# CVChess: Computer Vision Chess Analytics

Jay Hack and Prithvi Ramakrishnan

## Abstract

*We present a computer vision application and a set of associated algorithms capable of recording chess game moves fully autonomously from the vantage point of a consumer laptop webcam. This consists of two main algorithms, (1) a hough transform-based algorithm for finding a homography relating board coordinates to image coordinates, and (2) a model of chessboard colors and occlusions that allows us to account for and infer piece movement in real time. We provide a video demonstration of the application applied to a real chess game and describe experiments in which our developed algorithms significantly outperform a naive baseline. All code is open sourced and available on GitHub. (See Below)*

*Code*: github.com/jayhack/CVChess

*Video Demo*: youtube.com/watch?v=iZOA1ew-zYc

## 1. Introduction

While the game of chess has greatly benefitted from the information age, including analytical algorithms, online playing environments and game databases, there have been relatively few advances that improve the experience of over-the-board chess. One such problem is the tediousness of recording the sequence of moves performed in a game.

We aim to improve this aspect of the game by creating a robust, automated system that leverages computer vision in order to provide insights into chess games played on physical boards. In particular, we would like to implement a set of CV algorithms that allow us to determine the full state of the board at all times using widely available hardware (namely a Macbook Pro next to the board.) We will then provide users with recordings of their games that are easily portable to common chess databases and engines, so that a higher-level analysis might immediately take place.

This project is motivated by several established use-cases for similar systems, as well as problems with current systems that we can empathize with. Perhaps most significantly, the process of hand-recording games and inputting them into computers for algorithmic analysis is extremely tedious. Furthermore, real-time aids such as score-keeping, time-keeping, live analysis, and even automated coaching can be easily achieved given a utility for identify moves in real time. Given the prevalence of open research and tools for chess analysis, we suggest that our project will allow chess players the convenience and pleasure of playing chess on a physical board while maintaining the ability to leverage analytic advancements in computer chess.

## 2. Previous work

Previous work with determining chess positions from images is very limited. The majority of existing work has been conducted by independent hobbyists, independent of established research institutions, and without published experimentation or results. Most research on the problem of identifying chessboard pieces has made one or more of the following simplifications:

- The camera is mounted directly above the board, leading to minimal perspective distortion
- Plain background, making it easy to detect the chessboard
- Multiple cameras, leading to minimal total occlusion.

We present examples of such research below:

Research involving finding chessboard gridlines from a top-down view is a popular computer vision problem [1], and research has been made into exclusively detecting chessboard

corners. One report [2] leverages the checkerboard black-and-white colors surrounding chessbaord corners as features to detect chessboard corners, achieving comparable results to other combined edge-corner detectors.

Another report [3] uses a setup of two cameras with perpendicular views of the chessboard at relatively low perspective angles to reduce the probability of simultaneous occlusion of a piece or square. To aid in determining the orientation of the chessboard the setup also places a marker on the board at the corner of a particular setup. However, by maintaining the initial setup, the researchers achieve live gameplay with a computer tracking the positions of pieces on the board.

It is also notable that the OpenCV function *findChessboardCorners* does not perform well at acute angles such as that found by a Macbook Pro's camera.

# 3. Approach

Our approach attempts to do the following, in an effort to be pragmatic and useful as a general-use program:

- Require minimal setup overhead
- Leverage established, open source tools
- Maximize robustness

In an effort to do this, we solved the problem with a very simple setup: A Macbook Pro (or any computer with a camera) on the side of the board on white's left, with the screen roughly perpendicular to the table's surface. This allows for easy setup and quick initialization. A full cycle of our program works as follows:

1. The computer is set up next to an empty chessboard and the program takes a snapshot of the board, allowing it to determine a mapping between board squares and the image regions they correspond to.
2. The players set up pieces on the board and gameplay commences.
3. After each player makes a move, they hit any key on the laptop, as they would hit a chess clock in tournament play. The program keeps track of the position after each move, capturing an image of the current state of the board every time a key is pressed.
4. The program recognizes when the game is finished (i.e. an end state is reached), and terminates

# 4. Technical Details

## Terminology

- Board coordinates: (2x1) vectors that represent points on the chessboard; (0, 0) describes the top left corner of a8 and (8, 8) describes the bottom right corner of h1.
- Image coordinates: (2x1) vectors that represent points in the image.
- Board-Image Homography: A (3x3) projective matrix that maps homogenized board coordinates onto homogenized image coordinates.

