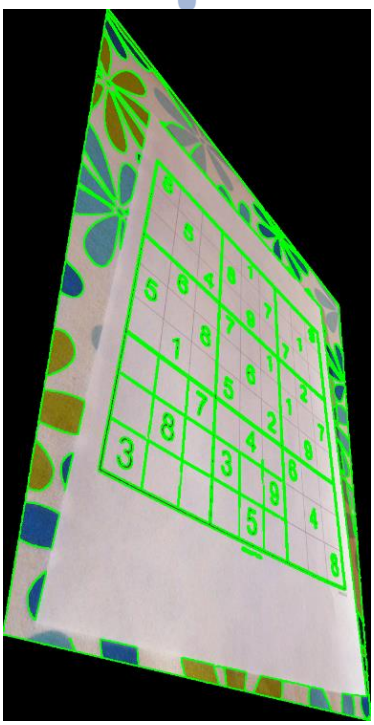
SODUKU IMAGE SOLVER

Project Report



FEBRUARY 13, 2023

ROEY LANGE – 318568383; BARAK DANIEL – 315571331

Introduction

This project aims to implement a comprehensive solution for parsing and solving a Sudoku puzzle from an image. The following steps outline the flow of our program (further explanation in the following sections):

- Image Processing: the first step in our pipeline is to process the input image (.jpg,.png) and convert it to a greyscale representation. The purpose of this step is to simplify the image. By using Gaussian blur, we reduce the amount of noise and other distractions that could interfere with the subsequent steps in our pipeline, a very important stage in the flow of our program (as we've learned in class).
- Edge Detection: Once we have the grayscale image, the next step is to detect the edges. To achieve this, we use Canny's edge detector algorithm, which finds a group of points that should contain the edges of the image (as learned in class).
- The next step is to detect the contours in the image. Contours are defined as the **boundaries of an object** in an image. In our case, we are interested in detecting the contours of the grid in the image (the frame of the sudoku puzzle).
- External Frame Detection: Once we have detected the contours, the next step is to find the external frame of the board (i.e. the grid). We use the approxPolyDP function from the cv2 library to approximate the **polygon that surrounds the contours**.
- Perspective Transformation: after we have found the external frame, the next step is to use Perspective Transformation to set the photo on a 450x450 pixel frame. The getPerspectiveTransform function from the cv2 library is used to perform the perspective transformation.
- Image Rotation: If needed, we rotate the image so that all the numbers are aligned with the X-Y default axis.
- Image Splitting: Split the image into cells, where each cell is 50x50 pixels.
- OCR Model: we use an OCR (Optical Character Recognition) model to predict the numbers in each cell.
- External Code for Solving the Sudoku: finally, we use an external code to solve the sudoku: https://github.com/Lakshmi-1212/Sudoku_Solver_LP/blob/main/Solver_LP.ipynb

Main External Functions Used in the Project

- Gaussian blur: a filter commonly used in image processing to reduce noise and image detail. As we learned in class, The idea behind the Gaussian blur is to apply a convolution operation to the image using a Gaussian kernel, which is a matrix of weights defined by a Gaussian function. The Gaussian function defines the shape of the kernel and determines the extent to which each pixel in the image contributes to the final result of the filter. The Gaussian blur works by first applying the kernel to each pixel (repetitive) in the image and normalizing its value by multiplying the original value. The values of the kernel are determined by the Gaussian function and are

typically normalized so that **the sum of the values in the kernel is equal to 1**. This normalization ensures that the overall intensity of the image is preserved after the blur operation. This is, in fact, a low-pass filter, since high-frequency information typically corresponds to fine details and noise in the image, while low-frequency information corresponds to larger structures and edges, so, by removing the high-frequency information, we can reduce the noise and improving the image's quality. In our project, the Gaussian blur makes it easier for subsequent processing steps, such as edge detection and identifying the contours of the grid.

- Canny's edge detection: a popular edge detector that we used plenty of times in our assignments. After the blurring mentioned in the previous step, this algorithm first and foremost uses gradient calculation, to point out where it is most likely that we have an edge. This is because, an edge is a line in an image where there is a significant change, and thus a gradient, the direction of the biggest change in each pixel, can guide us toward the edge lines. This is done by finding the derivative of the image's intensity, both on the x and y dimensions. Secondly, we use non-maxima suppression, which is (as the name suggests) suppressing the pixels which have a value lower than their neighbors, narrowing the possibilities of the points we suspect as edges. The final step is using two thresholds, t1, and t2, accordingly, where t1>t2. At first, we go over each pixel that wasn't yet suppressed in the earlier steps and zero the values of all the pixels whose values are smaller than t1. On the second iteration, we go over all the pixels that were left not zeroed (meaning all the pixels whose values are bigger than t1) and iterate over their neighbors. For each neighbor, we check if its original value was bigger than t2. If so, we "light" it back up to its original value. The pixels whose values are bigger than t2 but lower than t1 are called "weak edges", and, as mentioned above, are only kept if they're connected to a "strong" edge point in some way.
- Contour detection: a technique used in CV (specifically image processing) to detect an object's boundaries. In our code, we used it to identify the outline of the sudoku grid in the image. Contours can be defined as the boundaries of an object in an image that can be **represented by a continuous curve**. In other words, it is the set of points that make up the boundary of an object. In the OpenCV library, the **cv2.findContours** function is used to detect contours in an image. The principal concept of this function is taking a black-and-white/grayscale image and identifying the connected regions (represented as contours in the output). In this case, it enables us to find the sudoku grid by comparing the shape properties such as the aspect ratio, area, and perimeter. We use cv2.RETR_TREE to organize the returned contours into a 2-level hierarchy. This can be useful in our case, where we have objects (sudoku cells) nested inside other objects (the bigger grid, also known as the outer frame). We also used the approxPolyDP function to approximate the polygonal curves in the image, meaning we filtered and simplified the contours before converting them into straight lines, which is done in the next function.
- Perspective transformation: a technique in CV to transform an image into a different view. This involves mapping out the "world points" of the image, and where they would appear on the desired surface. In our project, the sole usage of the transformation is to extract and present the image on a 450x450 pixel surface, so that the final result is that each of the corners is located at one of the following: (0,0), (0,450), (450,0), (450,450). This will later help us crop the image to 9x9=81 cells, by knowing that each cell with the (i,j) out of the (0,0) to (8,8) possibilities starts at (i*50,j*50) and ends at ((i+1)*50, (j+1)*50)) pixels exactly. This is done by cv2.getPerspectiveTransform(), which calculates the transformation matrix that maps the four

