Magic Chess Board

Willem Van Dam (33500646) and Liam Foster (40199382)
willemdesu@gmail.com and liamfoster77@gmail.com

Figure 1 - Magic Chess Board

Abstract

The magic chess board is a physical chess board that moves the opponent's pieces automatically in response to the player's moves. The pieces each have a colored circle on top and a magnet embedded in the bottom. An overhead camera recognizes the pieces from their colored circle and an electromagnet underneath the board grips the piece from below using the embedded magnet. The electromagnet is mounted on a 2-axis, core-xy gantry system that is actuated by 2 24V NEMA 17 stepper motors. These motors are driven by two DM542T digital stepper drivers from StepperOnline, and are controlled by the green MECH 423 MSP board. The chess board uses a 24V power supply taken from a 3D printer for the motors and a 12V power supply for MSP. The electronics including drivers, motors and MSP are housed underneath the board and within the frame for a more clean look.

The general process for a move is as follows:

1. A picture is taken of the board with the overhead camera
2. The player makes a move
3. Another picture is taken and the player's move is decoded using computer vision
4. The player's move is fed into the chess AI
5. The chess AI's counter move is interpreted as a series of 2-axis trajectories
6. The trajectories and electromagnet control are sent in a packet to the MSP board
7. The MSP controls the stepper motors to move the 2-axis gantry
8. The MSP controls the electromagnet to grip and release the pieces as they move
9. The MSP sends a response packet once the move is complete
10. The camera takes another picture to update the computer vision and prompts the player for their next move

Objectives

The overall goal and vision of the project was to create an automatic chess board that could be used for practice by playing against a chess AI. The chess board would recognize the player's moves and respond automatically by moving the opposing pieces according to the chess AI's decisions. We planned to design and build:

- Chess pieces with magnets embedded in them
- An electromagnet capable of moving pieces from underneath the board
- A 2-axis gantry system to move the electromagnet
- A camera and mount to sense the piece position
- A chess board and frame to mount everything to and tie it all together

Most of our goals have been accomplished. Once calibrated the chess board is able to consistently recognize the piece type and team. It is also able to take output from the chess AI and accurately move the chess pieces to their destinations. It can take chess pieces consistently although sometimes the pieces hit each other, but the electromagnet does not grab multiple pieces at a time or make false moves.

Some goals that we didn't accomplish include castling as we ran out of time before the end of term. We wanted the gantry to be able to calibrate itself, but we were unable to implement limit switches in time. An automatic calibration program for the computer vision's colours would have also been ideal so that the board could be easily used under different lighting conditions.

Rationale

Our project was a combination of many subsystems and the complexity of the project overall made for an interesting challenge. Making this board on such a tight deadline was very stressful but also very rewarding, especially because in the end, it worked and was playable.

We found that there are automatic chess boards on the market, most notably SquareOff and Phantom on kickstarter but these don't use computer vision. Admittedly, from a product perspective, a board that uses sensors to detect piece position instead of a camera over the board is better as it blocks the board less. However, the additional challenge from the computer vision made our project more interesting.

Functional Requirements

The functional requirements for the chess board are tabled in Table 1 below:

Table 1 - Functional Requirements

Functional Requirements	**Responsible Person**
Mechanical Design	Liam
Motor Control of Gantry	Willem
Gripping/Engaging Pieces	Willem/Liam
Sensing Pieces with Camera	Willem
Chess Software	Willem

The mechanical design of the board includes all the physical parts of it including:

- The board itself including framing
- The pieces
- The gantry system, including pulleys, motor mounts, and electromagnet holding apparatus
- The overhead camera mount
- The electronics housing for the power supply, stepper drivers and MSP board

Functional Requirement #1: Mechanical Design

Approach and Design

The objective of the mechanical design is a physical board that allows the system to operate smoothly. Friction and complexity should be minimized. A full CAD model of the board was made in SolidWorks to ensure the board's cohesiveness.

The board is a ⅛" thick piece of 2'x2' hardboard. The game grid is made of 50mm x 50mm squares, spray-painted in white and centered in the middle of the board. There must be enough space between pieces on the board for the pieces to move between each other, especially for movement of the knights. The board is slotted into 4 2x4s that have a small groove cut into them and these 2x4s are screwed into each others' ends to keep the board constrained. The gantry system, motor and pulley mounts, and electronics housing are all mounted to the 2x4s. The board is pictured below in Figure 2 from a bird's-eye perspective. For consistency the sides of the board will be referred to following the displayed convention.

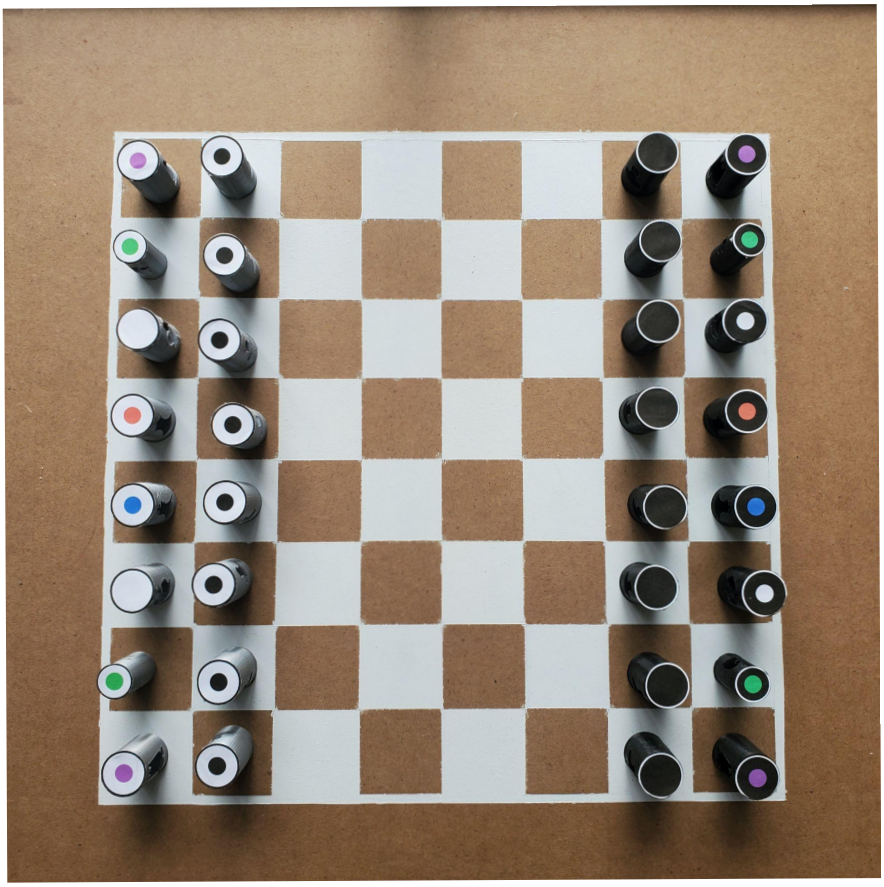

Figure 2 - Board seen from above

The pieces are uniformly sized 3D printed cylinders with a hole passing through the middle in the shape of the piece type. The pieces are pictured in Figure 3 below:

Figure 3 - Chess Pieces

The knights are slightly smaller in diameter to give them more clearance to pass between pieces when they move. Each side is printed in a different colour, but the pieces each have a multi-coloured paper circle taped to the top to aid with image processing. This is explained in more detail in the FR4 Section.

A small magnet is embedded in the base of each chess piece for the electromagnet to grip from underneath the board. Adhesive PTFE circles are stuck to the bottom of each piece to reduce friction and therefore the magnetic force required to move the piece.

The 2-axis gantry uses a core x-y motion system as it reduces the required torque to move pieces and is a more compact solution than a cartesian x-y. A underside picture and a screenshot of the board's CAD showing the basic design is pictured in Figure 4:

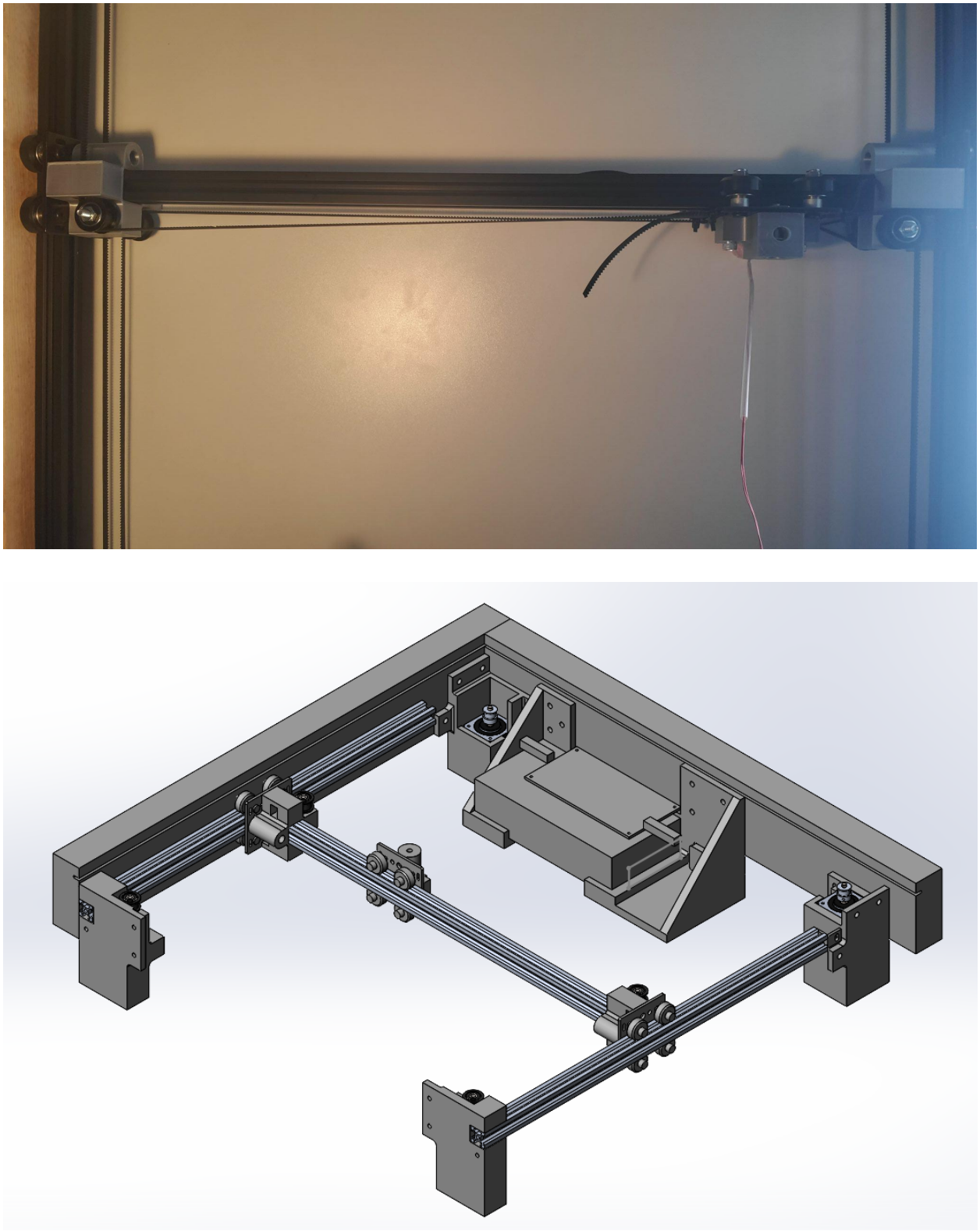

Figure 4 - 2-axis Core X-Y Gantry

Pictured in the screenshot above is also the electronics housing which holds the 24V power supply and the MSP board on top of it. The power supply plug and switch are also embedded into the housing and stick out the side of the board for easy access.

