Measuring Area from Images

Applied Physics 186 - Activity 4

Marc Jerrone Castro

Importing packages

```
[4]  1 import cv2
     2 from google.colab.patches import cv2_imshow
     3 import numpy as np
     4 import math
     5 !pip install latex
     6 import latex
     7 import matplotlib.image as mpimg
     8 import matplotlib.pyplot as plt
     9 import pandas as pd
```

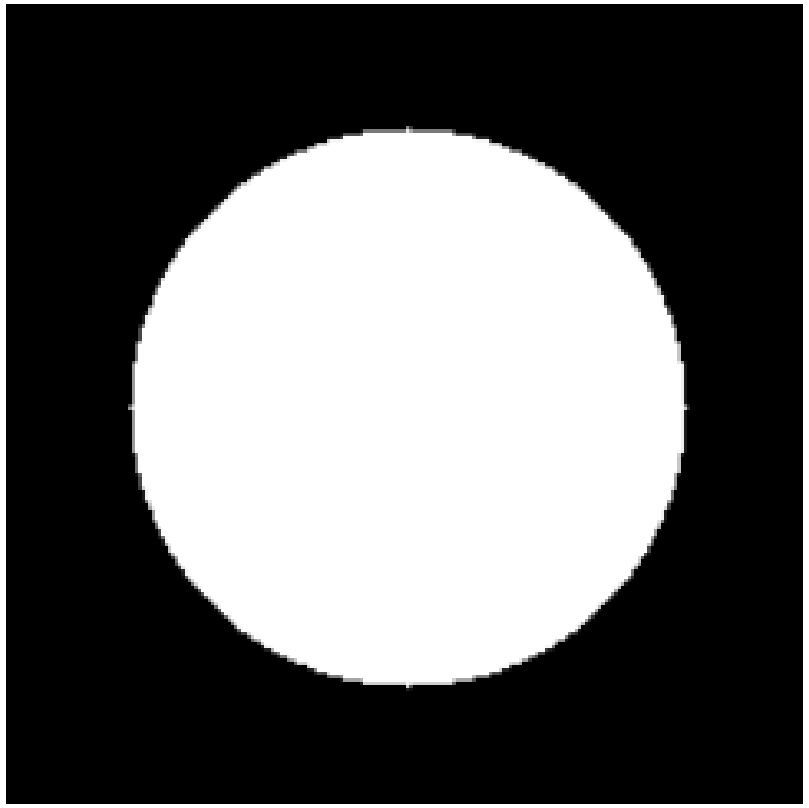
```
[5]  1 from google.colab import drive
     2 drive.mount('/content/drive')
```

I opted to use **OpenCV** as it is one of the leading python packages that handles **image processing related applications and algorithms**. As I was using **Google Colaboratory**, I had to import **cv2_imshow** separately as it is currently unsupported.

Additional packages such as **math**, **numpy**, and **pandas** were imported for calculating the areas of the images. **Matplotlib** was imported for visualization purposes.

Creating an image of the circle

```
[ ]  1 image = cv2.imread("/content/drive/My Drive/186/Activity4/201x201.png")
     2 cv2.circle(image,(101, 101), 70,(255, 255, 255),thickness = -1)
     3 cv2_imshow(image)
```



Initially, I loaded a **201px x 201px** image of a black background into the IPython notebook. Afterwards, I used the **OpenCV** package, **cv2.image**, to draw a circle of radius=70 onto the same background

Figure 1.0 An image of a circle of radius = 70

Getting the Edge Pixels of the Circle

```
[ ]  1 edges = cv2.Canny(image,255,255)
     2 cv2_imshow(edges)
```

