Submitted for
Dr. Mahmoud Khalil
Eng. Ahmed Salama

Computer Vision
CSE 483

# Major Task

## Submitted by:


| | |
|---|---|
| Omar Tarek Mohamed | 19P2772 |
| Shehab Adel Ramadan | 19P4512 |
| Amr Essam Kamal El-din | 19P5641 |
| Omar Salah Abd Elkader | 19P4606 |
| Mohamed Reda Mohamed Selim | 19P4160 |

# Table of Content

## Contents

# Table of figures

# INTRODUCTION

This report tackles the fascinating task of extracting and solving Sudoku puzzles from real-life sources that were collected through a camera lens, delving into the field of computer vision.
Our primary objective is to efficiently interpret the puzzle grid, negotiating the complexities of picture preprocessing and cleaning to produce a scale-neutral digitized, undistorted, and noiseless binarized (grayscale) grid.

This project encourages us to apply the theoretical knowledge we have learned in our course to real-world problem-solving situations by providing a practical application of the principles we have covered.

We address this puzzle by concentrating on the interface between image processing and Optical Character Recognition (OCR). Specifically, we investigate how fundamental OCR methods might be utilized to get values from the grid.

Even if the OCR component is outside the purview of the course, it gives us a chance to learn more about the topic and build a basic application of emerging technology.

The purposeful decision not to use machine learning for optical character recognition highlights how crucial it is to understand fundamental ideas before exploring more complex uses, giving us a more nuanced understanding of how technology can be used to address real-world issues.

In addition to shedding light on the complexities of Sudoku puzzle extraction, this paper intends to enable us to close the knowledge gap between theory and practice in the ever-evolving field of computer vision.

The project was challenging due to the fact of not using any kind of AI (Artificial Intelligence) to solve the sudoku problem, whether to OCR the numbers in image or to solve board.

# PROJECT STRUCTURE

The project is divided into 2 essential phases, phase 1 which is for preprocessing the sudoku board and extracting the tiles from it. Moreover, the second phase is to apply OCR to detect the numbers in board and introduce it in a 9x9 matrix and solve the sudoku board in that array. The project aimed to pass as many test cases as possible.

**All demonstrations done using the ideal testcase (Normal-1).**

## Phase 1

In phase 1 we applied generalized preprocessing techniques in order to pass as many testcases as possible.
Phase 1 divided into the following steps:

### Displaying image and its gray scaled version.

In this step we just display the input sudoku board image and preview its gray scaled version.

Displaying Image and its Gray Scaled Version

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

print("___________________ RGB IMAGE________________________ \n")
img = cv2.imread("/content/TestCases/01-Normal.jpg")  # Fix: Change "opencv2" to "cv2"
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img_rgb)
plt.show()

print("___________________ GRAY IMAGE________________________ \n")
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
plt.imshow(img_gray, cmap="gray")
plt.show()
```

*Figure 1: Step 1*

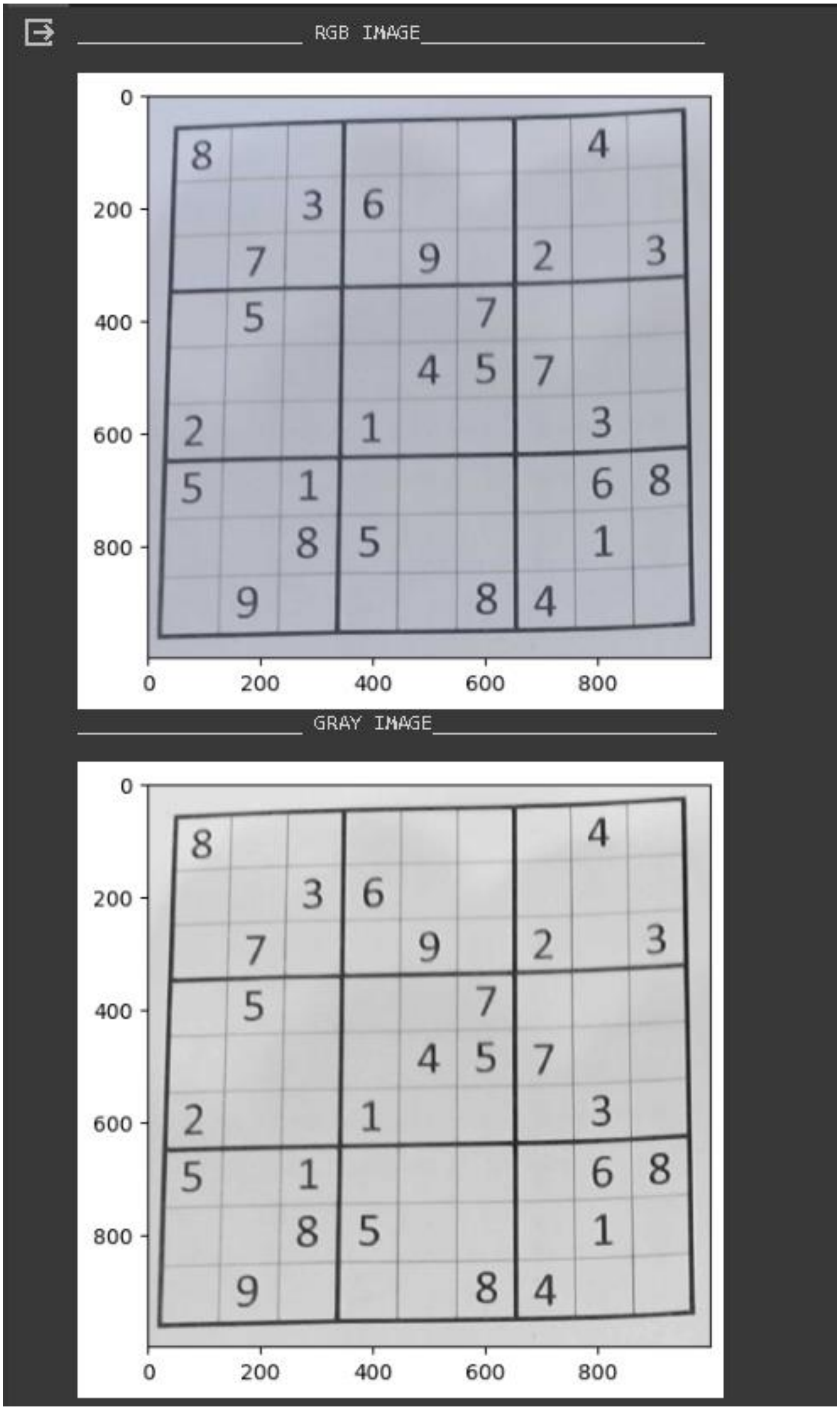


*Figure 2: Step 1.1*

# Detecting and handling salt and pepper noise

In this step, we apply Morphological opening and closing and get their difference to estimate salt and pepper noise. If the percentage of salt or pepper noise exceeds adjusted threshold. Then the input will pass through step 3 which apply median filter on the input to fix the salt and pepper noise.

**Detecting Salt And Pepper**

