MagiChess

[1]

Jack DeGuglielmo, CompE, Weishan Li, CompE,
Samantha Klein, EE, and Sai Thuta Kyaw, EE

***Abstract*— Traditionally, chess is played in person with a physical chess board. In modern times, chess can also be played online over a virtual chess board. Currently, there are only a few boardgames that can be played physically with an online opponent. Our aim is to bring a game of physical chess that can be played against anyone over the internet. MagiChess hopes to bring friends together in a game of chess while maintaining the satisfaction of seeing and moving physical chess pieces whilst providing a magical experience for the user.**

I. Introduction

For centuries, the game of chess has been played by two players sitting across a chessboard. The advent of digital technology in the last decades has brought virtual chess to computers and mobile phones and for the first time this has allowed players to be anywhere across the world. However, digital chess lacks a physical aspect and the satisfaction of seeing and moving your own pieces. Physical chess lacks the ability to play from anywhere and with anyone.

We have decided to close the gap between physical and digital chess. To do this, we plan to create a chess board that allows users to play with an AI or a remote human opponent that will:

- Sense location of chess pieces on the board
- Interface with LiChess server
- Automate piece moving

A. Significance

The game of chess is historically embedded in society and culture. Games and entertainment have a significant societal impact in that they offer opportunities to meet new people and bring members of a civilization together. Beyond the general importance of chess's history and impact on members of a society, its impact on cognition is actively being explored. For adolescents, links between chess and performance in mathematics is direct [1]. Additionally, for the elderly, the effects of playing chess can exercise and increase brain-protective activity [2]. For an incurable disease like dementia, this can help prevent symptoms and prolong a healthy life.

During unprecedented times of geographic isolation, internet connected physical chess is even more considerable.

[1]
W. Li from Granby, MA (e-mail: weishanli@umass.edu).
J. W. DeGuglielmo from Stoneham, MA (e-mail: jdeguglielmo@umass.edu).
S. L. Klein from Amherst, MA (e-mail: samanthaklei@umass.edu).
S. Thuta Kyaw from Rangoon, Burma (email: sthutakyaw@umass.edu).

B. Context and Competing Solutions in Marketplace

Internet connected, physical board games present themselves in both commercial and recreational solutions. In the context of chess, internet connected gameplay represents the most populous community on growing servers like Chess.com and Lichess.org [3, 4]. These online hosted chess servers allow players to matchmake global opponents with similar experience levels, record and replay games, and obtain analysis of their gameplay to better improve a user's technique. Beyond online chess servers, tournament style chess can better optimize game recording by using physical chess boards capable of recording moves. The following section describes both commercial and project level solutions for physical, internet connected chess.

Commercial Solutions

SquareOff

After a very successful crowd-funding campaign, SquareOff attracted tremendous attention in the field of commercial internet connected chess boards [5]. SquareOff sells multiple variations of internet-connected autonomous chess boards, but the most notable includes their Kingdom Sets. Some of the most advantageous capabilities of the Kingdom set includes the ability to interface with Chess.com and Lichess.org, integration with a smartphone app and the ability to autonomously move pieces [3, 4, 5]. Some other commercially beneficial characteristics include the capabilities like board versus board play and reasonable portability. Although SquareOff overcomes the prohibitive commercial market for automated chess boards, their tradeoffs for cost and manufacturing has likely influenced less optimal design. The use of resistive force sensors in chess cells to resolve physical moves create a less chess-like experience for the user. Furthermore, this creates a less intuitive, restrictive environment for physical users who want to make very quick moves.

DGT

Digital Gaming Technology (DGT) is another internet connected chess board popular for both its streaming and tournament appeal [6]. Some advantages of the DGT systems include its passive sensing of piece movements compared to resistive sensing. DGT targets the tournament scene of chess where both the recording and live broadcasting of games is paramount. Because of this alternative target audience, DGT Eboards lack entertaining features like the autonomous movement of pieces.

Projects

Beyond more professional and commercial implementations, university groups and project-like solutions exist abundantly. For the purposes of this paper, we highlight one similar system called Autopatzer from James Stanley. This system in particular has many advantages like the passive sensing movements, autonomous piece movements, and integrations with the Lichess.org chess server [7].

C. Societal Impacts

MagiChess would like to bring enjoyment to every person just by visualizing and interacting with the system while enjoying a game of chess with friends from across the world.

D. System Requirements and Specifications

Requirement
Autonomous Piece Movement
Provide Feedback (Audio/Visual)
Sense Physical Chess Moves by user
Playback previous games
Challenge a remote opponent
Transportable

Specification	Proposed Value	Actual Value
Piece movement speed range (cm/s)	4 - 5	5.29 - 11.1
Cell Size	4.5 cm x 4.5 cm	5cm x 5cm
Board Width	36 cm	40 cm
Mass (lbs)	< 50	36.8
Length (in)	< 40”	33.5”
Width (in)	< 35”	30”
Height (in)	< 10”	7”

Table 1: Requirements and Specifications

II. Design

A. Overview

In order for MagiChess to work as intended, there are multiple components that have to work together. While a central computer is necessary for communication to LiChess over the internet, the system has to be able to make physical moves as well as detect moves made by the user. In order to accomplish this, we organized MagiChess into the following subsystems: the gantry subsystem and the fast-scanning subsystem for hardware portion. As for the software portion, there is the main program that communicates with LiChess API and keeps track of the local gamestate as well as specialized programs for path-planning, and moves resolving, that works alongside gantry subsystem and the fast-scanning subsystem respectively. A visual representation of the hardware subsystems can be found in Figure 1.

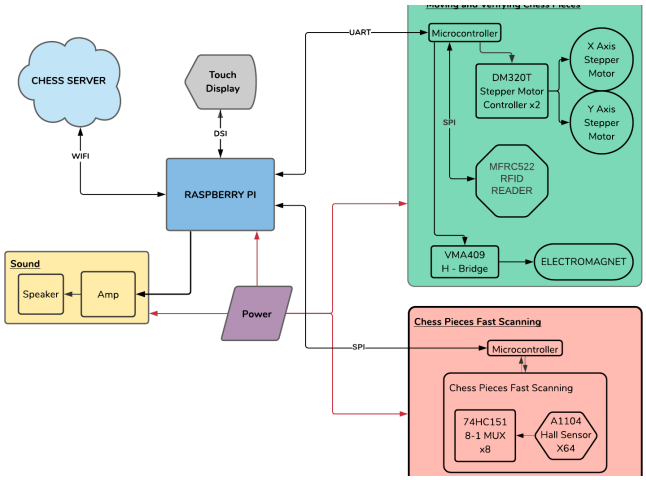


Figure 1: Hardware Diagram showing the connections between different subsystems and their corresponding components.

Per our specifications, and in order for MagiChess to give a magical experience for the user, we wanted the movement of chess pieces to be as fluid as possible. Initially, the team was thinking of a robotic arm that could move the pieces, but the user experience with the arm protruding from the chessboard is not magical nor smooth. After brainstorming, we settled on a XY Gantry system underneath the chessboard that’ll move the pieces with an electromagnet from below. This way, the movements of the chess pieces will be smooth, unobtrusive and magical. Since none of the team members are mechanical engineers and the project is an ECE-focused project, the team decided to spend a considerable portion of the budget on purchasing a laser engraver. Since the laser engraver operates on two-axis, we could easily modify it to our needs while saving us the time for designing the

subsystem. The path planning software works alongside the gantry system and microcontroller to create smooth, non-destructive movements along the chessboard.

In order for MagiChess to sense user gameplay, we decided to use hall effect (magnetic) sensors to detect the presence of a magnet embedded in the chess pieces. The team was initially deciding on whether to use RFID or hall sensors in order to sense movements of pieces made by the user. The team did significant research on multiplexing 64 RFID antennas and readers across the chess board. Although RFID can be more detailed in tracking the type of pieces, and due to its physical limitations, the team decided to go ahead with only hall-effect sensing. The subsystem responsible for doing the scanning of the whole chess board with hall sensors is to be called fast-scanning subsystem. The microcontroller in the gantry subsystem will communicate with the moves resolving software on the Raspberry Pi in order to detect, sense and process moves made by the user.

