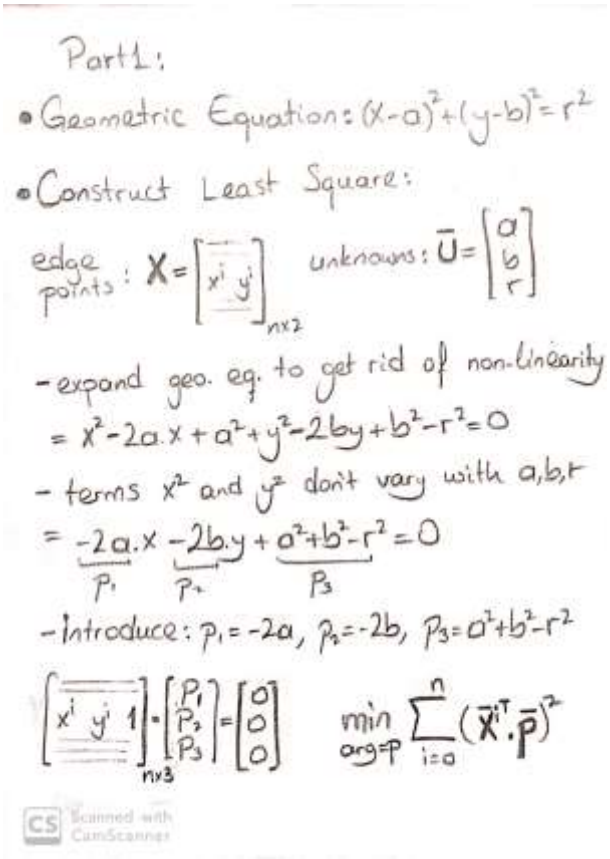
Tanay Cansin Dogan
150150119

Computer Vision: Assignment4

Part1:



Pseudo-code:

$(x, y)^i$: i^{th} edge point

```
For a:= a_min to a_max:
   For b:= b_min to b_max:
      For r:= r_min to r_max:
         For i:= 1 to n:
            If (x^i-a)^2 + (y^i-b)^2  ~= r^2 :
               Acc[a, b, r]++;
```

Part2:

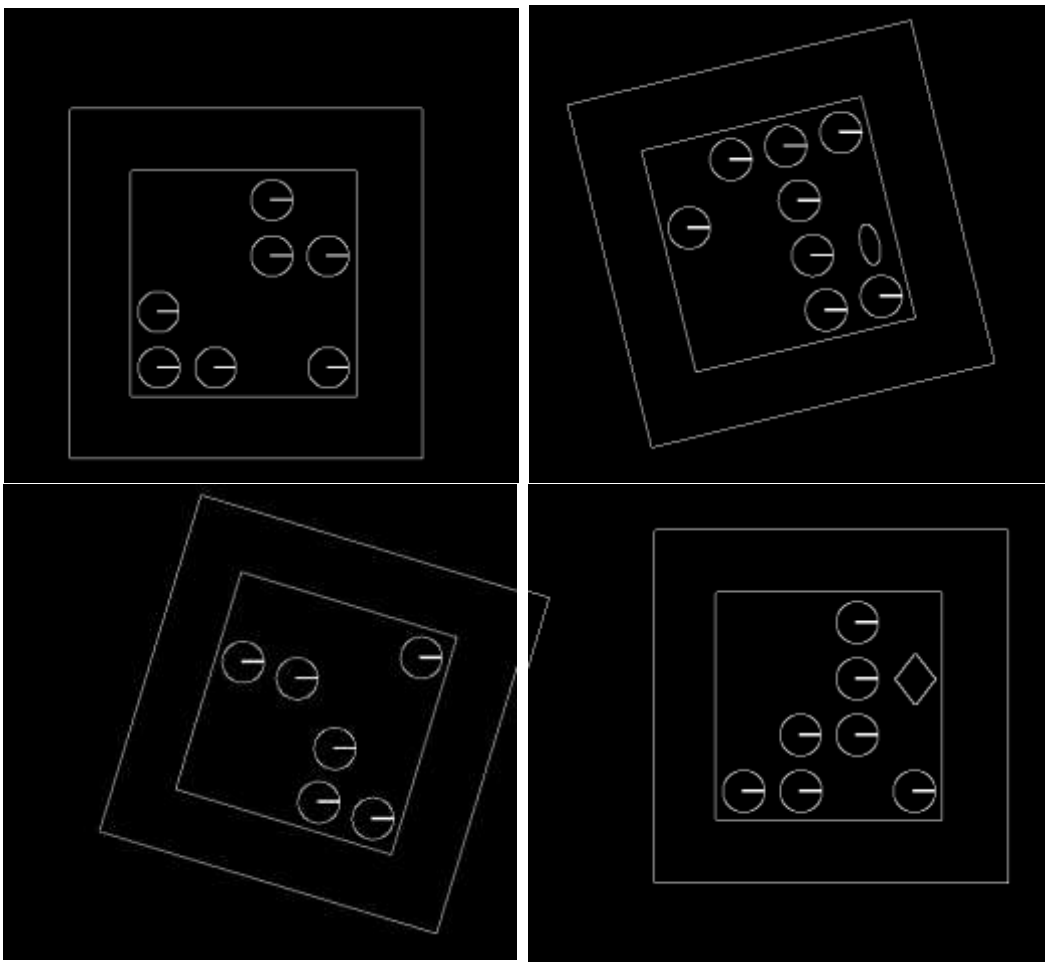
In part1, we write a pseudo-code for the Hough circle detection method, basically we iterate over image with parameters a, b and try all circles with varying the parameter r and count the number of edge points which fits in this equation. I wanted to make make my Hough function general for all images. Thus, instead of setting bounds for a, b, and r parameters, Hough function determines itself, and try every (if quantization=1) possible circle within those bounds.

In order to speed-up the algorithm I determined an upper and lower bound for parameters a, b according to the parameter r. Thus, algorithm's logic becomes "try to construct a circle with radius=r^i on center (a', b')".

Hough_circle( ) function sets upper limit of r according to the min(num_row, num_colm)/2, since any circle with greater r value, cannot be complete circle. For that reason to speed-up the algorithm, I also divided the marker image 4 equal pieces and examine each of those pieces separately.

Quantization	Run time
3	~3 min.
2	~9.5 min.
1	~80 min.

The function at quantization=2 or quantization=3 works faster but fails to detect all circles, especially the ones in rotated marker. Due to the runtime, I wouldn't recommend quant=1. The result of the run with quantization=1 as follow:



Part3:

In this part, I implemented a general area_growing( ) function and volume_growing( ) function. Those functions call required get_x_neighbors( ) function, then check those neighbors and decide to label 1 or not, according to the given threshold value. Those functions could have realized as recursively, but I did prefer it to realize it iteratively since it can be vectorised in this way, and also "stack overflow" is not a concern for larger NIfTI data (especially with 26-neighborhood).

- **a) 8-neighborhood:**

  In this part get_8_neighbors( ) function called inside the area_growing( ) function. This function takes 2D points and returns its 8 neighbors.

- **b) 4-neighborhood:**

  In this part get_4_neighbors( ) function called inside the area_growing( ) function. This function takes 2D points and returns its 4 neighbors.

- **c) 26-neighborhood:**

  In this part get_26_neighbors( ) function called inside the volume_growing( ) function. This function takes 3D points and returns its 26 neighbors.

- **d) 6-neighborhood:**

  In this part get_6_neighbors( ) function called inside the volume_growing( ) function. This function takes 3D points and returns its 6 neighbors.

- **e) Dice score results & Suggestion:**

  The performance of 8-neighbors: 0.9037632259828772

  The performance of 4-neighbors: 0.8060919096350647

  The performance of 26-neighbors: 0.8901082972687553

  The performance of 6-neighbors: 0.8909479095412806

  Since image/data corrupted by Poisson noise (a random type noise) applying arithmetic mean or geometric mean filters before segmentation operation, may improve result of the segmentation.