The first task is to find the homography from board coordinates to image coordinates from an empty board taken in the first frame, at the perspective angle from a Macbook Pro.

## 4.1 Initialization: Finding the Board-Image Homography

In order to find the homography from the board to the image, it is necessary to find a set (at least four, but preferably many more) of point correspondences from board points to image points. The following algorithm was performed to achieve this:

1. Find definite corners by narrowing the output of a Harris Corner Detector with a SIFT Descriptor Classifier and filtering on points that closely snap to high-threshold vertical and horizontal lines obtained by a Hough Transform.
2. Use a RANSAC-like algorithm with linear and geometric regressions to find indices of vertical and horizontal lines from the Hough transform and assign each remaining image point to an integer pair representing a board point.

### *Finding a subset of chessboard corners*

First, we ran a Harris Corner Detector on the image to get a large set of possible corners from the raw image.

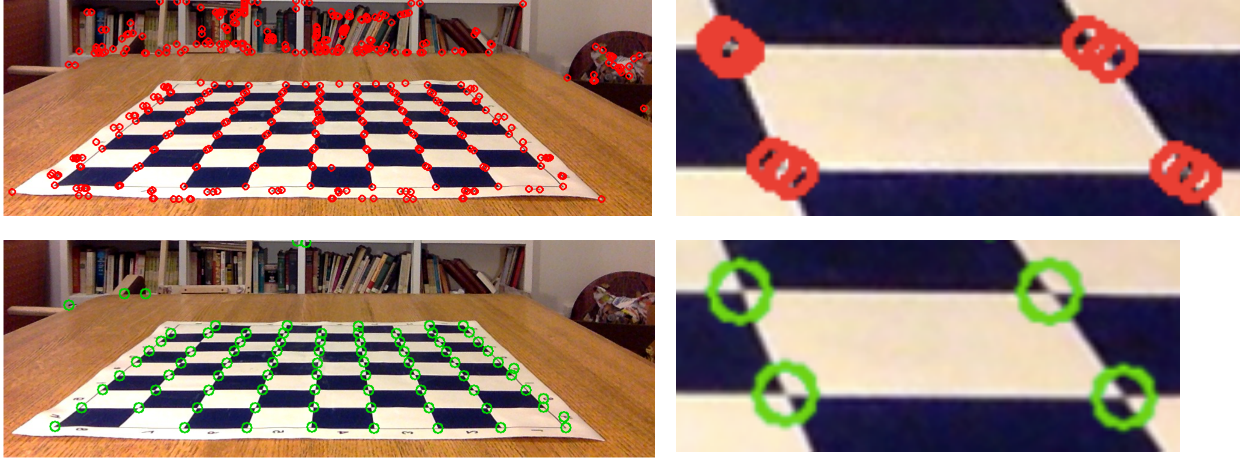

**Figure 1: On the top is an image with corners detected with a Harris Corner Detector overlaid. On the bottom is the same image, after the corners have been filtered with a SIFT Descriptor Classifier and clustered using mean-shift clustering.**

The results of a Harris corner detector to one of our preliminary images is shown in (fig. 1). Clearly, this has a high recall in finding chessboard corners, as every corner on the board is saliently marked. This provides a good population of candidates for more complicated classification procedures.

To alleviate this issue, we ran a SIFT Descriptor Classifier on the Harris Corners, namely a logistic regression classifier trained on roughly 2000 negative and 1000 positive examples. This both significantly reduced the number of false positives. In order to ensure that corners were not double counted, we applied a clustering procedure very similar to mean-shift clustering. However, it is worth noting that some corners are missing from the classifier, but indeed, the algorithm only needs a small set of corners with assigned board coordinates.

### *Assigning board indices to each corner*

To determine which board corners each of the corners returned by the SIFT Descriptor Classifier is, we fitted the corners to horizontal lines using a Hough Transform. In order to only capture lines that are actually lines on the board, we use a high threshold, at the expense of possibly missing lines.

We only consider points very close to both a vertical line and a horizontal line.