As for the main software, it has to interface with both the path-planning and move resolving as well as interfacing over the web with a chess server. One of the many options for the team includes web scraping or using stockfish to develop our own chess bot. We decided to use LiChess.com as it has a well documented API and is exactly what the team needed. To run our main application, the team had decided on using a Raspberry Pi 3B+ running on Raspberry Pi OS[8].

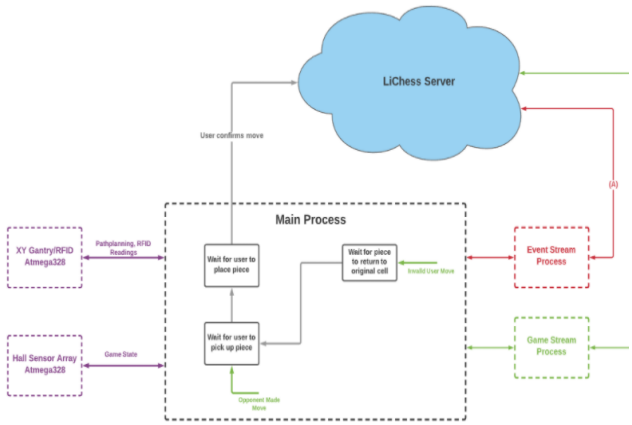


Figure 2. Software Diagram, showing our communication with the Lichess server and our microcontrollers

B. Gantry Subsystem

One of the most important subsystems for a magical chess experience is the gantry subsystem that is responsible for making moves. The hardware is a repurposed generic laser engraver with an electromagnet on the XY gantry head. The external dimensions of the gantry are 24.6” x 28.7” with a working area of 22.6” x 26.7”. The working area exceeds the requirement for MagiChess which needs an area of 25.6” x 15.74”. Moreover, the frame from the gantry is built from 20x20, 4H,SP and 20x40, 6H,SP metric aluminium extrusions. The gantry kit that we bought also includes two timing belts, bearing wheels, spacers, NEMA 14 stepper motors, acrylic supports and circuit for the laser engraver. The bearing wheels are mounted via spacers on the acrylic plates to move freely on both axes. The stepper motor along with the timing belts are mounted along the 40mm x 40mm extrusion to control the gantry stepping movements.

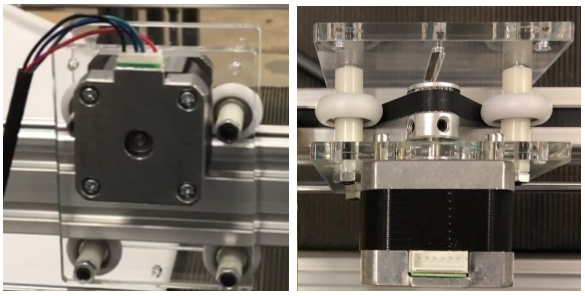


Figure 3: X axis gantry head. Bearing wheels, stepper motor, timing belt, and spacers mounted on the Acrylic Plate. Top view and side view.

Although the laser engraver kit includes it’s control circuitry along with the stepper driver motors, it is revealed upon testing that the circuit was not functional. Initially for MDR, we decided to use Velleman VMA 409: L298 H-Bridge based motor driver. Due to the lack of microstepping and finely tuned motor controls, the stepper motors performed poorly in both thermals as well as with vibration and noise. The vibration from the motors passively amplified by the gantry system made the gantry system screech. We needed a better system for quieter stepper motor movements. The team settled on using a DM320T from Leadshine as it supports microstepping, which steps the stepper motor in smaller increments reducing noise and vibration. The DM320T also includes a digital signal processing algorithm that minimizes resonance for further noise reduction along with idle-current reduction to reduce the heat output.

To provide a seamless experience, the gantry is also required to self-calibrate itself upon startup. We mounted two SPDT limit switches at the top left corners of the gantry - one for the X axis and one for the Y axis. The switches are wired in a normally-closed configuration in order to detect faults easily when the gantry system is operating. Every time the gantry system starts up, the Y axis and X axis

would home itself towards the switch into a known position.

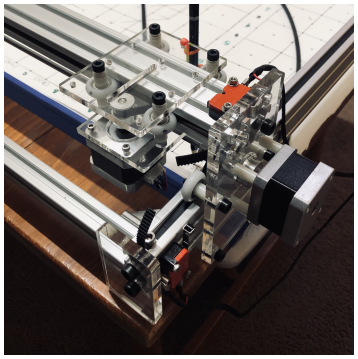


Figure 4: Limit Switches (red) mounted on the top, left corner of the gantry.

The gantry will be controlled by a separate embedded system running on an ATmega328p on a custom PCB. The microcontroller will interface with the stepper motor drivers, limit switches and H-Bridge for electromagnet as well as communicate with Raspberry Pi over UART. The ATmega328p is running at a frequency of 16MHz on DC 5V. The program is coded in C and is debugged on Microchip Studio on a Windows 10 laptop. The UART protocol for Gantry Control document that the team developed (see Appendix H) is used to define UART data for different commands the Raspberry Pi may need to drive the gantry. The microcontroller will then decode and execute the command. Digital GPIO, UART, Timer/Counters and interrupts were utilized in the program. In order to optimize the code for optimum speed and minimal memory, an application note "Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers" was referenced [9]. See Appendix G for more information on the program running on the gantry microcontroller.

In order to test the stepper motors, a sharp pointer with a mm ruler along with an arduino outputting square waves were used. The experiment was run 6 times each on both axes to find the distance travelled per step in the highest microstepping settings. The mean of the results were calculated to give the required mm/step for 1/32 microstepping settings were extrapolated.

Steps	Distance (mm)	Steps/mm
32000	199	160.8040201
31542	197	160.1116751
31702	198	160.1111111
32040	200	160.2
64114	400	160.285
64050	400	160.125
Mean		160.2728011

Table 2: Table showing the number of steps versus the distance travelled on X Axis of the gantry at 1/64 microstepping settings.

To get the correct step settings for 1/32 settings, the following formula was used to convert steps/mm for different microstepping settings.

$$step/mm_{(\frac{1}{x}microstepping)} = step/mm_{(\frac{1}{y}microstepping)} \times \frac{x}{y}$$

After the arithmetic mean was calculated for both X and Y axis, the formula was used to calculate for 1/32 microstepping settings. The following value is used for the movements in the gantry.

$$step/mm_{(\frac{1}{32}microstepping)} = 80.075\ steps/mm$$

Since the gantry has to move between 25 mm intervals, a constant was defined in the program for the number of steps per 25mm.

```
#define steps_per_unitlength 2001.875
```

An experiment was run on loop after initially integrating the gantry system with the Raspberry Pi to test the general functionality of the system as well as the accuracy of the gantry. The gantry was looping through 5 various moves without the chess pieces or the electromagnet. An LED in place of the electromagnet and a pencil was attached to the gantry head. The moves were then traced on a paper underneath the gantry to easily visualize the state of the experiment. While the gantry moves exactly as the Raspberry Pi wanted, the parallel lines over a single path in Figure 5 shows the gantry has been losing steps over each loop of the program.

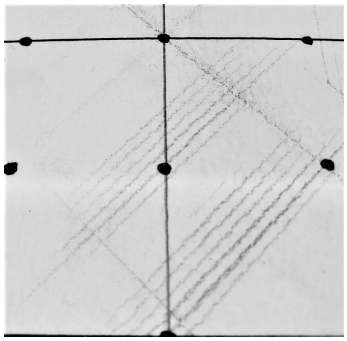


Figure 5: Parallel pencil marks over a singular path shows the gantry losing steps after every loop.