The electromagnet is the same coil used in the MECH 2 maglev project. As shown in Figure 5, it is mounted on a bolt in a 3D printed housing which is screwed onto a v-slot extrusion roller carriage on the moving stage of the gantry.

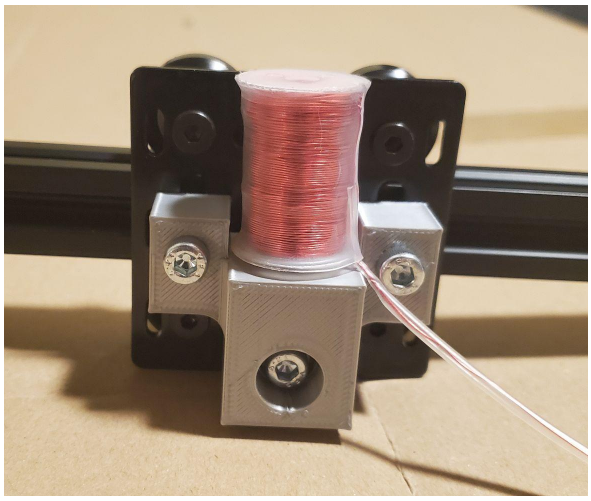

Figure 5 - Electromagnet mount

Timing belts are tied to this carriage which are routed to pulleys on two 3D printed end caps on the ends of the moving stage. The 3D printed end caps are mounted to their own v-slot extrusion roller carriages. From these pulleys, the timing belts are routed to the corner posts and then to the two NEMA 17 stepper motors that control the gantry. The motors sit in 3D printed housings in the opposite corners that also double as their own corner posts.

Figure 6 - Example 3D prints of end caps and motor posts

The camera we used is a Microsoft webcam that is mounted above the board using aluminum extrusion as shown below:

Figure 7 - Camera Mounting

Inputs and Outputs

The inputs of the mechanical design and their corresponding outputs are listed below. Many are dimensional and therefore were adjusted in CAD instead of writing equations to set them. Many others were determined with rapid prototyping rather than trying to quantify the non-linear magnetic relationships between the permanent magnets and the electromagnet.

- Diameter of embedded magnet → chess piece diameter → square side length → chess grid size → required camera height to capture entire board
- Diameter of electromagnet → chess piece diameter
- Chess piece height → computer vision consistency (discussed more in FR4 section)
- Board thickness → required electromagnet strength (discussed more in FR3 section)
- Friction in gantry → required motor torque
- Height of gantry → overall height of the board
- Size of electronics (power supply, stepper drivers) → overall height of the board

The embedded magnets are ¼” diameter x 1/10” tall rare-earth magnets from Lee Valley Tools. From testing with 3D prints, a chess piece diameter of 25mm as it is large enough for the electromagnet to be unable to grab multiple pieces at the same time. Since the knights must be able to move between pieces frequently, they were made 20mm in diameter.

When chess pieces are taken by the AI, the taken pieces must be moved between others to the edge of the board, so there must be sufficient space between pieces for taken pieces to pass between. For that reason, a square side length of 50mm was chosen and this results in a grid size of 400x400mm.

Parameters

The chess piece height was an important factor in the computer vision’s detection of pieces. The taller the pieces are, the more they misalign with the squares due to the fish-eye effect of the webcam’s lens. Height was therefore limited to 40mm as that is the maximum height that the camera would still detect when the piece was in the corner.

Underneath the board, between the gantry system and the framing is a pocket of space to house the electronics. This was adjusted based on the size of the power supply and stepper drivers which were decided as the board was designed.

The board thickness parameter affects the electromagnet's ability to grip pieces and was determined through testing with the electromagnet and the permanent magnets for the chess pieces.

Testing and Results

To determine the chess piece diameter, small discs of various sizes were printed and magnets were embedded in them. The test prints were placed next to each other to see at what diameter the pieces started to interfere with each other. This was found to be below 20mm so a chess piece diameter of 25mm was chosen.

We initially tested the electromagnet with ¼" thick cardboard, and realizing the electromagnet was plenty strong enough to move the chess pieces through it, a 24" x 24" x 1/8" piece of hardboard was purchased as the board. After performing more tests moving pieces around through the board, it was clear that the electromagnet was more than capable for our application.

Functional Requirement #2: Motor Control of Gantry

Approach and Design

The goal of this functional requirement was to develop the hardware and firmware for the control of the core-xy gantry. The control would need to go to specified X Y locations at a certain set speed.

Motor and Motor Driver Hardware

The motors that were chosen for this gantry were 24V 2A stepper motors. These higher voltage motors were chosen as the system was going to have a higher amount of inertia and drag. The motors from the lab were considered, but due to the lower voltage and need to duty cycle the motor driver output that made smooth control in Lab 3 difficult they were decided against.

The MSP430 that will be used to control the motors will be the one on the green board as it has the faster baud rate, and the built in motor driver which will be used for the electromagnet.

After choosing the 24V 2A stepper motors a new driver and power supply were needed. For power supply the 24V 15A power supplies commonly used in 3D printers was chosen. This was chosen as it has enough current for two 2A motors to run, and the type of power supply was easily on hand (from an existing 3D printer) and had a relatively small form factor.

Figure 8 - 24V Power Supply

Originally A4988 stepper motor drivers were chosen as they were small and could be easily placed under the board. The A4988 motor drivers are rated up to 2A and since we were going to be micro stepping the motors the current draw shouldn't be up to the 2A. The circuit diagram for the A4988 drivers is shown below.

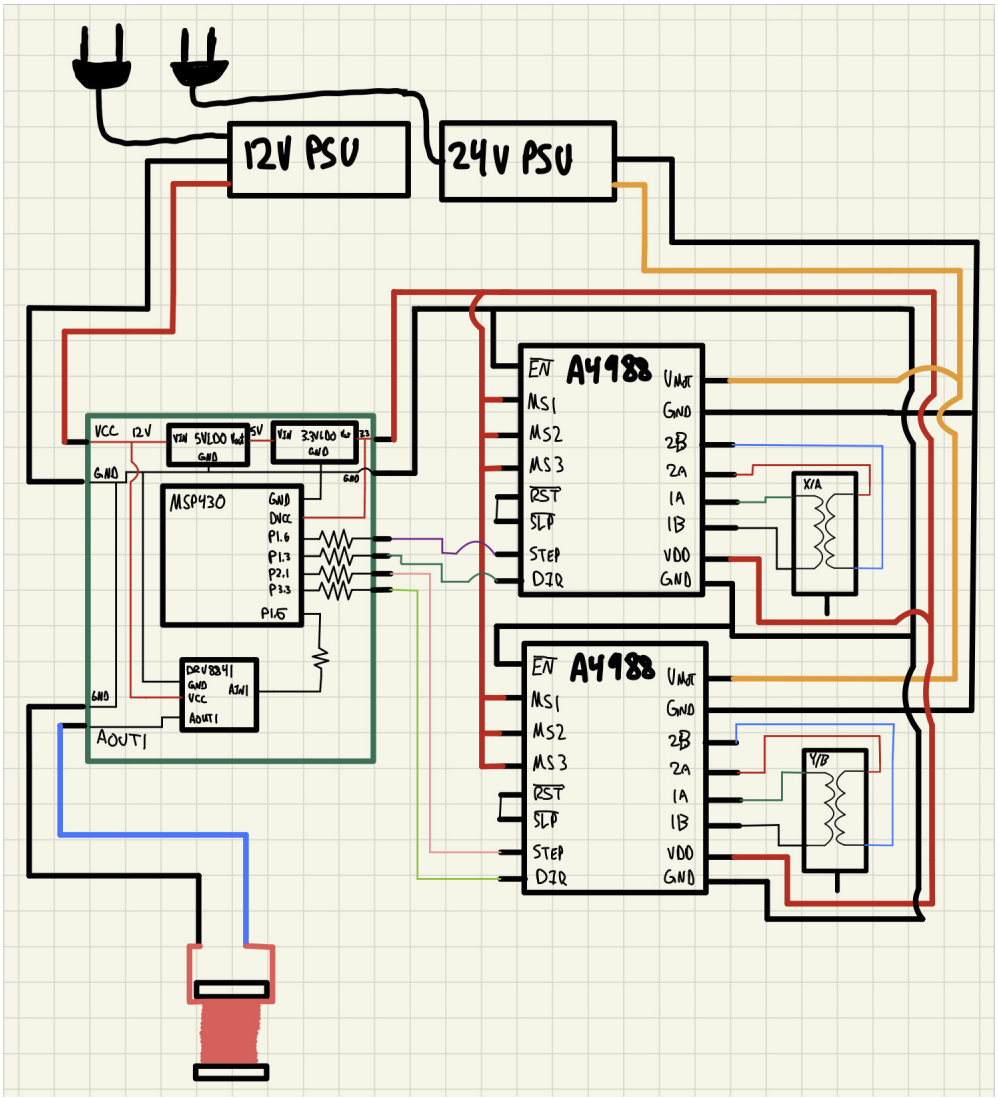

Figure 9 - A4988 Motor Driver Schematic

The circuitry was very simple as the GPIO used for stepping and direction of the motor drivers have built in resistors. The motor driver MS pins to choose the microstepping were all pulled high to select 1/16 step microstepping. The Enable, Reset, and Sleep pins were pulled low with either internal circuitry or tied to ground. The 24V signal was supplied to Vmotor and the 3.3V power from the green MSP430 board was supplied to the VDD of the motor driver. The stepper motor wires were connected to the output pins. The circuitry was made on a small protoboard as shown below.

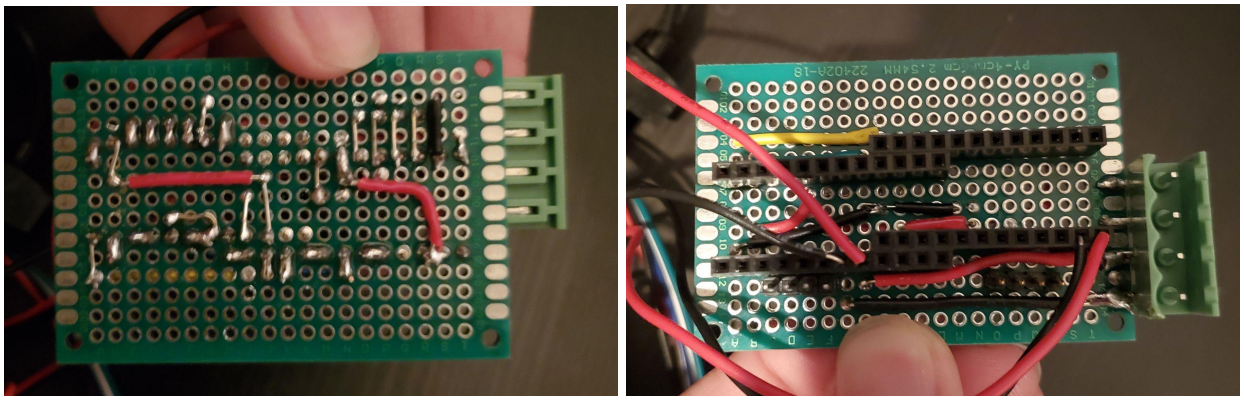

Figure 10 - Protoboard

However, while testing the full assembly one of the motor drivers stopped working at the last minute. The suspicion is that the motor driver burnt out due to too much current, or a short on the protoboard. As we were limited on time we decided to switch to the DM542T motor drivers. These are much more robust motor drivers that have higher current ratings, and more internal current limits and error detection. The downsides of them are the price (but we already had them on hand) and the size of the drivers. The new motor drivers had the same connections in terms of Step and Direction pins, and connection outputs to the motors. The only difference was that the microstepping and current limiting is set by DIP switches on the drivers. The same schematic as Figure 9 except with DM542T drivers and no connections for RST, SLP, or the MS pins is equivalent.

