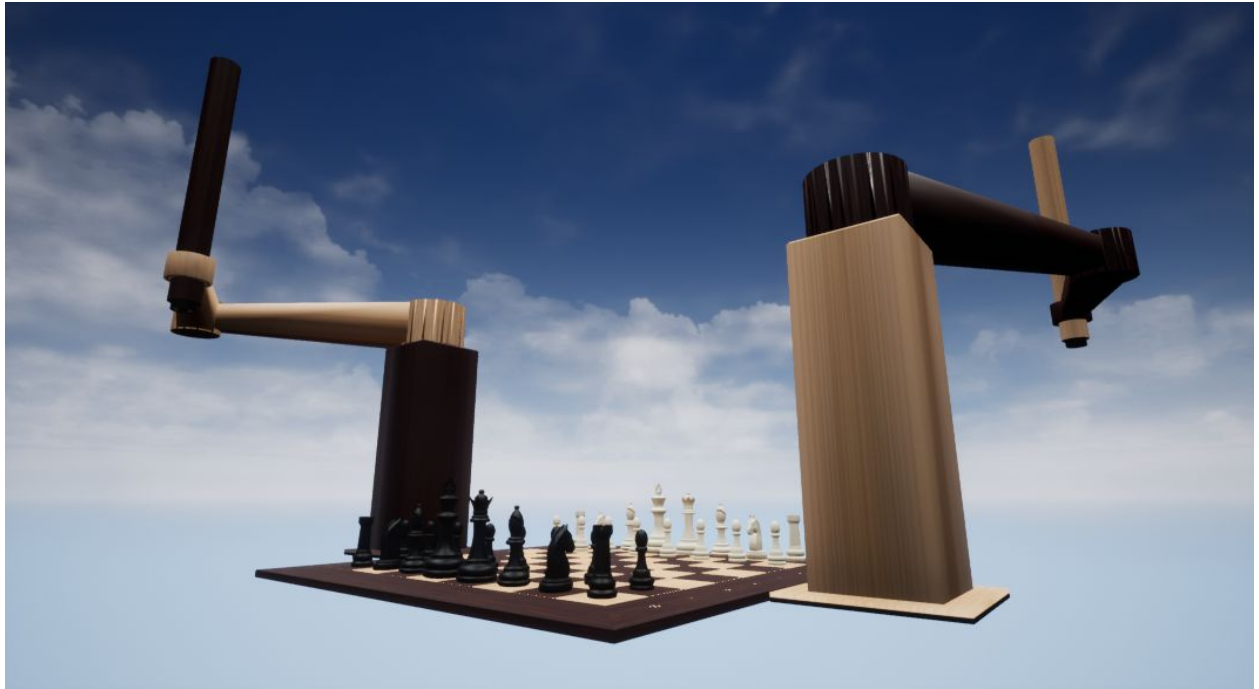
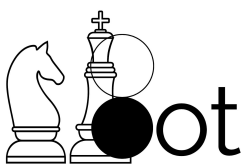


# ChessBot: A Robotics Controller and Simulator Capable of Playing Chess



Brett Levenson, Luke Cohen, and Matthew French  
Autonomous Robotics (EECS 467)  
University of Michigan, Ann Arbor  
March 27th, 2020

<https://www.youtube.com/watch?v=A8CCiv3MewY&t=3s>



## Table of Contents

[Abstract](#)

[I. INTRODUCTION](#)

[II.CHESSBOARD](#)

[III. COMMUNICATION PROTOCOL](#)

[IV. GAME LOOP](#)

[V. SIMULATOR](#)

[VI.IMAGE PROCESSING](#)

[VII. CHESS SOLVER](#)

[Universal Chess Interface](#)

[VIII. MOTION PLANNER](#)

[IX. WORKING REMOTELY](#)

[X. TIMELINE](#)

[XI. CONCLUSION](#)

[XII. CERTIFICATION AND PEER EVALUATION](#)

**Abstract**—This paper outlines our MDE project for Autonomous Robotics (EECS 467). Our goal for this project was to develop a robust software stack that can control a robot to play chess. Though we were forced to develop a virtual solution, we achieved this task by capturing an image from a virtual camera above a simulated board, using the image to determine an optimal chess move, and controlling simulated robotic arms to perform moves. We leveraged the simulation capabilities of the Unreal video game engine, and wrote other custom Python code to control the simulation and camera. We also built in the Stockfish chess solver to quickly determine an optimal move. After a semester of development, we resulted in a product capable of playing a game of chess.