points in the original image to the corresponding points in the new image. This transformation matrix is then used to warp the original image using the cv2.warpPerspective function.

- Image rotation: a process of transforming an image by rotating it **about its center**. In our implementation, image rotation is used to align the numbers in the grid with the X-Y default axis. As in the previous functions, we've made sure that the grid is a "flat" (no "depth" - Z axis dimension) 450x450 pixel map, all that is left to make sure of is that the grid is aligned in a 90-degree angle (this can result in one of the following: 0, 90, 180 or 270 degrees. To perform this rotation, the program first finds the center of the image. Next, the program calculates the orientation of the grid by finding the angle between the horizontal axis and the longest edge of the grid. This angle represents the **degree of rotation required** to align the grid with the X-Y default axis. Finally, the program performs the rotation by applying an affine transformation - a linear mapping between two spaces that preserves collinearity and ratios of distances.
- Tesseract OCR (**O**ptical **C**haracter **R**ecognition) text recognition: a free and open-source software library that can be used to extract text from images. In our project, we used tesseract to perform image rotation in cases where manual rotation was not enough to align the numbers in the grid with the X-Y axis. By analyzing the text lines in the image, it can determine the dominant direction of the text, which is then used to perform the rotation. This can be particularly useful for images that are skewed at an angle, making manual rotation difficult or impossible.
- Digit prediction using OCR model prediction: a pre-trained model (much like the tesseract), which its purpose is to predict the number/digit inside each grid image. The program uses the tf.keras.models.load_model to load the model we found online.
  a note on OCR models: mainly use CV techniques to identify the characters in an image. This process involves locating and segmenting the text regions in the image and recognizing the characters themselves. Trainable OCR models are typically based on machine learning algorithms, such as deep neural networks, which are trained on large datasets of images and corresponding text.
- Sudoku Solver- this is, as we've stated in the code, not our code, but rather a fast and aesthetically pleasing sudoku solver we've found online, with a link to the source of this code.

Functions Implemented in the Project

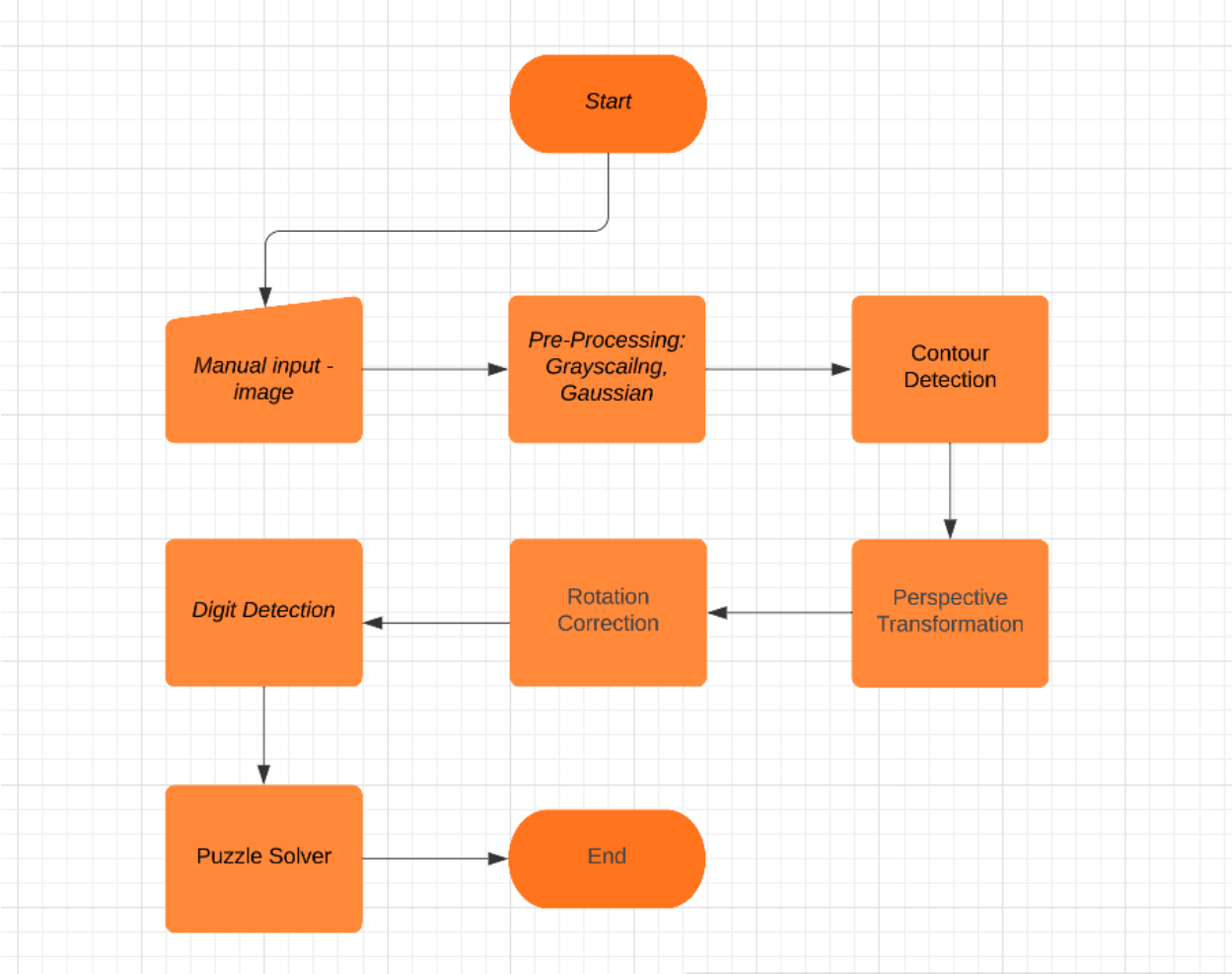
- get_perspective – take an image and the location of an interested region and return only the selected region with a perspective transformation. The location parameter is a 4-point array that represents the 4 corners of the interested region. These 4 points are used to create a perspective transformation matrix using the cv2.getPerspectiveTransform method. This matrix is then used to warp the original image into a rectangular shape with dimensions of width and height. The resulting image is returned from the function.
- split_sudoku_board - take a sudoku board and split it into a list of 81 individual cells, each containing a single element of the board. It is done by first splitting the board into 9 rows using np.array_split, then splitting each row into 9 columns. Finally, each column is resized to 48x48 pixels (reduce size and noise, not a very significant part) and normalized by dividing each pixel by 255 (bringing it to a value between 0-1 instead of 0-255).
- find_board_location - this function takes a list of contours and returns the location of the sudoku board within the contours. It starts by sorting the contours based on their area using cv2.contourArea and only keeping the top 15 contours. It then loops through the sorted