After checking the current, wiring, cable shielding and connections from the microcontroller to the gantry hardware, the experiment was carried out again, only to yield similar results. This reveals that the fault has nothing to do with the wiring, so a logic analyzer was used to probe into the stepping signals going into the motor driver. It was revealed there that one step was lost for every stepping function (step_straightX, step_straightY, step_Diagonal) that the microcontroller goes through. Upon a detailed review of the commands from the Raspberry Pi, it became apparent that for every loop, around 80 stepping functions have to be performed in the microcontroller. Moreover, the decimal point defined for steps_per_unitlength constant input for each step function loses on average another 50 steps per loop. Both combined gives an average step loss of 130 which corresponds to approximately 2 mm distance between the parallel lines as seen in Figure 6. To fix the one missing step for the stepping functions, the pins were toggled manually rather than using XOR function while generating the square waves. A new function round_Decimal was implemented to round off the decimal places arising from the steps_per_unitlength. The experiment was then run and after multiple loops, there is a single line over every path indicating no steps have been lost.

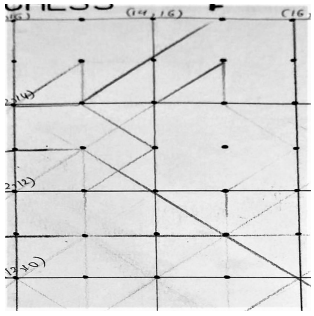


Figure 6: Straight, single pencil marks over a singular path shows the gantry stepping correctly even after multiple loops throughout the experiment.

While the gantry and Raspberry Pi works in unison, more tests reveal that after a certain time, the gantry resets itself. After isolating the fault to be a static discharge from the rubber timing belts and the wheel bearings, which overloads the microcontroller and force reset it. This was solved when the gantry is grounded to the metal enclosure on the power supply.

In order to control the electromagnet, the team ended up using Velleman VMA 409: L298 H-Bridge based motor controller along with an Adafruit P25/20 Electromagnet. Although the Electromagnet is rated at 5V, the MagiChess system overdrives it at 12V in order to get the required strength. To compensate for the heating, a heat sink along with a fan is attached underneath as seen in Figure 7.

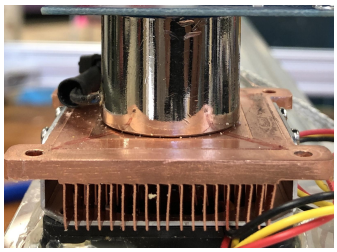


Figure 7: Electromagnet mounted on heatsink and fan.

C. Path Planning

Upon receiving a virtual opponent's move via the LiChess API, MagiChess is responsible for communicating automated moves to the gantry subsystem. After testing common path search methods and encountering application specific challenges, the need for a custom path planning algorithm was evident. Beyond the ability to avoid obstacle pieces, MagiChess's custom path planning algorithm employs optimizations to decrease end-to-end move time.

Astar (A) and Greedy Best First Search (BFS)*

For its completeness, optimality and best first searching, the A* algorithm was a notable contender. Although its resulting solution is guaranteed the shortest path, (given an admissible heuristic) its shortcomings are in its time, syntax and resource intensity [10]. Greedy Best First Search excels in its ease of implementation and limited time and resource complexity. However, without constraint and careful heuristic implementation, Greedy BFS can produce less predictable, less optimal solution paths which would produce error in MagiChess's application.

Challenges and motivations for a custom algorithm:

Collisions

The most important consideration for a successful path planning algorithm, beyond time requirements, is the ability to avoid obstructing chess pieces. The collision of two chess pieces can increase the risk of accidentally toppling pieces and losing the magnetic coupling of a transporting piece.

Illegal Moves

During integration testing of the gantry and path planning elements, our A* implementation generated some optimal, but illegal moves. Moves diagonally across occupied cells run the risk of colliding with obstructing pieces. With this in mind, a simpler path planning implementation was more attractive and allowed for agility when overlaying new constraints.

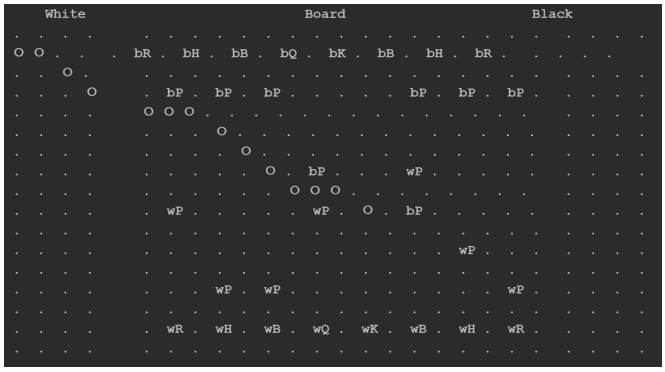


Figure 8: Solution path generated from the MagiChess path planning algorithm.

MagiChess Custom Path Planning Algorithm

Given the testing and analysis of related algorithms' performances, MagiChess implements a custom algorithm built on top of the Greedy BFS algorithm. By using a robust heuristic and overlaying custom move constraints, we achieve near optimality with minimal computational and time complexity. The following describes the process of finding a solution path.

1. *Translation of gamestate to position map*

   To support collision avoidance, the search space must include states to traverse around the perimeter of cells. Because a native 8 by 8 chess state space cannot support piece movement around the perimeter of a chess cell, MagiChess first expands the local gamestate to a 15 by 27 position map shown Figure 26.

2. *Heuristic calculation for each position state*

   After establishing the position map, MagiChess is responsible for calculating a heuristic value for each node. Using the destination cell of an opponent move, MagiChess can identify the position of the path's solution node. Using this solution node, MagiChess computes every other node's straight line distance from the destination position. These weights are used as our heuristic.

3. *Custom Path Planning Algorithm*

   Using the heuristic populated position map and positions for both the start and goal nodes, MagiChess can begin searching. Beginning with the start node, the custom algorithm will traverse unoccupied cells with the lowest heuristic value. If the lowest cost cell is unoccupied and meets additional move constraints, the algorithm greedily adds nodes to the solution path until reaching the goal node. A solution path of a captured piece is depicted in Figure 8.

4. *UART Transmission of path to 328p*

   Once the algorithm reaches a goal node, the solution path is complete and ready for transmission to the gantry system's MCU via UART. The communication protocol is extensively defined here.

Optimizations:

Straight line compression

One creative optimization discovered when looking to reduce move time includes straight line compression. If three or more nodes in a solution path form a straight line, we can interpolate this using two points. By only moving the gantry to two points in a line, we obtain a more efficient, smooth transportation of pieces.

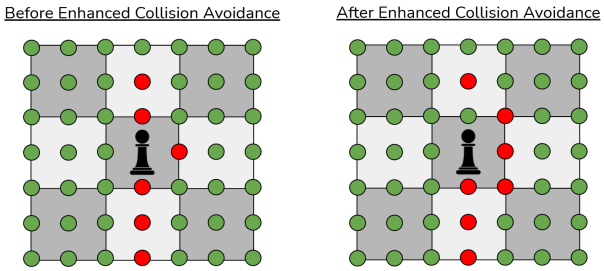


Figure 9: Solution path before overlaying move constraints (left). Solution path after overlaying move constraints (right).

Overlaying path constraints

Using traditional search algorithms, there exist optimal moves that risk piece collision. To account for this, the MagiChess path planning algorithm implements a lookahead constraint. By checking surrounding cells' occupancy state, the algorithm determines how close the path can travel past pieces. For example, when traversing around an occupied cell, the path must direct the gantry to maintain greater distance around obstacle pieces. Here, priority shifts from optimality to collision avoidance as expressed in Figure 9.

D. Fast Scan Subsystem

We developed a fast scanning subsystem to quickly verify if the board gamestate is correct as well as sense user moves. It's composed of 64 A1104LUA-T hall sensors. These hall sensors are arranged in an 8x8 array as seen in Figure 10. The hall sensors in each column are multiplexed using the 74HC151 multiplexer (an 8x1 mux). The outputs from the 8 multiplexers are connected to the fast scan's ATmega328p. The ATmega328p is then connected to the Raspberry Pi via SPI.

