A3: Chess Playing Robot

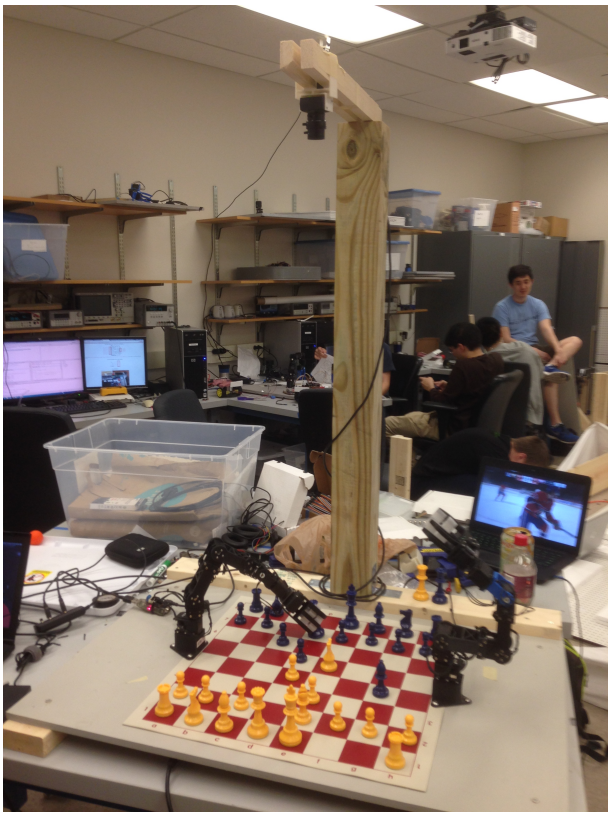
Alexander Berman, Justin Kaput, Zheng Hao Tan
EECS 467 Winter 2015 - A3 Report
{anberman, jkaput, tanzhao}@umich.edu

Introduction

Chess-playing robot is a robotic arm that is able to play chess with a human using its robotic arms. It demonstrates human computer interaction

There are 4 major sections for this project, which are image processing, arm kinematics, chess engine and artificial intelligence.

Setup



The image above shows how our chess-playing robot is set up. We have two robotic arms mounted on both ends of the board. The chessboard itself was also mounted on the board, and we built a stand so we could mount the overhead camera directly above the chessboard.

The image above shows how the camera is taped to the wooden piece at the top of the whole rig.

Chess Engine

The chess engine follows an object-oriented structure to abstract the code and allow it to be very modular. The main class is Game. Game has two instances of Player objects, which will be described later, and has a 2D array of Square objects that represents the chess board state. This contains methods to set player modes, print out board states, and command players to take their turn in order.

Each Square includes information about its location and has a Piece object pointer. Game has a function to move a piece and change the game state. This move function takes care of special cases like castling and promotion.

Piece is an abstract class, with a function to return the valid moves for a given piece. Piece has a function to return a string corresponding to type, and a function to return the pieces location on the board. Each type of chess piece (Pawn, Knight, King, etc.) is a derived class from Piece. They implement a virtual function that returns all possible blocks this piece can move to. This function also filters out moves that would put the players own king into check.

The abstract class Player has one virtual function to execute a turn. It contains information about if its the white player or the black player. It also contains a list of Piece object pointers to keep track of all of its uncaptured pieces. There is a function to remove a piece from this list for Game to call when capturing in its move function. It has a function that will return a boolean value if it is in check or cannot make a move.

There are two classes derived from Player that require human input. TextHumanPlayer is a debugging tool that uses neither the camera nor the arm. It first asks the user for a piece to move. It will then print out the valid squares that a piece can move to, at which the user will enter which move they desire to complete. It will then call the Games move function to update the game state. BoardHumanPlayer uses the camera to infer user moves. It works by finding the difference in location of colored pieces on the board. The method for finding the color and location of pieces relative to the board is described in the Image Processing section. For example it can see a white piece in a square for the first image, but in the next image that piece is gone and there is a white piece in a new square. The program will then infer that piece in the first location moved to the second location. In the same example if the piece appeared on a square a black piece was before, it would infer a capture. If it sees multiple pieces of the same color move and they are in corresponding coordinates to the rook and king, it will infer a castling maneuver. This program will catch bad inferences, from bad movements or bad images, and will ask the user to reset the board to before the move and try again. The user ending their turn is specified by them pressing the enter key.