contours, using cv2.approxPolyDP to find the contour that represents a **rectangle**. If it finds a contour with 4 sides, it sets board_location to that contour and returns it.

- locate_sudoku_board – gets an image as an input and returns both the sudoku board and its location in the original image. The first step is to **convert the image to grayscale** and apply a bilateral filter to it. Then, the function uses cv2.Canny to **detect edges** in the image. cv2.findContours is used to **find contours** in the edges image, which are then passed to the find_board_location (previous function) to **find the location of the sudoku board**. If no board is found, the image is rotated by 90 degrees and the process is repeated. If a board is found, a perspective transformation is applied to the original image using get_perspective (previous function), and the resulting image is returned as the sudoku board.
- find_numbers - takes a list of cells and applies an OCR model to predict the number in each cell, or 0 if the cell is missing. It uses the tf.keras.models.load_model method to load a pre-trained model, which is then used to make predictions using the predict method. The function returns a 9x9 array of integers, where each element represents the predicted number in the corresponding cell, or 0 if the cell is empty.
- try_rotating - this function takes an image of a sudoku board and tries to detect the orientation of the numbers in it. The function starts by **checking the size of the image and resizing it** to a smaller resolution if it's too large. Then, it performs a **demeaning operation** (subtracting the mean/average value on each pixel, so some of the values would be negative and some positive) on the image to **remove any overall brightness or contrast** differences that could interfere with the Radon transform and subsequent reconstruction process. Next, it performs the **Radon transform** on the image to find the **"busiest rotation", RMS-wise**, (where the total Root Mean Square is maximized) since it's an indication of patterns recurring in the image. **To rotate the image to the correct orientation**, we first tilt it using the calculations we did with the Radon transform. Then, If the image is still rotated at exactly 90/180/270 degrees, it is tilted back using Pytesseract. The function then extracts the orientation information and rotates the image based on that information. Finally, the rotated image is displayed using cv2_imshow and returned by the function.

In conclusion, this is our **Main code** - the main function of the program that is responsible for detecting the sudoku puzzle in the input image. It performs the following steps: **Load the input image** by loading the input image using the OpenCV function cv2.imread(). This function returns a NumPy array that represents the values of each pixel and the image loaded. Secondly, we **preprocess the image** - prepare it for sudoku detection. The preprocessing steps include converting the image to grayscale, thresholding the image to reduce noise, and applying morphological operations to close any gaps in the image. After that, we **detect the contours** - The next step is to detect the contours in the preprocessed image. Contours are the outlines of objects in an image, and they can be used to detect the boundaries of the sudoku puzzle in the image. To detect the contours, the function uses the OpenCV function cv2.findContours(). Then, we **find the largest contour**: find the largest contour in the image, which corresponds to the sudoku puzzle. To find the largest contour, we iterate through all the contours and select the one with the **largest area**. We then use **perspective transformation** to the image to **obtain a bird's eye** view of the sudoku puzzle, using the OpenCV function cv2.getPerspectiveTransform(). The perspective transform is applied to the image such that the sudoku puzzle is transformed into a perfect square. Before we can continue, we make sure that the square is rotated properly, whilst **ensuring the numbers has the correct**

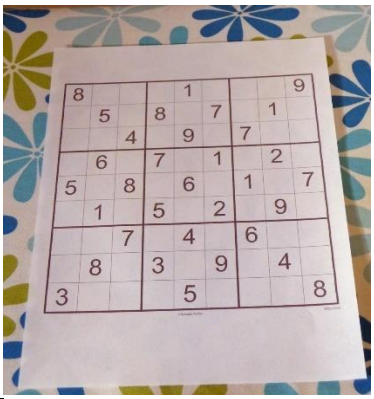
orientation. We then **divide the image into 81 cells**: each cell represents a single cell in the sudoku puzzle, followed by the **extraction of the digits**: the final step in the processing of the image. To extract the digits from each cell and solve the sudoku puzzle, we use a pre-trained model, iterate over each cell, and let the model predict the digit for us. Last but not least, we use a sudoku solver we found online to **solve the puzzle** extracted from the image.



