply by an LED turning on and off.

We do not gain anything from the carriage having a "home" at the end of the linear guide, as we do not use any form of sensor to detect when the carriage is back in position, instead relying entirely on the precision of the stepper motor to return the carriage. This approach seems to work fine. It also means that there is a simple improvement we could freely make. By making an opening at the top of the part of the dropper mechanism that guides discs down to the column, the player would be able to drop discs even when the dropper is positioned over that column. This would eliminate the need for a home entirely, making the carriage have to move less, for no loss in precision.

To go beyond being a simple prototype, besides fixing the above, some smaller things should also be improved. First of all, the rather messy wiring should be made neater, and the loose electronics in the form of the breadboards and Arduino should be attached somewhere convenient, ideally the back of the game board. Screw lengths could also be changed to be more appropriate in places. Finally, for the bot to be of real interest to play against, the AI should be improved, with the most important addition being some non-deterministic element to provide variability to games. For mass production, we would ideally not have so many small pieces. Instead of the two current metal v-slot profiles, you could possibly mold two similar parts, with feet and attachment points for the pulley, guide rails, etc. built in, reducing the need to 3D-print and fasten these parts. The two laser-cut boards that form the core of the Connect Four game work quite well for mass-production already.

Overall, the bot fulfills the requirements set. While it is prone to misaligning slightly, and thereby breaking, when set up correctly you can play multiple games of Connect Four in a row with the bot acting as a human adversary, as we set out to achieve.

REFERENCES

[1] Nashville Lawn Games. Connect four rule book: How to play. https://www.nashvillelawngames.com/wp-content/uploads/2019/10/Connect-Four-Rule-Guide-How-To-Play-Connect-4.pdf, October 2019. Accessed: 2026-05-20.

APPENDIX A
BILL OF MATERIAL

TABLE 1
Bill of Materials for the Connect Four Bot

Component	Amount	Specification / Purpose	Source
Mechanical Components			
M2×10 screws	4	Fastening component	–
M2 nuts	4	Fastening component	–
M3×14 screws	2	Fastening component	–
M3×16 screws	4	Fastening component	–
M3×20 screws	27	Fastening component	–
M3×25 screws	13	Fastening component	–
M3×40 screws	9	Fastening component	–
M3×50 screws	2	Fastening component	–
M3 nuts	57	Fastening component	–
8 mm metal rod	2	Linear guide rod	–
Double V-slot aluminum profile	2	Vertical support frame	–
Belt GT2 6 mm (70 cm)	1	Help moving the carriage	–
Linear Ball Bearings 6 mm	2	Help moving the carriage	–
Electronic Components			
FC-51 IR sensor module	7	Infrared reflection sensor for detecting the human players game disc	BitByg
HD44780U-compatible LCD	1	Display the game status	SparkFun
NEMA 17 stepper motor	1	Moving the carriage	3D Eksperten
Pololu A4988 stepper driver	1	Stepper motor driver carrier	Pololu
SG90 servo motor	1	Dispensing the discs using a rack and pinion	Ardustore.dk
Arduino UNO R3	1	Main microcontroller board	ArduinoStore
Switches	2	User input / control	Amazon
Potentiometer	1	Adjustable analogue input	–
Resistors	1	Resistance values to be confirmed	–
100 μF capacitor	1	Power smoothing / stabilisation	–
Breadboard	1	Circuit prototyping board	–
Long breadboard	1	Extended circuit prototyping board	–

APPENDIX B
TECHNICAL DRAWINGS

This section represent technical drawing of the different parts.

