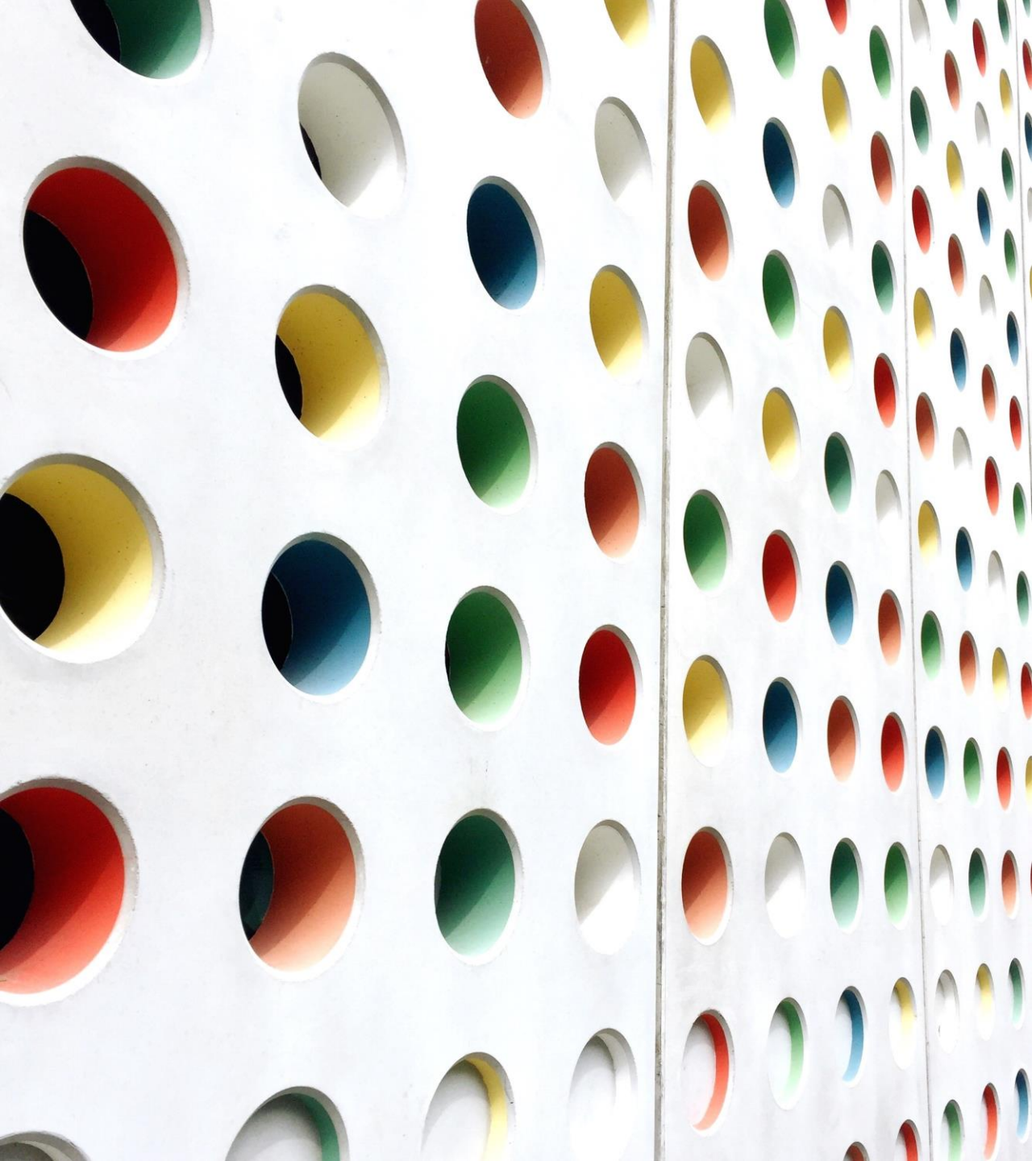


Computer Vision Homework 4

Matt Ferguson

August 3rd , 2022

Requirements

For each image:

- Load the image to memory and convert to grayscale.
- Binarize the image using Otsu's and inverse thresholding.
- Add 1-pixel of black padding.
- Write the binarized image to file.
- Implement the Pavlidis contour algorithm and extract the object contour.
- Remove the least relevant half of vertices 8 times using discrete curve evolution. Write the resulting contours as images.
- Calculate the flood area of the original and evolved contours.
- Calculate the Gauss area of the original and evolved contours.
- Print the filename, number of vertices, the flood area and the Gauss area of the original and evolved contours.

Criteria
Load, binarize and write image
Contour extraction (your Pavlidis function)
Area estimation from contour points (your Gauss area function)
Point removal from contour (your OnePassDCE function)
Proper console output

