KENNESAW STATE
UNIVERSITY

# CS 4732
# MACHINE VISION

## PROJECT 3

## MORPHOLOGICAL FILTERING

**INSTRUCTOR**

**Dr. Mahmut KARAKAYA**

**Michael Rizig**
**001008703**

# 1. ABSTRACT

In this project, we are given many different tasks to complete. To begin, we are given an image of the moon with a blur effect, and through the use of sharpening techniques such as Laplacian and Sobel filters, we are tasked with sharpening the image. The next step is to take an image of a fingerprint, generate the histogram for the image, and use the histogram to determine an ideal thresholding point, then threshold the image to convert it to binary. From there we use various techniques such as morphological dilation and erosion to remove holes from the image and improve the overall quality of the image. By utilizing various filter and structuring elements, we see different results, some better than others. Finally for our last task, we are given an image, and are asked to count the total number of cells, calculate the size of each cell, and show the boundary of the biggest cell in the image.

To view all edits, changes, and see step by step revision history, view this project on my GitHub:

https://github.com/michaelrzg/CS4732-Projects

# 2. TEST RESULTS

## 2.1 Sharpening

**(Only a few selected images are used here to highlight the effect. All output images can be found in output>log folder in the zip submission)**

**To sharpen our moon image, we utilize both the Laplacian and the Sobel filters to give the edges more contrast and make them 'sharper'. For Laplacian, we take the second derivative filter (Figure 1b) and apply it to the image as coefficients to the corresponding pixel neighborhood. This generates a Laplacian filtered image (Figure 1c) which we then subtract from our original image to get the output(Figure 1d). For Sobel, we make 2 passes, one with the horizontal Sobel filter (Figure 1e) and one with the vertical (Figure 1f). After the two passes we get a Sobel filtered image(Figure 1g) which we subtract from our original image to get our output image(Figure 1h).**

**Image 1a:** Original Image 'moon.jpg. **This is the input image.**

**Image 1b**: The Laplacian filter coefficients.

**İmage 1c:** Laplacian filtered image **(positive values only).**

**Image 1d:** Laplacian filtered image **(positive and negative values).**

**İmage 1e**: Original image minus Laplacian Filtered Image (**Output of Laplacian Filtering**) .

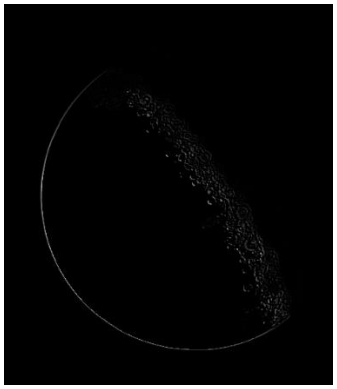
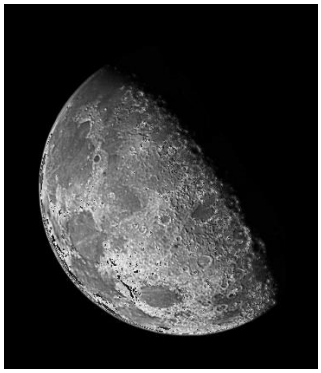
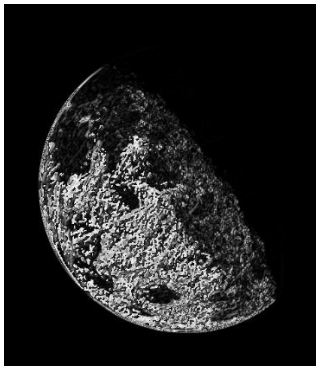
**Image 1f**: Horizontal Sobel Filter coefficients.

**Image 1g:** Vertical Sobel Filter coefficients.

**Image 1h:** Sobel Filtered Image

**Image 1i:** Original Image minus Sobel Filtered Image **(Output of Sobel Filtering)**

**Figure 1: (a)** Original Image, **(b)** Laplacian Filter Coefficients, **(c)** Laplacian filtered image (positive only), **(d)** Laplacian filtered image (positive and negative values) **(e)** Original image minus Laplacian Filtered Image (**Output of Laplacian Filtering**), **(f)** Horizontal Sobel Filter coefficients, **(g)** Vertical Sobel Filter coefficients, **(h)** Sobel Filtered Image, **(i)** Original Image minus Sobel Filtered Image **(Output of Sobel Filtering)**

<table>
<tr>
<td>  <br>(a)</td>
<td>$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$<br>(b)</td>
<td>  <br>(c)</td>
</tr>
<tr>
<td>  <br>(d)</td>
<td>  <br>(e)</td>
<td>$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$<br>(f)</td>
</tr>
<tr>
<td>$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$<br>(g)</td>
<td>  <br>(h)</td>
<td>  <br>(i)</td>
</tr>
</table>

## 2.2 Fingerprint Morphological Processing.

**(Only a few selected images are used here to highlight the effect. All output images can be found in output>log folder in the zip submission)**

**For the fingerprint image, the first step is to convert the original image *(2a)* into a binary image. To do this, we first must generate the histogram to determine a good thresholding point. Based on the images histogram distribution *(2b)* , we can see that there is a small spike around 75-80 and a large spike around 225. Based on this we can determine any value in between these two to be the threshold value, I choose 150 because it is a value far enough from both that we will not get any outlier values mixed up. Using this threshold we can generate the binary image *(2c)* of the fingerprints. The next step is to determine our size and shape of our structuring element. We begin with a 3x3 structuring element of a cross shape *(2d)*, as it seemed to work well in the slides. We start by dilating the image in order to close the holes found in the fingerprint. We do this by looking for 'hits' where at least one pixel is hit by the structuring element. If we get a *hit*, we make the target pixel 255. The result is a 'dilated' image where elements become enlarged effectively filling the holes in the image *(2e).* Next we 3x3 erode the image, which is done by looking for 'fit' or where all pixels in structuring element are hit, then we make the target 255. This sequence, called "closing operation" effectively shrinks the elements back to their original size and preserves the previous effects of dilation to the imperfections, giving us output for 3x3 filtering *(2f).* We repeat this effect with a 5x5 structuring element (2g) for 5x5 dilation (2h) and 5x5 erosion (2i) and see that the larger structuring element causes bodies to combine, making the erosion effect less effective.**

**Image 2a:** We begin with the original fingerpint.jpg. **This is the input image.**

**Image 2b:** Histogram Distribution of Original Fingerprint Image.

**İmage 2c:** Binary Image generated after thresholding at 150.

**İmage 2d:** 3x3 structuring element.

**Image 2e:** Original image after 3x3 Dilation (many holes 'closed')

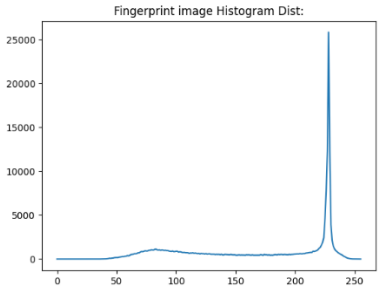
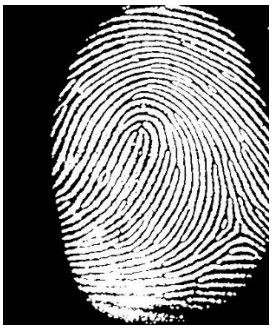
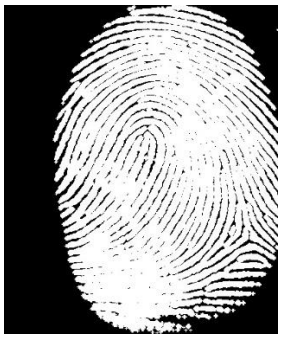
**Image 2f:** 3x3 Dilated image after 3x3 Erosion (**Output for 3x3 Morphological Processing**).

**Image 2g:** 5x5 Structuring Element

**Image 2h:** Original image after 5x5 Dilation.

**Image 2i:** 5x5 Dilated image after 5x5 Erosion.(**Output for 5x5 Morphological Processing)**

**Figure 2: (a)** Original fingerprint.jpg image, **(b)** Histogram Distribution of Original Fingerprint Image, **(c)** Binary Image generated after thresholding at 150, **(d) :** 3x3 structuring element, **(e)** Original image after 3x3 Dilation, **(f)** 3x3 Dilated image after 3x3 Erosion (**Output for 3x3 Morphological Processing), (g)** 5x5 Structuring Element **, (h)** Original image after 5x5 Dilation**. (i)** 5x5 Dilated image after 5x5 Erosion. (**Output for 5x5 Morphological Processing)**

<table>
<tr>
<td>  <br>(a)</td>
<td> 
<br>(b)</td>
<td>  <br>(c)</td>
</tr>
<tr>
<td>$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$<br>(d)</td>
<td>  <br>(e)</td>
<td>  <br>(f)</td>
</tr>
<tr>
<td>$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$<br>(g)</td>
<td>  <br>(h)</td>
<td>  <br>(i)</td>
</tr>
</table>

### 2.3 Cell Identification and Boundary Extraction

**(See 3.3 OUTPUT cellsBoundries.py in CODES section for detailed step by step documentation of each step in comments, as well as output for size, # of cells, etc)**