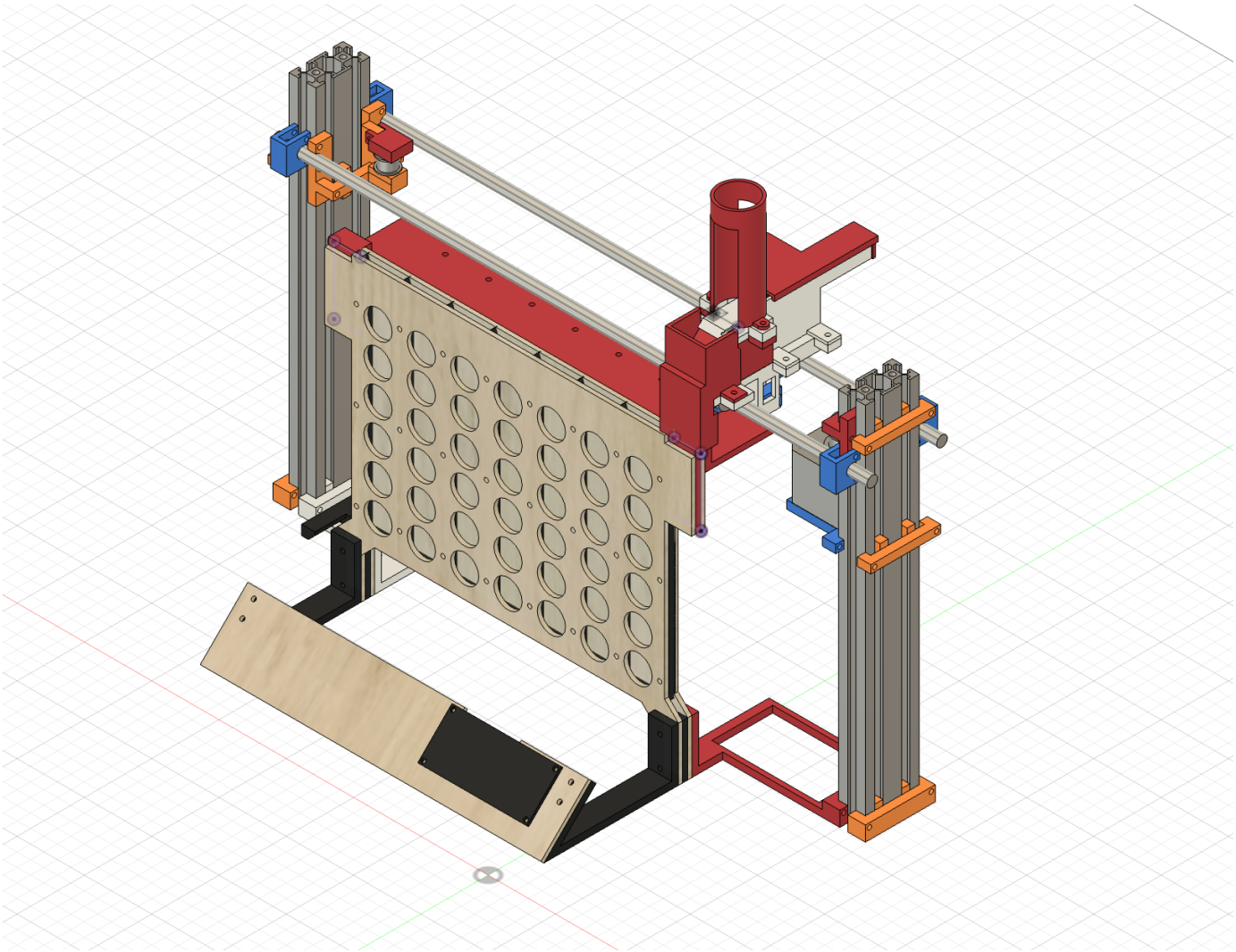


Fig. 3. Full assembly in Auto Fusion 360.