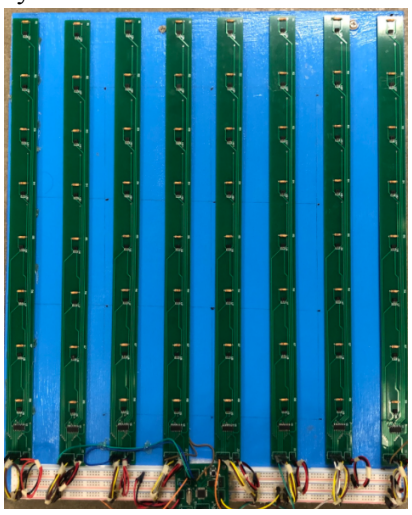


Figure 10: The CDR prototype for the fast scanning subsystem

The ATmega328p receives a command from the Raspberry Pi containing a column number. The 328p will then see which hall sensors in the column have a magnet above them. Then the 328p sends that data to the Pi via SPI. Using data from all of the columns, the Raspberry Pi is then able to detect what move was made.

We experimented with identifying chess pieces as a way of tracking them instead of sensing movement. We explored multiplexing 64 RFID antennas (one for each cell). Each chess piece would have its own RFID tag. However, due to the high costs of the system as well as difficulty encountered with creating a prototype, we were forced to abandon an all RFID solution.

We also considered a hybrid approach: a combination of 64 hall sensors as well as one RFID antenna (mounted on the gantry head). We did not end up using this hybrid method due to problems with the RFID and the copper plate on the hall sensor PCBs. More information on the RFID multiplexing approach and the hybrid approach can be found in Appendix A.

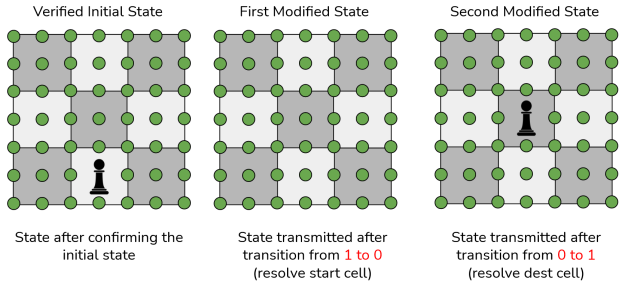


Figure 11: State by state walkthrough of resolving a simple pawn move.

Move resolution

Using sensed physical chess states from the Fast Scan Subsystem, the Raspberry Pi can accurately infer a physical user's move. To begin a physical user's move, the Pi's local gamestate and the physical board state must be verified as congruent. The fast scan interface will initially receive a binary representation of the gamestate and compare this to the Pi's local gamestate. If the congruence is verified, the GUI will prompt the user to begin making a move. Else, it will prompt a user to correct the physical gamestate by highlighting incongruent cells and restart the check.

By verifying congruence, MagiChess can infer the initial state of the chessboard (i.e. the location of pieces on the board). Using this and altered states transmitted during the fast scan period, MagiChess can resolve start cells and destination cells based on whether occupancy signals transition from high to low or low to high. An example or a pawn move is demonstrated in Figure 11. In the case of capturing,

assumptions of the pieces color must be assumed. If a transitioning cell is/was occupied by an opponent colored piece, MagiChess can infer that the piece is being captured. After more states are transmitted for the movement of the capturing piece (distinguished by the user's color), MagiChess can infer the start cell and verify the destination cell as with a non-capturing move.

E. GUI and Local Gamestate

In order to set up games and provide visual feedback, we implemented a graphical user interface for user interaction. We also needed to develop our own form of a gamestate to stay in update with the Lichess server and be able to display the current chessboard to the user. GUI development was a new experience, without having previously been taught in taken courses. That said, previous coding courses such as Data Structures, System Software, and Embedded Systems helped immensely with good coding practice, code organization, and code optimization. Concepts such as inheritance and object oriented programming were very influential in our software design.

GUI development was done using the Tkinter Python library[11]. Different widgets from the library including buttons and entries, allow the user to navigate the GUI and input text. Some of these widgets have additional functionality that sends messages to the Lichess server, which is discussed in (II-D). Tkinter allows for the creation of different page classes, which can be switched in and out of the display window. Each page contains its own widgets, allowing the user to navigate through different pages, with different options. The 'gui_pages.py' module is where most of the GUI implementation is made. 'gui_widgets.py' is a supporting module and the 'gui_test.py' is the main module, used to run the entire program.

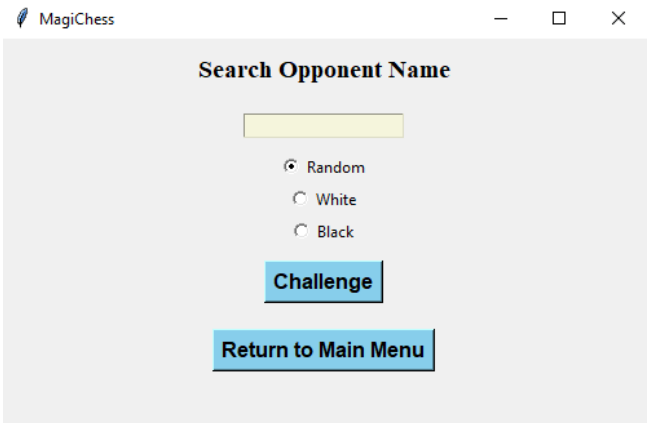


Figure 12. Challenge opponent page, including various widgets

Testing the GUI was straightforward, we would run the main application and confirm that certain widgets functioned the way we intended them to. Because of convenience, most of the GUI development was done on our PCs. This made it easier to test as we were more familiar with Windows OS and its various code editing applications. The GUI was later scaled down to fit and operate well on the Raspberry Pi and its touch display.

In addition to the GUI, the software application must also create, update, and display its own local gamestate. A gamestate object is created whenever a new chess game has been started by the user. An eight-by-eight two-dimensional array representing the chess board is created during the initialization process. Each cell is labeled, either as an empty cell or as the chess piece occupying the cell ('-' for empty, 'wP' for white pawn, 'bR' for black rook, etc.). Something to note is that because this gamestate needs to be displayed, the color (either black or white) of the user affects the orientation of the chessboard and therefore also affects how we map a chessboard coordinate system to array indices. Depending on the user's color, the board is flipped 180 degrees. Two-by-eight 'buffer zones' are also created to hold captured pieces. One holds white's captured pieces and the other holds black's captured pieces.

Updating our local gamestate involves the use of the Lichess server. Essentially our aim here is to make sure our local gamestate is the same as the Lichess server's gamestate at all times. Whenever a move is made by the user, we send a request message to the Lichess server indicating the intended move in the form of chessboard coordinates ('e2d4'). If the Lichess server verifies our move, its own gamestate is updated, and so we then make the same move on our local gamestate to reflect Lichess. The same goes for opponent moves. When it is the remote opponent's turn, our program will wait, until it receives a message from the Lichess server indicating the opponent's move. The move is once again in the form of chessboard coordinates and so we map the move to the indices of our local gamestate's chessboard array. Edge case moves, such as capturing and castling, are also updated properly. Pieces that are captured will be placed in their respective buffer zone cell and the piece doing the capture will be placed where the captured piece originally was. When castling occurs, we move the king to its destination, and then we move the respective rook. Edge case moves are taken into consideration in order for us to be able to constantly reflect the Lichess server's gamestate. Both creation and updating of the gamestate can be mainly found in the 'gamestate.py' module.