Examples

Online Images – Study Cases

First case: the image is not rotated, pictured with little to no tilt. This photo was our study case since we found it in a CV article regarding this matter.

Original image: 

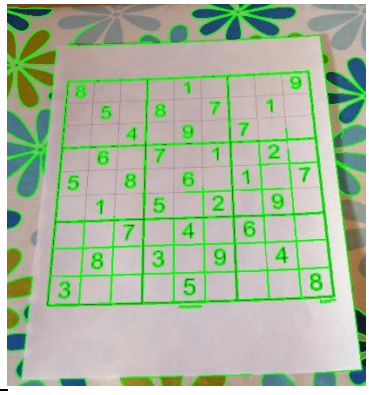
Detected contours: 

Image after Perspective Transformation: 

Puzzle inferred from the image:

```
[[8 0 0 0 1 0 0 0 9]
 [0 5 0 8 0 7 0 1 0]
 [0 0 4 0 9 0 7 0 0]
 [0 6 0 7 0 1 0 2 0]
 [5 0 8 0 6 0 1 0 7]
 [0 1 0 5 0 2 0 9 0]
 [0 0 7 0 4 0 6 0 0]
 [0 8 0 3 0 9 0 4 0]
 [3 0 0 0 5 0 0 0 8]]
```

```
+ ----------- + ----------- + ----------- +
| 8  7  2 | 4  1  3 | 5  6  9 |
| 9  5  6 | 8  2  7 | 3  1  4 |
| 1  3  4 | 6  9  5 | 7  8  2 |
+ ----------- + ----------- + ----------- +
| 4  6  9 | 7  3  1 | 8  2  5 |
| 5  2  8 | 9  6  4 | 1  3  7 |
| 7  1  3 | 5  8  2 | 4  9  6 |
+ ----------- + ----------- + ----------- +
| 2  9  7 | 1  4  8 | 6  5  3 |
| 6  8  5 | 3  7  9 | 2  4  1 |
| 3  4  1 | 2  5  6 | 9  7  8 |
+ ----------- + ----------- + ----------- +
```

The solve:

Second case: the image is not rotated or tilted; this time the OCR model works perfectly.

Original image: 

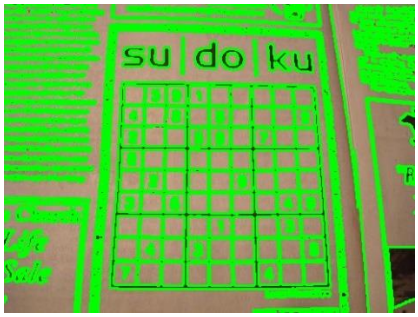
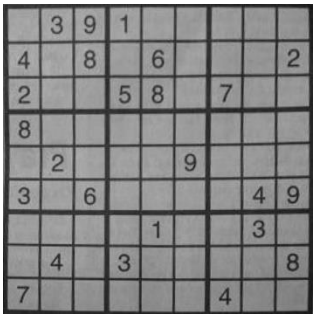
Detected Contours: 

Image after Perspective Transformation: 

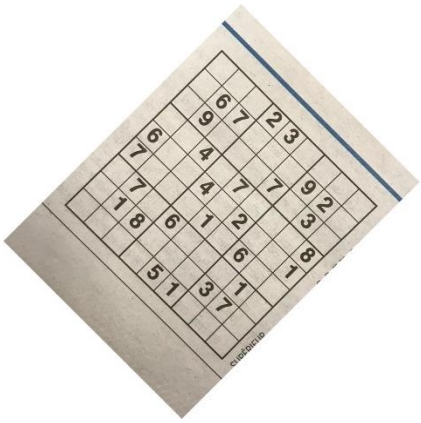
```
[[0 3 9 1 0 0 0 0 0]
 [4 0 8 0 6 0 0 0 2]
 [2 0 0 5 8 0 7 0 0]
 [8 0 0 0 0 0 0 0 0]
 [0 2 0 0 0 9 0 0 0]
 [3 0 6 0 0 0 0 4 9]
 [0 0 0 0 1 0 0 3 0]
 [0 4 0 3 0 0 0 0 8]
 [7 0 0 0 0 0 4 0 0]]
```

Puzzle inferred from the image:

```
+ ----------- + ----------- + ----------- +
| 5  3  9 | 1  7  2 | 6  8  4 |
| 4  7  8 | 9  6  3 | 1  5  2 |
| 2  6  1 | 5  8  4 | 7  9  3 |
+ ----------- + ----------- + ----------- +
| 8  9  7 | 6  4  5 | 3  2  1 |
| 1  2  4 | 8  3  9 | 5  7  6 |
| 3  5  6 | 7  2  1 | 8  4  9 |
+ ----------- + ----------- + ----------- +
| 9  8  5 | 4  1  6 | 2  3  7 |
| 6  4  2 | 3  5  7 | 9  1  8 |
| 7  1  3 | 2  9  8 | 4  6  5 |
+ ----------- + ----------- + ----------- +
```

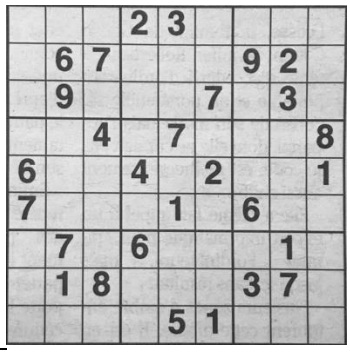
The solve:

Third case: the image is rotated at 45 degrees, not tilted on the Y-Z plane.