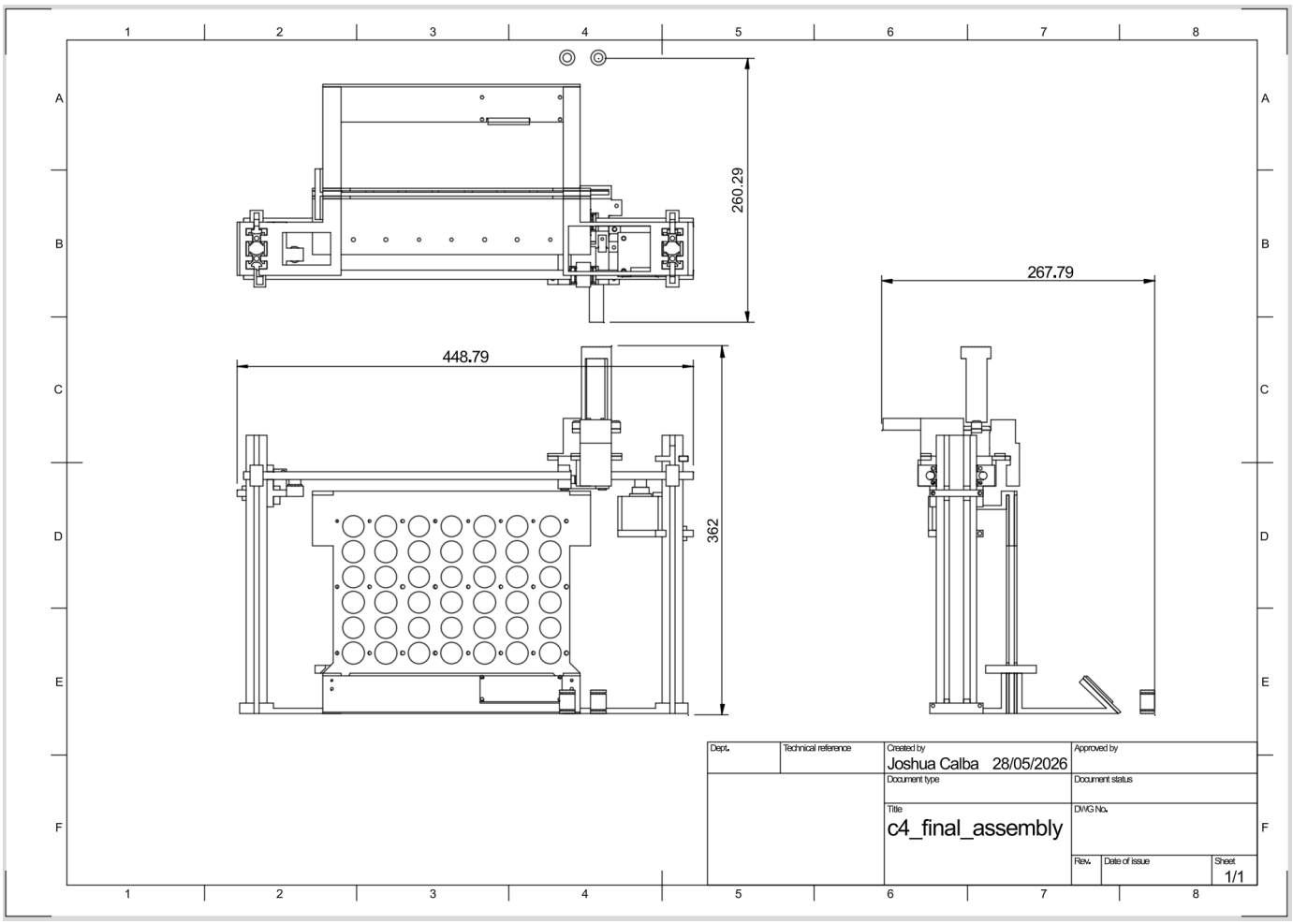


Fig. 4. Technical drawing of the full assembly

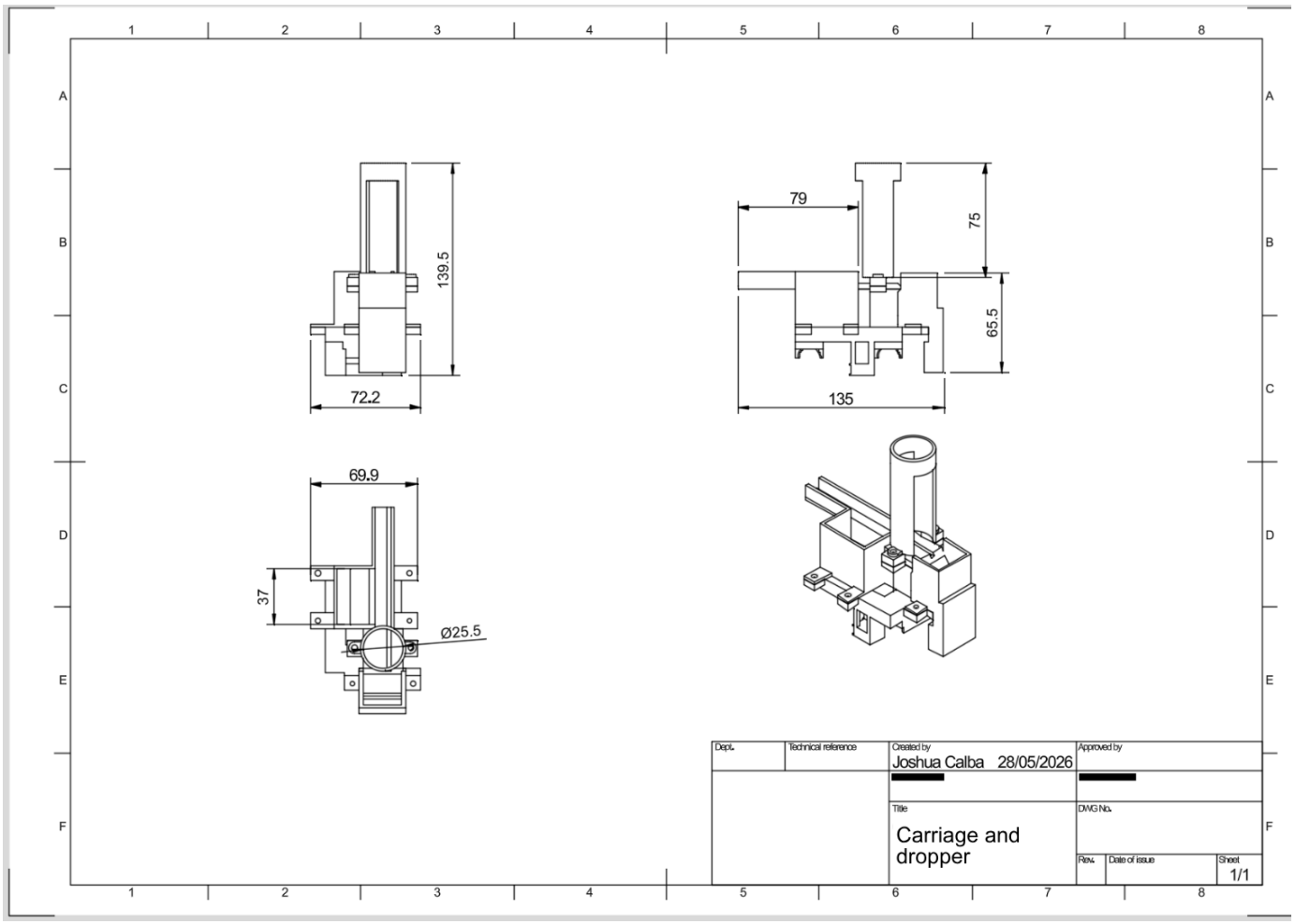


Fig. 5. Technical drawing of the carriage and dropper

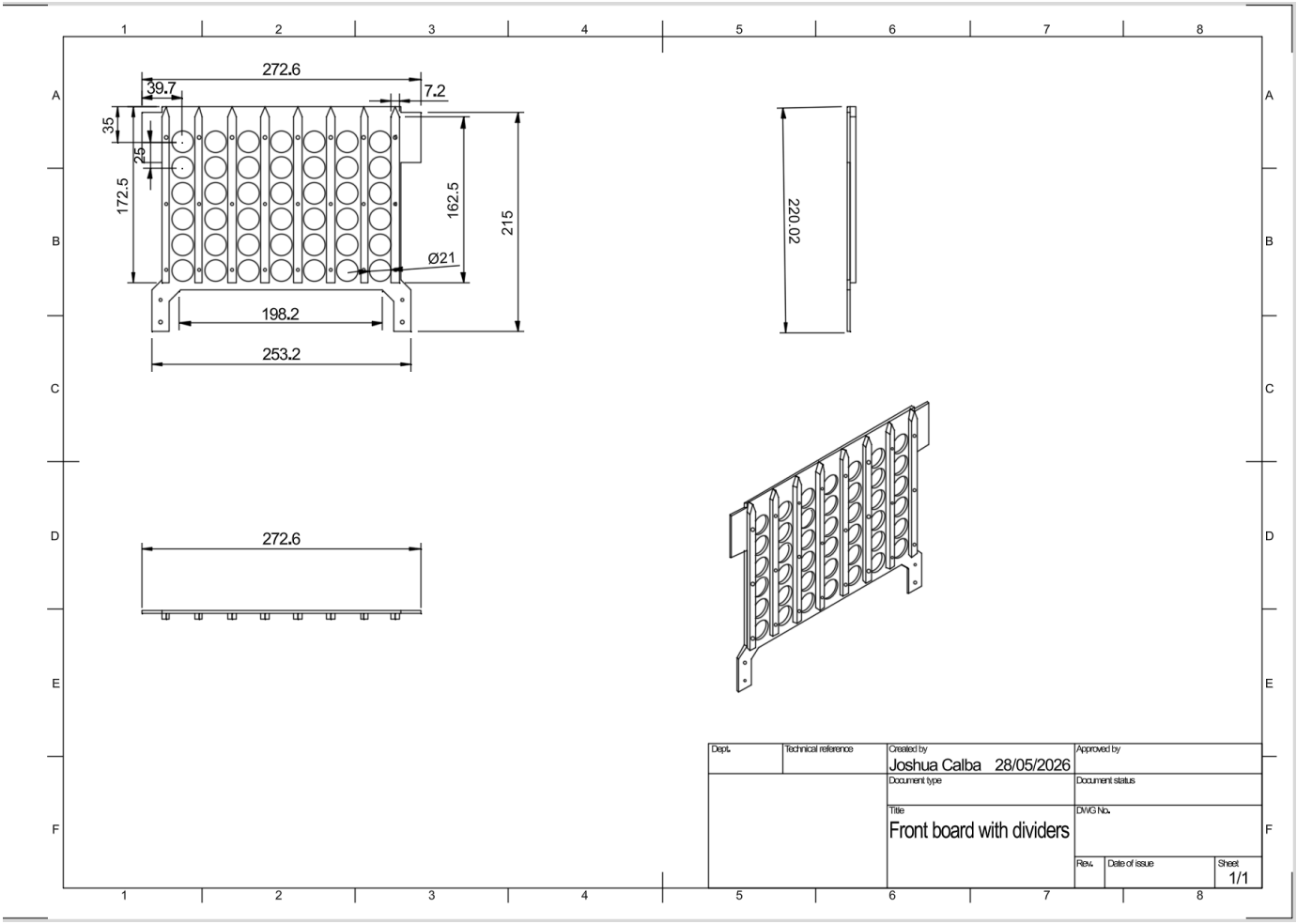


Fig. 6. Technical drawing of the front board with the vertical dividers

APPENDIX C
SCHEMATICS

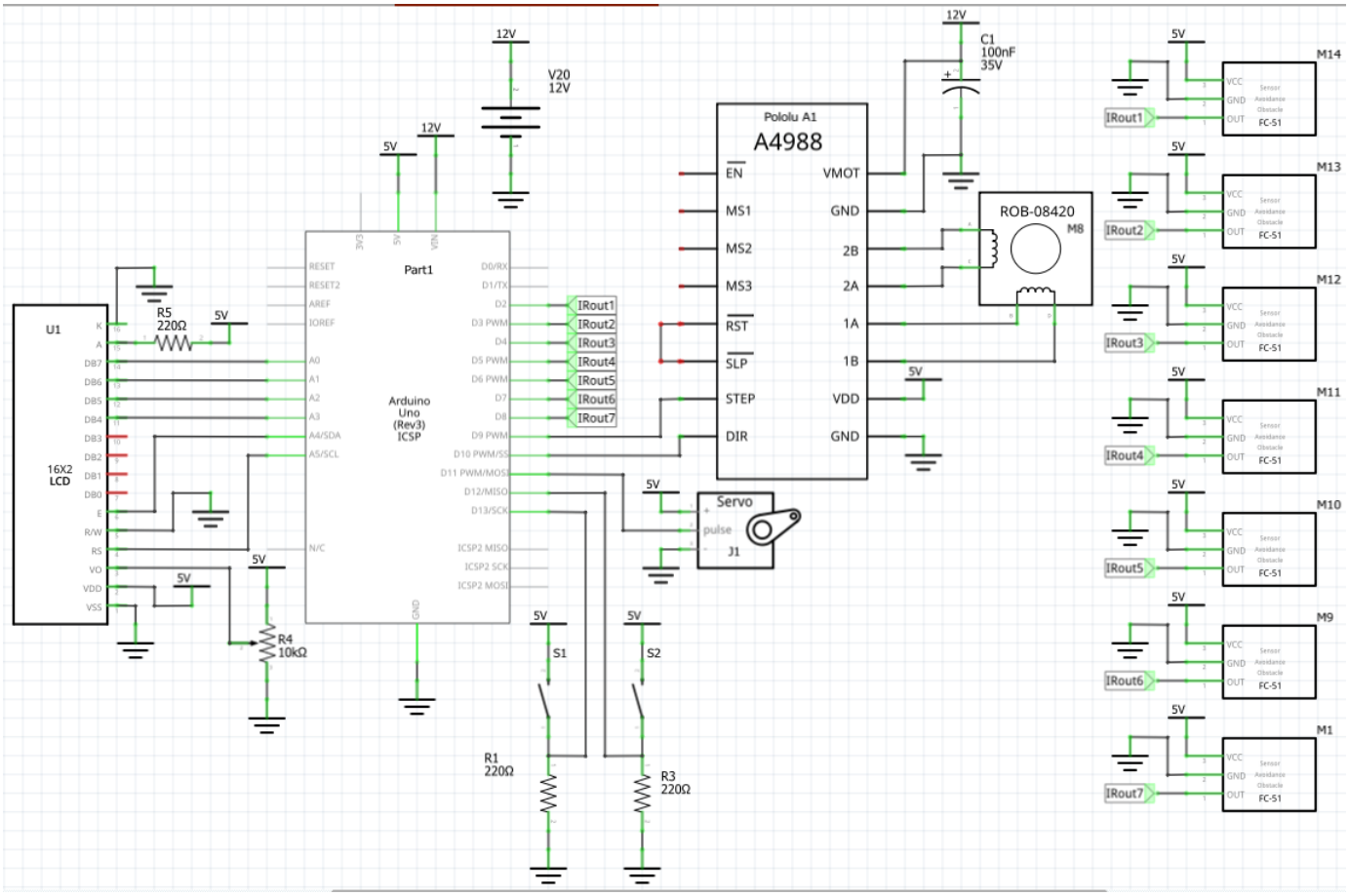


Fig. 7. Schematics of our Connect Four Bot

APPENDIX D
SOURCE CODE

Connect Four bot game logic

```
1
2  #include <LiquidCrystal.h>
3  #include <Servo.h>
4
5  //LCD pins
6  //RS, E, DB4, DB5, DB6, DB7
7  LiquidCrystal lcd(A5, A4, A0, A1, A2, A3);
8
9  //servo and pin
10 Servo myServo;
11 const int servoPin = 11;
12
13 //switch pins
14 const int switch1Pin = 12;
15 const int switch2Pin = 13;
16
17 //stepper driver and pins
18 const int stepPin = 9;
19 const int dirPin  = 10;
20 const int stepsPerColumn = 161;
21
22 //IR sensor pins
23 const int irPins[7] = {2, 3, 4, 5, 6, 7, 8};
24
25 //board setup
26 const int ROWS = 6;
27 const int COLS = 7;
```

```
28
29 const byte EMPTY = 0;
30 const byte HUMAN = 1;
31 const byte BOT   = 2;
32
33 byte board[ROWS][COLS];
34
35 //misc
36 bool humanStarts = true;
37 bool gameOver = false;
38 bool prevResetBut = false;
39 bool prevStartPlayerBut = false;
40 String lastMsg = "";
41 bool freshGame = true;
42
43
44
45 void setup() {
46   //fill board with zeroes
47   clearBoard();
48
49   //LCD
50   lcd.begin(16, 2);
51   showMessage("Welcome", true);
52
53   //serial
54   Serial.begin(9600);
55   Serial.println("starting");
56   //switches
57   pinMode(switch1Pin, INPUT);
58   pinMode(switch2Pin, INPUT);
59
60   //IR sensors
61   for (int i = 0; i < 7; i++) {
62     pinMode(irPins[i], INPUT);
63   }
64
65   //servo
66   myServo.attach(servoPin);
67   myServo.write(180);
68
69   //stepper driver
70   pinMode(stepPin, OUTPUT);
71   pinMode(dirPin, OUTPUT);
72
73   //set initial direction
74   digitalWrite(dirPin, HIGH);
75
76   delay(1000);
77 }
78
79 void loop() {
80   bool wasFresh = false;
81   //fresh game logic
82   if (freshGame){
83     showMessage("Game started", true);
84     delay(2000);
85     if (humanStarts){
86         showMessage("Make your move", true);
87     }
88     freshGame = false;
89     wasFresh = true;
90   }
91   //sensor readings
92   int sensorVal = -1;
93   bool resetBut = false;
94   bool startPlayerBut = false;
95   for (int i = 0; i < 7; i++) {
96     if (digitalRead(irPins[i]) == LOW){
97       sensorVal = i;
98     }
99   }
100  if (digitalRead(switch1Pin) == HIGH){