In order to assign each of these points board coordinates, we need to first determine which index (an integer in $\{1, 2, 3, \ldots, 8\}$) each vertical line and each horizontal line correspond to. We perform the following algorithms to determine these line correspondences, given four or five lines, either all vertical or all horizontal:

For vertical lines:

1. For each line, get its x-intercept with the bottom of the image of each line
2. Generate all possible line assignments that satisfy that if line m is to the left of line n, m < n
3. For each assignment, fit a linear regression on the (assignment, x-intercept) pair for each line and get the correlation coefficient
4. Return the assignment with the maximum correlation coefficient.

For horizontal lines

1. For each line, get the logarithm of the average height of the line above the bottom of the image
2. Generate all possible line assignments that satisfy that if line m is above line n, m < n
3. For each assignment, fit a linear regression on the (assignment, x-intercept) pair for each line and get the correlation coefficient
4. Return the assignment with the maximum correlation coefficient.

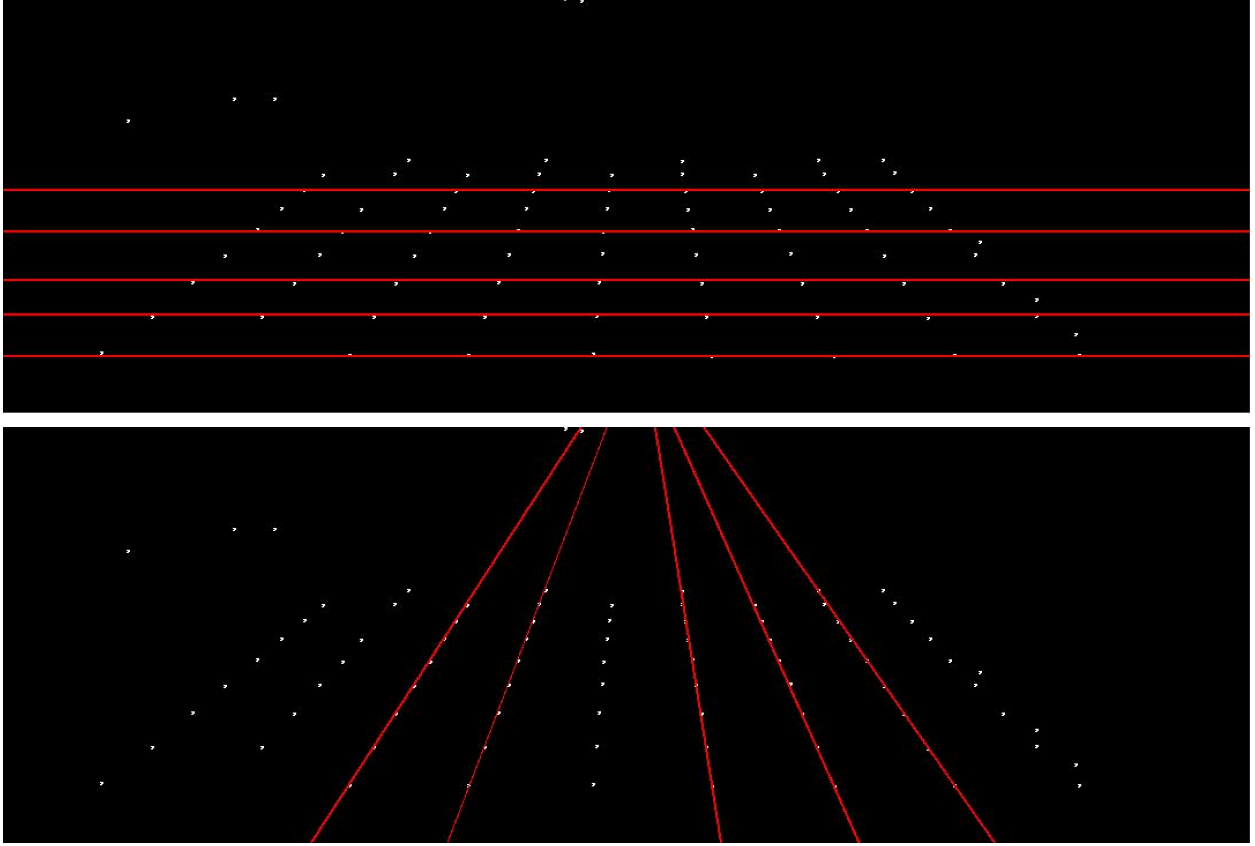

**Figure 2: On the top are the lines detected by a Hough Transform on the points given by the SIFT Descriptor Classifier, filtered to only horizontal lines. On the bottom are the lines, filtered to the vertical lines.**

We take advantage of the fact that the correct assignments will lead to an approximately linear sequence for vertical lines, and an approximately geometric sequence for horizontal lines.