```python
def contains_salt_or_pepper_noise(img_gray, threshold=0.5, kernel_size=3):
    # Apply morphological opening to remove salt noise
    opening = cv2.morphologyEx(img_gray, cv2.MORPH_OPEN, np.ones((kernel_size, kernel_size), np.uint8))

    # Apply morphological closing to remove pepper noise
    closing = cv2.morphologyEx(img_gray, cv2.MORPH_CLOSE, np.ones((kernel_size, kernel_size), np.uint8))

    # Calculate the absolute difference between the original and processed images
    diff_opening = cv2.absdiff(img_gray, opening)
    diff_closing = cv2.absdiff(img_gray, closing)

    # Calculate the percentage of changed pixels
    salt_percentage = np.sum(diff_opening > threshold) / img_gray.size
    pepper_percentage = np.sum(diff_closing > threshold) / img_gray.size

    # Check if either salt or pepper exceeds the threshold
    return salt_percentage > threshold or pepper_percentage > threshold

result = contains_salt_or_pepper_noise(img_gray)
print(result)
```

```
False
```

**Fixing Salt and Pepper**

```python
if(result):
  img_gray_filtered = cv2.medianBlur(img_gray, 9)
  print("Image contains mal7 we fefel.")

  # Display the original and filtered images side by side
  plt.subplot(1, 2, 1)
  plt.imshow(img_gray, cmap="gray")
  plt.title("Original Image")

  plt.subplot(1, 2, 2)
  plt.imshow(img_gray_filtered, cmap="gray")
  plt.title("Filtered Image")
  plt.show()
  img_gray = img_gray_filtered
```

*Figure 3: Step 2 and 3*

## Discovering image nature and classifying it

In this step we discover the image nature of the input image and classify whether its normal image (neither dark nor inverted) or if its dark or inverted. This step handles the inverted images as it reverts it to its correct form (white pixels for number and black for background).

**Discovering Image Nature and Fixing it in Case if its Negative (DarkMode Testcase #13)**

```
[ ] def analyze_image(gray_image):
        # Calculate histogram
        hist = cv2.calcHist([gray_image], [0], None, [256], [0, 256])

        # Calculate the percentage of dark pixels (intensity 0-127)
        dark_percentage = np.sum(hist[:128]) / np.sum(hist) * 100

        # Calculate the percentage of bright pixels (intensity 128-255)
        bright_percentage = np.sum(hist[128:]) / np.sum(hist) * 100

        # Analyze the result
        if dark_percentage > 20 and bright_percentage < 20:
            print( "The image is likely dark.")
            return gray_image
        elif dark_percentage > 59:
            print ("The image is likely inverted")
            inverted_img = 255 - gray_image

            # Display the original and inverted images side by side
            plt.subplot(1, 2, 1)
            plt.imshow(gray_image, cmap="gray")
            plt.title("Original Image")

            plt.subplot(1, 2, 2)
            plt.imshow(inverted_img, cmap="gray")
            plt.title("Inverted Image")

            plt.show()
            gray_image = inverted_img

            return gray_image
        else:
            print("The image is neither dark nor inverted. \n")
            return gray_image

    img_gray = analyze_image(img_gray)
    print('\n')
    hist = cv2.calcHist([img_gray], [0], None, [256], [0, 256])
    plt.plot(hist)
    plt.title("Pixel Intensity Histogram")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
```

*Figure 4: Step 4*

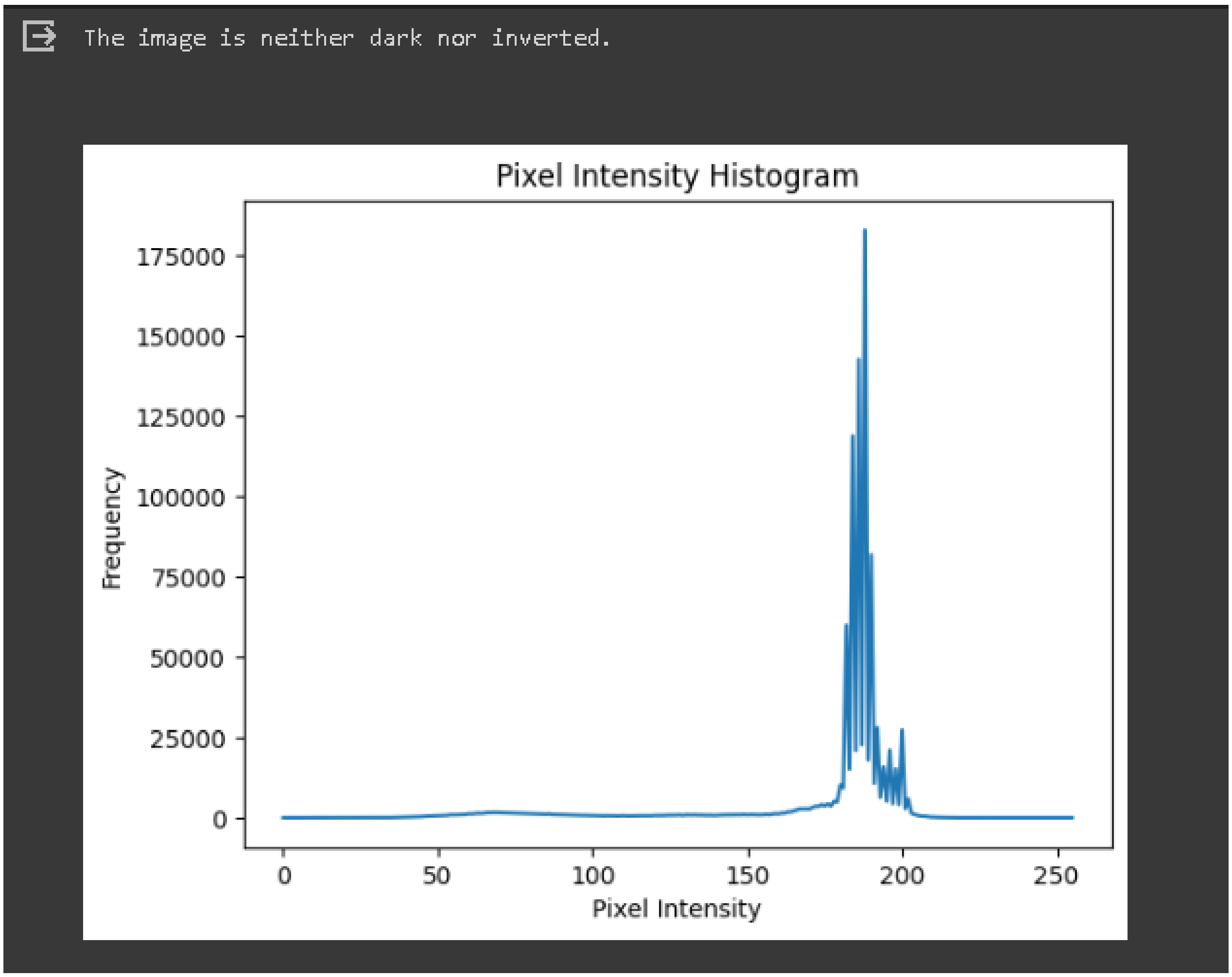


*Figure 5: Step 4.1*

## Generalized adaptive thresholding

In this step we apply generalized adaptive thresholding by acquiring the block size and C parameters from the image nature itself. This process is done by calculating the mean and standard deviation values of each of the input images, then dividing them by and adjusted factor to obtain generic block size and C values.

**Generalized Adaptive Thresholding**

```
[ ] mean_val = np.mean(img_gray)
    std_dev = np.std(img_gray)

    # Use adaptive values for blockSize and C
    adaptive_block_size = int(mean_val / 0.9)
    adaptive_block_size = adaptive_block_size - 1 if adaptive_block_size % 2 == 0 else adaptive_block_size
    adaptive_C = int(std_dev / 2)

    # Apply adaptive thresholding with the calculated parameters
    img_gray_threshed = cv2.adaptiveThreshold(
        src=img_gray,
        maxValue=255,
        adaptiveMethod=cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        thresholdType=cv2.THRESH_BINARY_INV,
        blockSize=adaptive_block_size,
        C=adaptive_C
    )

    # Display the thresholded image
    plt.imshow(img_gray_threshed, cmap="gray")
    plt.show()
```

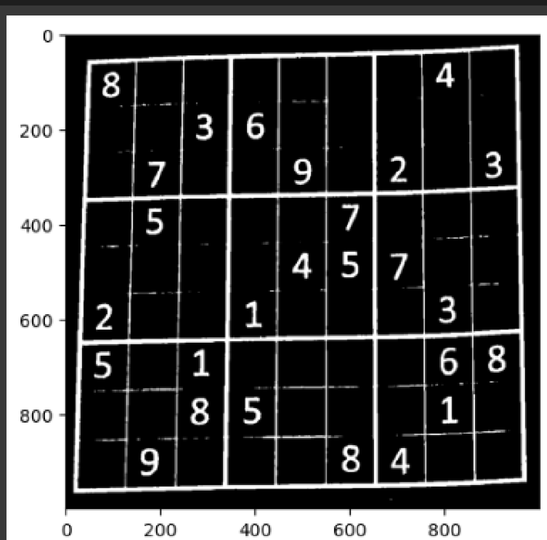


*Figure 6: Step 5*

## Generalized Hough transform

In this step we apply generalized Hough transform. We do this by acquiring the threshold, min_line_length, and max_line_gap parameters using  the input's image width and height allowing the parameters to be generic.

**Generalized Hough Transform**

```
[ ] print("___________________ GAUSSIAN SMOOTHING + MORPHOLOGICAL + HOUGH________________________ \n")
    # Apply Gaussian blur for noise removal
    blurred_img = cv2.GaussianBlur(img_gray_threshed, (5, 5), 0)

    # Apply morphological opening for further noise removal
    kernel = np.ones((3, 3), np.uint8)
    opened_img = cv2.morphologyEx(blurred_img, cv2.MORPH_OPEN, kernel)

    def generalized_hough_transform(image):
        # Calculate parameters based on image size
        height, width = image.shape[:2]
        rho = 1
        theta = np.pi / 180
        threshold = int(max(height, width) / 50)
        min_line_length = int(min(height, width) / 5)
        max_line_gap = int(min(height, width) / 80)

        # Apply Hough Line Transform
        lines = cv2.HoughLinesP(
            image=image,
            rho=rho,
            theta=theta,
            threshold=threshold,
            minLineLength=min_line_length,
            maxLineGap=max_line_gap
        )

        return lines


    lines = generalized_hough_transform(opened_img)


    # Draw lines on a black image
    tmp_img = np.zeros_like(img_gray_threshed, dtype=np.uint8)
    for x1, y1, x2, y2 in lines[:, 0]:
        cv2.line(tmp_img, (x1, y1), (x2, y2), (255, 0, 0), 2)

    # Display the result
    plt.imshow(tmp_img, cmap='gray')
    plt.show()
```

*Figure 7: Step 6*

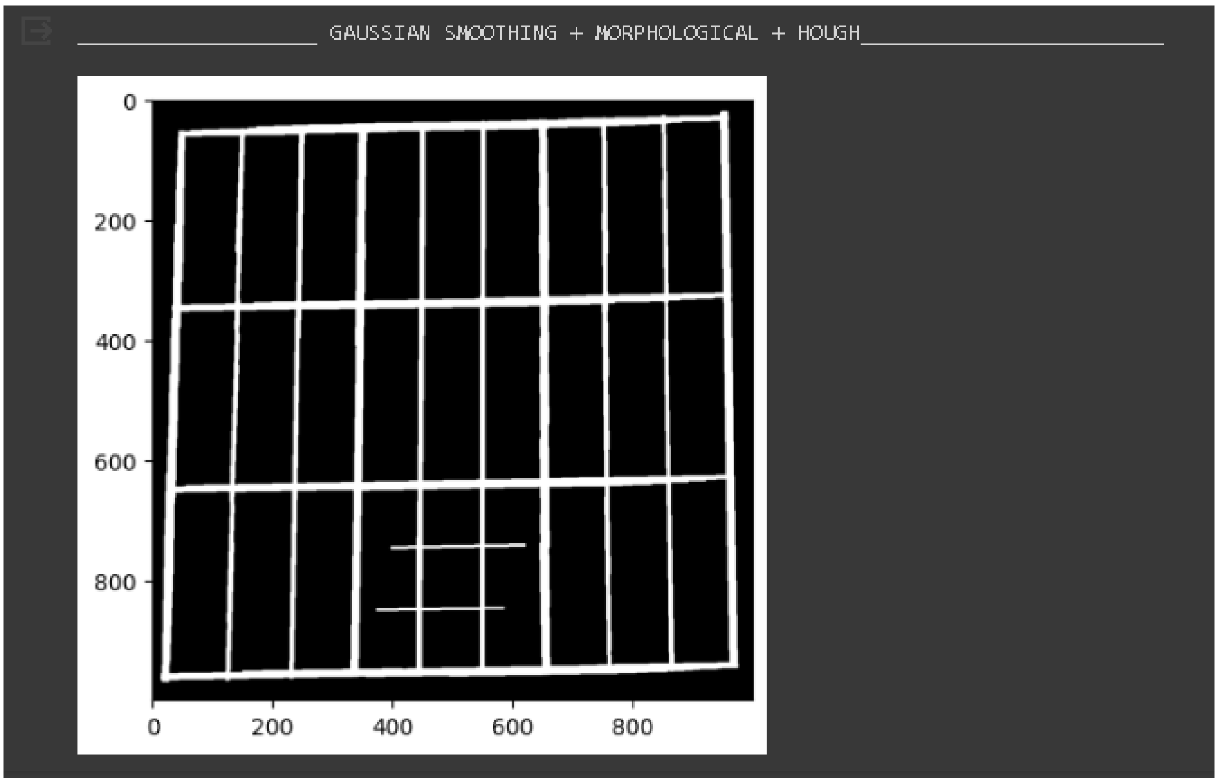


*Figure 8: Step 6.1*

## Finding intersections

In this step we find the intersection points along the sudoku board in the input image. We obtain them by estimating the angle between all lines in the image and estimate whether a point is an intersection point or not based on that angle.

Finding Intersections

```
[ ]
def angle_between_lines(line1, line2):
    l1x1, l1y1, l1x2, l1y2 = line1
    l2x1, l2y1, l2x2, l2y2 = line2
    a1 = np.rad2deg(np.arctan2(l1y2 - l1y1, l1x2 - l1x1))
    a2 = np.rad2deg(np.arctan2(l2y2 - l2y1, l2x2 - l2x1))
    return np.abs(a1 - a2)

def intersection_point(line1, line2):
    l1x1, l1y1, l1x2, l1y2 = line1
    l2x1, l2y1, l2x2, l2y2 = line2
    nx = (l1x1*l1y2-l1y1*l1x2)*(l2x1-l2x2)-(l2x1*l2y2-l2y1*l2x2)*(l1x1-l1x2)
    ny = (l1x1*l1y2-l1y1*l1x2)*(l2y1-l2y2)-(l2x1*l2y2-l2y1*l2x2)*(l1y1-l1y2)
    d = (l1x1-l1x2)*(l2y1-l2y2)-(l1y1-l1y2)*(l2x1-l2x2)
    px = int(nx / d)
    py = int(ny / d)
    return (px, py)

def point_on_line(point, line):
    def distance(pfrom, pto):
        return np.sqrt((pfrom[0] - pto[0])**2 + (pfrom[1] - pto[1])**2)
    diff = distance(point, line[0:2]) + distance(point, line[2:4]) - distance(line[0:2], line[2:4])
    return np.abs(diff) < 70

def find_intersections(lines):
    intersections = []
    num_of_lines = len(lines[:, 0])

    for i in range(num_of_lines):
        for j in range(i + 1, num_of_lines):
            line1 = lines[i, 0]
            line2 = lines[j, 0]
            if np.array_equal(line1, line2):
                continue
            a = angle_between_lines(line1, line2)
            if 80 < a < 100:
                p = intersection_point(line1, line2)
                if point_on_line(p, line1) and point_on_line(p, line2):
                    intersections.append(p[::-1])
    return intersections
```

*Figure 9: Step 7*

```python
# Assuming you have 'lines' from your Hough transform
tmp_img2 = np.zeros_like(img_gray_threshed, dtype=np.uint16)
intersections = find_intersections(lines)
for p in intersections:

    # Check if the indices are within the valid range
    if 0 <= p[0] < tmp_img2.shape[0] and 0 <= p[1] < tmp_img2.shape[1]:
        tmp_img2[p[0], p[1]] = 5000
plt.imshow(tmp_img2 + tmp_img, cmap="gray", vmin=0, vmax=1255)
plt.show()
```

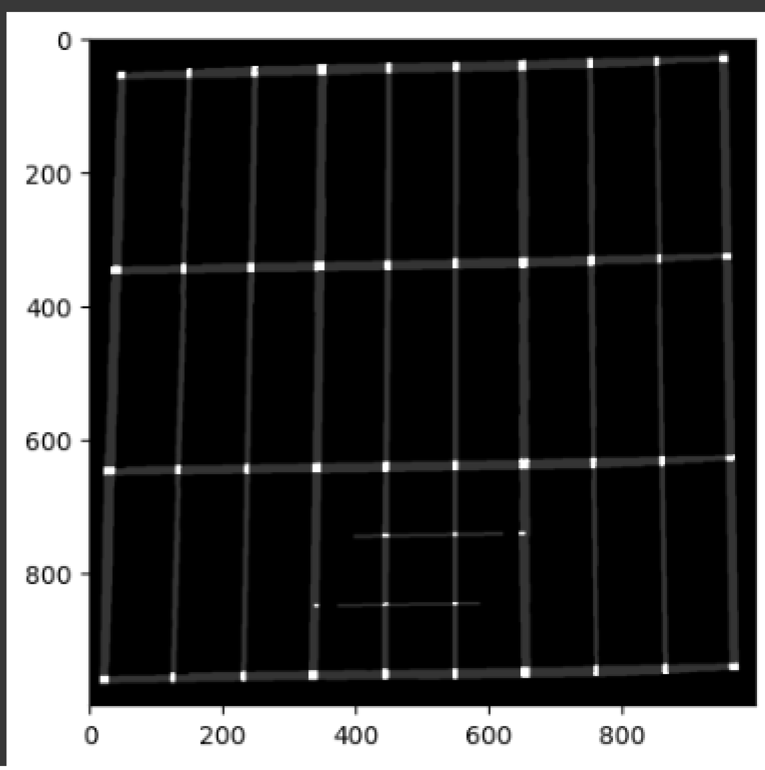


*Figure 10: Step 7.1*

## Detecting outer sudoku frame

In this step we detect the outer sudoku frame by acquiring the intersection points from the previous step, and sorting these to get the outermost four points (top-left – top-right - bottom-left - bottom-right). Then, we calculate the angles of the 4 corners of the frame to classy fig whether the 4 points create a frame or not. Also, an added distance threshold was introduced to fix the frame in case of any distortion in the image (missing frames in input image).

**Detecting Outer Sudoko Frame**

```
print("___________________GETTING OUTER FRAME________________________ \n")
p1 = sorted(intersections, key = lambda p: p[0] + p[1])[0] # topleft
p2 = sorted(intersections, key = lambda p: p[0] - p[1])[0] # topright
p3 = sorted(intersections, key = lambda p: p[0] + p[1])[-1] # bottright
p4 = sorted(intersections, key = lambda p: p[1] - p[0])[0] # bottleft

def calculate_angle(point1, point2, point3):
    # Calculate vectors
    vector1 = np.array([point1[1] - point2[1], point1[0] - point2[0]])
    vector2 = np.array([point3[1] - point2[1], point3[0] - point2[0]])
    # Calculate dot product
    dot_product = np.dot(vector1, vector2)
    # Calculate magnitudes
    magnitude1 = np.linalg.norm(vector1)
    magnitude2 = np.linalg.norm(vector2)
    # Calculate angle in degrees
    angle = np.degrees(np.arccos(dot_product / (magnitude1 * magnitude2)))
    return angle

#Added
difference_angle_threshold = 12; # 12 till #7
# differenced_value = 10
if np.absolute(calculate_angle(p4,p1,p2) - calculate_angle(p3,p2,p1)) > difference_angle_threshold:
    if p1[0] > p2[0]:
        p1 = (p2[0]   , p1[1])
    else:
        p2 = (p1[0], p2[1])
if np.absolute(calculate_angle(p2,p3,p4) - calculate_angle(p3,p2,p1)) > difference_angle_threshold:
    if p3[0] > p4[0]:
        p4 = (p3[0]   , p4[1])
    else:
        p3 = (p4[0], p3[1])

coords = np.int32([[p1[::-1], p2[::-1], p3[::-1], p4[::-1]]])
tmp_img3 = np.zeros_like(img_gray_threshed, dtype = np.int32)
tmp_img3 = cv2.polylines(tmp_img3, coords, isClosed=True, color=(2550,0,0))
plt.imshow(tmp_img3 + tmp_img, cmap="gray", vmax=1000)

# Assuming you have already obtained the outermost points p1, p2, p3, p4
outermost_points = [p1, p2, p3, p4]

print(outermost_points)
```

*Figure 11: Step 8*

```
valid_angles = [
    calculate_angle(outermost_points[0], outermost_points[1], outermost_points[2]),
    calculate_angle(outermost_points[1], outermost_points[2], outermost_points[3]),
    calculate_angle(outermost_points[2], outermost_points[3], outermost_points[0]),
    calculate_angle(outermost_points[3], outermost_points[0], outermost_points[1])
]

# Plot the original image with intersections and outermost points
plt.imshow(tmp_img, cmap="gray", vmax=1000)

# Highlight the outermost points in red
for point in outermost_points:
    plt.scatter(point[1], point[0], c='red', s=10, marker='o')  # Swap x and y coordinates

# Connect the outermost points with lines in red
lines = np.array([[outermost_points[0], outermost_points[1]],
                  [outermost_points[1], outermost_points[2]],
                  [outermost_points[2], outermost_points[3]],
                  [outermost_points[3], outermost_points[0]]])

for line in lines:
    plt.plot([line[0][1], line[1][1]], [line[0][0], line[1][0]], c='red')

plt.show()
```

```
___________________GETTING OUTER FRAME___________________

[(53, 43), (29, 958), (947, 974), (965, 17)]
```

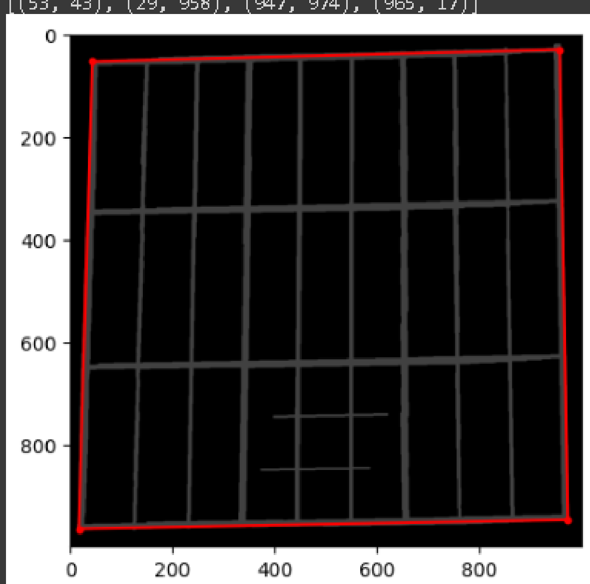


*Figure 12: Step 8.1*

## Perspective transform

This step is basic and simple, we just apply perspective transform in order to transform our view to the image so the tiles of the frame are clear and can be easily extracted.

**Perspective Transform**

```
[ ]
    print("____________________PERSPECTIVE TRANSFORM___________________________ \n")
    y, x = img_gray_threshed.shape
    src_coords = np.float32([[0,0], [x,0], [x,y], [0,y]])
    dst_coords = np.float32([[p1[::-1], p2[::-1], p3[::-1], p4[::-1]]])
    img_gray_threshed_warped = cv2.warpPerspective(
        src=img_gray_threshed,
        M=cv2.getPerspectiveTransform(dst_coords, src_coords),
        dsize=img_gray_threshed.shape[::-1]
    )
    plt.imshow(img_gray_threshed_warped, cmap="gray");
```

```
____________________PERSPECTIVE TRANSFORM________________________
```

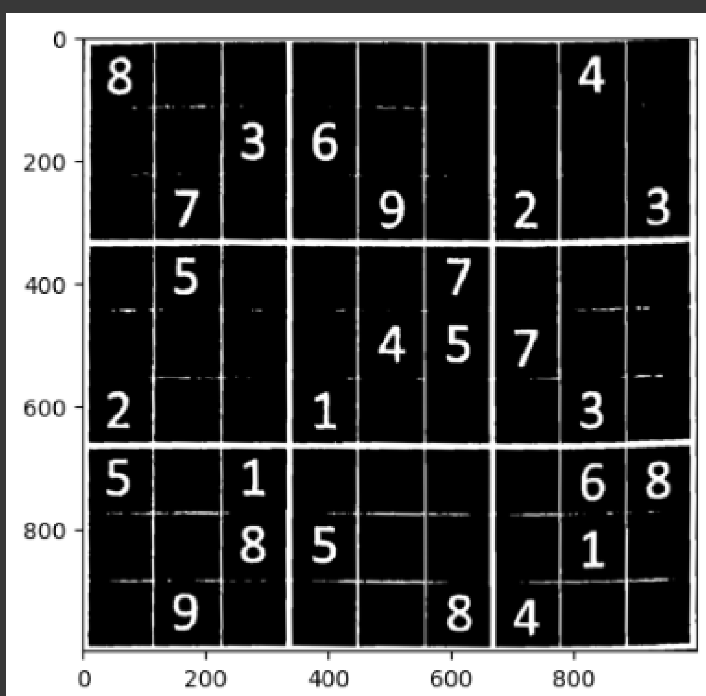


*Figure 13: Step 9*

## Separating tiles

This step is also simple as we just separate the tiles in order to preview each tile with its own number away from other tiles.

**Seperating Lines**

```
[ ]
    print("____________________SEPERATE TILES_________________________ \n")
    M = img_gray_threshed_warped.shape[0] // 9
    N = img_gray_threshed_warped.shape[1] // 9
    number_tiles = []
    for i in range(9):
        number_tiles.append([])
        for j in range(9):
            tile = img_gray_threshed_warped[i*M:(i+1)*M, j*N:(j+1)*N]
            number_tiles[i].append(tile)

    _, axes = plt.subplots(9, 9, figsize=(5, 5))
    for i, row in enumerate(axes):
        for j, col in enumerate(row):
            col.imshow(number_tiles[i][j], cmap="gray");
            col.get_xaxis().set_visible(False)
            col.get_yaxis().set_visible(False)
```

```
__________________SEPERATE TILES_______________________
```

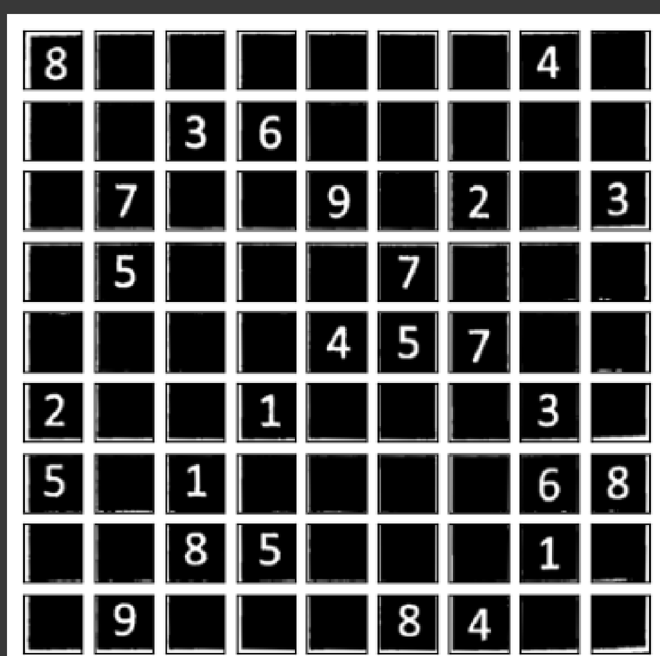

*Figure 14: Step 10*

## Plotting tiles

This is a visualization step as it only plots each tile in its own plot to visualize it.

**Plotting Each Tile Alone**

```
[ ]
    tiles = []

    # Plot each tile individually
    for i in range(9):
        for j in range(9):
            _, ax = plt.subplots(figsize=(1, 1))
            ax.imshow(number_tiles[i][j], cmap="gray")
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)
            plt.title(str(i + j))
            plt.show()

            # Append the subplot to the list
            tiles.append(ax)

    # Now subplot_list contains all the subplot axes
    # You can access them later for further modifications or analysis

    #SIZE IS 111X111
```

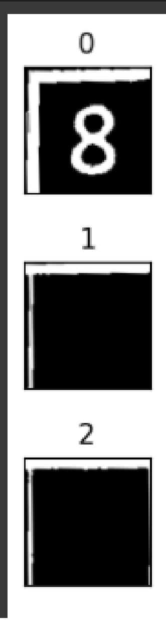


*Figure 15: Step 11*

# Phase 2

This phase mainly includes the major aspect of the project starting with preprocessing the tiles and ending with solving the sudoku board.

## Modifying tiles

In this step we apply a distance threshold on tiles so that any noise or border lines that are introduced in the tiles along that threshold would be eliminated resulting a tile with clear edges.

**Removing Edges**

```
[ ] # List to store modified tiles
    modified_tiles = []

    # Set the distance threshold from the image borders
    distance_threshold = 15

    # Eliminate white pixels within 15 pixels from any border of the image
    for i in range(9):
        for j in range(9):
            # Access the original tile
            current_tile = number_tiles[i][j]

            # Create a mask to identify pixels within the distance threshold from any border
            mask = np.zeros_like(current_tile)
            mask[distance_threshold:-distance_threshold, distance_threshold:-distance_threshold] = 255

            # Apply the mask to the original tile
            modified_tile = cv2.bitwise_and(current_tile, mask)

            # Append the modified tile to the list
            modified_tiles.append(modified_tile)

            # Plot the modified tile
            _, ax = plt.subplots(figsize=(1, 1))
            ax.imshow(modified_tile, cmap="gray")
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)
            plt.title(f'Modified Tile {i + 1}-{j + 1}')
            plt.show()
```


*Figure 16: Step 12*

## Cleaning tiles

In this step we apply Morphloical opening on all tiles in order to remove any small white pixel noise introduced in the tiles.

**Cleaning Tiles (Openning)**

```
[ ] # List to store cleaned tiles
    cleaned_tiles = []

    # Iterate through each filled tile and remove noise
    for index, modified_tile in enumerate(modified_tiles):
        # Create a rectangular kernel for morphological opening
        kernel = np.ones((5, 5), np.uint8) # was 5 5

        # Apply morphological opening to remove noise
        cleaned_tile = cv2.morphologyEx(modified_tile, cv2.MORPH_OPEN, kernel)

        # Append the cleaned tile to the list
        cleaned_tiles.append(cleaned_tile)

        # Plot the cleaned tile
        _, ax = plt.subplots(figsize=(1, 1))
        ax.imshow(cleaned_tile, cmap="gray")
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        plt.title(f'Cleaned Tile {index}')
        plt.show()

    # Now, 'cleaned_tiles' contains all the tiles with noise removed
    # You can further process or analyze these cleaned tiles as needed
```


*Figure 17: Step 13*

## Filling tiles

In this step we apply dilation on tiles in order to fill any broken segments of any number within the tiles.

**Filling Tiles (Dilation)**

```
# List to store smoothed and filled tiles
filled_tiles = []

# Iterate through each smoothed tile and apply morphological closing
for index, cleaned_tile in enumerate(cleaned_tiles):

    kernel = np.ones((5, 5), np.uint8)
    # Apply morphological closing to fill gaps in contours
    closed_tile = cv2.morphologyEx(cleaned_tile, cv2.MORPH_DILATE, kernel)

    # Append the closed tile to the list
    filled_tiles.append(closed_tile)

    # Plot the closed tile
    _, ax = plt.subplots(figsize=(1, 1))
    ax.imshow(closed_tile, cmap="gray")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.title(f'Filled Tile {index}')
    plt.show()
```


*Figure 18: Step 14*

## Opening tiles

In this step we apply opening once again to eliminate any uneliminated noise from previous steps. However, this time we use 2 different structuring elements. We use (9,9) mask for erosion and (7,7) mask for dilation. We noticed that this formation of 2 separate masks leads to a much better results for tiles preprocessing.

**Opened Tiles**

```
# List to store opened tiles
opened_tiles = []

# Iterate through each filled tile and apply morphological opening
for index, filled_tile in enumerate(filled_tiles):


    # Create a rectangular kernel for morphological erode
    erode_kernel = np.ones((9, 9), np.uint8)

    # Apply morphological erode to the filled tile
    eroded_tile = cv2.morphologyEx(filled_tile, cv2.MORPH_ERODE, erode_kernel)

    # Create a rectangular kernel for morphological dilate
    dilate_kernel = np.ones((7, 7), np.uint8)

    # Apply morphological dilate to the eroded tile
    opened_tile = cv2.morphologyEx(eroded_tile, cv2.MORPH_DILATE, dilate_kernel)

    # Append the opened tile to the list
    opened_tiles.append(opened_tile)

    # Plot the opened tile
    _, ax = plt.subplots(figsize=(1, 1))
    ax.imshow(opened_tile, cmap="gray")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.title(f'Opened Tile {index}')
    plt.show()
```


*Figure 19: Step 15*

## Centering tiles

In this step we center the remaining un-noisy number object in the center of tile. This allows us to perform nearly perfect template matching.

Centered Tiles

```python
# List to store centered tiles
centered_tiles = []

# Iterate through each modified tile with bounding boxes and apply contours
for index, opened_tile in enumerate(opened_tiles):
    # Find contours in the thresholded tile
    contours, _ = cv2.findContours(opened_tile, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Extract bounding boxes around each contour
    bounding_boxes = [cv2.boundingRect(contour) for contour in contours]

    # Create a blank canvas of the same size as the input tile
    tile_with_centered_objects = np.zeros_like(opened_tile)

    for x, y, w, h in bounding_boxes:
        # Calculate the centroid of the bounding box
        centroid_x = x + w // 2
        centroid_y = y + h // 2

        # Calculate the offset to move the object to the center of the tile
        offset_x = tile_with_centered_objects.shape[1] // 2 - centroid_x
        offset_y = tile_with_centered_objects.shape[0] // 2 - centroid_y

        # Move the object to the center without duplicating
        tile_with_centered_objects[y + offset_y:y + offset_y + h, x + offset_x:x + offset_x + w] = opened_tile[y:y + h, x:x + w]

    # Append the centered tile to the list
    centered_tiles.append(tile_with_centered_objects)

    # Plot the accumulated result of centered objects in the tile with bounding boxes
    _, ax = plt.subplots(figsize=(1, 1))
    ax.imshow(tile_with_centered_objects, cmap="gray")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.title(f'Tile {index} with Centered Objects')
    plt.show()
```

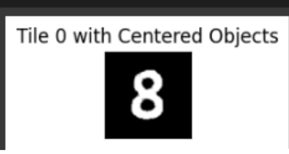


*Figure 20: Step 16*

## Smoothing tiles

In this step we apply gaussian smoothing on all tiles so that they appear smooth, and properly finish the number objects.

**Smoothed Tiles (Gaussian)**

```
[ ] # List to store smoothed tiles
    smoothed_tiles = []

    # Iterate through each centered tile and apply Gaussian blur
    for index, centered_tile in enumerate(centered_tiles):
        # Apply Gaussian blur to the centered tile
        smoothed_tile = cv2.GaussianBlur(centered_tile, (3, 3), 0)

        # Append the smoothed tile to the list
        smoothed_tiles.append(smoothed_tile)

        # Plot the smoothed tile
        _, ax = plt.subplots(figsize=(1, 1))
        ax.imshow(smoothed_tile, cmap="gray")
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        plt.title(f'Smoothed Tile {index}')
        plt.show()
```


*Figure 21: Step 17*

# Detecting numbers (Contours)

In this step we aim to detect numbers using contours in order to visualize them by applying a bounded box surrounding each number.

Detecting Numbers (Contours)

```
# Iterate through each modified tile and apply contours without cropping
for i in range(9):
    for j in range(9):
        # Access the modified tile
        current_tile = smoothed_tiles[i * 9 + j]

        # Apply thresholding to the modified tile
        _, thresholded_tile = cv2.threshold(current_tile, 128, 255, cv2.THRESH_BINARY)

        # Find contours in the thresholded tile
        contours, _ = cv2.findContours(thresholded_tile, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        # Extract bounding boxes around each contour
        bounding_boxes = [cv2.boundingRect(contour) for contour in contours]

        # Display the modified tile with bounding boxes around characters
        tile_with_boxes = current_tile.copy()
        for x, y, w, h in bounding_boxes:
            cv2.rectangle(tile_with_boxes, (x, y), (x + w, y + h), (255, 0, 0), 2)

        # Plot the modified tile with contours
        _, ax = plt.subplots(figsize=(1, 1))
        ax.imshow(tile_with_boxes, cmap="gray")
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        plt.title(f'Modified Tile {i + 1}-{j + 1} with Bounding Boxes')
        plt.show()
```

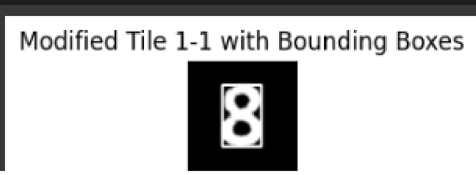


*Figure 22: Step 18*

## Edge detection (Canny)

In this step we apply Canny edge detection to visualize numbers in a better way. We use the red color to emphasize on edges.

**Edge Detection (Canny)**

```python
#Iterate through each smoothed tile and overlay red edges on the original centered tile
for index, smoothed_tile in enumerate(smoothed_tiles):
    # Apply Gaussian blur to the smoothed tile
    smoothed_tile = cv2.GaussianBlur(smoothed_tile, (3, 3), 0)

    # Apply Canny edge detection to get external edges
    edges = cv2.Canny(smoothed_tile, 30, 100)

    # Convert the grayscale centered tile to a three-channel image
    centered_tile_rgb = cv2.cvtColor(centered_tiles[index], cv2.COLOR_GRAY2RGB)

    # Add red edges to the three-channel image
    centered_tile_rgb[edges != 0] = [255, 0, 0]  # Red color for edges

    # Plot the original centered tile with red edges
    _, ax = plt.subplots(figsize=(1, 1))
    ax.imshow(centered_tile_rgb)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.title(f'Number with Red Edges {index}')
    plt.show()
```

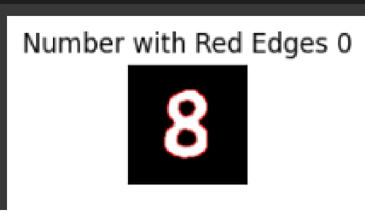


*Figure 23: Step 19*

## Thinning numbers

In this step we thin numbers so that we can represent them as thin lines of 1-pixels.

**Thinning Numbers**

```
# Function to extract the centerline of the digit
def extract_centerline(number_image):
    # Apply skeletonization to obtain the centerline
    skeleton = cv2.ximgproc.thinning(number_image)
    return skeleton

# List to store centerline images
thin_tiles = []

# Iterate through each smoothed tile and extract the centerline
for index, smoothed_tile in enumerate(smoothed_tiles):
    # Extract the centerline
    centerline = extract_centerline(smoothed_tile)

    # Append the centerline image to the list
    thin_tiles.append(centerline)

    # Display the result
    _, ax = plt.subplots(figsize=(1, 1))
    ax.imshow(centerline, cmap='gray')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.title(f'Centerline of Number {index}')
    plt.show()
```

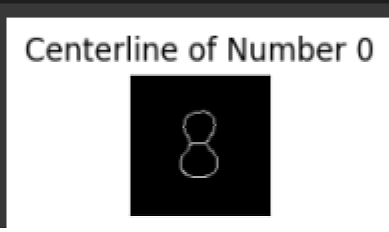


*Figure 24: Step 20*

# OCR

This is the final and the major stage of the computer vision major task (phase 2). After saving templates for each number starting off 1 to 9, we want to perform template matching using OpenCV library.

After all preprocessing done and visualizing all parameters and aspects of tiles are fixed for all testcases.

Therefore, now we can apply OCR using templates.

The templates folder is divided into 9 sub-folders named from 1 to 9. Each sub-folder contains templates of its own number.

The folder name acts as the label of the template matching process.

The process flows as follows:

1. **Creating empty 9x9 np (numpy) array**
2. **Isolating each tile**
3. **Iterate over temples folder**
4. **Iterate over templates in each sub-folder**
5. **Store the match percentage with each template**
6. **Obtain the highest match percentage acquired from all templates in all folders.**
7. **Storing the folder name (digit) as the number corresponding to the tile location in the sudoku frame.**

## OCR

```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Create a 9x9 empty matrix
SudukoBoard = np.zeros((9, 9), dtype=int)
print("here is the Sudoko board before OCR: \n")
print(SudukoBoard)
print("\n")
folder_path = "/content/templates"
for index, smoothed_tile in enumerate(smoothed_tiles):
    # Variables to store information about the best match
    best_match = None
    best_templateDir = None
    best_folder_number = None
    # Iterate over subfolders
    for folder_number in range(1, 10):
        folder_name = str(folder_number)
        folder_path_with_number = os.path.join(folder_path, folder_name)
        # Check if the folder exists
        if os.path.exists(folder_path_with_number):
            # Iterate over files in the folder
            for file_name in os.listdir(folder_path_with_number):
                # Check if the file has a '.png' extension
                if file_name.endswith('.png'):
                    templateDir = os.path.join(folder_path_with_number, file_name)
                    template = cv2.imread(templateDir, 0)
                    # Apply template matching
                    result = cv2.matchTemplate(smoothed_tile, template, cv2.TM_CCOEFF_NORMED)
                    _, max_val, _, _ = cv2.minMaxLoc(result)
                    threshold = 0.8
                    if max_val > threshold and (best_match is None or max_val > best_match):
                        # Update the best match information
                        best_match = max_val
                        best_templateDir = templateDir
                        best_folder_number = folder_number
    # Check if a match was found
    if best_match is not None:
        print(f'Tile {index}: Recognized digit. Max value is: {best_match}, Folder number is: {best_folder_number}')
        # Update the Sudoku board with the recognized digit
        SudukoBoard[index // 9, index % 9] = best_folder_number
    else:
        print(f'Tile {index}: No digit recognized')
print(" \n Sudoku Board after OCR:\n", SudukoBoard)
```

*Figure 25: Step 21*

```
here is the Sudoko board before OCR:

[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]


Tile 0: Recognized digit. Max value is: 0.9785306453704834, Folder number is: 8
Tile 1: No digit recognized
Tile 2: No digit recognized
Tile 3: No digit recognized
Tile 4: No digit recognized
Tile 5: No digit recognized
Tile 6: No digit recognized
Tile 7: Recognized digit. Max value is: 1.0, Folder number is: 4
Tile 8: No digit recognized
Tile 9: No digit recognized
Tile 10: No digit recognized
Tile 11: Recognized digit. Max value is: 0.9693885445594788, Folder number is: 3
Tile 12: Recognized digit. Max value is: 0.9697750210762024, Folder number is: 6
Tile 13: No digit recognized
Tile 14: No digit recognized
Tile 15: No digit recognized
Tile 16: No digit recognized
Tile 17: No digit recognized
Tile 18: No digit recognized
Tile 19: Recognized digit. Max value is: 0.9719265699386597, Folder number is: 7
```

*Figure 26: Step 21.1*

```
Tile 70: Recognized digit. Max value is: 1.0, Folder number is: 1
Tile 71: No digit recognized
Tile 72: No digit recognized
Tile 73: Recognized digit. Max value is: 0.9847251176834106, Folder number is: 9
Tile 74: No digit recognized
Tile 75: No digit recognized
Tile 76: No digit recognized
Tile 77: Recognized digit. Max value is: 0.9737653732299805, Folder number is: 8
Tile 78: Recognized digit. Max value is: 0.9655701518058777, Folder number is: 4
Tile 79: No digit recognized
Tile 80: No digit recognized

 Sudoku Board after OCR:
 [[8 0 0 0 0 0 0 4 0]
 [0 0 3 6 0 0 0 0 0]
 [0 7 0 0 9 0 2 0 3]
 [0 5 0 0 0 7 0 0 0]
 [0 0 0 0 4 5 7 0 0]
 [2 0 0 1 0 0 0 3 0]
 [5 0 1 0 0 0 0 6 8]
 [0 0 8 5 0 0 0 1 0]
 [0 9 0 0 0 8 4 0 0]]
```

*Figure 27: Step 21.2*

## Solving sudoku board

This is the final step of the major task which is solving the sudoku board.

Solving Sudoko Board

```
def is_valid(board, row, col, num):
    # Check if 'num' is not present in the current row, column, and 3x3 subgrid
    return (
        not np.any(board[row, :] == num) and
        not np.any(board[:, col] == num) and
        not np.any(board[(row//3)*3:(row//3)*3+3, (col//3)*3:(col//3)*3+3] == num)
    )

def find_empty_location(board):
    # Find the first empty location in the board
    for i in range(9):
        for j in range(9):
            if board[i, j] == 0:
                return i, j
    return -1, -1  # If no empty location is found
def solve_sudoku(board):
    row, col = find_empty_location(board)

    if (row, col) == (-1, -1):
        # If no empty location is found, the board is solved
        return True
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            # Try placing the current number
            board[row, col] = num

            # Recursively try to solve the rest of the board
            if solve_sudoku(board):
                return True  # The board is solved

            # If placing the current number doesn't lead to a solution, backtrack
            board[row, col] = 0

    # No number from 1 to 9 can be placed at this location
    return False
# Example usage:
# Assuming SudukoBoard is the 9x9 matrix you obtained after OCR
solved_board = SudukoBoard.copy()  # Create a copy to avoid modifying the original board
if solve_sudoku(solved_board):
    print("Sudoku board is solved:")
    print(solved_board)
else:
    print("No solution exists for the given Sudoku board.")
```

*Figure 28: Step 22*

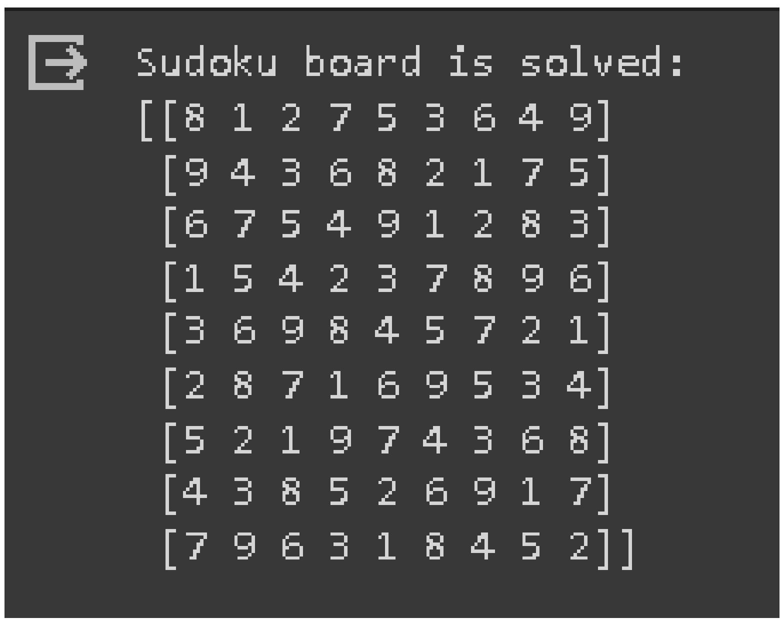


***Figure 29: Step 22.1***