101    resetBut = true;
102  }
103  if (digitalRead(switch2Pin) == HIGH){
104    startPlayerBut = true;
```

```
  }

  //decision making
  if (startPlayerBut == true && prevStartPlayerBut == false){
    //switch who starts when game is new
    if (humanStarts){
        humanStarts = false;
        showMessage("Bot goes first", false);
        delay(2000);
        showMessage(lastMsg, false);
    }
    else {
        humanStarts = true;
        showMessage("You go first", false);
        delay(2000);
        showMessage(lastMsg, false);
    }
  }
  else if (resetBut == true && prevResetBut == false){
    //reset game, make assumption that board has been cleared physically
    gameOver = false;
    freshGame = true;
    clearBoard();
    showMessage("Game reset", true);
    delay(2000);
  }
  else if (gameOver){
    //do nothing, screen should be showing correct message from elsewhere
  }
  else if (sensorVal != -1 || (wasFresh && !humanStarts)){
    showMessage("Thinking...", true);
    delay(1000);
    //update board state, check if player won by their move
    if (!wasFresh || humanStarts){
        bool success = makeMove(sensorVal, HUMAN);
        if (checkWin(HUMAN)) {
            gameOver = true;
            showMessage("You win!", true);
        }
        else if (checkDraw()) {
            gameOver = true;
            showMessage("Draw!", true);
        }
    }
    if (!gameOver){
        int botCol = chooseBotMove();
        showMessage("Moving", true);
        //move out
        digitalWrite(dirPin, HIGH);
        for (int i = 0; i < stepsPerColumn * (botCol + 1); i++) {
            digitalWrite(stepPin, HIGH);
            delayMicroseconds(1500);

            digitalWrite(stepPin, LOW);
            delayMicroseconds(1500);
        }
        //servo pushes out coin
        myServo.write(0);
        delay(1000);
        myServo.write(180);
        delay(1000);
        //move back
        digitalWrite(dirPin, LOW);
        for (int i = 0; i < stepsPerColumn * (botCol + 1); i++) {
            digitalWrite(stepPin, HIGH);
            delayMicroseconds(2000);

            digitalWrite(stepPin, LOW);
            delayMicroseconds(2000);
        }
        //update board state
        bool success = makeMove(botCol, BOT);
        // game over check
        if (checkWin(BOT)) {
            gameOver = true;
            showMessage("You lose!", true);
        }
```

```
        else if (checkDraw()) {
            gameOver = true;
            showMessage("Draw!", true);
        }
        else {
            showMessage("Make your move", true);