Derived from Player is AIPlayer, which is also an abstract class. This has a difficulty mode which can be set and is stored. There is a implemented function that will determine the AI move based on the algorithm described in the Artificial Intelligence section below. Two classes derive from AIPlayer and implement chess without human interaction. TextAIPlayer will simply call the games move function based on the AI move from the AIPlayer class. BoardAIPlayer will not only update the programs game state, but will move the arms to the appropriate locations to modify the physical board game state. The arm control is described further in the Kinematics section. BoardAIPlayer controls where the arms move in terms of the programs game state. If one arm physically cannot move a piece to the other side of the board, it will first attempt to pass it to the middle four squares if any of them are empty. Otherwise it will hold the piece in the air and prompt the user to move the piece from that arm to the other arm. Before and after every movement, it will take images to confirm a move did occur using similar logic to the BoardHumanPlayer inference. If the move did not occur, it will try until it is successful before ending the turn.

In this picture a TextHumanPlayer plays against an TextAIPlayer. The game will not let the player make invalid moves and can

```
  a b c d e f g h
8 R H B Q K B H R
7 P P P P P P P P
6
5
4
3
2 p p p p p p p p
1 r h b q k b h r
White Turn:
enter in coordinate of piece to move: f3
Invalid piece, try again
enter in coordinate of piece to move: f2
selected : Pawn
----------------------
valid moves:
f3
f4
----------------------
enter in coordinate of piece's destination: f5
Invalid Move, try again
enter in coordinate of piece to move: f2
selected : Pawn
----------------------
valid moves:
f3
f4
----------------------
enter in coordinate of piece's destination: f4
  a b c d e f g h
8 R H B Q K B H R
7 P P P P P P P P
6
5
4           p
3
2 p p p p p   p p
1 r h b q k b h r
Black Turn:
Best score: 0
Knight at 'g 8'
to 'f 6'
  a b c d e f g h
8 R H B Q K B   R
7 P P P P P P P P
6           H
5
4           p
3
2 p p p p p   p p
1 r h b q k b h r
```

tell the user what moves can be made. This functionality was needed to implement the AI, so it was easy to apply to the human interface. BoardHumanPlayer will also stop the human from making bad moves.

Game is in charge of both setting the type of player and the difficulty settings for the players if applicable. A string in the format $'' < character > \vee < character >''$ specifies the game mode. p denotes BoardHumanPlayer, b denotes BoardAIPlayer, h denotes TextHumanPlayer, and c denotes TextHumanPlayer. The first character in the string specifies the white player, the second character is always v, and the third character is the black player. For example, during the demonstration we use pvb which means a human white player and a robot black player. There is a function to set difficulty of AI Players given a string easy, medium, and hard. This function will do nothing if used on a human player mode. The main program plays games on loop. Modes like cvc are good for testing the robustness of the program, as the computer AI will battle itself on loop without human input. The program can run for many hours on end without any problems occurring. By setting the difficulties of the two AIs to different modes, we can also assess the effectiveness of the difficulty level on the AI. The AI which used deeper depth always beat the AI at a shallower depth in our final AI implementation. In the same way, bvb will test the robot arm movement on loop. The Game class has a function that returns a boolean when the game is over. When the Game is over the main program starts another Game with the same settings. The program is compiled into the binary named chess.

Artificial Intelligence

The AI for this system uses an implementation of minimax with alpha-beta pruning. To summarize the concept, the AI essentially searches through all possible moves for a set number of turns alternating players. It then selects the move on the first turn that results in the best heuristic score for that many turns. On the opposing players turns, the heuristic is the same, but the sign is flipped since an opponents good move is bad for the thinking player. The larger the set depth, the more it can predict the future and the smarter it becomes. The different difficulties are made by setting this maximum depth value. However, the deeper the tree, the search space increases exponentially. To keep the program from taking forever, pruning of this search tree is necessary. Alpha-beta pruning removes searching both players best moves. This greatly improves the time performance.

In our implementation, each branch is a thread and it will not enter or leave a branch if that branch is a bad move. At each level, a deep copy is made of the game, and the pieces are moved with the same functions as moves outside the AI. This means the deep copy will

be changed at each step rather than modifying the internal state and undoing the move afterwards. The latter method would be better for single thread code, but we multi-thread so deep copies are mandatory.

