Wizard's Chess

EE Senior Design 2020

Members: Ben Baileys, Delany Bolton, Fabio Frietas Lopes Soares, Luke Tholen, Shae Watkins

Table of Contents

1. Introduction

Imagine you are an eleven-year-old child. You open the mail to see that you have just received your letter informing you that you've been accepted into Hogwarts School of Witchcraft and Wizardry. It is the best day of your life; not only have you learned that magic is real and you possess this gift, but also because you no longer have to play with a run of the mill boring chessboard. Now you can play with Wizard's Chess! Wizard's Chess is a chessboard with pieces that can respond to your voice commands. It is a revolutionary way to play chess! Now imagine: sadly you did not receive a letter from Hogwarts outlining your acceptance when you were eleven. Instead, you grew up in the muggle world, studied hard, and attended the University of Notre Dame to become an esteemed Electrical Engineer. Senior Design rolls around and you are granted the unique opportunity to create your version of Wizard's Chess. Do you take the chance? Unfortunately not because a global pandemic broke out and in order to respect social distancing rules and the safety of others, you were required to stay home and participate in remote learning. However, the concept and design of Wizard's Chess still exists and is outlined in the following formal report.

While the idea of Wizard's Chess stems from a fictional movie about magic, the real-life version has many practical applications and would be quite popular in the modern world. Technology is becoming more and more geared towards full automation every day, and to develop and market a chessboard that is completely hands-free would be a welcome addition to the market. In order to make the project as realistic to the movie as possible, but also possible in real life, many different problems had to be addressed. First, a mechanical system that will work underneath the chessboard and can move the pieces to and from their grid specific locations in order to have proper gameplay is needed. The system also needs to be able to remove pieces from gameplay. In order to accomplish this, the group utilized a coreXY 3D printer coordinate system and an electromagnet was housed in the moveable carriage. The coreXY design choice allowed for precision movements and an electromagnet allowed for selective contact on the

chess pieces. The chess pieces would be magnetized and when intended to grab a particular piece, a current would be passed through the electromagnet thus allowing it to grab a piece and move it to the appropriate location. Testing was not able to be conducted on this, but another potential use of the electromagnet would be to utilize the ability to reverse the polarity of the electromagnet by reversing the current through it. If the chess pieces of opposing colors were of different polarity then the magnet would only pick up pieces of the correct team on each move. This would be controlled by the software. Precision movements and electromagnet control are very important when considering the size of a normal chessboard. The chessboard cannot be too big as to be cumbersome, however it needs to be big enough to allow the pieces to be moved around each other. Precision movements allow for more control and a smaller and less cumbersome board. The requirements and design of these mechanical systems are outlined below.

Additionally, this product requires several software subsystems. The first, a voice recognition system that allows users to speak aloud their intended move which prompts the board to respond by executing that move. The system has to be able to interpret a range of voices into recognizable commands. This system requires a microphone in order to listen and collect that data. After the command is heard and interpreted it has to be fed into a chess gameplay subsystem of code that analyzes the move and ensures it is a valid move, as well as checking and keeping track of all the pieces on the board. This system has the potential for AI gameplay and other functionalities. Once the command has been interpreted and the move has been ensured as valid the software then needs to communicate with the motors to control the coreXY system in order to move the piece to the necessary location. These systems will have to communicate with each other in order to perform other actions like removing a piece from gameplay, determining when the microphone should begin listening for a new command, etc. The software component of the project is critical and it connects all of the different systems into a working whole.

The system requirements for each of the detailed subsystems can be found in Section 2 of this report. The detailed project descriptions, including pictures, diagrams, schematics, analysis, and subsystem design can be found in Section 3. A description of how the entire system and all of its subsystems could be integrated and tested can be found in Section 4. While the project

could not actually be built and tested, there has been significant effort put into the thought process behind possible integration testing schemes and how successful they would be. Section 5 outlines the user operation manual and the installation manual. This would aid an individual not only in using the product but also determining its initial set-up and normal operating parameters and requirements. Section 6 includes the possible to-market design improvements. As the product was never actually completed there is a lot of room for improvements in this category. However, there is little experimental data to substantiate the feasibility of these improvements due to the fact that a working prototype was never constructed. Section 7 provides the conclusions drawn from this project and process, and Section 8 includes appendices, such as design schematics and software listings.

This project, though unfinished, would have been magnificent to lay your eyes upon. It would have accomplished the task of listening to voice commands and moving chess pieces accordingly in order to facilitate engaging gameplay. Numerous challenges were encountered during the preliminary build process, before spring break. Had the project been able to be finished these would have been addressed. It could be expected though that the final product would have had some shortcomings and those would have to be addressed in the next iterations of the design. However, as the working prototype was never constructed, there is no way to tell if the design would have worked as intended. Most of the code had been written and functioned correctly, however integrating the system was impossible remotely so there was no way to determine the actual functionality. Before spring break, there were still some significant mechanical drawbacks to the system and a plan was in place to heavily revamp that entire system with new 3D printed slider chassis and timing belt tweaking. If these procedures had been completed, there is a strong possibility that the mechanical aspect of the project would have worked very smoothly. It is expected that by the end of the semester a functional prototype would have been built that would have correctly interpreted voice commands and translated them into motor controls to move an electromagnet around a chess grid in order to move the chess pieces in a hands free manner. This is what the original design aimed to accomplish and therefore would have been a successful senior design project, if it was allowed to have been completed.

2. Detailed System Requirements

The following are the system requirements for each of the major parts of the Wizard Chess design.

Motors

- Power: Must be able to draw power from a wall outlet. Will require outlet adapters and cords. The motors require 12V so we will need to power them using a 12V converter.
- Torque: Each motor must prove enough torque to turn the coreXY pulley system, but not too much torque as to break the timing belts.
- Speed: Each motor must provide enough speed to move the electromagnet carriage fast enough to make gameplay happen in real time, an average of five seconds per piece move.
- Precision: The motors need to have a step size small enough to make precise movements within the chessboard grid, but large enough as to be appropriate for normal timed gameplay.

CoreXY

- Accuracy: Must be able to move pieces across the chessboard with accuracy(4 sq. in. grid squares), as well as move pieces in between other pieces(the knight).
- Robustness: Must be strong enough to withstand the tension strain imparted by the timing belt and the weight of the board placed atop.
- Timing Belt: Must be tight enough to effectively pull the electromagnet carriage around the board. It also must maintain this tension over time so the timing belt does not separate from the pulley system and ruin the ability to use the chessboard.

- Wheels: The wheels must be the correct size to fit into the v-slotted aluminum railing, and they must have enough friction as to not slip when in use.
- Y-grid movers: These 3-D printed pieces need to have enough durability to withstand the tension of the pulley system, as well as enough mass to support the y-grid guide railing. Additionally they need to be printed with enough precision to provide the wheels with a flush contact to the v-slot.

Voice Recognition

- Accuracy: Must be able to accurately interpret a broad range of voice profiles.
- Analysis: Must be able to analyze the voice for common chess commands and respond accordingly.
- Interfacing: Must be able to collect and analyze the data input from the microphone and subsequently transmit that information to the chess game tracker and the motors in order to calculate the required number and direction of steps the motor should take.

Chess Game Tracker

- Accuracy: Must be able to keep track of all game movements so to know where all pieces are on the board as well as know which moves would be considered illegal.
- AI: Must provide some AI functionality so that the computer could effectively play a chess game against a human opponent.
- Interfacing: Must be able to interface with both the voice recognition system and the motors to both interpret and respond to the commands given.

Electromagnet

- Range: Must have a small enough magnetic signature to be able to move one piece at a time while not affecting the placement of others on the board.
- Polarity: Must be able to effectively swap polarities in order to move both sides of chess pieces(they will have different polarities).

- Strength: Must be strong enough to maintain a magnetic connection with the chess piece through the wooden board as it is also moving.
- Weight: Must be light enough to be easily moved by the carriage of the coreXY system.
- Power: Must not draw too much power as to overheat the magnet, but should be able to handle normal operating levels and be strong enough to work effectively in these conditions.

Microphone

- Range: Must have a range of at least 5 feet in order to hear commands given at normal distances.
- Power: Must be able to derive power from the wall outlet.
- Interfacing: Must be able to communicate with the RaspberryPi in order to interpret the speech input.

Physical Build

- Weight: Must be light enough as to make transport feasible, if slightly cumbersome.
- Robustness: Must withstand natural wear and tear and normal indoor gameplay conditions. Must also protect all internal mechanisms from external factors that could harm or disrupt the system.
- Size: Must be large enough for the pieces to move effectively but small enough to easily be moved by two people.
- Thickness: The chessboard must be thin enough to allow the electromagnet to work, but thick enough to support the pieces as they glide across the surface.

3. Detailed Project Description

3.1 System Theory of Operation

Beginning with a simple game setup like traditional chess, we developed our entire project around the board and chess pieces. Wizard's Chess has the same rules as traditional chess, but with a fun twist. In theory, one of the two players will give a command on their turn. Commands follow the structure of "[piece name] to [board coordinate]" with the board coordinates being a letter and number corresponding to the row and column. A microphone takes the voice input and deciphers the command in a voice recognition program determining where the current piece is being moved from and too. This information is then passed to the internal game tracker and the motor control program. In the motor control program, rotation of the motors is determined to align the piece with the center of the electromagnet and then move the electromagnet to the new position. When the motors reach the initial location, the electromagnet switches on creating a magnetic connection with the bottom of the chess piece. As the motors move to the new position the magnet is drug along the top of the board to its final location at which point the electromagnet turns off. At this point it is the next player's turn. This process is repeated until a piece is overtaken by the opponent, when this happens, the internal game flags a piece being overtaken and moves the motors first to the piece being overtaken to remove it from the board. Once the overtaken piece is moved off the board the usual process continues.

3.2 System Block diagram

From the user's perspective, the Wizard's Chess Board takes in two inputs, the player's voice command and the current positions of the pieces on the board, and outputs the execution of the player's move and updates the positions of the pieces on the board. It is important to note that all the user will experience is giving voice commands to the board and seeing the board respond

with the move they just spoke, the rest is all hidden from view within the Wizard's Chessboard to keep the magic alive.

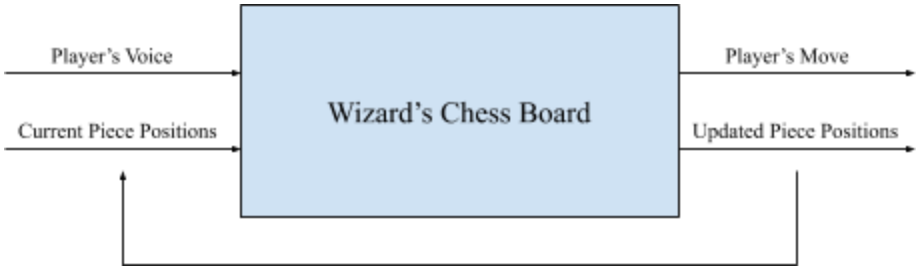


Figure 3.2-1

Within the chess board are four major subsections which operate without being visible to the user. These subsections are the voice control system, the chess code, the motor software and hardware, and the magnet software and hardware. Each of these subsections is built around a CoreXY system for moving the pieces. These subsections and the CoreXYsystem will all be further explained in the following subsections.

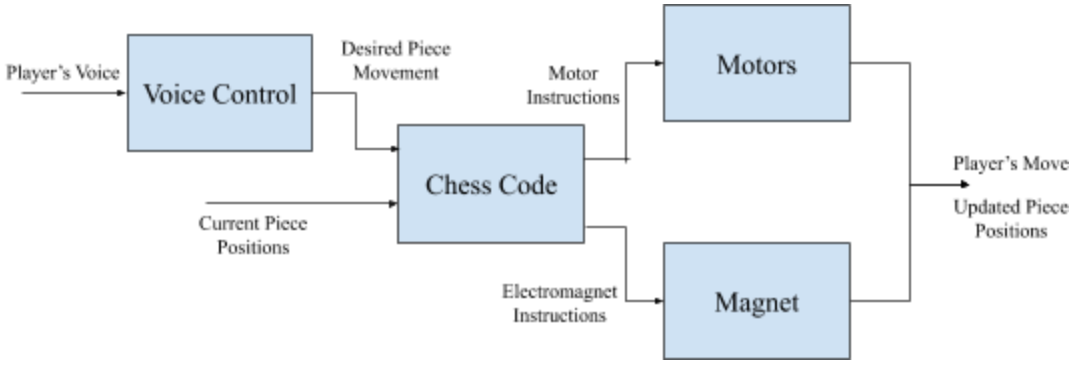


Figure 3.2-2

Once the user decides on the move they would like to make, the voice control software recognizes the move they wish to make and the chess code calculates the required motion of pieces required to execute this move. For example, a player could say "Queen to D6," which the voice command would translate into the name of the referenced piece and the destination of that piece. The main chess game code first verifies that the requested move is valid given the piece's position and type. Then, using the output of the voice control software and the stored initial positions of all of the pieces on the board, determines if it needs to remove an eliminated piece as well as how to move the specified piece to its final destination. These moves are actuated by sending a series of instructions to each motor of the CoreXY rig to move the magnet under the

desired piece and move that piece to its final destination. At the same time, the chess code also sends instructions to the electromagnet to activate and deactivate it in order to grab and release the piece as necessary to complete the player's move.