        }
    }
  }
  //finish up
  prevResetBut = resetBut;
  prevStartPlayerBut = startPlayerBut;
  delay(20);
}
//board functions
void clearBoard() {
  for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLS; col++) {
      board[row][col] = EMPTY;
    }
  }
}

bool isColumnFull(int col) {
  return board[0][col] != EMPTY;
}

int getNextOpenRow(int col) {
  for (int row = ROWS - 1; row >= 0; row--) {
    if (board[row][col] == EMPTY) {
      return row;
    }
  }
  return -1;
}

bool makeMove(int col, byte player) {
  if (col < 0 || col >= COLS) {
    return false;
  }
  int row = getNextOpenRow(col);
  if (row == -1) {
    return false;
  }
  board[row][col] = player;
  return true;
}

bool checkWin(byte player) {
  //horizontal check
  for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLS - 3; col++) {
      if (board[row][col] == player &&
          board[row][col + 1] == player &&
          board[row][col + 2] == player &&
          board[row][col + 3] == player) {
        return true;
      }
    }
  }

  //vertical check
  for (int row = 0; row < ROWS - 3; row++) {
    for (int col = 0; col < COLS; col++) {
      if (board[row][col] == player &&
          board[row + 1][col] == player &&
          board[row + 2][col] == player &&
          board[row + 3][col] == player) {
        return true;
      }
    }
  }

  //diagonal down-right check
  for (int row = 0; row < ROWS - 3; row++) {
    for (int col = 0; col < COLS - 3; col++) {
      if (board[row][col] == player &&
```

```
          board[row + 1][col + 1] == player &&
          board[row + 2][col + 2] == player &&
          board[row + 3][col + 3] == player) {
        return true;
      }
    }
  }

  // diagonal up-right check
  for (int row = 3; row < ROWS; row++) {
    for (int col = 0; col < COLS - 3; col++) {
      if (board[row][col] == player &&
          board[row - 1][col + 1] == player &&
          board[row - 2][col + 2] == player &&
          board[row - 3][col + 3] == player) {
        return true;
      }
    }
  }
  return false;
}

bool checkDraw() {
  for (int col = 0; col < COLS; col++) {
    if (!isColumnFull(col)) {
      return false;
    }
  }
  return true;
}
//bot ai
int chooseBotMove() {
  //check if BOT can win immediately
  for (int col = 0; col < COLS; col++) {
    if (isColumnFull(col)) continue;
    int row = getNextOpenRow(col);
    board[row][col] = BOT;
    bool winningMove = checkWin(BOT);
    board[row][col] = EMPTY;
    if (winningMove) {
      return col;
    }
  }

