Riley Deal
Image Processing & Computer Vision
Final Project
4/29/2020

Final Project: Low-level Image Features

OVERVIEW

Sudoku puzzles are a popular game which can take a human a long time of arduous thinking to complete. On the other hand, computers can solve them almost instantly; however, the difficulty comes when trying to turn the sudoku puzzle into something the computer can understand.

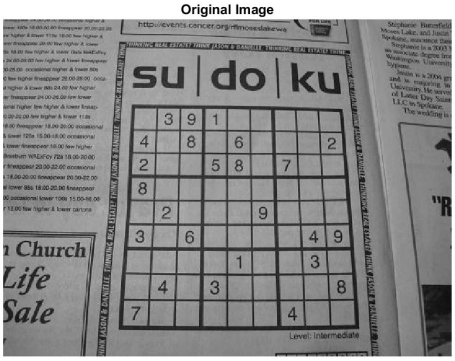
This project aims to solve this issue by taking an image of a sudoku puzzle and manipulating it into something the computer can solve by applying various image processing algorithms such as Canny edge detection, Hough transform, adaptive thresholding, projective transformation, connected component analysis, etc. Once the image is in a form which is solvable by the computer, the solution will be overlayed onto the image.

Discussion & Results

Each of the following sections will explain in detail a step towards the final goal of the project; a solved sudoku puzzle from an image. The images being used; shown in Figure 1, was retrieved from avidLearnerInProgress, but I have seen them in many examples so I am not sure where they originated [7]. All steps will use one of these images, but the image chosen between the two is dependent on how well they show the algorithm applied in the step. Most of the steps taken are based on a couple of tutorials; however, they use libraries to accomplish a solution. I have used ideas from these tutorials to create my own implementations of these libraries [1, 3, 4, 5, 6].

Original Image

Original Image



(Above) Figure 1 [Images/sudo-1_original.png] [Images/sudo-2_original.png]

The original sudoku images to be solved

Gaussian Averaging

The first step is to remove noise from the image via smoothing by applying a Gaussian filter. The Gaussian filter has two main arguments to change how it performs; a radius to determine how large the mask should be and a σ which determines the degree of smoothing. Figure 2 shows the image after being smoothed. In this case a radius of 5 and a σof 0.5 was used. This σvalue was used due to some of the sudoku lines being fairly faint. With a larger σ value these lines become lost which makes the future steps more difficult.

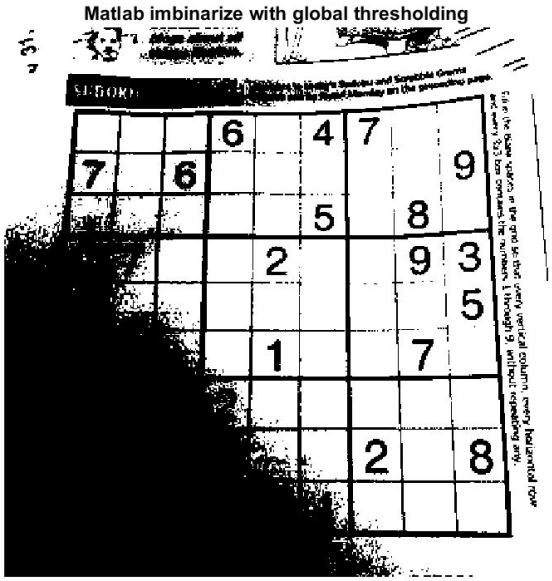
Gaussian blur; sigma = .5; radius=5

(Above) Figure 2 [Images/sudo-2_blurred.png]

Sudoku image with a slight Gaussian blur applied to remove noise

Adaptive Thresholding

The next step is to binarize the image. Right now the images are grayscale with every pixel having a value between 0 and 255. By binarizing the image we will be changing it to have only two values: 0 and 1. This means that somewhere in the range we currently have there needs to be a cutoff. One approach is called global thresholding, which is to take a value somewhere in the middle or perhaps a mean across the entire image. Anything above this value is 1 and anything below it is 0, but there are some issues with this. Figure 3 shows an image which has some regions which are inherently darker than others . This causes global thresholding to block out large regions which would be useful to have.



