The King's Gambit 2.0®
Honors Design and Development 2024

Jack Kellaher, Jason Nguyen, Youssef Ibrahim, *Juniors*
Noa Strupinsky, Hannah Bialik, Kael Jespersen, *Sophomores*
Ivan Marsic, Aaron Mazzeo, Rana Darst *Alumni Advisors*

***Abstract*—The King's Gambit 2.0 is a fully autonomous robot capable of recognizing, computing, and physically executing chess moves.This project combines hardware and software to design an autonomous, intelligent robot capable of playing a full game of chess against a human opponent. Our product aims to expand the scope of digital engines to the physical world. Playing against our product will allow anyone to interact directly with the mind of the greatest chess players.**

CONTENTS

I. INTRODUCTION

A. Problem Statement

THIS project was inspired by two main factors: the rapid growth of chess and the current methods with which people play chess. On one hand, people can play via online engines where you are paired against another human or even computer generated AI with a similar skill level to you. This affords users the opportunity to rapidly improve against like opponents in quick and efficient games, but it does not provide the same *feeling* as in-person chess because users are clicking for moves instead of physically adjusting their locations. Contrarily, people can play in-person chess against friends or family, but it is difficult to accurately gauge and pair skill levels. In this instance, one person may be significantly and objectively better than their opponent which leaves to little opportunity for improvement among both parties. This project aims to merge the best components of each aforementioned method, by providing the flexibility of computer generated AI opponents, with the same mechanics as in-person chess.

B. Solution & Design Components

This project seeks to provide anyone the experience of physically playing against a Chess Grandmaster without actually having to meet anyone face to face. To do this, we created a physical gantry and camera system that analyzes a physical chessboard and actively responds to moves made by picking up pieces and moving them across the board. To create this product, three design elements were considered; Theoretical, Digital, and Physical.

1) Theoretical Element: The theoretical element utilizes mechanical calculations to physically execute the instructions suggested by Stockfish API, via a gantry. Accomplishing this necessitates testing to effectively translate instructions to actuator movement. These mechanical calculations utilize noted differences in a piece's position in between turns. When a piece moves from one spot to another, the spot it used to occupy is represented by a negative number, and the spot it proceeds to occupy is represented by a positive number, generating a difference referred to as a "delta."

2) Digital Element: Our digital element utilizes a camera to detect the chessboard and the pieces on it, then uses software to convert the board and the pieces on it to digital values, namely FEN notation, the standard notation for chessboard arrangements. In addition to FEN notation are the aforementioned deltas. These are fed into Stockfish API, the world's most powerful CPU chess engine, which will then calculate the next best move based on the level of difficulty that the player sets it to, and feed this information to the gantry.

3) Physical Element: The physical element, the gantry, synthesizes the previous two elements to create a robot with great functionality. The design fits around any play space less than or equal to 16". Servo motors, stepper motors and timing belts actuate the gantry in discrete steps to pick up and move pieces. The frame consists of off-the-shelf hardware and custom-designed 3D printed components. A micro-controller and stepper motor drivers instruct the motors on the gantry. The electronic components and motors are powered from the laptop connected to the micro-controller and an external power supply, respectively.

II. Equipment and Methodology

A. Materials

The components were purchased through Amazon vendors. The complete bill of materials is below:

Bill of Materials

Material	Function	Quantity
Raspberry Pi Pico W	Instruction	1
Raspberry Pi 4B	Processing	1
Logitech HD Laptop Web-cam C615	Processing	1
TMC2209 Driver	Instruction	2
400 Tie-Point Breadboard	Instruction	2
LCD Display Module	Instruction	1
PNY 64GB SD Card	Instruction	1
Tactile Push Button	Instruction	3
NEMA 17 Stepper Motor	Actuation	2
S51 Servo Motor	Actuation	2
GT2 Pulley Wheel	Actuation	2
GT2 Idler	Actuation	2
GT2 Timing Belt	Actuation	2000mm
24V Power Supply	Power	1
3030 Aluminum T-Slot	Structure	2540mm
Bearing Rod (8/450mm)	Structure	2
8mm Hardened Steel Rod	Structure	1880mm
LM8UU Linear Bearing	Structure	6
L-Shaped T-Slot Joint	Structure	6
PLA	Structure	500g
PET-G	Structure	200g
M2 Screw (12/16/20mm)	Structure	23
M3 Screw (12/16mm)	Structure	22
M2 Nut	Structure	12
M3 Nut	Structure	14
PVC Pipe	Structure	1
Gooseneck Phone Holder	Structure	1

B. Camera and Processing

All of the image processing is done using Python, making use of multiple python libraries including:

- OpenCV
- Numpy
- Subprocess
- Stockfish library (pypi.org)
- Inference-sdk

The first step of the translation from the physical to the digital board is to take an image of the board. In order for this image to be correctly processed, it must be taken from directly over the board. The camera must be high enough to capture the entire board, and angled such that the bottom bound of the image is parallel to the bottom edge of the board. Additionally, the size of each square on the board must be consistent, the pieces must be marked with a clear and bright color, and the ambient lighting must be bright and consistent. These specifications ensure that the board and the configuration of the pieces on it are detectable.

Once the image is received, Artificial Intelligence is used to detect the bounds of the board. Using this information, the image is cropped to contain only the game board. This image is then perspective transformed to account for any minor inconsistencies in camera angle. The result of this stage, is an image with perfectly consistent squares, where the pieces appear as 'dots' on their respective squares.

```
    #AI Board Detection
def imageCropAndWarp(inputImPath, outputImPath):

    CLIENT = InferenceHTTPClient(
        api_url="https://outline.roboflow.com",
        api_key="ulxWhbY1nxtlA240URf6"
    )

    result = CLIENT.infer(inputImPath, model_id="
    chessboard-1hk4y/3")

    original_img = cv2.imread(inputImPath)

    coordinates = [(int(coord['x']), int(coord['y'])
    ) for coord in result.get("predictions")[0].get(
    "points")]

    #Calculate corners based on detected edges
    corners = find_corners(coordinates)

    #Perspective transform image
    transformed_image =
    perspective_transform_to_square (original_img,
    corners)

    #Ouput image to the correct file
    cv2.imwrite(outputImPath, transformed_image)
    return
```

With this new image, color threshold masks are used to identify the locations of the pieces of each player according to their colored markings. Our pieces are identified by green and purple markings, colors that contrast well with their surroundings, making them easily identifiable. We overlaid the image with threshold masks for each of these colors.

Because the image consist of uniformly sized and distributed chessboard squares, once the masked images from the green and purple masks are combined, the resulting image is split into an 8x8 grid to then process the image one cell at a time.

For each cell the script checks to see if there are more than 30 pixels of either green or purple in the cell. For each green piece, which corresponds to the white chess pieces, it will add a 5 to an 8x8 array stored as a file which holds the 'board state', the same is done for purple, but with a 2 added to the array (instead of a 5 which), representing the black pieces. Cells with less than 30 pixels of green or purple get stored as 0's and are considered empty cells.

```
    for i, cell in enumerate(cells):
        hsv = cv2.cvtColor(cell, cv2.COLOR_BGR2HSV)

        green_lower = np.array([35, 50, 50])
        green_upper = np.array([75, 255, 255])
        purple_lower = np.array([110, 50, 150])
        purple_upper = np.array([160, 255, 255])

        green_mask = cv2.inRange(hsv, green_lower,
    green_upper)
        purple_mask = cv2.inRange(hsv, purple_lower,
     purple_upper)
```

```
        green_count = cv2.countNonZero(green_mask)
        purple_count = cv2.countNonZero(purple_mask)

        if green_count > purple_count and
    green_count > 40:
            board_array[i // 8, i % 8] = 5
        elif purple_count > green_count and
    purple_count > 40:
            board_array[i // 8, i % 8] = 2
        else:
            board_array[i // 8, i % 8] = 0
```

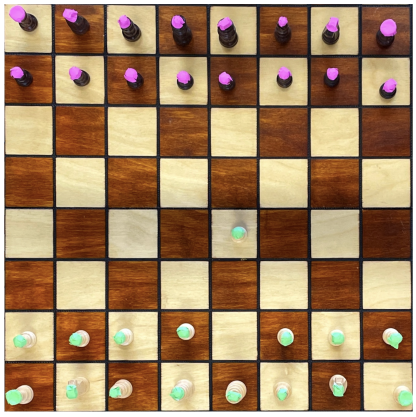

Fig. 1. Overhead view of the board.

C. Board States and Logic

Three special rules of chess allow us to process the image in the way we did, without going through the hassle of differentiating if a piece is a queen, pawn or rook.

- The board starts exactly the same
- Each move builds off of the previous move
- The game is played one move at a time

Because of these properties, we are able to track moves based on simple changes in the board state and identifying only if a piece moved and what color the piece was. We will be calling these changes 'deltas'. By tracking deltas and keeping track of the current board state, the previous board state, and the starting board position we are able to track an entire game and identify where each piece is and what color and type of piece it is.

since the current and previous board state are 8x8 matrices, what we do is we subtract the previous board from the current board and we get a new 8x8 array which has the deltas. In the new deltas array we know that a negative number is where a piece was and a positive number in that array is where that piece moved to.

The Third array is the starting board position, this starts at the beginning of the game with the standard chess board layout in an 8x8 array. This array now contains letter notations for pieces, ie. K for white king, N for white knight, q for black queen. The letters represent what kind of piece is in a position and the capitalization of those letters denotes the color. this array will be referred to as the 'mock fen' array, since FEN is written similarly with one major difference.

By taking the changes in the deltas array and overlaying that on the mock fen array we can make a new updated mock fen array. This new array corresponds to the new position on the physical board. However when doing this there are some considerations for slightly more complex moves that are not just piece movements. In chess the player can also capture pieces, castle their king, and en-passant.

Because of our choice in representing white pieces with 5 and black pieces with 2, we are able to still track these slightly more complicated moves. Imagine a white piece takes a black piece, On the differences array you would have a -5 where the white piece was originally but instead of a 5 where the white piece moved to it is now a 3 because it subtracted the 2 from the previous board position. Because these values don't have the same magnitude we can send that information to the gantry to let it know to move the piece in the black position first before moving the white piece

However it gets slightly more tricky when the capture is the other way around because when a black piece takes a white piece you get a -2 and a -3. You can still tell that the -2 was where the black piece started and because it moved to a square where there was previously a white piece it became a -3. We can still differentiate it but this time we take the most negative number and identify that that is where the piece moved to.

En-passant is calculated in the same way as normal captures since only 2 spots change, however when you castle 4 spots change. When we have the 4 spots change we just check to see what direction it happened and we know where the king and rook go and the calculations are done in the same way as normal piece movement.

Then comes the last part of the processing where we take this mock fen board and turn it into a FEN string so that it can be processed by the stockfish engine. Fen consists of several parts, the first is the piece placement data, then active color, castling rights, en-passant ability, and a half and full move counter for the 50 move rule

the piece placement data is obtained by taking the mock fen board and modifying it. in FEN consecutive zeros are labeled with a number, meanwhile in the mock fen array we have each zero individually shown to keep it an 8x8 array. We can simply compress the mock fen board into normal FEN notation and split up the rows by adding /'s in between each one

The rest of the fen is added accordingly in order of complexity, first because we are always playing one color and the robot plays the opposite color, the active color will always be the same for the computer so that is consistent, Full and half turn counters are done by checking how many times the code has been run to process a new board, and half moves will reset any time a pawn is moved or a piece is taken.

Castling rights is done by checking with each iteration of the image processing weather either the king or rook moved, castling rights for each color and their two respective directions are saved as a Boolean and are updated each time, if the king or rook on each side is moved, the Boolean is updated and is permanently set as false until the next game.

En-passant is checked by checking if a pawn moved each turn and then it checks if it moved 2 tiles or 1, if it moved 2 tiles it then checks to see if there are pawns of the opposite

color next to it. if there are then the en-passant is updated, and if not then it is set back to being null.

```
    neg_indices = np.argwhere(diff < 0)
    pos_indices = np.argwhere(diff > 0)

    if(len(pos_indices)>0 and len(neg_indices)>0):
        row1, col1 = neg_indices[0]
        row2, col2 = pos_indices[0]

    if (len(neg_indices) == 1 and len(pos_indices)
    == 1): # Standard movements (no takes) or white
    takes black
        mof[row2][col2] = mof[row1][col1]
        mof[row1][col1] = '0'
    elif (len(neg_indices) > 1 and len(pos_indices)
    == 0): #black takes white
        row1, col1 = neg_indices[0]
        row2, col2 = neg_indices[1]
        if (abs(diff[row1][col1])>abs(diff[row2][
    col2])):
            mof[row1][col1] = mof[row2][col2]
            mof[row2][col2] = '0'
        else:
            mof[row2][col2] = mof[row1][col1]
            mof[row1][col1] = '0'
    elif (len(neg_indices) == 2 and len(pos_indices)
     == 2): # castles
        negR1, negC1 = neg_indices[0]
        negR2, negC2 = neg_indices[1]
        posR1, posC1 = pos_indices[0]
        posR2, posC2 = pos_indices[1]
        if(negR1 == 7 and negR2 == 7): #white castle
            if(negC2 == 7): #KINGSIDE
                mof[negR1][negC1] = '0'
                mof[negR2][negC2] = '0'
                mof[posR1][posC1] = 'R'
                mof[posR2][posC2] = 'K'
            elif(negC1 == 0): #QUEENSIDE
                mof[negR1][negC1] = '0'
                mof[negR2][negC2] = '0'
                mof[posR1][posC1] = 'K'
                mof[posR2][posC2] = 'R'
        elif(negR1 == 0 and negR2 == 0): #black
    castle
            if(negC2 == 7): #KINGSIDE
                mof[negR1][negC1] = '0'
                mof[negR2][negC2] = '0'
                mof[posR1][posC1] = 'r'
                mof[posR2][posC2] = 'k'
            elif(negC1 == 0): #QUEENSIDE
                mof[negR1][negC1] = '0'
                mof[negR2][negC2] = '0'
                mof[posR1][posC1] = 'r'
                mof[posR2][posC2] = 'k'
    arr_str = "\n".join([" ".join(row) for row in
    mof])
```

D. Gantry Design

The design of the gantry frame is inspired by traditional Computer Numerical Control (CNC) machines and 3D printers. By implementing aspects from these existing machines into the gantry's design, we can avoid wasting time and resources attempting to fix the teething issues of an experimental design. 3030 aluminum T-slot extrusions make up the core of the frame, forming a square that the chess board will rest on. The square frame is held in place by L-shaped T-slot joints that fit within the inner-most channel of the square. 3D printed PLA joints slide onto the exterior of the square, and steel bearing rods fit into a channel inside the joint.

The gantry arm, which traverses over the board, also consists of aluminum T-slot extrusions, L-shaped T-slot joints, and 3D printed joints. At the bottom of the arm, 3D printed PET-G attachment pieces hold LM8UU linear bearings that slide along the steel bearing rods on the square frame. The PLA joints on the arm contain their own channels for steel bearing rods that run over the frame.

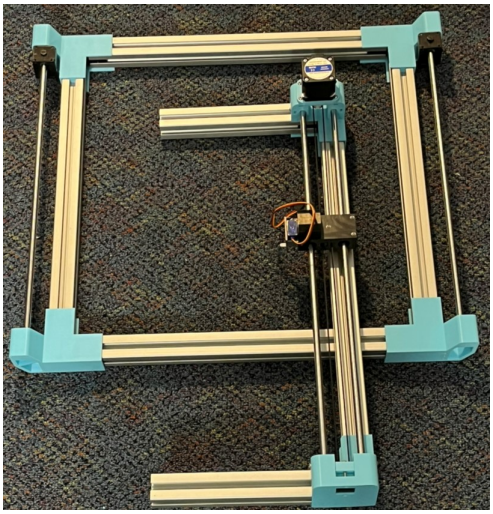

Fig. 2. Gantry Base and Arm with Prototype Tool-Head.

The tool-head is of a completely custom design, and consists of two halves. The first half, also known as the carrier, contains 4 channels for LM8UU linear bearings. Two run along the bearing rods on the gantry arm, allowing the tool-head to slide across the frame. The other two bearings serve to restrict the motion of the tool-head's second half, which will be referred to as the tool-head arm. A servo motor attached to the carrier converts its rotational motion into vertical, linear motion of the arm via a set of freely-rotating, mechanical linkages. The arm itself contains another servo motor which actuates a set of grabbers, which will come into contact with the chess piece.

The gantry arm is moved along the steel bearing rods via a system consisting of a stepper motor, pulley, idler, and timing belt. The components are attached to the frame through special connection points built into the 3D printed joints. The gantry arm itself also contains attachment points for another, similar system to move the tool-head along its own bearing rods.

The stepper motors and servo motors are hooked to a Raspberry Pi Pico W micro-controller, in tandem with two TMC2209 stepper motor drivers. The micro-controller is powered via a 5V USB connection. The 5V VSYS pin on the micro-controller powers the servo motors and stepper motor drivers, while the stepper motors draw from an external 24V power supply.

A camera mount was added in order for the camera to take the most accurate possible photos of the chess board and piece locations. This mount was created using a PVC pipe and a phone holder, and secured the camera approximately sixty-five centimeters above the chess board, a position where it is able to capture the entire chess board.

A user interface was added, which consisted of the LCD screen, buttons, and a custom mount modeled through Solid-

Works. The LCD screen is able to fit flush within the mount, and holes through the back of the mount were made for the button pins and wires that were soldered to them. The mount is able to fit securely within the aluminum T-slots. The buttons allow the player to control a variety of things before and after the game start.

E. Communication and Gantry Actuation

The LCD allows direct serial communication between the user and all of the robots components. The LCD consists of three buttons, with a variety of different options dependent on the state of the game. Before the game starts, the left and right most buttons control the difficulty of the Stockfish AI API, while the middle button starts the game with the selected difficulty. Once the game starts, the buttons now have different uses: the left most button now aborts the game which cancels the current ongoing game in case the player wants to start over, and the right most button gives the player the ability to end their turn after they make a move. The LCD displays current states of the game and any problems the robot finds. Scenarios where the player makes an invalid move, the robot takes a bad picture, and even if the processing of the robot malfunctions, are accounted for.

Once connected to their respective electronic components, the stepper motors and servo motors receive instructions from the micro-controller. A custom program written in Arduino, uploaded to the micro-controller, makes it possible to manually control the gantry via a serial communication interface. The main method of this program cycles through one of two possible switch cases. If the gantry does not currently hold a piece, the first switch case will be used. In this switch case, the user can communicate the next move to the micro-controller, whether that be to move the tool-head to a new position, move the tool-head arm vertically or actuate the grabber arms. Once the user inputs the next move through the serial port, the gantry will actuate that move and the micro-controller will wait for the next set of instructions. If the user instructs the gantry to pick up the piece, the loop will run through the second switch case. This switch case is nearly identical to the first, but the micro-controller will periodically instruct the grabber arms to push on the piece that is being carried. In theory, actuating the grabber arms while the tool-head is in motion will increase the frictional forces holding the piece and reduce the likely-hood that the piece falls.

This program utilizes two libraries to aid in smooth control over the stepper motors and servo motors. They are listed below:

- AccelStepper.h
- ServoC.h

The entire program can be found in Appendix A.

III. Results

We were successfully able to take pictures of our board and translate them into a digital board using Artificial Intelligence and our image processing software. From here, our program could generate a FEN string that could be sent to Stockfish API in order to generate the next best move. Our gantry was able to physically act upon this information, picking up pieces on the board and moving them in response to a move made by the player.

With all digital and physical components working successfully, the final prototype was completed, with a 3D rendering of it shown below.

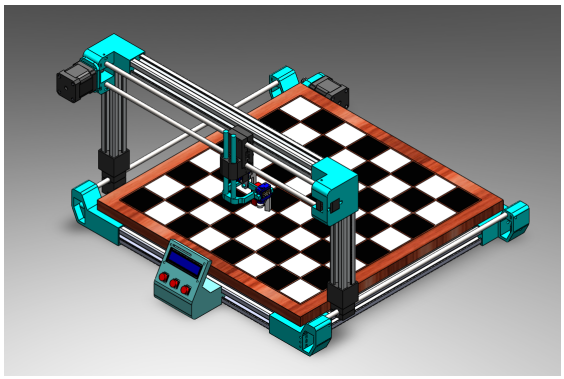

Fig. 3. 3D Rendering of the King's Gambit.

IV. Future Work

We aim to accomplish quality of life improvements, including a more optimized capturing system, as well as a more compatible and convenient product for travel and daily use. Improvements in cost can be made by using lighter materials on the gantry frame.

V. Conclusion

Currently, there doesn't exist a way for casual and competitive chess players to harness the insights of a chess engine when playing over a physical board. The King's Gambit 2.0 allows players of any skill to experience a physical game of chess while honing their skills against the highest level chess engines available; without having to use any digital interface of their own. Over the course of this semester, the authors were able to achieve functionality in each of the three elements of this project: our Computer Vision software successfully extracted the positions from the board, we connected with Stockfish API to find the optimal move, and the gantry successfully moved pieces across the board to execute those recommendations. We accomplished this using a combination of Computer Vision libraries, Stockfish's open source library, and Arduino code to drive the gantry movement. In our efforts to integrate these elements, we successfully enabled any human to play against the King's Gambit 2.0 in a full game of chess. In the future, we hope to create a more efficient product by incorporating the adjustments listed in the Future Work section. We hope that with these changes and continued development that the King's Gambit can evolve into a seamless experience available to all chess players, and can become commonplace for high school chess clubs and professional tournaments alike.

ACKNOWLEDGMENT

The authors would like to thank faculty members Ivan Marsic and Aaron Mazzeo, as well as Rana Darst for meeting with us and helping guide us through our project. Thank you to our DA Humza Syed for guiding us throughout the duration of the project. Thank you Dean Jean Patrick Antoine for being incredibly invested in our project with enthusiasm and motivation, and for affording us the opportunity to feature our project at the symposium.

APPENDIX A
CODE

```
#include <AccelStepper.h>
#include "ServoC.h"

ServoC myservo;  // create servo object to control a servo
ServoC myservo1;

#define servopin 16
#define servopin2 17 // assign pins for servos

int lastPos;
int pos1 = 15;
int pos2 = 55;
int pos = 0;    // variable to store the servo position

char column[] = {'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a'};
char row[] = {'8', '7', '6', '5', '4', '3', '2', '1'};

boolean hasPiece = false;
boolean armExtended = false;
boolean grabberClosed = true; // booleans to keep track of toolhead state

// Define some steppers and the pins they will use
AccelStepper stepper1(1,1,0);
AccelStepper stepper2(1,7,6);

int stepsPerSquare = 1960; // how many stepper motor steps to traverse a single square (980/inch, 44/mm)
int subDivideStep = 1; // how many discrete movements to traverse a single square
char instruct; //

unsigned long myTime;

void setup() {
    stepper1.setMaxSpeed(1400);
    stepper1.setAcceleration(2000);

    stepper2.setMaxSpeed(1400);
    stepper2.setAcceleration(2000);

    myservo.attach(servopin);// attaches the servo on pin 5 to the servo object
    myservo1.attach(servopin2);
    myservo.write(0);
    myservo1.write(pos1);
    lastPos = pos1;

    Serial.begin(9600);
}

void loop() {
  if (!hasPiece) {
    while(Serial.read() >= 0) {}
    while(Serial.available() == 0) {}
    instruct = Serial.read();
    switch (instruct) {
      case 'm':
      {
      Serial.println("move");
      Serial.println("Current Position");
      Serial.println(stepper1.currentPosition());
      Serial.println(stepper2.currentPosition());
      while(Serial.read() >= 0) {}
      while(Serial.available() == 0) {}
      String sInst = Serial.readString();
      char cInst[sInst.length()+1];
      sInst.toCharArray(cInst, sInst.length()+1);
      Serial.println(cInst[0]);
      Serial.println(cInst[1]);
      int j;
      for (j = 0; j<sizeof(column); j++){
        if (cInst[0] == column[j]) break;
      }
      Serial.println("x-axis");
      stepper2.moveTo(stepsPerSquare/subDivideStep*j);
      Serial.println("target x");
      Serial.println(stepper2.targetPosition());
      while (stepper2.distanceToGo() != 0) {
        stepper2.run();
      }
      for (j = 0; j<sizeof(row); j++){
        if (cInst[1] == row[j]) break;
      }
    // Controls y-axis
      Serial.println("y-axis");
      stepper1.moveTo(-stepsPerSquare/subDivideStep*j);
      Serial.println("target y");
      Serial.println(stepper1.targetPosition());
      while (stepper1.distanceToGo() != 0) {
      stepper1.run();
      }
      }
      break;

      case 'q':
      Serial.println("move up/down");
      if (armExtended) {
        for (pos = 0; pos <= 180; pos += 5) { // goes from 0 degrees to 180 degrees
          // in steps of 1 degree
          myservo.write(pos);    // tell servo to go to position in variable 'pos'
          delay(15);                          // waits 15ms for the servo to reach the position
        }
        for (int i = 0; i <= 30; i++) {
          myservo.write(180);
          delay(15);
        }
      } else {
        for (pos = 180; pos >= 0; pos -= 2) { // goes from 180 degrees to 0 degrees
          myservo.write(pos);
          // tell servo to go to position in variable 'pos'
          delay(15);                          // waits 15ms for the servo to reach the position
        }
      }
      armExtended = !armExtended;
      break;

      case 'e':
      Serial.println("grab/let go");
      if (grabberClosed) {
        int i;
        for (i = lastPos; i <= pos2; i += 2) {
          myservo1.write(i);
          delay(15);
```

```
        } // waits 15ms for the servo to reach the position
        lastPos = pos2;
      } else {
        int i;
        for (i = lastPos; i >= pos1; i -= 2) {
          myservo1.write(i);
          delay(15);
        }
        lastPos = pos1;
      }
      grabberClosed = !grabberClosed;
      if (armExtended && grabberClosed) {
        hasPiece = true;
      } else if (!grabberClosed) {
        hasPiece = false;
      }
      break;

      case 'p':
      Serial.println("setting zero");
      stepper1.setCurrentPosition(0);
      stepper2.setCurrentPosition(0);
      break;
    }

   // this is the case where a piece is held
  } else {
    myservo1.write(pos1);
    delay(15);
    while(Serial.read() >= 0) {
      myservo1.write(pos1);
      delay(15);
    }
    while(Serial.available() == 0) {
      myservo1.write(pos1);
      delay(15);
    }
    instruct = Serial.read();
    switch (instruct) {
      case 'm':
      {
      Serial.println("move");
      Serial.println("Current Position");
      Serial.println(stepper1.currentPosition());
      Serial.println(stepper2.currentPosition());
      while(Serial.read() >= 0) {
        myservo1.write(pos1);
        delay(15);
      }
      while(Serial.available() == 0) {
        myservo1.write(pos1);
        delay(15);
      }
      String sInst = Serial.readString();
      char cInst[sInst.length()+1];
      sInst.toCharArray(cInst, sInst.length()+1);
      Serial.println(cInst[0]);
      Serial.println(cInst[1]);
      int j;
      for (j = 0; j<sizeof(column); j++){
        if (cInst[0] == column[j]) break;
      }
      Serial.println("x-axis");
      stepper2.moveTo(stepsPerSquare/subDivideStep*j);
      Serial.println("target x");
      Serial.println(stepper2.targetPosition());
      myTime = millis();
      while (stepper2.distanceToGo() != 0) {
        if (millis() > myTime+25) {
            myservo1.write(pos1);
            myTime = millis();
        }
        stepper2.run();
      }
      for (j = 0; j<sizeof(row); j++){
        if (cInst[1] == row[j]) break;
      }
    // Controls y-axis
      Serial.println("y-axis");
      stepper1.moveTo(-stepsPerSquare/subDivideStep*j);
      Serial.println("target y");
      Serial.println(stepper1.targetPosition());
      while (stepper1.distanceToGo() != 0) {
        if (millis() > myTime+25) {
            myservo1.write(pos1);
            myTime = millis();
        }
        stepper1.run();
      }
      }
      break;

      case 'q':
      Serial.println("move up/down");
      if (armExtended) {
        for (pos = 0; pos <= 180; pos += 5) { // goes from 0 degrees to 180 degrees
          // in steps of 1 degree
          myservo.write(pos);
          myservo1.write(pos1);// tell servo to go to position in variable 'pos'
          delay(15);// waits 15ms for the servo to reach the position
        }
        for (int i = 0; i <= 30; i++) {
          myservo.write(180);
          delay(15);
        }
      } else {
        for (pos = 180; pos >= 0; pos -= 2) { // goes from 180 degrees to 0 degrees
          myservo.write(pos);
          myservo1.write(pos1);
          // tell servo to go to position in variable 'pos'
          delay(15);                       // waits 15ms for the servo to reach the position
        }
      }
      armExtended = !armExtended;
      break;

      case 'e':
      Serial.println("grab/let go");
      if (grabberClosed) {
        int i;
        for (i = lastPos; i <= pos2; i += 2) {
          myservo1.write(i);
          delay(15);
        } // waits 15ms for the servo to reach the position
        lastPos = pos2;
      } else {
        int i;
        for (i = lastPos; i >= pos1; i -= 2) {
          myservo1.write(i);
          delay(15);
        }
        lastPos = pos1;
      }
      grabberClosed = !grabberClosed;
      if (armExtended && grabberClosed) {
        hasPiece = true;
      } else if (!grabberClosed) {
        hasPiece = false;
      }
      break;

      case 'p':
      Serial.println("setting zero");
      stepper1.setCurrentPosition(0);
```

```
265         stepper2.setCurrentPosition(0);
266         break;
267     }
268     }
269 }
```