  //check if HUMAN can win next turn, and block
  for (int col = 0; col < COLS; col++) {
    if (isColumnFull(col)) continue;
    int row = getNextOpenRow(col);
    board[row][col] = HUMAN;
    bool humanWouldWin = checkWin(HUMAN);
    board[row][col] = EMPTY;
    if (humanWouldWin) {
      return col;
    }
  }

  //otherwise pick move that makes the longest BOT line, if tied pick densest area, if still tied pick the
      lowest possible placement
  int bestCol = -1;
  int bestRow = -1;
  int bestLineScore = -1;
  int bestDensityScore = -1;
  for (int col = 0; col < COLS; col++) {
    if (isColumnFull(col)) continue;

    int row = getNextOpenRow(col);

    board[row][col] = BOT;

    int lineScore = longestLineFrom(row, col, BOT);
    int densityScore = densityAround(row, col);

    board[row][col] = EMPTY;

    if (lineScore > bestLineScore ||
        (lineScore == bestLineScore && densityScore > bestDensityScore) ||
        (lineScore == bestLineScore && densityScore == bestDensityScore && row > bestRow)) {
```

```
      bestLineScore = lineScore;
      bestDensityScore = densityScore;
      bestRow = row;
      bestCol = col;
    }
  }
  return bestCol;
}

int longestLineFrom(int row, int col, byte player) {
  int best = 1;

  best = max(best, countDirection(row, col, 0, 1, player));   //horizontal
  best = max(best, countDirection(row, col, 1, 0, player));   //vertical
  best = max(best, countDirection(row, col, 1, 1, player));   //diagonal down-right
  best = max(best, countDirection(row, col, 1, -1, player));  //diagonal down-left

  return best;
}

int countDirection(int row, int col, int dRow, int dCol, byte player) {
  int count = 1;

  //forward
  int r = row + dRow;
  int c = col + dCol;
  while (r >= 0 && r < ROWS && c >= 0 && c < COLS && board[r][c] == player) {
    count++;
    r += dRow;
    c += dCol;
  }

  //backward
  r = row - dRow;
  c = col - dCol;
  while (r >= 0 && r < ROWS && c >= 0 && c < COLS && board[r][c] == player) {
    count++;
    r -= dRow;
    c -= dCol;
  }

  return count;
}

int densityAround(int row, int col) {
  int score = 0;
  for (int r = row - 1; r <= row + 1; r++) {
    for (int c = col - 1; c <= col + 1; c++) {
      if (r < 0 || r >= ROWS || c < 0 || c >= COLS) continue;
      if (r == row && c == col) continue;
      if (board[r][c] != EMPTY) {
        score++;
      }
    }
  }

  return score;
}

//misc functions
void showMessage(String msg, bool store) {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print(msg);
  if (store){
    lastMsg = msg;
  }
}
```