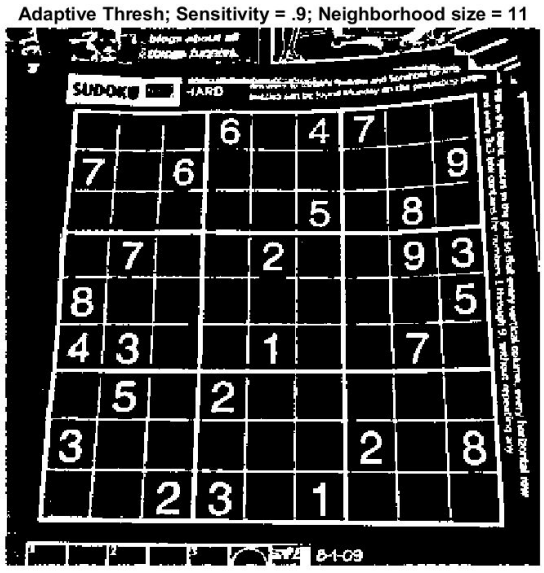
(Above) Figure 3 [Images/sudo-1_matlabglobal.png]

Sudoku image with some inherently dark regions where global thresholding struggles

The method I used to solve this issue is called adaptive thresholding. The basic idea is to divide the image into regions called 'neighborhoods'. The mean of each neighborhood is calculated and used for its respective neighborhood only. Anything above the mean becomes a 1 and anything below it becomes a 0 [8]. This solves the global thresholding issue because now the dark regions will use a mean relative to them. My adaptive thresholding implementation has two parameters: neighborhood size[1] (K) and sensitivity. Neighborhood size is used to divide the image into square neighborhoods of K-by-K dimensions. Sensitivity[2] is used to vary how the values are split in

[1] Neighborhood size can vary between 1 and the size of the image. With a neighborhood of 1 the image would become all black or all white. With a size equal to the image it would simply be global thresholding
[2] Sensitivity can vary between 0 and 1

relation to the mean For example, a sensitivity of 0.9 means that anything above 0.9*mean would become a 1. Figure 4 shows the same image as Figure 3, but with adaptive thresholding applied using a neighborhood size of 11 and sensitivity of 0.9.



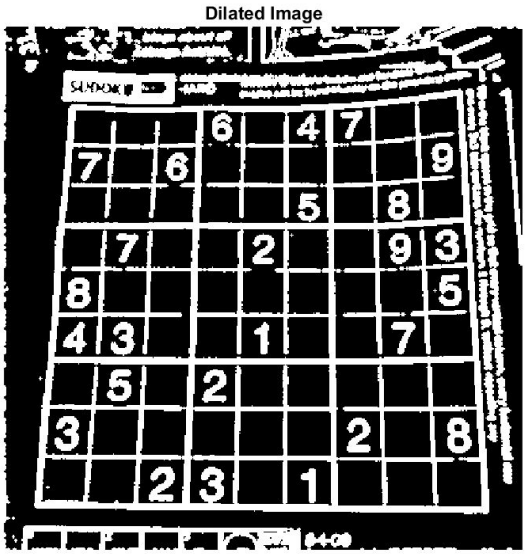
(Above) Figure 4 [Images/sudo-1_adaptive_thresh.png]

Sudoku image with adaptive thresholding applied in order to remedy the dark region issues

Dilation

As stated before, many of the lines within the sudoku boxes are fairly faint. Image dilation is fairly simple, first determine which mask will be used and then superimpose the mask over every pixel which has a value of 1 in the original image [9]. In this case, a 3x3 diamond/cross mask pattern[3] was used in order to stretch the lines both vertically and horizontally. Figure 5 shows the image after being dilated.

[3] This mask looks like: [0, 1, 0; 1, 1, 1; 0, 1, 0]

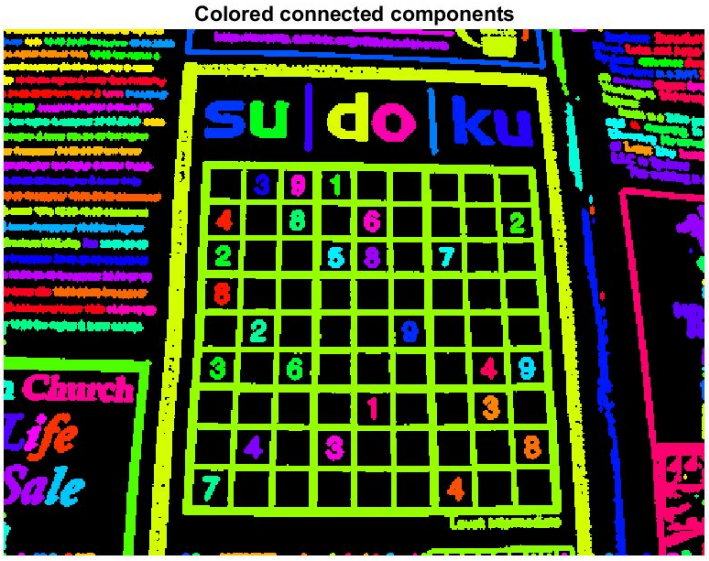


(Above) Figure 5 [Images/sudo-1_dilated.png]