Original image:

Detected Contours:

Image after Perspective Transformation: 

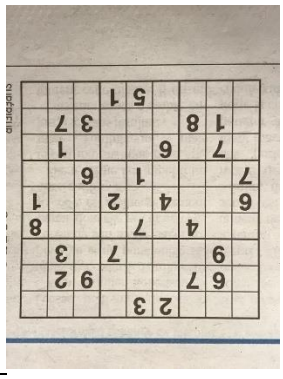
Puzzle inferred from the image:

```
[[0 0 0 2 3 0 0 0 0]
 [0 6 7 0 0 0 9 2 0]
 [0 9 0 0 0 7 0 3 0]
 [0 0 4 0 7 0 0 0 8]
 [6 0 0 4 0 2 0 0 1]
 [7 0 0 0 1 0 6 0 0]
 [0 7 0 6 0 0 0 1 0]
 [0 1 8 0 0 0 3 7 0]
 [0 0 0 0 5 1 0 0 0]]
```

The solve:

```
+ ----------- + ----------- + ----------- +
| 8  4  5 | 2  3  9 | 1  6  7 |
| 3  6  7 | 1  4  8 | 9  2  5 |
| 2  9  1 | 5  6  7 | 8  3  4 |
+ ----------- + ----------- + ----------- +
| 1  5  4 | 3  7  6 | 2  9  8 |
| 6  8  3 | 4  9  2 | 7  5  1 |
| 7  2  9 | 8  1  5 | 6  4  3 |
+ ----------- + ----------- + ----------- +
| 4  7  2 | 6  8  3 | 5  1  9 |
| 5  1  8 | 9  2  4 | 3  7  6 |
| 9  3  6 | 7  5  1 | 4  8  2 |
+ ----------- + ----------- + ----------- +
```

Fourth case: the image is rotated at 180 degrees, not tilted on the Y-Z plane.

Original image: 

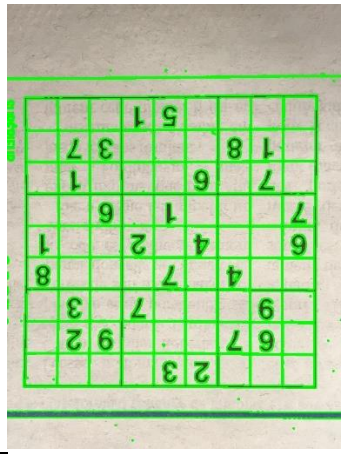
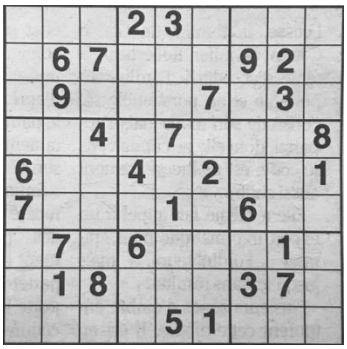
Detected Contours: 

Image after Perspective Transformation: 

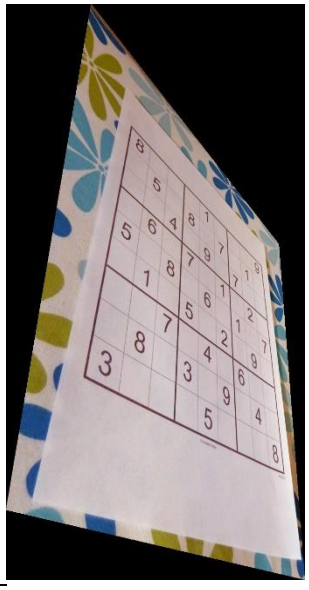
Puzzle inferred from the image:

```
[[0 0 0 2 3 0 0 0 0]
 [0 6 7 0 0 0 9 2 0]
 [0 9 0 0 0 7 0 3 0]
 [0 0 4 0 7 0 0 0 8]
 [6 0 0 4 0 2 0 0 1]
 [7 0 0 0 1 0 6 0 0]
 [0 7 0 6 0 0 0 1 0]
 [0 1 8 0 0 0 3 7 0]
 [0 0 0 0 5 1 0 0 0]]
```

The solve:

```
+ ----------- + ----------- + ----------- +
| 8  4  5 | 2  3  9 | 1  6  7 |
| 3  6  7 | 1  4  8 | 9  2  5 |
| 2  9  1 | 5  6  7 | 8  3  4 |
+ ----------- + ----------- + ----------- +
| 1  5  4 | 3  7  6 | 2  9  8 |
| 6  8  3 | 4  9  2 | 7  5  1 |
| 7  2  9 | 8  1  5 | 6  4  3 |
+ ----------- + ----------- + ----------- +
| 4  7  2 | 6  8  3 | 5  1  9 |
| 5  1  8 | 9  2  4 | 3  7  6 |
| 9  3  6 | 7  5  1 | 4  8  2 |
+ ----------- + ----------- + ----------- +
```

Fifth case: the image is tilted 60 degrees around the Y axis; 30 degrees around the Z axis.

Original image: 

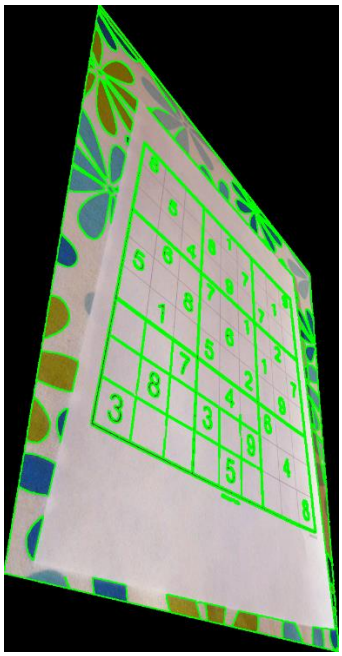
Detected Contours: 

Image after Perspective Transformation: 

NOTE: as much as you look closer to the top-right angle, where the image was most affected by the rotation, the vision is getting blurrier, and the lines become very thin and weak, in comparison to the first case we examined.