Listing 1. Servo motor control for the disc dispensing mechanism.

APPENDIX E
TEST

TABLE 2: Testing results for the disc dispensing and positioning mechanism.

No.	Target column	Result column	Status	Comments
1	6	6	Success	–
2	1	1	Success	–
3	6	5	Success	–
4	7	0	Success	–
5	6	6	Success	–
6	1	1	Success	–
7	4	4	Success	–
8	2	2	Success	–
9	5	5	Success	–
10	2	2	Success	–
11	6	6	Success	–
12	1	1	Success	–
13	5	5	Success	–
14	6	6	Success	–
15	1	1	Success	–
16	4	4	Success	–
17	5	5	Success	–
18	7	7	Success	–
19	4	4	Success	–
20	1	1	Success	–
21	2	2	Success	–
22	6	6	Success	–
23	6	6	Success	–
24	7	7	Success	–
25	1	1	Success	–
26	2	2	Success	–
27	3	3	Success	–
28	4	4	Success	–
29	5	5	Success	–
30	6	6	Success	–
31	7	7	Success	–
32	6	6	Success	–
33	5	5	Success	–
34	4	4	Success	–
35	3	3	Success	–
36	2	2	Success	–
37	1	1	Success	–
38	3	3	Success	–
39	5	5	Success	–
40	7	7	Success	–
41	2	2	Success	–
42	4	4	Success	–
43	6	6	Success	–
44	3	3	Success	–
45	5	5	Success	–
46	7	7	Success	–
47	1	1	Success	–
48	4	4	Success	–
49	2	2	Success	–
50	7	7	Success	–
51	6	6	Success	–
52	4	4	Success	–
53	6	6	Success	–
54	2	2	Success	–
55	1	1	Success	–

Continued on next page

TABLE 2: Testing results for the disc dispensing and positioning mechanism (continued).