These motor drivers were wired up using a small breadboard as the only thing needed were connecting wires to the motor and MSP430. Ideally this would be transferred to a protoboard later. The picture of the final motor driver packaging and wiring is shown below.

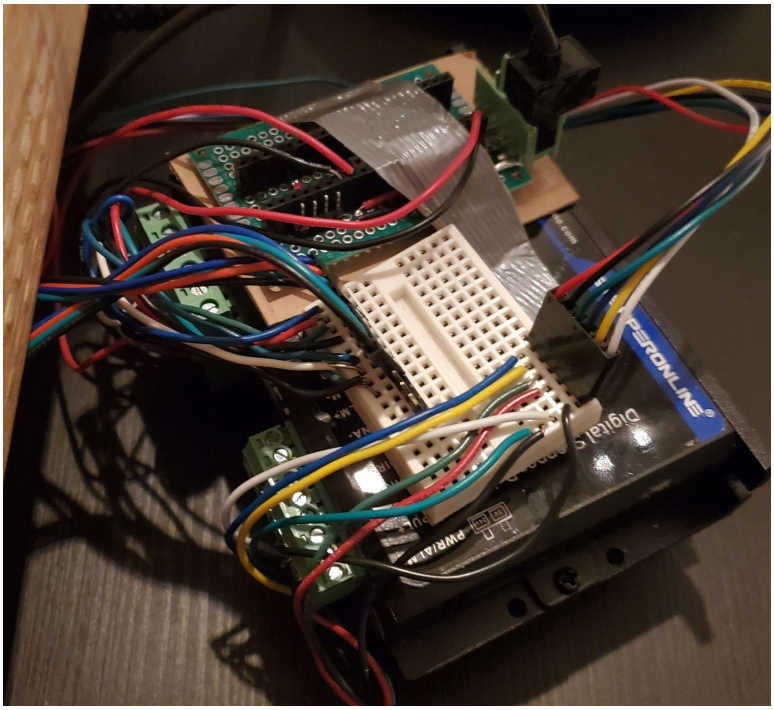

Figure 11 - Stepper driver package

All of the hardware was mounted to the bottom black side of the board to allow room for the gantry.

Motor Driver Firmware

The MSP430 needs to be able to communicate X Y locations over UART along with information on whether the magnet should be on or off for the movement. For this purpose a packet design similar to Lab 3 but with 7 bytes of data was used with the following packet format.

255	Instruction 0 1 2 no magnet magnet calibrate	X Loc MSB 0x00 → 0xFF	X Loc LSB 0x00 → 0xFF	Y Loc MSB 0x00 → 0xFF	Y Loc LSB 0x00 → 0xFF	Decoder Byte _ _ _ _ b

Figure 12. Packet Format for UART Communication

This packet format allows 16 bit values for the X and Y locations. The value for the X and Y locations count the number of microsteps to get from the 0,0 point to the location goal on the board.

Since the gantry is actually a corexy gantry to move in the X direction both motors will need to move. This requires a simple transformation of the X Y coordinates

to A B coordinates. This transformation was done in the software side, so the X Y coordinates being sent to and processed by the firmware is actually the AB coordinates.

One advantage of the chess game is that all pieces except the knight will move in either a straight line in the X or Y direction, or directly 45 degrees. Since the knight movement will be broken down into straight segments the timing of the X and Y motor movement won't need to be considered as they will both arrive at the same time.

The following code shows the initialization of global variables and parameters in the firmware:

```
#include <msp430.h>


#define bufferSize 150              // Buffer size for UART receiving

// UART Variables
unsigned volatile int circBuffer[bufferSize];                   // For storing received data packets
unsigned volatile int head = 0;                                 // circBuffer head
unsigned volatile int tail = 0;                                 // circBuffer tail
unsigned volatile int length = 0;                               // circBuffer length
unsigned volatile int rxByte = 0;                               // Temporary variable for storing each received byte
volatile int rxFlag = 0;                                        // Received data flag, triggered when a packet is received
volatile int rxIndex = 0;                                       // Counts bytes in data packet

// Control Variables and constants
unsigned volatile int xSpeed = 2000;
unsigned volatile int ySpeed = 2000;
unsigned volatile int xLoc = 0;
unsigned volatile int yLoc = 0;
unsigned volatile int xr = 0;
unsigned volatile int yr = 0;
volatile int xError = 0;
volatile int yError = 0;
unsigned volatile int magnetState = 0;
unsigned volatile int movingState = 0;
unsigned volatile int calibrationState = 0;
```

Figure 13. Firmware Parameters and Global Variables

The following code is the initialization of the clock rate, the timers for the X and Y motors, the magnet pins, UART, and local variables for UART.

```
// Main Loop for initialization, reading of UART buffer, and starting commands
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    // Configure Clocks
    CSCTL0 = 0xA500;                          // Write password to modify CS registers
    CSCTL1 = DCORSEL;                         // DCO = 16 MHz
    CSCTL2 |= SELM_3 + SELS_3 + SELA_3;       // MCLK = DCO, ACLK = DCO, SMCLK = DCO
    CSCTL3 |= DIVS_5;                         // Set divider for SMCLK (/32) -> SMCLK 500kHz

    // Configure pins for Magnet
    P1DIR |= BIT4;
    P1OUT &= ~(BIT4);

    // Configure timer B1 for X Stepper Motor
    TB1CTL |= TBSSEL_1 + TBIE + MC_1 + TBCLR;
    TB1CCTL1 |= OUTMOD_3;
    TB1CCR0 = 0;
    TB1CCR1 = xSpeed/2;

    // Configure Pin for B1 timer (X Stepper Motor)
    P1DIR |= BIT6;
    P1SEL0 |= BIT6;
    P1SEL1 &= ~BIT6;

    // Configure Pin for X Stepper Direction
    P1DIR |= BIT3;

    // Configure timer B2 for Y Stepper Motor
    TB2CTL |= TBSSEL_1 + MC_1 + TBIE + TBCLR;
    TB2CCTL1 |= OUTMOD_3;
    TB2CCR0 = 0;
    TB2CCR1 = ySpeed/2;

    // Configure Pin for B2 timer (Y Stepper Motor)
    P2DIR |= BIT1;
    P2SEL0 |= BIT1;
    P2SEL1 &= ~BIT1;

    // Configure Pin for Y Stepper Direction
    P3DIR |= BIT3;

    // Initialize Limit Switches
    P3DIR &= ~(BIT5 + BIT6); // X Axis and Y Axis Respectively
    P3REN |= BIT5 + BIT6; // Input w/ Pullup or down set
    P3OUT &= ~(BIT5 + BIT6); // Pull Down Resistor set
```

```
    // Initialize Limit Switch Interrupts
    P3IES |= BIT5 + BIT6; // Set on High to Low Transition
    P3IE |= BIT5 + BIT6;

    // Configure ports for UART
    P2SEL0 &= ~(BIT5 + BIT6);
    P2SEL1 |= BIT5 + BIT6;

    // Configure UART
    UCA1CTLW0 |= UCSSEL0;
    UCA1MCTLW = UCOS16 + UCBRF0 + 0x4900;   // Define UART as 19200baud rate
    UCA1BRW = 52;
    UCA1CTLW0 &= ~UCSWRST;
    UCA1IE |= UCRXIE;                       //enable UART receive interrupt
    _EINT();                                //Global interrupt enable

    // Circular Buffer Data Processing Variables
    unsigned volatile int commandByte, dataByte1, dataByte2, dataByte3, dataByte4, escapeByte, firstDataByte, secondDataByte;
```

Figure 14. Setup Code for Firmware

The UART receiving handling was done in the UART receive interrupt and added to a circular buffer as shown below.

```
// UART interrupt to fill receive buffer with data sent from C# program
#pragma vector = USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)
{
    rxByte = UCA1RXBUF;                 // rxByte gets the received byte

    // Check if 255 was received
    if (rxByte == 255 || rxIndex > 0)
    {
        // Check that the buffer isn't full
        if (length < bufferSize)
        {
            circBuffer[head] = rxByte;      // Buffer gets received byte at head
            length++;                       // Increment length

            if (head == bufferSize) { head = 0; }   // Check if head at end of buffer and if so put it at start
            else { head++; }                // Else, increment head

            // Check if receiving index is 6 or greater and if so reset
            if (rxIndex >= 6)
            {
                rxIndex = 0;                // Reset receiving index
                rxFlag = 1;                 // Set the data received flag
            }
            else { rxIndex++; }             // Increment rxIndex
        }
    }
}

```

Figure 15. UART Receive Interrupt

Sending of messages from the UART was done with a transmit message function that decoded the data bytes and sent it in the correct format. The messages that are sent back to the computer are done to indicate that the current instruction has finished, and the next location is ready to be received. This is seen in the code below.

```
// Function to transmit a UART package given arguments for package
void transmitPackage(unsigned int instrByte, unsigned int dataByte1, unsigned int dataByte2, unsigned int dataByte3, unsigned int dataByte4){
    unsigned int decoderByte = 0;
    if (dataByte1 == 255){
        decoderByte |= 8;
        dataByte1 = 0;
    }
    if (dataByte2 == 255){
        decoderByte |= 4;
        dataByte2 = 0;
    }
    if (dataByte3 == 255){
        decoderByte |= 2;
        dataByte3 = 0;
    }
    if (dataByte4 == 255){
        decoderByte |= 1;
        dataByte4 = 1;
    }
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = 255;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = instrByte;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = dataByte1;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = dataByte2;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = dataByte3;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = dataByte4;
    while (!(UCA1IFG & UCTXIFG));
    UCA1TXBUF = decoderByte;
    while (!(UCA1IFG & UCTXIFG));
}
```

Figure 16. Transmit over UART Firmware

The circular buffer with the UART messages was then processed in the main loop, and depending on the instruction byte different instructions were executed using a case switch statement.

```
131     while (1)
132     {
133         if (rxFlag)
134         {
135             // Get escape byte and command byte from buffer
136             escapeByte = circBuffer[head - 1];
137             commandByte = circBuffer[head - 6];
138 
139             // Handle the Data Bytes
140             // Check if the first bit of escape byte is 1 and if so set dataByte4 to 255
141             if (escapeByte & 1) { dataByte4 = 255; }
142             // Else, dataByte4 gets the value from the buffer
143             else { dataByte4 = circBuffer[head - 2]; }
144             // Check if the second bit of escape byte is 1 and if so set dataByte3 to 255
145             if (escapeByte & 2) { dataByte3 = 255; }
146             // Else, dataByte3 gets the value from the buffer
147             else { dataByte3 = circBuffer[head - 3]; }
148             // Check if the third bit of escape byte is 1 and if so set dataByte2 to 255
149             if (escapeByte & 4) { dataByte2 = 255; }
150             // Else, dataByte2 gets the value from the buffer
151             else { dataByte2 = circBuffer[head - 4]; }
152             // Check if the fourth bit of escape byte is 1 and if so set dataByte1 to 255
153             if (escapeByte & 8) { dataByte1 = 255; }
154             // Else, dataByte1 gets the value from the buffer
155             else { dataByte1 = circBuffer[head - 5]; }
156 
157             // DataByte gets the combination of dataByte1 & dataByte2
158             firstDataByte = (dataByte1 << 8) + dataByte2;
159             secondDataByte = (dataByte3 << 8) + dataByte4;
160 
```