Puzzle inferred from the image:

```
[[8 0 0 0 1 0 0 0 9]
 [0 5 0 8 0 7 0 1 0]
 [0 0 4 0 9 0 7 0 0]
 [0 6 0 7 0 1 0 2 0]
 [5 0 8 0 6 0 1 0 7]
 [0 1 0 5 0 2 0 9 0]
 [0 0 7 0 4 0 6 0 0]
 [0 8 0 3 0 9 0 4 0]
 [3 0 0 0 5 0 0 0 8]]
```

The solve:

```
+ ----------- + ----------- + ----------- +
| 8  7  2 | 4  1  3 | 5  6  9 |
| 9  5  6 | 8  2  7 | 3  1  4 |
| 1  3  4 | 6  9  5 | 7  8  2 |
+ ----------- + ----------- + ----------- +
| 4  6  9 | 7  3  1 | 8  2  5 |
| 5  2  8 | 9  6  4 | 1  3  7 |
| 7  1  3 | 5  8  2 | 4  9  6 |
+ ----------- + ----------- + ----------- +
| 2  9  7 | 1  4  8 | 6  5  3 |
| 6  8  5 | 3  7  9 | 2  4  1 |
| 3  4  1 | 2  5  6 | 9  7  8 |
+ ----------- + ----------- + ----------- +
```

Examples of external problems and how our project coped with them:

Sixth case: the image is not rotated, pictured with little to no tilt. Here, **the OCR model had caused some problems** with a few digit predictions (most likely since this model was trained on different data), but the image was processed properly. For more information, see section 4 under “project limitations”.

Original image: 

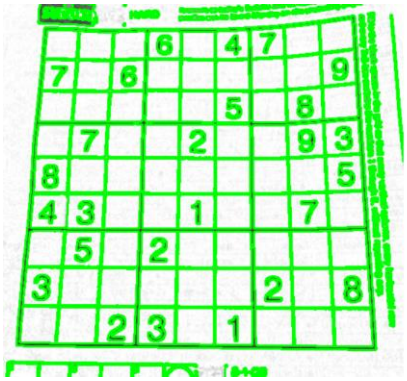
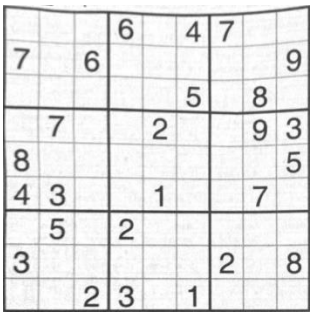
Detected contours: 

Image after Perspective Transformation: 

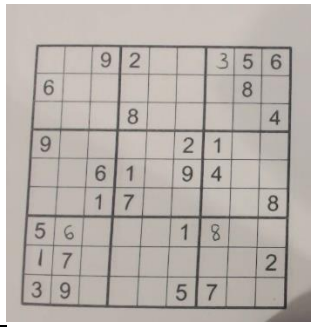
Puzzle inferred from the image:

```
[[0 0 0 6 0 4 7 0 0]
 [7 0 6 0 0 7 7 0 9]
 [0 0 0 0 0 5 0 8 0]
 [0 7 0 0 2 0 0 9 3]
 [8 0 0 0 0 0 0 0 5]
 [4 3 0 0 1 0 0 7 0]
 [0 5 0 2 0 0 0 0 0]
 [3 0 0 0 0 0 2 0 8]
 [0 0 2 3 0 1 0 0 0]]
```

The solve (after bad digit inference):

```
Solution Status = Infeasible
```

Seventh case: the image is not rotated, pictured with little to no tilt. Here, the image is very blurry (once we look at it full-sized) and has a few handwritten numbers. For more information, please see section 1 under “project limitations”.

Original image: 

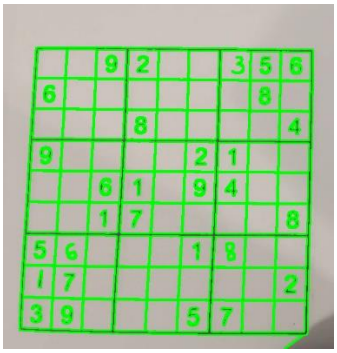
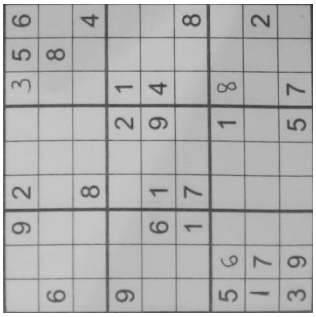
Detected contours: 

Image after Perspective Transformation: 

Puzzle Inferred:

```
[[9 0 4 0 0 8 0 9 0]
 [6 8 0 0 0 0 0 0 0]
 [2 0 0 4 4 0 8 0 4]
 [0 0 0 4 9 0 7 0 2]
 [0 0 0 0 0 0 0 0 0]
 [6 0 9 0 7 4 0 0 0]
 [8 0 0 0 8 7 0 0 0]
 [0 0 0 0 0 0 9 4 8]
 [0 8 0 8 0 0 6 7 2]]
```

(image is too blurry, some digits are handwritten – can't rotate the image)

The Solve (after bad inference):

```
Solution Status = Infeasible
```

Real-time problems - Our first try was using Whatsapp to send our images to the PC (to run the solver on). We saw that the compression Whatsapp is using corrupts the image and thus they couldn't be used. Later on, we understood that picturing the sudoku using the original photograph app of our phone had somewhat wrapped and edited the original photos, which also had the same effect on the pictures.

