Extracting sticker colors on Rubik's Cube
CPSC525 Project Report

Andrej Karpathy


Abstract

This project tackles the problem of real-time object detection and localization. More specifically, we attempt to locate a Rubik's Cube in a video stream and extract the sticker colors with minimal help from the user. The edge based approach we take is invariant to various camera settings and does not assume a particular color scheme for the cube.

1 Introduction

The Rubik's cube is a well-known puzzle that has captured attention of millions of people around the world. There is currently a large community of people all around the world that is reviving the trend and finding better and faster ways of solving the puzzle. A common need that arises is that people want to share particular configurations of the cube with each other, or enter their particular configuration into the computer to get hints on the fastest way to proceed towards the solution, or the fastest way to complete some particular step of some method. However currently the only way to enter the state into a computer is by manually defining all sticker colors on every face, which is a tedious process that can take a long time. The Rubik's cube, however, is a fairly regular object with many symmetries and it should be possible to design an algorithm that can read the sticker colors off every face of the cube with next to minimal effort from the user.

The final product of this project is a program that addresses this problem, and is distributable to anyone over the Internet. In light of this goal we wish to make as few assumptions about the setup as possible. The cameras that people use can be of varying sizes, frame rates and settings for contrast, brightness, white balance, hue and saturation. The performance of the final algorithm must therefore be invariant to all these settings. In addition, the cube can also be located in any part of the image and at any scale. The lighting conditions can also make the task significantly harder because the stickers on a cube can be very reflective when turned towards the light. Finally, Rubik's cubes come in varying color schemes so we cannot simply look for the standard colors blue, green, yellow, white, red and orange.

Despite the great number of challenges that are to be overcome, the task is a little easier because we have some a priori knowledge about the shape of the Rubik's cube and we can attempt to look for very specific and unique features in the image. In addition, since we want the program to work in a large variety of situations and in real time, we will trade a little bit of user convenience to gain a lot in robustness. More specifically, we will require the user to show the program one face at a time and tap space to confirm that the program correctly found the face of the cube. This 5-second procedure that the user has to go through is well worth the benefit gained in robustness of the system. One alternative to scanning faces that was initially considered was to track the entire three dimensional cube in the image, but this was found to be more frustrating for the user in preliminary testing because the cube has to be moved very slowly so that we do not lose track of it. With this approach we can easily swing the cube around 90 degrees in split second and right away read in the next face. While it is much faster and more reliable to read the cube in one face at a time, it comes with one downside: it is impossible to reconstruct the entire 3D configuration of stickers only knowing the stickers on every face independently. We have to know how the faces are arranged with respect to each other on the cube, and we must require the user to enter the cube in a predefined sequence.

2 Related work

Our goal is to quickly and reliably locate the Rubik's cube grid in the image. This is a fairly generic instance of a problem that has been tackled many times before. However, there is no single best approach to this task and most often specific algorithms are designed for every particular instance. Despite the lack of a general framework, the computer vision toolkit has grown over the years and many feature extraction methods have been proposed to streamline this process.

One of the most standard feature extraction methods is template matching, where we simply look for a particular set of pixels within the image. This could work well for our problem because the Rubik's cube has a very regular shape: For example we could be looking for the little crosses in the middle of the cube. A significant

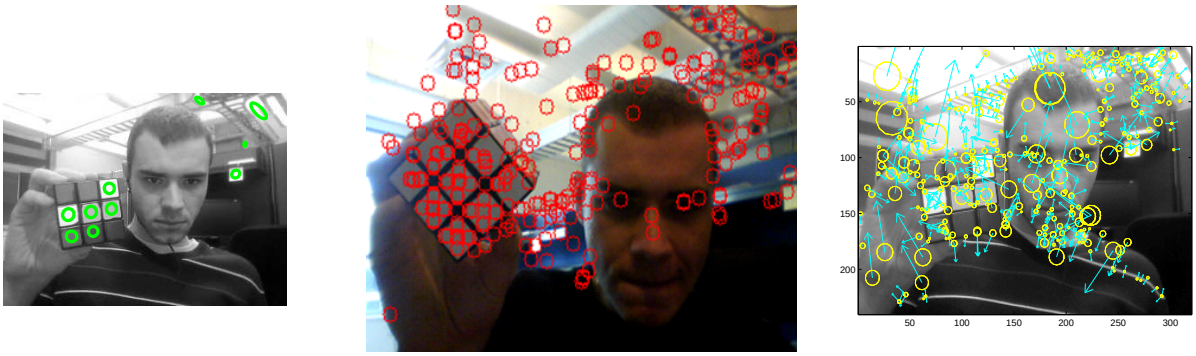


Figure 1: *Feature detectors tried.*

downside of this approach is that the cube can potentially be oriented in any direction and we would need several templates for every direction. Matching every little cross template to the entire image could potentially become a costly process. In addition, we would have to search for the templates in the Gaussian Pyramid because we wish to detect cubes at any scale.

An alternative approach with a long history is to consider edges in the image as the basic primitive for detection. In addition to extracting the edges from the image a common post-processing step is to use the Hough Transform to find edge line segments. This is a viable approach for our problem because the cube is very rich in long straight edges features. However, most scenes are fairly rich in strong edges so we are left with a problem of detecting our grid in a sea of outliers. A common approach for dealing with this problem is to use RANSAC because given a particular hypothesis it is fairly easy to compute its likelihood by simply iterating the rest of the edge segments and counting the number that align with the grid.

The last common approach is to extract interest points from the image. Many interest points detectors have been proposed in the literature, but here we only consider SIFT[2], SURF[3], and MSER[4]. One downside of this approach is that the cube is very regular and the same feature is likely to be detected several times. It can therefore become more difficult to establish a definite correspondence because we have to consider the relative arrangement of the detected features and deal with ambiguous interpretations. For example, detecting four stickers does not tell us enough about the position of the grid because we don't know which four stickers we have detected. And even if we somehow used their positions and calculated that they must all be neighbors in the grid, this still leaves four perfectly valid hypotheses because there is a total of 9 stickers in a 3x3 arrangement, and the 4 stickers can be in any of the four corners.

Once we have located the grid we would like to extract the sticker colors and match every sticker to its corresponding center. When dealing with color in images it is often recommended in the literature to abandon the RGB color space in favor of HSV. This is largely due to the fact that the RGB components are all correlated with the amount of light hitting the object, which complicates the the discrimination procedure that is critical to our task.

The last improvement that we consider making to our method is to improve the perceptual quality of the tracking. Once the grid is reliably located in the image, we would like to track it smoothly, instead of detecting from scratch every iteration and getting an unpleasant flickering detection. One popular tracking method is the Lucas-Kanade Optical Flow tracker[1] that works by computing the optical flow in the Gaussian pyramid and using the information from several scales to achieve more precise tracking.

3 Preliminary work

We first examine the feasibility of using various existing feature detectors for this problem. We found that while the SURF features detector consistently finds features on the stickers, the detection is unreliable when the cube is even slightly tilted. Similarly, the SIFT feature detector occasionally finds a feature right in the middle of a sticker, but very small tilts of the cube makes these features disappear. The MSER detector operates on regions of constant brightness suggesting that it should find features inside the stickers because these are fairly smooth regions of the image. To take it further, we can also look at the returned MSER regions, fit an ellipse to them, and then discard the regions that have a very high eccentricity, keeping only the more or less circular regions. MSER's disadvantages for this task are that the detection did not seem very robust (you can see on Figure 1 that not every sticker is consistently detected even after significant time spent fine tuning its parameters). MSER feature detector also took a fairly long time to run when compared with the other methods.

4 Implementation

To get the convenience of a high level programming language but also keep the application running in real time, the decision was made to implement the algorithm in Python and the OpenCV Library. OpenCV was originally implemented in C, but recently very good bindings for Python were made available on the official website.

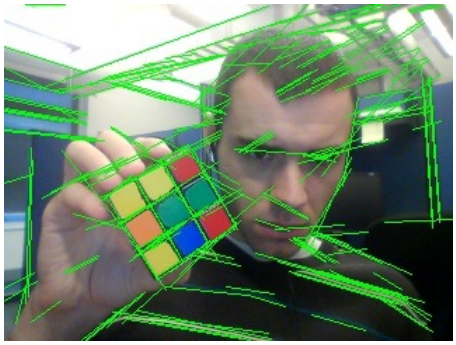
Sample frame captured from the camera.

The picture above shows a standard frame that comes from the camera in the beginning. To improve efficiency we resize the frame to 320x240 pixels. Standardizing the image size is a good idea not only for efficiency, but also because the few parameters that we must have in what follows are more likely to work.

After failing to find good interest point detectors for this task, the decision was made to continue with the edge-based approach. Intuition suggests that this approach should work well because strong parallel edges are a unique distinguishing feature of any Rubik's cube. The first step of the algorithm is to extract edges from the image, which is done by applying the Laplacian filter. We then apply the probabilistic Hough Transform to look for long edges. The hough transform takes a parameter that determines its sensitivity to gaps when extracting the line segments. In order to avoid hard thresholds that do not scale to novel camera setups, we keep track of the number of edge segments extracted from the image and tune this parameter slightly every iteration so that the number of edges extracted is always 50.

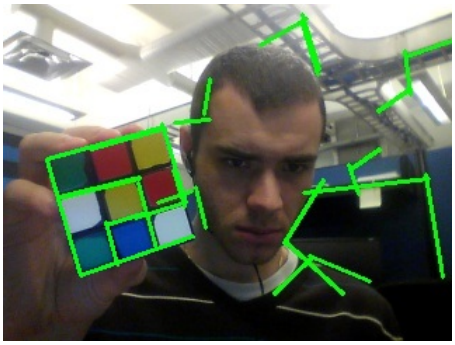
After Laplacian filter is applied

When the Rubik's cube is in the image, the algorithm will find many edges along the lines that separate the stickers. In practice these edges are so strong that the adaptive threshold on the hough transform will get very large and many background edges tend to get almost washed out in comparison.



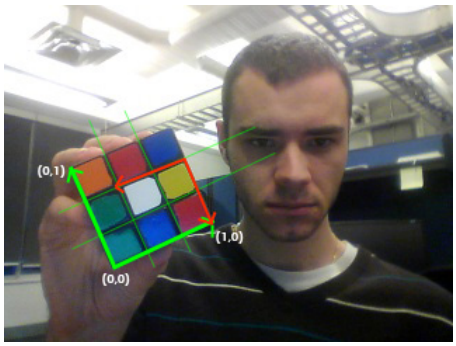
Edge segments after Hough Transform with adaptive threshold

The goal of the next step is to identify the 4 corners that surround the face of the face of the cube. Given the starting and ending position of all line segments we look for two lines that either share an end point or they intersect each other at one third or two thirds of their lengths. The former case corresponds to finding the outlining edges of the cube and the latter is attempting to find the line segments that form the inner part of the cube. Since we wish to find corners of the cube in this step, we have to do a little more vector math in the latter case and solve for the position of the closest corner to the intersection. The candidate pair must contain lines that are no more than 30% different in length or it is discarded. We also discard a pair of lines if the lower of the two angles between them is lower than 0.5 radians (or roughly 30 degrees), since this indicates that the lines are almost parallel.



The pairs of lines that are left over after the restrictions for lengths, angles and end points or intersections are enforced as described above.

Many of the parameters above are on the safe side and allow for several false positives to still come through. In the next step we iterate over every possible candidate corner and solve for the affine transformation that takes points from screen space to the coordinate system defined by the corner and the corresponding two lines. We then iterate over all line segments found by the original hough transform and project their end points to these local coordinates. Finally, we count the number of line segments for which both end points lie exactly on the grid. That is, in the local coordinate system, either the first or the second coordinate is exactly 1/3 or 2/3 within a margin of 0.05, and neither of the coordinates is outside of the range [-0.1,1.1]. If both points lie on the grid and the angle of the line segment is close to the angle of the axes, then this segment is counted as belonging to the grid. Note that we do not look for evidence along the outlines of the cube (corresponding to one of coordinates being 0 or 1). This is so because otherwise the algorithm can get fooled into finding a Rubik's cube of size 2x2 stickers inside the 3x3 Rubik's cube, since there is enough evidence on the outer lines. But if we only look for evidence on the inner lines, then these smaller grids will have very little evidence because their 1/3 and 2/3 lines pass almost directly through stickers.



Visualization of the coordinate system that we solve for. We look for evidence of other line segments along the lines 1/3 and 2/3 in each coordinate (displayed in soft green). An example of a false 2x2 grid that can sometimes be detected if we also look for evidence on the outer lines is depicted in red.

Sorting the above list of coordinate systems by the number of line segments that align with it we almost always find that the first few numbers are very high and almost equal, and then there are a few outliers that only align with at most 1 or 2 other line segments. This is so because these first few numbers usually correspond to the 4 corners of the cube. Finally we iterate over the candidates that are above the threshold of 5% of total number of edges and pick the one that best aligns with the detection from the previous iteration. Using the coordinate system we constructed above we can then find exactly the places of the stickers and display the entire grid to the user.

The outlined grid found in the image

We keep track of the number of times in a row in which we found the same grid from scratch. If this number is at least 3, then we can be very certain that this grid is not just an error and we initialize Lucas-Kanade Optical Flow tracking for four points which we place directly in the center of the 4 crosses found inside the grid (i.e. the four corners of the center sticker). This tracker is implemented in OpenCV and it is not only very fast but also very precise. Once the optical flow tracking is initialized, the occasionally flickering grid detection turns into a very smooth grid that follows the face of the cube effortlessly. We still keep checking every iteration that the tracking did not fail by continuously re-checking that the four points returned from the tracker satisfy certain constraints. These points form, to a very good approximation, a parallelogram. We therefore require that during the tracking, the opposite line segments that connect the four points are approximately equal in length every iteration. If this condition is not met then something likely went wrong with tracking and we revert back to detection mode.

When the user hits space, the colors of the stickers are read by averaging the color of a small neighborhood around the center of every sticker to avoid effects of noise, and the color configuration is saved and presented to the user. The user can also choose to redo any measurement by using the arrow keys and highlighting the face that should be re-read.

In the end we wish to exactly assign all stickers to their corresponding center and output the entire configuration. We know that there are exactly 9 stickers for every one of 6 possible colors and that the center pieces of every face store the color of that face (these stickers do not ever move on the cube). We proceed iteratively to associate every sticker to its center piece as follows. We keep finding the sticker that is closest in hue and saturation to one of the centers. We work in the HSV color space instead of RGB to gain some robustness toward lighting conditions. Every time we identify a sticker that can be classified to one of the centers with the highest probability, we fix its color. As a result of fixing one sticker we can easily rule out many other possibilities of colors of other stickers due to our prior knowledge. In particular, we know that there can be at most 9 stickers of any color and no edge piece contains the same sticker colors. Similarly no corner pieces (which have 3 stickers) can contain two stickers of the same color. We proceed to get even more elaborate: Opposing color stickers do not ever occur on the same piece (For example, on a regular cube there can be no green-blue edge because green and blue centers are on opposite sides and do not share an edge). The algorithm keeps track of the possibilities for every single sticker and every time we fix some sticker's color, we eliminate the appropriate possibilities from the neighbors.

5 Results

Using the program outlined above it is possible to read in the state of a Rubik's cube reliably in 5-10 seconds, significantly outperforming the existing manual methods. Please see the attached video for demonstration of the results.

The additional code that implements the constraints on the color of stickers was found to be extremely useful. In practice, anywhere between roughly 5-20 stickers (out of a total of 54) are forced to be of particular color in the end and do not have to be matched to any of the centers at all. Many stickers before that also only have to be matched to about 2 or 3 centers as opposed to the total of 6 possible centers. This naturally enhances the precision of the algorithm, because if we ever get a bad reading on one of the stickers, the sticker is not likely to get matched to its center right away, and will most likely end up being forced to its correct color in the end.

White color on the cube was found to cause a lot of problems because of its very low saturation. This causes the hue value to be very unreliable. What was found to work best in practice was to simply ignore stickers of low saturation until the very end. The stickers with high saturation were matched to their centers using the hue attribute first, and the unreliable stickers are left until the end. The hope is that by that time there will be forced to their correct colors by the constraints. This method also ends up working well when the lighting conditions are bad and the color gets washed out by the light on some of the stickers. These stickers will likely be ignored until the very end and then assigned much later, reducing our chances of identifying them incorrectly. In practice it was also found that using the RGB color space to match the stickers also worked well, with a naive approach of calculating the L2 norm of the residual between sticker color and the color of the centers

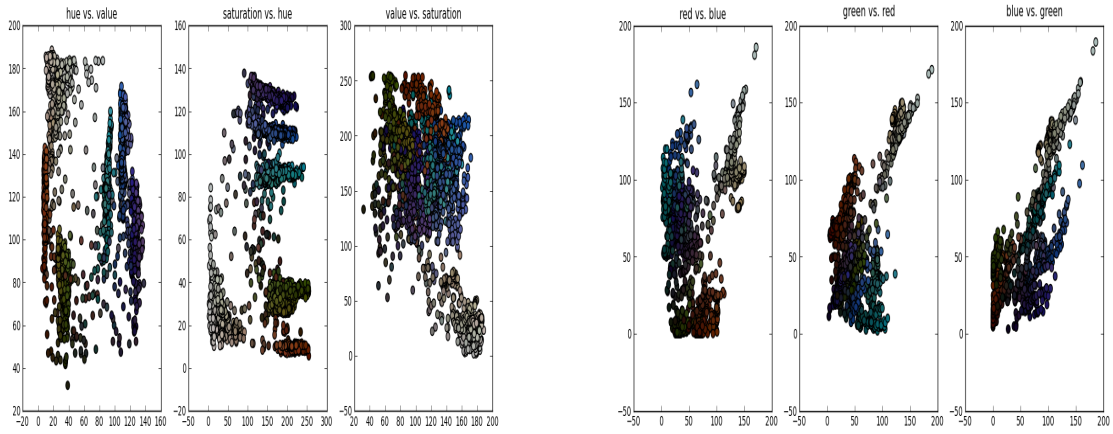


Figure 2: *Sample of observed colors of stickers in RGB and HSV color spaces in a trial run. The sticker colors are clearly more easy to separate in the HSV color space. Also note the 6 easily distinguishable clusters in the Hue-Saturation graph that correspond to every one of 6 colors of the cube.*

being considered. Similarly as done with HSV matching, in RGB matching we ignore colors with all components above 200 (which corresponds to white colors) until the end.

A few problems with the algorithm that are more tricky to solve also came up. Using an adaptive threshold is a good idea to gain in robustness for different camera settings, but it can also yield some difficulties when the surrounding environment contains many strong edges. These edges have a tendency to "steal" away the edge segments from the cube, making the detection much less likely. A possible solution to address this issue is to keep track of edges that are detected consistently every single iteration and subtract them. Unfortunately this is further complicated by the fact that the hough transform is a little noisy and does not always detect the same line segments. Alternatively we could attempt to subtract the background based on the level of activity in every region. Of course, the simplest solution to this problem is for the user to turn the camera in such a way that this is not much of an issue.

6 Conclusion

We developed an algorithm for quickly locating faces of a Rubik's cube in a real-time video stream, extracting the colors of the stickers, and fully reconstructing the entire state of the cube. The extracted state can then be easily copied and used in other pre-existing programs that deal with finding solutions to the puzzle, or simply displaying the state as a picture so that it can be shared with other people. The program vastly outperforms every existing method in speed of entry and makes minimal assumptions about the structure of the cube and the camera parameters.

Refereces

(1) Jean-Yves Bouguet, Pyramidal Implementation of the Lucas Kanade Feature Tracker, Intel Corporation, 2007
(2) D. Lowe, Distinctive image features from scale-invariant keypoints, International Journal of Computer Vision, vol. 2, no. 60, pp. 91110, 2004.
(3) H. Bay, T. Tuytelaars, and L. Van Gool, SURF: Speeded up robust features, in Proceedings of the European Conference on Computer Vision, pp. 404417, 2006.
(4) J. Matas, O. Chum, M. Urban, and T. Pajdla, Robust wide-baseline stereo from maximally stable extremal regions, in Proceedings of the British Machine Vision Conference, pp. 384393, 2002.