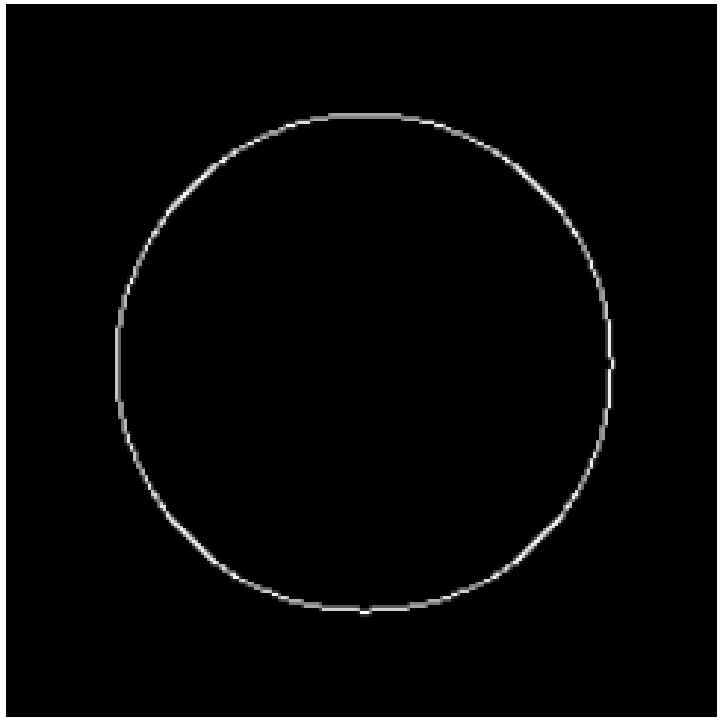


Figure 2.0 The edges of a circle of radius = 70

```
[ ]  1 indices = np.where(edges != [0])
     2 coordinates = zip(indices[0], indices[1])
     3 coords = list(coordinates)
```

Afterwards, I used another **OpenCV** package called, **cv2.Canny**, to extract the edges of the circle from **Figure 1**. It runs from an input image and a set minimum and maximum threshold – set as 255 to isolate only the white edges. The resulting image of the edge of the input image can be seen from **Figure 2**.

The following code block then determines all the **coordinates** of the edges by finding all pixel values **not equal to zero.** I then appended this into a list for further processing later on.

Determining the centroid of the circle and sorting the list with respect to θ.

```
[10]  1 x = [p[0] for p in coords]
      2 y = [p[1] for p in coords]
      3 centroid = (sum(x) / len(coords), sum(y) / len(coords))
      4 print(centroid)
```

```
(100.93103448275862, 100.93103448275862)
```

```
[11]  1 coords_polar = list()
      2 for k in coords:
      3   x =((k[0] - centroid[0]))
      4   y =((k[1] - centroid[1]))
      5   r = np.sqrt(x**2 + y**2)
      6   theta = (np.arctan2(y,x))
      7   coords_polar.append(tuple((tuple((k[0],k[1])),theta)))
```

```
[12]  1 sort_polar = sorted(coords_polar , key=lambda k: [k[1],k[0]])
      2 #Converting list into an array
      3 sorted_xy = []
      4 for polars in sort_polar:
      5   sorted_xy.append(polars[0])
```

We then determine **the x and y coordinates of the edges** and then determine the coordinates of the **centroid** of that circle by dividing the sum of all the x-axis coordinates by the total number of coordinates. The same method was done for the y-axis coordinates.

As one may observe the centroid was **not exactly located at (101,101)** but rather at **100.9** which deviates slightly from center. This may have been caused by the uneven parsing of the coordinates of the edges such that it is limited to the resolution of the segmentation

Determining the centroid of the circle and sorting the list with respect to θ.

```
[10]  1 x = [p[0] for p in coords]
      2 y = [p[1] for p in coords]
      3 centroid = (sum(x) / len(coords), sum(y) / len(coords))
      4 print(centroid)
```

```
(100.93103448275862, 100.93103448275862)
```

```
[11]  1 coords_polar = list()
      2 for k in coords:
      3   x =((k[0] - centroid[0]))
      4   y =((k[1] - centroid[1]))
      5   r = np.sqrt(x**2 + y**2)
      6   theta = (np.arctan2(y,x))
      7   coords_polar.append(tuple((tuple((k[0],k[1])),theta)))
```

```
[12]  1 sort_polar = sorted(coords_polar , key=lambda k: [k[1],k[0]])
      2 #Converting list into an array
      3 sorted_xy = []
      4 for polars in sort_polar:
      5   sorted_xy.append(polars[0])
```

Afterwards, we then calculated the corresponding angles at each of the equivalent coordinates with respect to the centroid. We then **sort** the **original x and y coordinates with respect to the calculated angles.**

As the appending the coordinates to the list may have altered order, we sorted it again before extracting the x-y coordinates

Calculating the Area of the Circle

```
[ ]  1 area = 0
     2 for t in range(len(sorted_xy)-1):
     3   A = ((sorted_xy[t][0]*sorted_xy[t+1][1]) - (sorted_xy[t][1]*sorted_xy[t+1][0]))
     4   area += A
```

Comparison of Areas (Calculated vs Expected)

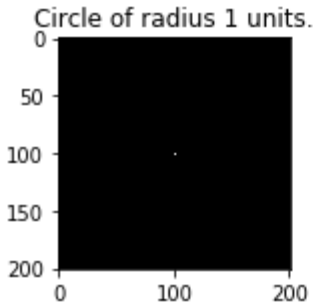
```
[ ]  1 print('The expected area of the circle is ',3.14*4900,'units^2.')
     2 print('The calculated area of the circle via Green\'s Theorem is ',area/2,'units^2.')
     3 print('Error in calculation between expected and calculated is', round((((3.14*4900)-(area/2))/(3.14*4900))*100,3),'%.')
```

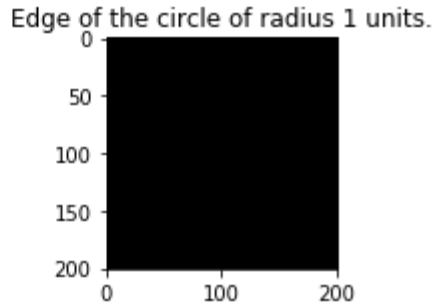
```
The expected area of the circle is  15386.0 units^2.
The calculated area of the circle via Green's Theorem is  15387.5 units^2.
Error in calculation between expected and calculated is -0.01 %.
```

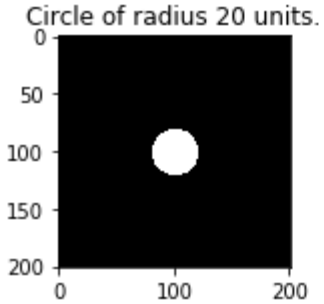
The **area of the circle** was then calculated using **Green's Theorem**, whereas, comparison of the **expected** and **calculated** areas reveals that the method was **accurate** as the **percent error** was only a small difference of **0.01%**.

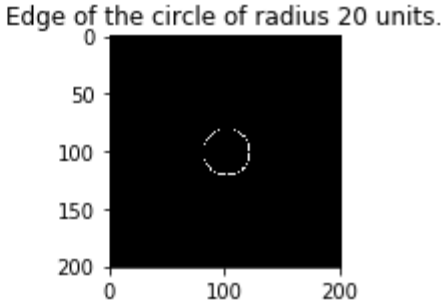
Calculating the Area for various radii

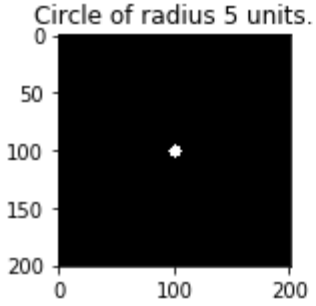
```
[ ]  1 trial_radius = [1,5,10,20,25,40,70,80,90,100]
     2 for z in trial_radius:
     3   a = area_calculation(int(z),True)
```

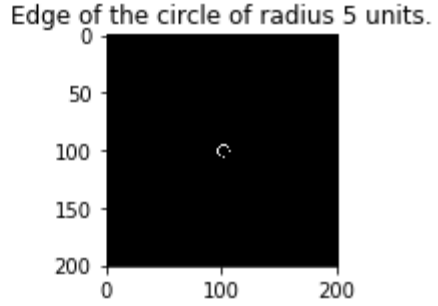


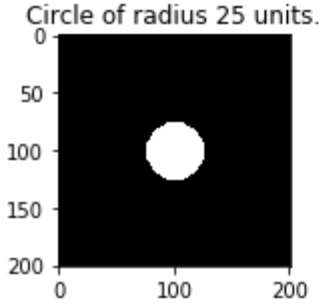


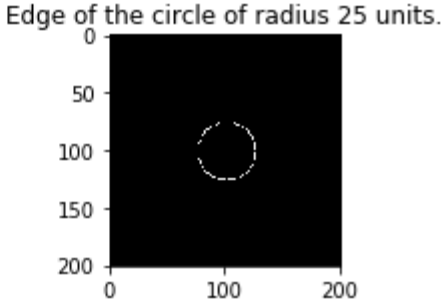


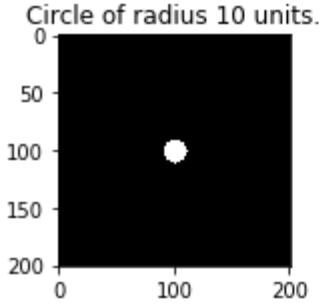


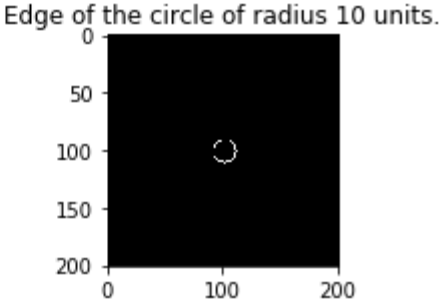


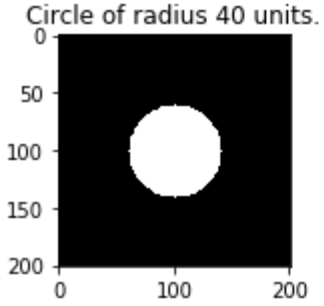


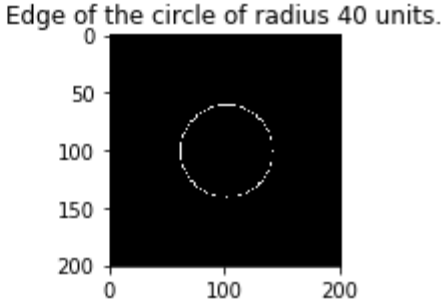












Afterwards, we then streamline the earlier process by defining it as the function **area calculation**, whereas , it uses an **input radius z** to generate subsequent **circles of the corresponding radius**. A sample of these images may be seen on the right with their corresponding edges on their right.