Real Life Images – Solution

Using the app **CameraScanner** we had the option to get the unedited, original-size pictures, and so the project was able to use them properly.

Eighth case: Straight, not tilted image using our own picture.

Original image:

Detected contours:

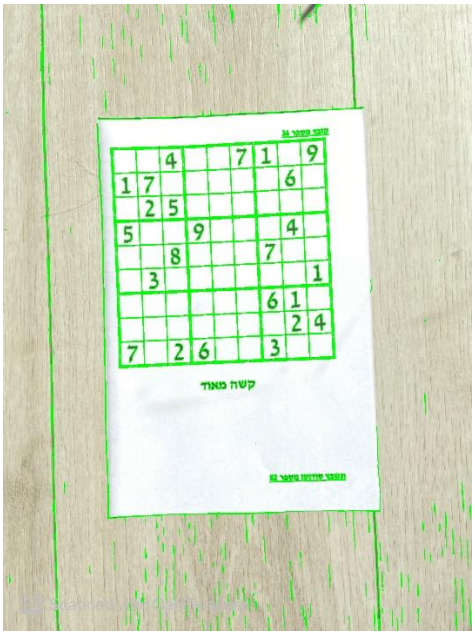
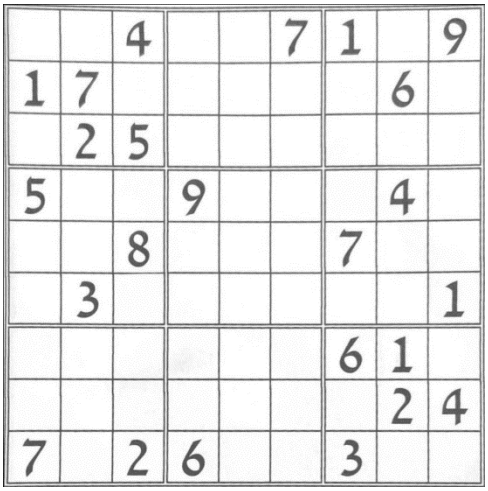


Image after Perspective Transformation:



Puzzle Inferred:

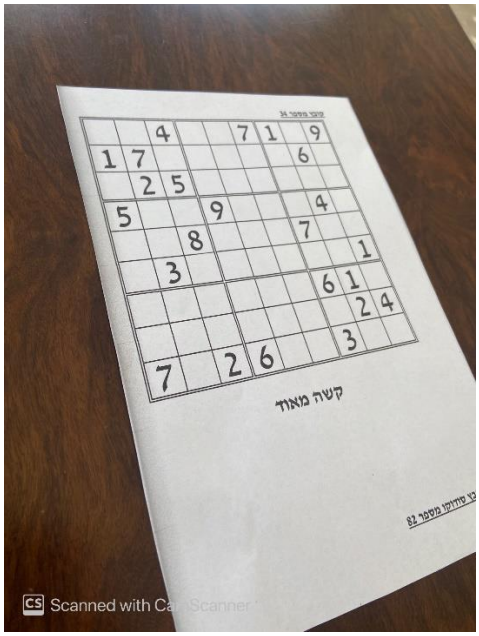
```
[[0 0 4 0 0 7 1 0 9]
 [1 7 0 0 0 0 0 6 0]
 [0 2 5 0 0 0 0 0 0]
 [5 0 0 9 0 0 0 4 0]
 [0 0 8 0 0 0 7 0 0]
 [0 3 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 6 1 0]
 [0 0 0 0 0 0 0 2 4]
 [7 0 2 6 0 0 3 0 0]]
```

The solve:

```
+ ----------- + ----------- + ----------- +
| 3  8  4 | 2  6  7 | 1  5  9 |
| 1  7  9 | 3  5  8 | 4  6  2 |
| 6  2  5 | 1  4  9 | 8  7  3 |
+ ----------- + ----------- + ----------- +
| 5  1  6 | 9  7  3 | 2  4  8 |
| 4  9  8 | 5  1  2 | 7  3  6 |
| 2  3  7 | 4  8  6 | 5  9  1 |
+ ----------- + ----------- + ----------- +
| 9  5  3 | 8  2  4 | 6  1  7 |
| 8  6  1 | 7  3  5 | 9  2  4 |
| 7  4  2 | 6  9  1 | 3  8  5 |
+ ----------- + ----------- + ----------- +
```

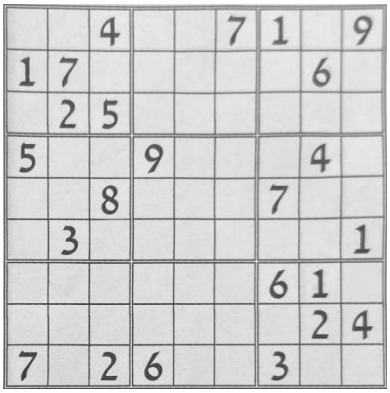
Ninth case: Tilted, rotated image using our own picture – left side perspective.