The heuristic for each board state is a simple weighted sum of pieces on the board, with a special case if the opponent is put into checkmate. Each type of piece is assigned a constant value. The heuristic is equal to the sum of the players pieces values, minus the sum of the opponents pieces values. If the opposing player is in checkmate, the heuristic adds a very large number proportionate to the shallowness of the board it is evaluating in the search. This will make it not only prefer moves that result in checkmate, but will make it prefer quicker checkmates over later checkmates. This also applies to avoiding sooner checkmates over later checkmates. The values for each piece is 3.33 for Bishop, 3.2 for Knight, 1 for Pawn, 8.8 for Queen, and 5.1 for Rook. In the below figure, the black AI makes a move to put its white opponent in check and capture a bishop, resulting in a heuristic score of 2.66.

```
  a b c d e f g h
8 R H     K B H R
7 P   P P   P
6         P   P
5   P           Q P
4       q     p
3
2 r p p   p p b p
1   h b     k   r
Black Turn:
Best score: 2.66
Queen at 'g 5'
to 'c 1'
  a b c d e f g h
8 R H     K B H R
7 P   P P   P
6         P   P
5   P             P
4       q     p
3
2 r p p   p p b p
1   h Q     k   r
White Turn:
is in Check
```

IMAGE PROCESSING

The image processing portion of the project allows us to detect the chess pieces on the chessboard.

For all the calibrations ranging from masking to blob detection are partially implemented in A2. Since we organized and abstracted quite a lot of functionalities into classes, we could easily plug it into our chess-playing robot program. One major thing that we had to change was to account for the chessboard without any given registration squares.

We chose the colors of the chessboard and pieces to be red, maize and blue respectively. These colors give us good and distinguishable HSV values, which will help us a lot onblob detection. We went with a brown and white chessboard previously, but we realized that the blue chess pieces were dark enough that it blends into the brown squares on the chessboard.

$vector < vector < block\_info >> board$;

$block\_info$ contains x and y coordinates, as well as occupied and isWhite variables for the pieces. We had image coordinates stored in a 2D vector as well and then access them if we wanted to pick up or place pieces. We had quite a few bugs implementing such as returning the center of the red squares instead of the center of the chess pieces, as there is no guarantee that the chess pieces are directly in the center even though it is within a red square.

Since we only blob detect the red squares, all white squares and their image coordinates will need to be estimated to be halfway between two given red square coordinates. We went with several methods trying to reconstruct this, but we eventually settled with calculating the white square coordinates via the red squares on the left and right of it instead of just grabbing the leftmost and rightmost red square then divide them into 4 protions. By doing it relative to its neighbors, we managed to reduce the error of the white

squares accumulated across the chessboard.

The chess pieces will also be blob detected, and we figured out which square they were in by comparing it (absolute difference) with the image coordinates matrix that we saved.

The images above shows the setup for the chess pieces when they are blob detected in the chess program.

Since we build the whole chess board off the registered red squares, the red squares and their coordinates will have to be perfect (all 32 of them) where every image coordinate are consistent. Sometimes, these coordinates will go off if there are too few/many squares detected. If this is not acccurate, it will also throw off our white square coordinates, chess piece position coordinates, and arm coordinates when it tries to reach for a chess piece. To do this, we figured out that it was better to add a feature to the chess game program that allows us to calibrate the board until it was perfect before the game could start.

There are 6 distinct pieces (Pawn, Knight, Bishop, King, Queen, Rook) for each player, and we didn't have enough colors to distinguish these pieces from each other. To overcome this problem, we blob detected the board and saved the pieces based off the initial configuration, which are assumed to be correct at the start of every game. For example, the pawn will always be in the row in front of the other pieces, and rooks will always be in the corners of the board, and this will ensure that the state for each chess piece is saved correctly during gameplay.

One of the harder things we had to overcome was rotations and translations of the chessboard. Their orientations, especially when they are rotated 90 degrees clockwise/counterclockwise, will throw off our white square coordinates when we add a

positive/negative offset from red squares on the left/right of it. We solved this by blob detecting all red squares first, then sort them by the x-axis (left to right) then check the first 2 leftmost red squares on the first 2 rows. We then check if the top left corner square is red by checking which of the two row squares have a greater x value.

After having this working, and due to the inconsistency of the blob detection squares that we get, we eventualy mounted the overhead camera to the board, and we no longer need to account for any other orientation.

Coordinate Conversions

We maintained four sets of coordinates for our project - board coordinates, image coordinates, left arm coordinates, and right arm coordinates. Board coordinates are the indices of the squares, ranging from 0 to 7 in both the x and y directions. The x direction corresponds to a-h on the chessboard, with 0 matching a and 7 matching h. The y direction corresponds to 1-8 on the chessboard, essentially shifting the numbers down by 1 to work with the C++ convention of 0-indexed arrays.

The image coordinates are the pixel numbers from the camera image. At the beginning of the program, we take a picture of the board and store the centroids of each square in a 2-dimensional vector. We index this vector using board coordinates, making it easy to go from a board coordinate and convert it into image coordinates.

Once we have the image coordinates, we need to convert those into arm coordinates so that we can pick an arm and tell it where we want it to go. During development, we tried four different methods of converting point-to-point correspondences into a transformation matrix. First, we used the LU decomposition method that we had used for A2. This used three point-to-point correspondences and the gsl_linalg library to produce the transformation matrix. This produced an unreliable transformation matrix, where the arm would sometimes move to where we clicked and other times would not. Then we asked other groups what they had done to produce this matrix for A2, and they pointed us in the direction of simple matrix inversion. This showed similar results to the LU decomposition method. We then tried the SVD method shown in the lecture slides. After a couple tries, we couldnt get this method to work correctly, and the resulting transformation matrix produced wildly incorrect mappings. At this point, we grew discouraged and saved the mappings of every square for each arm in their respective coordinates in a text file. This produced strange results, where the reproduced x and y values were the same, but the position of the arm was not where we had calibrated it to. We talked to Colin about this issue, and he pointed us in the direction of the homest library. This takes any number of point-to-point correspondences and produces a 3x3 affine transformation matrix. This method produced better results than any of the previous methods, and so this is what we used for the final project demo.

Arm Kinematics

For this project, we ended up rewriting almost all of the kinematics from A2. When picking up, we used the geometric method from A2 to determine the joint angles when the radius to the point of pickup allowed us to maintain the wrist perpendicular to the ground position. When this radius was too large, we used the kinematic equations and a version of gradient descent to determine the best angles. When given an (x, y, z) point that we want the arm to move to, we first get the base angle using atan2(y, x). Then, we assume the rest of the arm moves in a 2-dimensional plane, and find the angles that get us closest to the desired radius and height in that plane,

where the radius is the square root of the sum of the squares of x and y, and the height is z. Our heuristic for gradient descent tries to minimize not only the distance between the end effector and the desired location, but also the angle between the last link and the plane perpendicular to the ground. This makes the arm prefer positions closer to the result of the geometric solution over solutions where the final link is nearly parallel to the ground. When we pick up a piece, we record the angle and height for use later. We moved the arm two joints at a time, moving first the wrist and base, then the elbow and shoulder. This made the arm drop in on the piece, rather than sweeping across the top of the pieces. Once we grabbed the piece, the arm moved up to a height so that we could turn the base freely without hitting other pieces.

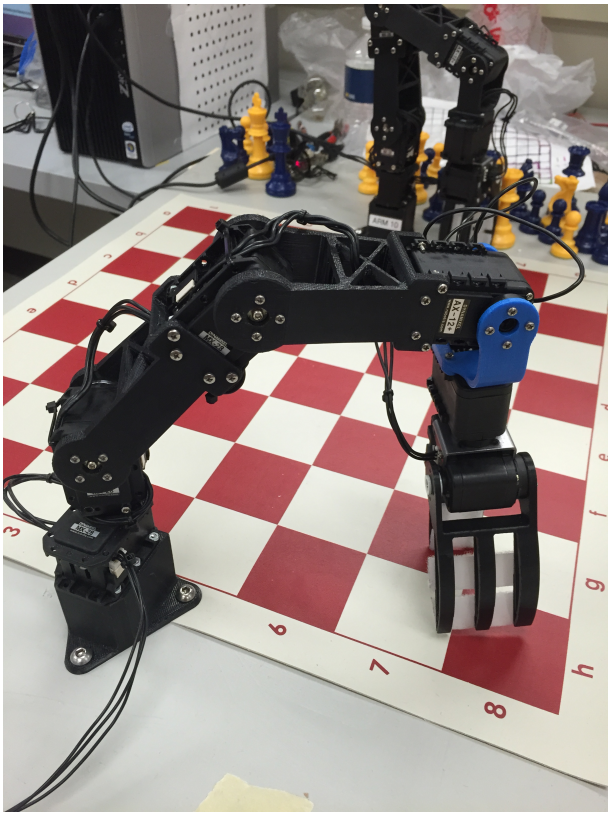
When placing a piece, we use the angle and height that we picked the piece up at to modify the end effector of our robot. This lets us use a similar heuristic as when picking the piece up for gradient descent, and make sure we place the piece as closely to normal to the playing surface as possible. Once we had the angles, we moved the arm similarly to when we picked it up - first moving the base and wrist followed by the elbow and shoulder. Moving the arm this way would occasionally cause the arm to push the fingertips into the board, and would require us to unplug the arm. Since this was a rare but persistent error, we developed a mechanism for making it so we did not have to rerun the program when this happened. Instead of killing the program and restarting, we introduced a timeout that required that the arm move to its desired location within ten seconds, or else it would default to the straight up home position. This allowed enough time for us to unplug the arm, replug it in, and restart the driver. Once the driver was restarted, the arm would return to its home position then continue from where it left off. The arm was much less prone to this error when moving from its home position.

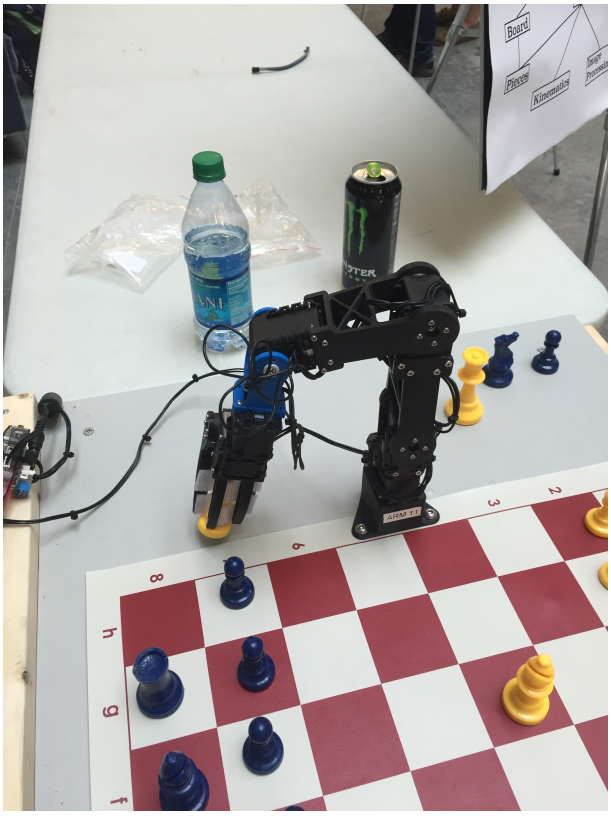
Picking up and placing a piece is simple for the case where the same arm is used to pick up and place the piece, but becomes more difficult when different arms are needed to reach the pick up and place locations. We toyed with several ways to accomplish this, including a direct handoff between the arms or using small cups to flip the piece between the arms. Eventually we decided on using a combination of techniques to accomplish this. If there was one of the center four squares open, we would use it to place the piece from the arm that picked it up, then pick it up from the other arm to move it to the desired location. If none of the four center squares was available, we defaulted to using a human to move the piece from one arm to another. In this case the arm that picked up the piece would stand straight up and release the tension on the fingers. When the piece was pulled out of the fingers by the human, the fingers on that arm would close, and the fingers on the opposite arm would open. Once the human put the piece in the fingers of the other arm, those fingers would tighten on the piece and the arm would resume moving to where it needed to move.

GAMEPLAY

To run the main chess game, all calibrations have to be done on the side, including the color/image calibrations, arm kinematic calibrations as well as any masking functions that we do on the side.

We then run the chess program, which will prompt us to remove all the chess pieces from the board. This has to be done because the blobs of the red squares might not be big enough to be considered a blob by our program, and we need to check if there are 32 of them as well.





After that, the chess pieces go on the board in the correct order, and at every turn, the program takes a snapshot of the board state and does the necessary checks to make the next move.

The images below shows the chess-playing robotic arm in action.

Other Challenges

On demo day, we had trouble with our color calibrations in Tishman Hall because of the brightness levels that change quite frequently. Since the atrium has an open top design which allows natural sunlight to come through, the major light source wasn't reliable as clouds drift past and cover up sunlight periodically. This causes our calibrated squares to be inaccurate and thus, no blobs were detected at certain timestamp in the game.

The blobs will disappear and reappear depending on the lighting conditions, which will throw off all our values for determination of the board state. We tried bounding the HSV ranges during brighter and darker conditions and hope that it will give us a better HSV range to capture all the blobs.

This was a problem too because by expanding the HSV range for red squares, we were getting false positives on blobs that formed on white squares, which were not what we wanted to be considered as red blobs. This is a flaw in using a HSV color space implementation as somehow the whiter squares had HSV values within the red HSV range.

If we had sufficient time to account for this, we would give it database of blobs with color labels that go with them. We will then apply a machine learning algorithm such as Support Vector Machines (SVM) and train it via a training set of red blobs of different shades. We can then repeat this process for maize and blue chess pieces as well. We can then use confidence values for predicted colors in blobs during the application to identify which color the blob belongs to if any. With a larger training set that has good variance in lighting, our chess program should be able to identify the blobs' color with greater accuracy.

The chess program was working in the lab and in the Tishman Hall early in the morning before sunrise. We managed to play several games of chess, and robot only misses the chess pieces occasionally. The image below shows an IA playing with our chess program.