Lastly, we need to display our local gamestate to the user in the form of our own chessboard. To do this we used the Pygame Python library. Pygame allows us to open a new window for the current game and draw shapes, add text, and load images. The chessboard and buffer zone cells are drawn on the board in alternating color. The chess pieces are downloaded images that can be loaded into the program and drawn on top of the chessboard. While a game is in progress, our program continues in a loop, drawing the board, pieces, and buffer zones. Between every draw cycle, our local gamestate will update itself to moves made by the user or the opponent. Because we draw based on the gamestate's 8x8 board and the two 2x8 buffer zones, our program will draw the updates that occur in our gamestate. Below, in Figure 13, reveals what the user will be seeing on the Raspberry Pi touch screen when they are playing a game.

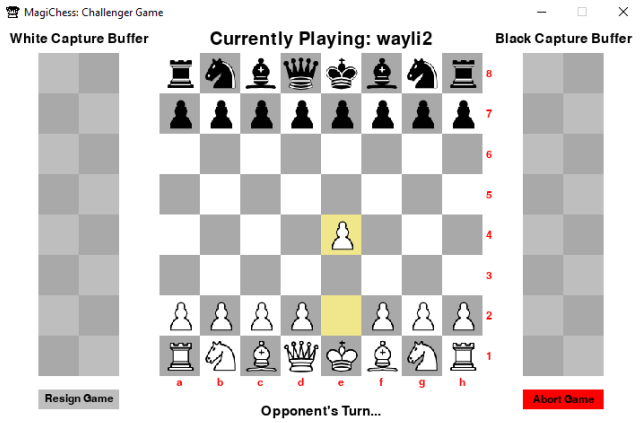


Figure 13. Chessboard reflecting local gamestate

Testing our gamestate was slightly more challenging and less straightforward than the GUI. First we needed to test that the 8x8 board was changing properly according to given chess moves. Our experimenting involved inputting a series of various moves and observing changes in the board. This also allowed us to test our mapping functionality (from chessboard coordinates to array indices). Then, we could test our buffers out, by making moves to cells that contain an opponent piece. Finally, once we knew both the board and buffer were working properly, we tested making moves and observing whether our chessboard would reflect our gamestate board, such as in Figure 13.

F. Lichess API

MagiChess would not have been possible without the Lichess server and its incredibly well documented API[12]. The Lichess server allows users on its website to play with our application and chessboard through its API. Courses, such as Networking, Embedded Systems, and System Software, that emphasize communication from one device to another helped us in this block System Software was especially helpful for its material in multiprocessing, as we needed multiple processes in our program to simultaneously communicate with Lichess and run our main program.

The Lichess API is accessible to its users through a generated api key. The Requests Python library[13] allowed us to send request messages to the Lichess server, where we would use our api key as authentication. The API contains a large variety of various functions that can be called. Some of the more important ones to us were, start an event stream, start a game stream, challenge an opponent, make a move, resign a game, and abort a game. These API functions allow us to play a full game of chess with a remote opponent.

A significant part of our application relies on the event stream and the game stream. A simple API call is used to start both of these streams. However, these calls return a single event response from the server. As we want to catch events in real time and respond to them, we must create separate processes in the background. The Multiprocessing Python library[14] allows us to create these separate processes. Once started, they will run in a loop, making stream requests in every loop. When an event stream is started, Lichess logs the user in on the Lichess server, thus we start the event stream when the user starts our application. Events that we receive from Lichess are in the form of key-value JSON objects. When we receive these stream events, we place them into their respective queues and grab these events from the queue in our main program. Example event stream events could be incoming challenges and outgoing challenges being accepted/rejected. Game streams on the other hand are created when the user enters a new game with an opponent. Once again, the game stream becomes a separate process from the main, and will be constantly requesting game stream events from Lichess. The game stream queue will contain the messages that we receive from Lichess regarding updates that occur in the current game. Example game stream events could be changes in the game state (either from user or opponent), chat updates, and game over cases (abort, resign, checkmate). An example response message received from the game stream can be seen in Figure 14. Specifically, this is an event that indicates a change in Lichess's game state.

```
{
    "type": "gameState",
    "moves": "e2e4 c7c5 f2f4 d7d6 g1f3 b8c6 f1c4 g8f6 d2d3 g7g6 e1g1 f8g7 b1c3",
    "wtime": 7598040,
    "btime": 8395220,
    "winc": 10000,
    "binc": 10000,
    "status": "started"
},
```


Figure 14. Game stream response indicating a change in game state

Aside from the game stream and event stream, the main process will also send request messages to the Lichess server. When a user makes a move, challenges an opponent, resigning a game, aborting a game, and cancelling a challenge are all requests that the main process will send to Lichess. An example of a request message can be seen here. This is how we send a challenge request to an opponent, where 'username' is the name of the opponent we are challenging, 'configurations' are the match configurations, and 'api_key' is our generated api key used by Lichess for authentication.

The responses we receive will indicate whether the request was successful, and some will also contain even more valuable information. As an example, a message in response to a challenge request will contain the game id of that specific game, which we will need to make a move. Figure 15 shows a challenge request response from the server, and here is how we use the game id extracted from the response to make a move.


```
{
   "id": "VU0nyvsW",
   "url": "https://lichess.org/VU0nyvsW",
   "color": "random",
   "direction": "out",
 - "timeControl": {
       "increment": 2,
       "limit": 300,
       "show": "5+2",
       "type": "clock"
   },
 - "variant": {
       "key": "standard",
       "name": "Standard",
       "short": "Std"
```


Figure 15. Part of a response received from a challenge request. Notice the game id as the first key-value pair

Testing communication with Lichess was quite difficult to do, as we could not physically see what was happening. Once we sent a request, the only feedback we would receive is the response message from the Lichess server. Luckily for us, status codes and some error messages were provided along with the response message from Lichess, so we would atleast get a small grasp of what issues were causing the problem. An even bigger challenge was being able to handle the game and event stream responses received in the separate processes. To test these challenges, we worked one request at a time, making sure that we were receiving the proper response back and being able to extract the correct information from each response. We also made sure to monitor the status of any extra processes we branched out from the main program and terminated any processes that were hanging.

III. The Refined Prototype

A. *Prototype Overview*

The final MagiChess prototype is built from 8020 aluminium extrusions and the dimensions will be 33.5" x 30" x 7" respectively. In order to reduce the height, the X axis extrusion of the gantry is rotated by 90°. The XY gantry is then mounted on the frame alongside the power supply and the Raspberry Pi. Acrylic panels are used to mount the remaining hardware onto the frame to give a clean look. JST connectors are used to connect between the components across different subsystems.

Figure 16:Jack DeGuglielmo and Weishan Li working on the aluminium frame assembly in M5.

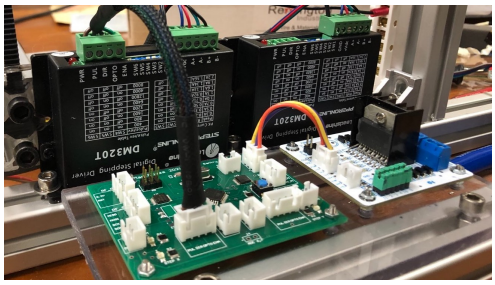


Figure 17:Gantry Control related PCBs mounted on acrylic panel

B. *List of Hardware and Software*

Hardware	***Make/Model***
Single Board Computer	Raspberry Pi 3b+

Hall Sensors	A1104LUA-T
Multiplexor	74HC151
Electromagnet	Adafruit P25/20
Stepper Motor	NEMA 14
Stepper Motor Controller	DM-320T
Electromagnet Controller	Velleman VMA409
RFID Reader	MFRC522
MCU	ATmega328p
Level Shifter	LSF0204-Q1
Dual Voltage Power	Toshiba 5/12V PSU
High Voltage Power	TE Tech 24V PSU

Table 3. List of Hardware and Software

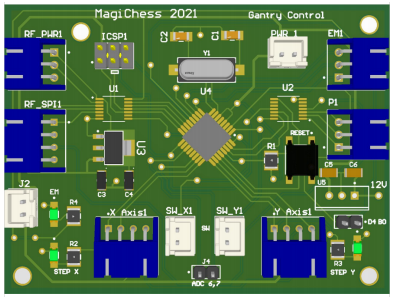
Software subsystems for MagiChess includes,

1. Main software providing GUI and interfacing with LiChess as well as fast scanning and path planning
2. Moves Resolving Software
3. Path Planning and Gantry Communication Software
4. Gantry Control Software (Embedded System)
5. Hall-Sensor Multiplexing (Embedded System)

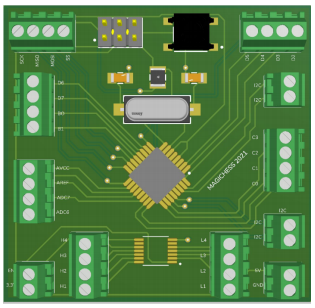
C. *Custom Hardware*

The MagiChess system uses four different custom PCBs in order to work, each with its own functionality.