Sudoku image with a cross dilation mask applied to stretch out fine lines

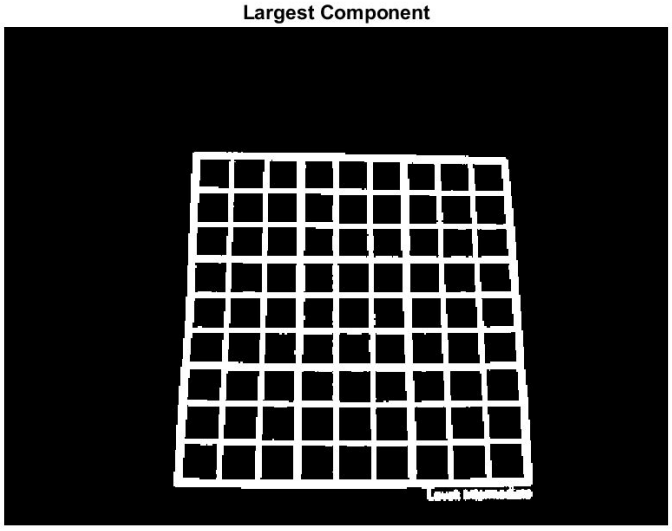
Connected Component Analysis

At this point, the goal was to extract the main sudoku grid. There are several methods I have seen for doing this such as finding the largest rectangle, finding the contour lines which give the greatest area, etc. The approach I decided to use was using connected components to determine what the largest blob is in the image. This method uses pseudocode adapted from the lecture slides to find and label all connected components. First set all values of 1 to -1 to indicate they need to be labeled. Next, go over the entire image looking for values of -1. When found, the program will recursively visit any surrounding neighbors which also have a value of -1 until none are remaining. The gives each connected component its own label from 1 to K. Figure 6 shows what the components look like when colored for visibility. Now that all connected components are labeled, the idea is that the largest one is usually going to be the sudoku grid. We can simply remove any component which is not the largest, giving us Figure 7.



(Above) Figure 6 [Images/sudo-2_coloredcomponents.png]

Sudoku image post connected component analysis with each label colored



(Above) Figure 7 [Images/sudo-2_largestcomponent.png]

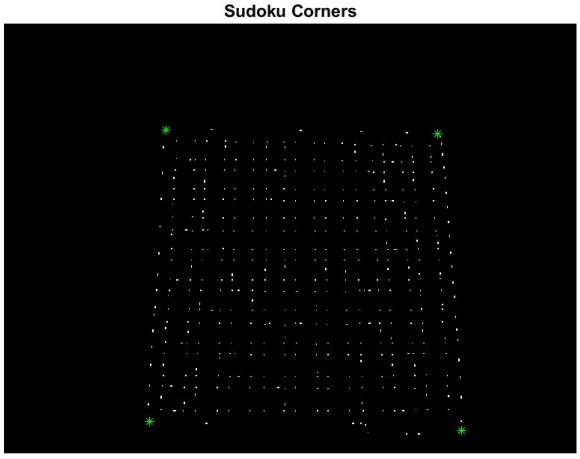
Sudoku image with only the largest component remaining

Projective Transformation

With the grid singled out from the rest of the image, we are almost in a solvable state. However, most images are very likely to be distorted in some way such as rotations, indirect camera angles, etc. In order to solve this issue the easiest approach I have seen is to apply a projective transformation. The only requirement for this is to find four known points in the original image and four points where we would like them to be projective for the final image. Ideally, we would like our four points to be the corners of the sudoku puzzle so we can map them to the corners of the image.

To accomplish this, we can find all potential corners by using the Harris-Stephens corner detection algorithm to get our candidate corners. This implementation of Harris-Stephens uses an upfront Gaussian filter with $\sigma = 0.5$. Next, a Sobel mask is used to compute gradients in both the x and y directions. These gradients are squared and another Gaussian mask is applied with $\sigma = 1$. These can now be used to calculate the Harris response value of every pixel; which determines if the pixel is an edge, corner, or nothing. An alpha value of 0.04[4] was used in this step. Finally, local suppression is used within a 3x3 grid to allow only values above .01*maximum response value and avoid clustering of many points in one location.

After finding all candidate points with Harris-Stephens we can find the points closest to the corners of the image, which are the corners of the sudoku. Figure 8 shows all candidate corners as well as the corners determined to be the corners of the puzzle.