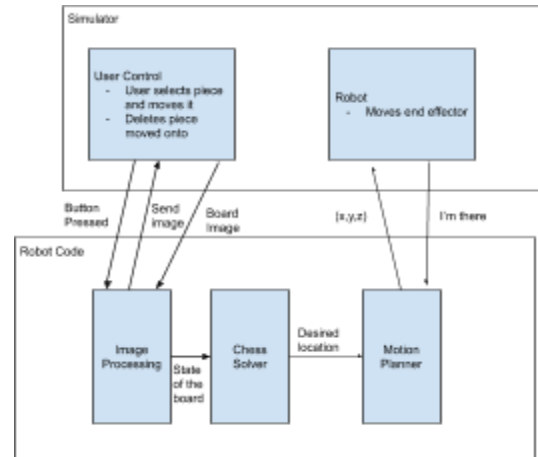
## I. INTRODUCTION

For our MDE project for EECS 467 - Autonomous Robotics, we were tasked to come up with an open-ended project relating to the concepts covered in the course. Our team decided to build a robot capable of playing a human in chess. Ideally, the robot would sense the board, and make moves using a robotic arm. Initially we planned to implement both a hardware and software solution to playing the game. However, because of the Covid-19 outbreak, we pivoted to implementing a simulated version of the robot, which allowed us the opportunity to not only implement the robotic controller, but also a simulator for the robot as well. Also, because we had no physical system, we added the capability for two robotic arms to play against one another.

Therefore, the code structure was split up into two main domains—one being the simulator and one being the robot controller. Our main vision for this design was to allow us to easily add in a real robot in place of the simulator. This can be seen in Figure 1 shown right where we have the simulator code communicating through sockets to the robot controller. This allows our simulator to be replaceable with real hardware with less modification.

From a high level, the process is controlled by a main game loop. This game loop is the conductor for all parts of the system and is responsible for grabbing an image from the simulator and passing it to the image processing pipeline. The imaging pipeline uses a computer vision algorithm to determine the current state of the board pieces. From there, we use a chess solver to determine the optimal move to be made. Once we know what move to make, we use a motion

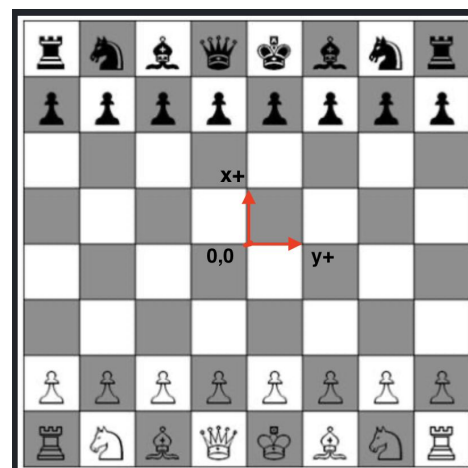
planner to generate a path for the robotic arm to follow in order to perform the requested move without colliding with other pieces. Finally, the simulator uses inverse kinematics to transform the robot's path from world coordinates into motor angles for the simulated robot.



**Figure 1:** The high level overview of all the software components and how they were separated out into the robot code and the simulator code.

## II. CHESSBOARD

The chessboard we used in simulation was designed to fit the regulations for a real board. We centered the world frame in the center of the chessboard. One interesting thing to note about the coordinate system of our simulator is that it follows the left hand rule, so our positive z-axis is out of the page. This can be shown in the figure below.



**Figure 2:** Chessboard with the left-handed world coordinate frame centered in the middle of the board.

Our real arm would have had an electromagnet to hold a chess piece with metal inside. We kept this scheme when building our simulator. Therefore, in addition to the sizing of the chessboard, we also store a map of each piece's height, shown in Table 1 below, so that we know to what level to lower the arm in order to make good contact with the piece. This also allowed us to determine the lowest point at which we could move a chess piece with the arm to avoid collisions with even the tallest piece on the board.

**Table 1:** Sizing of each chess piece.

|        | Required Height (cm) | Height (cm) | Base (cm) | Base/Height |
|--------|----------------------|-------------|-----------|-------------|
| King   | 9.5                  | 9.6         | 4.1       | 43.0        |
| Queen  | 8.5                  | 8.4         | 3.8       | 45.0        |
| Bishop | 7.0                  | 7.0         | 3.0       | 43.0        |
| Knight | 6.0                  | 5.9         | 2.7       | 46.0        |
| Rook   | 5.5                  | 5.5         | 2.4       | 44.0        |
| Pawn   | 5.0                  | 5.0         | 2.3       | 46.0        |

We then converted this table into a map in our code, we multiplied each piece height by our conversion factor (50 Unreal Units/cm) to convert them to sizes in our world frame. This allowed us to easily change out the part sizes without much code refactoring.

Additionally, the size of the chess board and piece requirements are described in the table below. We used these to ensure the simulated chess board was in compliance with the real world chess board.