Figure 17. Decoding of Data Packet

```
162             // Handle the command Bytes
163             switch(commandByte)
164             {
165             case 0: // Move with Magnet Off
166                 // Turn off magnet P1.4
167                 P1OUT &= ~BIT4;
168                 magnetState = 0;
169                 // Set new XY goal position
170                 xr = firstDataByte;
171                 yr = secondDataByte;
172                 movingState = 1;
173                 // Check initial x and y error to only start motors that are necessary
174                 xError = xr-xLoc;
175                 yError = yr-yLoc;
176                 if (xError != 0){
177                     TB1CCR0 = xSpeed;
178                 }
179                 if (yError != 0){
180                     TB2CCR0 = ySpeed;
181                 }
182                 break;
183             case 1: // Move with Magnet On
184                 // Turn on magnet P1.4
185                 P1OUT |= BIT4;
186                 magnetState = 1;
187                 // Set new XY goal position
188                 xr = firstDataByte;
189                 yr = secondDataByte;
190                 movingState = 1;
191                 // Check initial x and y error to only start motors that are necessary
192                 xError = xr-xLoc;
193                 yError = yr-yLoc;
194                 if (xError != 0){
195                     TB1CCR0 = xSpeed;
196                 }
197                 if (yError != 0){
198                     TB2CCR0 = ySpeed;
199                 }
200                 break;
```

Figure 18. Execution of Magnet Off and On Instructions

The received X and Y locations are saved in a reference value variable, and the X and/or Y motors are started as necessary.

The timers for the motor step pulses were set to execute an interrupt on each pulse to increment or decrement the location tracking variables and set the direction of motion, or stop the motor if the error between the reference location and actual location is 0. There is an interrupt for the X and Y axis motors.

```
// Timer B1 Overflow Interrupt: Increment X Stepper
// Triggers on each timer/step pulse
#pragma vector = TIMER1_B1_VECTOR
__interrupt void IncrementXStepper(void){
    // If flag for moving is set
    if (movingState){
        // Calculate error, set direction, and inc or dec Location counter
        xError = xr - xLoc;
            if (xError == 0){
                TB1CCR0 = 0;
            }
            else if (xError > 0){
                xLoc++;
                P1OUT |= BIT3;
            }
            else if (xError < 0){
                xLoc--;
                P1OUT &= ~BIT3;
            }
    }
    TB1CTL &= ~TBIFG;
}

// Timer B2 Overflow Interrupt: Increment Y Stepper
// Triggers on each timer/step pulse
#pragma vector = TIMER2_B1_VECTOR
__interrupt void IncrementYStepper(void){
    // If flag for moving is set
    if (movingState){
        // Calculate error, set direction, and inc or dec Location counter
        yError = yr - yLoc;
           if (yError == 0){
                TB2CCR0 = 0;
            }
            else if (yError > 0){
                yLoc++;
                P3OUT |= BIT3;
            }
            else if (yError < 0){
                yLoc--;
                P3OUT &= ~BIT3;
            }
    }
    TB2CTL &= ~TBIFG;
}
```

Figure 19. Stepper Timer Interrupt Functions

Although not a part of the original functional requirement, a method of automatically calibrating the motors on startup, or whenever necessary was implemented into firmware. However, the mechanical design wasn't able to fit the limit switches necessary so the code wasn't tested or implemented in the final design. The process for calibration was if the interrupt for a GPIO pin that the limit switches were connected to was started, the motors would stop, and the corresponding location that the limit switch was mounted at would be set to current location value. The initial code is shown below, although not tested. Not attaching the limit switches does not affect the execution of the control code, so the limit switches are optional if calibration isn't needed. Calibration would have been set to instruction 2 and could be triggered by the software.

```
201             case 2: // Zero Steppers Not fully implemented but not fully necessary
202                 calibrationState = 1;
203                 while(calibrationState == 1){
204                     P3OUT |= BIT3;
205                     P1OUT |= BIT3;
206                     TB1CCR0 = xSpeed;
207                 }
208                 xr = 1000; // Might not need these parts if only triggers on falling edge
209                 movingState = 1;
210                 TB1CCR0 = xSpeed;
211                 calibrationState = 1;
212                 while(calibrationState == 1){
213                     P3OUT &= ~BIT3;
214                     P1OUT &= ~BIT3;
215                     TB2CCR0 = xSpeed;
216                 }
217                 yr = 1000;
218                 movingState = 1;
219                 TB1CCR0 = xSpeed;
220                 transmitPackage(2, xLoc>>8,xLoc&0xFF,yLoc>>8,yLoc&0xFF);
221                 break;
```

Figure 20. Calibration Instruction

```
321 // Limit Switch Interrupt
322 // Not fully implemented as limit switches didn't fit in mechanical design
323 #pragma vector = PORT3_VECTOR
324 __interrupt void SwitchToggle (void){
325     // If limit switch GPIO is brought low stop both motors, and set corresponding motor to location values
326     TB1CCR0 = 0;
327     TB2CCR0 = 0;
328     calibrationState = 0;
329     if (P3IV & BIT5){
330         xLoc = 0; // X Location of Limit Switch
331         P3IFG &= ~BIT5;
332     }
333     if (P3IV & BIT6){
334         yLoc = 4000; // Y location of limit switch
335         P3IFG &= ~BIT6;
336     }
337 }
```

Figure 21. Limit Switch Interrupt

Inputs and Outputs

The high level inputs and outputs of this functional requirement are as follows:

Table 2. Inputs and Outputs of Motor Controller

Inputs	Outputs
24V Power to Motor Driver	Position of the Gantry by rotation of stepper motors
12V Power to MSP430 Green Board	Power State (12V/0V) of Electromagnet
Command Data Packet (see Figure 12)	

The lower level inputs and outputs are shown in Figure 9 and explained in the previous section such as the pulse and direction inputs to the motor driver.

Parameters

There were a handful of parameters that could be changed in the development of the motor controllers. However, all of the parameters were tuned by testing at different values until a valid value was found.

The motor speed could be changed by modifying the frequency of the pulses sent to the motor driver. The speed of the motors needed to be the same across both motors, so if one motor was slower than the other for some reason it could be tuned separately (although this was not necessary for us). The other constraint on the motor speed was the ability to stay connected to the magnet in the piece above while moving. After some slight changes, a speed of 32kHz was decided for each microstep.

Another parameter that could be changed is the number of microsteps. The microstepping affects how accurate the position could be, the smoothness of movement, and the torque provided. Our first priority was how smooth the movement was as jerky movements would lose the magnetic connection between the piece and the magnet. We also needed enough torque to not stall out, and enough accuracy to not notice any missed steps or positional errors sent by the software over the course of a game. We decided on 16 microsteps as it provided very smooth movement with enough torque and was more than accurate enough for positioning.

Another parameter that could be changed is a duty cycle for the 12V power given to the electromagnet. If 12V ended up being too high and resulted in overheating of the magnet we could PWM the output of the motor driver used for it and reduce the

effective voltage into the magnet. After testing we found that we needed the full 12V whenever the motor was on so the PWM implementation was removed to simplify the code and a simple 12V on off signal was used for the magnet.

Testing and Results

Although the full mechanical design was the last function to be completed there were some intermediate functional tests that could be done to validate the motor control.

One thing that could be tested was the general direction and length of movement from different commands. The exact positioning would not be easy to quantify but some tests were done using a serial communicator app from Lab 3 and giving small or large movement commands in the X or Y axis and seeing if the motor moves in the right direction and for a long or short amount of time. This could give some general debugging of whether the motors were functioning properly.

Another test that was done was probing the output of the motor driver used for the magnet to see if 12V was being sent for the right instructions.

Another thing that was tested with the magnet was sending a serial instruction to power the magnet on and check if the magnet would attract the pieces on the board. After testing this I found that as long as the magnet was around the center of the square it would attract only the piece within the same square and not other pieces. The exact positioning of the piece in the square was not critical which meant the complicated design of using a reed switch to detect where the piece actually was was unnecessary.

After the full mechanical design was finished the motor positions were able to be calibrated and more accurate positional testing was able to be done, but this was tested along with functional requirement 5 so the inputs to the test were the row and column instead of X Y location.

Functional Requirement #3: Gripping & Engaging Pieces

Approach and Design

The objective of this functional requirement was to ensure the electromagnet was strong enough to attract the chess pieces through the board with enough force to move them across it. The electromagnet also had to turn on or off at the right times when moving the piece or positioning itself before moves.

To achieve this, originally the electromagnet was to be controlled by a MOSFET, then we looked into using a PWM signal from an additional stepper motor driver chip to vary the strength as needed. This was abandoned in favour of the much simpler solution of grounding one lead and plugging the other into the A2 phase of the stepper motor driver on the green board. The A2 phase is set either high or low through port 1.4 on the MSP to deliver either 0V or 12V to the coil.

Inputs and Outputs

The inputs and corresponding outputs are listed below:

- Friction between pieces and board → acceleration felt by chess piece
- Thickness of the board → acceleration felt by chess piece
- Voltage applied to electromagnet → strength of electromagnetic field
- Electromagnet control byte → GPIO pin 1.4 high/low → phase A2 high/low → electromagnet on/off (12V/0V)

The needed magnetic force was first estimated according to the following diagram:

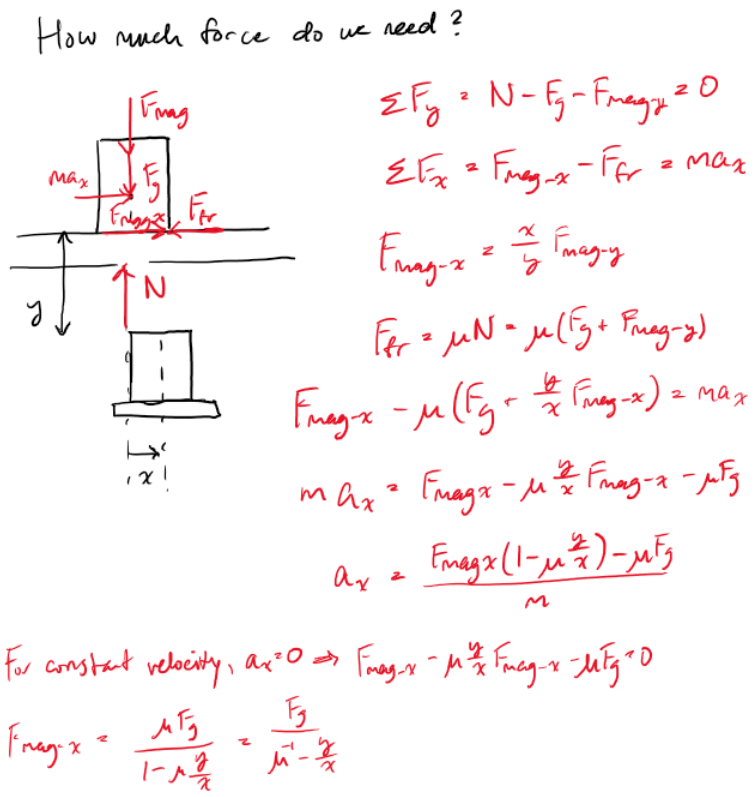

Figure 22- Electromagnet force estimation

From the relationships drawn above it was clear that the board thickness and the friction between pieces and the board needed to be minimized to maximize the acceleration produced by the magnet.

The electromagnet control byte is sent to the MSP at the same time as the motor control bytes as the instruction byte as shown in Figure 13 in the FR2 Section. The electromagnet is off if the byte is 0, and on if the byte is 1. The last instruction option (instruction byte = 2) doesn't change the electromagnet, i.e. it leaves the magnet in whatever state it was already in.

Parameters

The main parameter to be adjusted is the current through the coil as this adjusts the strength of the electromagnet field. However, as stated previously, varying the voltage was abandoned in favour of using the A2 phase of the MSP board.

Testing and Results

The electromagnet control was tested by sending the control packets with the instruction byte set or reset and seeing if the magnet would attract the chess piece through the board. The motor control bytes were left the same for these tests as the motors won't move if it's already at its destination. As this was a pretty simple test, it didn't require many trials to make it consistent.

Functional Requirement #4: Detecting Pieces and Player Move with Camera

Approach and Design

The objective of this functional requirement is to use computer vision and a webcam to detect the move that the player did. The main flow of how this works is to take a picture of the board before the player moves, and take a picture after the player moves. The images are then processed and compared to each other to determine where the piece that changed moved from and where it moved to. All of the source code is found in the src/ folder in the submission zip.

Five different classes were used, although three of the classes are enumerated type classes. The Piece class is an enumerated type that represents the different types of chess pieces, such as the king, queen, bishop, knight, rook, and pawn. The BoardCols class is an enumerated type that represents the different columns on a chess board, such as column A, B, C, etc. The Team class is an enumerated type that represents the two teams in a game of chess, white and black. The Board class represents the chess board itself and contains functions for capturing and analyzing images from the webcam, detecting the locations of the pieces on the board, determining the colors of the pieces, and returning the current state of the board as an array. The Game class is used to run the process of generating the board state before and after the player move, decipher the move that was taken, and forward the move on to the AI.

Board Class

The general usage of the Board class will be described as it is used by the Game Class. The high level overview of the Board class is that it is used to take the picture of the board, process the picture of the board into a layout in an array that is defined as a parameter of the class, and later convert row and column positions to A and B locations that are sent to the firmware by the Game Class. There are internal functions to the Board class that are used for breaking down the process of analyzing the image that are only called by a super method in the Board class and not used externally. The flow chart of the high level usage of the Board class by other classes is shown in the image below.

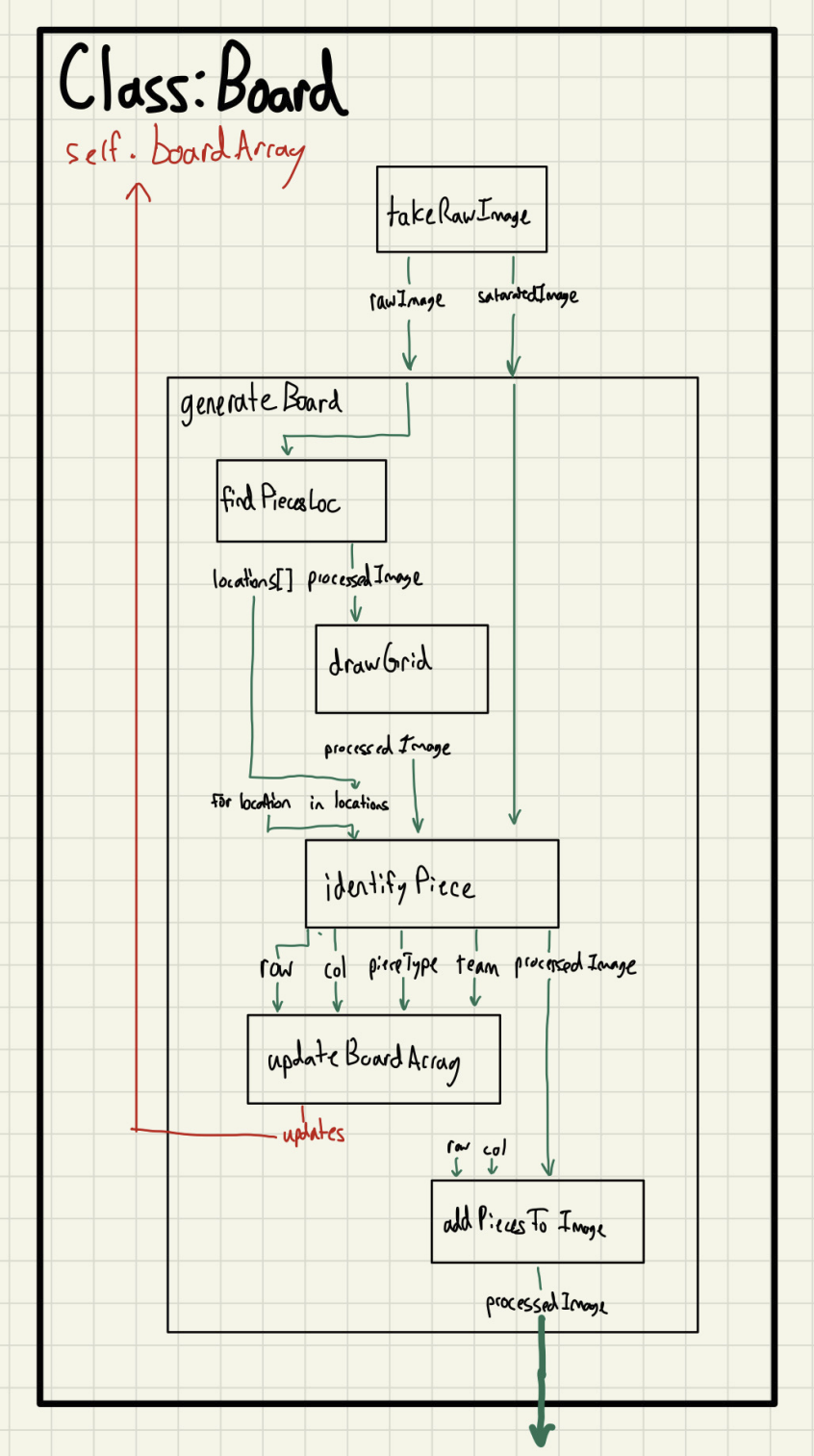

Figure 23. Board Class Chart

The takeRawImage method is called after initializing a new Board object which takes a picture using the webcam and makes a saturated image to be used later in color

recognition. The saturation is done by converting the image to HSV and increasing the saturation and decreasing the brightness. This value along with some other capture settings such as exposure can be tuned to improve the picture quality and recognition. Both the raw image and the saturated image are returned to be used later.

The generateBoard super method is then called which runs a series of internal functions to analyze the image taken.

The findPiecesLoc method uses HoughCircles to identify each circular piece. The method returns a list of the locations of the centers of each circle, and draws the circles and centers on the picture and returns this as a processed image to be used by later methods and viewed for debugging.

The processed image is then passed into the drawGrid method which draws the chess grid over the image to help with debugging. The parameters for the size of the grid and location are set in the initialization of the Board class and set based on the camera location.

A for loop in the generateBoard class then iterates through each location and identifies each piece, updates the boardArray, and adds the piece to the processed image.

The piece is identified by first finding the row and column that the pixel location corresponds to using the same constants that are used in drawing the grid. Then it detects the team by converting the image to gray scale and reading the color of a pixel offset from the center of the circle by 10 pixels in the X and Y direction. This will either be close to black, or close to white. The value of the piece (1 for white or -1 for black, from the enum) is then returned as the team value. The piece type is detected by checking for the color of the center pixel in the saturated image and checking it against the color ranges for each piece defined in the dictionary that is defined when the board is initialized. This value from the enum of the piece types is then saved as the pieceType variable. All of these values are then returned.

The boardArray is updated by finding the index of the piece location (using the row and column values) and writing the team enum value and piece type enum value to another array at that location. This makes the board array have 8 rows, 8 columns, and a depth of 2.

Lastly for each location the piece team letter (B or W) and the piece type letter, along with the color values for each center pixel are added onto the processed picture for debugging.

After all of the circle locations are processed the Game class continues on with the next steps. The processed image can also be viewed if enabled (for debugging) and an example of it is shown below.

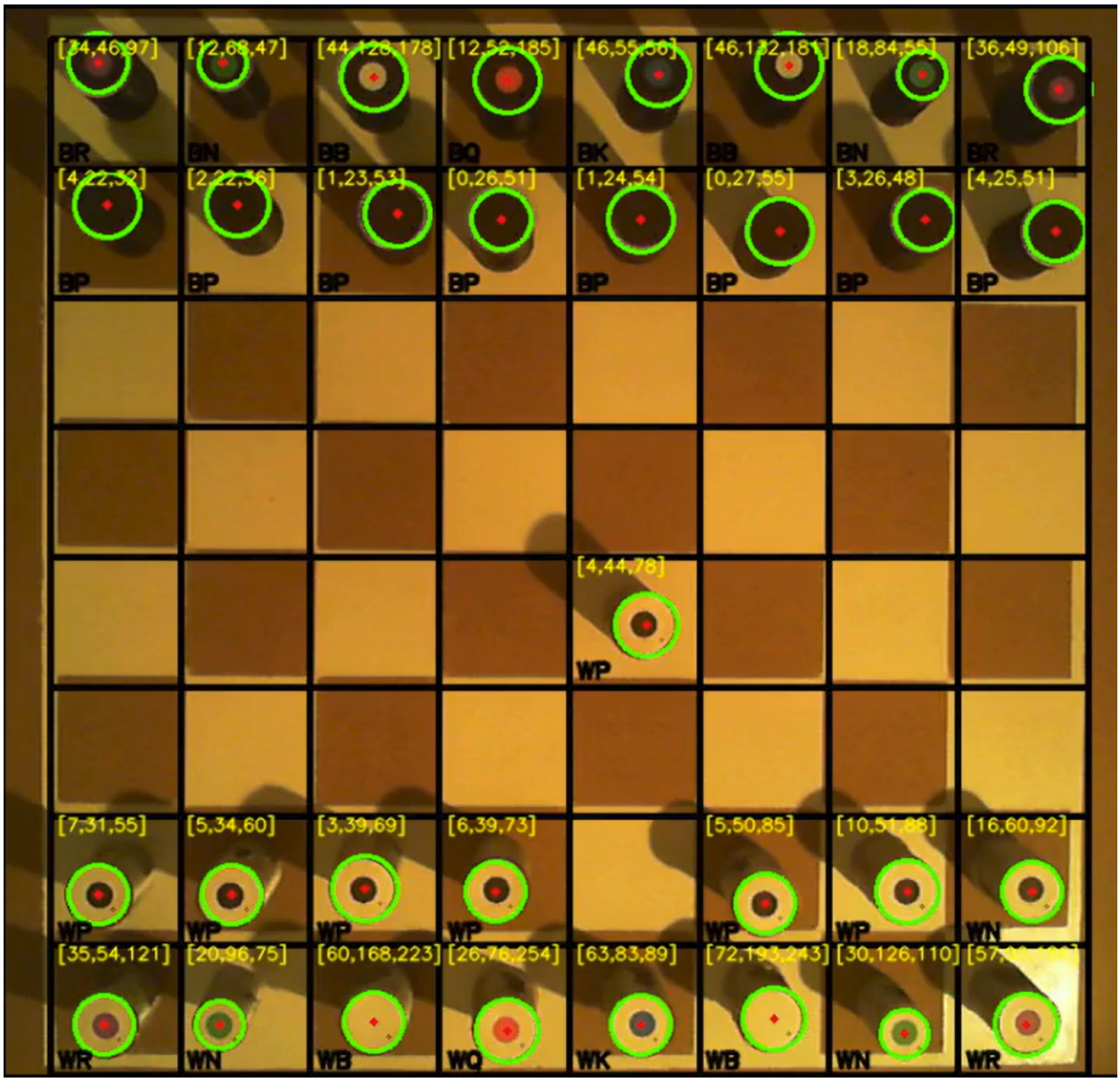

Figure 24. Example Processed Image

As seen in the above picture the grid is drawn on the image, every piece is recognized as a circle, the first letter of each square is the team, and the second letter indicates the type. In yellow at the top left of each square the BGR values for the piece color are printed. This helps when tuning the piece type color dictionary for different lightings. After it is tuned the recognition of each piece is quite consistent. Although as

can be seen later some error correcting was implemented to minimize the importance of correctly recognizing the type of unmoved pieces.