Pavlidis Function

Here we implement our Pavlidis contour extraction function. We input a binary image. Our first while loop extracts the contour start point as the first white pixel at the vertical center (halfway down the image) when approaching the contour from the left. Our Pavlidis function traces the contour of the binary image and outputs a set of vertices as (x, y) points in a NumPy array. We call our Pavlidis contour function in our main script. This is a more efficient implementation of Pavlidis in our opinion as our direction is represented by a list with 4 elements, and rotation is accomplished by adding or subtracting by 1. This direction list can be visualized as a compass. This potentially alleviates a large decision-tree like algorithm structure.

```
def pavlidis(img):
    x,c=0,0
    y=img.shape[0]//2

    while c == 0:
        x+=1
        if img[y,x]!=0:
            c=1

    b1_value,b2_value,b3_value = 0,0,0
    b1_coord,b2_coord,b3_coord= 0,0,0
    directions = 'up', 'right', 'down', 'left'
    direction='right'
    input=(y,x)
    result=np.array([input])

    c=0
    start_pos=input
    while c<2:

        y,x=input
        if input==list(start_pos):
            c=c+1

        if direction =='up':
            b1_value, b1_coord=img[y-1,x-1], [y-1,x-1]
            b2_value, b2_coord=img[y-1,x], [y-1,x]
            b3_value, b3_coord=img[y-1,x+1], [y-1,x+1]
        elif direction =='right':
            b1_value, b1_coord=img[y-1,x+1], [y-1,x+1]
            b2_value, b2_coord=img[y,x+1], [y,x+1]
            b3_value, b3_coord=img[y+1,x+1], [y+1,x+1]
        elif direction =='down':
            b1_value, b1_coord=img[y+1,x+1], [y+1,x+1]
            b2_value, b2_coord=img[y+1,x], [y+1,x]
            b3_value, b3_coord=img[y+1,x-1], [y+1,x-1]
        elif direction =='left':
            b1_value, b1_coord=img[y+1,x-1], [y+1,x-1]
            b2_value, b2_coord=img[y,x-1], [y,x-1]
            b3_value, b3_coord=img[y-1,x-1], [y-1,x-1]

        block_values= b1_value,b2_value,b3_value
        block_coords=b1_coord,b2_coord,b3_coord

        if b1_value==255:
            bimg[y,x]=255
            direction=directions[(((directions.index(direction)-1))%4)]
        elif b1_value==0 and b2_value==0 and b3_value==0:
            direction=directions[(((directions.index(direction)+1))%4)]
        elif b2_value==255:
            bimg[y,x]=255
        elif b3_value==255:
            bimg[y,x]=255

        for i, value in enumerate(reversed(block_values)):
            if value==255:
                input = block_coords[2-i]

        result=np.append(result,[input],axis=0)

    result, ind=np.unique(result,axis=0, return_index=True)
    result=result[np.argsort(ind)]
    return (result)
```

Gauss and Flood Area Functions

Here are the functions we call in the main script to compute the area of our contour polygons. We call these functions for our initial contour and for each contour resulting from discrete curve evolution iteration. Notably the flood area and Gauss area agreed on the hand0.png area value with a negligible margin of difference. However, there was a notable discrepancy between the US.png area values. The provided flood fill area function was about half the Gauss area for the US.png polygons.

```
def gaussarea(x,y):
    gauss_area=0.5*np.abs(np.dot(x,np.roll(y,-1))-np.dot(y,np.roll(x,-1)))
    return(gauss_area)
```

```
def fillarea(ctr):
    maxx = np.max(ctr[:, 0]) + 1
    maxy = np.max(ctr[:, 1]) + 1
    contourImage = np.zeros( (maxy, maxx) )
    length = ctr.shape[0]
    for count in range(length):
        contourImage[ctr[count, 1], ctr[count, 0]] = 255
        cv2.line(contourImage, (ctr[count, 0], ctr[count, 1]), \
        (ctr[(count + 1) % length, 0], ctr[(count + 1) % length, 1]), \
        (255, 0, 255), 1)
    fillMask = cv2.copyMakeBorder(contourImage, 1, 1, 1, 1, \
    cv2.BORDER_CONSTANT, 0).astype(np.uint8)
    areaImage = np.zeros((maxy, maxx), np.uint8)
    startPoint = (int(maxy/2), int(maxx/2))
    cv2.floodFill(areaImage, fillMask, startPoint, 128)
    area = np.sum(areaImage)/128
    return area
```

Discrete Curve Evolution Function

Here is the function called to perform discrete curve evolution one vertex at a time. We invoke this function iteratively such that the least relevant half of our vertices are removed between contour outputs. The least relevant half of vertices are removed 8 times sequentially such that the final evolved contour will have over 99% of its vertices removed. Our function will calculate the relevance of each vertex, and then determine the location of the first relevance minima. This first minima corresponds to a vertex we delete from the contour. After a single vertex is removed, the contour is updated. When we call the function again, the updated contour has its relevance computed and the least relevant point of that contour will be removed. This is done as relevance values will change as vertices are removed.

```
def discrete_curve_evolution(x,y):
    dce_theta= np.arctan((y-np.roll(y,1))/(x-np.roll(x,1)))-np.arctan((np.roll(y,-1)-y)/(np.roll(x,-1)-x))
    dce_relevance_num=np.abs(dce_theta)*np.sqrt((x-np.roll(x,1))**2+(y-np.roll(y,1))**2)*np.sqrt((np.roll(x,-1)-x)**2+(np.roll(y,-1)-y)**2)
    dce_relevance_denom=np.sqrt((x-np.roll(x,1))**2+(y-np.roll(y,1))**2)+np.sqrt((np.roll(x,-1)-x)**2+(np.roll(y,-1)-y)**2)
    dce_relevance=dce_relevance_num/dce_relevance_denom

    dce_relevance_array=np.column_stack((dce_relevance,x,y))
    remove_index=dce_relevance_array[:,0].argmin()
    dce_relevance_array=np.delete(dce_relevance_array, remove_index, axis=0)

    x=dce_relevance_array[:,1]
    y=dce_relevance_array[:,2]
    dce_contour=dce_relevance_array[:,1:]
    return(x,y, dce_contour)
```