Original image:



Detected contours:

Image after Perspective Transformation:



Puzzle Inferred:

```
[[0 0 4 0 0 7 1 0 9]
 [1 7 0 0 0 0 0 6 0]
 [0 2 5 0 0 0 0 0 0]
 [5 0 0 9 0 0 0 4 0]
 [0 0 8 0 0 0 7 0 0]
 [0 3 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 6 1 0]
 [0 0 0 0 0 0 0 2 4]
 [7 0 2 6 0 0 3 0 0]]
```

The solve:

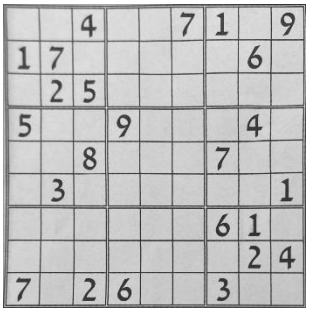
```
+ ----------- + ----------- + ----------- +
| 3  8  4 | 2  6  7 | 1  5  9 |
| 1  7  9 | 3  5  8 | 4  6  2 |
| 6  2  5 | 1  4  9 | 8  7  3 |
+ ----------- + ----------- + ----------- +
| 5  1  6 | 9  7  3 | 2  4  8 |
| 4  9  8 | 5  1  2 | 7  3  6 |
| 2  3  7 | 4  8  6 | 5  9  1 |
+ ----------- + ----------- + ----------- +
| 9  5  3 | 8  2  4 | 6  1  7 |
| 8  6  1 | 7  3  5 | 9  2  4 |
| 7  4  2 | 6  9  1 | 3  8  5 |
+ ----------- + ----------- + ----------- +
```

Tenth case: Tilted, rotated image using our own picture – right side perspective.

Original image:

Detected contours:

Image after Perspective Transformation:



Puzzle Inferred:

```
[[0 0 4 0 0 7 1 0 9]
 [1 7 0 0 0 0 0 6 0]
 [0 2 5 0 0 0 0 0 0]
 [5 0 0 9 0 0 0 4 0]
 [0 0 8 0 0 0 7 0 0]
 [0 3 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 6 1 0]
 [0 0 0 0 0 0 0 2 4]
 [7 0 2 6 0 0 3 0 0]]
```

The solve:

```
+ ----------- + ----------- + ----------- +
| 3  8  4 | 2  6  7 | 1  5  9 |
| 1  7  9 | 3  5  8 | 4  6  2 |
| 6  2  5 | 1  4  9 | 8  7  3 |
+ ----------- + ----------- + ----------- +
| 5  1  6 | 9  7  3 | 2  4  8 |
| 4  9  8 | 5  1  2 | 7  3  6 |
| 2  3  7 | 4  8  6 | 5  9  1 |
+ ----------- + ----------- + ----------- +
| 9  5  3 | 8  2  4 | 6  1  7 |
| 8  6  1 | 7  3  5 | 9  2  4 |
| 7  4  2 | 6  9  1 | 3  8  5 |
+ ----------- + ----------- + ----------- +
```

Project Limitations

We should acknowledge that there are limitations to our project:

- Blurry Images: The project is not designed to handle blurry images, as the blurred pixels can lead to incorrect contour detection and cause errors in the output – the seventh case is an example.
- Tilted Images: If the input image is too tilted, contour detection and perspective correction can become difficult, resulting in incorrect output.
- Bad Image Format/Bad Image Ratio: The project is optimized for a specific image format and aspect ratio, and if the input image deviates too much from these parameters, it may not produce accurate results.
- Corruption Caused by Compression/App Auto Edit: As explained earlier, corruption caused by compression or automatic editing by apps can make it difficult for this project to use certain images. For instance, images sent through WhatsApp are compressed for size optimization, while using the default camera app may result in automatic image editing to improve visual appeal. These processes can cause issues for our code.
- OCR Limitations: The OCR model used in this project is pre-trained and not specifically designed for this project. It may have limitations in detecting characters, especially in cases where the font or writing style differs significantly from the training data - the sixth case is an example.
- Lighting Conditions: If the lighting conditions in the input image are not optimal, the OCR model may have difficulty recognizing the characters, leading to incorrect results.
- Background Distractions: If there are too many distractions in the background of the input image, the contour detection and perspective correction may become incorrect, leading to incorrect output.

Overall, it's important to keep in mind that the project is designed for a specific set of conditions and limitations, and its accuracy may decrease if the input image deviates from these conditions.

Conclusion

In this project, we set out to develop an image-based sudoku solver that would be capable of accurately identifying and solving sudoku puzzles in images. Through the use of computer vision and image processing techniques, we were able to develop a solution that was able to effectively identify sudoku grids in images, even if they were tilted or not perfectly straight. This was a challenging task, but we were able to achieve our goal using various techniques such as contour detection, perspective transformation, and image normalization.

Initially, detecting a sudoku grid in an image was a daunting task, and our solution struggled even with basic images. However, with persistence and diligent effort, we have made significant progress over time. Our solution now works not only with straightforward images but also with more complex ones, a testament to our team's hard work and determination. Iterative changes were made to the code, testing different approaches and algorithms, until a reliable method for detecting the grid was found. The final solution was capable of accurately detecting grids in a wide range of images, including those with tilts in

both the X-Y axis and over time, even on the Z-axis. This success was a testament to the power of trial and error, and the importance of perseverance in the face of difficulties.

In addition to its success with a variety of images, the project also has the potential to work in a real-time environment, using photos taken by a phone camera. By integrating the image processing algorithms with a live environment, the solution could be used to solve Sudoku puzzles in real-time, which would be a valuable tool for Sudoku enthusiasts or those who want to quickly check their solutions. The ability to operate on real-time photos demonstrates the versatility and potential of the project.

While this project has achieved its goals even beyond our initial expectations, there is still room for further improvement. One potential area of improvement would be to optimize the algorithm to make it run faster, which would enable it to handle larger images more efficiently, instead of using a traceback algorithm. Additionally, there is scope for integrating more advanced computer vision techniques to further improve the accuracy of the solution. Overall, this project serves as a proof of concept for the application of computer vision in solving real-world problems, while acknowledging the potential to be further developed and applied in a range of different domains.