No.	Target column	Result column	Status	Comments
56	4	4	Success	–
57	3	3	Success	–
58	7	7	Success	–
59	5	5	Success	–
60	3	3	Success	–
61	2	2	Success	–
62	7	7	Success	–
63	4	4	Success	–
64	6	6	Success	–
65	7	7	Success	–
66	1	1	Success	–
67	4	4	Success	–
68	3	3	Success	–
69	2	2	Success	–
70	6	6	Success	–
71	5	5	Success	–
72	2	2	Success	–
73	1	1	Success	–
74	7	7	Success	–
75	4	4	Success	–
76	6	6	Success	–
77	3	3	Success	–
78	7	7	Success	–
79	3	3	Success	–
80	2	2	Success	–
81	1	1	Success	–
82	6	6	Success	–
83	4	4	Success	–
84	5	5	Success	–
85	2	2	Success	–
86	1	1	Success	–
87	4	4	Success	–
88	7	7	Success	–
89	3	3	Success	–
90	6	6	Success	–
91	1	1	Success	–
92	5	5	Success	–
93	2	2	Success	–
94	4	4	Success	–
95	6	6	Success	–
96	3	3	Success	–
97	7	7	Success	–
98	1	1	Success	–
99	2	2	Success	–
100	4	4	Success	–

APPENDIX F
MISCELLANEOUS

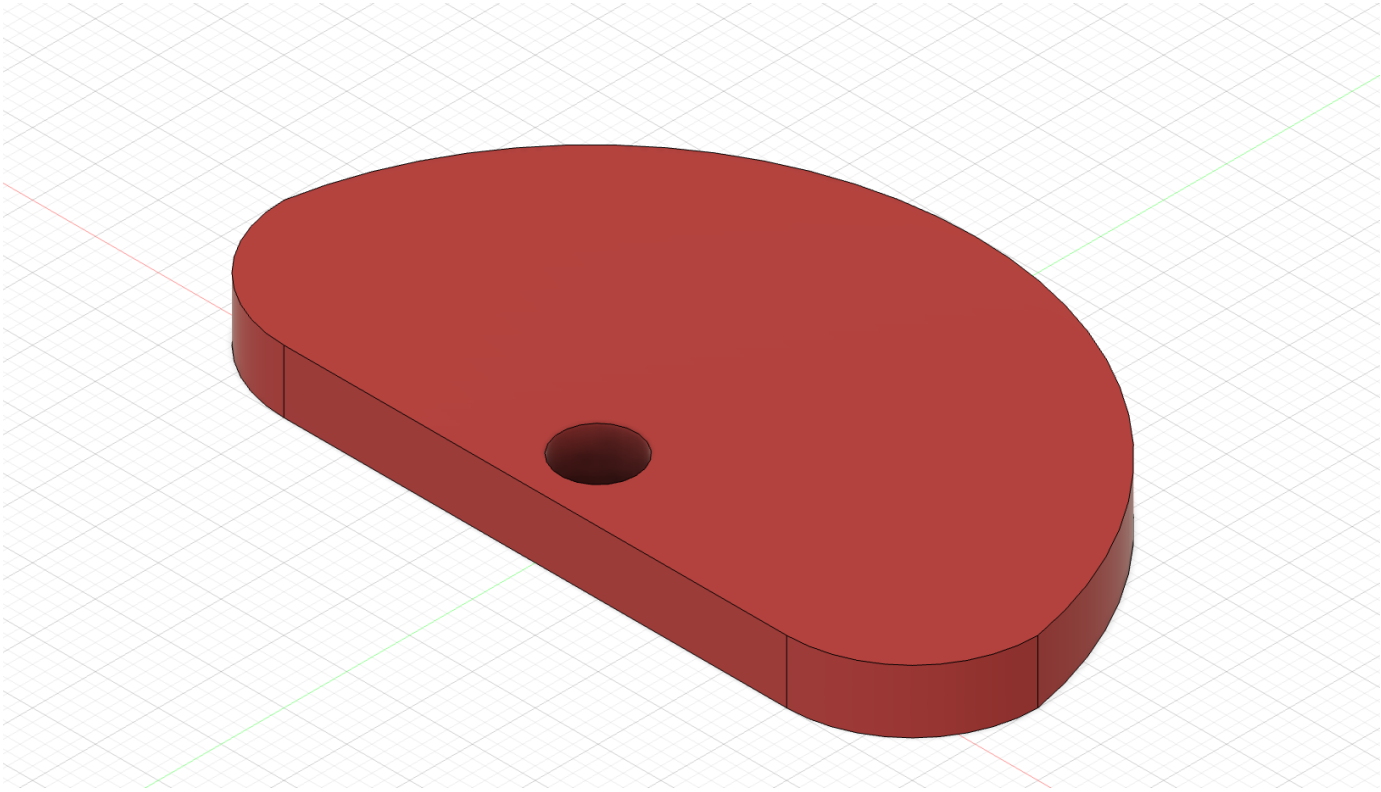


Fig. 8. Initial design for the disc dispenser mechanism