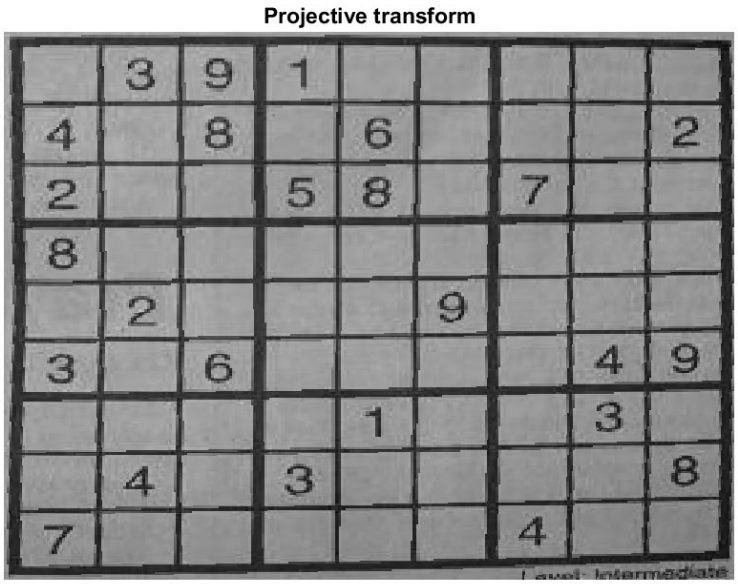


(Above) Figure 8 [Images/sudo-2_corners.png]

All candidate corners of the sudoku grid, with the true corners marked as green stars

[4] Alpha can vary, but the standard value is a constant between [0.04, 0.06] [11]

Once these have been determined the Homography matrix ‘H’ can be calculated. With this any point in the original image can be converted to a point in the new image and visa-versa[5] [10]. As shown in Figure 9, this transformation gives a much better view of the puzzle than we had before.



(Above) Figure 9 [Images/sudo-2_transform.png]

The sudoku grid after the projective transform has been applied

Line detection

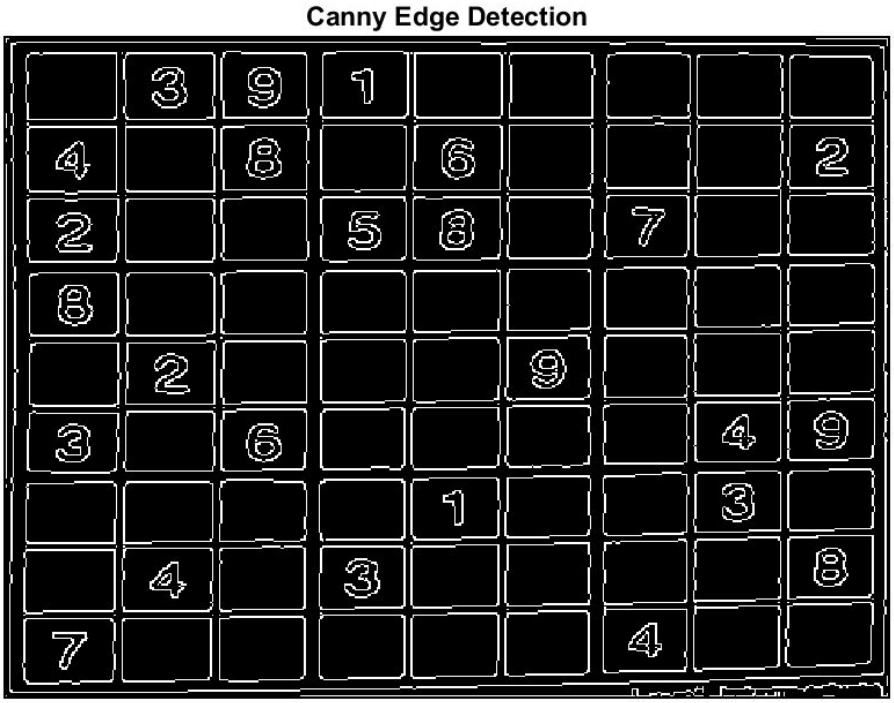
The next goal was to isolate each box in the puzzle in order to retrieve the numbers. My first approach to this was to use the Hough transform due to it often being used to find where straight lines exist in an image. For this, the Canny edge detector must be applied first. The Canny edge detector finds the edges within a picture by using the change in gradient.