Main Script

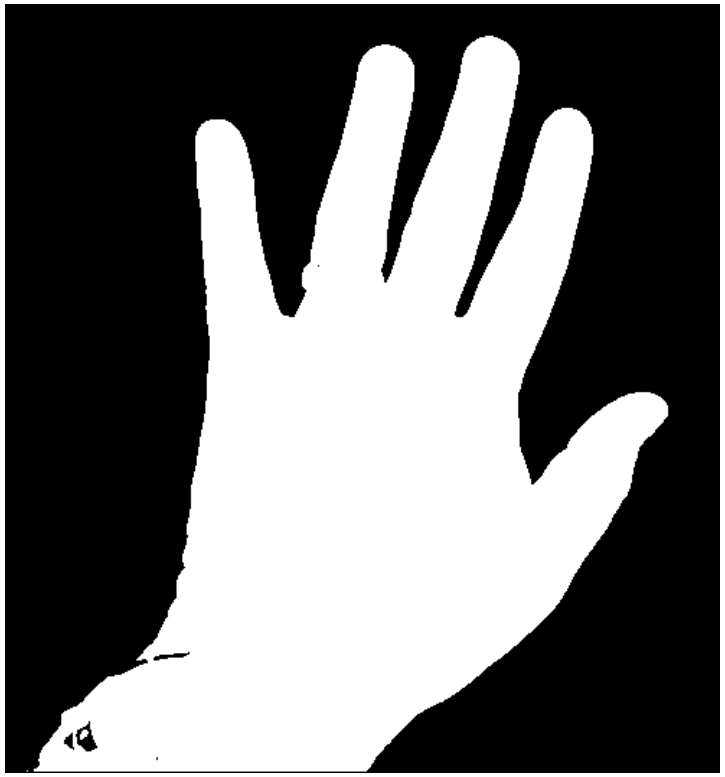
Here we invoke the previously discussed functions. We begin by iterating through the provided files. An image file is loaded into memory, and then converted into grayscale. The image is subjected to Gaussian blur. We binarize the image using Otsu and inverse thresholding. Then we add 1 pixel of black padding around the image. We write the binarization result to a .png file.

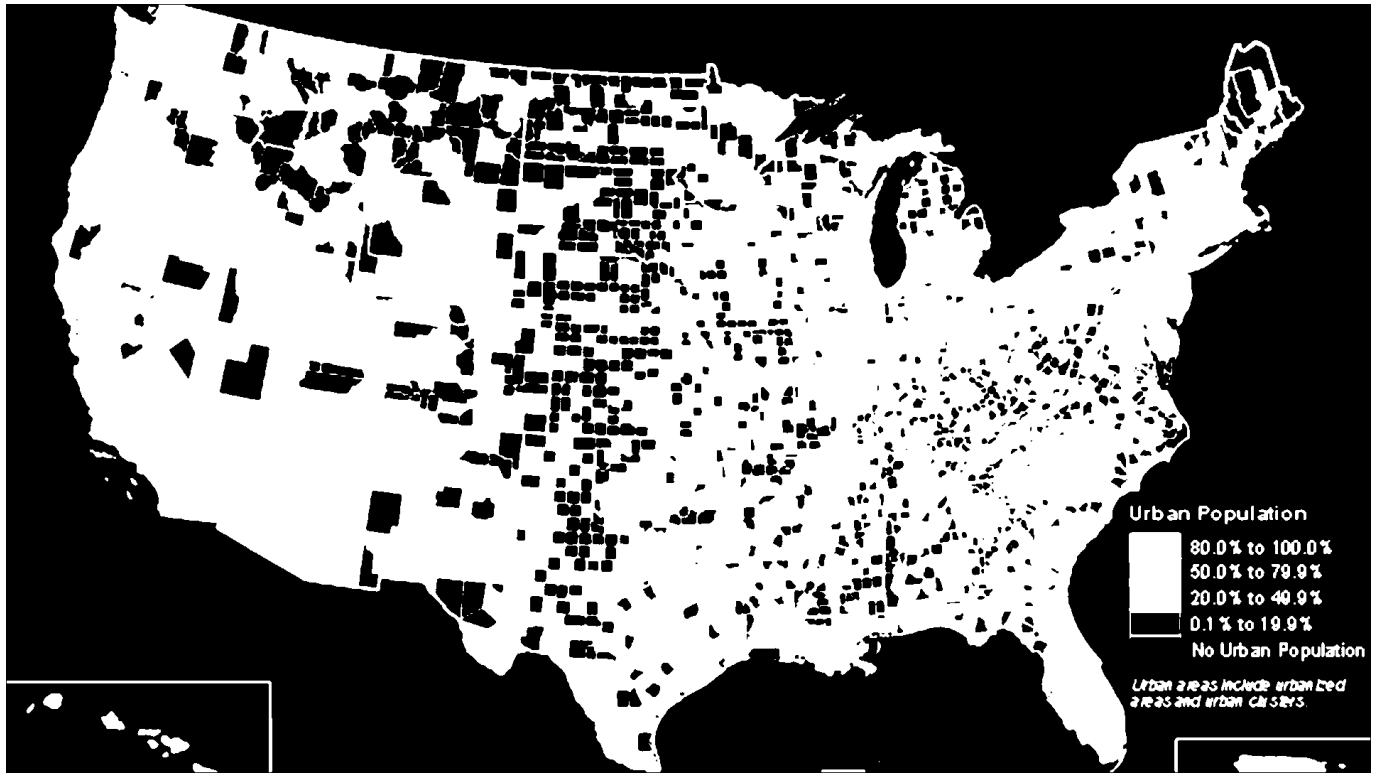
We call our Pavlidis function which returns a NumPy array of x and y values representing the contour. We print the vertices, Gauss area, and flood fill area of this original contour. We iterate through the discrete curve evolution function such that the least relevant half of our contour vertexes are removed 8 times sequentially. Between each sequence an evolved contour is output. As each of the evolved contours are created, we print the number of vertices, and we compute and print the Gauss and flood fill areas.

```
files=['hand0','US']
for file in files:

    o_img=cv2.imread(r'C:\Users\Matt\OneDrive\Virginia Tech\CV\Contour\\'+file+'.png')
    img=np.mean(o_img.copy(),2)
    img=cv2.GaussianBlur(img,(3,3),0)
    img=img.astype(np.uint8)
    threshold,img=cv2.threshold(img,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
    img=cv2.copyMakeBorder(img,1,1,1,1,cv2.BORDER_CONSTANT,None,value=(0,0,0))
    img_out=cv2.imwrite(r'C:\Users\Matt\Desktop\Results\Binary_'+file+'.png',img)
    bimg=np.zeros(img.shape[:2])

    contour=pavlidis(img)

    xc=np.float64(contour[:,1])
    yc=np.float64(contour[:,0])

    print('Filename: '+file+'.png') #File Name
    print(contour.shape[0]) #Vertices
    print(str(gaussarea(xc,yc))) #Gauss Area
    print(str(fillarea(np.int32(np.concatenate((xc.reshape(-1,1),yc.reshape(-1,1)),axis=1))))) #Flood Area

    for i in range(1,9):

        x=xc.copy()
        y=yc.copy()

        for j in range(int(contour.shape[0]*(1-1/2**i))):

            x,y,dce_points=discrete_curve_evolution(x,y)

        print(str(dce_points.shape[0])) #Vertices
        print(str(gaussarea(x,y))) #Gauss Area
        print(str(fillarea(np.int32(np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1))))) #Flood Area

        dce_points=dce_points.astype(np.int32)
        blank_img_2=np.zeros(img.shape).astype(np.int32)
        dce_img=cv2.polylines(blank_img_2,[dce_points], True, (255,255,255),1)
        cv2.imwrite(r'C:\Users\Matt\Desktop\Results\\'+file+'DCE_'+str(i)+'.png',dce_img)
```

Image Binarization

Here we see the output of thresholding. We load the images, convert the images to grayscale, and apply Otsu/inverse thresholding to the images. These steps are accomplished in our main script.





Console Output

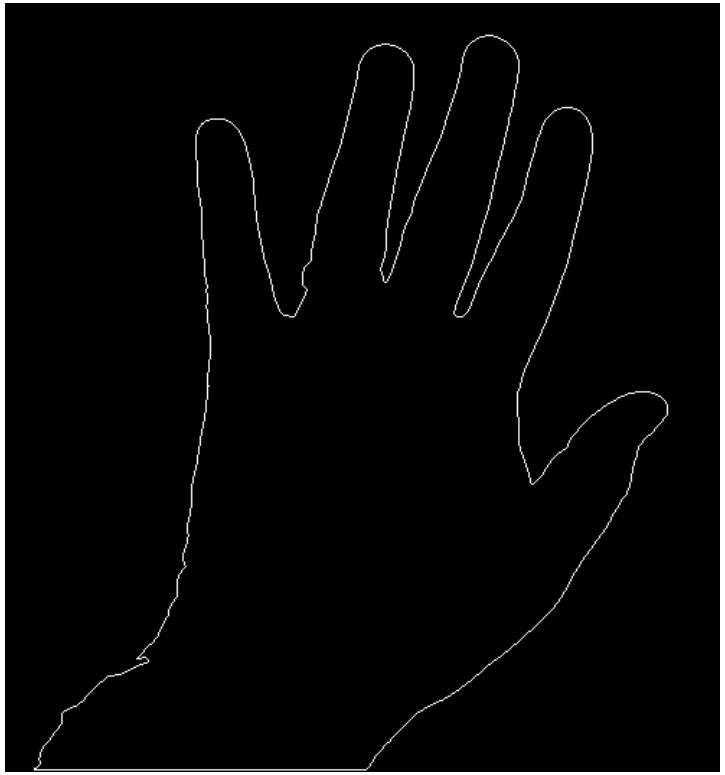
hand0.png	Vertices	Gauss Area	Flood Area
Iteration 0	2430	110930.5	109717
Iteration 1	1215	110930.5	109717
Iteration 2	608	110912.5	109713
Iteration 3	304	111082.5	109891
Iteration 4	152	110897	109706
Iteration 5	76	110291.5	109092
Iteration 6	38	111242	110053
Iteration 7	19	106341.5	105353
Iteration 8	10	100120	99318

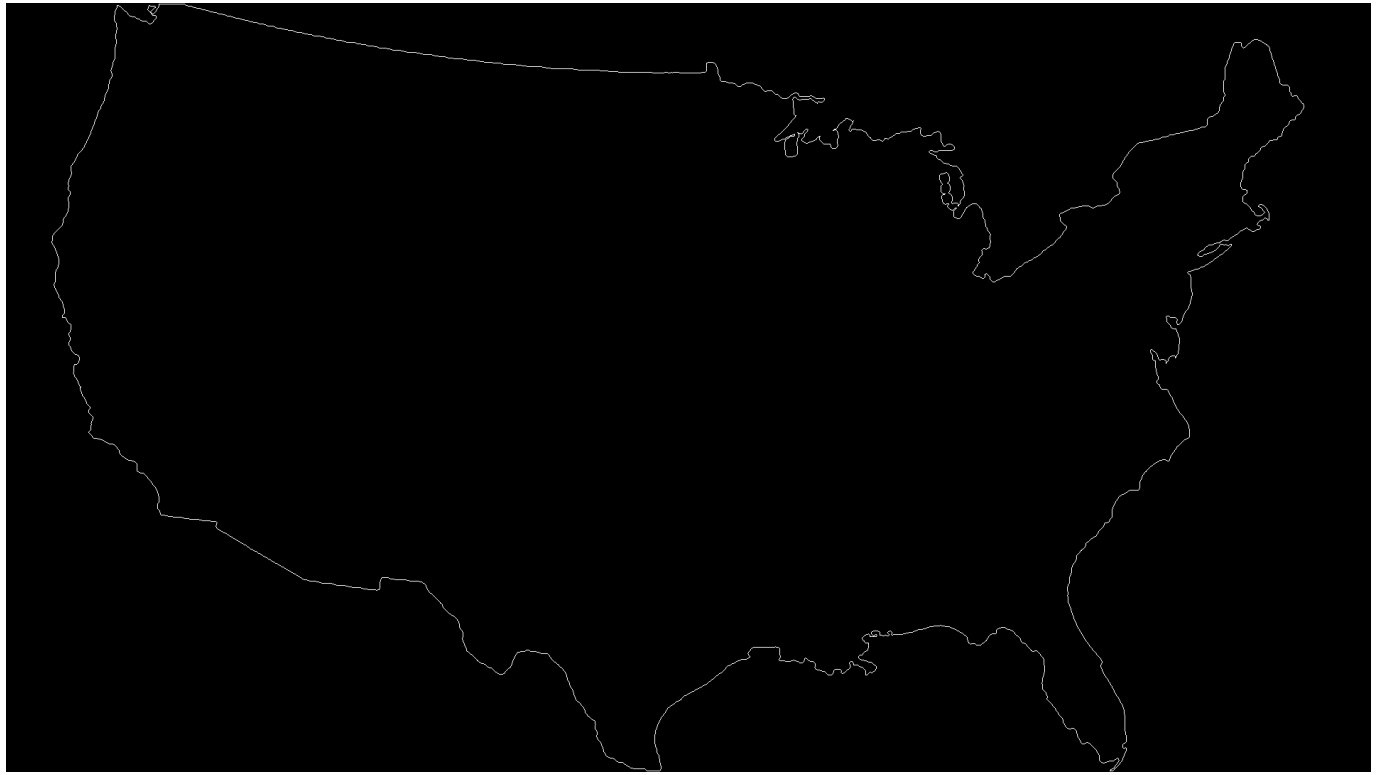
US.png	Vertices	Gauss Area	Flood Area
Iteration 0	6750	1157388	533898
Iteration 1	3375	1157388	533898
Iteration 2	1688	1157430.5	533885
Iteration 3	844	1157767	533721
Iteration 4	422	1157976	533885
Iteration 5	211	1158372.5	530549
Iteration 6	106	1164106.5	527043
Iteration 7	53	1163811	522899
Iteration 8	27	1154184	528518

Here we see the properties of our original contour (iteration 0) and our evolved contours (iteration 1-8). Notably area did not vary significantly between iterations and was exactly equivalent between iteration 0 and iteration 1. It seems suspicious at first that the area is exactly equivalent between iteration 0 and iteration 1. However, we confirmed the contours are being passed accurately to our area functions by debugging. Debugging allowed us to inspect the variables loaded in memory that are passed to our area functions and confirm they have the right number of contours. There is an issue with the provided flood fill area function yielding areas across that are about half the size of the Gauss area. This issue is specific to the US.png file for an unknown reason. We present each original and evolved contour in the following slides.

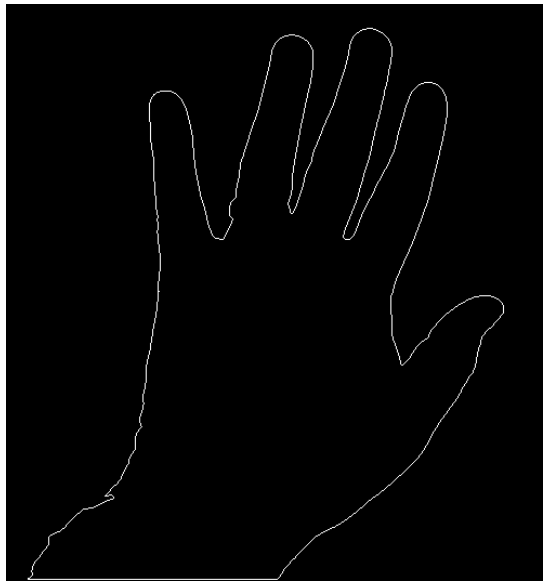
Pavlidis Contours

Here we see the output of our Pavlidis contour function (iteration 0 on the console output table). This contour represents the first input to our discrete curve evolution. We perform discrete curve evolution on this contour until half of vertices have been removed. We repeat until half of vertices have been removed 8 times sequentially. Each time the vertices are cut in half, we output a polygon and its properties. In the following slides we display the resulting evolved contours for hand0.png and US.png.

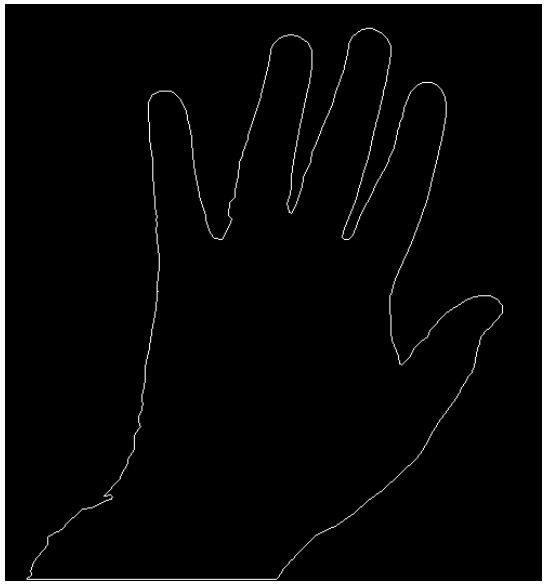




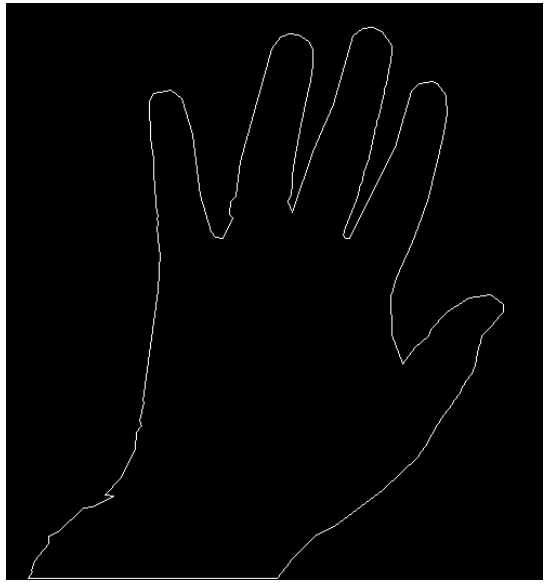
Iteration 1



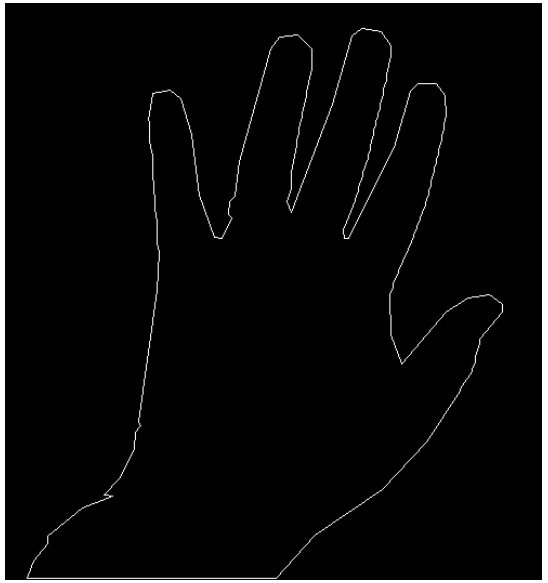
Iteration 2



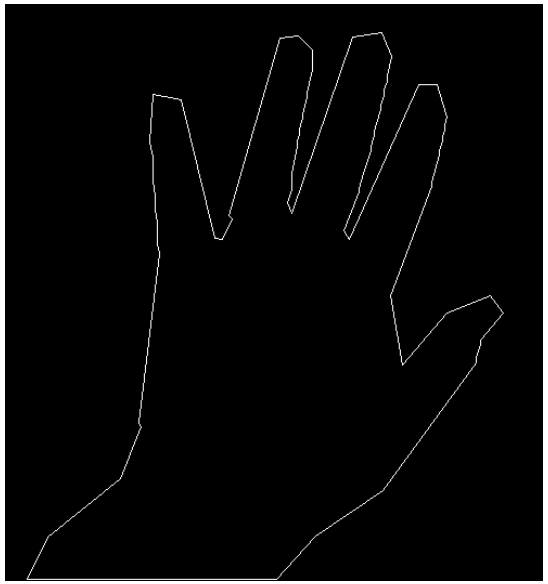
Iteration 3



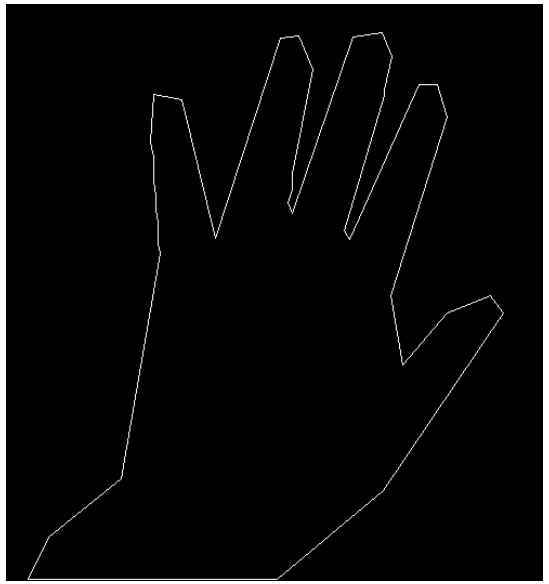
Iteration 4



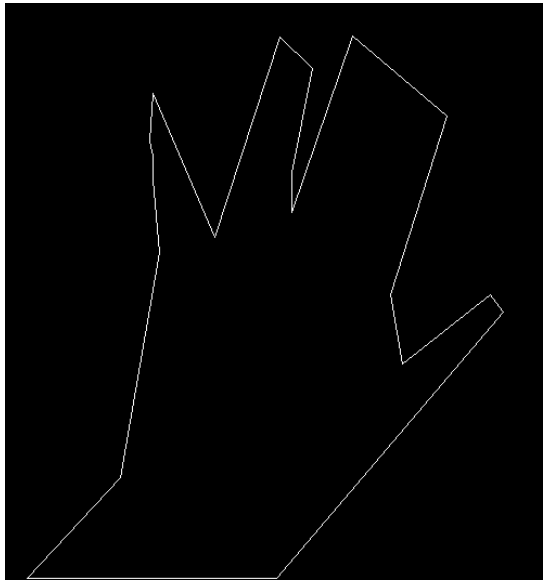
Iteration 5



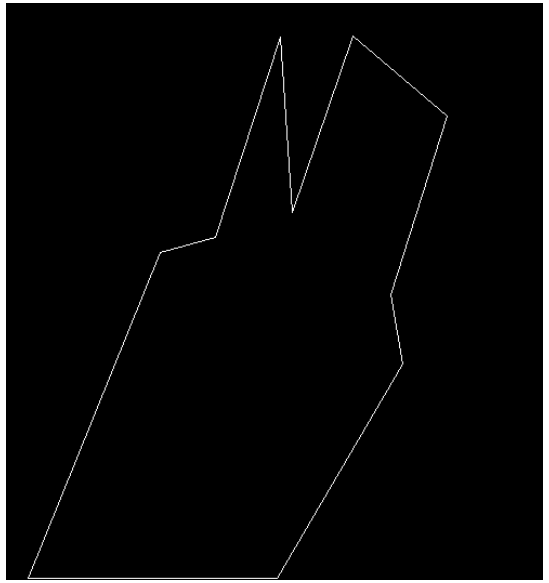
Iteration 6



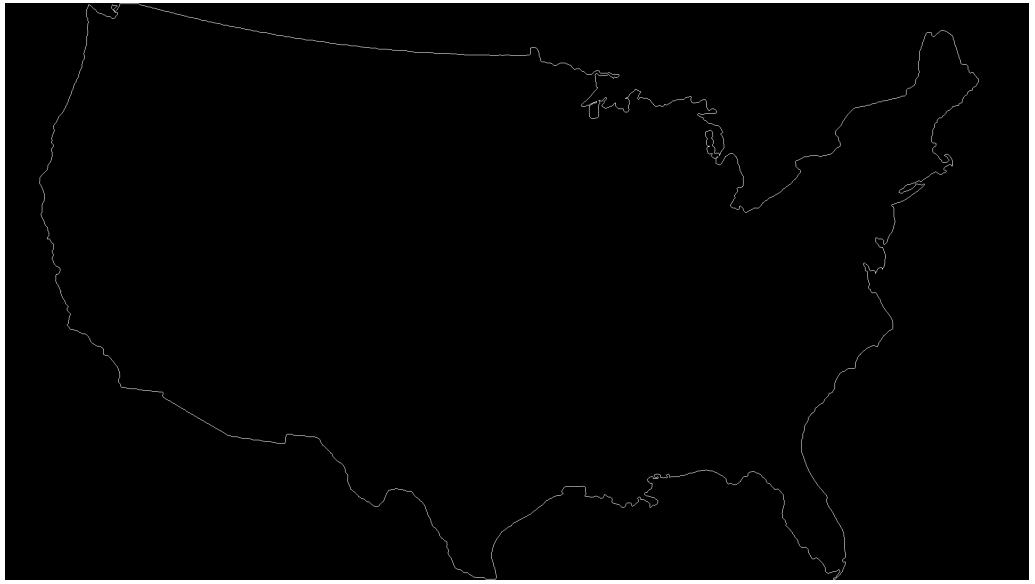
Iteration 7



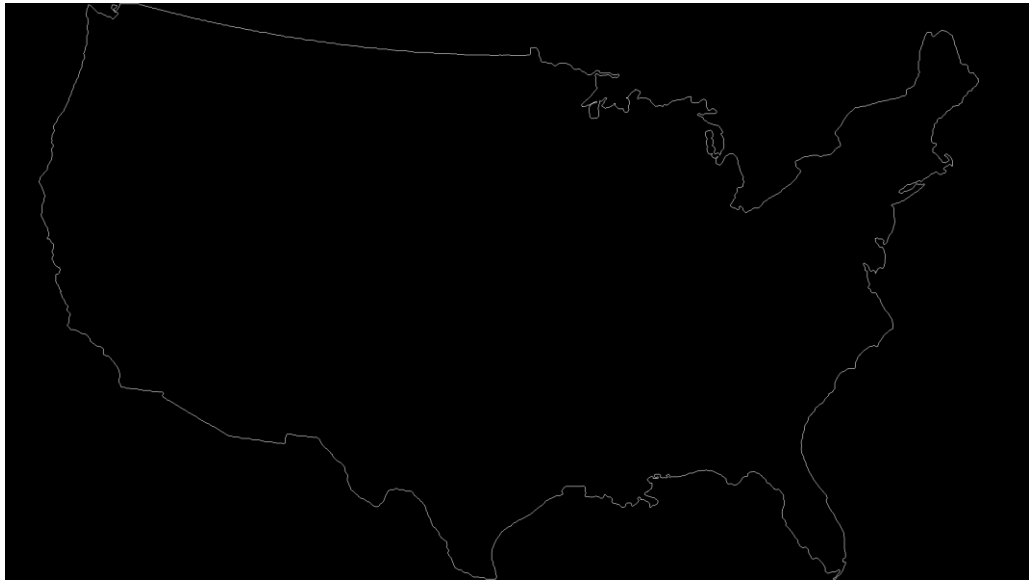
Iteration 8



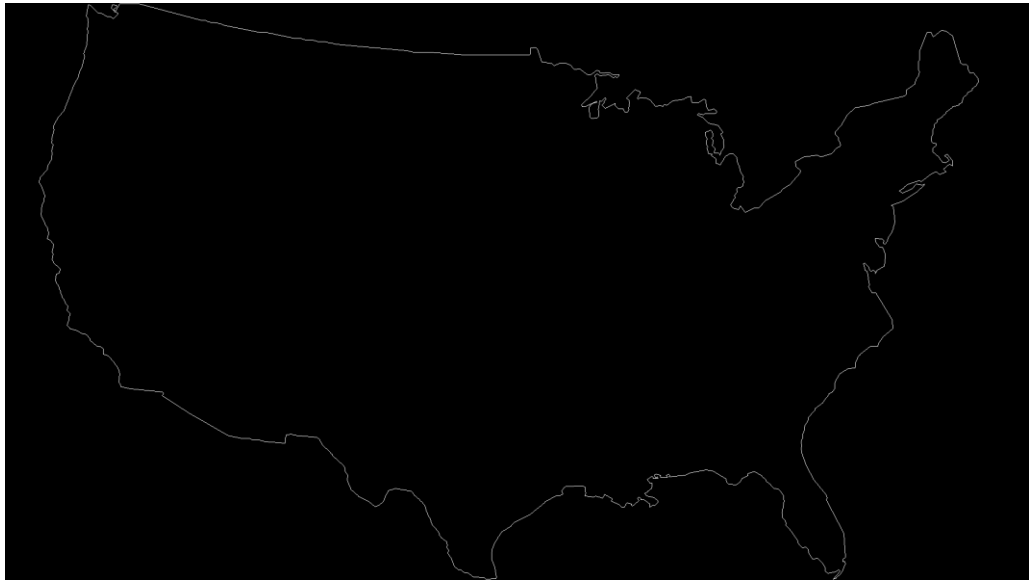
Iteration 1



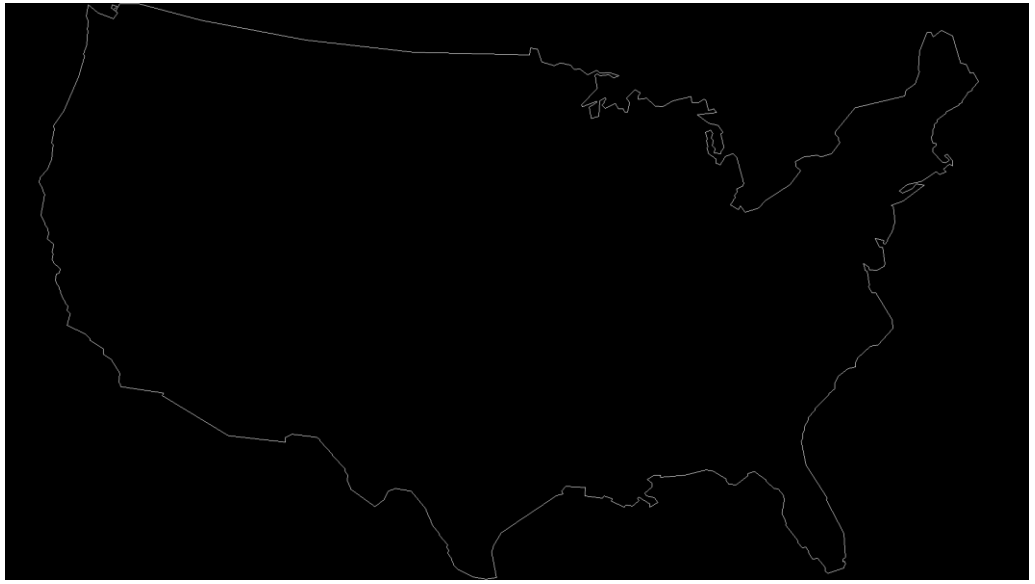
Iteration 2



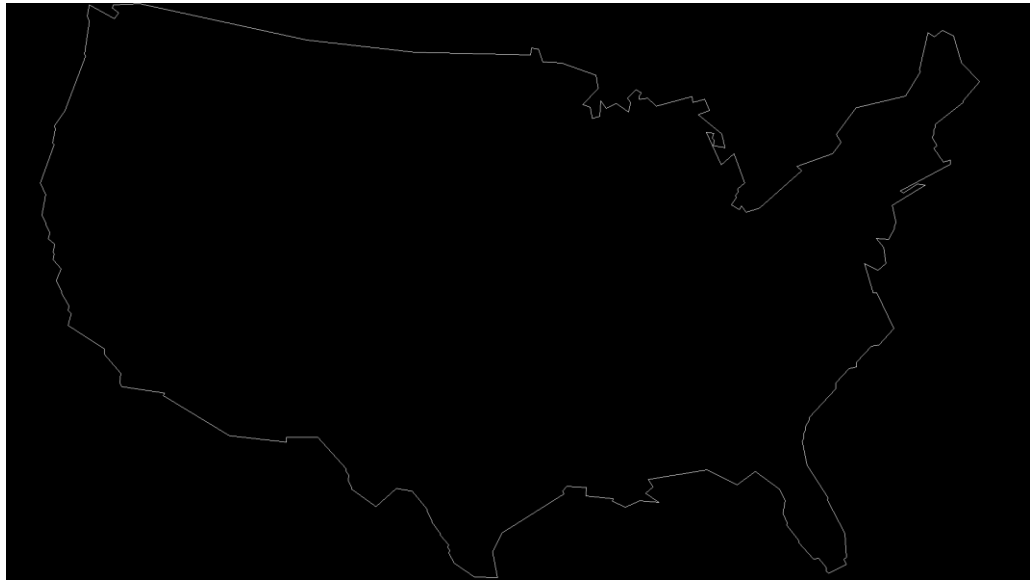
Iteration 3



Iteration 4



Iteration 5



Iteration 6

Iteration 7

Iteration 8