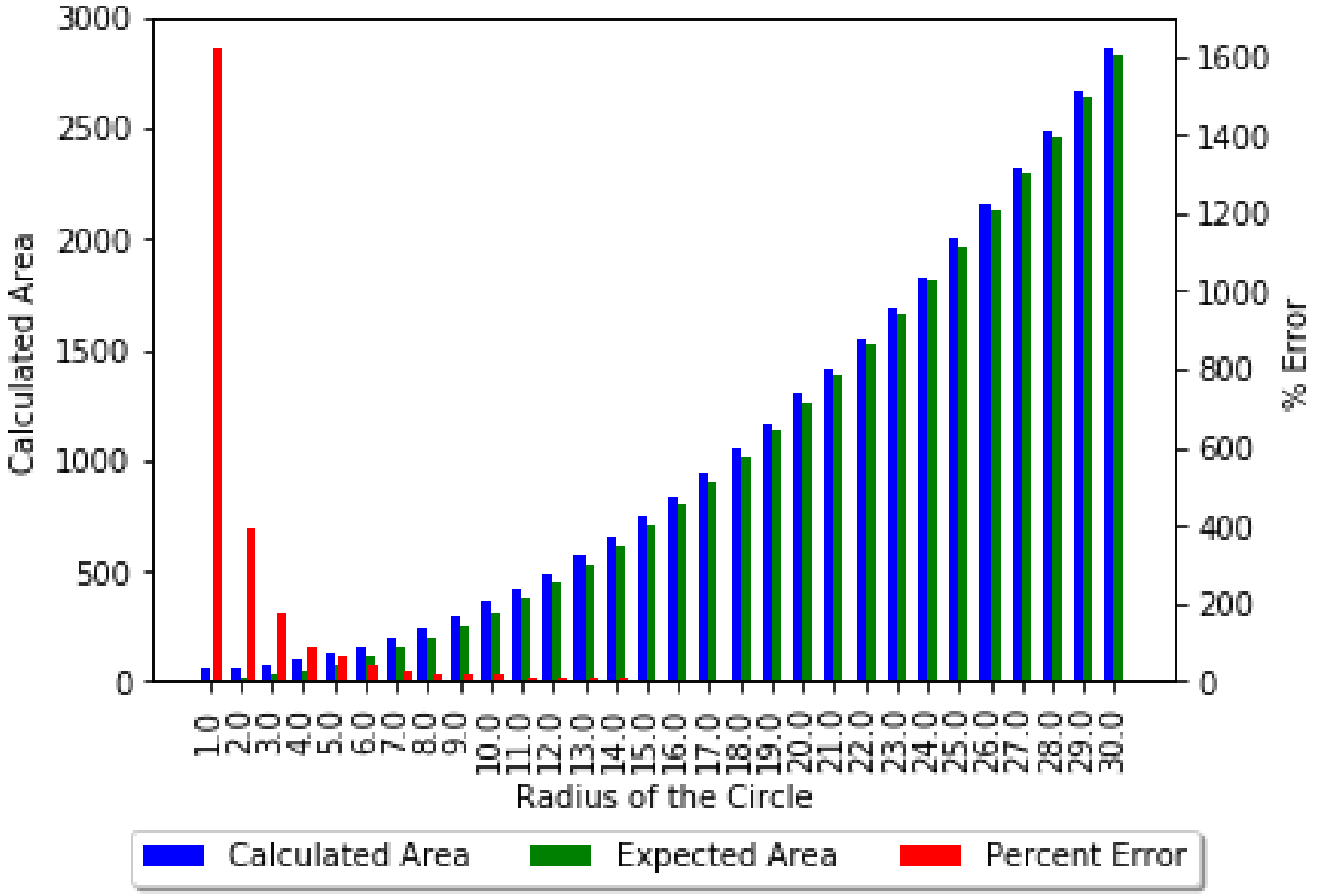


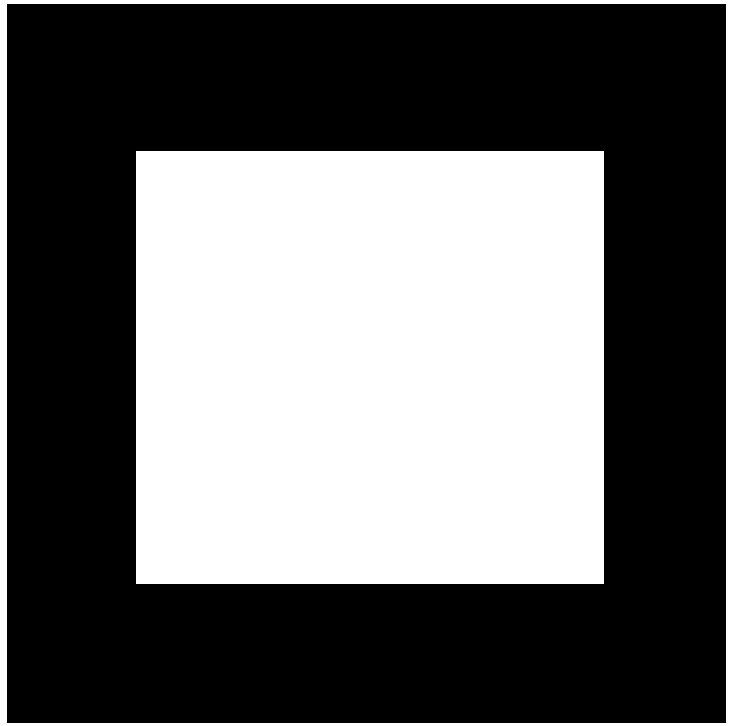
Figure 3.0 Comparison of calculated and expected areas of circles of various radii. Percent errors for each calculation is then tabulated along with their corresponding area calculations

From **Figure 3**, we can observe that there is a **large deviation between the expected and calculated areas**. A large percentage error **exponentially decreases as we increase the radius** of the circle. For the circles with smaller radii, the generated circles have low resolutions and as pixels are generally composed of square bits rather than circular, the circles generally form a curve using a stair-like transition for every movement in the x-y axis.

As such, the circles would have pixelated edges and would be greatly inaccurate for smaller circles . This then transitions to the failure of the method for lower resolutions/radii as a large discrepancy exists between the calculated and expected area.

Rectangle

```
[ ]  1 image2 = cv2.imread("/content/drive/My Drive/186/Activity4/201x201.png")
     2 cv2.rectangle(image2, (36,41), (166,161), (255, 255, 255), thickness= -1)
     3 cv2_imshow(image2)
```



A similar method was then employed for calculating the **area of rectangles**. Using the same **201px x 201px black background**, we utilize the **OpenCV** package **cv2.rectangle** to generate a white rectangle with inputs of the coordinate of **the top right vertex** and the coordinate of **the bottom right vertex.**

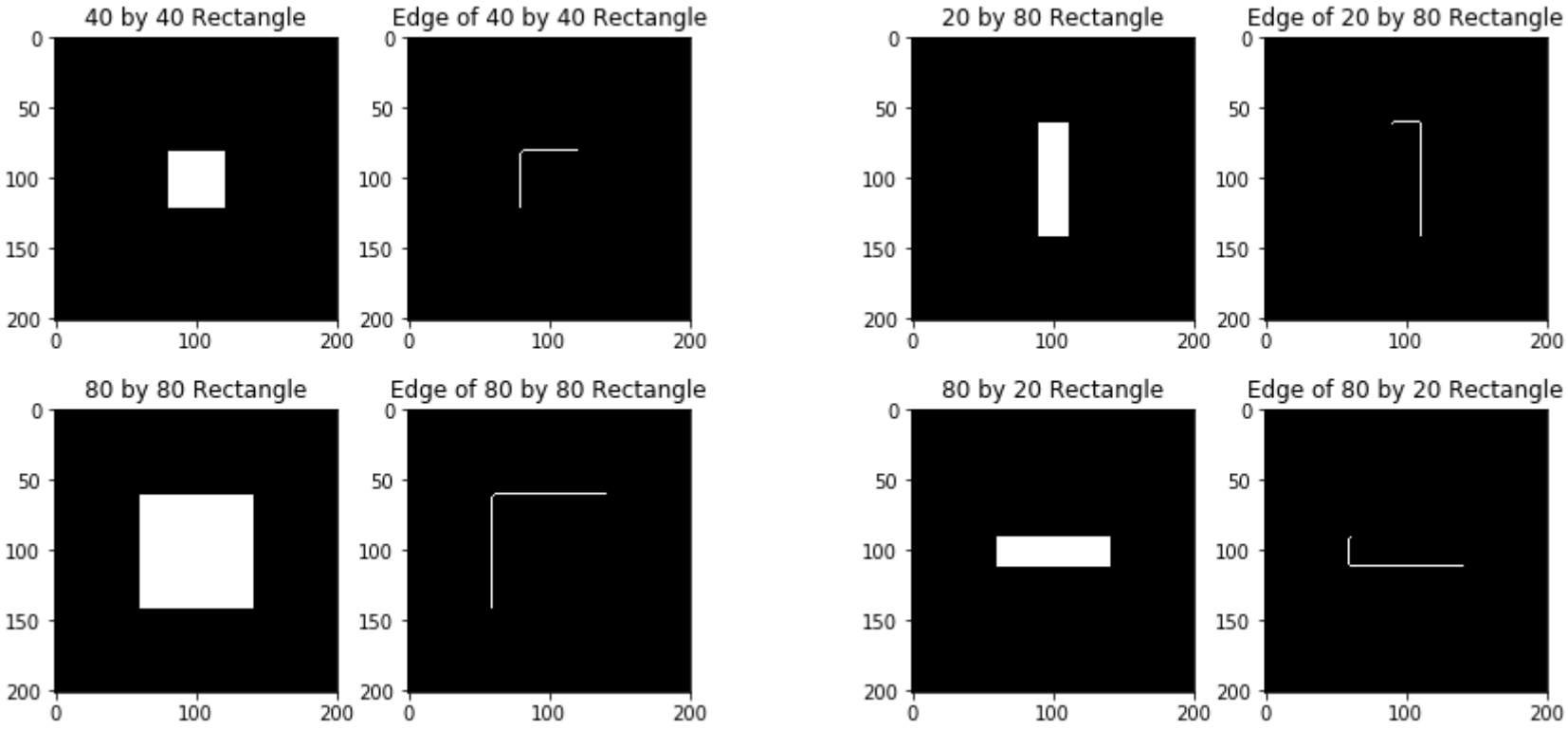
Figure 4.0 An image of a rectangle generated using OpenCV