Finally, we notice that this will only give us accurate results up to a shift, since results up to a shift will all give equal correlation coefficients. To determine which of the possible shifts is correct, we simply determine which of the possible shifts is correct by seeing which of the homography correctly matches the most Harris Corners (after filtering with the SIFT Descriptor Classifier) found earlier.

### *Computing the Board Image Homography*

Only a subset of board corners would be detected by the above algorithm. We are guaranteed, however, that the chessboard consists of an evenly-spaced grid lying on a plane. Hence, there is a projective transformation relating points on the board (in board coordinates) with points in the image, which can be computed from a small set of known point correspondences between the image and board. This homography allows one to find the image coordinates of any point on the board, therefore enabling one to select image regions corresponding to certain squares. Here we outline our approach to finding this homography.

Let $P_i$ , $P_i'$ be corresponding points in board/image coordinates, respectively, for $i \in \{1, \ldots, n\}$. Then $\exists H \in \mathbb{R}^{3\times3}$ such that $\forall i, P_i' = HP_i$.

$H$ can be determined by the following overdetermined system of equations $Ph = 0$ where $P$ is given by:

$$\begin{bmatrix} p_{1x} & p_{1y} & 1 & 0 & 0 & 0 & -u_1 p_{1x} & -u_1 p_{1y} & -u_1 \\ 0 & 0 & 0 & p_{1x} & p_{1y} & 1 & -v_1 p_{1x} & -v_1 p_{1y} & -v_1 \\ & \vdots & & & \vdots & & & \vdots & \\ p_{nx} & p_{ny} & 1 & 0 & 0 & 0 & -u_n p_{nx} & -u_n p_{ny} & -u_n \\ 0 & 0 & 0 & p_{nx} & p_{ny} & 1 & -v_n p_{nx} & -v_n p_{ny} & -v_n \end{bmatrix}$$

and $h$ is a columnized representation of $H$.

In order to solve this system of equations, we apply SVD to P and construct a matrix from the last row of the the third return matrix from SVD.

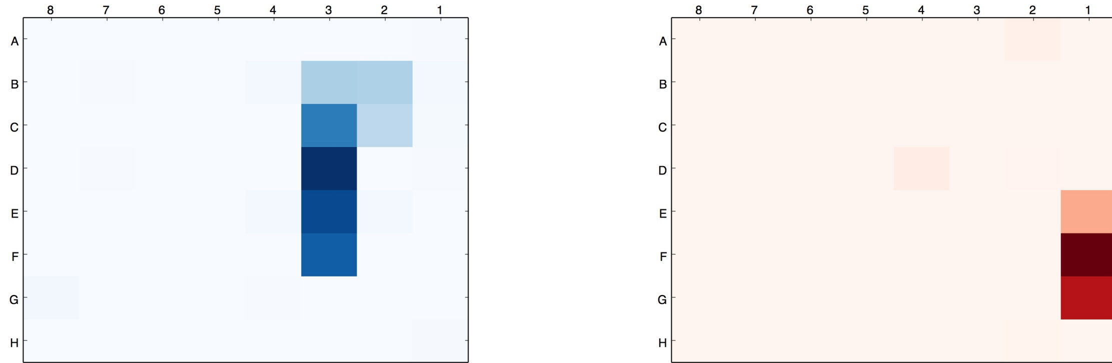


**Figure 3: Example heatmaps generated by the program. The left heatmap displays squares with a large increase in piece color and the right heatmap displays squares with a large decrease in piece color. The correct move, which the program finds correctly, is Nf3, or g1-f3.**

## 4.2 Live Gameplay: Determining Moves

Once the homography has been determined, the players set up the pieces and the game begins.

At the beginning of the game, when the pieces have just been set, the program takes a snapshot. Using the centroids from a k-means clustering algorithm with k = 4, we determine the four most common colors on the board. Those four colors will be:

1. The color of the white squares
2. The color of the black squares
3. The color of the white pieces
4. The color of the black pieces

In live gameplay, after each move is played, a new image is taken. The image of the board before the move was played (taken after the previous move was taken) and an image of the board after the move was played are compared as follows:

1. Match each pixel of each square area to one of the four categories colors generated via k-means clustering.
2. Find two heatmaps for increase and decrease of the color of the piece being moved in the actual moves in each square, and divide each of the heatmaps by the the sum of its values.
3. For each move
    a. Generate two expected heatmaps for increase and decrease of the color of the piece being moved in each square, and divide each of the heatmaps by the sum of its values.
    b. Find the cosine similarity between the expected and actual heatmaps
4. Return the move with the lowest cosine similarity

Several approaches were examined to generate expected heatmaps.

