PS #5 Report

Problem 1: Crack Detection

Given the binary image, the first step is to close the image to clean noise around the edges of the blobs. Closing is an iterative process of dilating and eroding to remove small noise and sharpen edges. Through experimentation, I found that 4 dilations and 1 erosion yielded good results for both images. Based on how this is tuned, some blobs could start to connect to each other or larger blobs could separate at their thinnest spots. I used the eye test with the original color images and binary images to check that features weren't being lost with my closing.

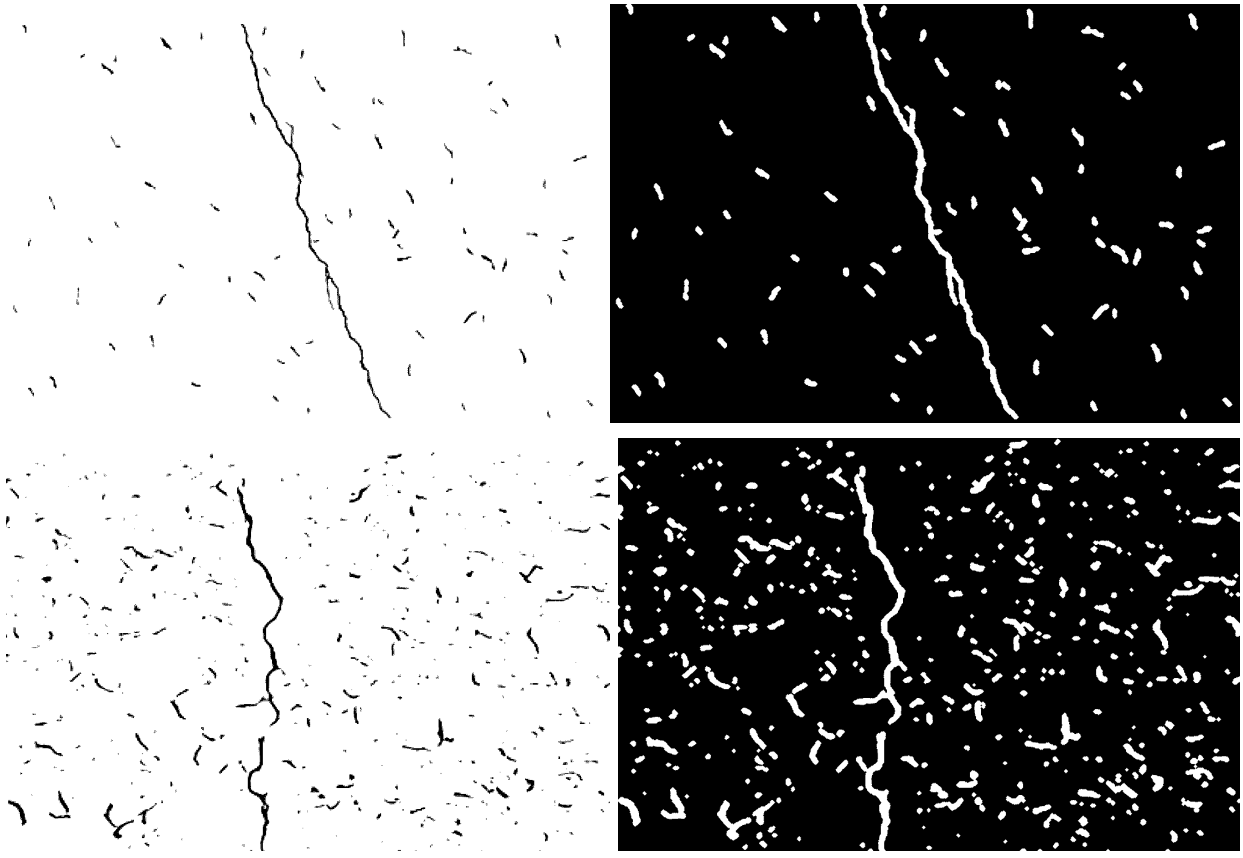

Fig. 1: Original binary images (left) and closed images (right) for wall1 (top) and wall2 (bottom).

After closing, `cv.findContours()` was used to identify all possible blobs that could be cracks. The contours were drawn onto the closed images.

Fig. 2: The closed contours on each of wall1 an wall2. Those with areas greater than 3000 px^2 were taken to be cracks.

Using the area information given by the contour finding method, contours with areas greater than 3000 px^2 were kept as cracks and all others were discarded. These blobs were then thinned using the following method:

While the current image is not all black:

1. Erode the image (becomes current image)
2. Erode the image and dilate the result (this becomes the "opening")
3. Subtract the opening from the current image

Once the current image becomes all black, take the previous current image as the thinned result.

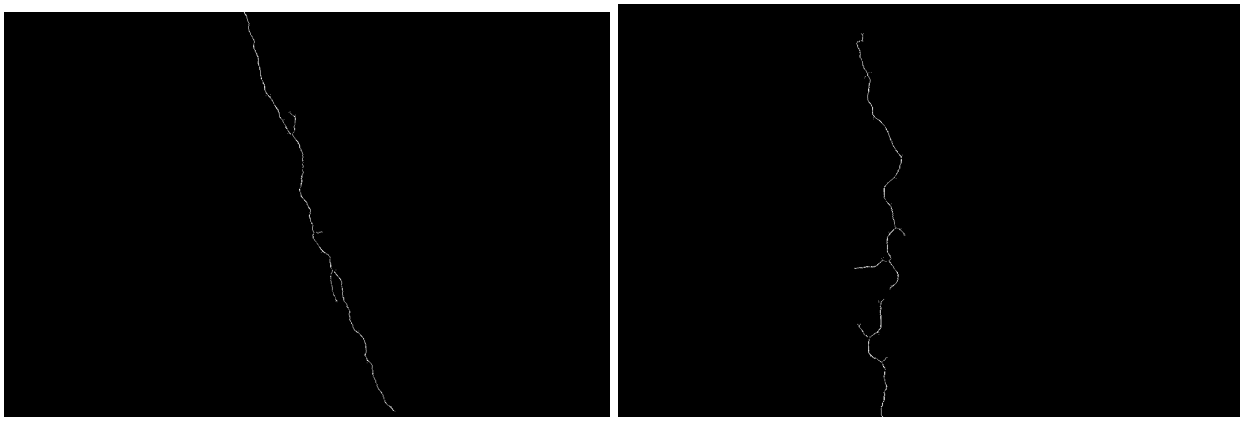

Fig. 3: Thinned results of the wall1 and wall2 cracks.

```python
# A method to thin all blobs in an image. Follows the technique outline in the ps5-tips
def thin(img):
    k_e = cv.getStructuringElement(cv.MORPH_CROSS, (3,3))
    img1 = img.copy()
    thinned = np.zeros(img1.shape)
    # While the image isn't all black
    while cv.countNonZero(img1) != 0:
        # Erode the image
        er = cv.erode(img1, k_e)
        # Erode it again, then dilate
        op = cv.morphologyEx(er, cv.MORPH_OPEN, k_e)
        # Subtract the "opening" from the eroded image
        subset = er - op
        # Recombine
        thinned = cv.bitwise_or(subset, thinned)
        img1 = er.copy()
    return thinned
```

Fig. 4: Thinning method implementation.

Problem 2: Object Detection and Manipulation

I broke the building of the catalog into 5 main steps:

1. Blob detection
2. Contour detection
3. Bounding rectangle detection
4. Object orientation
5. Object transfer into catalog image

Blob Detection

Blob detection was done using similar methods to Problem 1. The input image was grayscaled, binarized, and then closed using erosion and dilation. The method that I found worked best for these images was 3 erosions followed by 3 dilations.

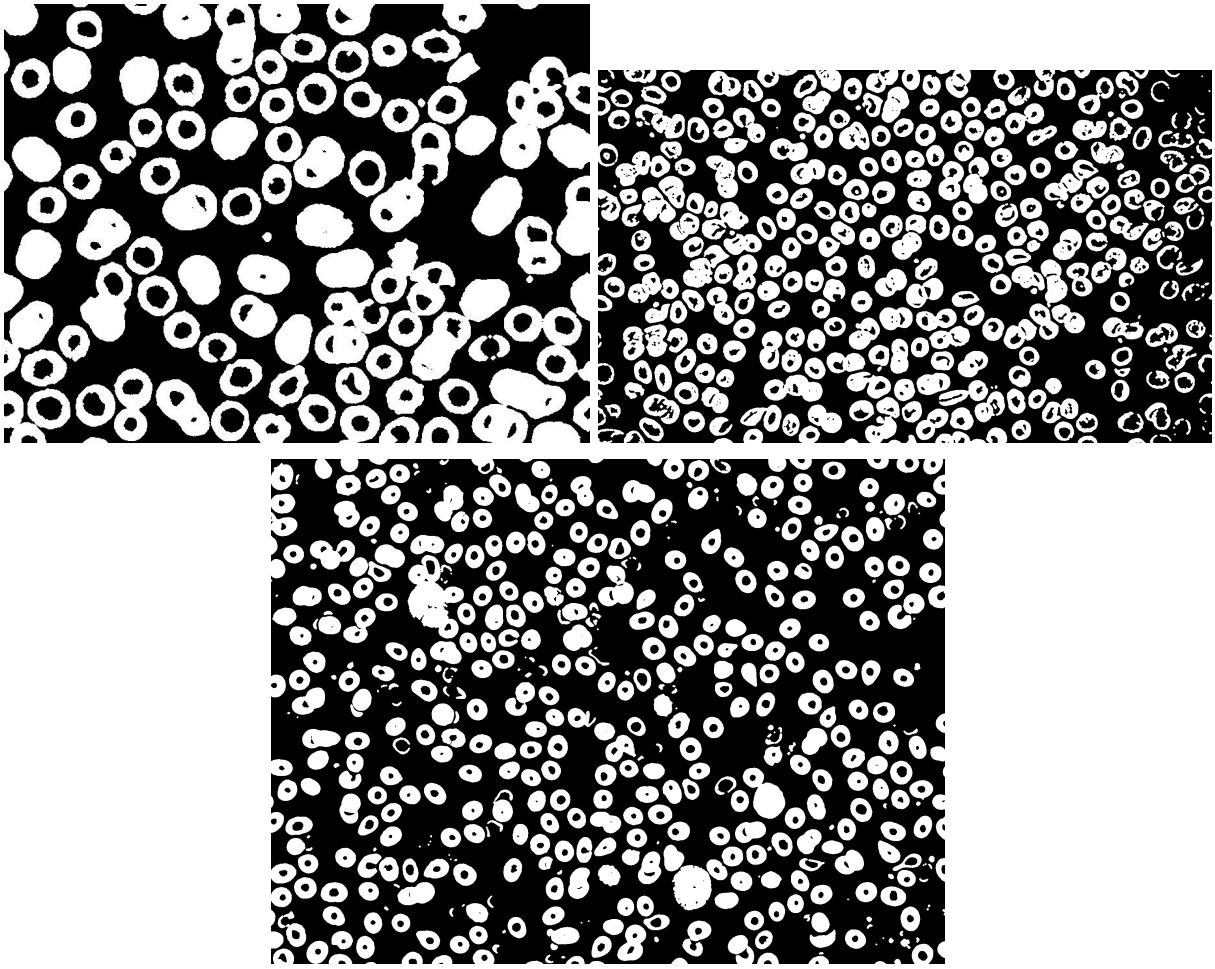

Fig. 5: Each of the blood cell images after closing.

Contour Detection, Bounding Rectangle Detection, and Object Orientation

Using the cv.findContours() method, closed contours for the cells were found. Using the contour hierarchy, cells were identified as contours that were not children (fully contained by) of

other contours. Then, using cv.minAreaRect(), each cell was given a bounding box with a corresponding center, dimensions, and angle of rotation from horizontal. Each cell was then rotated using cv.warpAffine() and the angle given by the bounding rectangle. The cell was also isolated by performing a bitwise_and operation with a black mask around everything other than the contour and its interior.

```
# Grab contours, rotate them, and insert them into the catalog image
for comp in zip(contours, hier):
    cont = comp[0]
    hier = comp[1]
    # Only grab outer contours
    if hier[3] < 0:
        cells += 1
        # Mask the original image around the contour to just have the cell
        mask = np.zeros(img.shape[:2], np.uint8)
        cv.drawContours(mask, [cont], -1, (255,255,255), -1)
        result = cv.bitwise_and(img, img, mask=mask)

        # Get min bounding rectangle and use it to rotate the cell
        rect = cv.minAreaRect(cont)
        rot_mat = cv.getRotationMatrix2D(rect[0], 90-rect[2], 1)
        rot_res = cv.warpAffine(result, rot_mat, (result.shape[1], result.shape[0]))
```

Fig. 6: Implementation of the contour detection and objection orientation.

Catalog Construction

To construct the catalog, I started with a black image whose size was determined by the maximum bounding rectangle size and the number of contours. The image was constructed as a grid of blocks size `max_h + 10 x max_w + 10.` Each cell will then be placed into one of these blocks.

```
# Catalog image is a grid based on the number of contours and the maximum contour size
max_w = math.ceil(max_w)
max_h = math.ceil(max_h)
grid_size = math.ceil(len(contours) ** 0.5)
cell_h = max_h + 10
cell_w = max_w + 10
catalog_img = np.zeros((cell_h * grid_size, cell_w * grid_size, 3))
```

Fig. 7: Construction of the grid of the catalog image.

Then, for each cell, the grid position is calculated based on what number cell it is and the dimensions of the grid. The pixels of the cell within the contour are then copied to the corresponding grid position in the catalog image.

```
# Calculate next open grid position in catalog
row = int(cells / grid_size)
col = (cells % grid_size) - 1

# Place the cell into the catalog grid
catalog_center = (int((col * cell_w + cell_w/2)), int(row * cell_h + cell_h/2))
catalog_img[catalog_center[0]-int(cell_w/2):catalog_center[0]+int(cell_w/2)][catalog_center[1]-int(cell_h/2):catalog_center[1]+int(cell_h/2)] = \
    rot_res[int(rect[0][1])-int(cell_h/2):int(rect[0][1])+int(cell_h/2)][int(rect[0][0])-int(cell_w/2):int(rect[0][0])+int(cell_w/2)]
```

Fig. 8: Transferring of the rotated cells from the original image into the catalog image.

Unfortunately, there are indexing errors that I was not able to solve by the time of this report, so the catalog images could not be generated. However, the rotating and isolating of each cell is functional.