Determining the centroid of a rectangle and sorting the list with respect to θ.

```
x = [p[0] for p in coords]
y = [p[1] for p in coords]
centroid = ((min(x)+max(x))/2 , (min(y)+max(y))/2)
coords_polar = list()
for k in coords:
  x =((k[0] - centroid[0]))
  y =((k[1] - centroid[1]))
  r = np.sqrt(x**2 + y**2)
  theta = (np.arctan2(y,x))
  coords_polar.append(tuple((tuple((k[0],k[1])),theta)))
sort_polar = sorted(coords_polar , key=lambda k: [k[1],k[0]])
```

A minor difference for calculating the area between circles and rectangles, whereas, the **centroid of the rectangle** was calculated using the **average value of maximum and minimum values** of each of the axii. We then sort the x-y coordinates once again with respect to the angle we had calculated.

Generating rectangles and their corresponding edges



Afterwards, we generate the rectangles with different dimensions and extract edges from each of the rectangles. Upon observation, the edges were not fully formed, however, this is most likely due to the low resolution of the image as coordinates of the edges showed locations not present within the edge figures above. Observing the edges at larger resolutions would most likely result to a complete edge images.

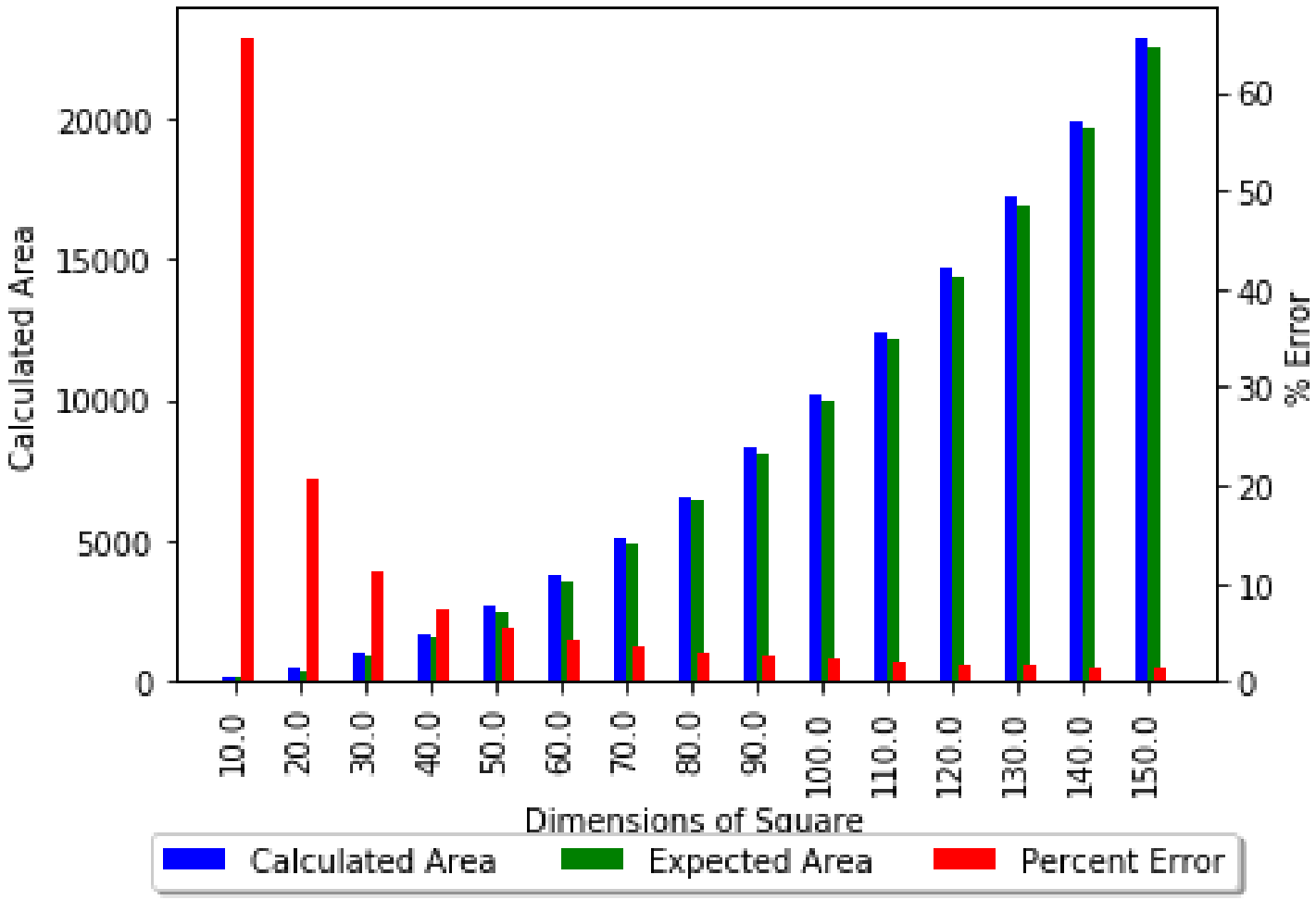


Figure 4.0 Comparison of calculated and expected areas of squares of various dimensions. Percent errors for each calculation is then tabulated along with their corresponding area calculations for better representation.

For the comparison between rectangular images, we had **limited** the dimensions such that the images were **all squares** as to **facilitate a better comparison** between each variation. We can observe that a **similar trend** to Figure 3.0 occurs ,whereas, **lower or smaller rectangles would result to a higher deviation** in contrast to the expected area. These similar deviations maybe attributed to additional pixels being included into the edges of the rectangle. Due to the rectangle being relatively small, the algorithm detects that majority, if not most, of the pixels are part of the edge – where in fact only a small portion of it is. This might have something to do with the **failure of Intensity gradients for smaller dimension shapes.**

Google Map Image of the Quezon City Circle Fountain and Area Calculation

Figure 5.0 Picture of one of the arcs around the Liwasang Aurora Fountain in Quezon Memorial Circle

As for the additional task in this activity, I applied the method for calculating the area of a 2D image for this particular area. I was curious as to how large this particular piece of land as it was quite large and would hold a number of tourists and locals during events such as Music fests and historical tours.

Google Map Image of the Quezon City Circle Fountain and Area Calculation

Figure 6.0 Eagle eye view of Picture of the Liwasang Aurora Fountain in Quezon Memorial Circle retrieved from Google Maps.

Google Map Image of the Quezon City Circle Fountain and Area Calculation

```
The theoretical area of the CS atrium is 5644.766343237435 m^2.
The area of the CS atrium calculated using Green's Theorem is 5544.0215301807 m^2.
The percent error of the calculation is -1.784747267305922 %.
```

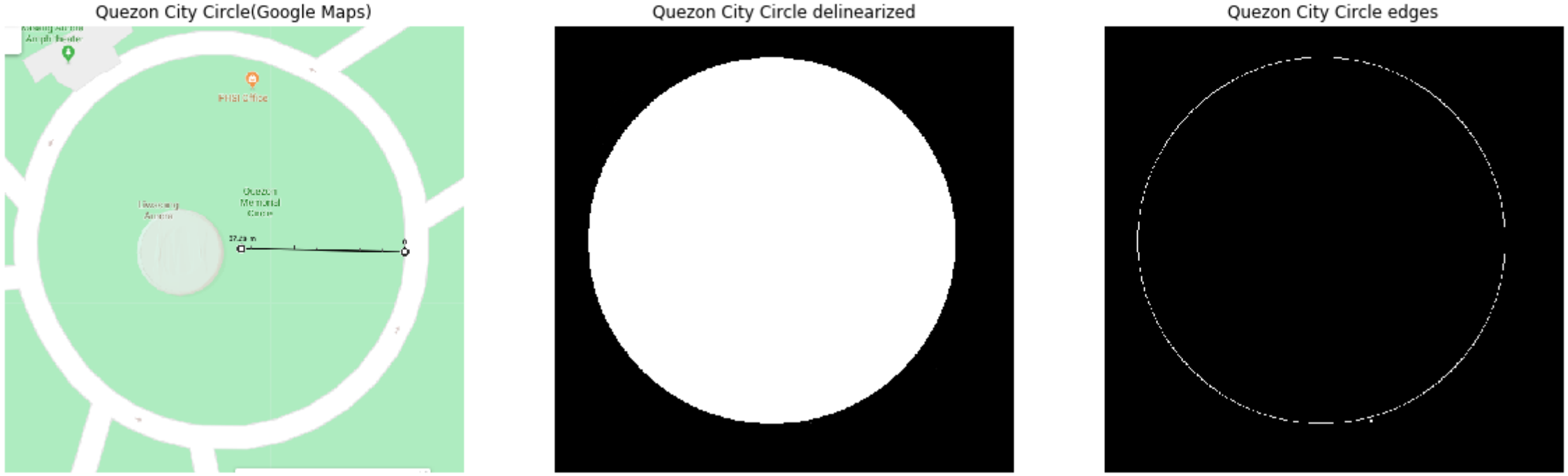


Figure 7.0 Image from the Liwasang Aurora Fountain retrieved from Google Maps and its corresponding delinearized image and edge image.

An image of the Liwasang Aurora Fountain area was first taken from Google Maps. Take note that before taking a snapshot of the area, I had previously used a built-in tool to measure a certain distance within the area. Afterwards, I uploaded the snapshot into ImageJ and scaled it correspondingly with respect the distance I measured earlier on. Afterwards I determined the pixel to meter scaling of the image using the same process – 5.1 px/m. Afterwards I determined the radius to be equal to 42.4m to use later on.

Google Map Image of the Quezon City Circle Fountain and Area Calculation

```
The theoretical area of the CS atrium is 5644.766343237435 m^2.
The area of the CS atrium calculated using Green's Theorem is 5544.0215301807 m^2.
The percent error of the calculation is -1.784747267305922 %.
```

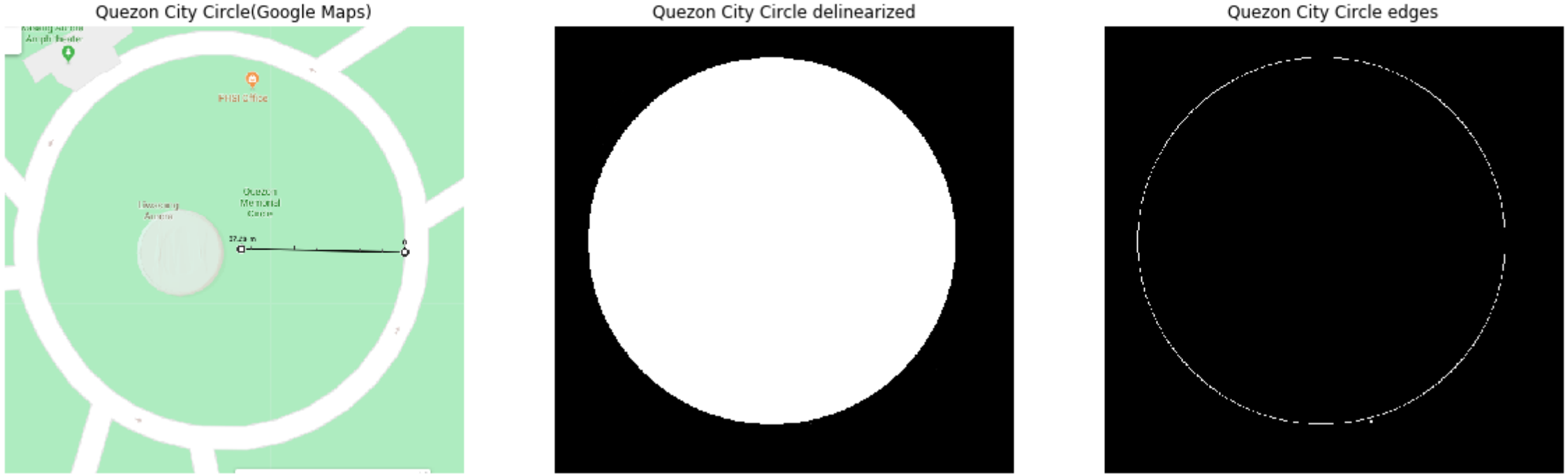


Figure 7.0 Image from the Liwasang Aurora Fountain retrieved from Google Maps and its corresponding delinearized image and edge image.

Afterwards, I limited the selection to only the area of and around the Liwasang Aurora fountain. I highlighted those within the area white, and those outside as black as to simulate binary image of the area of interest. Using the method for calculating the pixel coordinates of the edges and determining the area, I had observed that an accurate representation of the area was achieved as only a small deviation of 1.8% was observed. I was also shocked to see that the Fountain Area was more or less bigger than the CS atrium.

Self-Evaluation

Technical correctness - 5

Quality of presentation - 5

Initiative - 2