My implementation of the Canny edge detector applies an upfront Gaussian mask with $\sigma = 1$ to help smooth out noise. From here the 2D edge detection mask I applied was again the Sobel mask, followed by a Gaussian mask on the gradients with $\sigma = 1$[6]. Non-maximum suppression was performed by calculating the direction of the gradient and then checking the pixels in the position

[5] Due to homography being invertible: $X' = HX$ and $X = H^{-1}X'$

[6] Gaussian masks are usually in the range of [1,2] for this operation with a value of 1.4 being common [12]

and negative direction of the gradient to determine if the pixel is a local maximum. My double threshold values were calculated using the mean. From here the strong threshold was calculated as .8*mean and the weak threshold was .26*mean[7]. I implemented a stack based hysteresis algorithm by placing all strong edges on a stack. For each strong edge found I checked if a surrounding pixel was a weak edge and if so, changed it to be a strong edge and added it to the stack. The results of the Canny edge detector can be seen in Figure 10.



(Above) Figure 10 [Images/sudo-2_canny.png]

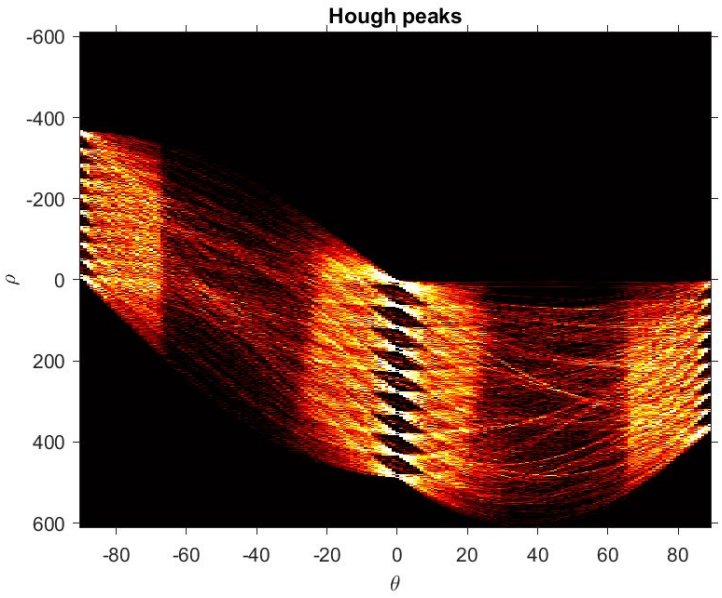
The sudoku grid after Canny edge detection has been applied

Once edge detection is complete, the Hough transform can be applied. The Hough algorithm takes each edge pixel and calculates possible lines which could intersect with the given edge point[8]. This is done by calculating the ρ for each θ in the range[9]. Each of these (ρ, θ) pairs can be

[7] These threshold values are often highly context dependent, but many implementations can achieve good results using the median or mean multiplied by a constant with a ratio between 3:1 and 2:1 for the strong to weak threshold [15].

[8] The general line equation $y = mx + b$ does not work in this case due to the inability to graph vertical lines; so the Hesse normal form is used instead: $r = x\cos\theta + y\sin\theta$. Lines can be given by the pair: (ρ, θ).

used to describe a line. By calculating all (ρ, θ) pairs for a given edge point, a unique sinusoidal curve is created since only that set of lines equations could possibly all pass through this point. These potential lines are used to "vote" on which lines may be possible in an image. If points are collinear then they will generate some of the same (ρ, θ) pairs which will have a higher amount of votes than points which are not. The lines with high amounts of votes will be visible as bright intersections within the graph of the Hough transform as shown in Figure 11.



(Above) Figure 11 [Images/sudo-2_hough.png]

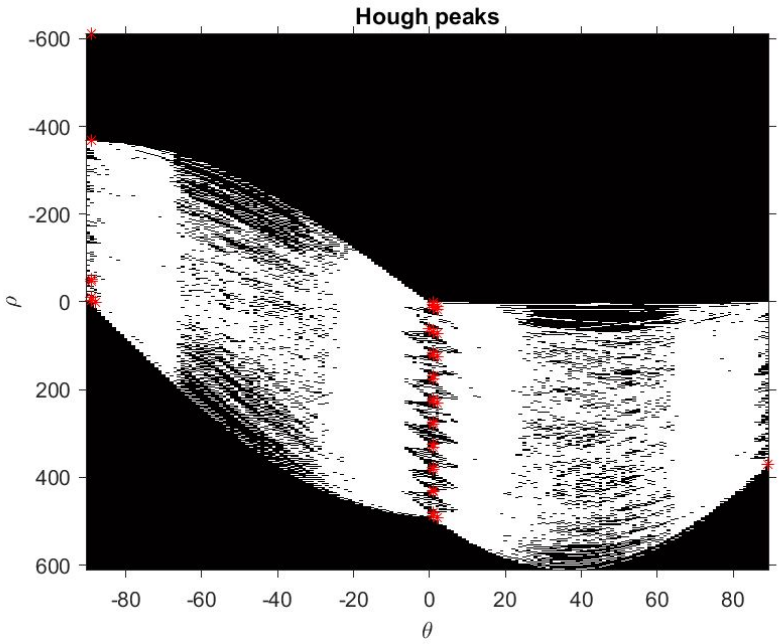
Clean Hough transform created by limiting the range of θ to +-22.5° of the gradient direction