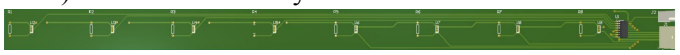
a) Gantry Control PCB



b) Hall Sensor Control PCB



c) Hall Sensor Array PCB



d) Power/Data distribution PCB

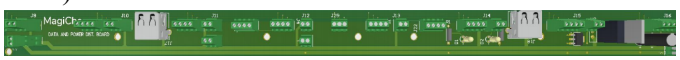


Figure 18: a) Gantry Control PCB, b) Hall Sensor Control PCB, c) Hall Sensor Array PCB, and d) Power/Data distribution PCB

The Gantry Control PCB houses the Microcontroller and related components that runs the embedded system related to gantry control. As a backup feature, there are two power input ports, along with four extra ports for the unused GPIO pins on the microcontroller. Level shifters are used for compatibility between the MCU running at 5V and the Raspberry Pi running at 3.3V. Reset buttons as well as LEDs makes troubleshooting and diagnostic easier. The Hall Sensor Control PCB is designed to be more versatile in terms of port assessment for both the microcontroller and the level shifter. The Hall sensor array PCB will sit underneath each column under the chessboard to provide a clean way to mount hall sensors for chess piece movement detection. It has a 8 to 1 multiplexor that will multiplex hall sensors on different rows. Lastly, as the name suggests, the power and data distribution PCB has a dual functionality. While its main function is to connect 8 hall sensor array PCBs in a neat function, it also includes power rails for distributing power across different hardware subsystems.

D. *Prototype Functionality*

By the end, all elements of the prototype were functioning properly. Our software could successfully communicate with the LiChess server, detect piece movement, and control the gantry and electromagnet. The gantry and electromagnet reliably moved chess pieces correctly. The hall sensor matrix correctly tracked magnet movement.

Our prototype was successfully able to play multiple games in a row. We have some issues with the electromagnet failing to catch a piece. Occasionally the magnet will stop at a piece, turn on, and then not move it. We think this happens with a bit flipping If this occurs, we simply power off the system and turn it back on. This resolves all of those issues.

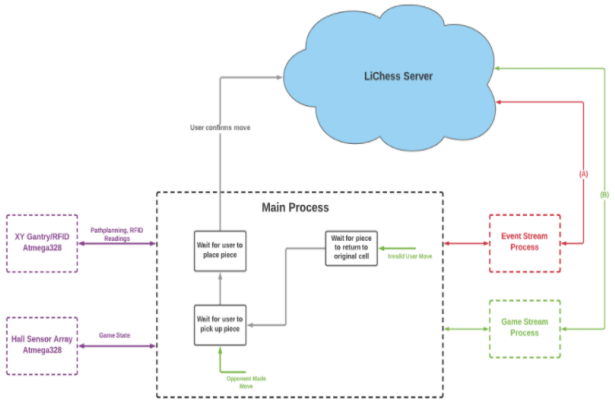


Figure 19. Software Diagram for reference

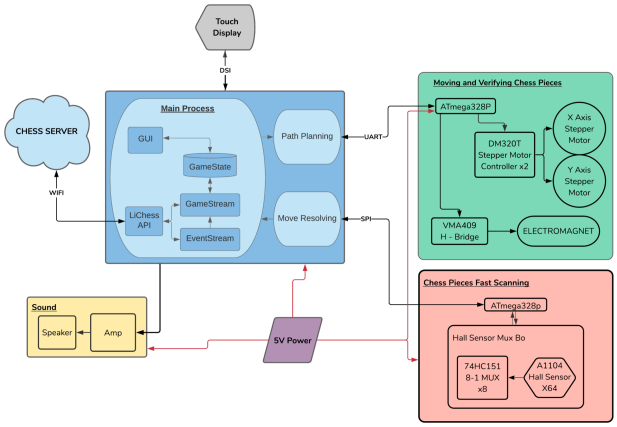


Figure 20: Hybrid Hardware and Software Diagram for reference

E. Prototype Performance

Our prototype is able to autonomously move chess pieces at a speed of 5.29 - 11.1 cm/s. The cell size and board size meet our specification. The size of our board is under our limit. Our system successfully detects a physical move quickly and our fault tolerance is reasonable.

IV. Conclusion

A. The Finished System

The finished product can be seen in Figure 20. The whole system is integrated into our aluminum frame and can be transported as a single unit. In the bottom right corner, we have our dual power supplies; the bottom middle, we have our Raspberry Pi touch display for the GUI; in the bottom left is our motor controllers, motor drivers, H-bridge, and gantry control board; in the middle are our hall sensor PCBs; under the acrylic surface, we have our gantry system with the electromagnet head; and under the Pi display, we have our Raspberry Pi and hall sensor control board. Figures 21-22 show close up images of our integrated system along with our custom PCBs.

The final system can be powered and run on the Raspberry Pi OS displayed on the Pi touch screen. The user can then start the application and interact with the GUI to start a chess game with an opponent. The system will sense user moves, interface with the Lichess server, and automate piece moving to allow for a complete game of chess.

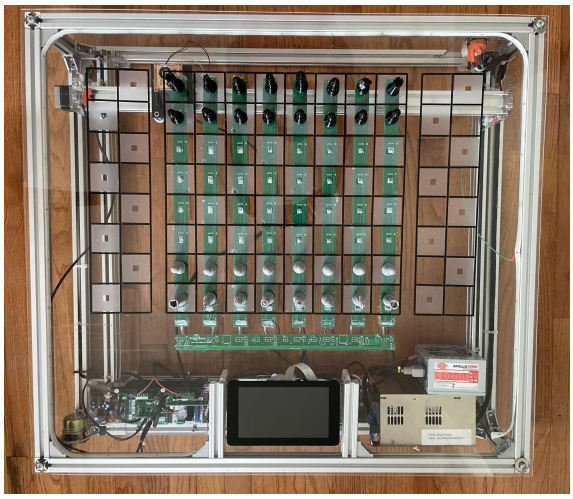


Figure 21. The finished system in a single unit

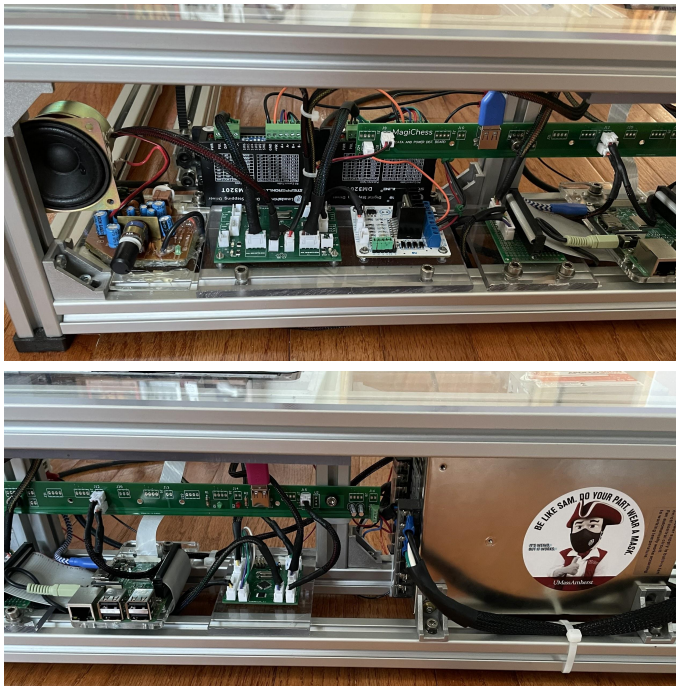


Figure 22 and 23. Close ups of our integrated system along with custom PCBs

B. Additional Features and Future Directions

Aside from the core functionalities of our system to fulfil our requirements, we've also implemented several additional features to make for a better experience. These include: audio feedback, toppling the king on checkmate, replaying games, frosting

every other cell, resetting our board state, and a popup keyboard for the GUI.

Though our project is fully implemented and functioning, there still exists many issues and elements we'd like to work on and add. These include: general error handling, promotion, en passant, resuming ongoing games, multiple virtual players, and playing against bots.

Acknowledgment

MagiChess would not have been possible without the advice and support from our advisor, Shira Epstein. Her continuous support as well as knowledge gave us helpful advice and multiple approaches to the problems the team encountered. We would also like to extend our gratitude towards our evaluators, Professor Krishna and Professor McLaughlin, along with the course coordinators, Professor Hollot, Professor Soules, Professor Malloch for their feedback and advice during design reviews and check-ins. Their valuable feedback had pushed both the team and the project forward. We would like to thank Keith Shimeld and Terry Bernard for helping us procure numerous parts required for the project. Finally, we would like to give a shoutout to Lichess.org and their open-source chess server. Their meticulous documentation of the API allowed us to integrate their server with MagiChess.

References

[1] C. Meloni and R. Fanari, "CHESS TRAINING EFFECT ON META-COGNITIVE PROCESSES AND ACADEMIC PERFORMANCE," *Proceedings of the 16th International Conference on Cognition and Exploratory Learning in Digital Age (CELDA 2019)*, 2019.

[2] P.-J. Chen, S.-Y. Yang, C.-S. Wang, M. Muslikhin, and M.-S. Wang, "Development of a Chinese Chess Robotic System for the Elderly Using Convolutional Neural Networks," *Sustainability*, vol. 12, no. 10, p. 3980, 2020.

[3] "Chess.com." [Online]. Available: https://www.chess.com/. [Accessed: 04-May-2021].

[4] "Lichess.org," *lichess.org*. [Online]. Available: https://lichess.org/. [Accessed: 04-May-2021].

[5] "Square Off," *Square Off*. [Online]. Available: https://squareoffnow.com/. [Accessed: 04-May-2021].

[6] "Digital Game Technology," *Digital Game Technology - Digital Game Technology*. [Online]. Available: http://www.digitalgametechnology.com/index.php?mavikthumbnails_display_ratio=2. [Accessed: 04-May-2021].

[7] J. Stanley, "Autopatzer: my automatic chess board," *RSS*. [Online]. Available: https://incoherency.co.uk/blog/stories/autopatzer.html. [Accessed: 04-May-2021].

[8] Raspberry Pi, "Raspberry Pi OS," *Raspberry Pi*. [Online]. Available: https://www.raspberrypi.org/software/. [Accessed: 04-May-2021].

[9] "Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers," [pdf] Available: https://ww1.microchip.com/downloads/en/AppNotes/doc8453.pdf. [Accessed: 04-May-2021].

[10] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Hoboken: Pearson, 2021.

[11] "Graphical User Interfaces with Tk," *Graphical User Interfaces with Tk - Python 3.9.5 documentation*. [Online]. Available: https://docs.python.org/3/library/tk.html. [Accessed: 04-May-2021].

[12] "Pygame Front Page," *Pygame Front Page - pygame v2.0.1.dev1 documentation*. [Online]. Available: https://www.pygame.org/docs/. [Accessed: 04-May-2021].

[13] "HTTP for Humans™¶," *Requests*. [Online]. Available: https://docs.python-requests.org/en/master/. [Accessed: 05-May-2021].

[14] "multiprocessing - Process-based parallelism," *multiprocessing - Process-based parallelism - Python 3.9.5 documentation*. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html. [Accessed: 04-May-2021].

[15]

[16] Wyatt, J., 2012. *TRF7960A RFID Multiplexer Example System*. [pdf] Texas Instruments. Available at:

<https://www.ti.com/lit/an/sloa167/sloa167.pdf>
[Accessed 5 May 2021].

[17] Houde, F., 2018. *Why, When, and How to use I2C Buffers*. [pdf] Texas Instruments. Available at: <https://www.ti.com/lit/an/scpa054/scpa054.pdf> [Accessed 5 May 2021].

APPENDIX

A. Design Alternatives

Gantry Alternative

The team's choice for using an XY gantry underneath a chessboard as opposed to a robotic arm was explained earlier. MagiChess strives to provide an unobtrusive magical experience in the movement of chess pieces and only a XY gantry system is able to achieve this.

The decision to use an XY Gantry system also brings numerous challenges for choosing chess pieces and magnets across different surfaces to bring a magical experience. In order to accomplish this, we need the chess pieces to be

- Big enough to enjoy gameplay
- Small enough to slide through other pieces
- Smooth and light enough to overcome static friction
- Heavy enough to cancel the repelling forces from magnets from nearby pieces.

A mini testbench was made in order to mimic the movements made by the gantry. Numerous experiments were carried out on the testbench before settling on our final design.

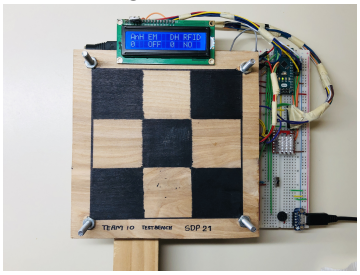


Figure 24. Mini-Testbench for MagiChess prototype

The team settled on using 6x6 mm cylindrical neodymium magnets on chess pieces with a base diameter of 21mm. The heights of the pieces range from 67mm to 40mm and the pieces will be 3d printed. A velvet sticker was used for the bottom to allow smooth sliding across the chessboard.

Fast Scanning Alternative

As stated in section II.D., we experimented with identifying chess pieces as a way of tracking them instead of sensing movement. We explored multiplexing 64 RFID antennas (one for each cell). Each chess piece would have its own RFID tag.

We looked at an application report by Texas Instruments titled "TRF7960A RFID Multiplexer Example System" and went about trying to multiplex several RFID antennas with a single RFID reader chip [16]. The team purchased RFID tags, RFID antennas, RFID readers, and RF switches as well as breakout boards. However, when we went to solder the RFID switches, we discovered that while the package was the same between the board and the switch (QFN16), the switch was too small for the board. This is illustrated in Figure 25.

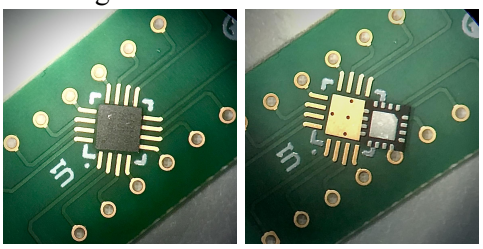


Figure 25. The RF switch vs the breakout board

Next the team explored the feasibility of attaching 64 RFID readers on a single I2C bus. According to our research, an I2C bus can have over a hundred devices on it [17]. However, the specific reader chip we bought for prototyping only had two possible addresses. After all of these setbacks and after considering the cost to continue testing, the team decided to abandon an all-RFID solution.

We also considered a hybrid approach: a combination of 64 hall sensors as well as one RFID antenna (mounted on the gantry head). We did not end up using this hybrid method due to the copper plate on the hall sensor PCBs causing RFID to not work.

B. Technical Standards

The MagiChess project uses a variety of standardized communication protocols for both wired and wireless scenarios. IEEE 802.11n-2009 wireless networking standard was used to connect the Raspberry Pi to the internet. Wired communication protocols such as UART and SPI were used to communicate between the two embedded systems and the Raspberry Pi. Python programming language along with standard libraries for UART, SPI, multiprocessing, Tkinter, os, time, and signal was used for programming the Raspberry Pi. The embedded systems for the gantry and fast scanning were programmed using ANSI C.

As for the components on the PCB, 32TQFP, 14TSSOP and 16SOIC packages were used for the various ICs along with 1210 and 1206 SMD packages were used for the resistor and capacitor respectively.

C. Testing Methods

After the final assembly of the system, the team started by testing different subsystems starting with power and wiring to troubleshoot possible problems in hardware that resulted from migrating.

Initially the subsystems were tested separately running the same program on the Raspberry Pi due to challenges caused by the pandemic. For the Gantry subsystem, moves were looped over time and a virtual

game was played to see the pieces moving. As for the Fast Scanning program, a testbench program was made to output moves made by the user with a magnet. The system has yet to be tested, but the plan for testing after integration will be to play a game of chess over the internet. We then would like to expand testing with friends over a safe distance to stress the system to its max.

D. Project Expenditures

Item category	Cost
Gantry + Stepper motors	$202.44
Parts for pre-MDR testing	$102.49
Prototype costs	$29.49
PCB costs	$74.96
PCB population costs	$99.03
Remaining assembly costs	$86.59
Total	$595

Table 4. Project Expenditures

E. Project Management

Although our team is geographically separated due to circumstances of the pandemic, we excel in our responsiveness and communication. One exemplary characteristic of our team includes our agility. We are all respectful and accommodating to when plans change or the direction of the project transitions. A balance of technical expertise in computer and electrical engineering helps tremendously in the context of a multidisciplinary project like MagiChess. The team met regularly to discuss new ideas or to work collaboratively on tasks. According to available Zoom statistics, as a full team (all four members present), we met for an average 3.75 hours each week. This does not include additional time met in pairs or with three of us.

The most indicative example of our team's capabilities and success was our pre-CDR sprint. As with most complex projects, a fully integrated system represents a large checkpoint in development. The team worked diligently to integrate all subsystems and validated the prototype in a compelling live demonstration. Furthermore we can highlight the success of MagiChess during the Senior Design Demo Days. Attendees were visibly impressed especially with the capability to topple pieces electromagnetically.

F. Beyond the Classroom

Sam-- My responsibilities have been mainly circuit design as well as MCU programming. Tutorials found on the internet were the most helpful in this area. I had to improve my debugging skills in both the hardware and software side. I learned Altium in order to design several PCBs I was responsible for. Probably the biggest challenge was coordinating with the rest of the group (including deadlines, times to work on parts of the project together, and having parts delivered to me since I was fully remote). The biggest takeaway from this project would be the importance of staying organized. Skills you can learn, but if the project isn't organized, things will quickly go downhill.

Weishan-- Through the course of this project, my responsibilities lied in Raspberry Pi software development. Therefore, skills involving good code practice, code organization, and code optimization were essential when creating our MagiChess application. Various concepts in object oriented programming such as inheritance and abstraction helped make sure our code was organized and robust. Other skills such as time complexity helped with optimizations and efficiency. Resources that helped me were the documentation of the various new libraries that I had to learn for the program. Application development was an enjoyable experience and is definitely a path I would be interested in pursuing.

Sai-- In order to work on the gantry subsystem, I had to learn how the stepper motor works along with the terminology for the stepper motor drivers. The datasheet has helped clear any questions that I may have on a component. Furthermore, the application notes provided a helpful insight on code development and wiring. YouTube videos helped me start Altium Designer but my previous experiences in VLSI layout and KiCad has helped tremendously.

Jack-- My responsibilities included predominantly software and hardware interaction. The ability to troubleshoot root cause issues was very demanding and often my greatest challenge. Working with concurrently running software and hardware components exercised my understanding of system software design and encouraged very creative, unintuitive techniques to troubleshoot problems. When so many variables exist, it is critically important to isolate functional elements. This helps to more

accurately diagnose bugs and build a more informed picture of the operation of a system.

G. *Gantry Control Microcontroller Code*

The code for the ATmega328p embedded system for the gantry control system was written using ANSI C with AVR libraries on Microchip Studios. Upon reset, the UART is initialized after GPIO pins are configured for the respective input and output settings. The system will then self-calibrate by moving the stepper motors to the upper left corner of the gantry. The stepper motors are moved towards the limit switch until it is pressed. The X-Axis is homed first then followed by the Y-Axis. The gantry head will then move a defined number of steps to a known location before waiting for more commands from the Raspberry Pi. The decoding of the commands can be found in the Protocol for UART Gantry Communication Document see Appendix H.

Upon receiving relevant commands from the Raspberry Pi, the microcontroller will call different functions in order to execute the commands.

In order to go to preset locations shown below, the microcontroller relies on calculating the required steps to take from the current position rather than using a lookup table array.

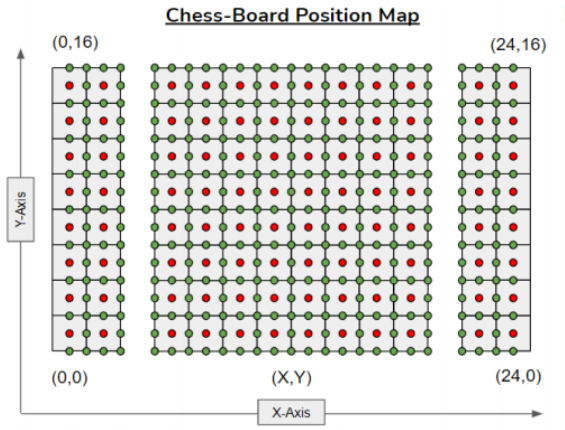


Figure 26. MagiChess Position Map

This significantly reduces the memory usage as well as makes the programming easier as only one known location has to be hard coded. While transporting a piece (i.e electromagnet is on), the gantry head can only travel either straight or in diagonals at 45 degrees. If the gantry head is travelling (i.e electromagnet is off), the gantry head will move diagonally first to the nearest straight line destination before moving straight.

RFID was also implemented into the code, but due to its physical limitations, the team did not end up using it. The GPIO pins for H-Bridge control are hard coded for switching the electromagnet.

Figure 25. Jack DeGuglielmo using a dremel to indent the acrylic top layer

H. *Protocol for UART Gantry Communication*

The team has implemented the protocol for UART Gantry control and communication to define different commands needed by the Raspberry Pi to make physical moves. These include transmitting the X and Y addresses, electromagnet control as well as commands for gantry to go to a specific location. UART with a baud rate of 9600 with 8-Bit Data packets was used. The 3 MSB is used to define the type of message while the remaining bits are used to convey the message depending on the type sent. The document can be found on our website for further information.

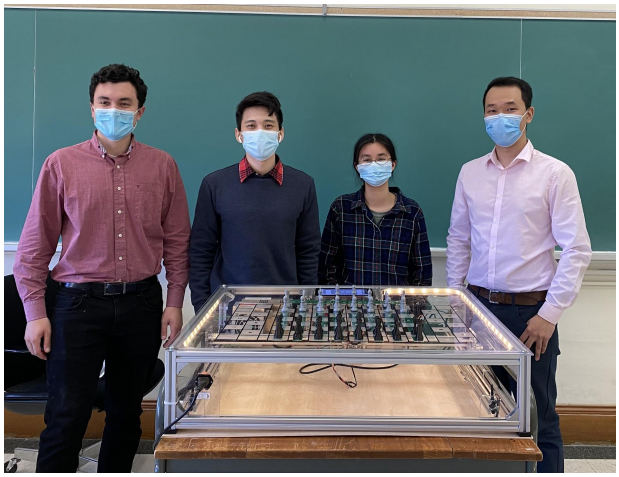


Figure 26. Team 10's Group Photo with our MagiChess Project After the Second Demo Day