AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

CREDIT HOURS ENGINEERING PROGRAMS

Sudoku Extractor and Solver

(CSE483) Computer Vision

Submitted to:

Dr. Mahmoud Ibrahim Khalil

Eng. Ahmad Salama Abdelaziz

Submitted by:

Ahmed Sameh Saad Elsherefy	20P8929
Mohamed Amr Abdelkhaleq	20P1643
Adham Haythem Mohamed	20P1852
Youssef Walid Hamed	20P1859

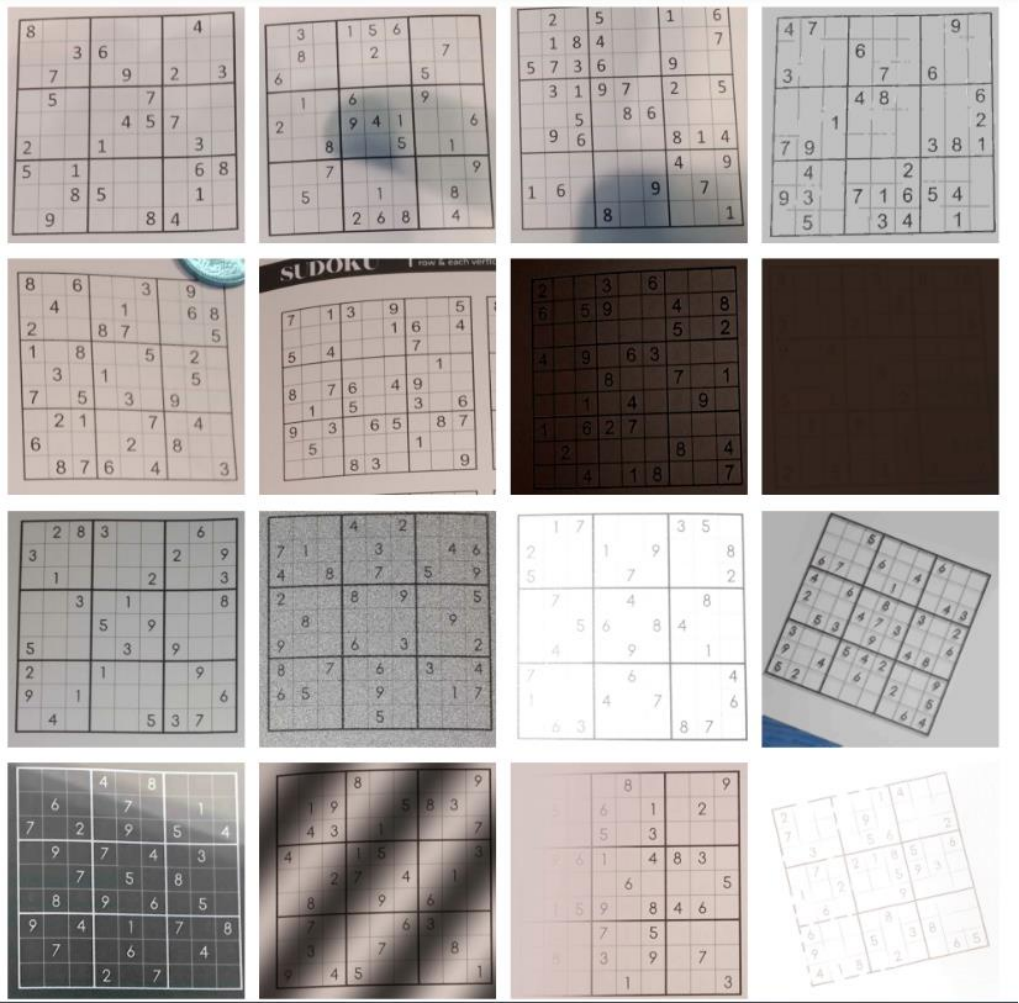
Project Repo

Load Images:

Code:

```
imgs = []
imgs.append(cv2.imread('01-Normal.jpg'))
imgs.append(cv2.imread('02-TheMightyFinger.jpg'))
imgs.append(cv2.imread('03-WhereBorder.jpg'))
imgs.append(cv2.imread('04-CompressoEspresso.jpg'))
imgs.append(cv2.imread('05-Ya3amSheel2elCoinYa3am.jpg'))
imgs.append(cv2.imread('06-FarAndCurved.jpg'))
imgs.append(cv2.imread('07-2elNoor2ata3.jpg'))
imgs.append(cv2.imread('08-MeshShayef7agaYa3am.jpg'))
imgs.append(cv2.imread('09-Normal2.jpg'))
imgs.append(cv2.imread('10-Mal7wFelfel.jpg'))
imgs.append(cv2.imread('11-FlashBang.jpg'))
imgs.append(cv2.imread('12-BrokenPrinter.jpg'))
imgs.append(cv2.imread('13-DarkMode.jpg'))
imgs.append(cv2.imread('14-Sine.jpg'))
imgs.append(cv2.imread('15-GoneWithTheWind.jpg'))
imgs.append(cv2.imread('16-SomethingWentTerriblyWrongHere.jpg'))

NUM_IMAGES = 16
NUM_ROWS_PLOT = 4
NUM_COLS_PLOT = 4

fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    if i < NUM_IMAGES:
        axes[i].imshow(imgs[i], cmap='gray')
        axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output:



Explanation:

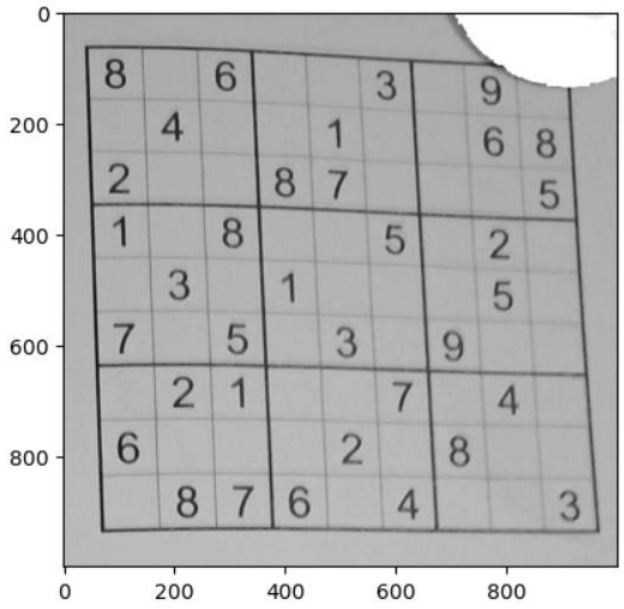
First in this cell after importing the required libraries, we load the test case images from our disk. We then specify NUM_IMAGES as 16 to use it later when displaying the processed images. We will view the images in 4 rows and 4 columns using matplotlib library as shown in the code.

Remove Yellow Coin:

Code Part 1:

```
lowerValues = np.array([10, 50, 70])
upperValues = np.array([35, 255, 255])
imgsWithoutCoins = []
imgIndexsWithCoins = []
for i in range(0, NUM_IMAGES):
    grayscaleImage = cv2.cvtColor(imgs[i], cv2.COLOR_BGR2GRAY)
    hsvImage = cv2.cvtColor(imgs[i], cv2.COLOR_BGR2HSV)
    coinMask = cv2.inRange(hsvImage, lowerValues, upperValues)

    # Increase the size of the mask using dilation
    dilateKernel = np.ones((15, 15), np.uint8)
    coinMask = cv2.dilate(coinMask, dilateKernel, iterations=1)

    morphKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (9, 9))
    coinMask = cv2.morphologyEx(coinMask, cv2.MORPH_CLOSE, morphKernel, None, None, 1, cv2.BORDER_REFLECT101)

    if cv2.countNonZero(coinMask) > 30000:
        imgIndexsWithCoins.append(i)
        imgWithoutCoins = cv2.add(grayscaleImage, coinMask)
    else:
        # If the coin mask is smaller than the threshold, use the original grayscale image
        imgWithoutCoins = grayscaleImage

    imgsWithoutCoins.append(imgWithoutCoins)

plt.imshow(imgsWithoutCoins[4], cmap='gray')
plt.show()
```

Output Part 1:



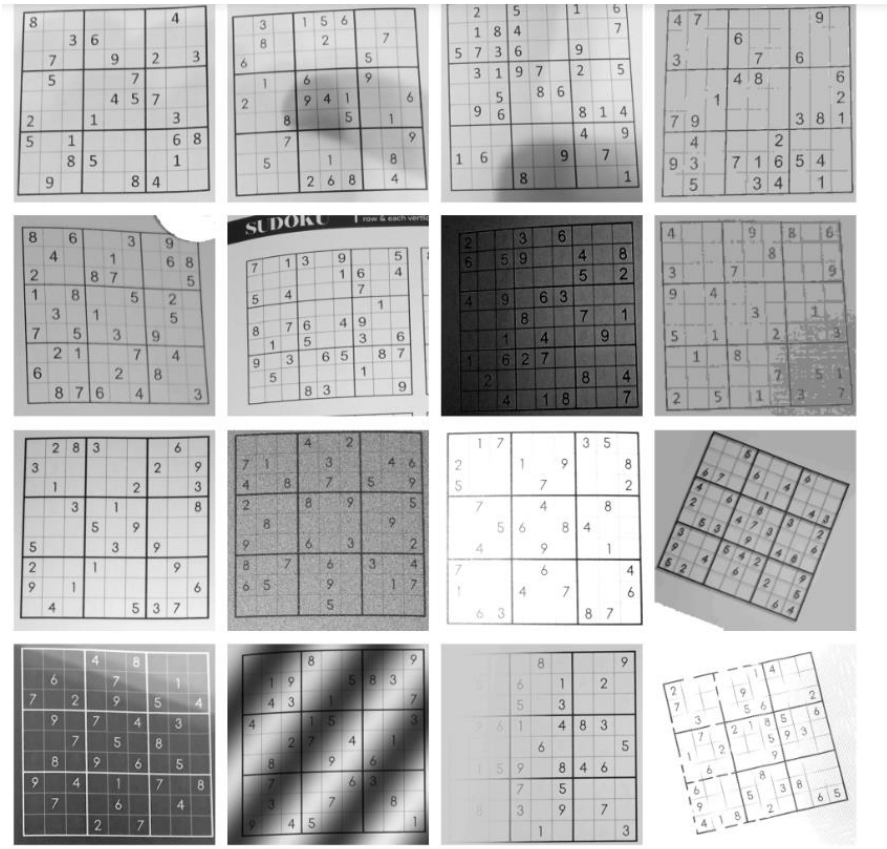
Explanation Part 1:

Here in this section we want to remove the yellow coin object from the border of the image to be processed as background. First we specify the color values of the coin as HSV values, then we loop on all of our test case images to see if the predefined HSV values occur or not. If found, we will use morphological close operation to remove the coin from the image and append it in the list of imgWithoutCoins, we also save the number of the image that contains yellow object to be used later.

Code Part 2:

```
grayImgs = []
zeroImgs = []
for i in range(0,NUM_IMAGES):

  if len(imgsWithoutCoins[i].shape) == 2:  # Single-channel image
      grayImgs.append(imgsWithoutCoins[i])
  else:  # Multi-channel image, convert to grayscale
      grayImgs.append(cv2.cvtColor(imgsWithoutCoins[i], cv2.COLOR_BGR2GRAY))

  zeroImgs.append(np.zeros((grayImgs[i].shape),np.uint8))

fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    if i < NUM_IMAGES:
        axes[i].imshow(grayImgs[i], cmap='gray')
        axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output Part 2:



Explanation Part 2:

After removing the yellow objects from the images we then convert the images to grayscale image using cv2.cvtColor function in opencv2. The grayscale images is then appended to grayImgs list. A copy of the image is taken in a numpy array in zeroImgs list.

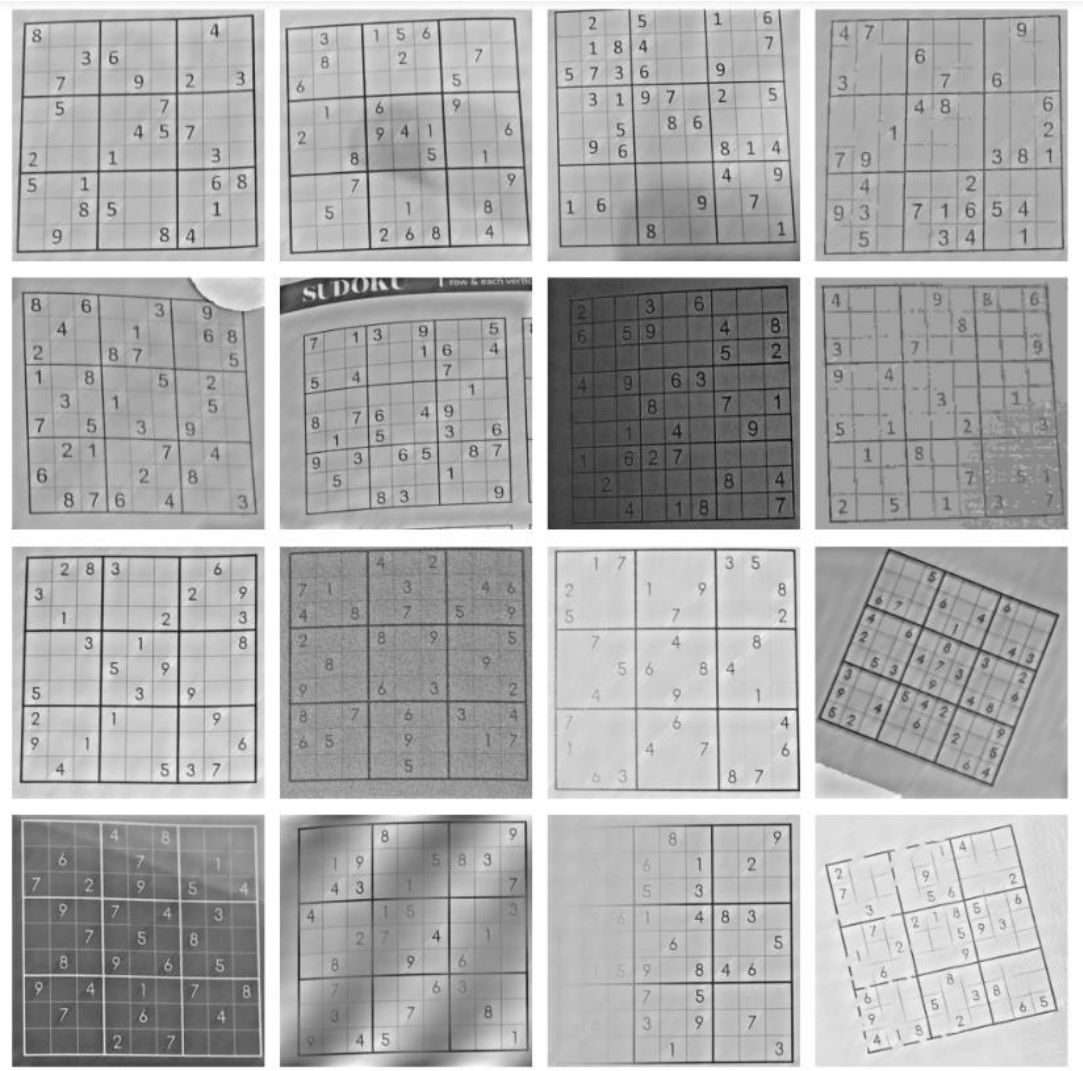
Removing Periodic Noise:

Code:

```
def distance(point1, point2):
    return sqrt((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2)


def butterworthLP(D0, imgShape, n):
    base = np.zeros(imgShape[:2])
    rows, cols = imgShape[:2]
    center = (rows / 2, cols / 2)
    for x in range(cols):
        for y in range(rows):
            base[y, x] = 1 / (1 + (distance((y, x), center) / D0) ** (2 * n))
    return base

frequencyFilteredImgs = []
for imgNum in range(0,NUM_IMAGES):
    fourier_transform = np.fft.fft2(grayImgs[imgNum])
    center_shift = np.fft.fftshift(fourier_transform)

    fourier_noisy = 20 * np.log(np.abs(center_shift))

    rows, cols = grayImgs[imgNum].shape
    crow, ccol = rows // 2, cols // 2

    for x in range(0, rows):
      for y in range(0, cols):
        if (x == y):
          for i in range(0, 10):
            center_shift[x - i, y] = 1


    filtered = center_shift * butterworthLP(80, grayImgs[imgNum].shape, 10)
```

```
    filtered = center_shift * butterworthLP(80, grayImgs[imgNum].shape, 10)

    f_shift = np.fft.ifftshift(center_shift)
    denoised_image = np.fft.ifft2(f_shift)
    frequencyFilteredImgs.append(np.real(denoised_image))



min_value = np.min([np.min(array) for array in frequencyFilteredImgs])
max_value = np.max([np.max(array) for array in frequencyFilteredImgs])

# Scale and convert each array to uint8
frequencyFilteredImgs = [(array - min_value) / (max_value - min_value) * 255 for array in frequencyFilteredImgs]
frequencyFilteredImgs = [np.round(scaled_array).astype(np.uint8) for scaled_array in frequencyFilteredImgs]


fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    if i < NUM_IMAGES:
        axes[i].imshow(frequencyFilteredImgs[i], cmap='gray')
        axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output:



Explanation:

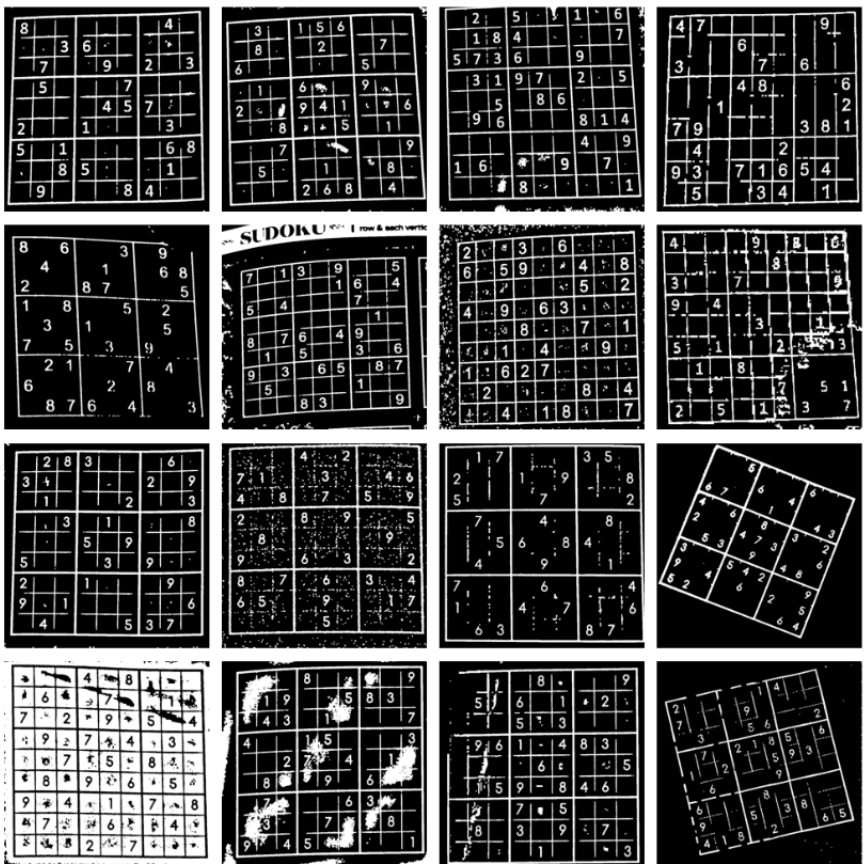
In this section we perform Fourier transform to remove periodic noise from the grayImgs list. The code defines a function distance(point1, point2) to calculate the Euclidean distance between two points in a 2D space. Another function, butterworthLP, generates a Butterworth low-pass filter in the frequency domain based on specified parameters. For each image, It computes the 2D Fourier Transform, shifts the zero-frequency component to the center of the spectrum, and then applies the Butterworth low-pass filter to retain low-frequency components. After that it inverse shifts the modified spectrum, and then computes the inverse Fourier Transform to obtain a denoised image. The frequency filtered image is then added to frequencyFilteredImgs list to be used later.

Preprocessing:

Code:

```
blurImgs = []
threshImgs = []
threshImgsWithoutDilate = []
for i in range(NUM_IMAGES):
  blurImgs.append(cv2.GaussianBlur(grayImgs[i],(5,5),0))
  if i in imgIndexsWithCoins:
      _, thresh_img = cv2.threshold(blurImgs[i], 125, 255, cv2.THRESH_BINARY_INV)
      kernel2 = np.ones((15, 15), np.uint8)
      kernel3 = np.ones((3, 3), np.uint8)
      threshImgs.append(cv2.dilate(thresh_img, kernel2, iterations=1))
      threshImgsWithoutDilate.append(cv2.dilate(thresh_img, kernel3, iterations=1))
  else:
      threshImgs.append(cv2.adaptiveThreshold(blurImgs[i],255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,131,1))
      threshImgsWithoutDilate.append(cv2.adaptiveThreshold(blurImgs[i],255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,1

kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(11,11))
closedImgs = []
for i in range(NUM_IMAGES):
  close = cv2.morphologyEx(threshImgs[i], cv2.MORPH_CLOSE, kernel)
  div = np.float32(threshImgs[i])/(close)
  closedImgs.append(np.uint8(cv2.normalize(div,div,0,255,cv2.NORM_MINMAX)))

  close = cv2.morphologyEx(threshImgsWithoutDilate[i], cv2.MORPH_CLOSE, kernel)
  div = np.float32(threshImgsWithoutDilate[i])/(close)
  threshImgsWithoutDilate[i] = np.uint8(cv2.normalize(div,div,0,255,cv2.NORM_MINMAX))

  kernel = np.ones((5, 5), np.uint8)
  closedImgs[i] = cv2.erode(closedImgs[i], kernel, iterations=1)
  closedImgs[i] = cv2.dilate(closedImgs[i], kernel, iterations=1)

  kernel = np.ones((5, 5), np.uint8)
  threshImgsWithoutDilate[i] = cv2.erode(threshImgsWithoutDilate[i], kernel, iterations=1)
  threshImgsWithoutDilate[i] = cv2.dilate(threshImgsWithoutDilate[i], kernel, iterations=1)

fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    if i < NUM_IMAGES:
        axes[i].imshow(threshImgsWithoutDilate[i], cmap='gray')
        axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output:



Explanation:

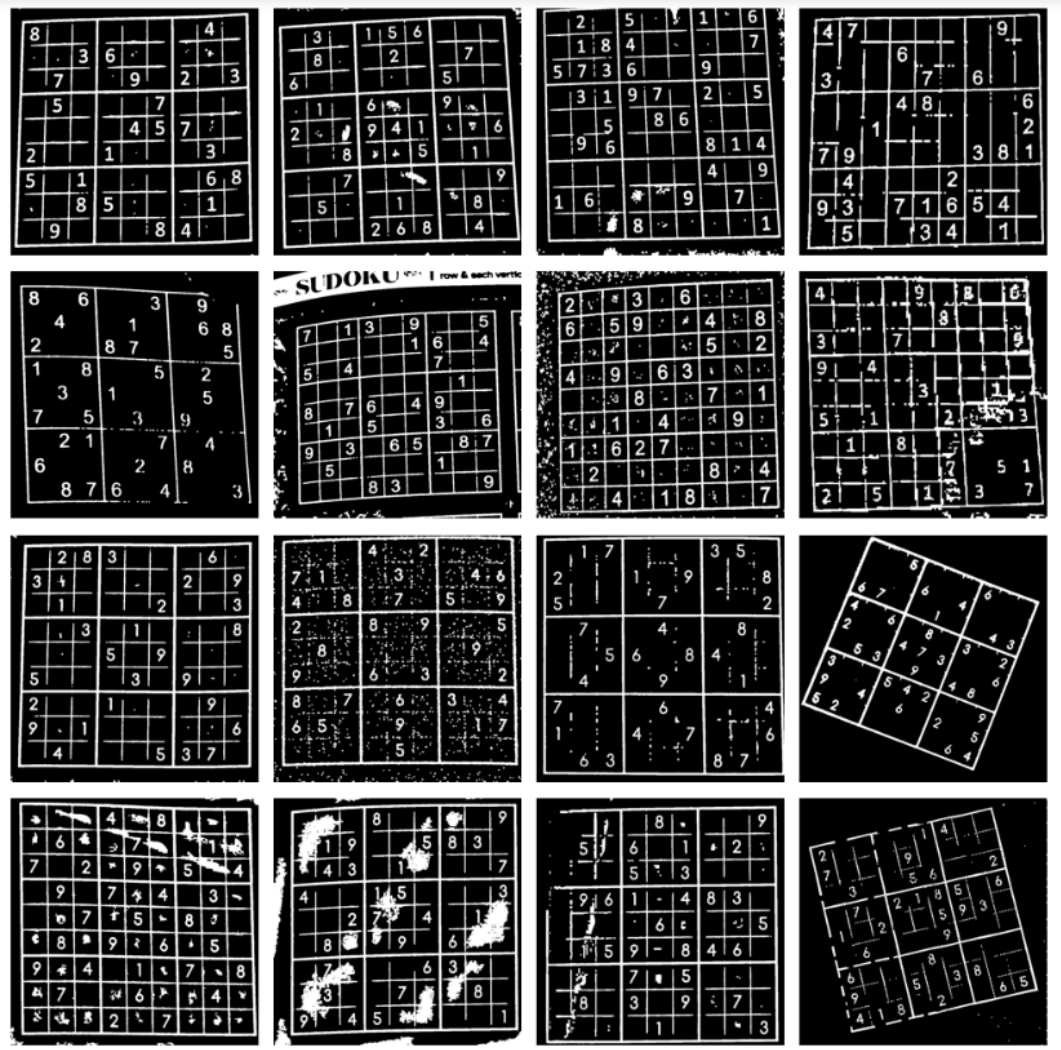
First, we initialize 3 lists blurImgs, threshImgs, and threshImgsWithoutDilate. We blur the images using gaussian blur with 5*5 filter and then append it to blurImgs list. If the image had a yellow object we will dilate the image and store the dilated image in the threshImgsWithoutDilate, if it didn't have a yellow object it goes into the else branch and an adaptive threshold is done then added to the list. Then we apply close as a morphological operation to the two lists the threshImgsWithoutDilate and threshImgs to be used later. We also apply erosion then dilation to all images in the list.

Invert Certain Images:

Code:

```
for i in range(0,NUM_IMAGES):
    _, binary_mask = cv2.threshold(closedImgs[i], 128, 255, cv2.THRESH_BINARY)

    # Count the number of white and black pixels in the binary mask
    num_white_pixels = np.sum(binary_mask == 255)
    num_black_pixels = np.sum(binary_mask == 0)
    # Check if dark pixels are more than white pixels
    if num_black_pixels < num_white_pixels:
        closedImgs[i] = cv2.bitwise_not(closedImgs[i])
        frequencyFilteredImgs[i] = cv2.bitwise_not(frequencyFilteredImgs[i])
        threshImgsWithoutDilate[i] = cv2.bitwise_not(threshImgsWithoutDilate[i])

    if num_black_pixels/num_white_pixels < 3.8:
        num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(closedImgs[i], connectivity=8)
        filtered_mask = np.zeros_like(closedImgs[i])
        filtered_mask2 = np.zeros_like(threshImgsWithoutDilate[i])
        min_component_size = 300

        for label in range(1, num_labels):
                # Check the size of the connected component
                if stats[label, cv2.CC_STAT_AREA] >= min_component_size:
                    # Retain the pixels belonging to this connected component
                    filtered_mask[labels == label] = 255
                    filtered_mask2[labels == label] = 255

        closedImgs[i] = cv2.bitwise_and(closedImgs[i], closedImgs[i], mask=filtered_mask)

        threshImgsWithoutDilate[i] = cv2.bitwise_and(threshImgsWithoutDilate[i], threshImgsWithoutDilate[i], mask=filtered_mask2)


fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    if i < NUM_IMAGES:
        axes[i].imshow(threshImgsWithoutDilate[i], cmap='gray')
        axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output:



Explanation:

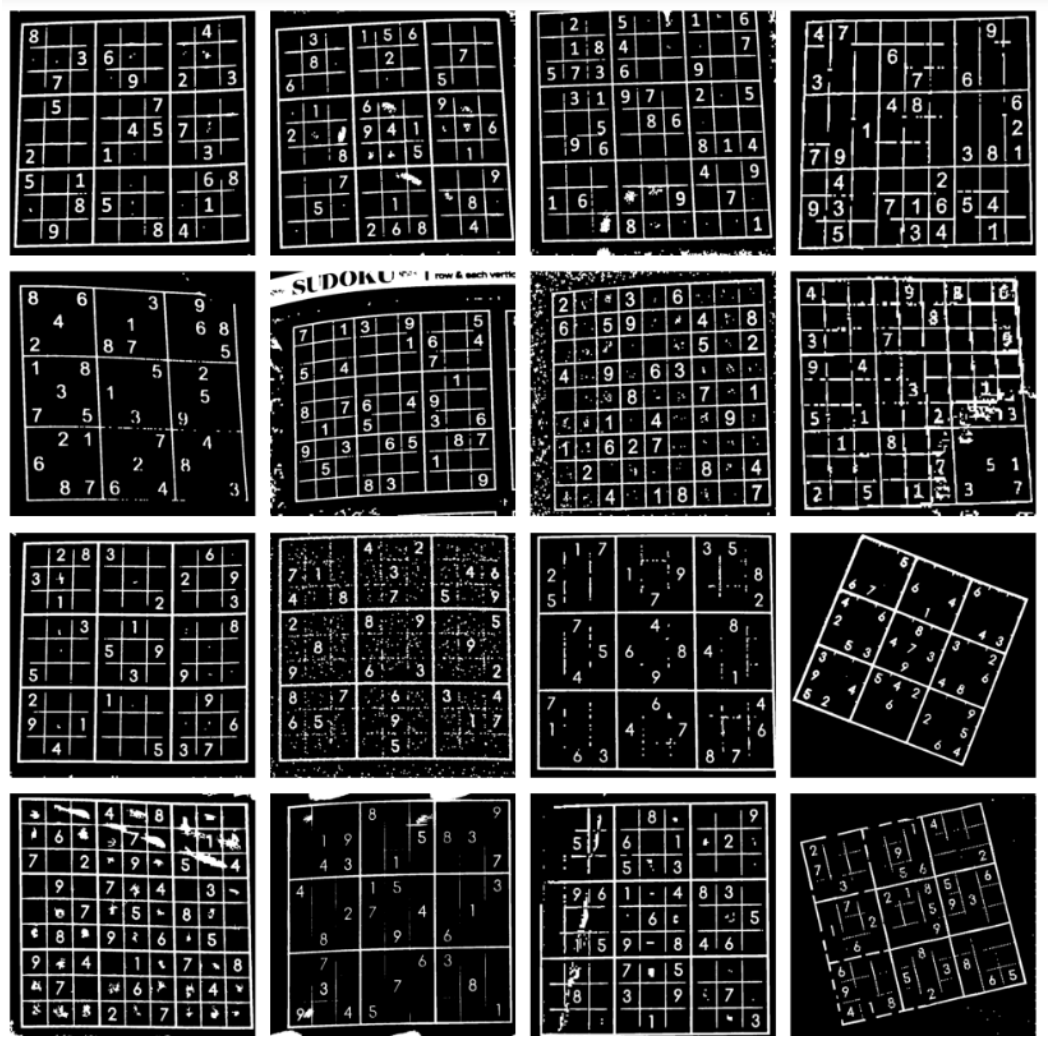
In this section we aim to invert the images that have white pixels count bigger than the black pixels count. We do that by counting the number of black and white pixels using np.sum function. If the number of black is less than white, we invert the cimage with cv2.bitwise_not function. If the ratio of black pixels in respective to the white pixels is bigger than a certain threshold, that means that the images has a very aggressive salt and pepper noise. To solve that we use the connected components method with the min_component_size equal to 300 pixels to remove the noise. After filtering the noise the image is added to the lists we created earlier. Observe picture number 12 as it got inverted, also pictures number 9 and 6 was denoised heavily by removing salt and pepper noise.

Apply Frequency Filter:

Code:

```
maxFilteredImageI = -1
maxFilteredImage = None
maxValue = -1
for i in range(0,NUM_IMAGES):
    num_black_pixels_grey_image = np.sum(closedImgs[i] == 0)
    _, thresh_img = cv2.threshold(frequencyFilteredImgs[i], 113, 255, cv2.THRESH_BINARY_INV)
    num_black_pixels_frequency_image = np.sum(thresh_img == 0)
    value = num_black_pixels_frequency_image/num_black_pixels_grey_image
    if value > maxValue:
        maxValue = value
        maxFilteredImage = thresh_img
        maxFilteredImageI = i
closedImgs[maxFilteredImageI] = maxFilteredImage
threshImgsWithoutDilate[maxFilteredImageI] = maxFilteredImage
```

Output:



Explanation:

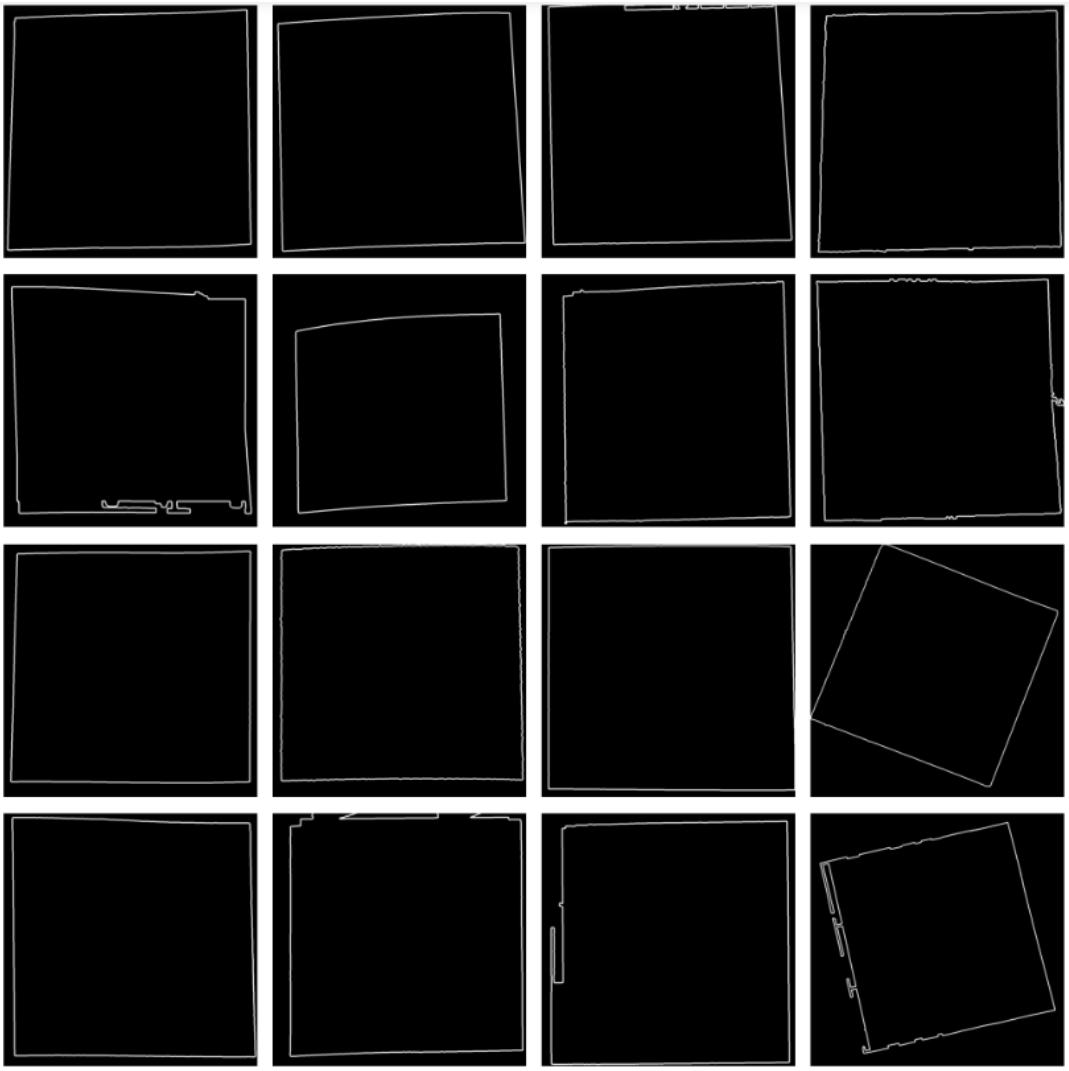
This is the last step of preprocessing before moving on to finding the sudoku grid. In this step we aim to apply frequency filter we created earlier to the images that need that only. We receive the ratio of black pixels before doing frequency transform to the number of black pixels after doing the frequency transform. We only apply Fourier transform to the biggest ratio image, that is done by looping on all the images and getting the image with the maximum ratio we explained earlier. Observe image number 13 after removing the periodic noise.

Contours:

Code:

```
contourImgs = []
bestCnts = []
finalContourImgs = []
finalBestCnts = []
rotatedRects = []

for i in range(NUM_IMAGES):
    contourImgs.append(np.zeros((closedImgs[i].shape), np.uint8))
    contour, hier = cv2.findContours(closedImgs[i], cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    max_area = 0
    bestCnts.append(None)

    for cnt in contour:
        area = cv2.contourArea(cnt)
        if area > max_area:
            max_area = area
            bestCnts[i] = cnt
        cv2.drawContours(contourImgs[i], [cnt], 0, 255, 2)

    if max_area < 400000:
      # Do some operations before getting contour as there are probably gaps in the image
      median = cv2.medianBlur(grayImgs[i], 9)
      cannyImg = cv2.Canny(median, 50, 150)
      kernel2 = np.ones((15, 15), np.uint8)
      dilated_edges = cv2.dilate(cannyImg, kernel2, iterations=4)
      eroded_edges = cv2.erode(dilated_edges, kernel2, iterations=2)
      contour, hier = cv2.findContours(eroded_edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
      max_area = 0
      for cnt in contour:
          area = cv2.contourArea(cnt)
          if area > max_area:
              max_area = area
              bestCnts[i] = cnt

    # Store rotated rectangle in the array
    if bestCnts[i] is not None:
        epsilon = 0.01 * cv2.arcLength(bestCnts[i], True)
        approx = cv2.approxPolyDP(bestCnts[i], epsilon, True)

        rotated_rect = cv2.minAreaRect(approx)
        rect_center, rect_size, rect_angle = rotated_rect
        rect_size = (rect_size[0] - 2 * 25, rect_size[1] - 2 * 50)
        #rect_center = (rect_center[0] - 10, rect_center[1] + 5)
        rect_size = (rect_size[0] + 5, rect_size[1] + 5)
        rotated_rect = (rect_center, rect_size, rect_angle)
        rotatedRects.append(rotated_rect)

    # Draw rotated rectangle
    if bestCnts[i] is not None:
        box = np.int0(cv2.boxPoints(rotated_rect))
        cv2.drawContours(contourImgs[i], [box], 0, 255, 2)
        finalContourImgs.append(np.zeros((closedImgs[i].shape), np.uint8))
        contourFinal, hierFinal = cv2.findContours(contourImgs[i], cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        max_areaFinal = 0
        finalBestCnts.append(None)

        for cnt2 in contourFinal:
            areaFinal = cv2.contourArea(cnt2)
            if areaFinal > max_areaFinal:
                max_areaFinal = areaFinal
                finalBestCnts[i] = cnt2
        cv2.drawContours(finalContourImgs[i],[finalBestCnts[i]],0,255,3)


fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    if i < NUM_IMAGES:
        axes[i].imshow(finalContourImgs[i], cmap='gray')
        axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output:



Explanation:

In this section we aim to find the sudoku grid of the image using the best contour method to prepare for perspective transformation. First we initialize lists contourImgs, bestCnts, finalContourImgs, finalBestCnts, rotatedRects as empty lists. We loop on all the test case images beginning with appending the shape of the image with pixels zero to the contourImgs list. Then we use the cv2.findContours method on closedImgs image, after that we loop on all of the resulted contours to get the best contour and draw it on contourImgs image that we appended earlier.

If the maximum area of the contour is less than a certain number that means the contour did not correctly predict the sudoku grid, so we need to process the image more before retrying to draw the contour. First we apply median blur with 9*9 filter then we apply Canny edge detector to highlight the edges of the image then we dilate those edges with 15*15 filter to further increase the chance of the contour being drawn correctly. After eroding the image with the same filter but with a smaller number of iterations, we then redraw the contour and overwrite the max_area and bestCnt values.

Then we need to ensure if the image is rotated that the contour is drawn properly, that is done by the next if statement by using the function cv2.arcLength to calculate the epsilon value and cv2.approxPolyDP to calculate approx that will be used later to get the area of the rotated rectangle using cv2.minAreaRect function. We then adjust the coordinates of the rotation and the center of the rotation by playing with the numbers in the following lines. After that, the rotated rectangle will be appended to the rotatedRects list.

After that we redraw the contour on top of the previously drawn rectangles to correct the errors. First we draw the contours using cv2.drawContours function as explained earlier. Then we try to get the best contour similar to the code explained earlier. Finally, the best contour is appended to the finalContourImgs list and maximum area is appended to max_areaFinal to be used later.

Perspective Transform:

Code:

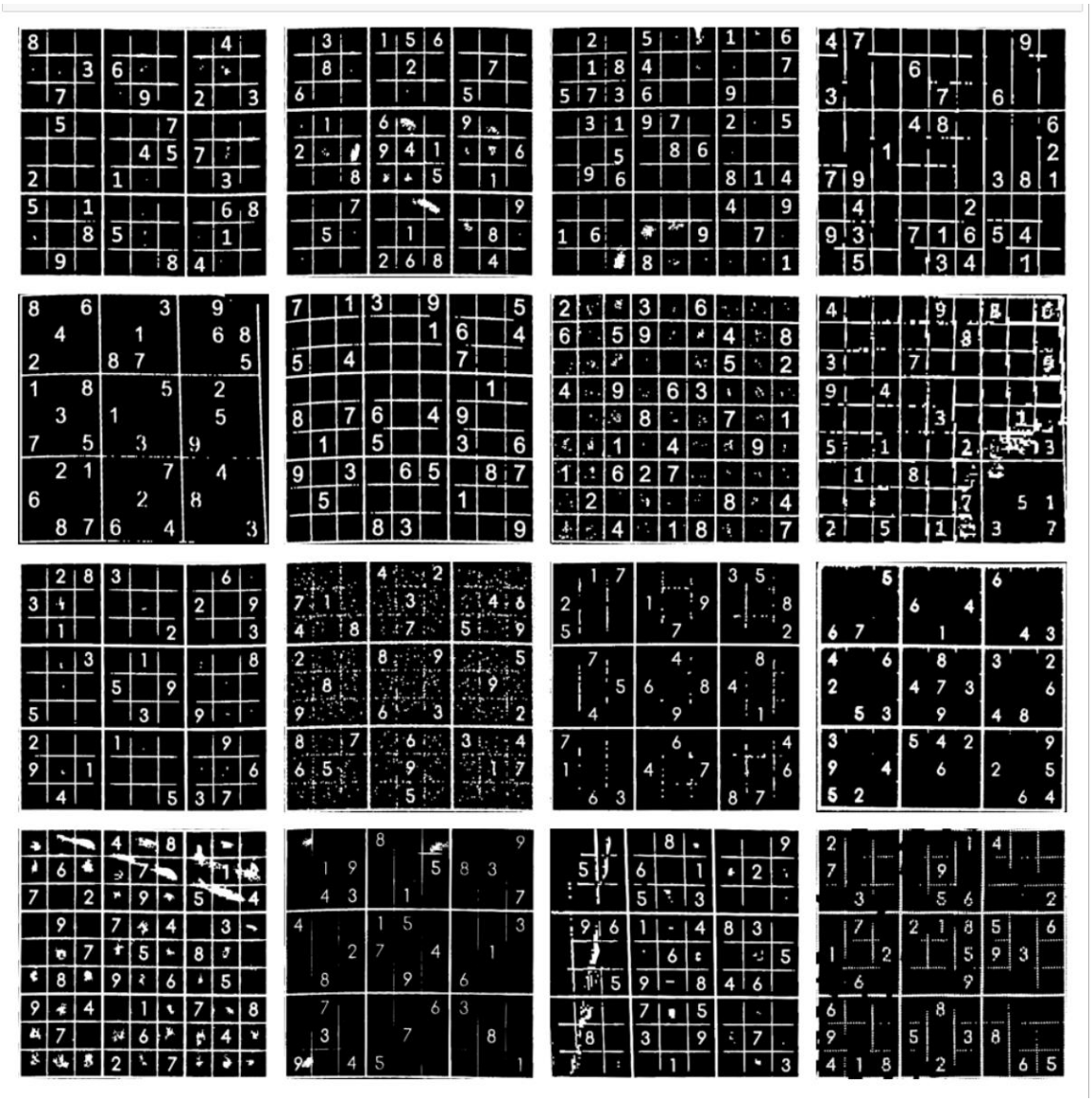
Presepctive Transform

In [36]:
```
perspectiveImgs = []
for i in range(NUM_IMAGES):
  try:
    invertedImg = cv2.bitwise_not(threshImgsWithoutDilate[i])
    matrix = cv2.getPerspectiveTransform(pts1[i],pts2[i])
    perspectiveImgs.append(cv2.warpPerspective(threshImgsWithoutDilate[i],matrix,(width,height)))
  except:
    perspectiveImgs.append(0)

fig, axes = plt.subplots(NUM_ROWS_PLOT, NUM_COLS_PLOT, figsize=(10, 10))
axes = axes.flatten()
for i in range(NUM_IMAGES):
    try:
      if i < NUM_IMAGES:
          axes[i].imshow(perspectiveImgs[i], cmap='gray')
          axes[i].axis('off')
    except:
      pass
plt.tight_layout()
plt.show()
```

Explanation:

This cell here carries out perespective transform on the images so we have a clear view of all the sudoku grids and be able to extract each tile from the grid so they can be later passed on to the OCR where each tile is detected whether it is a number or not. We call the .getPerspectiveTransform function on the sorted points outputted from the cells before and then call .warpPerspective function on both (threshImgsWithoudDilate) list, the matrix outputted from the previous line and the width and height of the images to get the final image ready for tile splitting Here we have the suduko grids after the perspective transform, Nice and Clean !

Output:



Separate the number tiles:

Code:

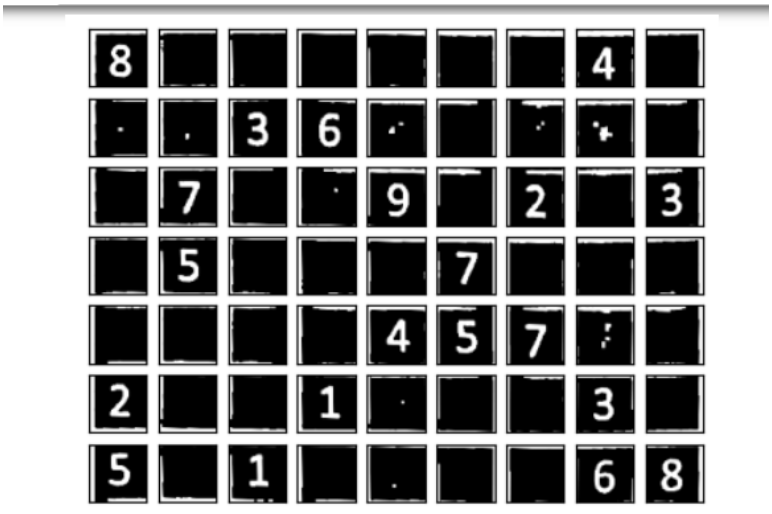
In [37]:
```
number_tiles = []
for x in range(NUM_IMAGES):
  try:
    number_tiles.append([])
    M = perspectiveImgs[x].shape[0] // 9
    N = perspectiveImgs[x].shape[1] // 9
    for i in range(9):
        number_tiles[x].append([])
        for j in range(9):
            tile = perspectiveImgs[x][i*M:(i+1)*M, j*N:(j+1)*N]
            number_tiles[x][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[x][i][j], cmap="gray");
            col.get_xaxis().set_visible(False)
            col.get_yaxis().set_visible(False)
  except:
    pass
```

Explanation:

This cell here is for separating the tiles of the sudoku grid, we calculate the dimensions M and N of each tile based on the image dimensions divided by 9. Nested loops iterate through each row and column of the Sudoku puzzle, extracting individual tiles and storing them in the number_tiles list.

Output:

Here is a sample output of one of the sudokus, as we can see each tile is separated from the other, ready for OCR!



Preprocessing of the tiles:

Code:

```
def preprocessTile(image):
  original_height, original_width = image.shape[:2]

  # Convert the image to grayscale if it is not already
  if len(image.shape) > 2 and image.shape[2] == 3:
      gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  else:
      gray = image.copy()
  # Apply threshold to get the binary image
  clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))

  # Apply CLAHE to the image
  img_clahe = clahe.apply(gray)
  darkened_image = cv2.addWeighted(img_clahe, 0.8, np.zeros(gray.shape, gray.dtype), 0, 50)

  _, thresh = cv2.threshold(darkened_image, 185, 255, cv2.THRESH_BINARY)
  return thresh
```

Explanation:

This cell here is for the preprocessing of the tiles, we make sure first that the image is grayscale if it isn't already and then we carry out CLAHE (Applies Contrast Limited Adaptive Histogram Equalization) to enhance local contrast. then we combine the result with a black image using a weighted addition, darkening the image. Then we apply thresholding afterwards for noise removal.

Code:

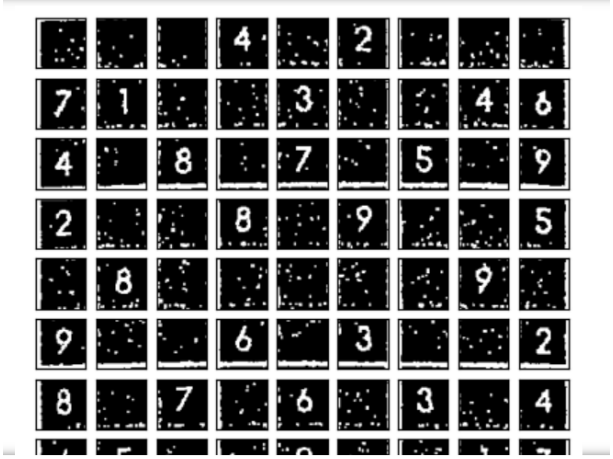
In [46]:
```
tiles = []
for x in range(NUM_IMAGES):
    tiles.append([])

    for i in range(9):
        tiles[x].append([])
        for j in range(9):
            tile = preprocessTile(number_tiles[x][i][j])
            count = np.count_nonzero(tile == 255)
            if count > 20:
                tiles[x][i].append(tile)
            else:
                img = cv2.medianBlur(number_tiles[x][i][j], 9)
                img = cv2.adaptiveThreshold(number_tiles[x][i][j],255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY,41,1)
                tiles[x][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(tiles[x][i][j], cmap="gray");
            col.get_xaxis().set_visible(False)
            col.get_yaxis().set_visible(False)
```

Explanation:

Here we loop through the tiles and apply the function preprocessTile() on each one, and after that we count the number of white pixels and then check if the number is greater than a certain value (20) then we append the tile directly, if not we apply median blur followed by adaptive thresholding to apply further preprocessing for certain images such as the images that have salt & pepper noise on them

Output:

Here we can see the output of one of the sudokus (the salt & pepper one) after further preprocessing was applied, clear tiles!



Prepare Templates:

Code:

Prepare Templates

In [47]:
```
templates = []
templatesDigits = []

for i in range(9):
    for j in range(9):
        img = cv2.imread(f"digitsTemplates/{i}{j+1}.png", cv2.IMREAD_GRAYSCALE)
        if img is not None:
            _, thresh = cv2.threshold(img, 71, 255, cv2.THRESH_BINARY)
            templates.append(thresh)
            templatesDigits.append(j+1)
        else:
            continue


for i in range(9):
    img = cv2.imread(f"noiseTemplates/{i}{0}.png", cv2.IMREAD_GRAYSCALE)
    if img is not None:
        _, thresh = cv2.threshold(img, 71, 255, cv2.THRESH_BINARY)
        templates.append(thresh)
        templatesDigits.append(0)

    else:
        continue

print(f"Number of 1's: {templatesDigits.count(1)}")
print(f"Number of 2's: {templatesDigits.count(2)}")
print(f"Number of 3's: {templatesDigits.count(3)}")
print(f"Number of 4's: {templatesDigits.count(4)}")
print(f"Number of 5's: {templatesDigits.count(5)}")
print(f"Number of 6's: {templatesDigits.count(6)}")
print(f"Number of 7's: {templatesDigits.count(7)}")
print(f"Number of 8's: {templatesDigits.count(8)}")
print(f"Number of 9's: {templatesDigits.count(9)}")
print(f"Number of noise templates: {templatesDigits.count(0)}")


fig, axes = plt.subplots(8, 6, figsize=(5, 5))
axes = axes.flatten()
for i in range(len(templates)):
    axes[i].imshow(templates[i], cmap='gray')
    axes[i].axis('off')
plt.tight_layout()
plt.show()
```

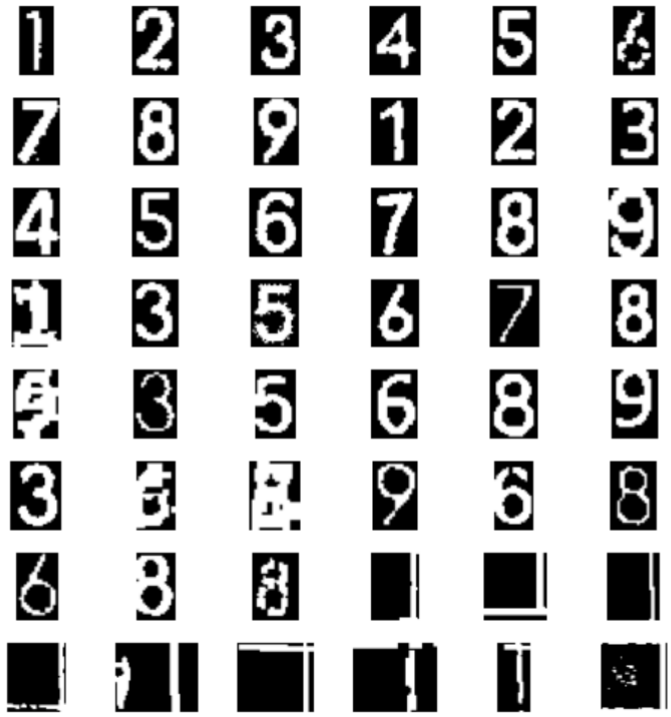
Explanation:

In this cell we prepare the templates that will be used for template matching later on, we have 2 lists: one for the noise templates that will be used to match any tiles with noise & another list for the digit templates that will match the digits in the sudoku. we drew the templates manually 😳 so there are some noise that have been produced whilst drawing so we apply thresholding again in order to remove the noise produced, however we only apply thresholding only if there is a template read hence the if condition that checks if the image is not none. We apply this logic for both templates, the noise, and the digits.

Output:

Here we can see the templates that will be used for matching templates in the following cell.

```
Number of noise templates: 9
```



Template Matching:

Code:

Template Matching

[57]:
```
newPredictedTiles = []
board = []
for x in range(16):
        suduko = []  # Initialize an empty list for each Sudoku grid
        confidenceList = []
        counter = -1
        for i in range(9):
            rows = []  # Initialize an empty list for each row in the Sudoku grid
            for j in range(9):
                count = np.count_nonzero(tiles[x][i][j] == 255)
                max_confidence = -1
                matched_digit = None
                for z, template in enumerate(templates):
                    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(11,11))

                    result = cv2.matchTemplate(tiles[x][i][j], template, cv2.TM_CCOEFF_NORMED)
                    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)

                    if max_val >= max_confidence:
                        if max_val <= 0.6:
                            matched_digit = 0
                        else:
                            max_confidence = max_val
                            matched_digit = templatesDigits[z]  # Since indices are 0-based
                rows.append(matched_digit)
                confidenceList.append(max_confidence)
                counter += 1
                newPredictedTiles.append(matched_digit)

                # Plotting the result
                plt.subplot(9, 9, i * 9 + j + 1)
                plt.imshow(tiles[x][i][j], cmap='gray')
                plt.title(matched_digit)
                plt.axis('off')
            suduko.append(rows)  # Append the completed row to the suduko grid
        board.append(suduko)  # Append the completed suduko grid to the board

        fig = plt.gcf()
        fig.set_size_inches(6, 6)
        plt.tight_layout()
        plt.show()
```

Explanation:

here in this cell we initialize 2 lists, one for the predicted digit from each tile and one to store the sudoku grids. We have 3 loops, the outer loop is for looping throughout each sudoku, in our case 15, the second loop iterates through the rows of the grid and the third loop is for the columns of the grid. For each tile we set the max_confidence to -1 and matched_digit to None and then we loop through the templates that we have and call the function. matchTemplate() that matches the tile with the template. the max_val here represents the maximum confidence returned from the template matching we carried out, if this value is greater than the max_confidence we initially set and greater than 0.6 meaning our algorithm is confident it has matched a digit correctly, we set the max_confidence variable with max_val and we set matched_digit variable to templatesDigits[z]. We then append the matched digit to our rows list. the rows list is appended after each row is finished with template matching to the sudoku list which represents the entire sudoku grid, and finally each sudoku grid is appended to the board list which represents the 16 boards we have, each one with a predicted value for all the tiles.

Output:

Here we can see a sample output, each tile has its predicted value, empty cells are predicted as 0 and the cells with numbers in them have the predicted digit above each tile.