**Table 2:** Table of requirements for the sizing of the chessboard and pieces

|                            | Required Value | Measured Value |
|----------------------------|----------------|----------------|
| Square Size                | 5 - 6 cm       | 5.2 cm         |
| Square Size                | 2 - 2.5 in     | 2.04724 in     |
| Square Size > 2x Pawn Base | >4.6 cm        | 5.2 cm         |
| King Height                | 3.375 - 4.5 in | 3.74016 in     |
| King Base/King Height      | 40 - 50 %      | 43 %           |
| King Base/Square Size      | 75 - 80 %      | 79 %           |

### III. COMMUNICATION PROTOCOL

In order to have all the different processes of this system communicate with one another we used sockets to send data. This also has the added benefit of drawing clear lines of abstraction between the different parts of the code which can be easily substituted out. In terms of the ports, we relied on one port to handle all the communication with the Unreal Engine simulator and a separate port for the image processing and main game loop process.

#### Messaging Information

- All IPs = 127.0.0.1 (localhost)
- Port 8555 = UE4 Simulation
- Port 8556 = Image Processor/conductor

**Figure 3:** Ports used to communicate with the simulator from the robot code.

All of our communication is done over UDP sockets because we didn't want to deal with the overhead of the TCP handshakes. In a real robot, we would have so many messages being sent that if we accidentally dropped some it wouldn't be that big of an issue. Although we only send one image as it is requested by the main game loop, in a real system we would be streaming the video feed which is the common use case of UDP.

We described a set of messages to send between the robot code and the simulation. If we were to run this code on an actual robot, we would only need to handle these messages. All the supported messages are shown below, so I just want to highlight a few of them. We support a simple 'r', to ask the simulator to see if it is ready. We are able to teleport pieces to simulate a human moving them. We can command the arm to move to given coordinates and enable or disable the electromagnet. When requesting an image, the simulator responds to it by sending a 250x250 array of uint8s for the intensity of every pixel.

**Table 3:** Table of available commands to control the simulator.

| Command                 | Format                | Parameters  |
|-------------------------|-----------------------|---|
| Echo "ready"            | "r"                   | n/a   |
| Request image           | "i"                   | n/a   |
| Teleport piece to space | "p,<piece>,<space>"   | piece = "blackbishop1"<br>space = "e5"            |
| Move arm to location    | "a,<arm>,<x>,<y>,<z>" | arm = "b" or "w"<br>x, y, z = "1750.05"           |
| Change magnet state     | "m,<arm>,<state>"     | arm = "b" or "w"<br>state = "0" (off) or "1" (on) |

For any command that controls a robot, we have to specify which robot arm we are controlling, which allows this to be expanded out to multiple possible robots in the simulator.

In order for the python code parts to communicate, this includes the image processor, the chess solver wrapper, and the motion planner, we use simple function calls to hook up all the parts.

#### IV. GAME LOOP

Our game loop is a high level main function that uses the processes described below to execute a game of chess. We have allowed for both human vs robot interaction as well as robot vs robot. The main functionality of the loop is shown in the figure below.

```
def game_loop_human():
    board, board_sum = ip.init_board_state()

    while(not is_game_over()): #somehow need to define when the game is over
        move = user_input()
        cs.send_move_to_simulator(board, move)
        #requests image -> receives image -> processes image -> calls brett's stuff
        board_sum, board = ip.process_image(board, board_sum)
        # wait_for_ready()

def game_loop_robot():
    board, board_sum = ip.init_board_state()

    while(not is_game_over()): #somehow need to define when the game is over
        board_sum, board = ip.process_image(board, board_sum, 'w')
        # wait_for_ready()
        board_sum, board = ip.process_image(board, board_sum, 'b')
        # wait_for_ready()
```

**Figure 4:** Game loop functionality differs for human versus robot and robot versus robot modes.

These functions both call the process\_image() function which in turn, requests an image from the simulator, processes that image in comparison to the previous image it received, and updates the board state based on the difference. This updated board state is passed into a parameter of the chess solver that uses the stockfish binary to find the move that

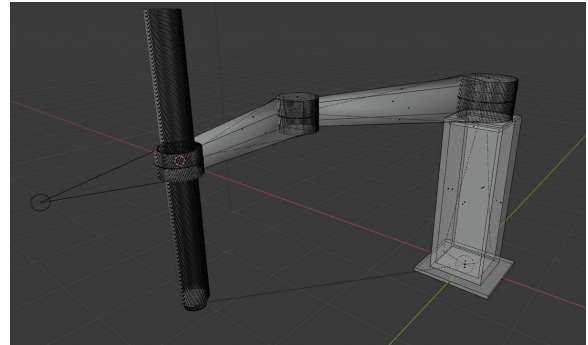
the robot should make. With this move, our code plans a movement for our robotic arm and transmits commands to our simulator. Using these commands, the simulator moves our robotic arm on a collision free path to pick up, move, and place the correct chess piece in its new position and loops back to the start of the game\_loop function.

Below, we will discuss in depth each of the processes highlighted above.

#### V. SIMULATOR

We chose the Unreal Engine to build our simulator for a number of reasons. Firstly, the simulator is responsible for providing a visual depiction of our controller's effect. That is, it shows the state of the chessboard and the motion of the chess pieces and robotic arms around that board. The cover page shows how this simulator looks. Because the Unreal Engine is a video game engine, it has much of the required support for manipulating meshes and holding a game state. Secondly, the engine is open source, which allowed us to take advantage of a few key processes to enable simple image capture. Thirdly, Matthew has worked with the engine in the past.

The robotic arm meshes shown in the simulator are custom built. We used Blender software to generate a mesh and armature which allows the mesh to move. A SCARRA arm mesh is shown in the figure below.



**Figure 5:** Custom SCARRA arm mesh and armature modeled in blender software.

We imported these meshes and placed them along with the chessboard and pieces into the engine. We then arranged the objects in the arrangement of the starting chess state.

The two SCARRA arms used in the simulator have to be controlled with inverse kinematics. The motion planner produces a path of world coordinates which must be converted into the angle of motion for each link of the arm. We leveraged the inverse kinematics solver provided in the unreal engine to create custom dynamic animations of the SCARRA arm models.

The simulator is driven by messaging. In particular, like a physical robotic system, the simulator is purely responsive to the commands of the controller. Therefore, it was essential to implement socket messaging to control the simulator. The simulator uses that messaging pathway to listen for commands from the controller portion of the code. A table of the available commands are shown in Table 3. The simulator receives a command and handles it by performing an appropriate action.

One command requires special functionality from the simulator—the “i” command. The multithreaded, open-source implementation allowed us to dive deep into the rendering core of the engine. We added functionality that is executed every time the engine renders a video frame which saves that current frame into a variable. Then using an API that allows us to queue commands on our computer's graphics card, we convert the frame into a grayscale image to be stored. This function is shown in the figure below. Whenever the “i” command is sent, the current camera is set to be the overhead camera, the frame is generated and converted, and the resulting grayscale image is sent to the image processor.

```
void FUDPSockets::SaveFrame(const FTexture2DRHIFRef& BackBuffer)
{
    FRHICommandListImmediate& RHICmdList = FRHICommandListExecutor::GetImmediateCommandList();
    FIntRect Rect(0, 0, BackBuffer->GetSizeX(), BackBuffer->GetSizeY());
    {
        FScopeLock Lock(&mutex);
        frame.Reset();
        RHICmdList.ReadSurfaceData(BackBuffer, Rect, frame, FReadSurfaceDataFlags());
    }
}
```

**Figure 6:** Command to convert every rendered frame into a grayscale image using our computer's GPU.

## VI. IMAGE PROCESSING

Before the COVID-19 outbreak, image processing was the main way for our code to sense what was how the human player was making moves. Our initial plan was to have an overhead camera above the board to capture different images after both the player and robot made a move and then use different image processing techniques to discern the new state of the

board. When we switched over to simulation, we decided that we would keep this aspect of our project both because it gave us a somewhat simple way to update our game state between our simulator and our code as well as gave us adaptability in case we ever decided to implement this project in real life.

Our initial image processing pipeline worked by taking an “imperfect” photo of the board and using corner detection techniques in open cv to find an affine transformation from the skewed photo to a squared image showing only the board and pieces.



**Figure 7:** Our original image processing plan showing the three stages the images will go through. Starting off as the raw image, finding the grid, rotating the board to fill the screen, and then averaging the color in the squares to find pieces.

We pivoted on that notion due to our ability to receive “perfect” images (serialized) from our simulator over a UDP connection. Our new image processing pipeline works by using one image from before the move was made and one image after the move was made to find which piece moved during that turn. We found the correct move by subtracting the two images from each other, and discretizing the resulting image by summing and normalizing each checkerboard shown resulting in an 8x8 array of differing sums.

Using the numpy argmax function to find the largest and second largest difference, we were able to find out which pieces were moved, and combine that with our internal state of the board, to determine which piece was moved, and to which square. This required trivial logic based solely on the two squares that were affected during that move. With the updated state of the board, we can send that to the chess solver to determine the next steps.

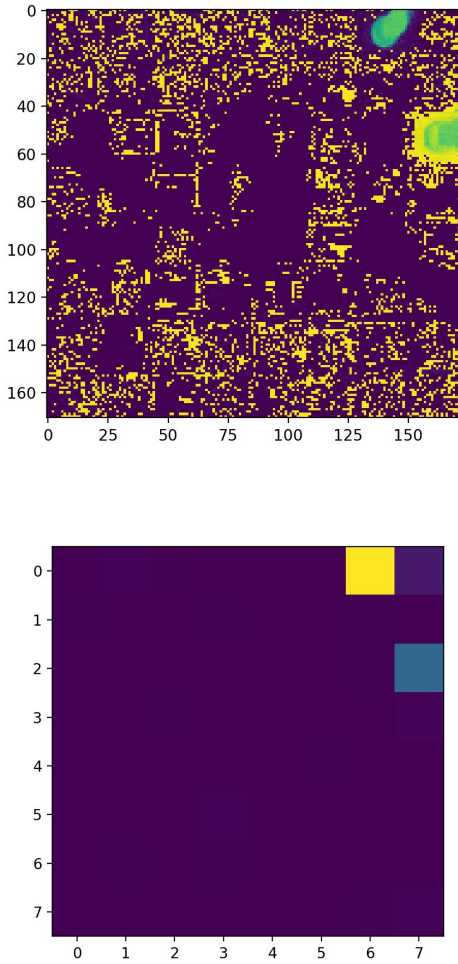


## VII. CHESS SOLVER

Chess is a problem that many people have already tried to solve, and was one of the first challenges tackled by computers. Because of this, we decided to leverage the preexisting work done by other people in creating AIs that can play chess. Additionally, for the AI to be sufficient for this task it would become a major time sink and detract from more pressing parts of this project. It is for these reasons that we have decided to use an open-source chess solver as the brains behind our robotic player.

We looked at many different open source chess programs and settled on the Stockfish project because of its runtime performance and superior move planning. This easily slid into our code's pipeline, receiving the new state of the board as a matrix and the player's turn, and then it output a move to be made and whether or not the end position is occupied, requiring it to be removed before the piece can be translated.

In order to interface with the Stockfish binary, we created a wrapper class which spun up Stockfish as its own subprocess, and could then communicate with it to inform it of the current chessboard state and then to grab the resulting move. This class takes as input the current state of the board from the image processing code as an 8x8 string matrix, where each cell can be either blank, or have the name of the chess piece, along with its color and count. Below you can see an example of the initial board state being passed into the chess solver. From this state, this class then outputs the optimal move to be made on that board given a current player.



**Figure 8:** Raw and discretized image showing a knight moving from g8 to h6

```
#move or capture
if(diff_val1 > MOVE_THRESH):
    #move from index 2 to 1
    board_state1 = board_state[move_index1[0]][move_index1[1]]
    board_state2 = board_state[move_index2[0]][move_index2[1]]

    if((board_state1 == ' ') & (board_state2 != ' ')):
        board_state[move_index1[0]][move_index1[1]] = board_state2
        board_state[move_index2[0]][move_index2[1]] = ' '

    #move from index 1 to 2
    elif((board_state1 != ' ') & (board_state2 == ' ')):
        board_state[move_index2[0]][move_index2[1]] = board_state1
        board_state[move_index1[0]][move_index1[1]] = ' '

    #capture
    elif((board_state1 != ' ') & (board_state2 != ' ')):
        if(board_state1[0] == player):
            board_state[move_index2[0]][move_index2[1]] = board_state1
            board_state[move_index1[0]][move_index1[1]] = ' '
        elif(board_state2[0] == player):
            board_state[move_index1[0]][move_index1[1]] = board_state2
            board_state[move_index2[0]][move_index2[1]] = ' '
    else:
        move = None
        board_state = board_state
```

**Figure 9:** Board update logic based on discretized image sum

```
12 test_input = [['Br1', 'Bn1', 'Bb1', 'Bq', 'Bk', 'Bb2', 'Bn2', 'Br2'],
13 ['Bp1', 'Bp2', 'Bp3', 'Bp4', 'Bp5', 'Bp6', 'Bp7', 'Bp8'],
14 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
15 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
16 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
17 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
18 ['Wp1', 'Wp2', 'Wp3', 'Wp4', 'Wp5', 'Wp6', 'Wp7', 'Wp8'],
19 ['Wr1', 'Wn1', 'Wb1', 'Wq', 'Wk', 'Wb2', 'Wn2', 'Wr2']]
```

**Figure 10:** Matrix of the initial chess board state that would be passed as input into the chess solver.

### Universal Chess Interface

The Stockfish program works based off of the Universal Chess Interface (UCI), which is an open communication protocol used to enable chess engines to communicate with a user interface. This means that in order to request a move from the chess engine, we first needed to convert our request into a UCI

formatted string. More specifically there were three main commands we relied on: position, go, and quit.

The first command, “position”, is used to tell the engine the state of the board. This is important because the UCI protocol is stateless, which means everytime we want to query for a move, we need to remind the chess engine of the entire state of the board. In order to describe the state of the board, we use Forsyth–Edwards Notation (FEN) which is a standard notation for writing a particular board position of a chess game. This is a simple system where each piece on the board is represented by an ASCII character, and the case of it tells whether or not it’s a white or black piece. Gaps between pieces become a number indicating how many empty cells are between. In the figure below is the code that is used to generate this string from a given board.

```

123 # Convert board to Forsyth–Edwards Notation (FEN) string
124 def generate_fen(board):
125     output = ''
126
127     curr_count = 0
128     for row in board:
129         for item in row:
130             if item == '' or item == ' ':
131                 curr_count += 1
132             else:
133                 if curr_count > 0:
134                     output += str(curr_count)
135                     curr_count = 0
136                 output += convert_piece(item)
137
138             if curr_count > 0:
139                 output += str(curr_count)
140                 curr_count = 0
141
142             output += '/'
143
144     return output[:-1]
145

```

**Figure 11:** Code that generates the FEN string given a specific board configuration

Applying the code to the initial board state, we can see the corresponding FEN string below. Additionally, we have two other cases of initial board states and their resulting FEN string to highlight the many possible cases that this format can support.

```

12 test_input = [['Br1', 'Bn1', 'Bb1', 'Bq', 'Bk', 'Bb2', 'Bn2', 'Br2'],
13               ['Bp1', 'Bp2', 'Bp3', 'Bp4', 'Bp5', 'Bp6', 'Bp7', 'Bp8'],
14               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
15               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
16               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
17               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
18               ['Wp1', 'Wp2', 'Wp3', 'Wp4', 'Wp5', 'Wp6', 'Wp7', 'Wp8'],
19               ['Wr1', 'Wn1', 'Wb1', 'Wq', 'Wk', 'Wb2', 'Wn2', 'Wr2'],
20               # FEN: rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/8/NBQKBNr
21
22 test_input2 = [['Wn2', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
23               ['Wp1', ' ', 'Bp5', 'Bk', ' ', 'Bb8'],
24               [' ', ' ', 'Bp2', ' ', ' ', 'Bp7', ' ', ' '],
25               ['Br1', ' ', ' ', ' ', 'Bp6', ' ', ' ', ' '],
26               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
27               [' ', ' ', 'Bb1', ' ', ' ', 'Wb1'],
28               [' ', ' ', 'Wp5', ' ', 'Wk', 'Wp8'],
29               [' ', 'Wr1', ' ', ' ', ' ', ' ', ' ', ' '],
30               # FEN: N7/P3pkip/3p2p1/r4p2/8/4b2b/4P1KP/1R6
31
32 test_input3 = [['Wk', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
33               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
34               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
35               [' ', ' ', ' ', ' ', 'Wp', ' ', 'Bk', ' '],
36               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
37               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
38               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
39               [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
40               # FEN: K7/R4R/5Q1N/R8/1P

```

**Figure 12:** Multiple initial board states that could be passed in and their corresponding FEN string which we have to generate.

So putting the parts of this command together, we end up getting “position fen [fen string]”, which is then sent to the chess engine. After that, we send the “go” command which tells the engine to compute the best move for the current player. Finally we send the “quit” message, which terminates the Stockfish process. Now that the process has ended, we scrape through the returned output to find the optimal move to make.

The output we get from Stockfish includes a lot of superfluous information we need to filter out because we are not interested in what moves it considered, but rather just the final move found.

An example of the output of Stockfish can be seen below.

```

Stockfish 11.64 by T. Rasmussen, M. Costalba, J. Kiiski, G. Linscott
position fen rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/8/NBQKBNr
go
info depth 1 seldepth 1 multipv 1 score cp 154 nodes 20 nps 10000 tbhits 0 time 2 pv e7e5
info depth 2 seldepth 2 multipv 1 score cp 135 nodes 55 nps 27500 tbhits 0 time 2 pv e7e5 bxb3
info depth 3 seldepth 3 multipv 1 score cp 184 nodes 162 nps 81000 tbhits 0 time 2 pv d7d5 c2c3 e7e5
info depth 4 seldepth 4 multipv 1 score cp 184 nodes 625 nps 312500 tbhits 0 time 2 pv h7h5 c2c3 e7e5 e2e3
info depth 5 seldepth 5 multipv 1 score cp 151 nodes 791 nps 395500 tbhits 0 time 3 pv d7d5 c2c3 e7e5
info depth 6 seldepth 6 multipv 1 score cp 42 nodes 4167 nps 694500 tbhits 0 time 6 pv g7g6 c2c3 g8f6 e2e3 h7h5 d2d4
info depth 7 seldepth 7 multipv 1 score cp 184 nodes 6485 nps 810625 tbhits 0 time 8 pv g8f6 e2e3 c7c6
info depth 8 seldepth 8 multipv 1 score cp 35 nodes 11558 nps 945500 tbhits 0 time 12 pv g8f6 e2e3 b8c5 d2d4 d7d5 g1f3 c8g4 b1c3
bestmove g8f6 ponder e2e3

```

**Figure 13:** Output of Stockfish generating the move from the initial board state for a white player.

As we can see in the provided output, we only care about the final line where it tells us the “bestmove”. We therefore, scan the output for just this final move, and use that as the move to pass on to the motion planner. Something to spotlight is how the move is formatted in long algebraic notation. This means the string is two squares concatenated together to define the move to be made, where the first two characters are the starting square, and the second two characters are the ending square. It is important to note that it doesn’t give any more information, which means our code needs to determine which piece is there and if the final square has a piece in it that needs to be discarded.

Once we have read in the move to be made by Stockfish, we extract all the information about the move from the stored board state. This includes which piece is being moved, and if the new location has a collision. This information is then bundled up and sent to the motion planner to carry out the move.

## VIII. MOTION PLANNER

After the Chess Solver determines the next optimal move for ChessBot to make, a path will need to be planned to move the piece from its current position to the target position. In particular, ChessBot must



move the piece in a way such that the rest of the game board is not disrupted.

We implemented this as a seven-point path. At the start of its turn, ChessBot's arm is in a resting state away from the player and off the board (point 0). This position ensures that the player can perform their move and that the camera can clearly see the entire game board. From this position, ChessBot moves its arm to the location above the desired piece (point 1). It then descends until the end effector is touching the top of the desired piece (point 2). We determine this height using a height lookup table that holds every piece's height. After the electromagnet is actuated, the arm ascends to a height above the board which is greater than the maximum piece height (point 3). The arm then performs the optimal move (point 4). When the arm reaches the final location, it descends down to the piece's height (point 5) and the electromagnet releases the piece. Finally, the arm ascends (point 6) and returns to the resting position (point 7), away from the board.

If ChessBot's move requires that the player's piece be removed from a spot before ChessBot moves its own piece to that same location, i.e. Chessbot "takes" the player's piece, before point 1, ChessBot moves the taken piece to a designated area off the game board. From here, it then continues to point 1.

Another edge case we cover is castling where we move the king and rook over two spaces, flipping their spaces. This requires an additional move after picking up and dropping the king to the new cell. We then have to pick up the rook, and move it over to the space between the initial and final position of the king. We accomplish this by checking for the two moves that are castle moves, and add an additional instruction to those moves.

The first step of the motion planner is to convert the chess board locations from the board notation, to world coordinates. We used the chessboard description above to know how to convert between coordinate systems.

We know that each square cell is 5.2 cm, which allows us to locate every board location. Additionally, in order to convert from real world units to simulator units, we have a conversion of 5000 uu = 1m. This means that each square is 260uu. In the code below you can see the equations used to convert from chess board location to a X,Y coordinate in space.

```
123 # convert chess location (A5) to X,Y coordinate
124 def location_to_coordinate(location):
125     if location == 'DISCARD':
126         return DISCARD_LOCATION
127
128     row = 8 - int(location[1])
129     col = ord(location[0]) - 97
130
131     return [row * -SQUARE_LEN_UU + BOARD_OFFSET, col * SQUARE_LEN_UU - BOARD_OFFSET]
132
```

**Figure 14:** Function to convert a board cell to a coordinate in space.

In order to convert from the board space to physical space, we first take the grid cell and move it to a range of [0,7] in both dimensions. We turned everything into a constant to make the code easy to change to different board sizes. The two steps are to shift over the origin to the center of the board, and then shift the piece based on which grid cell it falls into.

In order to construct the motion planner, we wrote our own system to stack function calls together to build up the functionality of the robot. The overarching idea is to store robot movements in a list of "moves", with each function call just appending to this global list of moves. We then can pass all these moves to the simulator to be executed as one unit. This can be seen in the code below.

```
51 commands = [] # list of commands that will be sent to the simulator
52
53 if endPiece != '':
54     # piece in location we need to remove first
55     commands += discard_piece(endLoc, endPiece)
56
57 commands += pickup_piece(startLoc, startPiece)
58 commands += drop_piece(endLoc, startPiece)
59 commands += move_out_of_way(robot)
60
61 send_commands(robot, commands)
```

**Figure 15:** Architecture to allow multiple robot actions to be chained together and have the result sent to the simulator.

Now that we have a location in the world frame for both the starting location and the ending cell, we then need to consult our internal board map to see if we need to remove a piece at the ending cell. If we are doing an attack, we then need to discard the piece. To do this, we call the `discard_piece` function which returns a list of moves to pick up and drop off the piece in the designated zone.

```
112 # moves captured piece off the board
113 def discard_piece(location, piece_name):
114     moves = []
115     # pickup piece
116     moves += pickup_piece(location, piece_name)
117
118     # drop piece at discard zone
119     moves += drop_piece('DISCARD', piece_name)
120
121     return moves
122
```

**Figure 16:** Function to return the moves to discard a piece.

Once the piece is out of the way, we can start point 1 as described above, picking up the piece. The first

step is to find the location in 3D space. We then drop above it, grab it, and raise it to the lowest height that will not collide with another piece. This step sets the robot up for the next phase.

```
62 # Returns command to move piece at location (a3) with piece_name (pawn)
63 # to the lowest height needed to avoid hitting pieces below
64 def pickup_piece(location, piece_name):
65     moves = []
66     # go to above the piece
67     piece_cords = location_to_coordinate(location)
68     moves.append("a,%s,%d,%d,%d" % (piece_cords[0], piece_cords[1], STANDARD_HEIGHT))
69     # drop to appropriate arm height
70     moves.append("a,%s,%s,%d,%d" % (piece_cords[0], piece_cords[1], PIECE_HEIGHT_MAP[piece_name]))
71     # activate magnet
72     moves.append("m,%s,1")
73     # lift to standard lifting height
74     moves.append("a,%s,%s,%d,%d" % (piece_cords[0], piece_cords[1], STANDARD_HEIGHT))
75     return moves
```

**Figure 17:** Function to pick up a piece and hover above the square.

The second phase is to drop the piece. This is accomplished by the drop\_piece function, which does the exact opposite as the pickup\_piece function. It moves to above the drop location, lowers to the correct height, disengages the electromagnet, and then raises to the lowest height needed to not hit any other pieces.

```
62 # Returns commands to move piece to location
63 # Assumes location is already empty
64 def drop_piece(location, piece_name):
65     moves = []
66     # go to above the empty square
67     piece_cords = location_to_coordinate(location)
68     moves.append("a,%s,%s,%d,%d" % (piece_cords[0], piece_cords[1], STANDARD_HEIGHT))
69     # drop to appropriate arm height
70     moves.append("a,%s,%s,%d,%d" % (piece_cords[0], piece_cords[1], PIECE_HEIGHT_MAP[piece_name]))
71     # deactivate magnet
72     moves.append("m,%s,0")
73     # lift to height right above tallest piece
74     moves.append("a,%s,%s,%d,%d" % (piece_cords[0], piece_cords[1], TALLEST_HEIGHT))
75     return moves
```

**Figure 18:** Function to drop a piece on the board and then move the arm to float above it.

Now that we have finished the move, we need to get the robot out of the way of the board. We accomplish this by calling the move\_out\_of\_way function, which has the appropriate robot arm be sent to its resting position outside of the space above the board. This simply drives the arm to be in a location so that it isn't blocking another robot, the human player, or the Camera.

```
102 # moves the arm out of the way
103 def move_out_of_way(robot):
104     moves = []
105     location = BLACK_DISCARD_LOCATION if robot == 'b' else WHITE_DISCARD_LOCATION
106     moves.append("a,%s,%s,%d,%d" % (location[0], location[1], STANDARD_HEIGHT))
107     return moves
```

**Figure 19:** Function to move the robot arm out of the way of the board airspace.

Once all these functions have finished running, we have a large list of commands ready to be sent to the robot or simulator to be carried out. This is handled by the send\_commands function, which also makes sure the correct robot is being actuated based on which player is making the move. It sends one move

at a time, and waits for an acknowledgement that it has finished the action from the server before it sends the next move.

## IX. WORKING REMOTELY

From the beginning of the project, we made sure to support working remotely on this project. To accomplish this we made sure to have written documentation for all parts, that way we would always be on the same page for this project. Additionally, we set up recurring FaceTime calls to make sure we were always on the same page. For more immediate communication we relied on text messaging and Slack to send images and ask each other implementation details.

The way we structured the project allowed us to easily parallelize our work because each person could be assigned a module, and as long as they followed the agreed upon interfaces, it could easily slide in with the rest of the code.

## X. TIMELINE

Throughout this project we followed timeline shown below to make sure that we reached all of our deliverables.

| April 6                             | April 13  | April 20  | April 27  |
|-------------------------------------|---|---|---|
| Project Update 3                    | Project Update 4  | Project Update 5  | Final Deliverable                                   |
| Finish and tune simple CV algorithm | Have Chess API giving out moves<br>Arm controlled by controller | Implement full game loop to play entire game of chess<br>Complete messaging | Implement collision detection<br>Add robot v. robot |

**Figure 20:** Proposed timeline

## XI. CONCLUSION

We successfully implemented both the code needed to control a robot to play chess as well as a simulator to simulate up to two robots playing chess at a time. Additionally, we implemented support for a human player to play against the robot opponent.

## XII. CERTIFICATION AND PEER EVALUATION

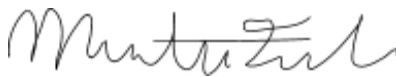
I participated and contributed to team discussions on each problem, and I attest to the integrity of each solution. Our team met as a group both physically and virtually throughout this semester.

A handwritten signature in black ink that reads "Brett Levenson". The script is cursive and fluid.

*Brett Levenson*

A handwritten signature in black ink that reads "Luke Cohen". The script is bold and cursive.

*Luke Cohen*

A handwritten signature in black ink that reads "Matthew French". The script is cursive and somewhat stylized.

*Matthew French*