Each of the mentioned subsections will now be explained in greater detail.

3.3 CoreXY

The mechanical portion of the project is inspired by various CoreXY systems such as 3-D printers. The concept behind this design is to allow movement in a designated space either horizontally, vertically, or diagonally. By utilizing two motors, two timing belts rotate at specific rates to allow for a central component to be moved from an original (x,y) coordinate to a new coordinate.

Our framework design consists of a compressed wood particle board cut such that there is a hole in the center where the central component can move freely. Along two opposite sides of the particle board are aluminum v-slot framing extrusions that are secured using fasteners. At one end of each extrusion there is a motor which is secured to the particle board utilizing industrial glue. Opposite of the motors is a screw with timing belt pulleys spaced to specific heights by washers and nuts, still allowing the pulleys to freely rotate. This setup is the foundation for other components to move about the determined space.

For movement in the x direction, two chassis were designed and 3-D printed to straddle the framing extrusions supported by three v-slot wheels with bearings that sit in the side slots of the extrusion. The chassis we printed are very similar to the one seen in figure 3.3-1. The spacing of the wheels is such that a third v-slot framing extrusion can be connected between the two chassis without causing any torque. Additionally, each chassis has two locations where timing belt pulleys are secured using screws, washers, and nuts; again these pulleys are able to rotate freely.

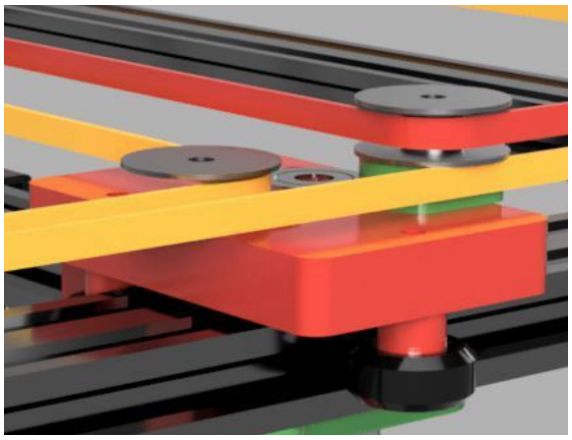


Figure 3.3-1

On the extrusion running between the two chassis is another 3-D printed chassis that is designed to move in the y direction supported by four v-slot wheels with bearings, two on the top of the extrusion and two on the bottom. The design we based our central chassis on is seen in figure 3.3-2. Again, these wheels are spaced such that there is no torque on the chassis. At the top of this chassis are four slots in which timing belts can be secured and space for an electromagnet to be installed.

Figure 3.3-2

The most important portion of the mechanical setup is the timing belts which are secured to the central chassis and loop around a system of pulleys surrounding the interior then finally around either of the motors. A schematic of the belt wrapping can be seen in figure 3.3-3. Utilizing programming of the motors, which will be detailed later, the two motors move in synchronous and asynchronous patterns to cause movement in the x and y direction. Due to limitations beyond our control, the motor program was not fully tested with a complete core design.

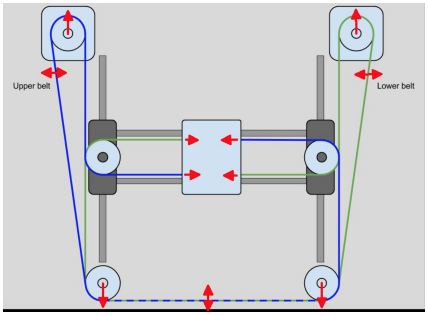
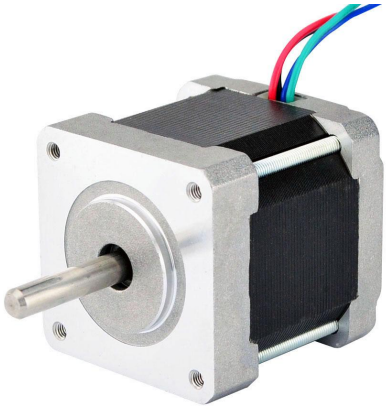


Figure 3.3-3

3.4 Motors

In order to implement our CoreXY system, we needed two motors in order to move the central chassis to its position underneath the specified grid square. For this purpose, we used stepper motors since they allow us to make very precise movements, down to the exact number of steps, allowing for easy fine tuning of piece movement. Through online research, we came across the Stepper Motor Nema 17. This device, seen in figure 3.4-1, is commonly used in 3D printing applications which also utilize the CoreXY method. Since the requirements for a 3D printer are very similar to ours for the chessboard, we assumed these motors would provide the necessary torque, speed, and precision. It also fit the necessary dimensions for our board size to easily be concealed beneath the board. These motors allow for rotation in both directions (CW or CCW) depending on input from the motor controller (and therefore from the raspberry pi). If both motors spin in the same direction, the chassis moves horizontally and if they spin in opposing directions the chassis is moved vertically. The motor has two pairs of wires corresponding to two sets of coils within the motor (figure 3.4-2). The powering of these wires is controlled by the motor controller to generate the necessary chassis motion.



.

Figure 3.4-1

Each motor is connected to its own motor controller (figure 3.4-3). These motor controllers (Shield Stepper Motor Driver V44 A3967) have five inputs: STEP, DIR, MS1, MS2, ENABLE. The STEP pin causes the stepper motor to move a single step per rising edge. The DIR pin sets the direction of motor rotation, either CW or CCW, depending on if this pin is pulled high or low. The ENABLE pin allows the motor controller to send signals to the four leads on the motor whenever it is held low. Since we always wanted our motors enabled, we tied this pin to ground. The MS1 and MS2 pins are for making microsteps. This level of precision is unnecessary for our use considering the 2mm step size of each motor step compared to the size of our board.

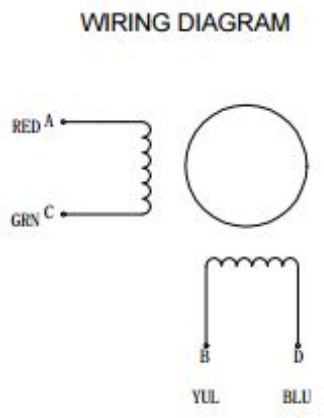


Figure 3.4-2

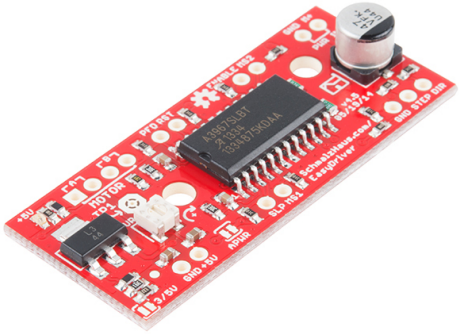


Figure 3.4-3

These motor controllers were operated by the RaspberryPi through a python file, cleanedRaspStepper.py (Appendix 8.17). This file takes in the motion requested by the main chess code and turns it into commands for the motor controller so the desired chassis movement is actuated. The main chess code sends the axis (horizontal or vertical), direction (up/down or left/right), and distance (number of grid squares) to the cleanedRaspStepper.py file which sends the necessary commands to the STEP and DIR pins on both of the motor controllers.

The diagram below (figure 3.4-4) shows how the RaspberryPi, motor controller, and stepper motor are all wired together.

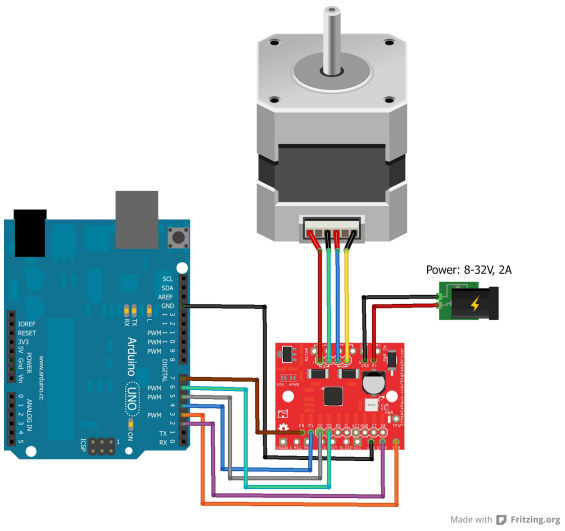
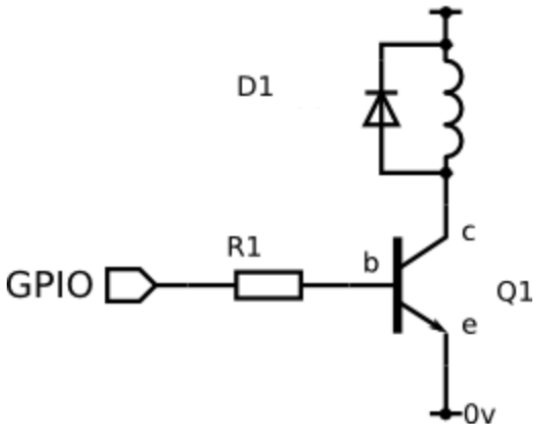


Figure 3.4-4

3.5 Magnet

The movement of the chess pieces will be controlled by an electromagnet fixed to the coreXY grid assembly. The electromagnet is controlled by the same program as the motors in order to turn on when the motor gets to its starting point and turn off once it gets to its destination. Unfortunately, since we were unable to begin combining the subsystems before having to leave campus, the electromagnet was unable to be tested combined with the motors. The magnet was able to be tested on its own and did function as expected.

The electromagnet assembly consisted of a circuit used for driving a relay. If we imagined the electromagnet as an inductor then the circuit we needed to accomplish this looked something like the picture below:



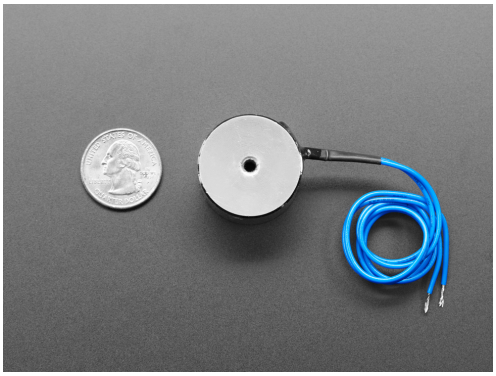
Where R1 is a resistor, Q1 is a transistor, and D1 is a diode. The inductor in the circuit represents the inductive load, electromagnetic as we mentioned earlier. This circuit protects the output from the voltage spike that occurs when inductive load is turned off. The diode prevents the voltage spike from surpassing the maximum rating of the transistor.

By using this circuit we are able to control the electromagnet using the raspberry pi as the GPIO and a 12V power supply for our power rail. The raspberry pi is a standard raspberry pi equipped to run python and the power supply is rigged to plug into a standard wall outlet and convert it to a 12V or 5V source. This fits well into our design because the motors will also be powered by the GPIO of the raspberry pi and the same power supply.

The components that make up the electromagnet circuit have some customization so long as they meet a few requirements. The transistor in the image is a BJT but we actually used a MOSFET instead. The resistor has to be fit to drive a gate voltage of about 4.5V for the transistor. The transistor itself needs to be able to handle a 12V drain to source voltage and a 1A current to be able to work in unison with the electromagnet.

The electromagnet we used is an adafruit 5V electromagnet. This magnet has a 10kg holding force which translates to being able to pick up about 4lbs. max. It could also only move ferromagnetic materials so I will include in this assembly the ferromagnetic metal disks we fixed

to the bottom of the chess pieces. Although it is a 5V electromagnet, we had to use the 12V power supply because we were moving the metal disks through a thin wood surface. Because of this separation between the magnet and what it was attracting, powering it with 12V worked better for our design. Pictured below are the electromagnet and the metal disks. The magnet is next to a quarter for reference and the disks are also about the size of a quarter.



3.6 Voice Control

Related to voice control, the main component worked during the semester was a Python code that is able to listen to voice commands via the microphone and output a string. Then, this string is used to power the movement of the pieces on the chess. The team came across several different possibilities and options for this audio to text conversion. Big technology companies, such as IBM, Google, Facebook, have their own Application programming interface (API) which can be used. Another option is specialized Python libraries which, different than some APIs, are free and unlimited. For this project, the team decided that the best option would be the API provided by Google. The reason for this choice is that Google provides a variety of tutorials and documentation online. From different research, most people point it to Google as being the most reliable and capable of converting audio to string.

Google provides a Python library called “speech_recognition” which can be easily downloaded. This process starts with the code connecting the microphone and recording the voice until the user stops speaking. From this point, the code can store the voice from the user. At this point, the team decided that there is no need to archive the voices commands, however, for the future it could be interesting as we could track how each player uses and provide analysis

and tips of how this user can improve. Related to Google's API we are using a generic key which is given by default by the library. This key is intended only for personal and testing purposes. For example, if the team decided that we would sell this product, we would have to acquire a paid key via Google services. It is also interesting to notice that when we use this generic key, there is a limit of 50 requests per day. During one of our tests, we used more than 50 requests in one day and we still could use the API. On a quick search, people mention that this limit is done in case there are too many requests. In other words, the limit exists but it is more flexible than 50 requests per day.

Related to the code, we created a python file called "Voice.py" that handles this request of Speech-to-Text. It is a fairly straightforward code that starts with the activation of the microphone which captures the speech until the user stops speaking. With this audio, we make the request to the API which returns a dictionary with three keys:

- "success": a boolean indicating whether or not the API request was successful
- "error": `None` if no error occurred, otherwise a string containing an error message if the API could not be reached or speech was unrecognizable
- "transcription": `None` if speech could not be transcribed,otherwise a string containing the transcribed text

As the microphone used does not provide the best audio quality, we also added a "Adjust for Ambient Noise" feature which automatically adjusts the sensitivity to ambient noise.

One example of this API is the user saying "Move Knight to C9". This string is treated by several functions to make it more reliable against words that we are not expecting. Our code looks for chess pieces' names (e.g. king, knight, queen) and a number which we know that will be after a letter. With this, the code can filter and return "Knight C9" which are the two pieces of information that we are interested in. One of the problems that we had is the API to understand "night" instead of "knight". This kind of problem was easily solved with string replacement which it receives, for example, "night" and replaces it with "knight". As we have more voice commands from the user, we would be able to create even better filters and replacements making, over time and usage, a more robust code.

3.7 Chess Code

Related with the tracking code, the team created several python files. For example, for each chess piece(e.g. Rook.py, Queen.py, Pawn.py) has their own python file. This part of the code is responsible to receive the current position of the piece intended to be moved and return all the possible positions to allow for this piece to move. The python code called “Piece.py” is responsible for moving the piece and also checking if there is any piece in the new position that should be taken out of the board. As a team, we tried to break in smaller code as much as possible enabling us to debug and figure it out where the problem could be.

The idea behind the main.py file is to make the call of the other files. Even though we tried to make this file as simple as possible, this is probably the most important and complex part of the code. It is also used to start the game and provide the options of 1 player vs. Ai or 1 player vs. another player. Also, it has to make the call to the voice API and trigger the motions of the step motors. The step motors have its unique python file that is called back and forth by the main.py. The main file also works close with the Board.py file which keeps track of the current status of the board.

The AI.py, as the name suggests, is responsible for receiving a movement from the user and trying, as best as possible, to return a movement that can defeat the user on the long run. For this project, we decided to use the Minimax approach. This approach creates a search tree from which the algorithm can choose the best move. This uses tree of all possible moves is explored to a given depth, and the position is evaluated at the ending “leaves” of the tree.After that, we return either the smallest or the largest value of the child to the parent node, depending on whether it’s a white or black to move. This depth for search can be chosen with n numbers of expansions. For this project, we have been using between 1 and 3 for testing. It is important to notice that after 3 it will require much more time to process all the possibilities. To make as fast as possible, we are using the multiprocessing python library that makes use of multi processors available on the computer. As you can imagine, the effectiveness of the minimax algorithm is heavily based on the search depth we can achieve. A future possibility could be the implementation of the Alpha-beta pruning which is an optimization method to the minimax

algorithm that allows us to disregard some branches in the search tree. This approach helps to evaluate the minimax search tree much deeper, while using the same resources.

3.8 Interfaces

The interfaces of the project are very minimal considering that it should be ready to use right out of the box. Some of these interfaces include an LCD testing screen for troubleshooting the program, a keyboard to troubleshoot the audio commands, and you can also connect a monitor to see a virtual display of the game to see how the pieces should be moving. Another interface is the power supply which is a simple wall connection to the power supply which steps it down to a 5V and 12V source that we can use to power everything we need.

4. System Integration Testing

4.1 Completed Testing.

Our final demonstration before moving to remote learning for the remainder of the semester was the subsystem demonstration. Here, we demonstrated each of the subsystems described in section 3 working independently of one another. The results of the testing will be summarized here. The agenda for the demonstration can be found on our team website under the agenda for 4 March 2020. The following list reflects the status of our subsystems as of our last group demonstration on this date:

- Electromagnet
    - Status: Working
        - Runs off of the RPi3
        - Switching circuit to handle new 10kg magnet (previous magnet did not generate strong enough field to penetrate the board and attract the piece)
        - Trouble with the weight of the pieces (might have to look into new pieces)
        - Need a gate voltage of 4V to operate the mosfet
- Chess Tracker Code
    - Status: Working, Partially Integrated

        - Integrated with voice recognition and motor driver code
        - Takes chess moves as inputs and calculates required distances to give to motor driver
- Voice Recognition Code
    - Status: Working, Integrated
        - Integrated with the Chess Tracker Code
        - Listens and takes voice input
- Motor Driver Code
    - Status: Working, Partially Integrated
        - Integrated with Chess Tracker Code
        - Can move the center piece in both the x and y directions to place magnet in desired place
- Microphone System
    - Status: In Progress, Not Integrated
        - Can record .wav files and play them through headphone jack on Pi
        - Write program that will output what the Voice Recognition code needs
- CoreXY
    - Status: In Progress
        - Use 3D printer in Hesburgh to efficiently print sliders
        - Adjust center bar and electromagnet holder
        - Finalize CoreXY system tweaks

4.2 Future Testing

In order to finalize the design of the chessboard all of the subsystems would need to be brought together to work in unity as one. In section 4.1 the status of all of the subsystems was shown to be at or near their desired end state. However, we still needed to connect all of these pieces which were functioning in isolation together. This was the plan for the second half of the spring semester following spring break.

Our first step would have been to get all of our main chess software to the point where it can take in a voice command through the RaspberryPi and output a series of commands to the motor controllers to execute that move. This would entail tying the voice command code, chess move calculation code, and motor controller code together into a single functional unit. The next step would have been to assemble the board so the magnet was mounted on the CoreXY chassis and its motion controlled by the stepper motor, all of which was secured beneath the playing surface and out of sight of the user. This step would have involved a heavy amount of physical building to construct the compartment for all of the electronics as well as mount the playing surface so that it is level and at the necessary height so the electromagnet can attract and drag the pieces across the board. We also theorized that the hard surface would need some friction-reducing treatment to allow the pieces to more smoothly and easily glide across it.

Had we been able to come back these are the steps we would have taken to complete our senior design project. Sadly, due to the cancellation of in-person classes for the remainder of our senior year, our project remains in the state that it was as described in section 4.1.

5. Users Manual/Installation Manual

Wizard's Chess ideally has very little initial setup. Everything needed to run the game is self contained within the chess board. The motors are fixed to the base of the board and they're connected to the coreXY within the confines of the board. The electromagnet is fixed to the coreXY. The individual chess pieces will already have the ferromagnetic disks attached.

Since the raspberry pi will already have the program for the motors, magnet, and chess tracker, there will be no necessary installation for the initial setup. All that is needed is to plug the board into any wall outlet and set the pieces on their respective tiles, the program would start to run. There will also be a reset button on the board incase of some malfunction in initialization of the board.

To tell if the board is on and functioning properly the user will hear the Harry Potter theme music playing from the speakers. The motors will also calibrate at the start. Once the

board is on, the pieces must be set on their correct tiles(If you are new to playing chess go to the Wizard's Chess website and click the "How To Play" link). The way to start is to make the first move. To make a move one must announce the name of the piece and the position they want it moved to, for example, "knight to A5." After making a move the board will be waiting for the opposite player to make their move. The game will end only once a checkmate is reached. To reset the game use the reset button located on the board.

Since we were unable to complete a finished model of the board there are a few other details that would have improved the user experience and help with troubleshooting. Some of these ideas include a "repeat move" option in case the magnet loses its connection to a piece mid move. An LCD display could also be helpful as a backup user interface if something is not working properly. It could display error messages if needed and at the very least alert the players of who's turn is next. The speaker would also be a finishing touch. It would play the theme music at the start of the game and also once checkmate is reached.

6. To-Market Design Changes

There are many different changes that could be made to the chessboard if it were going to market. In an ideal world, a functional prototype would have been made and then improvements could be suggested off of that. However, since this is not the case, this section will describe what the best possible outcome for this product could look like. There are many different subsystems of this product, however, this will touch on three main sections: the mechanics, the presentation and visual design, and the software.

Starting with the mechanics, currently, a coreXY system is being utilized with 3D printed plastic sliders and 3D printer sliding wheels and a timing belt. The entire system runs on two aluminum v-bars as the tracks. As this was being designed, significant issues arose on the functionality of this system, especially related to the imprecision of the 3-D printed pieces and the torque placed on the moving parts. In order to improve this, the 3-D printed pieces could be swapped for metal shop cut aluminum pieces that have been specified to the exact dimensions

needed for the board. That way the entire system could be mounted and run smoothly. This would add some additional weight, but not enough to significantly hamper movement. The wheels would also benefit from the precision of these custom cut pieces because they would sit exactly flush with the V-slot of the aluminum rail which would greatly improve the movement of the system. In regards to the magnet, some difficulty was encountered with getting the magnetic force required to move chess pieces, especially through the thin layer of birch plywood that was the board. To improve this a stronger magnet could be purchased and combined with either a thinner chessboard top or a top that has been coated in a substance like polyurethane to decrease the coefficient of friction.

As for visual improvements, the entire system could then be mounted inside a wooden case with a hinged lid. The hinged lid would allow for easy access to the mechanics for inspections and maintenance. This would also protect the mechanics of the system significantly more than it just being an open system. Additionally, a clear plexiglass top could be used if the user desired to see the magnet mechanism move and work as they played with the chessboard. It would just have to be engraved in such a way that simultaneously allows for visual of the chess squares, but does not hamper the movement of the pieces. As a final touch, an LCD display could be added into the side of the wooden chess box to display game stats such as time elapsed, whose turn it is etc.

As for the software, there are countless functionality improvements that could be added on. Currently, the system can recognize voices and interpret the specific words needed to move the chess pieces across the board. This, in turn, is the input to the motors which then calculate how far they need to turn and in what direction to accomplish the desired move. A functionality that was not added on in addition to the microphone, was a speaker in order to have the chessboard provide fun and engaging feedback on the game, similar to how the chess pieces in Harry Potter talk. If WiFi capabilities were added and a user interface, such as an app were developed, the two players could also play the game remotely, or just one of them could.

In summary, the proposed improvements are as follows:

- Mechanical
    - Custom Cut Aluminum Sliders
- Presentation
    - See-through Top
    - Chessboard Case and Mechanics Housing
    - Hinged Lid
    - LCD Display
- Software
    - Speaker Functionality
    - Wifi Functionality/App for Remote Gameplay

7. Conclusion

Wizard's Chess was meant to showcase all of the knowledge and skills developed over four years of electrical engineering undergraduate education at Notre Dame. Due to circumstances far beyond the control of any person, COVID-19, the final product is far short of what the team members had hoped and strived toward. Structurally, a rough first draft was fulfilled and in the process of being tweaked to function to the standards set for the final product. Each subsystem was brought to the point of functioning independently; the motors were connected to drivers and functionally rotated to correspond to various directional movements, a couple of electromagnets were tested to select the best fit for magnetism through the board to a chess piece, and the voice recognition system was in the process of being downloaded to function with a microphone circuit in the chess board. Moving beyond the physical aspects, code was developed for every aspect of the process, from basic chess mechanics and tracking to voice recognition and interpretation. Overall, the parts were all in place to bring together a useful and fully functional final product. Our hope is that our documentation and detailed reports have set the groundwork for future students to complete what we were unfortunately prohibited from

finishing. *Wizard's Chess* has the potential to be an amazing showcase of ingenuity, creativity, and overall a fun way to play chess.

8. Appendices

8.1 Voice.py

```
import speech_recognition as sr

#%%
'''
   // Voice Recognition (Speech-to-Text) - Google Speech Recognition
API
   -> This API converts spoken text (microphone) into written text
(Python strings)
   -> Personal or testing purposes only
   -> Generic key is given by default (it may be revoked by Google at
any time)
   -> If using API key, quota for your own key is 50 requests per day
'''

#%%

def recognize_speech_from_mic(recognizer, microphone):
    """Transcribe speech from recorded from `microphone`.

    Returns a dictionary with three keys:
    "success": a boolean indicating whether or not the API request
was
               successful
    "error":   `None` if no error occured, otherwise a string
containing
               an error message if the API could not be reached or
               speech was unrecognizable
    "transcription": `None` if speech could not be transcribed,
```

```
                otherwise a string containing the transcribed text
    """
    # check that recognizer and microphone arguments are appropriate
type
    if not isinstance(recognizer, sr.Recognizer):
        raise TypeError("`recognizer` must be `Recognizer` instance")

    if not isinstance(microphone, sr.Microphone):
        raise TypeError("`microphone` must be `Microphone` instance")

    # adjust the recognizer sensitivity to ambient noise and record
audio
    # from the microphone
    with microphone as source:
        recognizer.adjust_for_ambient_noise(source) # #  analyze the
audio source for 1 second
        audio = recognizer.listen(source)

    # set up the response object
    response = {
        "success": True,
        "error": None,
        "transcription": None
    }

    # try recognizing the speech in the recording
    # if a RequestError or UnknownValueError exception is caught,
    #   update the response object accordingly
    try:
        response["transcription"] =
recognizer.recognize_google(audio)
    except sr.RequestError:
        # API was unreachable or unresponsive
        response["success"] = False
        response["error"] = "API unavailable/unresponsive"
    except sr.UnknownValueError:
        # speech was unintelligible
        response["error"] = "Unable to recognize speech"
```

```
    return response

#%%


def main():
    recognizer = sr.Recognizer()
    mic = sr.Microphone(device_index=1)
    response = recognize_speech_from_mic(recognizer, mic)
    print('\nSuccess : {}\nError   : {}\n\nText from
Speech\n{}\n\n{}' \
          .format(response['success'],
                  response['error'],
                  '-'*17,
                  response['transcription']))
    return(response['transcription'])

if __name__ == "__main__":
    main()
```

8.2 termcolor.py

```
# coding: utf-8
# Copyright (c) 2008-2011 Volvox Development Team
#
# Permission is hereby granted, free of charge, to any
person obtaining a copy
# of this software and associated documentation files (the
"Software"), to deal
# in the Software without restriction, including without
limitation the rights
# to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell
# copies of the Software, and to permit persons to whom the
```

```
Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice
shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN
# THE SOFTWARE.
#
# Author: Konstantin Lepa <konstantin.lepa@gmail.com>

"""ANSII Color formatting for output in terminal."""

from __future__ import print_function
import os


__ALL__ = ['colored', 'cprint']

VERSION = (1, 1, 0)
```

```
ATTRIBUTES = dict(
    list(
        zip(
            [
                'bold',
                'dark',
                '',
                'underline',
                'blink',
                '',
                'reverse',
                'concealed'
            ],
            list(range(1, 9))
        )
    )
)
del ATTRIBUTES['']

HIGHLIGHTS = dict(
    list(
        zip(
            [
                'on_grey',
                'on_red',
                'on_green',
                'on_yellow',
                'on_blue',
                'on_magenta',
                'on_cyan',
                'on_white'
```

```
            ],
            list(range(40, 48))
        )
    )
)

COLORS = dict(
    list(
        zip(
            [
                'grey',
                'red',
                'green',
                'yellow',
                'blue',
                'magenta',
                'cyan',
                'white',
            ],
            list(range(30, 38))
        )
    )
)

RESET = '\033[0m'


def colored(text, color=None, on_color=None, attrs=None):
    """Colorize text.

    Available text colors:
        red, green, yellow, blue, magenta, cyan, white.
```

```
    Available text highlights:
        on_red, on_green, on_yellow, on_blue, on_magenta,
on_cyan, on_white.

    Available attributes:
        bold, dark, underline, blink, reverse, concealed.

    Example:
        colored('Hello, World!', 'red', 'on_grey', ['blue',
'blink'])
        colored('Hello, World!', 'green')
    """
    if os.getenv('ANSI_COLORS_DISABLED') is None:
        fmt_str = '\033[%dm%s'
        if color is not None:
            text = fmt_str % (COLORS[color], text)

        if on_color is not None:
            text = fmt_str % (HIGHLIGHTS[on_color], text)

        if attrs is not None:
            for attr in attrs:
                text = fmt_str % (ATTRIBUTES[attr], text)

        text += RESET
    return text


def cprint(text, color=None, on_color=None, attrs=None,
**kwargs):
    """Print colorize text.
```

```
    It accepts arguments of print function.
    """

    print((colored(text, color, on_color, attrs)),
**kwargs)


if __name__ == '__main__':
    print('Current terminal type: %s' % os.getenv('TERM'))
    print('Test basic colors:')
    cprint('Grey color', 'grey')
    cprint('Red color', 'red')
    cprint('Green color', 'green')
    cprint('Yellow color', 'yellow')
    cprint('Blue color', 'blue')
    cprint('Magenta color', 'magenta')
    cprint('Cyan color', 'cyan')
    cprint('White color', 'white')
    print(('-' * 78))

    print('Test highlights:')
    cprint('On grey color', on_color='on_grey')
    cprint('On red color', on_color='on_red')
    cprint('On green color', on_color='on_green')
    cprint('On yellow color', on_color='on_yellow')
    cprint('On blue color', on_color='on_blue')
    cprint('On magenta color', on_color='on_magenta')
    cprint('On cyan color', on_color='on_cyan')
    cprint('On white color', color='grey',
on_color='on_white')
    print('-' * 78)
```

```
    print('Test attributes:')
    cprint('Bold grey color', 'grey', attrs=['bold'])
    cprint('Dark red color', 'red', attrs=['dark'])
    cprint('Underline green color', 'green',
attrs=['underline'])
    cprint('Blink yellow color', 'yellow', attrs=['blink'])
    cprint('Reversed blue color', 'blue',
attrs=['reverse'])
    cprint('Concealed Magenta color', 'magenta',
attrs=['concealed'])
    cprint('Bold underline reverse cyan color', 'cyan',
           attrs=['bold', 'underline', 'reverse'])
    cprint('Dark blink concealed white color', 'white',
           attrs=['dark', 'blink', 'concealed'])
    print(('-' * 78))

    print('Test mixing:')
    cprint('Underline red on grey color', 'red', 'on_grey',
           ['underline'])
    cprint('Reversed green on red color', 'green',
'on_red', ['reverse'])
```

8.3 Rook.py

```
from Piece import Piece
from Coordinate import Coordinate as C

WHITE = True
BLACK = False
```

```
class Rook (Piece):

    stringRep = 'R'
    value = 5

    def __init__(self, board, side, position,
movesMade=0):
        super(Rook, self).__init__(board, side, position)
        self.movesMade = movesMade

    def getPossibleMoves(self):
        currentPosition = self.position

        directions = [C(0, 1), C(0, -1), C(1, 0), C(-1, 0)]
        for direction in directions:
            for move in
self.movesInDirectionFromPos(currentPosition,

direction, self.side):
                yield move
```

8.4 Queen.py

```
from Piece import Piece
from Coordinate import Coordinate as C

WHITE = True
BLACK = False
```

```
class Queen(Piece):

    stringRep = 'Q'
    value = 9

    def __init__(self, board, side, position, movesMade=0):
        super(Queen, self).__init__(board, side, position)
        self.movesMade = movesMade

    def getPossibleMoves(self):
        currentPosition = self.position

        directions = [C(0, 1), C(0, -1), C(1, 0), C(-1, 0),
C(1, 1),
                      C(1, -1), C(-1, 1), C(-1, -1)]
        for direction in directions:
            for move in
self.movesInDirectionFromPos(currentPosition,

direction, self.side):
                yield move
```

8.5 Piece.py

```
from Coordinate import Coordinate as C
from Move import Move
```

```
WHITE = True
BLACK = False
X = 0
Y = 1


class Piece:

    def __init__(self, board, side, position, movesMade=0):
        self.board = board
        self.side = side
        self.position = position
        self.movesMade = 0

    def __str__(self):
        sideString = 'White' if self.side == WHITE else
'Black'
        return 'Type : ' + type(self).__name__ + \
               ' - Position : ' + str(self.position) + \
               " - Side : " + sideString + \
               ' -- Value : ' + str(self.value) + \
               " -- Moves made : " + str(self.movesMade)

    def movesInDirectionFromPos(self, pos, direction,
side):
        for dis in range(1, 8):
            movement = C(dis * direction[X], dis *
direction[Y])
            newPos = pos + movement
            if self.board.isValidPos(newPos):
                pieceAtNewPos =
self.board.pieceAtPosition(newPos)
```

```
                if pieceAtNewPos is None:
                    yield Move(self, newPos)

                elif pieceAtNewPos is not None:
                    if pieceAtNewPos.side != side:
                        yield Move(self, newPos,
pieceToCapture=pieceAtNewPos)
                    return

    def __eq__(self, other):
        if self.board == other.board and \
           self.side == other.side and \
           self.position == other.position and \
           self.__class__ == other.__class__:
            return True
        return False

    def copy(self):
        cpy = self.__class__(self.board, self.side,
self.position,
                             movesMade=self.movesMade)
        return cpy
```

8.6 Pawn.py

```
from Rook import Rook
from Bishop import Bishop
from Knight import Knight
from Queen import Queen

from Piece import Piece
from Coordinate import Coordinate as C
from Move import Move

WHITE = True
BLACK = False


class Pawn(Piece):

    stringRep = 'P'
    value = 1

    def __init__(self, board, side, position,
movesMade=0):
        super(Pawn, self).__init__(board, side, position)
        self.movesMade = movesMade

    # @profile
    def getPossibleMoves(self):
        currentPosition = self.position

        # Pawn moves one up
        movement = C(0, 1) if self.side == WHITE else C(0,
-1)
        advanceOnePosition = currentPosition + movement
        if self.board.isValidPos(advanceOnePosition):
```

```
            # Promotion moves
            if
self.board.pieceAtPosition(advanceOnePosition) is None:
                col = advanceOnePosition[1]
                if col == 7 or col == 0:
                    piecesForPromotion = \
                        [Rook(self.board, self.side,
advanceOnePosition),
                         Knight(self.board, self.side,
advanceOnePosition),
                         Bishop(self.board, self.side,
advanceOnePosition),
                         Queen(self.board, self.side,
advanceOnePosition)]
                    for piece in piecesForPromotion:
                        move = Move(self,
advanceOnePosition)
                        move.promotion = True
                        move.specialMovePiece = piece
                        yield move
                else:
                    yield Move(self, advanceOnePosition)

        # Pawn moves two up
        if self.movesMade == 0:
            movement = C(0, 2) if self.side == WHITE else
C(0, -2)
            advanceTwoPosition = currentPosition + movement
            if self.board.isValidPos(advanceTwoPosition):
                if
self.board.pieceAtPosition(advanceTwoPosition) is None and
\
```

```
self.board.pieceAtPosition(advanceOnePosition) is None:
                    yield Move(self, advanceTwoPosition)

        # Pawn takes
        movements = [C(1, 1), C(-1, 1)] \
            if self.side == WHITE else [C(1, -1), C(-1,
-1)]

        for movement in movements:
            newPosition = self.position + movement
            if self.board.isValidPos(newPosition):
                pieceToTake =
self.board.pieceAtPosition(newPosition)
                if pieceToTake and pieceToTake.side !=
self.side:
                    col = newPosition[1]
                    # Promotions
                    if col == 7 or col == 0:
                        piecesForPromotion = \
                            [Rook(self.board, self.side,
newPosition),
                             Knight(self.board, self.side,
newPosition),
                             Bishop(self.board, self.side,
newPosition),
                             Queen(self.board, self.side,
newPosition)]
                        for piece in piecesForPromotion:
                            move = Move(self, newPosition,
pieceToCapture=pieceToTake)
                            move.promotion = True
```

```
                            move.specialMovePiece = piece
                            yield move
                    else:
                        yield Move(self, newPosition,
pieceToCapture=pieceToTake)

        # En passant
        movements = [C(1, 1), C(-1, 1)] \
            if self.side == WHITE else [C(1, -1), C(-1,
-1)]
        for movement in movements:
            posBesidePawn = self.position + C(movement[0],
0)
            if self.board.isValidPos(posBesidePawn):
                pieceBesidePawn =
self.board.pieceAtPosition(posBesidePawn)
                lastPieceMoved =
self.board.getLastPieceMoved()
                lastMoveWasAdvanceTwo = False
                lastMove = self.board.getLastMove()

                if lastMove:
                    if lastMove.newPos - lastMove.oldPos ==
C(0, 2) or \
                       lastMove.newPos - lastMove.oldPos ==
C(0, -2):
                        lastMoveWasAdvanceTwo = True

                if pieceBesidePawn and \
                   pieceBesidePawn.stringRep == 'P' and \
                   pieceBesidePawn.side != self.side and \
```

```
                    lastPieceMoved is pieceBesidePawn and \
                    lastMoveWasAdvanceTwo:
                     move = Move(self, self.position +
movement,

pieceToCapture=pieceBesidePawn)
                     move.passant = True
                     move.specialMovePiece = pieceBesidePawn
                     yield move
```

8.7 MoveNode.py

```
class MoveNode:

    def __init__(self, move, children, parent):
        self.move = move
        self.children = children
        self.parent = parent
        self.pointAdvantage = None
        self.depth = 1

    def __str__(self):
        stringRep = "Move : " + str(self.move) + \
                    " Point advantage : " +
str(self.pointAdvantage) + \
                    " Checkmate : " +
str(self.move.checkmate)
        stringRep += "\n"

        for child in self.children:
```

```
            stringRep += " " * self.getDepth() * 4
            stringRep += str(child)

        return stringRep

    def __gt__(self, other):
        if self.move.checkmate and not
other.move.checkmate:
            return True
        if not self.move.checkmate and
other.move.checkmate:
            return False
        if self.move.checkmate and other.move.checkmate:
            return False
        return self.pointAdvantage > other.pointAdvantage

    def __lt__(self, other):
        if self.move.checkmate and not
other.move.checkmate:
            return False
        if not self.move.checkmate and
other.move.checkmate:
            return True
        if self.move.stalemate and other.move.stalemate:
            return False
        return self.pointAdvantage < other.pointAdvantage

    def __eq__(self, other):
        if self.move.checkmate and other.move.checkmate:
            return True
        return self.pointAdvantage == other.pointAdvantage
```

```
    def getHighestNode(self):
        highestNode = self
        while True:
            if highestNode.parent is not None:
                highestNode = highestNode.parent
            else:
                return highestNode

    def getDepth(self):
        depth = 1
        highestNode = self
        while True:
            if highestNode.parent is not None:
                highestNode = highestNode.parent
                depth += 1
            else:
                return depth
```

8.8 Move.py

```
class Move:

    def __init__(self, piece, newPos, pieceToCapture=None):
        self.notation = None
        self.check = False
        self.checkmate = False
        self.kingsideCastle = False
```

```
        self.queensideCastle = False
        self.promotion = False
        self.passant = False
        self.stalemate = False

        self.piece = piece
        self.oldPos = piece.position
        self.newPos = newPos
        self.pieceToCapture = pieceToCapture
        # For en passant and castling
        self.specialMovePiece = None
        # For castling
        self.rookMove = None

    def __str__(self):
        displayString = 'Old pos : ' + str(self.oldPos) + \
                        ' -- New pos : ' + str(self.newPos)
        if self.notation:
            displayString += ' Notation : ' + self.notation
        if self.passant:
            displayString = 'Old pos : ' + str(self.oldPos)
+ \
                            ' -- New pos : ' +
str(self.newPos) + \
                            ' -- Pawn taken : ' +
str(self.specialMovePiece)
            displayString += ' PASSANT'
        return displayString

    def __eq__(self, other):
        if self.oldPos == other.oldPos and \
           self.newPos == other.newPos and \
```

```
            self.specialMovePiece == other.specialMovePiece:
             if not self.specialMovePiece:
                 return True
             if self.specialMovePiece and \
                self.specialMovePiece ==
other.specialMovePiece:
                 return True
             else:
                 return False
         else:
             return False

    def __hash__(self):
         return hash((self.oldPos, self.newPos))

    def reverse(self):
         return Move(self.piece, self.piece.position,
                     pieceToCapture=self.pieceToCapture)
```

8.9 main.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from Board import Board
from InputParser import InputParser
from AI import AI
import sys
```

```
import random
import Voice
import speech_recognition as sr

import colorama
colorama.init()

from word2number import w2n

WHITE = True
BLACK = False




#%%
'''
   // Voice Recognition (Speech-to-Text) - Google Speech
Recognition API
   -> This API converts spoken text (microphone) into
written text (Python strings)
   -> Personal or testing purposes only
   -> Generic key is given by default (it may be revoked by
Google at any time)
   -> If using API key, quota for your own key is 50
requests per day
'''

#%%

def recognize_speech_from_mic(recognizer, microphone):
```

```
    """Transcribe speech from recorded from `microphone`.

    Returns a dictionary with three keys:
    "success": a boolean indicating whether or not the API
request was
               successful
    "error":   `None` if no error occured, otherwise a
string containing
               an error message if the API could not be
reached or
               speech was unrecognizable
    "transcription": `None` if speech could not be
transcribed,
               otherwise a string containing the
transcribed text
    """
    # check that recognizer and microphone arguments are
appropriate type
    if not isinstance(recognizer, sr.Recognizer):
        raise TypeError("`recognizer` must be `Recognizer`
instance")

    if not isinstance(microphone, sr.Microphone):
        raise TypeError("`microphone` must be `Microphone`
instance")

    # adjust the recognizer sensitivity to ambient noise
and record audio
    # from the microphone
    with microphone as source:
        recognizer.adjust_for_ambient_noise(source) # #
analyze the audio source for 1 second
```

```
        audio = recognizer.listen(source)

    # set up the response object
    response = {
        "success": True,
        "error": None,
        "transcription": None
    }

    # try recognizing the speech in the recording
    # if a RequestError or UnknownValueError exception is
caught,
    #   update the response object accordingly
    try:
        response["transcription"] =
recognizer.recognize_google(audio)
    except sr.RequestError:
        # API was unreachable or unresponsive
        response["success"] = False
        response["error"] = "API unavailable/unresponsive"
    except sr.UnknownValueError:
        # speech was unintelligible
        response["error"] = "Unable to recognize speech"

    return response

#%%

# This will recieve the string said by the user
# This function has to convert/filter this string to a
specific chess movement
def translate_voice_string_to_chess_command(raw_string):
```

```
    string_to_parse = raw_string

    string_to_parse = string_to_parse.lower() # make the
whole string lowercase;

    #temp = re.findall(r'\d+', string_to_parse) # This will
get all the different numbers
    #numbers = list(map(int, temp))

    # Get the position to be moved to:
    # Look for the space (" ") and get until the end of the
string
    # E.g. Pawn a3 -> It should get a3
    try:
        position = string_to_parse[string_to_parse.index("
")+1:]
    except:
        position = 99999

    king = string_to_parse.find("king")
    queen = string_to_parse.find("queen")
    bishop = string_to_parse.find("bishop")
    knight = string_to_parse.find("knight")
    night = string_to_parse.find("night")
    rook = string_to_parse.find("rook")
    pawn = string_to_parse.find("pawn")

    if (king != -1 and position !=99999):
        movement = "K" + position
    elif (queen != -1 and position !=99999):
        movement = "Q" + position
```

```
    elif (bishop != -1 and position !=99999):
        movement = "B" + position
    elif ((knight != -1 and position !=99999) or (night !=
-1 and position !=99999)):
        movement = "N" + position
    elif (rook != -1 and position !=99999):
        movement = "R" + position
    elif (pawn != -1 and position !=99999):
        movement = position
    else:
        movement = "not found"

    return movement

def prepare_xphysical_movement(pos_origin,pos_destiny):
    xaxis_difference = pos_destiny[0] - pos_origin[0]

    # Calculating Direction (i.e. if goes right or left)
    if xaxis_difference < 1: # If true, this negative value
means that we move to left
        dir = 0 # 0 = left
    else:
        dir = 1 # 1 = right

    dist = abs(xaxis_difference) # Number of squares to
move
    axis = 0 # This will move on horizontal

    return axis,dir,dist

def prepare_yphysical_movement(pos_origin,pos_destiny):
    yaxis_difference = pos_destiny[1] - pos_origin[1]
```

```
    # Calculating Direction (i.e. if goes up or down)
    if yaxis_difference < 1: # If true, this negative value
means that we move down
        dir = 0 # 0 = move down
    else:
        dir = 1 # 1 = move up

    dist = abs(yaxis_difference) # Number of squares to
move
    axis = 1 # This will move on vertical

    return axis,dir,dist


def askForPlayerSide():
    playerChoiceInput = input(
        "What side would you like to play as [wB]?
").lower()
    if 'w' in playerChoiceInput:
        print("You will play as white")
        return WHITE
    else:
        print("You will play as black")
        return BLACK


def askForDepthOfAI():
    depthInput = 2
    try:
        depthInput = int(input("How deep should the AI look
for moves?\n"
```

```
                                    "Warning : values above 3
will be very slow."
                                    " [2]? "))
    except KeyboardInterrupt:
        sys.exit()
    except:
        print("Invalid input, defaulting to 2")
    return depthInput


def printCommandOptions():
    undoOption = 'u : undo last move'
    printLegalMovesOption = 'l : show all legal moves'
    randomMoveOption = 'r : make a random move'
    quitOption = 'quit : resign'
    moveOption = 'a3, Nc3, Qxa2, etc : make the move'
    options = [undoOption, printLegalMovesOption,
randomMoveOption,
               quitOption, moveOption, '', ]
    print('\n'.join(options))


def printAllLegalMoves(board, parser):
    for move in
parser.getLegalMovesWithNotation(board.currentSide,
short=True):
        print(move.notation)


def getRandomMove(board, parser):
    legalMoves = board.getAllMovesLegal(board.currentSide)
    randomMove = random.choice(legalMoves)
```

```
    randomMove.notation =
parser.notationForMove(randomMove)
    return randomMove


def makeMove(move, board):
    print("Making move : " + move.notation)
    board.makeMove(move)


def printPointAdvantage(board):
    print("Currently, the point difference is : " +

str(board.getPointAdvantageOfSide(board.currentSide)))


def undoLastTwoMoves(board):
    if len(board.history) >= 2:
        board.undoLastMove()
        board.undoLastMove()


def startGame(board, playerSide, ai):
    parser = InputParser(board, playerSide)
    while True:
        print()
        print(board)
        print()
        if board.isCheckmate():
            if board.currentSide == playerSide:
                print("Checkmate, you lost")
            else:
```

```
                print("Checkmate! You won!")
            return

        if board.isStalemate():
            if board.currentSide == playerSide:
                print("Stalemate")
            else:
                print("Stalemate")
            return

        if board.currentSide == playerSide:
            # printPointAdvantage(board)
            move = None
            command = input("It's your move."
                            " Type '?' for options. ? ")


            # Block for getting the voice from the user
            recognizer = sr.Recognizer()
            mic = sr.Microphone(device_index=1)
            print("Start Listening")
            response =
recognize_speech_from_mic(recognizer, mic)
            print("After response Listening")
            print('\nSuccess : {}\nError   : {}\n\nText
from Speech\n{}\n\n{}' \
                  .format(response['success'],
                          response['error'],
                          '-'*17,
                          response['transcription']))
            full_voice_command = response['transcription']
            print("End Listening")
```

```
            # End: Block for getting the voice from the
user

            # Make the necessary treatments to get from the
voice string the correct movement
            move_to_be_made =
translate_voice_string_to_chess_command(full_voice_command)


            if command.lower() == 'u':
                undoLastTwoMoves(board)
                continue
            elif command.lower() == '?':
                printCommandOptions()
                continue
            elif command.lower() == 'l':
                printAllLegalMoves(board, parser)
                continue
            elif command.lower() == 'r':
                move = getRandomMove(board, parser)
            elif command.lower() == 'exit' or
command.lower() == 'quit':
                return
            try:
                move = parser.parse(move_to_be_made) #
originally this was parser.parse(command)
            except ValueError as error:
                print("%s" % error)
                continue
            makeMove(move, board)

            # Make Luke's part here:
```

```
            print(move.oldPos)
            print(move.newPos)

            xaxis, xdir, xdist =
prepare_xphysical_movement(move.oldPos,move.newPos)
            yaxis, ydir, ydist =
prepare_yphysical_movement(move.oldPos,move.newPos)

            print("Here")

        else:
            print("AI thinking...")
            move = ai.getBestMove()
            move.notation = parser.notationForMove(move)
            makeMove(move, board)

def twoPlayerGame(board):
    parserWhite = InputParser(board, WHITE)
    parserBlack = InputParser(board, BLACK)
    while True:
        print()
        print(board)
        print()
        if board.isCheckmate():
            print("Checkmate")
            return

        if board.isStalemate():
            print("Stalemate")
            return

        # printPointAdvantage(board)
```

```
        if board.currentSide == WHITE:
            parser = parserWhite
        else:
            parser = parserBlack
        move = None
        command = input("It's your move,
{}.".format(board.currentSideRep()) + \
                        " Type '?' for options. ? ")



        if command.lower() == 'u':
            undoLastTwoMoves(board)
            continue
        elif command.lower() == '?':
            printCommandOptions()
            continue
        elif command.lower() == 'l':
            printAllLegalMoves(board, parser)
            continue
        elif command.lower() == 'r':
            move = getRandomMove(board, parser)
        elif command.lower() == 'exit' or command.lower()
== 'quit':
            return
        try:
            move = parser.parse(command)
        except ValueError as error:
            print("%s" % error)
            continue
        makeMove(move, board)
```

```
board = Board()

try:
    if len(sys.argv) >= 2 and sys.argv[1] == "--two":
        twoPlayerGame(board)
    else:
        playerSide = askForPlayerSide()
        print()
        aiDepth = askForDepthOfAI()
        opponentAI = AI(board, not playerSide, aiDepth)
        startGame(board, playerSide, opponentAI)
except KeyboardInterrupt:
    sys.exit()
```

8.10 Knight.py

```
from Piece import Piece
from Coordinate import Coordinate as C
from Move import Move

WHITE = True
BLACK = False


class Knight(Piece):

    stringRep = 'N'
    value = 3
```

```
    def __init__(self, board, side, position,
movesMade=0):
        super(Knight, self).__init__(board, side, position)
        self.movesMade = movesMade

    def getPossibleMoves(self):
        board = self.board
        currentPos = self.position
        movements = [C(2, 1), C(2, -1), C(-2, 1), C(-2,
-1), C(1, 2),
                     C(1, -2), C(-1, -2), C(-1, 2)]
        for movement in movements:
            newPos = currentPos + movement
            if board.isValidPos(newPos):
                pieceAtNewPos =
board.pieceAtPosition(newPos)
                if pieceAtNewPos is None:
                    yield Move(self, newPos)
                elif pieceAtNewPos.side != self.side:
                    yield Move(self, newPos,
pieceToCapture=pieceAtNewPos)
```

8.11 King.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from Piece import Piece
```

```
from Move import Move
from Coordinate import Coordinate as C

WHITE = True
BLACK = False


class King (Piece):

    stringRep = 'K'
    value = 100

    def __init__(self, board, side, position,
movesMade=0):
        super(King, self).__init__(board, side, position)
        self.movesMade = movesMade

    def getPossibleMoves(self):
        currentPos = self.position
        movements = [C(0, 1), C(0, -1), C(1, 0), C(-1, 0),
C(1, 1),
                     C(1, -1), C(-1, 1), C(-1, -1)]
        for movement in movements:
            newPos = currentPos + movement
            if self.board.isValidPos(newPos):
                pieceAtNewPos =
self.board.pieceAtPosition(newPos)
                if self.board.pieceAtPosition(newPos) is
None:
                    yield Move(self, newPos)
                elif pieceAtNewPos.side != self.side:
                    yield Move(self, newPos,
```

```
pieceToCapture=pieceAtNewPos)

        # Castling
        if self.movesMade == 0:
            inCheck = False
            kingsideCastleBlocked = False
            queensideCastleBlocked = False
            kingsideCastleCheck = False
            queensideCastleCheck = False
            kingsideRookMoved = True
            queensideRookMoved = True

            kingsideCastlePositions = [self.position + C(1,
0),
                                       self.position + C(2,
0)]
            for pos in kingsideCastlePositions:
                if self.board.pieceAtPosition(pos):
                    kingsideCastleBlocked = True
                    break

            queensideCastlePositions = [self.position -
C(1, 0),
                                        self.position -
C(2, 0),
                                        self.position -
C(3, 0)]
            for pos in queensideCastlePositions:
                if self.board.pieceAtPosition(pos):
                    queensideCastleBlocked = True
                    break
```

```
            if kingsideCastleBlocked and
queensideCastleBlocked:
                return

            otherSideMoves = \
                self.board.getAllMovesUnfiltered(not
self.side,

includeKing=False)
            for move in otherSideMoves:
                if move.newPos == self.position:
                    inCheck = True
                    break
                if move.newPos == self.position + C(1, 0)
or \
                   move.newPos == self.position + C(2, 0):
                    kingsideCastleCheck = True
                if move.newPos == self.position - C(1, 0)
or \
                   move.newPos == self.position - C(2, 0):
                    queensideCastleCheck = True

            kingsideRookPos = self.position + C(3, 0)
            kingsideRook =
self.board.pieceAtPosition(kingsideRookPos) \
                if self.board.isValidPos(kingsideRookPos) \
                else None
            if kingsideRook and \
               kingsideRook.stringRep == 'R' and \
               kingsideRook.movesMade == 0:
                kingsideRookMoved = False
```

```
            queensideRookPos = self.position - C(4, 0)
            queensideRook =
self.board.pieceAtPosition(queensideRookPos) \
                if self.board.isValidPos(queensideRookPos)
\
                else None
            if queensideRook and \
               queensideRook.stringRep == 'R' and \
               queensideRook.movesMade == 0:
                queensideRookMoved = False

            if not inCheck:
                if not kingsideCastleBlocked and \
                   not kingsideCastleCheck and \
                   not kingsideRookMoved:
                    move = Move(self, self.position + C(2,
0))
                    rookMove = Move(kingsideRook,
self.position + C(1, 0))
                    move.specialMovePiece = \

self.board.pieceAtPosition(kingsideRookPos)
                    move.kingsideCastle = True
                    move.rookMove = rookMove
                    yield move
                if not queensideCastleBlocked and \
                   not queensideCastleCheck and \
                   not queensideRookMoved:
                    move = Move(self, self.position - C(2,
0))
                    rookMove = Move(queensideRook,
self.position - C(1, 0))
```

```
                    move.specialMovePiece = \

self.board.pieceAtPosition(queensideRookPos)
                    move.queensideCastle = True
                    move.rookMove = rookMove
                    yield move
```

8.12 InputParser.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import re
from Pawn import Pawn


class InputParser:

    def __init__(self, board, side):
        self.board = board
        self.side = side

    def parse(self, humanInput):
        regexCoordinateNotation =
re.compile('(?i)[a-h][1-8][a-h][1-8][QRBN]?')
        if regexCoordinateNotation.match(humanInput):
            return
self.moveForCoordinateNotation(humanInput)
        regexAlgebraicNotation =
re.compile('(?i)0-0|0-0-0|(?:[KQRBNP]?[a-h]?[1-8]?x?[a-h][1
-8]|[Pa-h]x?[a-h])(?:=?[QRBN])?')
```

```
        if regexAlgebraicNotation.match(humanInput):
            return
self.moveForShortAlgebraicNotation(humanInput)
        if re.compile('(?i)O-O|O-O-O').match(humanInput):
            return
self.moveForShortAlgebraicNotation(humanInput.upper().repla
ce("O","0"))
        raise ValueError("Invalid move: %s" % humanInput)

    def moveForCoordinateNotation(self, notation):
        for move in self.board.getAllMovesLegal(self.side):
            if
self.board.getCoordinateNotationOfMove(move).lower() ==
notation.lower():
                move.notation = self.notationForMove(move)
                return move
        raise ValueError("Illegal move: %s" % notation)

    # Only handles SAN, not long-algebraic or descriptive
    def moveForShortAlgebraicNotation(self, notation):
        shortNotation = notation.replace("x","")
        moves = self.getLegalMovesWithNotation(self.side,
False)
        for move in moves:
            if move.notation.replace("x","") ==
shortNotation: # Bxc3 versus bxc3
                return move
        for move in moves:
            if move.notation.replace("x","").lower() ==
shortNotation.lower():
                return move
        moves = self.getLegalMovesWithNotation(self.side,
```

```
True)
        for move in moves:
            if move.notation.replace("x","") ==
shortNotation: # Bxc3 versus bxc3
                return move
        for move in moves:
            if move.notation.replace("x","").lower() ==
shortNotation.lower():
                return move
        shortNotation =
notation.lower().replace("p","").replace("=","")
        if
re.compile('[a-h][1-8]?[qrbn]?').match(shortNotation):
            for move in moves:
                if type(move.piece) is Pawn and not
move.pieceToCapture and
self.board.getCoordinateNotationOfMove(move).replace("=",""
).lower().endswith(shortNotation):
                    return move
            for move in moves:
                if type(move.piece) is Pawn and not
move.pieceToCapture and re.sub("[1-8]", "",
self.board.getCoordinateNotationOfMove(move)).replace("="
"").lower().endswith(shortNotation):
                    return move # ASSUME lazy pawn move
(P)c is unambiguous
        shortNotation =
shortNotation.lower().replace("x","")
        if
re.compile('[a-h]?[a-h][1-8]?[qrbn]?').match(shortNotation)
:
            for move in moves:
```

```
                if type(move.piece) is Pawn and
move.pieceToCapture and
self.board.getCaptureNotation(move).replace("x","").lower()
.endswith(shortNotation):
                    return move # ASSUME lazier pawn
capture (P)b(x)c3 is unambiguous
            for move in moves:
                if type(move.piece) is Pawn and
move.pieceToCapture and re.sub("[1-8]", "",
self.board.getCaptureNotation(move).replace("x","")).lower(
).endswith(shortNotation):
                    return move # ASSUME laziest pawn
capture (P)b(x)c is unambiguous
        raise ValueError("Illegal move: %s" % notation)

    def notationForMove(self, move):
        side = self.board.getSideOfMove(move)
        moves = self.getLegalMovesWithNotation(side)
        for m in moves:
            if m == move:
                return m.notation

    def getLegalMovesWithNotation(self, side, short=True):
        moves = []
        for legalMove in self.board.getAllMovesLegal(side):
            moves.append(legalMove)
            legalMove.notation =
self.board.getAlgebraicNotationOfMove(legalMove, short)

        duplicateNotationMoves =
self.duplicateMovesFromMoves(moves)
        for duplicateMove in duplicateNotationMoves:
```

```
            duplicateMove.notation = \

self.board.getAlgebraicNotationOfMoveWithFile(duplicateMove
, short)

        duplicateNotationMoves =
self.duplicateMovesFromMoves(moves)
        for duplicateMove in duplicateNotationMoves:
            duplicateMove.notation = \

self.board.getAlgebraicNotationOfMoveWithRank(duplicateMove
, short)

        duplicateNotationMoves =
self.duplicateMovesFromMoves(moves)
        for duplicateMove in duplicateNotationMoves:
            duplicateMove.notation = \

self.board.getAlgebraicNotationOfMoveWithFileAndRank(duplic
ateMove, short)

        return moves

    def duplicateMovesFromMoves(self, moves):
        return list(filter(
            lambda move:
            len([m for m in moves if m.notation ==
move.notation]) > 1, moves))
```

8.13 Coordinate.py

```
class Coordinate(tuple):

    def __new__(cls, *args):
        return tuple.__new__(cls, args)

    def __reduce__(self):
        return (self.__class__, tuple(self))

    def __add__(self, other):
        return Coordinate(self[0] + other[0], self[1] +
other[1])

    def __sub__(self, other):
        return Coordinate(self[0] - other[0], self[1] -
other[1])
```

8.14 Board.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from Pawn import Pawn
from Rook import Rook
from King import King
from Bishop import Bishop
from Knight import Knight
```

```
from Queen import Queen
from Coordinate import Coordinate as C
from termcolor import colored

from Move import Move

WHITE = True
BLACK = False


class Board:

    def __init__(self, mateInOne=False, castleBoard=False,
                 passant=False, promotion=False):
        self.pieces = []
        self.history = []
        self.points = 0
        self.currentSide = WHITE
        self.movesMade = 0
        self.checkmate = False

        if not mateInOne and not castleBoard and not
passant and not promotion:
            self.pieces.extend([Rook(self, BLACK, C(0, 7)),
                                Knight(self, BLACK, C(1,
7)),
                                Bishop(self, BLACK, C(2,
7)),
                                Queen(self, BLACK, C(3,
7)),
                                King(self, BLACK, C(4, 7)),
                                Bishop(self, BLACK, C(5,
```

```
7)),
                                Knight(self, BLACK, C(6,
7)),
                                Rook(self, BLACK, C(7,
7))])
            for x in range(8):
                self.pieces.append(Pawn(self, BLACK, C(x,
6)))
            for x in range(8):
                self.pieces.append(Pawn(self, WHITE, C(x,
1)))
            self.pieces.extend([Rook(self, WHITE, C(0, 0)),
                                Knight(self, WHITE, C(1,
0)),
                                Bishop(self, WHITE, C(2,
0)),
                                Queen(self, WHITE, C(3,
0)),
                                King(self, WHITE, C(4, 0)),
                                Bishop(self, WHITE, C(5,
0)),
                                Knight(self, WHITE, C(6,
0)),
                                Rook(self, WHITE, C(7,
0))])

        elif promotion:
            pawnToPromote = Pawn(self, WHITE, C(1, 6))
            pawnToPromote.movesMade = 1
            kingWhite = King(self, WHITE, C(4, 0))
            kingBlack = King(self, BLACK, C(3, 2))
            self.pieces.extend([pawnToPromote, kingWhite,
```

```
kingBlack])

        elif passant:
            pawn = Pawn(self, WHITE, C(1, 4))
            pawn2 = Pawn(self, BLACK, C(2, 6))
            kingWhite = King(self, WHITE, C(4, 0))
            kingBlack = King(self, BLACK, C(3, 2))
            self.pieces.extend([pawn, pawn2, kingWhite,
kingBlack])
            self.history = []
            self.currentSide = BLACK
            self.points = 0
            self.movesMade = 0
            self.checkmate = False
            firstMove = Move(pawn2, C(2, 4))
            self.makeMove(firstMove)
            self.currentSide = WHITE
            return

    def __str__(self):
        return
self.wrapStringRep(self.makeStringRep(self.pieces))

    def undoLastMove(self):
        lastMove, pieceTaken = self.history.pop()

        if lastMove.queensideCastle or
lastMove.kingsideCastle:
            king = lastMove.piece
            rook = lastMove.specialMovePiece

            self.movePieceToPosition(king, lastMove.oldPos)
```

```
            self.movePieceToPosition(rook,
lastMove.rookMove.oldPos)

            king.movesMade -= 1
            rook.movesMade -= 1

        elif lastMove.passant:
            pawnMoved = lastMove.piece
            pawnTaken = pieceTaken
            self.pieces.append(pawnTaken)
            self.movePieceToPosition(pawnMoved,
lastMove.oldPos)
            pawnMoved.movesMade -= 1
            if pawnTaken.side == WHITE:
                self.points += 1
            if pawnTaken.side == BLACK:
                self.points -= 1

        elif lastMove.promotion:
            pawnPromoted = lastMove.piece
            promotedPiece =
self.pieceAtPosition(lastMove.newPos)
            self.pieces.remove(promotedPiece)
            if pieceTaken:
                if pieceTaken.side == WHITE:
                    self.points += pieceTaken.value
                if pieceTaken.side == BLACK:
                    self.points -= pieceTaken.value
                self.pieces.append(pieceTaken)
            self.pieces.append(pawnPromoted)
            if pawnPromoted.side == WHITE:
                self.points -= promotedPiece.value - 1
```

```
            elif pawnPromoted.side == BLACK:
                self.points += promotedPiece.value - 1
            pawnPromoted.movesMade -= 1

        else:
            pieceToMoveBack = lastMove.piece
            self.movePieceToPosition(pieceToMoveBack,
lastMove.oldPos)
            if pieceTaken:
                if pieceTaken.side == WHITE:
                    self.points += pieceTaken.value
                if pieceTaken.side == BLACK:
                    self.points -= pieceTaken.value
                self.addPieceToPosition(pieceTaken,
lastMove.newPos)
                self.pieces.append(pieceTaken)
            pieceToMoveBack.movesMade -= 1

        self.currentSide = not self.currentSide

    def isCheckmate(self):
        if len(self.getAllMovesLegal(self.currentSide)) ==
0:
            for move in self.getAllMovesUnfiltered(not
self.currentSide):
                pieceToTake = move.pieceToCapture
                if pieceToTake and pieceToTake.stringRep ==
"K":
                    return True
        return False

    def isStalemate(self):
```

```
        if len(self.getAllMovesLegal(self.currentSide)) ==
0:
            for move in self.getAllMovesUnfiltered(not
self.currentSide):
                pieceToTake = move.pieceToCapture
                if pieceToTake and pieceToTake.stringRep ==
"K":
                    return False
            return True
        return False

    def getLastMove(self):
        if self.history:
            return self.history[-1][0]

    def getLastPieceMoved(self):
        if self.history:
            return self.history[-1][0].piece

    def addMoveToHistory(self, move):
        pieceTaken = None
        if move.passant:
            pieceTaken = move.specialMovePiece
            self.history.append([move, pieceTaken])
            return
        pieceTaken = move.pieceToCapture
        if pieceTaken:
            self.history.append([move, pieceTaken])
            return

        self.history.append([move, None])
```

```
    def getCurrentSide(self):
        return self.currentSide

    def makeStringRep(self, pieces):
        stringRep = ''
        for y in range(7, -1, -1):
            for x in range(8):
                piece = None
                for p in pieces:
                    if p.position == C(x, y):
                        piece = p
                        break
                pieceRep = ''
                if piece:
                    side = piece.side
                    color = 'blue' if side == WHITE else
'red'
                    pieceRep = colored(piece.stringRep,
color)
                else:
                    pieceRep = ' '
                stringRep += pieceRep + ' '
            stringRep += '\n'
        return stringRep.rstrip()

    def makeUnicodeStringRep(self, pieces):
        DISPLAY_LOOKUP = {
            "R": '♜',
            "N": '♞',
            "B": '♝',
            "K": '♚',
            "Q": '♛',
```

```
            "P": '♟',
        }

        stringRep = ''
        for y in range(7, -1, -1):
            for x in range(8):
                piece = None
                for p in pieces:
                    if p.position == C(x, y):
                        piece = p
                        break
                on_color = 'on_cyan' if y % 2 == x % 2 else
'on_yellow'
                pieceRep = colored('  ', on_color=on_color)
                if piece:
                    side = piece.side
                    color = 'white' if side == WHITE else
'grey'
                    pieceRep = colored(piece.stringRep + '
', color=color, on_color=on_color)
                stringRep += pieceRep
            stringRep += '\n'
        return stringRep.rstrip()

    def wrapStringRep(self, stringRep):
        sRep = '\n'.join(
            ['%d  %s' % (8-r, s.rstrip())
             for r, s in enumerate(stringRep.split('\n'))]
+
            [' '*21, '   a b c d e f g h']
            ).rstrip()
        return sRep
```

```
    def rankOfPiece(self, piece):
        return str(piece.position[1] + 1)

    def fileOfPiece(self, piece):
        transTable = str.maketrans('01234567', 'abcdefgh')
        return str(piece.position[0]).translate(transTable)

    def getCoordinateNotationOfMove(self, move):
        notation = ""
        notation += self.positionToHumanCoord(move.oldPos)
        notation += self.positionToHumanCoord(move.newPos)

        if move.promotion:
            notation +=
str(move.specialMovePiece.stringRep)

        return notation

    def getCaptureNotation(self, move, short=False):
        notation = ""
        pieceToMove = move.piece
        pieceToTake = move.pieceToCapture

        if type(pieceToMove) is Pawn:
            notation += self.fileOfPiece(pieceToMove)
        else:
            notation += pieceToMove.stringRep
        notation += 'x'
        if short:
            notation += pieceToTake.stringRep
        else:
```

```
            notation +=
self.positionToHumanCoord(move.newPos)

        if move.promotion:
            notation +=
str(move.specialMovePiece.stringRep)

        return notation

    def currentSideRep(self):
        return "White" if self.currentSide else "Black"

    def getAlgebraicNotationOfMove(self, move, short=True):
        notation = ""
        pieceToMove = move.piece
        pieceToTake = move.pieceToCapture

        if move.queensideCastle:
            return "0-0-0"

        if move.kingsideCastle:
            return "0-0"

        if not short or type(pieceToMove) is not Pawn:
            notation += pieceToMove.stringRep

        if pieceToTake is not None:
            if short and type(pieceToMove) is Pawn:
                notation += self.fileOfPiece(pieceToMove)
            notation += 'x'

        notation += self.positionToHumanCoord(move.newPos)
```

```
        if move.promotion:
            notation += "=" +
str(move.specialMovePiece.stringRep)

        return notation

    def getAlgebraicNotationOfMoveWithFile(self, move,
short=True):
        # TODO: Use self.getAlgebraicNotationOfMove instead
of repeating code
        notation = ""
        pieceToMove = self.pieceAtPosition(move.oldPos)
        pieceToTake = self.pieceAtPosition(move.newPos)

        if not short or type(pieceToMove) is not Pawn:
            notation += pieceToMove.stringRep
        notation += self.fileOfPiece(pieceToMove)

        if pieceToTake is not None:
            notation += 'x'

        notation += self.positionToHumanCoord(move.newPos)
        return notation

    def getAlgebraicNotationOfMoveWithRank(self, move,
short=True):
        # TODO: Use self.getAlgebraicNotationOfMove instead
of repeating code
        notation = ""
        pieceToMove = self.pieceAtPosition(move.oldPos)
        pieceToTake = self.pieceAtPosition(move.newPos)
```

```
        if not short or type(pieceToMove) is not Pawn:
            notation += pieceToMove.stringRep

        notation += self.rankOfPiece(pieceToMove)

        if pieceToTake is not None:
            if short and type(pieceToMove) is Pawn:
                notation += self.fileOfPiece(pieceToMove)
            notation += 'x'

        notation += self.positionToHumanCoord(move.newPos)
        return notation

    def getAlgebraicNotationOfMoveWithFileAndRank(self,
move, short=True):
        # TODO: Use self.getAlgebraicNotationOfMove instead
of repeating code
        notation = ""
        pieceToMove = self.pieceAtPosition(move.oldPos)
        pieceToTake = self.pieceAtPosition(move.newPos)

        if not short or type(pieceToMove) is not Pawn:
            notation += pieceToMove.stringRep

        notation += self.fileOfPiece(pieceToMove)
        notation += self.rankOfPiece(pieceToMove)

        if pieceToTake is not None:
            notation += 'x'

        notation += self.positionToHumanCoord(move.newPos)
```

```
        return notation

    def humanCoordToPosition(self, coord):
        transTable = str.maketrans('abcdefgh', '12345678')
        coord = coord.translate(transTable)
        coord = [int(c)-1 for c in coord]
        pos = C(coord[0], coord[1])
        return pos

    def positionToHumanCoord(self, pos):
        transTable = str.maketrans('01234567', 'abcdefgh')
        notation = str(pos[0]).translate(transTable) +
str(pos[1]+1)
        return notation

    def isValidPos(self, pos):
        if 0 <= pos[0] <= 7 and 0 <= pos[1] <= 7:
            return True
        else:
            return False

    def getSideOfMove(self, move):
        return move.piece.side

    def getPositionOfPiece(self, piece):
        for y in range(8):
            for x in range(8):
                if self.boardArray[y][x] is piece:
                    return C(x, 7-y)

    def pieceAtPosition(self, pos):
        for piece in self.pieces:
```

```
            if piece.position == pos:
                return piece

    def movePieceToPosition(self, piece, pos):
        piece.position = pos

    def addPieceToPosition(self, piece, pos):
        piece.position = pos

    def clearPosition(self, pos):
        x, y = self.coordToLocationInArray(pos)
        self.boardArray[x][y] = None

    def coordToLocationInArray(self, pos):
        return (7-pos[1], pos[0])

    def locationInArrayToCoord(self, loc):
        return (loc[1], 7-loc[0])

    def makeMove(self, move):
        self.addMoveToHistory(move)
        if move.kingsideCastle or move.queensideCastle:
            kingToMove = move.piece
            rookToMove = move.specialMovePiece
            self.movePieceToPosition(kingToMove,
move.newPos)
            self.movePieceToPosition(rookToMove,
move.rookMove.newPos)
            kingToMove.movesMade += 1
            rookToMove.movesMade += 1

        elif move.passant:
```

```
            pawnToMove = move.piece
            pawnToTake = move.specialMovePiece
            pawnToMove.position = move.newPos
            self.pieces.remove(pawnToTake)
            pawnToMove.movesMade += 1

        elif move.promotion:
            pieceToTake = move.pieceToCapture
            self.pieces.remove(move.piece)
            if pieceToTake:
                if pieceToTake.side == WHITE:
                    self.points -= pieceToTake.value
                if pieceToTake.side == BLACK:
                    self.points += pieceToTake.value
                self.pieces.remove(pieceToTake)

            self.pieces.append(move.specialMovePiece)
            if move.piece.side == WHITE:
                self.points += move.specialMovePiece.value
- 1
            if move.piece.side == BLACK:
                self.points -= move.specialMovePiece.value
- 1
            move.piece.movesMade += 1

        else:
            pieceToMove = move.piece
            pieceToTake = move.pieceToCapture

            if pieceToTake:
                if pieceToTake.side == WHITE:
                    self.points -= pieceToTake.value
```

```
                if pieceToTake.side == BLACK:
                    self.points += pieceToTake.value
                self.pieces.remove(pieceToTake)

            self.movePieceToPosition(pieceToMove,
move.newPos)
            pieceToMove.movesMade += 1
        self.movesMade += 1
        self.currentSide = not self.currentSide

    def getPointValueOfSide(self, side):
        points = 0
        for piece in self.pieces:
            if piece.side == side:
                points += piece.value
        return points

    def getPointAdvantageOfSide(self, side):
        pointAdvantage = self.getPointValueOfSide(side) - \
            self.getPointValueOfSide(not side)
        return pointAdvantage
        if side == WHITE:
            return self.points
        if side == BLACK:
            return -self.points

    def getAllMovesUnfiltered(self, side,
includeKing=True):
        unfilteredMoves = []
        for piece in self.pieces:
            if piece.side == side:
                if includeKing or piece.stringRep != 'K':
```

```
                    for move in piece.getPossibleMoves():
                        unfilteredMoves.append(move)
        return unfilteredMoves

    def testIfLegalBoard(self, side):
        for move in self.getAllMovesUnfiltered(side):
            pieceToTake = move.pieceToCapture
            if pieceToTake and pieceToTake.stringRep ==
'K':
                return False
        return True

    def moveIsLegal(self, move):
        side = move.piece.side
        self.makeMove(move)
        isLegal = self.testIfLegalBoard(not side)
        self.undoLastMove()
        return isLegal

    # TODO: remove side parameter, unneccesary
    def getAllMovesLegal(self, side):
        unfilteredMoves =
list(self.getAllMovesUnfiltered(side))
        legalMoves = []
        for move in unfilteredMoves:
            if self.moveIsLegal(move):
                legalMoves.append(move)
        return legalMoves
```

8.15 Bishop.py

```
from Piece import Piece
```

```
from Coordinate import Coordinate as C

WHITE = True
BLACK = False


class Bishop (Piece):

    stringRep = 'B'
    value = 3

    def __init__(self, board, side, position, movesMade=0):
        super(Bishop, self).__init__(board, side, position)
        self.movesMade = movesMade

    def getPossibleMoves(self):
        currentPosition = self.position
        directions = [C(1, 1), C(1, -1), C(-1, 1), C(-1,
-1)]
        for direction in directions:
            for move in
self.movesInDirectionFromPos(currentPosition,

direction, self.side):
                yield move
```

8.16 AI.py

```
from Board import Board
from MoveNode import MoveNode
from InputParser import InputParser
import copy
import random
from multiprocessing import Pool


WHITE = True
BLACK = False


class AI:

    depth = 1
    board = None
    side = None
    movesAnalyzed = 0

    def __init__(self, board, side, depth):
        self.board = board
        self.side = side
        self.depth = depth
        self.parser = InputParser(self.board, self.side)

    def getFirstMove(self, side):
        move = list(self.board.getAllMovesLegal(side))[0]
        return move

    def getAllMovesLegalConcurrent(self, side):
        p = Pool(8)
        unfilteredMovesWithBoard = \
```

```
            [(move, copy.deepcopy(self.board))
             for move in
self.board.getAllMovesUnfiltered(side)]
        legalMoves = p.starmap(self.returnMoveIfLegal,
                               unfilteredMovesWithBoard)
        p.close()
        p.join()
        return list(filter(None, legalMoves))

    def minChildrenOfNode(self, node):
        lowestNodes = []
        for child in node.children:
            if not lowestNodes:
                lowestNodes.append(child)
            elif child < lowestNodes[0]:
                lowestNodes = []
                lowestNodes.append(child)
            elif child == lowestNodes[0]:
                lowestNodes.append(child)
        return lowestNodes

    def maxChildrenOfNode(self, node):
        highestNodes = []
        for child in node.children:
            if not highestNodes:
                highestNodes.append(child)
            elif child < highestNodes[0]:
                highestNodes = []
                highestNodes.append(child)
            elif child == highestNodes[0]:
                highestNodes.append(child)
        return highestNodes
```

```
    def getRandomMove(self):
        legalMoves =
list(self.board.getAllMovesLegal(self.side))
        randomMove = random.choice(legalMoves)
        return randomMove

    def generateMoveTree(self):
        moveTree = []
        for move in self.board.getAllMovesLegal(self.side):
            moveTree.append(MoveNode(move, [], None))

        for node in moveTree:
            self.board.makeMove(node.move)
            self.populateNodeChildren(node)
            self.board.undoLastMove()
        return moveTree

    def populateNodeChildren(self, node):
        node.pointAdvantage =
self.board.getPointAdvantageOfSide(self.side)
        node.depth = node.getDepth()
        if node.depth == self.depth:
            return

        side = self.board.currentSide

        legalMoves = self.board.getAllMovesLegal(side)
        if not legalMoves:
            if self.board.isCheckmate():
                node.move.checkmate = True
                return
```

```
            elif self.board.isStalemate():
                node.move.stalemate = True
                node.pointAdvantage = 0
                return
            raise Exception()

        for move in legalMoves:
            self.movesAnalyzed += 1
            node.children.append(MoveNode(move, [], node))
            self.board.makeMove(move)
            self.populateNodeChildren(node.children[-1])
            self.board.undoLastMove()

    def getOptimalPointAdvantageForNode(self, node):
        if node.children:
            for child in node.children:
                child.pointAdvantage = \

self.getOptimalPointAdvantageForNode(child)

            # If the depth is divisible by 2,
            # it's a move for the AI's side, so return max
            if node.children[0].depth % 2 == 1:
                return(max(node.children).pointAdvantage)
            else:
                return(min(node.children).pointAdvantage)
        else:
            return node.pointAdvantage

    def getBestMove(self):
        moveTree = self.generateMoveTree()
        bestMoves = self.bestMovesWithMoveTree(moveTree)
```

```
        randomBestMove = random.choice(bestMoves)
        randomBestMove.notation =
self.parser.notationForMove(randomBestMove)
        return randomBestMove

    def makeBestMove(self):
        self.board.makeMove(self.getBestMove())

    def bestMovesWithMoveTree(self, moveTree):
        bestMoveNodes = []
        for moveNode in moveTree:
            moveNode.pointAdvantage = \

self.getOptimalPointAdvantageForNode(moveNode)
            if not bestMoveNodes:
                bestMoveNodes.append(moveNode)
            elif moveNode > bestMoveNodes[0]:
                bestMoveNodes = []
                bestMoveNodes.append(moveNode)
            elif moveNode == bestMoveNodes[0]:
                bestMoveNodes.append(moveNode)

        return [node.move for node in bestMoveNodes]

    def isValidMove(self, move, side):
        for legalMove in self.board.getAllMovesLegal(side):
            if move == legalMove:
                return True
        return False

    def makeRandomMove(self):
        moveToMake = self.getRandomMove()
```

```
        self.board.makeMove(moveToMake)


if __name__ == "__main__":
    mainBoard = Board()
    ai = AI(mainBoard, True, 3)
    print(mainBoard)
    ai.makeBestMove()
    print(mainBoard)
    print(ai.movesAnalyzed)
    print(mainBoard.movesMade)
```

8.17 cleanedRaspStepper.py

```
# Rasp Pi Set Up
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)

# Declare pin functions on RedBoard
STP1 = 20   # motor moves on this pin rising edge
DIR1 = 21   # DIR pin low = CCW, high CW when open side
motor connector faces in
MS11 = 16
MS12 = 12
EN1 =  25   # Low to Enable mtor, High to disable

STP2 = 19   # motor moves on this pin rising edge
DIR2 = 26   # DIR pin low = CCW, high CW when open side
motor connector faces in
MS21 = 13
MS22 =  6
```

```
EN2 =   5  # Low to Enable mtor, High to disable

# Setup pins
GPIO.setup(STP1, GPIO.OUT)
GPIO.setup(DIR1, GPIO.OUT)
GPIO.setup(MS11, GPIO.OUT)
GPIO.setup(MS12, GPIO.OUT)
GPIO.setup(EN1, GPIO.OUT)

GPIO.setup(STP2, GPIO.OUT)
GPIO.setup(DIR2, GPIO.OUT)
GPIO.setup(MS21, GPIO.OUT)
GPIO.setup(MS22, GPIO.OUT)
GPIO.setup(EN2, GPIO.OUT)

# Declare Constants
stepPeriod = .0005 # time between of motor steps in sec
stepsPerGrid = 200 # number of steps to move one grid
square
                   # stepsPerGrid = squareWidth in cm * 40

# Function to move Gantry horizontally; motors same
direction
def horizMove(dir, dist): # dir = -1 means left, +1 means
right, dist in number of grid squares
    try:
        # Set directions of both motors to match desired
motion
        GPIO.output(DIR1, GPIO.LOW) if dir<0 else
GPIO.output(DIR1, GPIO.HIGH)
        GPIO.output(DIR2, GPIO.LOW) if dir<0 else
GPIO.output(DIR2, GPIO.HIGH)
```

```
        numSteps = stepsPerGrid*dist

        for x in range (0,numSteps):
            GPIO.output(STP1,GPIO.HIGH) #Trigger one step
forward
            GPIO.output(STP2,GPIO.HIGH)
            time.sleep(stepPeriod/2)       # /2 b/c only
move on rising edge
            GPIO.output(STP1,GPIO.LOW)  #Pull step pin low
so it can be triggered again
            GPIO.output(STP2,GPIO.LOW)
            time.sleep(stepPeriod/2)

        print("Returning to Main")
        time.sleep(1)
        main()

    except KeyboardInterrupt:                # If CTRL+C is
pressed, exit cleanly:
        GPIO.cleanup()                       # cleanup all
GPIO
    finally:                                 # force to
always clean up on way out
        GPIO.cleanup()                       # cleanup all
GPIO


# Function to move Gantry Vertically; motors oppposing
directions
def vertMove(dir, dist): # dir = -1 means down, +1 means
up, dist in number of grid squares
    try:
```

```
        # Set directions of both motors to match desired
motion
        GPIO.output(DIR1, GPIO.LOW) if dir<0 else
GPIO.output(DIR1, GPIO.HIGH)
        GPIO.output(DIR2, GPIO.HIGH) if dir<0 else
GPIO.output(DIR2, GPIO.LOW)
        numSteps = stepsPerGrid*dist

        for x in range (0,numSteps):
            GPIO.output(STP1,GPIO.HIGH)     #Trigger one
step forward
            GPIO.output(STP2,GPIO.HIGH)
            time.sleep(stepPeriod/2)        # /2 b/c only
move on rising edge
            GPIO.output(STP1,GPIO.LOW)      # Pull step pin
low so it can be triggered again
            GPIO.output(STP2,GPIO.LOW)
            time.sleep(stepPeriod/2)

        print("Returning to Main")
        time.sleep(1)
        main()

    except KeyboardInterrupt:               # If CTRL+C is
pressed, exit cleanly:
        GPIO.cleanup()                      # cleanup all
GPIO
    finally:                                # force to
always clean up on way out
        GPIO.cleanup()                      # cleanup all
GPIO
```

```
# main loop, get input on axis, direction, and distance
def main():
    axis = input("Choose Axis: -1 for vertical, +1 for
horizontal: ")
    if axis == 1:
        dir = input("Enter direction: -1 for Left, +1 for
Right: ")
        dist = input("Enter distance: number of grid
squares: ")
        horizMove(dir,dist)
    elif axis == -1:
        dir = input("Enter direction: -1 for Down, +1 for
Up: ")
        dist = input("Enter distance: number of grid
squares: ")
        vertMove(dir,dist)
    else:
        print("Invalid input, please try again")
        main()

main()  # ener main loop
```

Complete hardware schematics

Complete Software listings

Relevant parts or component data sheets (do NOT include the data sheets for the microcontroller or other huge files but give good links to where they may be found.)