My goal was to make my implementations of the Hough transform work as close to the Matlab version as possible so I could compare them easily. Matlab provides the equations they use to calculate the ρ, distance, diagonal, etc. so these were used to ensure the same values are used [16]. My implementation utilizes the gradient direction found in the Canny algorithm, which can be used to limit the range of θ used in calculating the potential lines. By doing so, we are only calculating the lines most likely to have collinear points in the image. This lowers the amount of computed lines by 75%, but keeps the lines most likely to pass thresholding.

Once the Hough transform is complete a common next step is to the most likely candidates to be lines by only considering lines which are above a certain threshold of votes. The algorithm i used

[9] The resolution of both ρ and θ can be varied for a tradeoff in accuracy for speed. The default ρ is 1, meaning 1 sample per pixel with a fixed range of two times the diagonal distance of the image. The default θ is 1° with a range of 180°; such as [-90, 90] or [0, 180].

for this was to determine all points above the maximum number of votes * 0.3 to be true lines. Using relation to the maximum votes is common, but the ratio is variable based on the context. Only voted lines above this threshold which are also local maximum of a 3x3 section can be considered true lines. My implementation then returns the K best (ρ, θ) peaks, as shown in Figure 12.

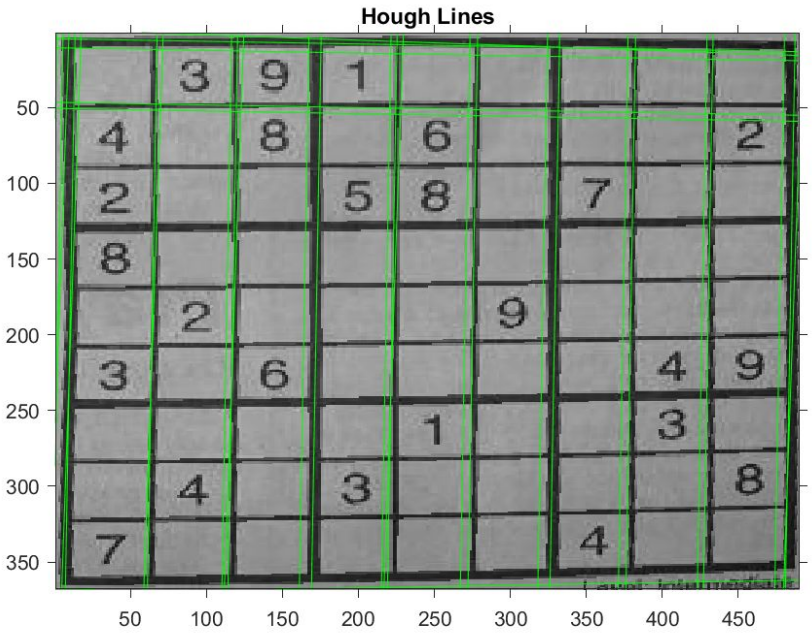


(Above) Figure 12 [Images/sudo-2_houghpeaks.png]

The top 50 Hough peaks above a threshold of 0.3*max # of votes determined by my Hough peak algorithm

These candidates can be used in a step referred to as “back-projection”, “reverse Hough transform”, or “de-Houghing” [17]. The basic idea is to go from the (ρ, θ) coordinates back to the x-y coordinate system[10]. In my solution I solved for two points using the min and max X or Y values; depending on the value of θ[11]. These two points can then be used to draw a line across the image showing the candidate line voted by the Hough transform as shown in Figure 13. Notably, most of the vertical lines are found. However, horizontal lines are still having issues with detection. I think this is partially due to some issues with my Hough implementation from project 3 and that some additional work such as removing duplicate lines and adjusting line equations such as via least squares method could also improve this solution.

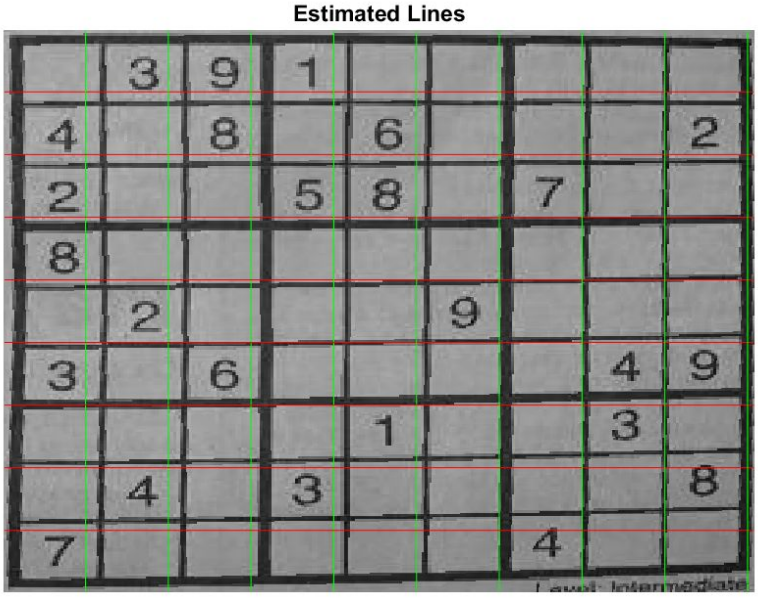
[10] This can be done by reversing the Hesse normal form: y= -cos(θ)/sin(θ)*x + ρ/sin(θ) and x = -sin(θ)/cos(θ)*y + ρ/cos(θ).
[11] Solve for y when θ is closer to the x-axis and solve for x when θ is closer to the y-axis.



(Above) Figure 13 [Images/sudo-2_houghlines.png]

Hough results from my algorithm with a threshold of .3*max, plotted with my line drawing algorithm

At this point, I did not have time to further explore line detection via the Hough transform. I did notice that in most scenarios the projective transform had done a great job with lining up the sudoku grid. What this meant is that it might be possible to simple guess where the lines should be and go from there. This was accomplished by simply drawing 9 horizontal and 9 vertical lines based on the image size. The results of this are quite accurate as shown in Figure 14.



(Above) Figure 14 [Images/sudo-2_estimatedlines.png]

Line results from estimating 9 horizontal and 9 vertical lines in the image

Number Recognition

With the sudoku grid squares separated out by lines, we can finally extract them. The results of this extraction can be seen in Figure 15. From here the goal was to use deep learning to train a network for identifying these images. Unfortunately, this is the one place in the project where I was unable to come up with a solution. My approach was using the Matlab deep learning tutorial which trains on the MNIST number dataset [14].

(Above) Figure 15 [Images/box_8.png] [Images/box_9.png]

Results from extracting the grid squares from a sudoku puzzle.

Despite having a test accuracy of over 98%, this network performed very poorly on the extracted sudoku dataset. This could be due to many things such as the differences in number angles, fonts, sizes, and most importantly the gridlines present in the image. One idea to solve this was to use the connected component analysis from the previous steps to find the largest components in the cell, assuming it would be the number. However, it was often the gridlines surrounding the number since the cells were mainly guessed as shown in Figure 16. This could perform better by taking the largest component near the center of the image. This means that this step could not be fully completed.

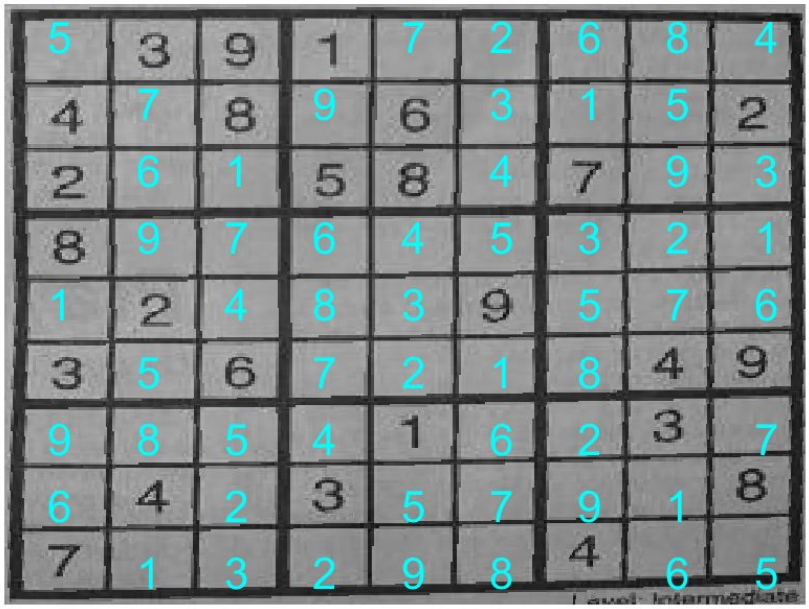
(Above) Figure 16 [Images/box_8_success.png] [Images/box_9_lost.png]

Results from applying connected component analysis on the grid squares from Figure 15

Solving the Sudoku

Despite the previous step being unfinished, the final step can still be created and tested. In order to accomplish this a few sudoku puzzles were hard coded into Matlab as 9x9 matrices. These were then solved recursively using code adapted from [13]. Now that the solution is known and we have a rough idea of where each grid square is in the sudoku we can overlay the solution on the image as shown in Figure 17. The idea from this was obtained from [2].