A naive solution that was initially implemented was to fill the expected heatmap for increase in piece color with zeros, except for the square into of which a piece was moved, in which the value is 1. Conversely, the expected heatmap for decrease in piece color would be filled with zeros, except for the square out of which a piece was moved, in which the value is 1.

This algorithm initially yielded promising results for the first few moves of a game, but generated incorrect predictions when the square that were most occluded by the introduction of a piece was not the square into which the piece was moved, but instead the square immediately behind or two squares behind it. This was a common occurrence, so we modified our algorithm to reflect that.

The solution that we selected was to fill the expected heatmap for increase in piece color with zeros, except for the square into of which a piece was moved and the three squares behind it, in which the values are 1. Conversely, the expected heatmap for decrease in piece color would be filled with zeros, except for the square out of which a piece was moved and the three squares behind it, in which the value is 1.

## 5. Experiments

### 5.1 Homography testing

We tested the program's ability to generate the correct homography, and found that while certain conditions affect the accuracy of the generated homography, under conditions that would be optimal for human chess, the homography is generated with good accuracy.

Most significantly, varying lighting conditions significantly, especially when there is severe glare on top of several of the points in a row or column will frequently lead to an incorrect homography.

However, in conditions when the image is taken by a Macbook Pro and visually has appropriate lighting, the program generates the correct homography in 21 out of 25 test cases that we ran. In conditions with poor lighting, significant obtrusive glare, or other similar issues, this accuracy was reduced to 7 out of 15 test cases.

### 5.2 Live game testing

We also tested the program in a number of complete games played to determine how many moves (single player moves) the program correctly predicts until it makes a mistake. (We used this metric instead of the total count of the number of incorrect moves in a game because after the first incorrect move, following moves are less likely to be correct, since the correct move may no longer be legal.)

| Game | Total moves | First incorrect move |
|---|---|---|
| 1 | 48 | No incorrect moves |
| 2 | 39 | 18 |
| 3 | 75 | 59 |
| 4 | 28 | No incorrect moves |
| 5 | 68 | 40 |

## 6. Applications

Our intention with this application was to provide an API on top of which other developers could create useful and insightful applications. Here we discuss potential future directions for development.

While our application provides one with the ability to receive a standard notation for any given chess game played on a standard board, we have not yet made efforts to integrate this with existing chess engines. We believe that the real-time application of existing analytical algorithms would offer a great deal of utility to players. In particular existing algorithms for determining a player's strengths and strategic characteristics, predicting future moves and assessing their performance improvements (e.g. assigning chess ratings, etc.) should be relatively easy to add to our system. Such algorithms have never been applied to chess on real boards using commercially available and common hardware, to our knowledge.

In addition, we anticipate that users would benefit from the availability of coaching resources in games on real boards. While several chess coaching utilities currently exist (including those available through *Stockfish*, an open source and very popular chess engine), current implementations require that one play online or in a much more controlled and resource-intensive apparatus.

Finally, we believe that one of the most exciting directions that *CVChess* will progress in the future is its integration into augmented reality systems. Due to the fact that it can offer precise coordinates for real-world phenomena, it could be used, with relatively little adaptation, as a supplement to a projector pointing at a board, annotating games in real time. While animations projected onto the board may become distracting, we maintain that simple, minimalist aids projected onto the board would enhance the game playing experience. This includes, but is not limited to, visual indicators for when pieces are in check/causing check, areas one can move into during check, lines of attack from specific pieces and more.

In conclusion, we have made the first step in developing a low-cost, robust platform on which

we hope many useful chess-related applications are developed in the future. We are open to collaboration; please contact the authors if you are interested in developing on top of CVChess.

# 7. References

[1] Martin Martin. Finding a Chessboard. 2009.

[2] Chua Huiyan, Le Vinh, Wong Lai Kuan. Chess Vision. 2007.

[3] Stuart Bennett and Joan Lasenby. ChESS – Quick and Robust Detection of Chess-board Features. 2012.