Game Class

The Board class is mostly used to take a snapshot of the current state of the board and record it in an easier to process array to detect the actual move that is made.

The Game class handles the coordination of taking the board snapshots, comparing the boardArrays, and returning a correctly formatted string in chess notation of the players move to be used by the Chess AI. The Game class also has some methods used when making the AI move, but those will be explained in Functional Requirement 5.

There are only two methods of the game class used in detecting the player move. These methods are called by the main.py file even though all of the classes explained so far were in the chess.py file. This process will be explained in the next functional requirement.

The first method that is called is captureBoard. This is the method that generates the boardArray as explained in the Board class above.

The other method is detectMove, which takes the two boardArrays (one from before the players move (preBoard) and one after (postBoard)) and identifies the move that was made. The main strategy behind detecting the move is to subtract the postBoard from the preBoard. Then using numpy's nonzero method a list of changed indices in the array is made. This changed indices list is a list of three arrays, one for the row, one for the column (both zero indexed), and one for the properties that changed (0 for team or 1 for type). An example array would be:

Table 3. Example Changed Indices List

Row	1	3	3	4	4	6	6
Column	1	3	3	4	4	4	4
Property	1	0	1	0	1	0	1

Each column of the above array is a set in itself. So the first "changed indices" is row 1 column 1 where the type of the piece changed (property 1). Let's take the example list above as the output of the changed indices from comparing the preBoard and postBoard pictures below.

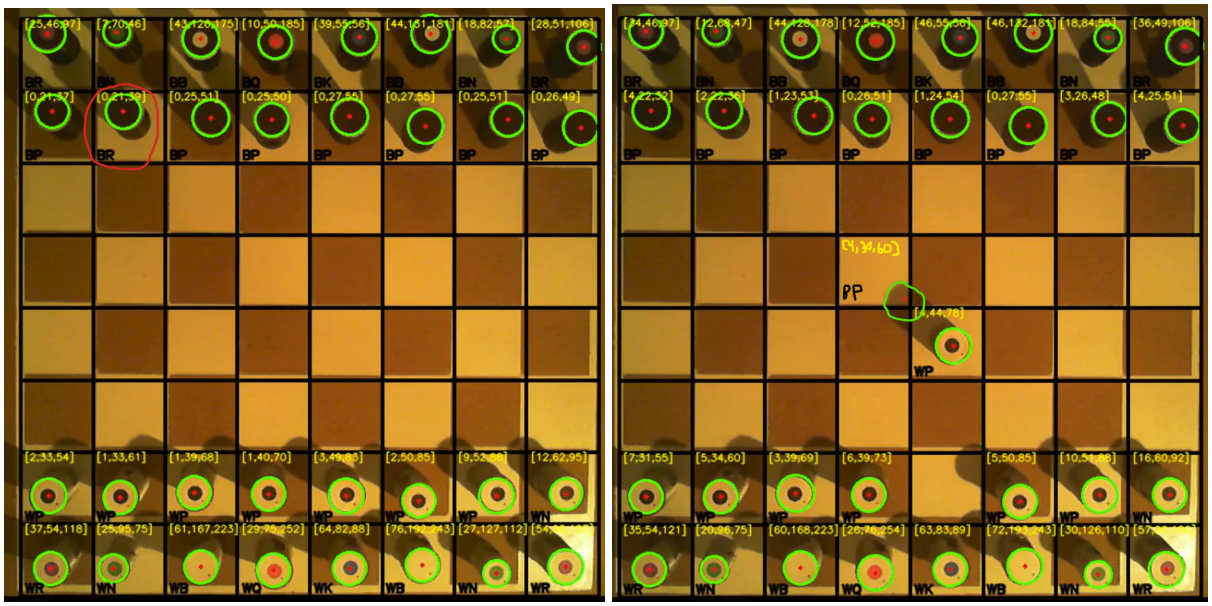

Figure 25. preBoard on Left and postBoard on Right

Notice that for the sake of this example the piece circled in red in the left picture is mis-identified as a black rook instead of a pawn, but in the postBoard on the right the piece is correctly identified as a black pawn. Also after the kings pawn is moved in the postBoard the camera accidentally identifies the end of the shadow of the white pawn as another circle where the center is just barely in the square to the top right of it. This would result in a black pawn, for example, being identified in that square. The array in the table above is the output that would result from this series of images.

The 0 index for the rows and columns are in the top left of the board. So the top left square would have a coordinate of [0,0] and the top right square would have a coordinate of [0,7]. As seen in the array square [1,1] has the type of piece property (1) changed between boards but not the team (property 0). While squares [3, 3],[4,4] and [6,4] all have had a team change and a type change. All squares that do not have labels in them in the images above are set as NULL team and NULL type.

The detectMove method first filters out all changed indices where only the type of piece changed. This is done by removing elements in the changed indices list where only the property 1 is set. This would filter out the first column, array square [1,1]. Since the team recognition is very consistent the main failures we had were the type of piece being misidentified. Filtering the squares where the team didn't change but the type did change as that always indicates a vision error. Since in chess you can not take one of your own pieces (same team but different type before and after the move). The only time this would be affected is if you are promoting a pawn, but this was not considered in the scope of this project.

The next filtering checks to make sure only 2 unique squares were changed during the move. If the shadow hadn't been detected as a circle the remaining two changed squares (where the piece moved from and where it moved to) would be the only squares detected and the method would move onto the next step. In this case since it recognized more than two changed squares it would prompt the user to input the move they did manually in string notation. This would be E2 E4 in the case of this move.

For the sake of explanation let's assume now that the vision did not detect the shadow as another piece. The changed indices array would look like this after filtering out the [1,1] mistake:

Table 4. Example Improved Changed Indices List after Filtering

Row	4	4	6	6
Column	4	4	4	4
Property	0	1	0	1

The final step in detecting the move is to see which of these changed squares [4,4] and [6,4] was the position before the move, and which was the position after the move. To do this the values of the team enum and the scope of the white pieces perspective is used. First to explain the values of the enums, if a square is NULL the team value is 0, if a square has a white piece on it the team value is 1, and if the piece is black the team value is -1. Of all of the permutations of a square changing only some of them are valid moves.

Table 5. Move Types

Validity	White Start Position - Valid	White End Position - Valid		Black Move - INVALID		
Pre Team	WHITE	NULL	BLACK	WHITE	NULL	BLACK
Post Team	NULL	WHITE	WHITE	BLACK	BLACK	NULL
Example	White moves away from square	White piece moves onto empty square	White captures black piece	Black captures white piece	Black moves to empty square	Black moves away from square
Pre - Post	1-0 = 1	0 - 1 = -1	-1 - 1 = -2	1- -1 = 2	0 - -1 = 1	-1 - 0 = -1

There is some repetition in the output of preBoard - postBoard value between black moves and white moves. But since only the white player moves, and the chances of the computer misrecognizing a valid black move and not recognizing a valid white move is low the preBoard - postBoard team value can indicate whether the square was the position before the move, or the position after the move. Anything with a subtracted value of 1 is the initial position and anything with a subtracted value of - 1 or -2 would be the final position. As of writing this I realize that a better method would be to check postBoard at one of the squares and see if the team value is 1 or 0, 1s would mean it is the final position and 0 would be the initial position. But the more convoluted method was implemented and never ran into an error.

The initial position and final position were then translated into the string format using the column letter and row number by taking the name of the enum for the column and concatenating the string. This string is what the main output of this functional requirement is as it is passed to the chess AI.

Inputs and Outputs

The high level inputs and outputs of this functional requirement are as follows:

Table 6. Inputs and Outputs of Piece and Move Detection

Inputs	Outputs
Board (subject in webcam picture)	String in format “A2 A4” with initial position and final position of piece moved

There are not many inputs or outputs to this function. When called by the chess game this functional requirement will take a picture of the board and output the string if possible, if not it will prompt the user for the string. One prerequisite for the board picture is that the lighting is sufficient. Some amount of this can be tuned with the parameters explained below.

Parameters

There were many parameters that could be changed and tuned in this functional requirement. Most of these parameters were used to modify the raw image in a way that is easier to identify.

The first parameters that could be changed were the inputs to the Hough circle command. These parameters were the inverse ratio of the accumulator, the minimum distance between circle centers, gradient for the detected edges, accumulator

threshold, minimum circle radius, and maximum circle radius. These were adjusted to only recognize the circles that were the sizes of the full piece, and the gradient was increased to prevent shadows from being recognized as pieces. The image inputted into the Hough circles function was also converted to gray scale to simplify the analysis.

Another important set of parameters were the image processing parameters. This included the exposure time, the saturation multiplier, and the brightness multiplier. The exposure time ended up being the best value to change depending on the amount of light in the room for improving the image quality and definition. The saturation could be increased in conditions where there was too much light of a certain shade to increase the difference between the piece colors. The piece colors also were chosen to be as different from each other as possible. In very bright or dark environments the brightness could be adjusted to reduce washout or blackouts.

Another set of parameters that needed to be tuned were the color ranges for the teams and piece types. The team color range was pretty static as there was seldom any misidentifying of a black or white piece. The color ranges of the type of piece however was very sensitive to the type of light in the environment. These parameters were tuned by looking at the processed image from the Board class and changing the max and min BGR values for each channel to only include the detected colors in the processed image. These values would need to be slightly modified before each session as natural light or composition of room lights were quite variable in our test environments. However, after the values were tuned they were very consistent throughout the game.

Testing and Results

Most of the testing for this functional requirement revolved around identifying the pieces correctly. This was one of the first things tested, before the pieces were even made. Initial tests were done with printouts of the colored circles on different colored backgrounds and with different localities to other pieces. This allowed the initial tuning and image processing to be tested and developed until the pieces could be recognized properly. Below is an example of the images I was working with.

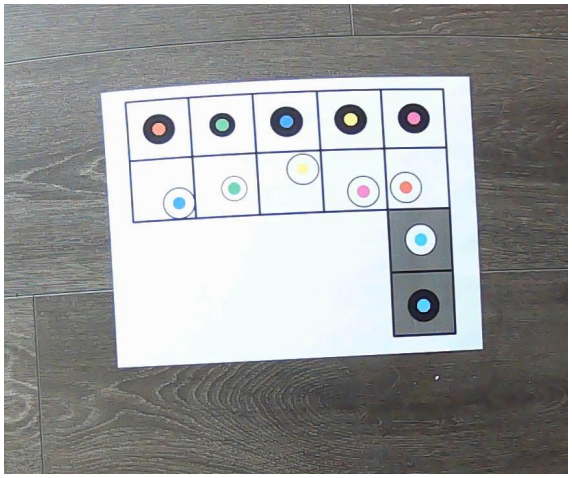

Figure 26. Color Testing Paper

Through this testing I modified the colors used in identification to increase the ability to differentiate between them. For example I changed the yellow pieces to white centers, and the lighter shade of blue to black type colors. I also tested with QR codes that could be tested in grayscale and not be as susceptible to the lighting of the environment.

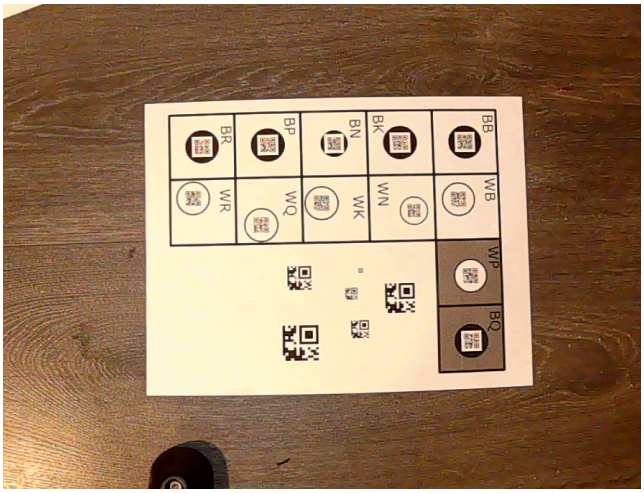

Figure 27. QR Code Testing

Unfortunately for the size of the pieces we were using and the height of the camera to see the full board the QR codes were too small to read. In the image above only the largest and second largest QR code were able to be detected. I even tried Japanese MicroQR codes but there was no python library available to decode them yet.

Another test that I did to try and tune the color recognition ranges was to record all of the colors and graph their intensities to figure out which colors were detected as the most unique. The results are shown below.

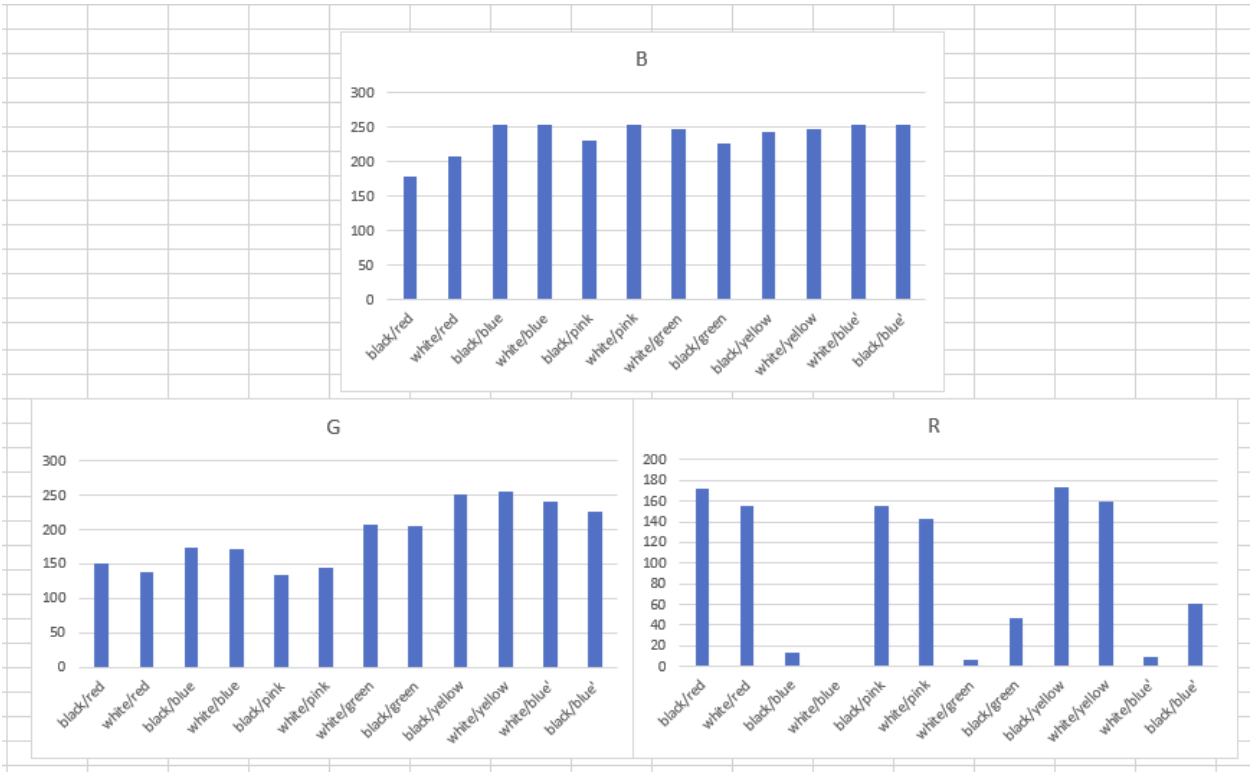

Figure 28. BGR Channels for Color Test

From these results I was able to deduce that the natural light in the environment was increasing the intensity of the blue and green channels but did not have much of a red element. This allowed the red adjacent colors (red, pink, and yellow) to be very different in the red channel than the blueish and greenish colors. However the blue channel proved difficulty to use to differentiate between colors. This is what led me to remove the light blue color. I was also able to compare two similar colors and see what their greatest differentiating feature was to compare in the software. In the end after figuring out how to change the exposure I was able to get more balanced color channels and just manually choosing the thresholds ended up being sufficient. But this test helped in the initial stages of development.

Functional Requirement #5: Chess Game and AI Software Integration

Approach and Design

The objective of this functional requirement is to implement the chess game and AI into the existing functional requirements that detect the players move, and can move the pieces with XY coordinates and magnet state commands. This will need to include the main logic of the chess game and the AI software. Since the goal of this project wasn't to code a chess game or write an AI ourselves the chess game and AI made by Dirk94 and made publicly available on GitHub (https://github.com/Dirk94/ChessAI) was used. This software consisted of four files, main.py, ai.py, board.py, and piece.py. The only file that I modified was the main.py file. The chess.py file in the source code was wholly written by me. This is the file that you run to play the magic chess game. The existing software would play through a game by printing the chess board to the command line and prompting the user for a move in the notation A2 A4 (for example). This would move the pawn in A2 to A4 (if it was the first move). The chess AI would then choose the best move (treated as a black box for our project), and would make an ai_move object that would contain the position the AI piece moved from and where it would move to.

There were two main points that my code would need to integrate into: generating the user move string using the player move detection code from functional requirement 4, and generating a list of coreXY trajectories with magnet state that will execute the move that the AI has chosen. The rest of the game was left mostly untouched. The main.py file and chess.py file can be found in the src/ folder of the submission.

Player Move Integration

This was implemented into the main function by first creating a new Game object from the chess.py file. I took out where the normal game would prompt the player for a move and instead run the Game class method of generating the preBoard. Then it prints a prompt for the player to make their move and hit enter. Then it calls the Game class method to generate the postBoard and detect the move using those two boards. This string is then substituted with the variable that the user inputted string would have been stored in and the game continues as normal. One feature of the chess game is that it will detect if the move was an illegal move. In our case this is the place where the validity of the move according to chess rules is checked. If the move was invalid it will prompt the user to manually enter their move as they would have before.

AI Move Integration

The normal chess game will generate an AI move by passing 0 indexed row and columns that indicate the initial position of the piece, and which is the final position of the piece. This ai_move object is passed into a method in the Game class in chess.py that will create a list of movements that the firmware will need to handle to properly move the piece.

In the Game class there is an initially empty list (moveArray) that buffers the full sequence of moves that the chess board will need to execute. Each move will be a single straight line either with the magnet on or off. The moveArray is populated with a Move class which has a message variable that is in the data packet format that is sent over serial.

To convert the row and column coordinate system location to the 16 bit microstep coreXY A and B coordinate system a method in the Board class is used that multiplies the position value by the pitch of the timing belt, the teeth on the pinion, the 16 step microstepping, and the mm dimensions of the board. This is done in a similar method to Lab 3. The main difference is that the X and Y coordinate system is translated to an A B coordinate system used in CoreXY where A = X + Y and B = X - Y. This would normally result in A being in the range of 0 and maxBoardStepCount and B would be between -maxBoardStepCount and +maxBoardStepCount. This is a problem as the 16 bit locations transferred to firmware is not signed. Therefore the B value is offset to also be within the range of 0 and maxBoardStepCount. The position of the gantry is calibrated by turning off the motors and sending the current position of the gantry in row column coordinates to the converting function. Then when the motors are turned on it is properly calibrated and the exact position of 0 is not necessary.

The first thing that is checked is whether the final position of the AI move is currently a white piece. This is determined by looking at the team value of the postBoard array from before. If there is a white piece then the piece will be captured. The first moves sent to the moveArray will instruct the gantry and electromagnet to go to the location of the captured piece, move it to the left edge of the square with the magnet on, then move to the center row of the board with the magnet on, then move to the out of bounds edge of the board where the player will need to remove the piece. Then the trajectory planning continues as normal. If the piece is not being captured this part is skipped.

Next the code should check for if the move is a castling move, however since the AI did not do this at all and it was a difficult sequence to plan it was considered out of the scope of the project.

The next thing that it checks for is whether the AI piece that is being moved is a knight, also using postBoard. If the piece is a knight it will first move send instructions to move

to one corner of the square it is in, depending on the direction it is heading, then instructs it to move along the line of the grid until it is lined up with it's final square, then finally it moves to the center of the square and releases the piece.

If the AI piece was not a knight then the piece can be moved directly to the final location as there will be no pieces in the way of a straight line to the final location.

After all of these moves are converted to Move objects each move is individually sent over serial to the MSP. After a move is sent the software waits for the firmware to respond with a message of the same instruction number (which the firmware is coded to do after it has finished its instruction) and then sends another move until all of the moves are complete.

Finally the software returns to the normal flow of the chess game program to repeat the sequence again throughout the game.

The chess game will continue in this way until there is a check mate or the game ends.

Inputs and Outputs

Since this functional requirement controls the full game there are no top level inputs or outputs for the chess game and AI integration. Since there are two times when the control of the game is handed off to other functional requirements (detecting player move, and making AI move) the inputs from the first function (player move detection) and the output to the second function (making AI move) will be analyzed.

Table 7. Inputs and Outputs of Chess Game for Player Move Detection

Inputs	Outputs
Player move in string format "A2 A4" as explained above	Sequence of coreXY trajectories with AB location in 16 bit format and electromagnet state (see Figure 12 for format)

The UI shown to the user is somewhat optional as in an ideal game of chess where there are no vision errors or ai movement errors the user only needs to press enter after making moves. However the UI does show the board state as the chess game understands it along with messages prompting for manual entry or waiting for player to make their move as shown below.

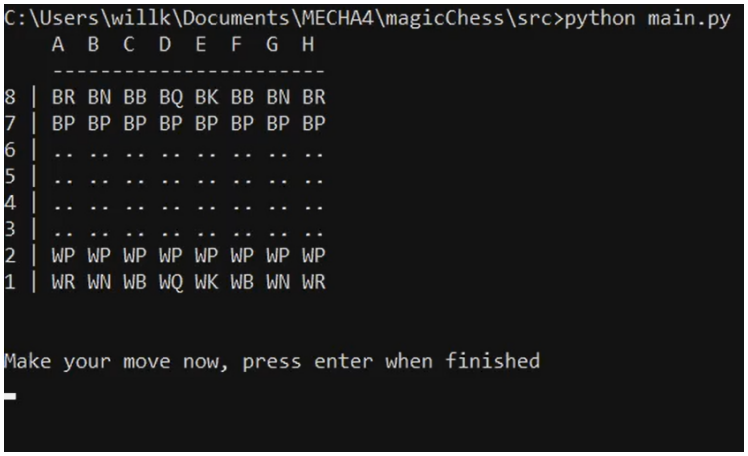

Figure 29. Normal UI Prompt Waiting for Move

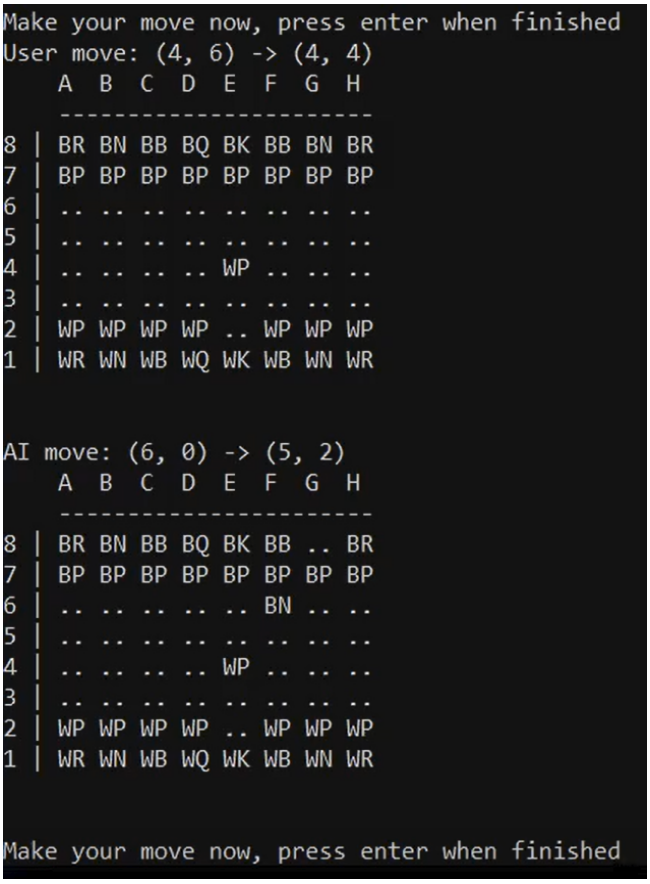

Figure 30. Normal UI Output after AI Move

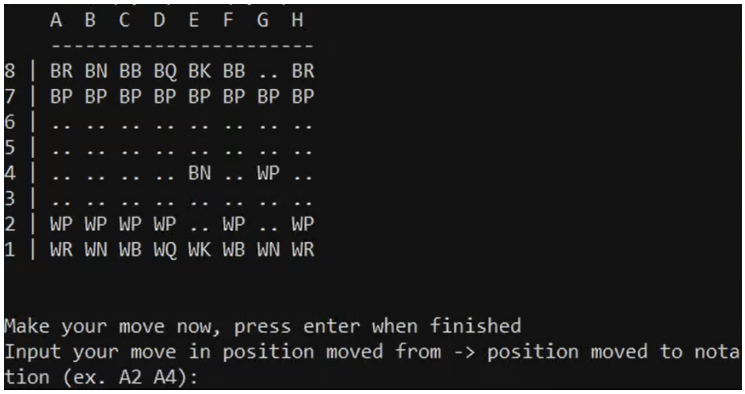

Figure 31. UI Output when Incorrectly recognized Move or Invalid Move

Parameters

There were not many parameters to change in this functional requirement as it consisted of more integration elements. The only parameters that could be tuned were the width and height of the board in software to make the conversions to board coordinates accurate. These were tuned by moving along to specific squares in the board and adjusting in case it overshot or undershot the square.

Testing and Results

As integration is the last function to be implemented most of the testing of this function happened with the full system by running full games. However, there were some tests that were done to incrementally test the system.

Some first tests were done by playing the chess game and AI alone to understand how it works.

The player detection portion was able to be tested with just the webcam and the board and the pieces. This allowed testing to happen as the gantry was being built. This testing was done by adding breakpoints throughout the chess game code and checking to see if the move detected was correct.

The ai move portion was able to be tested in two parts. Firstly, a manual series of moves were added to the move array to test the method of moving the knights or pieces that were being captured to make sure that the theory behind the movement pattern was correct. This also helped test the serial connection with sending and waiting for receiving messages. The second part that was tested was done by stepping through the logic behind the ai_move to trajectory generation method and ensuring that the correct locations were being sent to the MSP430.

System Evaluation

Testing the whole system was only done once we were confident that the individual subsystems were functional (camera detects pieces, motors move electromagnet to correct position etc.). It would be a fruitless endeavour to test the whole board without that confidence as we would need to fix other smaller bugs first.

To test the whole board, the camera position is first verified and adjusted so that the board grid lines up with the grid in software. The computer vision is then calibrated for the lighting conditions by adjusting the RGB values for each piece in the software until it recognizes each one. This can be challenging under non-uniform lighting conditions. Finally, the gantry is calibrated manually by moving the electromagnet under a rook, turning it on, and checking that the piece is centered in its square.

Figure 32. Calibration and Mechanical Adjustments Testing

Once calibration is complete, a game is started with many interrupts in the software for debugging. After each move, the computer vision output is displayed to verify that it is recognizing each move and each piece on the board. The first move made by the AI is monitored to ensure the pathing is correct and that the final position of the piece is sufficiently accurate. Positional accuracy is especially

important to make sure that taken pieces are removed from the board without significantly impacting other pieces.

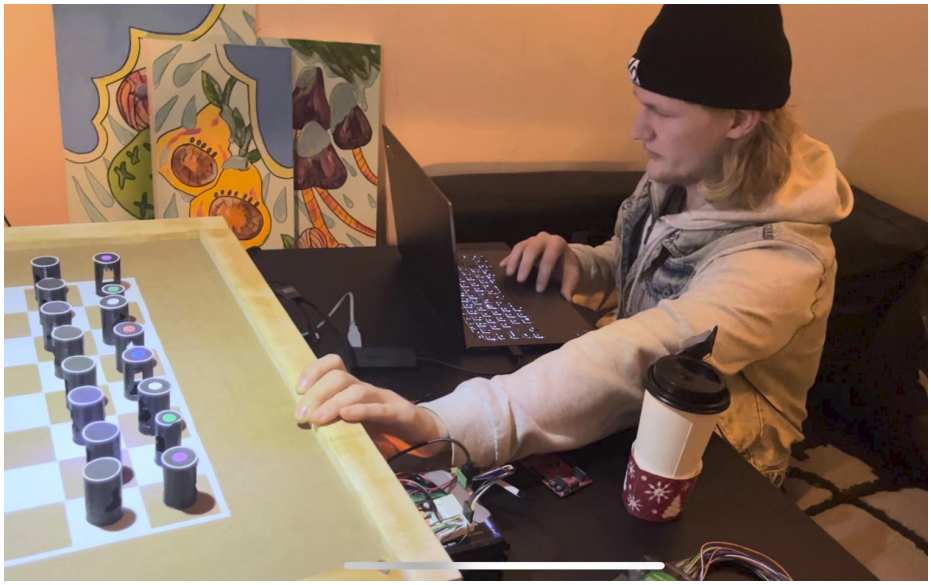

Figure 33. Debugging Software and Vision

Next, to test taking pieces, bad moves are made to give the AI easy pickings. Usually the AI takes the piece and this move is monitored closely to ensure it follows the right trajectory. The AI's recognition of taken pieces and whether its king is in check is also tested by making bad trades and sacrificing pieces to make sure the AI doesn't make illegal moves.

Reflections

What We Would Do Differently

Most of our functional requirements performed well in practice. Some shortcomings include

Joule heating of the electromagnet. Applying 12V to the coil frequently would make it heat up usually not to extreme temperatures. Occasionally, the coil would be left on between test runs and it would heat up so much that the heat shrink around it would start melting slightly. This is a bit of a fire hazard so we would add heat protection around the coil to help prevent fires. A resistor to limit the current would potentially fix the issue or mechanical support for a small fan would help keep the coil cooler.

Manual computer vision calibration. The RGB values for each piece must be adjusted in software for different lighting conditions in a somewhat painstaking process. This makes the board annoying to set up, and could potentially ruin the game if the board is moved to a different room or left for a while and day turns to night for example. An automatic calibration program would make the board much more user-friendly and robust.

Manual motor calibration. The motor must be positioned and zeroed manually whenever the motors are turned back on. Again, this makes the board annoying to set up and could lead to part failure down the line if the motors bottom out the gantry. As discussed in previous sections, we meant to implement limit switches so that the motors could zero the gantry after each move but we ran out of time.

Belt tensioning. Currently there is no easy way to adjust the timing belt tension of the gantry. The belts must be re-tied to the gantry to adjust the tension which is pretty uncomfortable to do under the board. A threaded adjustment knob that pushes the pullets farther apart would make tensioning the belts much easier and user-friendly.

MSP Communication Loss. Occasionally we found that the MSP would lose connection to the computer or would drop response packets, meaning that the game would freeze. We would need to power cycle the board and sometimes the game progress would be lost. If the communication was lost between moves, sometimes the electromagnet would be left on and would heat up without our knowledge. We're unsure about the reason for the communication loss but we suspect it is because the power protection in the MSP board power cycles the board if it sees current overdraw. To fix this the electromagnet would need to be isolated on a different circuit to ensure it is turned off or a different driver could be used with better current limiting.

Three Things We Learned in MECH 423

Some of the most useful things we learned in MECH 423 are

- Event-driven programming
- How to write firmware and software in tandem from the ground up
- Improving divide-and-conquer design skills to break up larger problems into smaller pieces

These three concepts are used prevalently in industry and embedded systems that mechatronics engineers are usually involved in. These concepts and skills will be valuable to us as engineers once we enter the workforce.

Knowledge Limits and Three Things We Want to Learn

There are many functionalities to the MSP board that we didn't learn such as the real-time clock, the DMA and FRAM controllers, and the two other communication protocols SPI and I2C. This is also just one MCU that we have experience with when there are many different kinds available. Practical multi-threaded applications are another knowledge gap.

From our limits in knowledge we would like to learn

- SPI and I2C communication protocols
- How to work with more complex MCUs
- How to use multi-threading in a real-life project

Strategies to acquire that knowledge include making personal projects or joining design competitions. The chess board can also be improved upon and maybe the MSP can be changed for a different MCU or we can try using a different communication protocol further down the line with it.