**For this part of the project, I start by counting how many cells are in our input image (3a). The first step of this process is to threshold the image so we can deal with binary values rather than changing rgb values. After thresholding, we erode the image to reduce the effect of any random holes or specs around the cells (3b). To count how many cells are in the image, we perform connected component analysis. We start by performing a DFS on each pixel that has a nonzero value and has not been id'd yet and give all reachable pixels the same id value. After performing this dfs on each unique hit, we then combine all touching id values to produce our final set of Id values which should each represent a unique element. The results are then color coded to for easy viewing (3c). The next step is to calculate the size of each cell. To do this we first find each unique id value, then count how many of each value appears in our image. The code for this and output values can be found in the codes section. Finally, to determine the largest value, we simply find the max value in our count and match it to the correct id(3d). To boundary the largest cell, we first erode the image with respect to the largest cell's id. This is done by checking the sum of the neighborhood operation for erosion and if it matches largest id * 5, then we know all 5 pixels hit a largest cell pixel(3e). Finally, we take the eroded image and subtract it from our largest cell to produce the boundary for our largest cell. (3f)**

**Image 3a:** We begin with the original cells.jpg. **This is the input image.**

**Image 3b:** Input image after threshold and erosion.

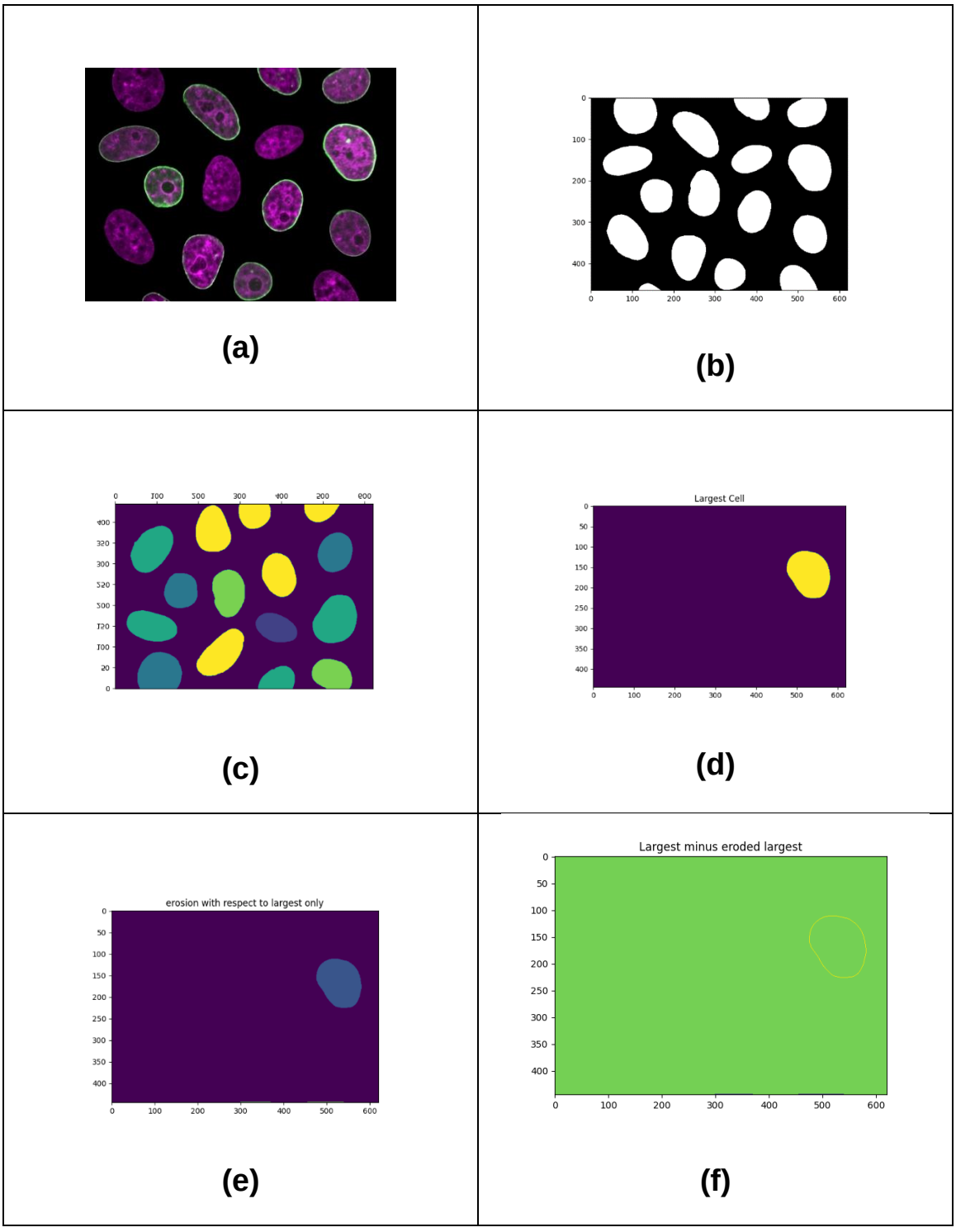
**Image 3c:** Image after connected component analysis.

**Image 3d:** Largest Cell Extraction

**Image 3e:** Eroded Largest Cell

**Image 3f:** Largest Cell minus the eroded largest cell (to show boundary of largest cell)

**Figure 3: (a)** Input image, **(b)** Input image after threshold and erosion**, (c)** Image after connected component analysis, **(d)** Largest Cell Extraction, **(e)** Eroded Largest Cell**, (f)** Largest Cell minus the eroded largest cell (to show boundary of largest cell)



### 2.4 Discussion

In this project, we performed images sharpening, morphological processing, and connected component analysis to produce interesting results. In part 1, we utilize Laplace and Sobel 2nd derivative and first derivative filters to sharpen the moon image which has some blur. In my opinion the Laplace filter did a better job, as it did not over sharpen, The Sobel vertical and horizontal filters over sharpened the image in my opinion, highlighting smaller elements that didn't need to be highlighted. In part 2, we found that using the larger structuring element proved to over-dilate the image which resulted in the image blending, with discrete elements blending into fewer bigger elements despite the 5x5 erosion. We saw that the 3x3 erosion did the job almost perfectly, with only some very minor imperfections remaining. For part three, we used a combination of morphological and connected component analysis to count how many elements we have. This part was particularly challenging as any small particles would be counted as a cell. To counter this, I utilized thresholding and erosion to remove any outstanding pixels and focus on larger bodies. For calculating size, we simply used loops and IDs to count each cell's size. Finally, by eroding the largest cell and subtracting the erosion from the original, we can get the boundary of the largest cell. SEE 3.3 OUTPUT in codes section for counts and sizes. Given more time, it would be interesting to expand on the cell detection and see its possible use cases for training AI models on medical research.

## 3. CODES

### 3.1 Code for Sharpening.py

```
# Michael Rizig
# Project 3: Morphological Filers
# 001008703
# File 1: Sharpening
# 7/5/2024

#Import necessary tools
import matplotlib.pyplot as plottool
import cv2
from skimage import io
import numpy as np
# below function defines the which is the sharpening filter used

def laplacian(i,j):
    #instead of using loops, im utilizing linear alg to turn the filter into a
linar combinaiton of the two maticies. (c1v1 + c2v2 .. cmvm)
    s= padded[i-1, j-1]*lapFilter[0][0]+padded[i-1, j]*lapFilter[0][1]+padded[i-
1, j + 1]*lapFilter[0][2]+padded[i, j-1]*lapFilter[1][0]+
padded[i][j]*lapFilter[1][1]+padded[i, j + 1]*lapFilter[1][2]+padded[i + 1, j-
1]*lapFilter[2][0]+padded[i + 1, j]*lapFilter[2][1]+padded[i + 1, j +
1]*lapFilter[2][2]
    #correcting for negative values by taking absolute value
    if(s[1]<0):
        return 256- s
    return s
```

```
def sobel(i,j):
    s=0
    #take the defined sobel mask/filter and apply it to each pixel in the range
x-1,y-1 to x+1,y+1
    #as before, we are utilizing linar combinations.
    #one pass for vertical
    s= padded[i-1, j-1]*sobelV[0][0]+padded[i-1, j]*sobelV[0][1]+padded[i-1, j +
1]*sobelV[0][2]+padded[i, j-1]*sobelV[1][0]+ padded[i][j]*sobelV[1][1]+padded[i,
j + 1]*sobelV[1][2]+padded[i + 1, j-1]*sobelV[2][0]+padded[i + 1,
j]*sobelV[2][1]+padded[i + 1, j + 1]*sobelV[2][2]
    #one pass for horizontal
    s+= padded[i-1, j-1]*sobelH[0][0]+padded[i-1, j]*sobelH[0][1]+padded[i-1, j +
1]*sobelH[0][2]+padded[i, j-1]*sobelH[1][0]+ padded[i][j]*sobelH[1][1]+padded[i,
j + 1]*sobelH[1][2]+padded[i + 1, j-1]*sobelH[2][0]+padded[i + 1,
j]*sobelH[2][1]+padded[i + 1, j + 1]*sobelH[2][2]

    #correcting for negative values by taking absolute value
    if(s[1]<0):
        return 256- s
    return s

#define lapical filter we are using (from slides):
lapFilter = [[1,1,1],[1,-8,1],[1,1,1]]

#sobel filters vertical and horizontal (from slides):
sobelV = [[-1,-2,-1],[0,0,0],[1,2,1]]
sobelH = [[-1,0,1],[-2,0,2],[-1,0,1]]

#define path
path = 'input/moon.jpg'

#read image
moonimage = io.imread(path)
#cvt color
moonimage = cv2.cvtColor(moonimage,cv2.COLOR_BGR2RGB)
#created padded version (for edges of image)
padded = cv2.copyMakeBorder(moonimage,1,1,1,1,cv2.BORDER_CONSTANT,value=[0,0,0])

#define an output image to be same dimentions etc as input image
output = moonimage.copy()
#create a blank image for lapician filter results
lapic = np.zeros([moonimage.shape[0], moonimage.shape[1], 3], dtype=np.uint8)

#create a blank image for sobel filter results
sob = np.zeros([moonimage.shape[0], moonimage.shape[1], 3], dtype=np.uint8)
```

```
#display initial image
plottool.imshow(padded)
plottool.show()

#apply lapician image enhancement
for i in range(moonimage.shape[0]):
   for j in range(moonimage.shape[1]):
        lapic[i][j] = laplacian(i,j)
        output[i][j] = output[i][j]+lapic[i][j]

#display the lapician filter generated
plottool.imshow(lapic)
plottool.show()

#save filter
#cv2.imwrite('output/sharpen/laplacian/laplacianFilter1.jpg',lapic)

#display output image (output - lapican filter generated)
plottool.imshow(output)
plottool.show()

#save output
#cv2.imwrite('output/sharpen/laplacian/outLaplace.jpg',output)

#reset output for sobel
output = moonimage.copy()

#apply sobel
for i in range(moonimage.shape[0]):
   for j in range(moonimage.shape[1]):
        sob[i][j] = sobel(i,j)

#combine sobel
output=output + sob

#display the sobel filter generated
plottool.imshow(sob)
plottool.show()
#save filter
cv2.imwrite('output/sharpen/sobel/sobelFilter.jpg',sob)

#display output image (output - sobel filter generated)
plottool.imshow(output)
plottool.show()
```

```
#save output
cv2.imwrite('output/sharpen/sobel/outSobel.jpg',output)
```

---

### 3.2 Code for MorphologicalProcessing.py

```
# Michael Rizig
# Project 3: Morphological Filers
# 001008703
# File 2: Morph1
# 7/5/2024

#Import nessessary tools
import matplotlib.pyplot as plottool
import cv2
from skimage import io
import numpy as np

def tresh(x):
    if x>150:
        return [0,0,0]
    return [255,255,255]

def dialation(i,j):
    s=0
    s= padded[i-1, j-1]*struct[0][0]+padded[i-1, j]*struct[0][1]+padded[i-1, j +
1]*struct[0][2]+padded[i, j-1]*struct[1][0]+ padded[i][j]*struct[1][1]+padded[i,
j + 1]*struct[1][2]+padded[i + 1, j-1]*struct[2][0]+padded[i + 1,
j]*struct[2][1]+padded[i + 1, j + 1]*struct[2][2]
    if s[0]>0:
        return [255,255,255]
    return [0,0,0]

def erosion(i,j):
    s=0
    s= padded[i-1][j-1][0]*struct[0][0]+padded[i-1][j][0]*struct[0][1]+padded[i-
1][j + 1][0]*struct[0][2]+padded[i][j-1][0]*struct[1][0]+
padded[i][j][0]*struct[1][1]+padded[i][j + 1][0]*struct[1][2]+padded[i + 1][j-
1][0]*struct[2][0]+padded[i + 1][j][0]*struct[2][1]+padded[i + 1][j +
1][0]*struct[2][2]
    if s>1024: # value chosed bc 256 * 4 = 1024, if the value is greater than
1024, than 5 pixels must have been hit
        return [255,255,255]
    return [0,0,0]
```

```
def dialation5(i,j):
    s=0
    s= padded[i-2][j-2]*struct1[0][0] + padded[i-2][j-1]*struct1[0][1] +
padded[i-2][j]*struct1[0][2] + padded[i-2][j+1]*struct1[0][3] + padded[i-
2][j+2]*struct1[0][4] + padded[i-1][j-2]*struct1[1][0] + padded[i-1][j-
1]*struct1[1][1] + padded[i-1][j]*struct1[1][2] + padded[i-1][j+1]*struct1[1][3]
+ padded[i-1][j+2]*struct1[1][4] +  padded[i][j-2]*struct1[2][0] + padded[i][j-
1]*struct1[2][1] + padded[i][j]*struct1[2][2] + padded[i][j+1]*struct1[2][3] +
padded[i][j+2]*struct1[2][4] +  padded[i+1][j-2]*struct1[3][0] + padded[i+1][j-
1]*struct1[3][1] + padded[i+1][j]*struct1[3][2] + padded[i+1][j+1]*struct1[3][3]
+ padded[i+1][j+2]*struct1[3][4] + padded[i+2][j-2]*struct1[4][0] +
padded[i+2][j-1]*struct1[4][1] + padded[i+2][j]*struct1[4][2] +
padded[i+2][j+1]*struct1[4][3] + padded[i+2][j+2]*struct1[4][4]
    if s[0]>0:
# value chosed bc if the value is greater than 0, than at least 1 must have been
hit
        return [255,255,255]
    return [0,0,0]

def erosion5(i,j):
    s=0
    s= padded[i-2][j-2][0]*struct1[0][0] + padded[i-2][j-1][0]*struct1[0][1] +
padded[i-2][j][0]*struct1[0][2] + padded[i-2][j+1][0]*struct1[0][3] + padded[i-
2][j+2][0]*struct1[0][4] + padded[i-1][j-2][0]*struct1[1][0] + padded[i-1][j-
1][0]*struct1[1][1] + padded[i-1][j][0]*struct1[1][2] + padded[i-
1][j+1][0]*struct1[1][3] + padded[i-1][j+2][0]*struct1[1][4] +  padded[i][j-
2][0]*struct1[2][0] + padded[i][j-1][0]*struct1[2][1] +
padded[i][j][0]*struct1[2][2] + padded[i][j+1][0]*struct1[2][3] +
padded[i][j+2][0]*struct1[2][4] +  padded[i+1][j-2][0]*struct1[3][0] +
padded[i+1][j-1][0]*struct1[3][1] + padded[i+1][j][0]*struct1[3][2] +
padded[i+1][j+1][0]*struct1[3][3] + padded[i+1][j+2][0]*struct1[3][4] +
padded[i+2][j-2][0]*struct1[4][0] + padded[i+2][j-1][0]*struct1[4][1] +
padded[i+2][j][0]*struct1[4][2] + padded[i+2][j+1][0]*struct1[4][3] +
padded[i+2][j+2][0]*struct1[4][4]
    if s>2048: # value chosed bc 256 * 8 = 2048, if the value is greater than
2048, than 9 pixels must have been hit
        return [255,255,255]
    return [0,0,0]

#image path
impath = 'input/fingerprint.jpg'

#import image
fingerprint = io.imread(impath)
```

```
#fix colors
fingerprint = cv2.cvtColor(fingerprint,cv2.COLOR_BGR2RGB)

#create ouput image:
output = fingerprint.copy()

#display image
plottool.imshow(fingerprint)
plottool.show()

#calculate histogram for image using cv2
hist = cv2.calcHist([fingerprint], [0], None, [256], [0,256])

#display original image's histogram distribution
plottool.plot(hist)
plottool.title("Fingerprint image Histogram Dist:")
plottool.savefig('output/morph/finger/hist.png')
plottool.show()

#define output for thresholding
thresholdImage = fingerprint.copy()

#apply threshold function
for i in range(fingerprint.shape[0]):
    for j in range(fingerprint.shape[1]):
        thresholdImage[i][j] = tresh(fingerprint[i][j][0])

#display results of threshold
plottool.imshow(thresholdImage)
plottool.title("Thresholded Image:")
plottool.show()

#created padded verion for filtering:
padded =
cv2.copyMakeBorder(thresholdImage,1,1,1,1,cv2.BORDER_CONSTANT,value=[0,0,0])

#save threshold image
cv2.imwrite('output/morph/finger/thresh.jpg',thresholdImage)

#define structuring element:
struct = [[0,1,0],[1,1,1],[0,1,0]]
struct1=[[0,0,1,0,0],[0,0,1,0,0],[1,1,1,1,1],[0,0,1,0,0],[0,0,1,0,0]]

#3x3 dialate image:
```

```
for i in range(fingerprint.shape[0]):
    for j in range(fingerprint.shape[1]):
        output[i][j] = dialation(i,j)

#display results of dialation
plottool.imshow(output)
plottool.title("Dialated 3x3:")
plottool.show()

#save results for dialation
cv2.imwrite('output/morph/finger/dialated.jpg',output)

#updated padded
padded =  cv2.copyMakeBorder(output,1,1,1,1,cv2.BORDER_CONSTANT,value=[0,0,0])

#3x3 erode image:
for i in range(fingerprint.shape[0]):
    for j in range(fingerprint.shape[1]):
        output[i][j] = erosion(i,j)

#display results of 3x3 erosion
plottool.imshow(output)
plottool.title("Eroded 3x3:")
plottool.show()

#save results for erosion
cv2.imwrite('output/morph/finger/eroded.jpg',output)

#reset padded image for 5x5 filter
padded
=  cv2.copyMakeBorder(thresholdImage,2,2,2,2,cv2.BORDER_CONSTANT,value=[0,0,0])

#5x5 dialaion
for i in range(fingerprint.shape[0]):
    for j in range(fingerprint.shape[1]):
        output[i][j] = dialation5(i,j)


#display results of 5x5 dialation
plottool.imshow(output)
plottool.title("Dialated 5x5:")
plottool.show()

#save results for dialation
cv2.imwrite('output/morph/finger/dialated5.jpg',output)
```

```
#updated padded
padded =  cv2.copyMakeBorder(output,2,2,2,2,cv2.BORDER_CONSTANT,value=[0,0,0])

#5x5 erode :
for i in range(fingerprint.shape[0]):
    for j in range(fingerprint.shape[1]):
        output[i][j] = erosion5(i,j)

#display results of 5x5 erosion
plottool.imshow(output)
plottool.title("Eroded 5x5:")
plottool.show()

#save results for 5x5 morphological erosion
cv2.imwrite('output/morph/finger/eroded5.jpg',output)
```

---

### 3.3 Code for cellBoundries.py

```
# Michael Rizig
# Project 3: Morphological Filers
# 001008703
# File 3: Cell Boundaries
# 7/6/2024

#Import nessessary tools
import matplotlib.pyplot as plottool
import cv2
from skimage import io
import numpy as np
import random

#same erosion function found in previous file (morphologicalprocessing.py)
def erosion(i,j):
    s=0
    s= padded[i-1][j-1][0]*struct[0][0]+padded[i-1][j][0]*struct[0][1]+padded[i-
1][j + 1][0]*struct[0][2]+padded[i][j-1][0]*struct[1][0]+
padded[i][j][0]*struct[1][1]+padded[i][j + 1][0]*struct[1][2]+padded[i + 1][j-
1][0]*struct[2][0]+padded[i + 1][j][0]*struct[2][1]+padded[i + 1][j +
1][0]*struct[2][2]
    if s>1024: # value chosed bc 256 * 4 = 1024, if the value is greater than
1024, than 5 pixels must have been hit
        return [255,255,255]
    return [0,0,0]
```

```
#this function is to determine which components are conneceted
def connectedComponentAnalysis():
    #create static identifier for unique values
    id = 100

    #step 1: first pass to give temperary id to elements
    #loop through image
    for i in range(1,cellsImage.shape[0]):
        for j in range(1,cellsImage.shape[1]):
            #if a value is not 0 and has not been id'd yet
            if padded[i][j][0] >0 and componentsArray[i][j]==0:
                # call depth first seatch on this pixel
                dfs(padded,i,j,id)
                #update id
                id+=50
    # correct for previous padding
    ccomponentsArray = componentsArray[0:componentsArray.shape[0]-
20,0:componentsArray.shape[1]]

    #step 2: combine connected id's to find final number
    # this nested loop below compares each pixel to its 8-nieghbors and choses
the smallest value of the bunch
    for i in range(1,ccomponentsArray.shape[0]-1):
        for j in range(1,ccomponentsArray.shape[1]-1):
            if ccomponentsArray[i+1][j] < ccomponentsArray[i][j] and
ccomponentsArray[i+1][j]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i+1][j]
            if ccomponentsArray[i+1][j+1] < ccomponentsArray[i][j] and
ccomponentsArray[i+1][j+1]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i+1][j+1]
            if ccomponentsArray[i][j+1] < ccomponentsArray[i][j] and
ccomponentsArray[i][j+1]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i][j+1]
            if ccomponentsArray[i-1][j+1] < ccomponentsArray[i][j] and
ccomponentsArray[i-1][j+1]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i-1][j+1]
            if ccomponentsArray[i-1][j] < ccomponentsArray[i][j] and
ccomponentsArray[i-1][j]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i-1][j]
            if ccomponentsArray[i-1][j-1] < ccomponentsArray[i][j] and
ccomponentsArray[i-1][j-1]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i-1][j-1]
            if ccomponentsArray[i][j-1] < ccomponentsArray[i][j] and
ccomponentsArray[i][j-1]!=0:
```

```
                ccomponentsArray[i][j] = ccomponentsArray[i][j-1]
            if ccomponentsArray[i+1][j-1] < ccomponentsArray[i][j] and
ccomponentsArray[i+1][j-1]!=0:
                ccomponentsArray[i][j] = ccomponentsArray[i+1][j-1]

    #finally we count how many values are total after combining
    count=0
    last=[]
    for i in range(ccomponentsArray.shape[0]):
        for j in range(ccomponentsArray.shape[1]):
            if ccomponentsArray[i][j]!=0 and
last.count(ccomponentsArray[i][j])==0 :
                count+=1
                last.append(ccomponentsArray[i][j])

    #how many unique numbers = unique cells
    print("Number of unique cells: ", count)

    #change label values for colors to look more unique in image
    for i in range(ccomponentsArray.shape[0]):
        for j in range(ccomponentsArray.shape[1]):
            ccomponentsArray[i][j] = ccomponentsArray[i][j]

    #return our components array
    return ccomponentsArray, count

#simple depth first search to find each connected component
def dfs(image, i, j , id):

    #create stack for depth first search
    stack = []

    #id current pixel
    componentsArray[i][j] = id

    #dfs, adding each un-id'd to the stack an id'ing it, then popping next
element off stack
    while True:
        for p in range(-1,2):
            for q in range(-1,2):
                if image[(i+p)%image.shape[0]-1][j+q][0]!=0 and
componentsArray[(i+p)%image.shape[0]-2][j+q]==0:
                    stack.append((i+p,j+q))
                    componentsArray[(i+p)%image.shape[0]-2][j+q]=id
```

```
        #base case (to break out of loop eventually)
        if len(stack)==0:
            break

        #move on to next stack value
        val = stack.pop()
        #id next stack value
        componentsArray[val[0]%image.shape[0]-2,val[1]] = id
        #update i,j so we continue to move around image
        i,j = val


#define image path for cell image
cellsPath = 'input/cell.jpg'

#load image
cellsImage = io.imread(cellsPath)

#color correction
cellsImage = cv2.cvtColor(cellsImage, cv2.COLOR_BGR2RGB)

#display original imge
plottool.imshow(cellsImage)
plottool.show()

#threshold the image
for i in range (cellsImage.shape[0]):
    for j in range(cellsImage.shape[1]):
        if cellsImage[i][j][0] > 10:
            cellsImage[i][j]=[255,255,255]
        else:
            cellsImage[i][j] = [0,0,0]

#pad image
padded = cv2.copyMakeBorder(cellsImage,1,1,1,1,cv2.BORDER_CONSTANT,value=[0,0,0])

#create an array of integers to store where each connected component is
componentsArray =
np.full((cellsImage.shape[0],cellsImage.shape[1],1),0,dtype=int)

#define structure for erosion
struct = [[0,1,0],[1,1,1],[0,1,0]]

#define output for erosion
output = cellsImage.copy()
```

```
#erode image to remove any extruding inperfections that may interfere with our
algoritm
for i in range (cellsImage.shape[0]):
    for j in range(cellsImage.shape[1]):
        output[i][j] = erosion(i,j)

#update image to our eroded image
cellsImage = output

#display image after threshold and erosion
plottool.imshow(cellsImage)
plottool.savefig('output/morph/cells/thresholded&Eroded.png')
plottool.show()

#apply connected component analysis to determine how many cells are in the image
out, count = connectedComponentAnalysis()

#display output of connected component function
plottool.imshow(out)
#plottool.savefig('output/morph/cells/CCA.png')
plottool.show()

#create array to hold count values
cellSizes = [0 for i in range(count)]
values = []

# find all unique values from image
for i in range (out.shape[0]):
    for j in range(out.shape[1]):
        if out[i][j]!=0 and values.count(out[i][j])==0:
            values.append(out[i][j])

#count each unique value and store it in cellSizes array
for i in range (out.shape[0]):
    for j in range(out.shape[1]):
        if out[i][j]!=0:
            cellSizes[values.index(out[i][j])]+=1
#display sizes of cells
print(values)
print(cellSizes)

#find label for largest cell
largestCell = values[cellSizes.index(max(cellSizes))]
```

```
#create an image to store original bounds of largest cell
largestOriginalBound = out.copy()

#find largest bounds
for i in range(largestOriginalBound.shape[0]):
    for j in range(largestOriginalBound.shape[1]):
        largestOriginalBound[i][j]=0
        if out[i][j] == largestCell:
            largestOriginalBound[i][j] = largestCell

#show largst cell
plottool.imshow(largestOriginalBound)
plottool.title("Largest Cell")
plottool.savefig('output/morph/cells/largestCell.png')
plottool.show()

#now we erode the image using the same struct as before but now with respect to
only the largst cell
#output for erosion
boundryOut = out.copy()

#erosion process
for i in range (out.shape[0]-1):
    for j in range(out.shape[1]-1):
        #same as erosion function from previous file(morphologicalProcessing.py)
        s= out[i-1][j-1][0]*struct[0][0]+out[i-1][j][0]*struct[0][1]+out[i-1][j +
1][0]*struct[0][2]+out[i][j-1][0]*struct[1][0]+
out[i][j][0]*struct[1][1]+out[i][j + 1][0]*struct[1][2]+out[i + 1][j-
1][0]*struct[2][0]+out[i + 1][j][0]*struct[2][1]+out[i + 1][j +
1][0]*struct[2][2]
        # if s = largestcell*5, then all 5 pixels in filter hit a largest cell
value, so make target pixel largest cell value
        if s==largestCell*5:
            boundryOut[i][j]=largestCell
        else:
            #else set to 0
            boundryOut[i][j]=0

#display eroded largest cell
plottool.imshow(boundryOut)
plottool.title("erosion with respect to largest only")
plottool.savefig('output/morph/cells/erodedlargest.png')
plottool.show()

#subtract eroded cell from original to get boundry
```

```
largestOriginalBound-=boundryOut

#finally, display boundry.
plottool.imshow(largestOriginalBound)
plottool.title("Largest minus eroded largest")
plottool.savefig('output/morph/cells/LargestMinusErodedLargest.png')
plottool.show()
```

---

**OUTPUT FOR 3.3:**

**Number of unique cells: 16**

**#cell ID's assigned:**

**[array([100]), array([150]), array([200]), array([1000]), array([16900]), array([17050]), array([17150]), array([24700]), array([33100]), array([35250]), array([43400]), array([43850]), array([49750]), array([57500]), array([59750]), array([63650])]**

**#cell sizes:**

**[8062, 3897, 5733, 8159, 9472, 5324, 6863, 7091, 5510, 6703, 8153, 6256, 7356, 3863, 2736, 1832]**