Solved Sudoku



(Above) Figure 17 [Images/box_9_solved.png]

The solution to the sudoku overlayed on the image using the best guess of grid square locations

Conclusion

Despite still having some issues with the Hough transform and being unable to accomplish the number recognition via deep learning I believe this was a great final project to choose as it reinforced what we had done in the previous projects while also requiring me to learn several new image processing techniques which we had touched on in class. I honestly did not think I would be able to accomplish as much as I did in this project due to how many image processing techniques were needed that I never knew at the start of the semester. However, after putting in the effort and accomplishing what I believe to be an awesome final project my confidence and knowledge in image processing is significantly higher than it was at the beginning of this class.

References

[1] Patel, N. (2017, Oct 20). *Solving Sudoku: Part II*. medium.com. https://medium.com/@neshpatel/solving-sudoku-part-ii-9a7019d196a2

[2] geaxgx1. (2017, Aug 30). Augmented Reality Sudoku solver : OpenCV, Keras [Video]. YouTube.com. https://www.youtube.com/watch?v=QR66rMS_ZfA

[3] Pingel, J. (2018, Nov 15). *Sudoku Solver: Image Processing and Deep Learning*. blogs.mathworks.com. https://blogs.mathworks.com/deep-learning/2018/11/15/sudoku-solver-image-processing-and-deep-learning/

[4] Sinha, U. *SuDoKu Grabber in OpenCV*. aishack.in. Retrieved Feb 28, 2020, from https://aishack.in/tutorials/sudoku-grabber-opencv-plot/

[5] Emara, T. (2018, May 19) *Real-time Sudoku Solver*. emaraic.com. http://emaraic.com/blog/realtime-sudoku-solver

[6] Banko, B. (2011, Aug 8). *Realtime Webcam Sudoku Solver*. codeproject.com. https://www.codeproject.com/Articles/238114/Realtime-Webcam-Sudoku-Solver

[7] avidLearnerInProgress, (2018, Mar 19). *Sudoku Solver*. GitHub, https://github.com/avidLearnerInProgress/sudoku-solver-openCV-python

[8] Mathworks. *adaptthresh.* Mathworks.com https://www.mathworks.com/help/images/ref/adaptthresh.html

[9] Wikipedia contributors. (2019, February 28). Dilation (morphology). In *Wikipedia, The Free Encyclopedia*. Retrieved 02:45, May 1, 2020, from https://en.wikipedia.org/w/index.php?title=Dilation_(morphology)&oldid=885533893

[10] Massachussets Institute of Technology. (2014, Oct. 20). *Perspective Transform Estimation.* wp.optics.arizona.edu. https://wp.optics.arizona.edu/visualopticslab/wp-content/uploads/sites/52/2016/08/Lectures6_7.pdf

[11] Wikipedia contributors. (2020, April 21). Harris Corner Detector. In *Wikipedia, The Free Encyclopedia*. Retrieved 02:46, May 1, 2020, from https://en.wikipedia.org/w/index.php?title=Harris_Corner_Detector&oldid=952289460

[12] Wikipedia contributors. (2020, April 14). Canny edge detector. In *Wikipedia, The Free Encyclopedia*. Retrieved02:46, May 1, 2020, from https://en.wikipedia.org/w/index.php?title=Canny_edge_detector&oldid=950845342

[13] Computerphile. (Feb 12, 2020). Python Sudoku Solver - Computerphile. Youtube.com. https://www.youtube.com/watch?v=G_UYXzGuqvM

[14] Mathworks. *Create Simple Deep Learning Network for Classification.* Mathworks.com https://www.mathworks.com/help/deeplearning/ug/create-simple-deep-learning-network-for-classification.html

[15] OpenCV. *Canny Edge Detector.* docs.opencv.org https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html

[16] Mathworks. *Hough.* Mathworks.com https://www.mathworks.com/help/images/ref/hough.html

[17] Fischer, R., Perkins, S., Walker, A., Wolfart, E. (2003). *Hough Transform*. homepages.inf.ed.ac.uk. https://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm