In [1]:

```
'''
FloodFill modified to traverse the entire image and output different color.
TODO: Sequential Labeling: https://www.youtube.com/watch?v=ticZclUYy88
TODO: do a pre-process-3 for tumor cells
TODO: filter out components based on area to entire image area ratio.
'''
```

Out[1]:

```
'\nFloodFill modified to traverse the entire image and output different color.\n'
```

In [2]:

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

In [3]:

```
options = {
    'img_dir': './img'
}
```

In [61]:

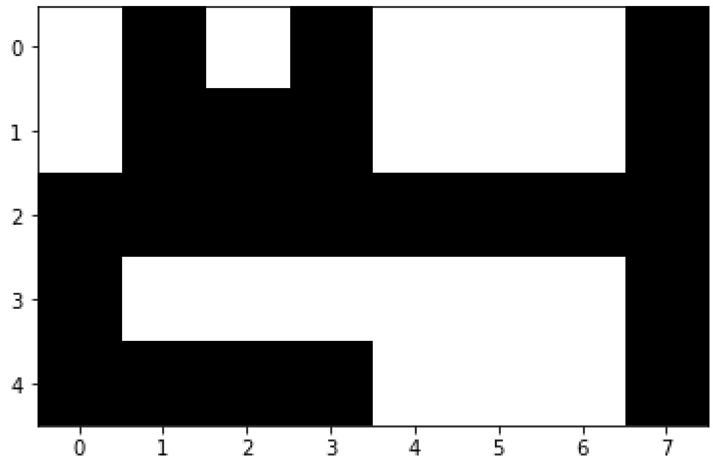
```
def get_unique_colors(img):
    return (np.unique(img.reshape(-1, img.shape[2]), axis=0))

def getNextNewColor(usedColors):
    newColor = (np.random.choice(range(256), size=3))
    while np.any([np.all(uc == newColor) for uc in usedColors]): # if ne
wColor matches any of the oldColors
        newColor = (np.random.choice(range(256), size=3))
    return newColor

def floodfill(surface, x, y, oldColors, usedColors):
    if surface[x][y] not in oldColors: # Has new color already. No need
 to look.
        return surface, usedColors

    colorOfFocus = surface[x][y].copy()
    newColor = getNextNewColor(usedColors)
    usedColors = np.vstack([usedColors, newColor])

    # Add first coord into stack
    theStack = [(x, y)]

    while len(theStack) > 0:
        x, y = theStack.pop()

        if x < 0 or x > surface.shape[0]-1 or y < 0 or y > surface.shape
[1]-1: # Out of Bounds
            continue

        if np.all(surface[x][y] == colorOfFocus):
            surface[x][y] = newColor
            theStack.append((x+1, y))  # right
            theStack.append((x-1, y))  # left
            theStack.append((x, y+1))  # down
            theStack.append((x, y-1))  # up

    return surface, usedColors

def flood_fill_multi(img, debug=False):
    oldColors = get_unique_colors(img)
    usedColors = get_unique_colors(img)

    if debug:
        print("Used Colors")
        plt.imshow(usedColors)
        plt.show()

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            img, usedColors = floodfill(img, i, j, oldColors, usedColors
)

    return img, usedColors

def gen_color_key(color):
    return "_".join(str(channel) for channel in color)
```

```
def get_largest_components(img, usedColors, n=2):
    h = {}
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            color = img[i][j]re
            color_key = gen_color_key(color)
            if color_key in h.keys():
                h[color_key] += 1
            else:
                h[color_key] = 1

    h_desc = [item[0] for item in sorted(h.items(), key = lambda kv:(kv[
1], kv[0]))]
    h_desc_rev_filt = list(reversed(h_desc))[:n]
    top_n_components = [[int(ck) for ck in colorkey.split('_')] for colo
rkey in h_desc_rev_filt]
    return top_n_components

def filter_out_colors(img, colors, bgColor):
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            curr_color = img[i][j]
            if not np.any([np.all(c == curr_color) for c in colors]):
                img[i][j] = bgColor
    return img
```

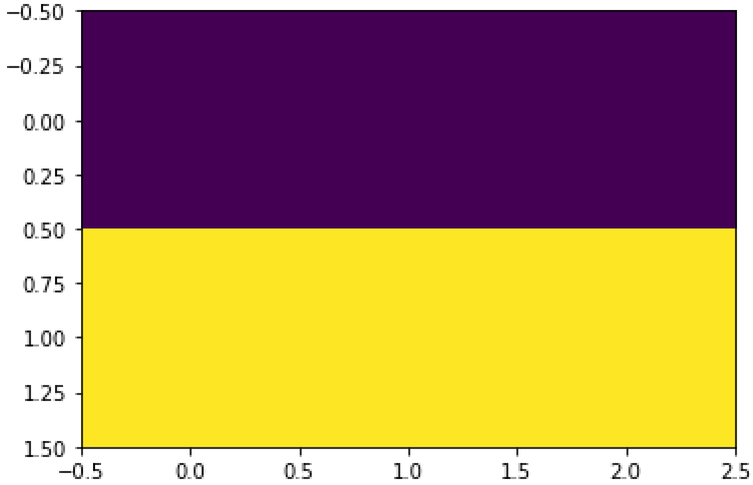
In [173]:

```
def test_flood_fill_multi():
    a = np.array([[(255,255,255), (0,0,0), (255,255,255), (0,0,0), (255,255,255), (255,255,255), (255,255,255), (0,0,0)],
                  [(255,255,255), (0,0,0), (0,0,0), (0,0,0), (255,255,255), (255,255,255), (255,255,255), (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), (255,255,255), (255,255,255), (255,255,255), (255,255,255), (255,255,255), (255,255,255), (0,0,0)],
                  [(0,0,0), (0,0,0), (0,0,0), (0,0,0), (255,255,255), (255,255,255), (255,255,255), (0,0,0)]])

    print("Orig img")
    plt.imshow(a)
    plt.show()

    a, usedColors = flood_fill_multi(a, debug=True)

    print(usedColors)
    print("FloodFill | after")
    plt.imshow(a)
    plt.show()

    largest_colors = get_largest_components(a, usedColors, n=3)
    print("largest_colors", largest_colors)
    plt.imshow(np.array([largest_colors]))
    plt.show()

    img = filter_out_colors(a, largest_colors, [255, 255, 255])
    plt.imshow(img)
    plt.show()

test_flood_fill_multi()
```

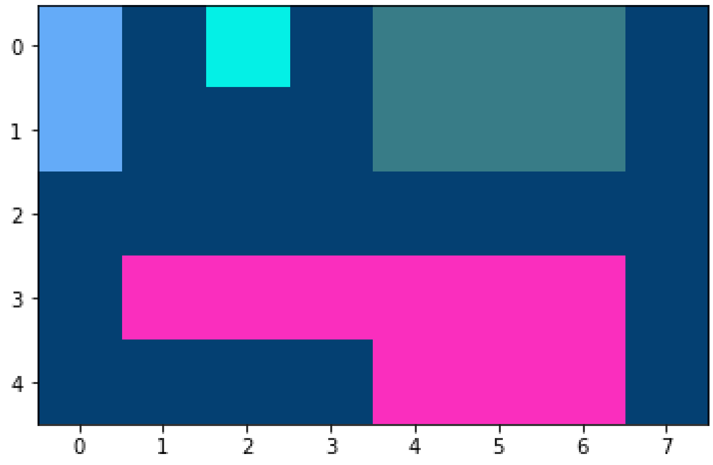
```
Orig img
```



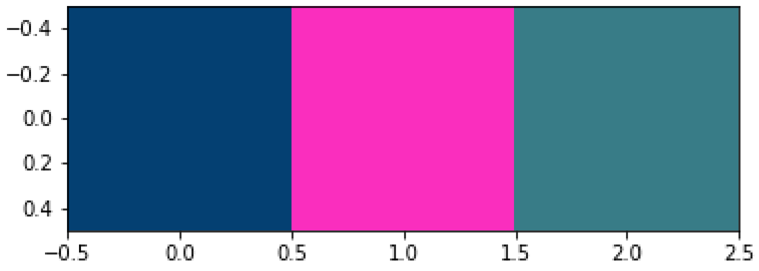
```
Used Colors
```

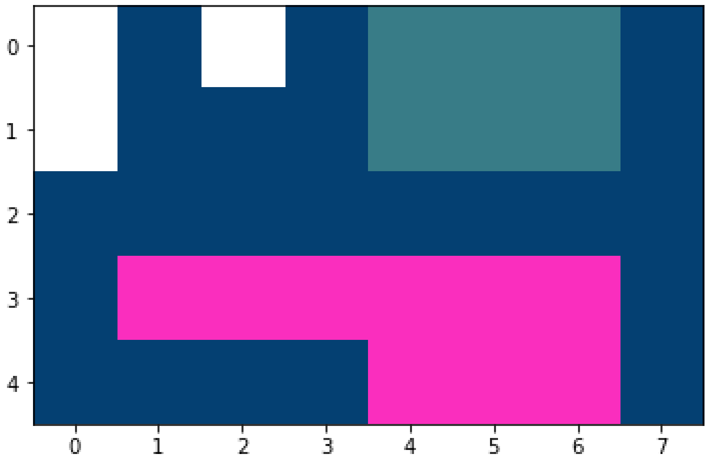


```
[[  0   0   0]
 [255 255 255]
 [100 171 248]
 [  4  64 114]
 [  4 240 230]
 [ 56 124 135]
 [250  46 190]]
FloodFill | after
```



```
largest_colors [[4, 64, 114], [250, 46, 190], [56, 124, 135]]
```



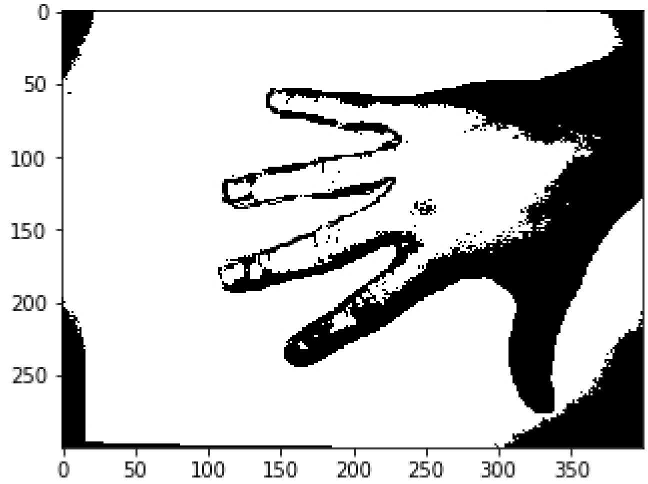


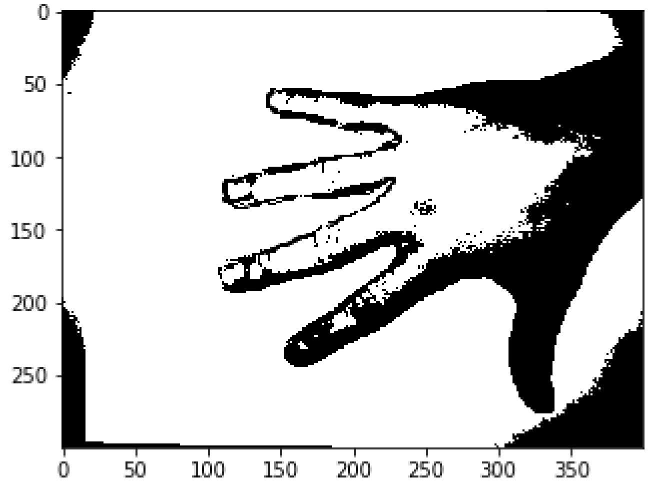
In [343]:

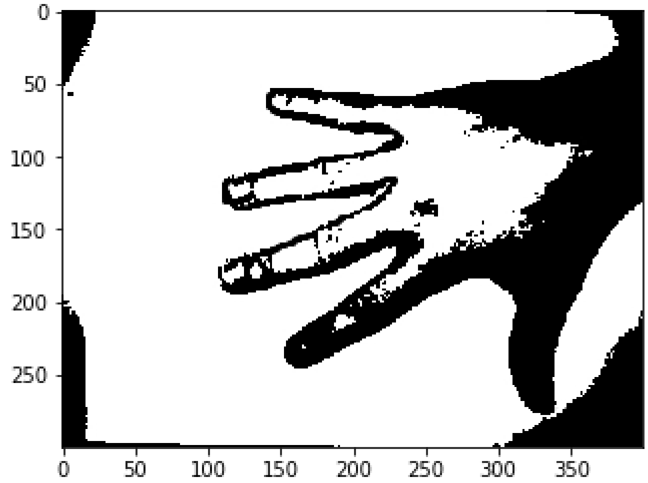
```
def img_preprocess_0(img):
    kernel_10x10 = np.ones((10,10),np.uint8)
    img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_10x10) # Open. Fill in any holes.
    img = cv2.GaussianBlur(img, (5,5), 0)
    img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_10x10) # Open. Fill in any holes.
    img = cv2.GaussianBlur(img, (5,5), 0)

    _, img = cv2.threshold(img,250,255,cv2.THRESH_BINARY)
    return img

def img_preprocess_1(img):
    kernel_2x2 = np.ones((2,2),np.uint8)
    img = cv2.erode(img, kernel_2x2, iterations=1)
    img = cv2.GaussianBlur(img, (5,5), 0)
    img = cv2.dilate(img, kernel_2x2, iterations=1)
    _, img = cv2.threshold(img,175,255,cv2.THRESH_BINARY)
    return img

def img_preprocess_2(img):
    kernel_5x5 = np.ones((5,5),np.uint8)
    kernel_3x3 = np.ones((3,3),np.uint8)

    img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_5x5) # Open. Fill in any holes.
    img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_3x3) # Close. Remove small blob
    _, img = cv2.threshold(img,100,255,cv2.THRESH_BINARY)
    return img

def img_preprocess_3(img):
    kernel_10x10 = np.ones((10,10),np.uint8)
    kernel_7x7 = np.ones((7,7),np.uint8)
    kernel_3x3 = np.ones((3,3),np.uint8)

    img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_10x10) # Close. Remove small blob
    img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_10x10) # Close. Remove small blob
    img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_7x7) # Close. Remove small blob
    img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_3x3) # Close. Remove small blob
    _, img = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
    return img
```

In [342]:

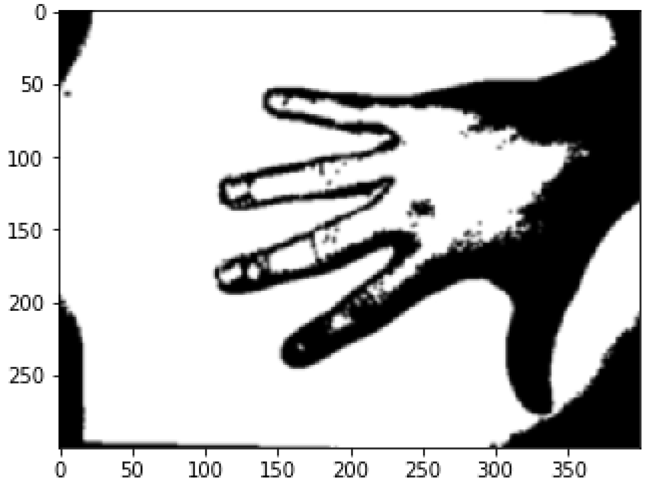
```
# img_path = options['img_dir'] + "/" + 'open-bw-partial.png'
# img = cv2.imread(img_path)

# plt.imshow(img)
# plt.show()

# img = img_preprocess_1(img)

# plt.imshow(img)
# plt.show()
```

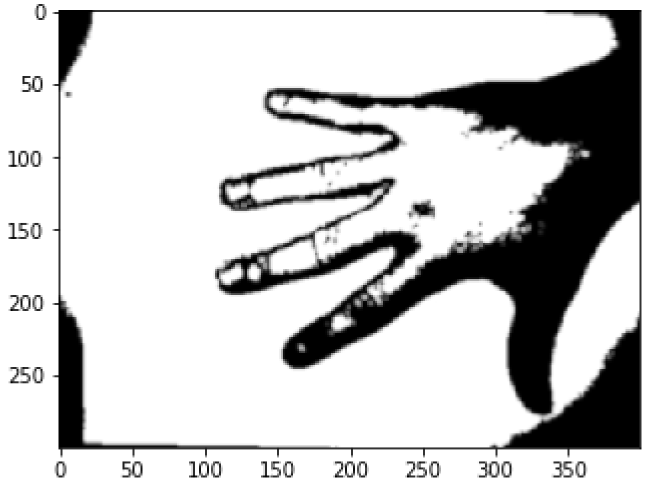

0
50
100
150
200
250
0
50
100
150
200
250
300
350

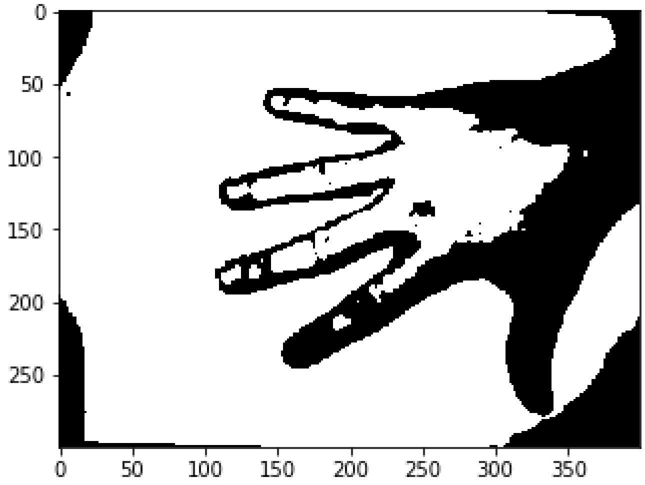

0
50
100
150
200
250
0
50
100
150
200
250
300
350

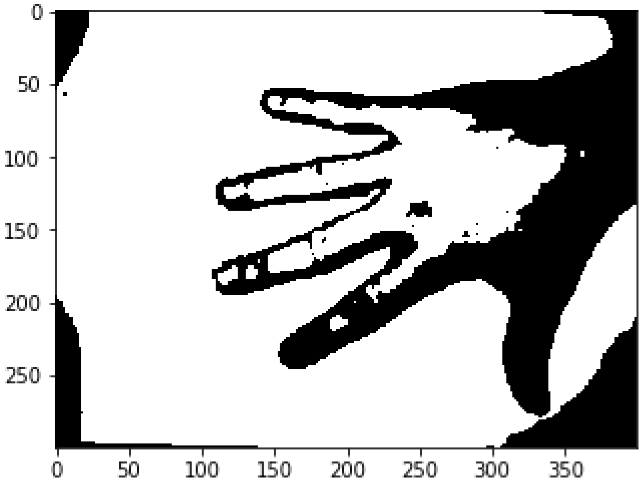

0
50
100
150
200
250
0
50
100
150
200
250
300
350

BLUR


0
50
100
150
200
250
0
50
100
150
200
250
300
350


0
50
100
150
200
250
0
50
100
150
200
250
300
350


0
50
100
150
200
250
0
50
100
150
200
250
300
350


0
50
100
150
200
250
0
50
100
150
200
250
300
350

In [344]:

```
def get_connected_components(img_path, preprocess_mode, n=2, debug=False
, output_res=False):
    # Open Hand
    print("img path=", img_path)  # options['img_dir'] + "/" + 'open-bw-
full.png'
    img = cv2.imread(img_path)

    if debug:
        print("Original")
        plt.imshow(img)
        plt.show()

    if preprocess_mode == 0:
        img = img_preprocess_0(img)
    if preprocess_mode == 1:
        img = img_preprocess_1(img)
    elif preprocess_mode == 2:
        img = img_preprocess_2(img)
    else:
        img = img_preprocess_3(img)

    img, usedColors, = flood_fill_multi(img, debug=debug)

    if debug:
        print("After preprocess and floodfill multi")
        plt.imshow(img)
        plt.show()

    largest_colors = get_largest_components(img, usedColors, n=n)
    if debug:
        print("Largest_colors", largest_colors)
        plt.imshow(np.array([largest_colors]))
        plt.show()

    if n > 1:
        largest_colors = largest_colors[1:]

    img = filter_out_colors(img, largest_colors, [255, 255, 255])
    if debug or output_res:
        print("After filter out smallest colors")
        plt.imshow(img)
        plt.show()

    return img, largest_colors
```

In [156]:

```
def get_next_cw_pos(center, curr): # TODO, p = center, b = current pos
#     '''
#     C is left of center.
#     [[...],
#      [C center X]]
#     '''
    if curr[1] == center[1] and curr[0]+1 == center[0]:
        return [curr[0], curr[1]-1]
#     '''
#     C is left-top of center.
#     [[C X X],
#      [X center X]]
#      '''
    elif curr[1]+1 == center[1] and curr[0]+1 == center[0]:
        return [curr[0]+1, curr[1]]
#     '''
#     C is top of center.
#     [[X C X],
#      [X center X]]
#     '''
    elif curr[1]+1 == center[1] and curr[0] == center[0]:
        return [curr[0]+1, curr[1]]
#     '''
#     C is top-right of center.
#     [[X X C],
#      [X center X]]
#     '''
    elif curr[1]+1 == center[1] and curr[0]-1 == center[0]:
        return [curr[0], curr[1]+1]
#     '''
#     C is right of center.
#     [[X X X],
#      [X center C]]
#     '''
    elif curr[1] == center[1] and curr[0]-1 == center[0]:
        return [curr[0], curr[1]+1]
#     '''
#     C is right-bot of center.
#     [[X X X],
#      [X center X],
#      [X X C]]
#     '''
    elif curr[1]-1 == center[1] and curr[0]-1 == center[0]:
        return [curr[0]-1, curr[1]]
#     '''
#     C is bot of center.
#     [[X X X],
#      [X center X],
#      [X C X]]
#     '''
    elif curr[1]-1 == center[1] and curr[0] == center[0]:
        return [curr[0]-1, curr[1]]
#     '''
#     C is left-bot of center.
#     [[X X X],
#      [X center X],
#      [C X X]]
#     '''
    elif curr[1]-1 == center[1] and curr[0]+1 == center[0]:
        return [curr[0], curr[1]-1]
#     '''
#     C is left of center.
#     [[X X X],
#      [C center X],
#      [X X X]]
#     '''
    elif curr[1] == center[1] and curr[0]+1 == center[0]:
        return [curr[0], curr[1]-1]
    else:
        print("ERROR")
```

In [238]:

```
def test_get_next_cw_pos():
    # Test get_next_cw_pos
    img = np.array([[[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]]])

    center = [1,1]

    in_coord = [0,0]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [1,0]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [2,0]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [2,1]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [2,2]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [1,2]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [0,2]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))
    in_coord = [0,1]
    print("in({}). out({})".format(in_coord, get_next_cw_pos(center, in_
coord)))

test_get_next_cw_pos()
```

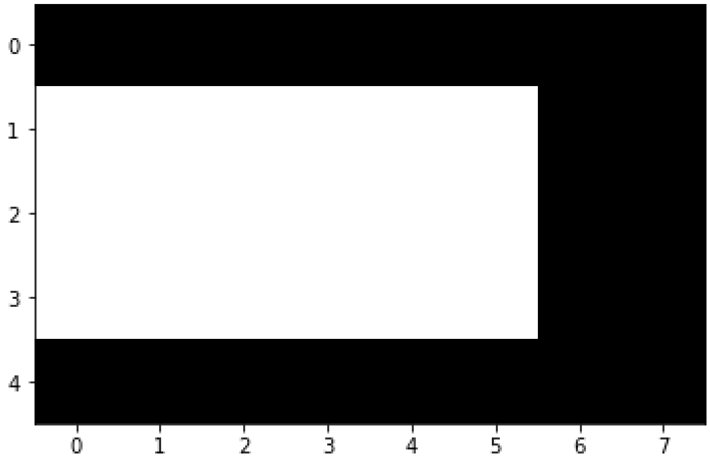
```
in([0, 0]). out([1, 0])
in([1, 0]). out([2, 0])
in([2, 0]). out([2, 1])
in([2, 1]). out([2, 2])
in([2, 2]). out([1, 2])
in([1, 2]). out([0, 2])
in([0, 2]). out([0, 1])
in([0, 1]). out([0, 0])
```

In [376]:

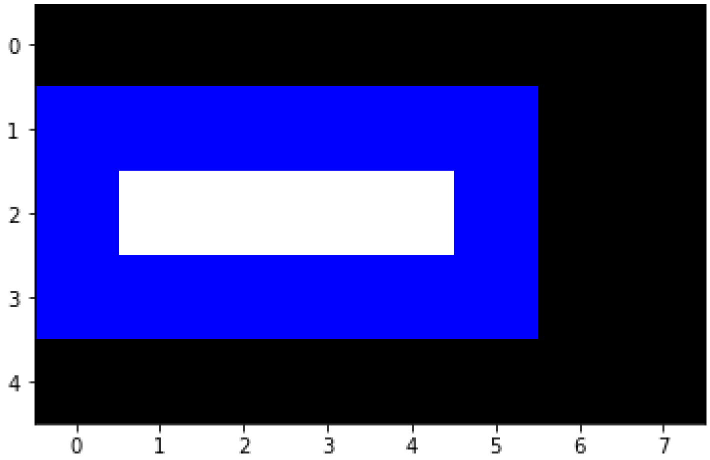
```
def boundary_tracing(img, target_colors, boundary_draw_color, debug=False):

    print("debug={}".format(debug))

    B = []
    ptColor = [255,255,255]
    start = None

#     print("shape:", img.shape)

    #   From bottom to top and left to right scan the cells of T until a black pixel, s, of P is found.
    for j in range(img.shape[0]):
        if start is not None:
            break
        for i in range(img.shape[1]):

            if start is not None:
                break

            if np.any([np.all(img[j][i] == tc) for tc in target_colors]): # is ptColor
                start = [i,j]
                if debug:
                    print("Found first black pixel (i,j) = ({})".format(start))

    if start is None:
        print("ERROR | Start is None")
        return None, None

    B.append(start)
    p = start
    b = [start[0]-1, start[1]] # TODO: border handle cases.
    c = get_next_cw_pos(p, b)
    if debug:
        print("About to start. Next move is c={}, b={}".format(c,b))

    while not np.all(c == start): # while c != start
        if c[0] < 0 or c[0] > img.shape[1]-1 or c[1] < 0 or c[1] > img.shape[0]-1: # Out of bounds
            b = c
            c = get_next_cw_pos(p, b)
            if debug:
                print("out of bounds. Continue . Next move is c={}, b={}".format(c,b))
        elif np.any([np.all(img[c[1]][c[0]] == tc) for tc in target_colors]): # color at c is pointColor
            B.append(c)
            p = c
            c = get_next_cw_pos(p, b)
            if debug:
                print("Add c into B. Next move is B={}, c={}, b={}.".format(B,c,b))
        else:
            b = c
            c = get_next_cw_pos(p, b)
            if debug:
                print("No find black pixel. Next move is c={}, b={}".format(c,b))


    # Draw Boundary with orig
    boundary_overlay_img = img.copy()
    for b_coord in B:
        boundary_overlay_img[b_coord[1]][b_coord[0]] = boundary_draw_color

    # Draw just boundary
    boundary_img = np.ones(img.shape) * 255
    for boundary in B:
        boundary_img[boundary[1], boundary[0]] = (0,0,0)

    return B, boundary_overlay_img, boundary_img
```

In [246]:

```
def test_boundary_tracing():
    # Input
    img = np.array([[[0,0,0], [0,0,0], [0,0,0], [0,0,0], [0,0,0], [0,0,0
], [0,0,0], [0,0,0]],
                    [[255,255,255], [255,255,255], [255,255,255], [255,255,
255], [255,255,255], [255,255,255], [0,0,0], [0,0,0]],
                    [[255,255,255], [255,255,255], [255,255,255], [255,255,
255], [255,255,255], [255,255,255], [0,0,0], [0,0,0]],
                    [[255,255,255], [255,255,255], [255,255,255], [255,255,
255], [255,255,255], [255,255,255], [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]]])

    plt.imshow(img)
    plt.show()

    # Boundary Tracing Algorithm
    boundary, boundary_img = boundary_tracing(img, [[255,255,255]], [0,0
,255], debug=True)
    print("boundary=", boundary)
    plt.imshow(boundary_img)
    plt.show()
test_boundary_tracing()
```


0
1
2
3
4
0
1
2
3
4
5
6
7

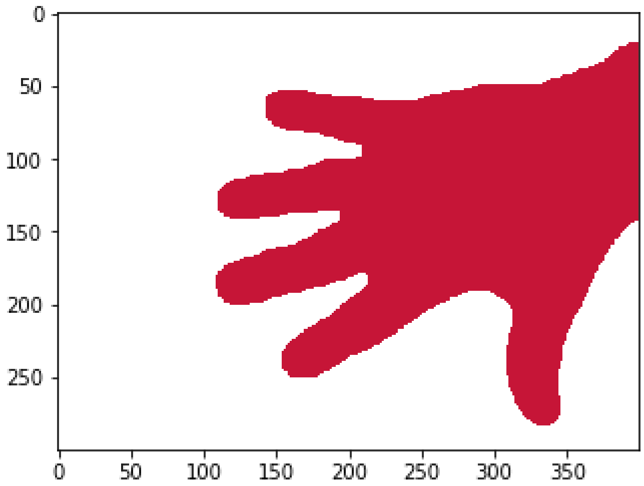
```
debug=True
Found first black pixel (i,j) = ([0, 1])
About to start. Next move is c=[-1, 0], b=[-1, 1]
out of bounds
No find black pixel. Next move is c=[1, 0], b=[0, 0]
No find black pixel. Next move is c=[1, 1], b=[1, 0]
Add c into B. Next move is B=[[0, 1], [1, 1]], c=[2, 0], b=[1, 0].
No find black pixel. Next move is c=[2, 1], b=[2, 0]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1]], c=[3, 0], b=[2,
0].
No find black pixel. Next move is c=[3, 1], b=[3, 0]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1]], c=[4,
0], b=[3, 0].
No find black pixel. Next move is c=[4, 1], b=[4, 0]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1]],
c=[5, 0], b=[4, 0].
No find black pixel. Next move is c=[5, 1], b=[5, 0]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1]], c=[6, 0], b=[5, 0].
No find black pixel. Next move is c=[6, 1], b=[6, 0]
No find black pixel. Next move is c=[6, 2], b=[6, 1]
No find black pixel. Next move is c=[5, 2], b=[6, 2]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2]], c=[6, 3], b=[6, 2].
No find black pixel. Next move is c=[5, 3], b=[6, 3]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3]], c=[6, 4], b=[6, 3].
No find black pixel. Next move is c=[5, 4], b=[6, 4]
No find black pixel. Next move is c=[4, 4], b=[5, 4]
No find black pixel. Next move is c=[4, 3], b=[4, 4]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3], [4, 3]], c=[3, 4], b=[4, 4].
No find black pixel. Next move is c=[3, 3], b=[3, 4]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3], [4, 3], [3, 3]], c=[2, 4], b=[3, 4].
No find black pixel. Next move is c=[2, 3], b=[2, 4]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3], [4, 3], [3, 3], [2, 3]], c=[1, 4], b=[2, 4].
No find black pixel. Next move is c=[1, 3], b=[1, 4]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3], [4, 3], [3, 3], [2, 3], [1, 3]], c=[0, 4], b=
[1, 4].
No find black pixel. Next move is c=[0, 3], b=[0, 4]
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3], [4, 3], [3, 3], [2, 3], [1, 3], [0, 3]], c=[-1,
4], b=[0, 4].
out of bounds
out of bounds
out of bounds
Add c into B. Next move is B=[[0, 1], [1, 1], [2, 1], [3, 1], [4, 1],
[5, 1], [5, 2], [5, 3], [4, 3], [3, 3], [2, 3], [1, 3], [0, 3], [0,
2]], c=[-1, 1], b=[-1, 2].
out of bounds
boundary= [[0, 1], [1, 1], [2, 1], [3, 1], [4, 1], [5, 1], [5, 2], [5,
3], [4, 3], [3, 3], [2, 3], [1, 3], [0, 3], [0, 2]]
```


0
1
2
3
4
0
1
2
3
4
5
6
7

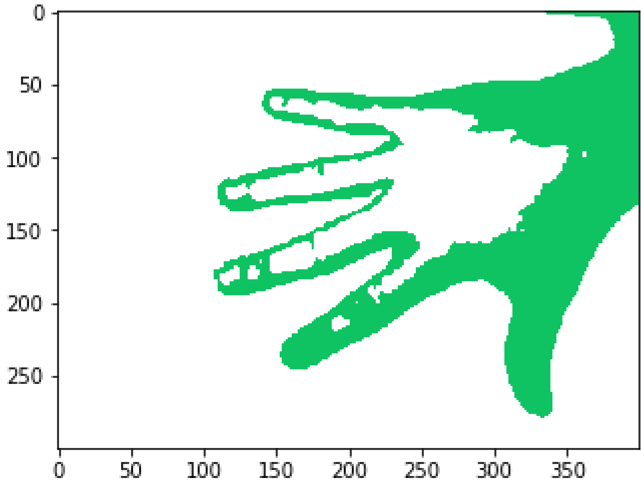
In [345]:

```python
debug = False
output_res = False
open_full_cc, open_full_colors = get_connected_components(options['img_dir'] + "/" + 'open-bw-full.png', preprocess_mode=0, n=2, debug=debug, output_res=output_res)
plt.imshow(open_full_cc)
plt.show()

open_partial_cc, open_partial_colors = get_connected_components(options['img_dir'] + "/" + 'open-bw-partial.png', preprocess_mode=1, n=2, debug=debug, output_res=output_res)
plt.imshow(open_partial_cc)
plt.show()

open_fist_cc, open_fist_colors = get_connected_components(options['img_dir'] + "/" + 'open_fist-bw.png', preprocess_mode=2, debug=debug, n=3, output_res=output_res)
plt.imshow(open_fist_cc)
plt.show()

tumor_cc, tumor_colors = get_connected_components(options['img_dir'] + "/" + 'tumor-fold.png', preprocess_mode=3, debug=debug, n=2, output_res=output_res)
plt.imshow(tumor_cc)
plt.show()
```

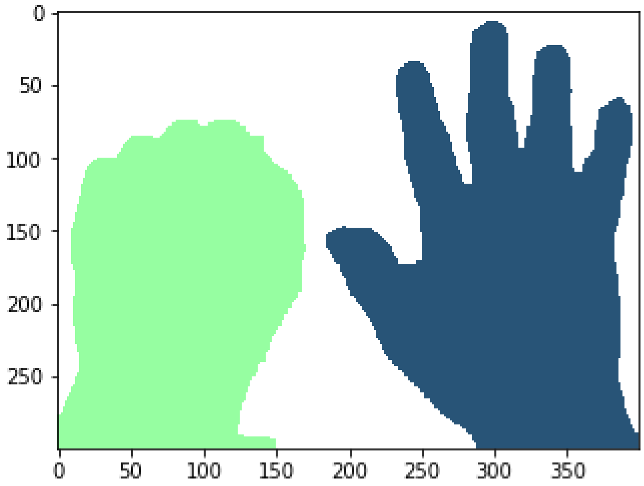
```
img path= ./img/open-bw-full.png
```



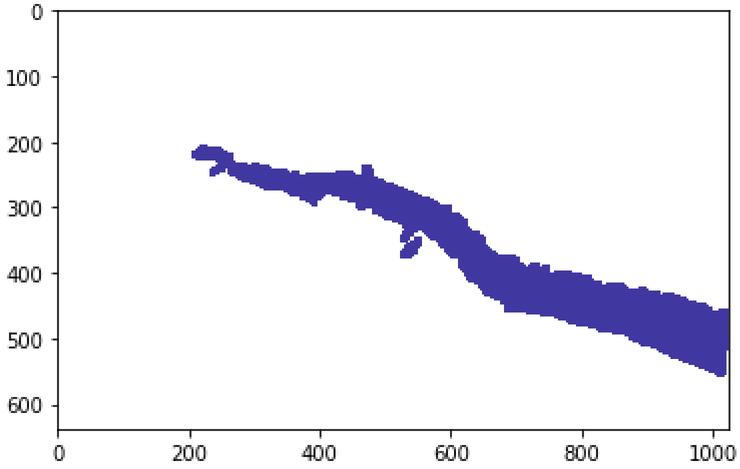
```
img path= ./img/open-bw-partial.png
```



```
img path= ./img/open_fist-bw.png
```



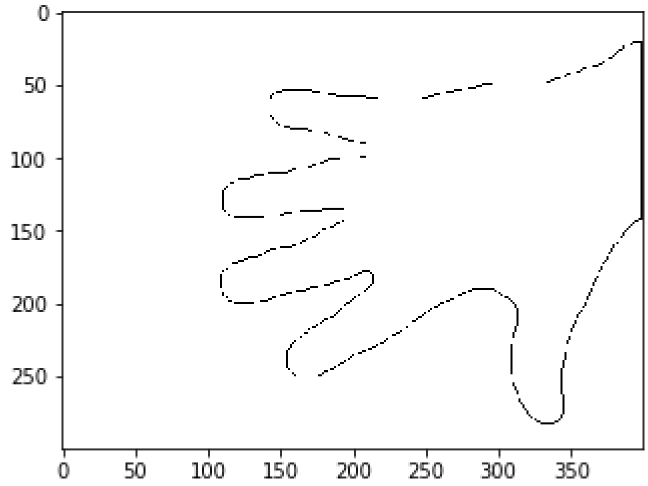
```
img path= ./img/tumor-fold.png
```


0
100
200
300
400
500
600
0
200
400
600
800
1000

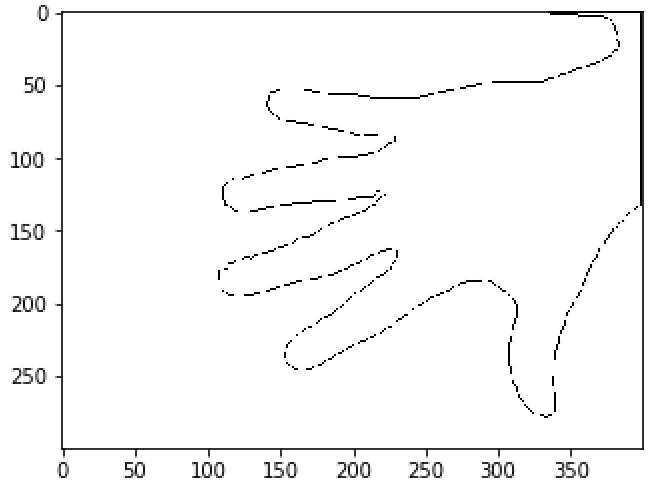
In [379]:

```
open_full_boundaries, open_full_overlay, open_full_boundary_img = boundary_tracing(open_full_cc, open_full_colors, [0,0,0], debug=False)
plt.imshow(open_full_boundary_img)
plt.show()
cv2.imwrite('./result/open_full_boundary_img.png', open_full_boundary_img)

open_partial_boundary, open_partial_overlay, open_partial_boundary_img = boundary_tracing(open_partial_cc, open_partial_colors, [0,0,0], debug=False)
plt.imshow(open_partial_boundary_img)
plt.show()
cv2.imwrite('./result/open_partial_boundary_img.png', open_partial_boundary_img)

open_fist_boundary_1, open_fist_overlay_1, open_fist_boundary_img_1 = boundary_tracing(open_fist_cc, [open_fist_colors[0]], [0,0,0], debug=False)
open_fist_boundary_2, open_fist_overlay_2, open_fist_boundary_img_2 = boundary_tracing(open_fist_cc, [open_fist_colors[1]], [0,0,0], debug=False)
for i in range(open_fist_boundary_img_1.shape[1]):
    for j in range(open_fist_boundary_img_1.shape[0]):
        if np.all(open_fist_boundary_img_2[j][i] == [0,0,0]):
            open_fist_boundary_img_1[j][i] = [0,0,0]
plt.imshow(open_fist_boundary_img_1)
plt.show()
cv2.imwrite('./result/open_fist_boundary_img.png', open_fist_boundary_img_1)

tumor_boundary, tumor_overlay, tumor_boundary_img = boundary_tracing(tumor_cc, tumor_colors, [0,0,0], debug=False)
plt.imshow(tumor_boundary_img)
plt.show()
cv2.imwrite('./result/tumor_boundary_img.png', tumor_boundary_img)
```

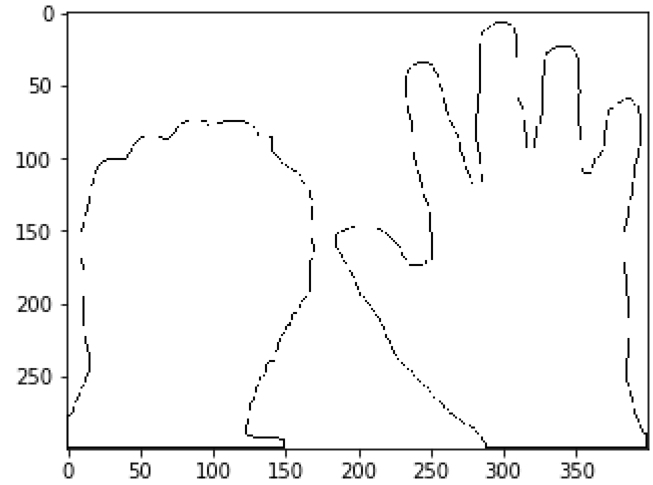
```
debug=False

Clipping input data to the valid range for imshow with RGB data ([0..1]
for floats or [0..255] for integers).
```



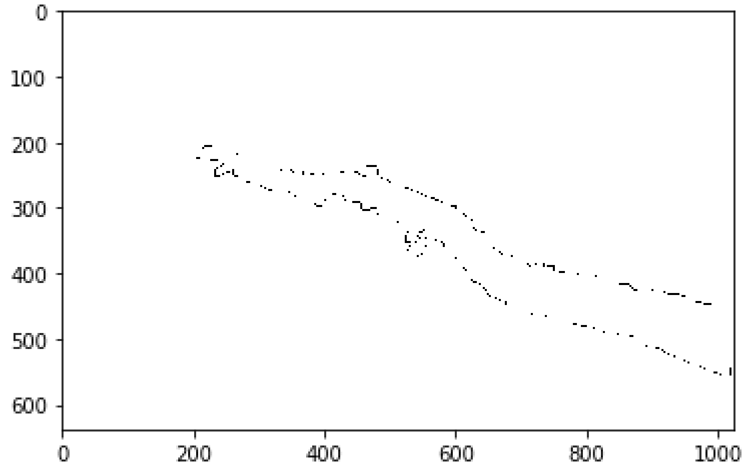
```
Clipping input data to the valid range for imshow with RGB data ([0..1]
for floats or [0..255] for integers).

debug=False
```



```
debug=False
debug=False

Clipping input data to the valid range for imshow with RGB data ([0..1]
for floats or [0..255] for integers).
```



```
debug=False

Clipping input data to the valid range for imshow with RGB data ([0..1]
for floats or [0..255] for integers).
```

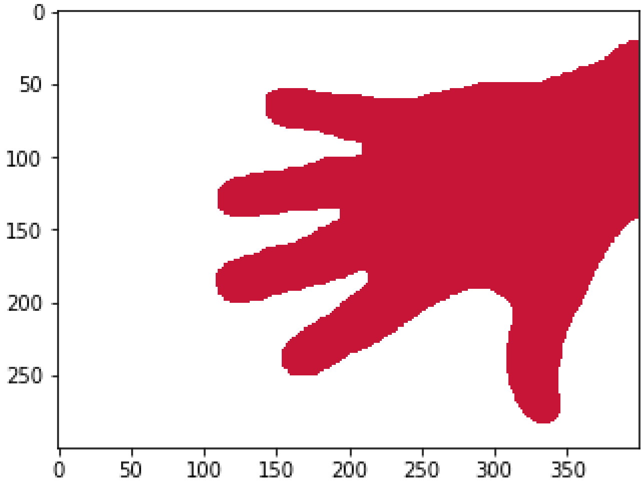


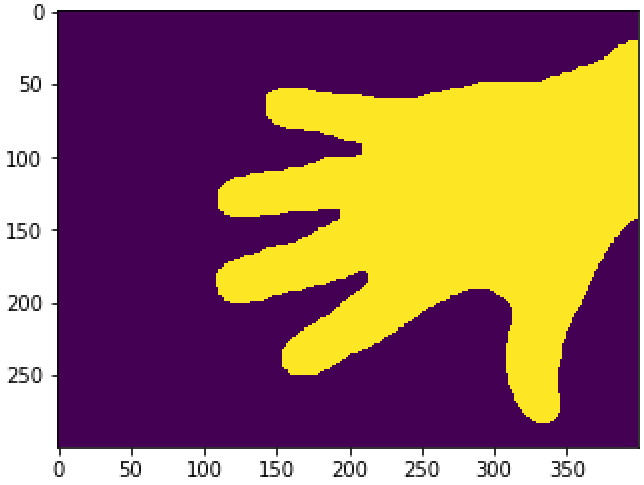
```
Out[379]: True
```

In [401]:

```
def skeletonize(img):
    """ OpenCV function to return a skeletonized version of img, a Mat o
bject"""

    #  hat tip to http://felix.abecassis.me/2011/09/opencv-morphological
-skeleton/
    ret,img = cv2.threshold(img,127,255,0)

    img = img.copy() # don't clobber original
    skel = img.copy()

    skel[:,:] = 0
    kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))

    count = 0

    while True:
        eroded = cv2.morphologyEx(img, cv2.MORPH_ERODE, kernel)
#         plt.imshow(eroded)
#         plt.title("Eroded")
#         plt.show()

        temp = cv2.morphologyEx(eroded, cv2.MORPH_DILATE, kernel)
#         plt.imshow(temp)
#         plt.title("Dilate")
#         plt.show()

        temp  = cv2.subtract(img, temp)
#         plt.imshow(temp)
#         plt.title("substract")
#         plt.show()

        skel = cv2.bitwise_or(skel, temp)
#         plt.imshow(skel)
#         plt.title("bitwise or")
#         plt.show()

        img[:,:] = eroded[:,:]
        count += 1
        print("count=", count)
        if cv2.countNonZero(img) == 0:
            break


    return skel

# Open Full Hand
plt.imshow(open_full_cc)
plt.show()

open_full_gs = cv2.cvtColor(open_full_cc, cv2.COLOR_BGR2GRAY)
open_full_gs = cv2.bitwise_not(open_full_gs)
plt.imshow(open_full_gs)
plt.show()

open_full_skeleton = skeletonize(open_full_gs)
```

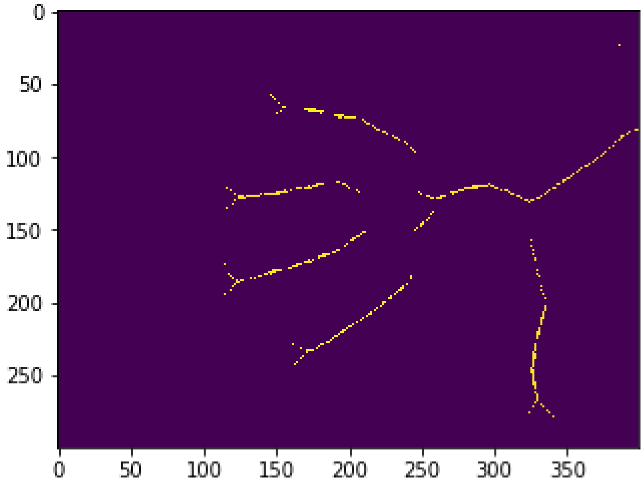
```
plt.imshow(open_full_skeleton)
plt.show()

# Open Hand Partial
plt.imshow(open_partial_cc)
plt.show()

open_partial_gs = cv2.cvtColor(open_partial_cc, cv2.COLOR_BGR2GRAY)
open_partial_gs = cv2.bitwise_not(open_partial_gs)
plt.imshow(open_partial_gs)
plt.show()

open_partial_skeleton = skeletonize(open_partial_gs)
plt.imshow(open_partial_skeleton)
plt.show()

# Open Fist
plt.imshow(open_fist_cc)
plt.show()

open_fist_gs = cv2.cvtColor(open_fist_cc, cv2.COLOR_BGR2GRAY)
_, open_fist_gs = cv2.threshold(open_fist_gs,250,255,cv2.THRESH_BINARY)
open_fist_gs = cv2.bitwise_not(open_fist_gs)
plt.imshow(open_fist_gs)
plt.show()

open_fist_skeleton = skeletonize(open_fist_gs)
plt.imshow(open_fist_skeleton)
plt.show()

# Tumor
plt.imshow(tumor_cc)
plt.show()

tumor_gs = cv2.cvtColor(tumor_cc, cv2.COLOR_BGR2GRAY)
tumor_gs = cv2.bitwise_not(tumor_gs)
plt.imshow(tumor_gs)
plt.show()

tumor_skeleton = skeletonize(tumor_gs)
plt.imshow(tumor_skeleton)
plt.show()
```

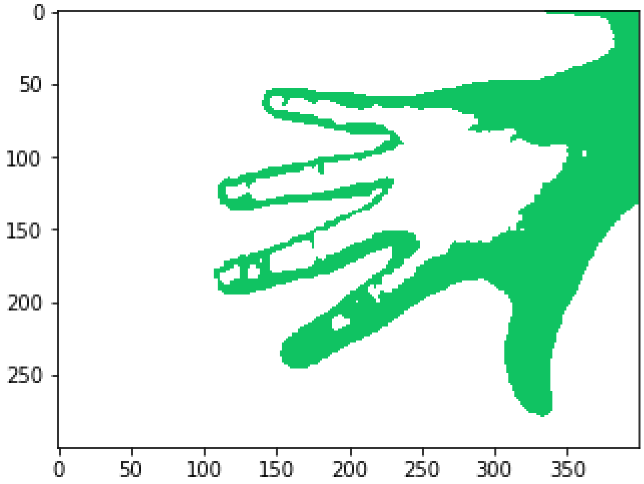

0
50
100
150
200
250
0
50
100
150
200
250
300
350

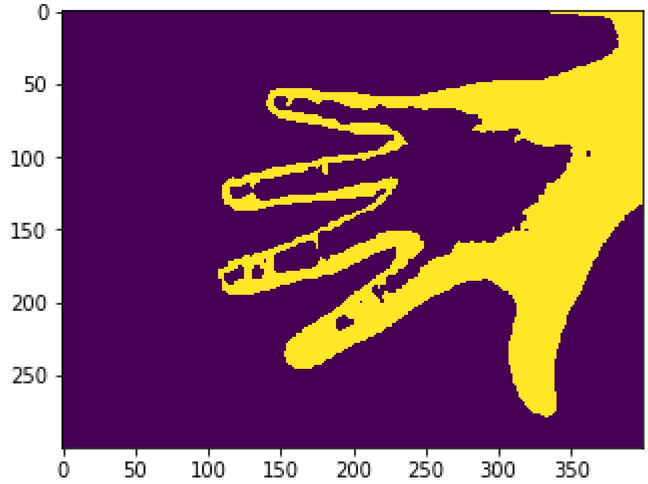

0
50
100
150
200
250
0
50
100
150
200
250
300
350

```
count= 1
count= 2
count= 3
count= 4
count= 5
count= 6
count= 7
count= 8
count= 9
count= 10
count= 11
count= 12
count= 13
count= 14
count= 15
count= 16
count= 17
count= 18
count= 19
count= 20
count= 21
count= 22
count= 23
count= 24
count= 25
count= 26
count= 27
count= 28
count= 29
count= 30
count= 31
count= 32
count= 33
count= 34
count= 35
count= 36
count= 37
count= 38
count= 39
count= 40
count= 41
count= 42
count= 43
count= 44
count= 45
count= 46
count= 47
count= 48
count= 49
count= 50
count= 51
count= 52
count= 53
count= 54
count= 55
count= 56
count= 57
```

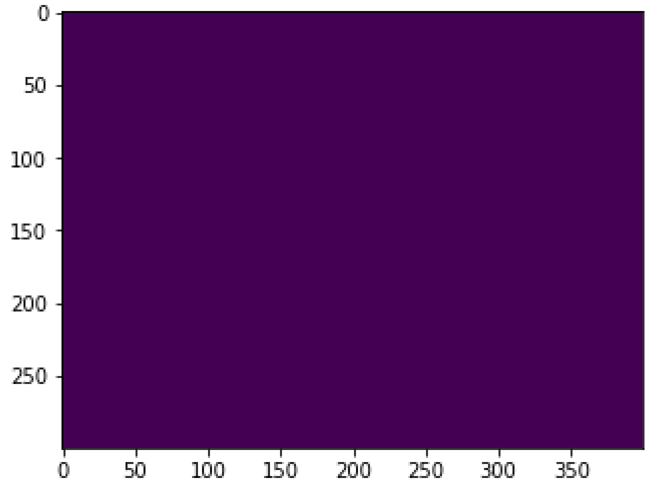
```
count= 58
count= 59
count= 60
count= 61
count= 62
count= 63
count= 64
count= 65
count= 66
count= 67
count= 68
count= 69
count= 70
count= 71
count= 72
count= 73
count= 74
count= 75
count= 76
count= 77
count= 78
count= 79
count= 80
count= 81
count= 82
count= 83
```

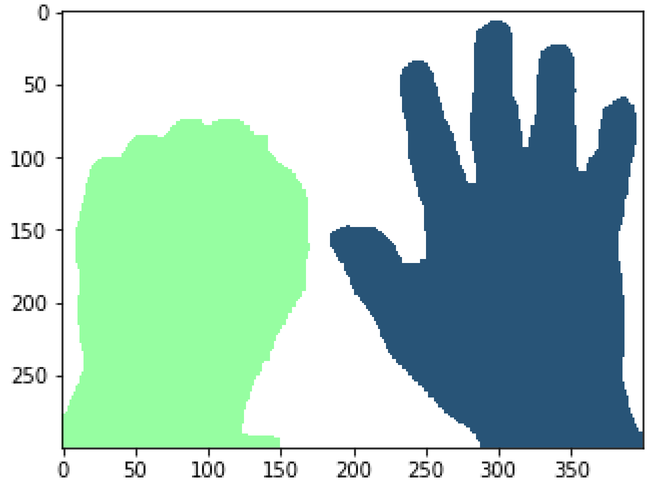


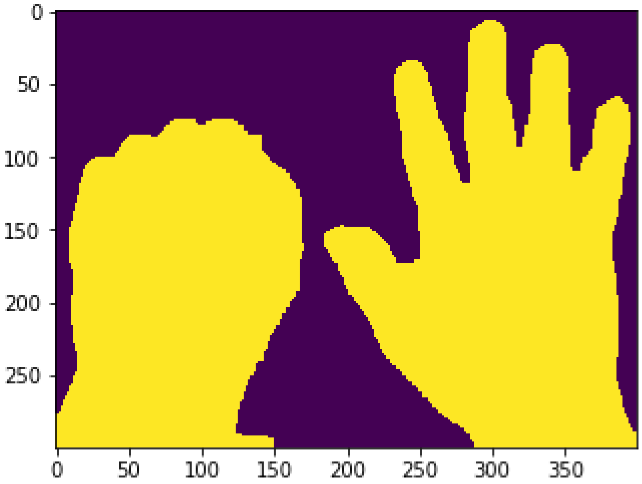



0
50
100
150
200
250
0
50
100
150
200
250
300
350

```
count= 1
```

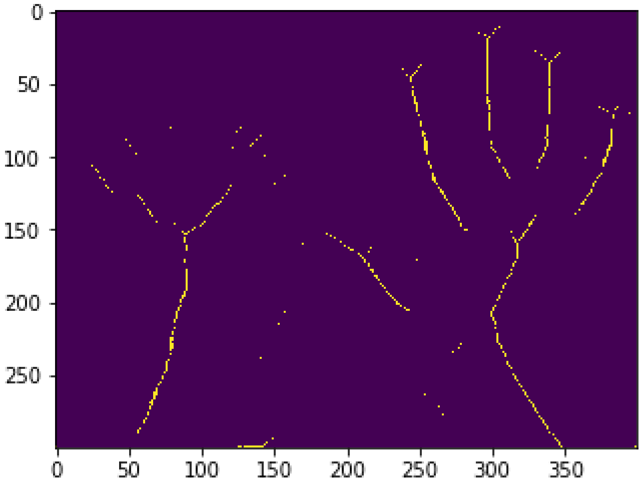

0
50
100
150
200
250
0
50
100
150
200
250
300
350

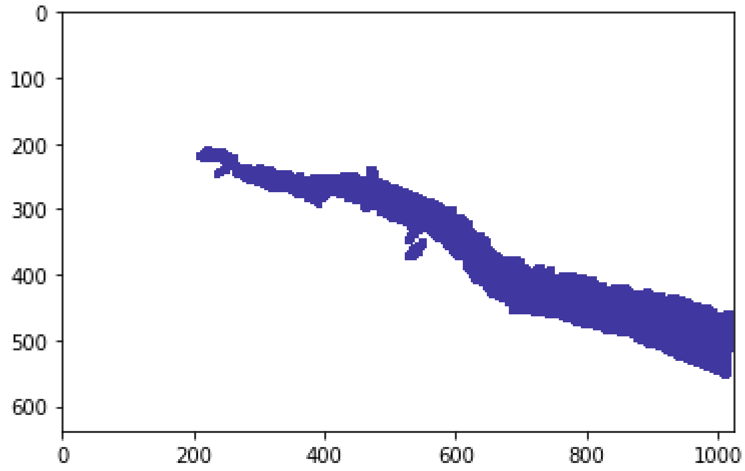

0
50
100
150
200
250
0
50
100
150
200
250
300
350

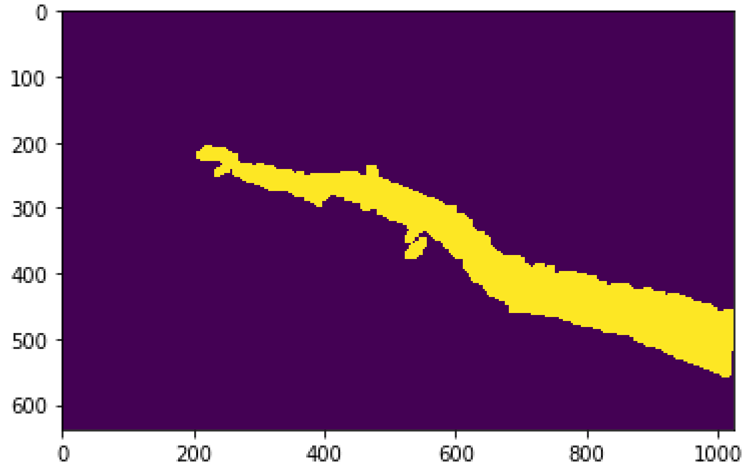

0
50
100
150
200
250
0
50
100
150
200
250
300
350

```
count= 1
count= 2
count= 3
count= 4
count= 5
count= 6
count= 7
count= 8
count= 9
count= 10
count= 11
count= 12
count= 13
count= 14
count= 15
count= 16
count= 17
count= 18
count= 19
count= 20
count= 21
count= 22
count= 23
count= 24
count= 25
count= 26
count= 27
count= 28
count= 29
count= 30
count= 31
count= 32
count= 33
count= 34
count= 35
count= 36
count= 37
count= 38
count= 39
count= 40
count= 41
count= 42
count= 43
count= 44
count= 45
count= 46
count= 47
count= 48
count= 49
count= 50
count= 51
count= 52
count= 53
count= 54
count= 55
count= 56
count= 57
```

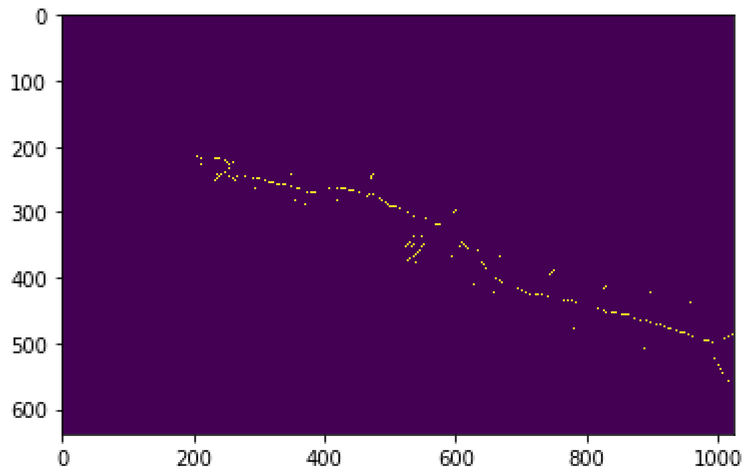
```
count= 58
count= 59
count= 60
count= 61
count= 62
count= 63
count= 64
count= 65
count= 66
count= 67
count= 68
count= 69
count= 70
count= 71
count= 72
count= 73
count= 74
count= 75
count= 76
count= 77
count= 78
count= 79
count= 80
count= 81
count= 82
count= 83
count= 84
count= 85
count= 86
count= 87
```




0
100
200
300
400
500
600
0
200
400
600
800
1000


0
100
200
300
400
500
600
0
200
400
600
800
1000

```
count= 1
count= 2
count= 3
count= 4
count= 5
count= 6
count= 7
count= 8
count= 9
count= 10
count= 11
count= 12
count= 13
count= 14
count= 15
count= 16
count= 17
count= 18
count= 19
count= 20
count= 21
count= 22
count= 23
count= 24
count= 25
count= 26
count= 27
count= 28
count= 29
count= 30
count= 31
count= 32
count= 33
count= 34
count= 35
count= 36
count= 37
count= 38
count= 39
count= 40
count= 41
count= 42
count= 43
count= 44
count= 45
count= 46
count= 47
count= 48
count= 49
count= 50
count= 51
count= 52
```



In [ ]:

In [ ]: