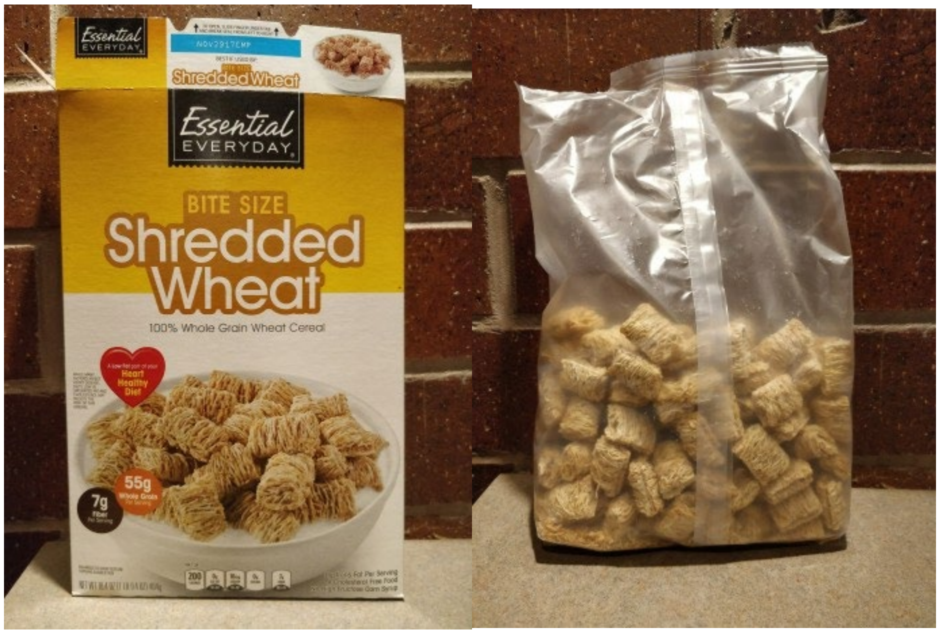
## Part 1:

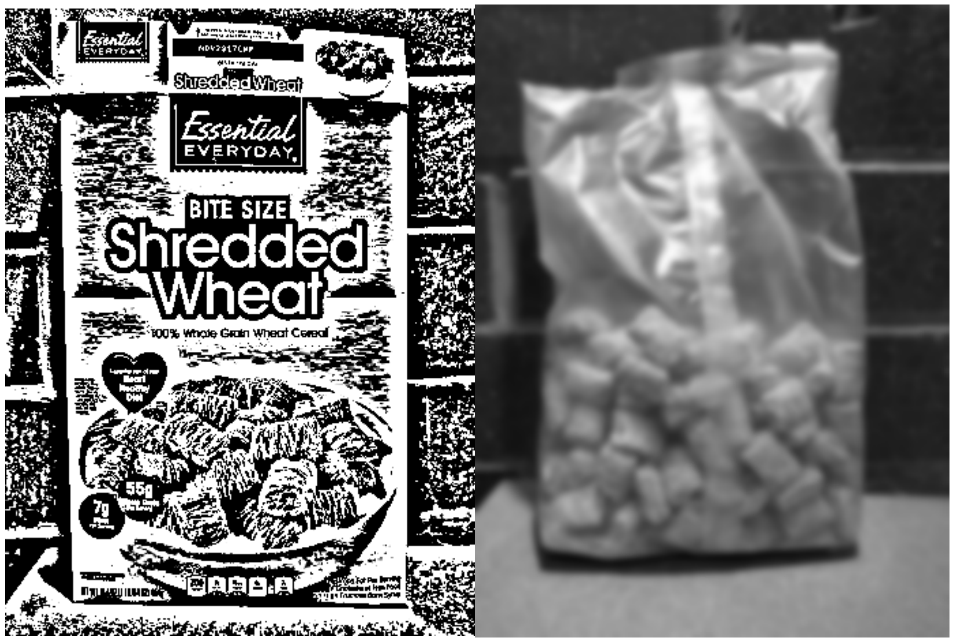
1. The original, filtered, and hybrid images for all 3 examples (i.e., 1 provided and 2 pairs of your own).

   Provided image:

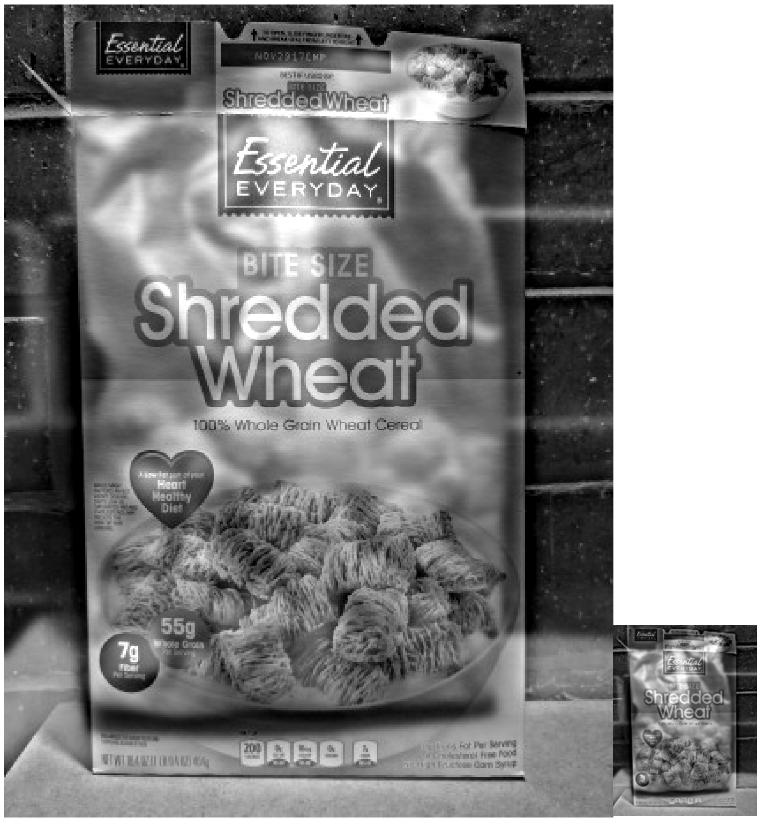
   Original:



   Filtered:

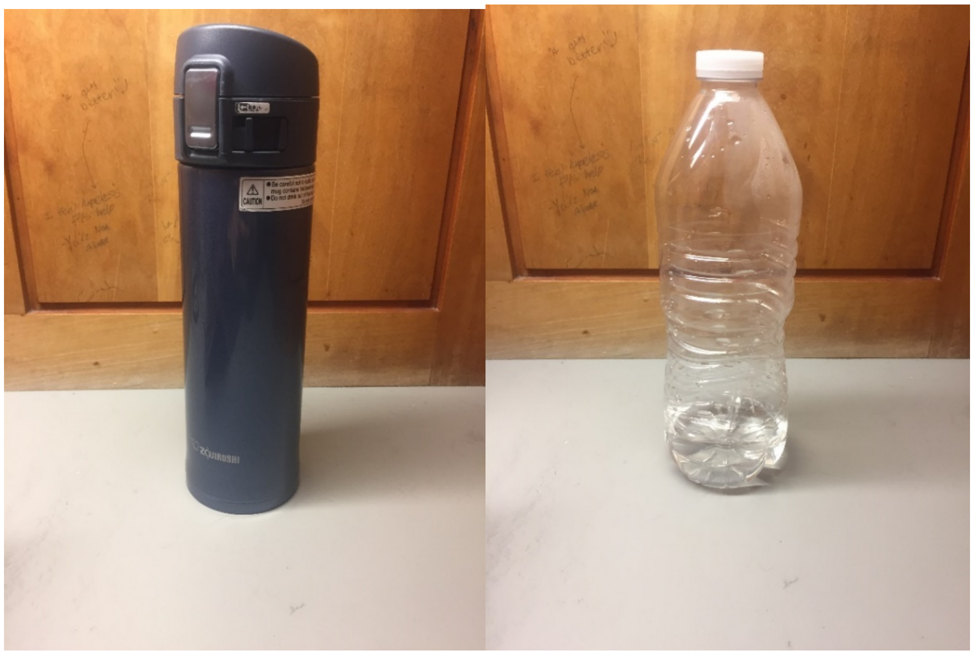


Hybrid image:

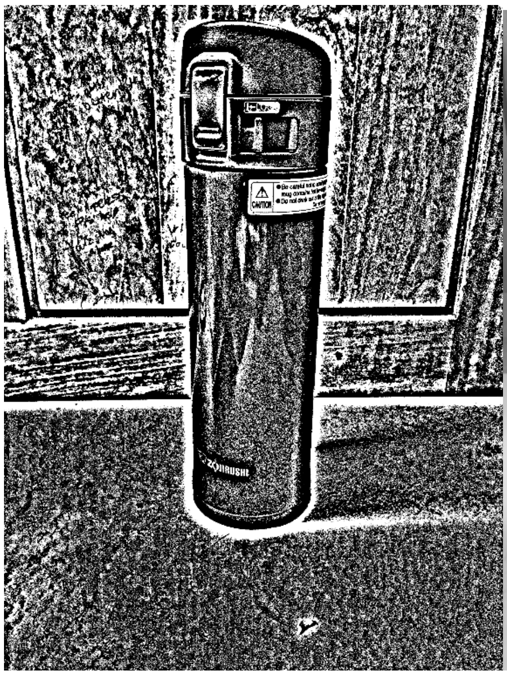


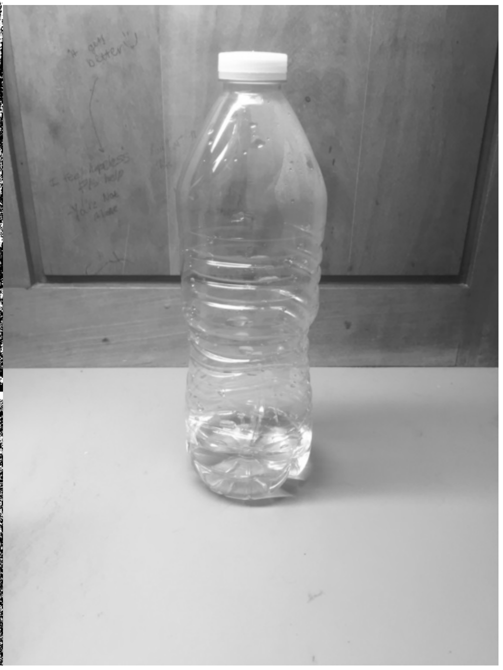
Own image 1:

Original:

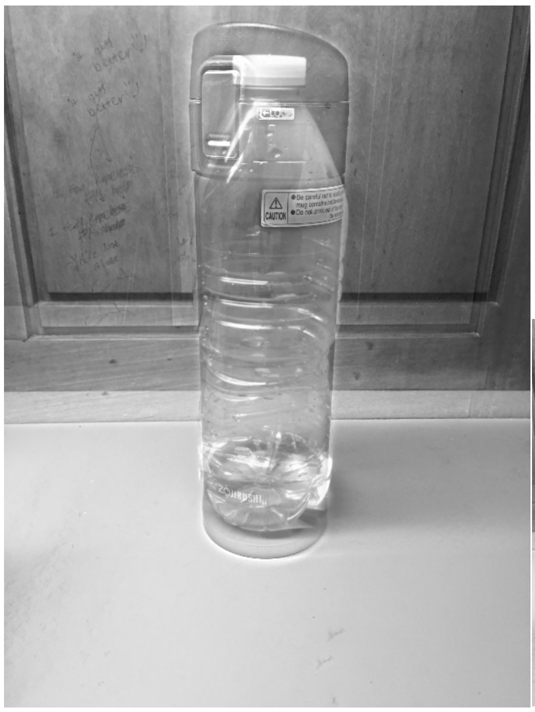


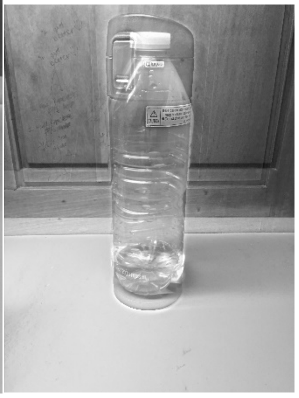
Filtered:





Hybrid:

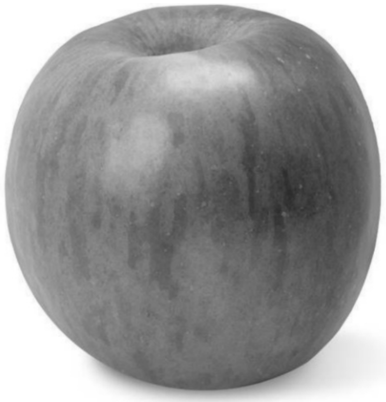


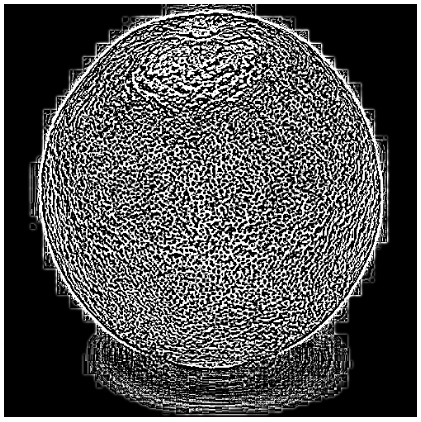


Own image 2:

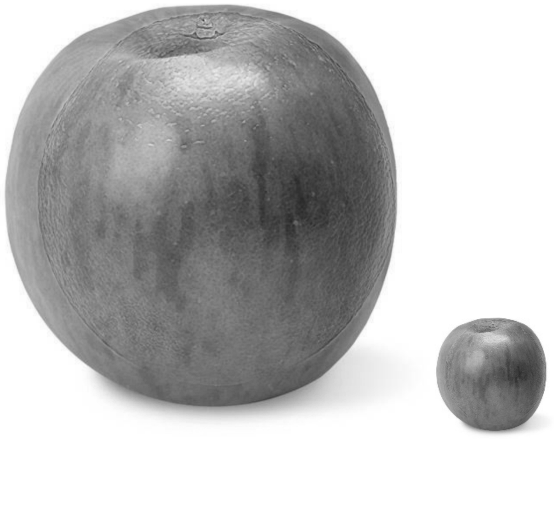
Original:

Filtered:





Hybrid:



2. Explanation of any implementation choices such as library functions or changes to speed up computation.

   When applying a low-pass filter, I used a function from scipy library which is "scipy.ndimage.gaussian_filter" to speed up my computation.

   Also, I used the same function for my high-pass filter.

3. For each example, give the two σ values you used. Explain how you arrived at these values. Discuss how successful your examples are or any interesting observations you have made.

|  | Low-threshold | High-threshold |
|---|---|---|
| Provided image | 2.5 | 6 |
| Own image 1 | 2 | 20 |
| Own image 2 | 0.5 | 1.5 |

   I tried multiple values to figure out what σ would work the best. Here are some observations I found.

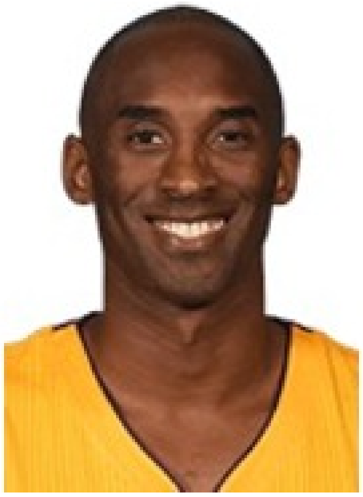
   1. High-threshold value should always be higher than the other one. In other words, the sigma value of detailed image is higher than that of blurry image.
   2. Optimal σ values vary from image to image
   3. Increasing low-threshold will make image blurrier. Also, if increasing high-threshold, the image will contain less details.

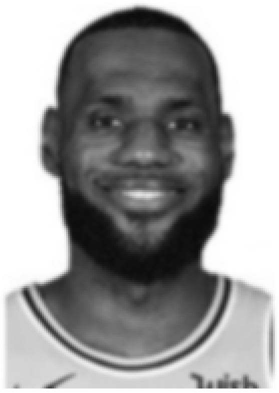
4. Optional: discussion and results of any bonus items you have implemented.

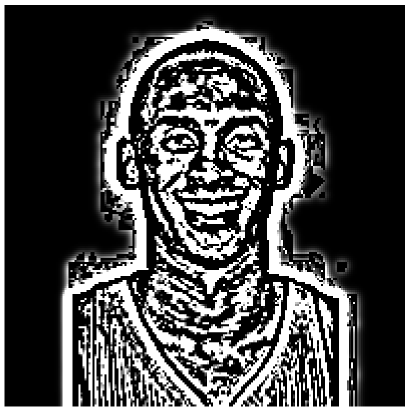
   Possible reasons of failure when implementing hybrid images:

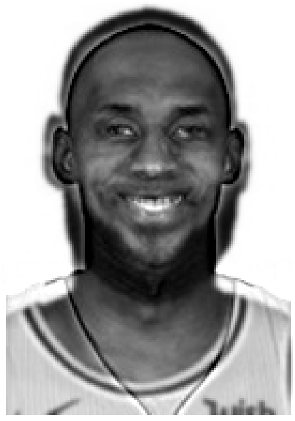
   a. The alignment of inputs. It sometimes would produce artifacts like an image border. Like the example I show below, there is clear border cutting across the hybrid image.

Example:



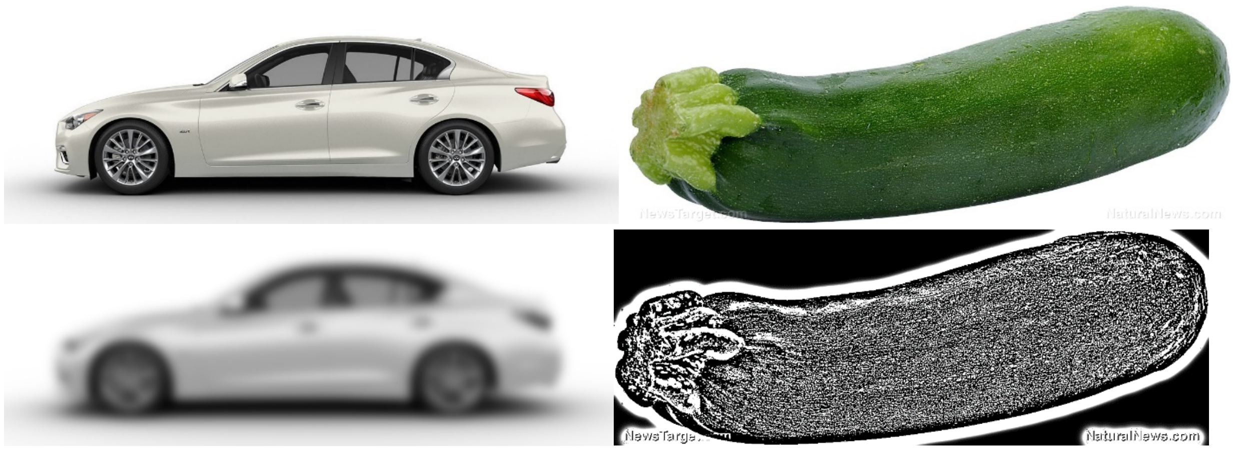


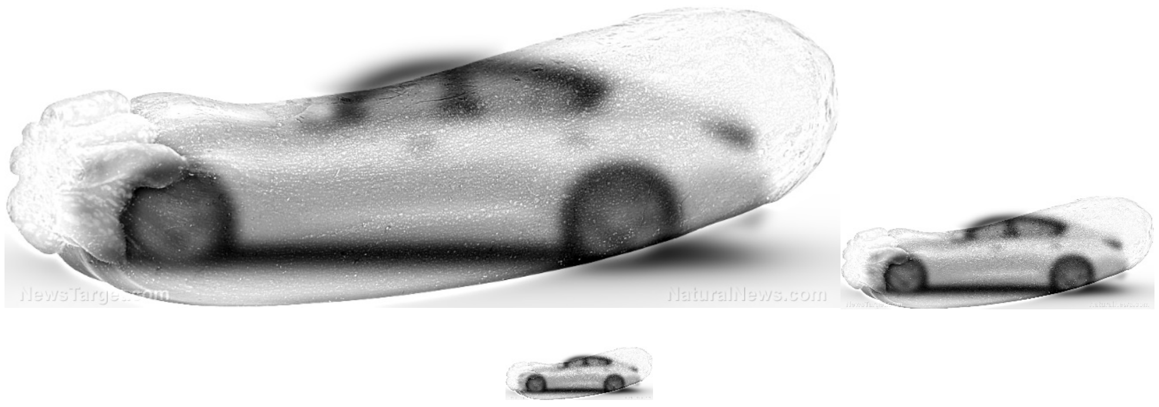




b. Shape of objects are quite different. These two objects are clearly visible at different viewing distances.

Example:

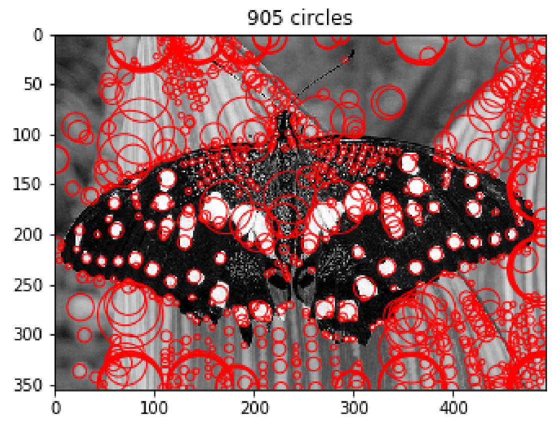




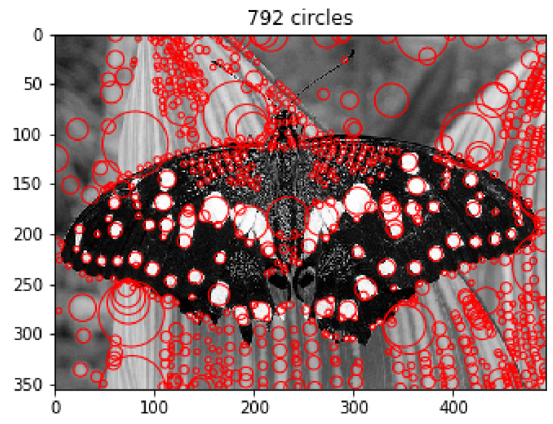
## Part 2:

1. The output of your circle detector on all the images (four provided and four of your own choice), together with running times for both the "efficient" and the "inefficient" implementation.

   Provided 1:

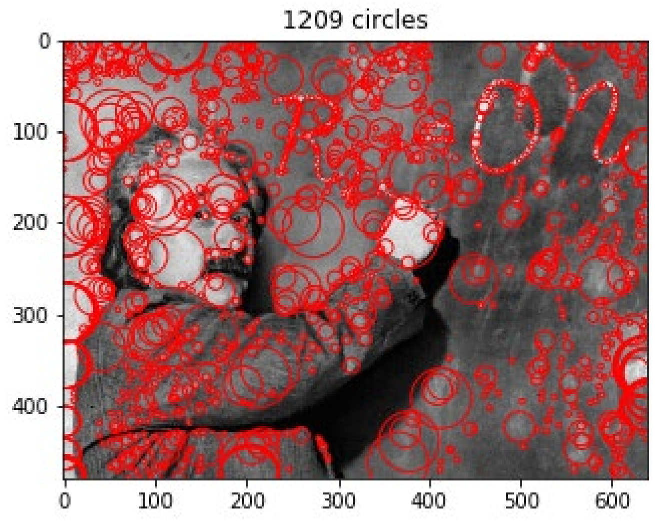


(Efficient)

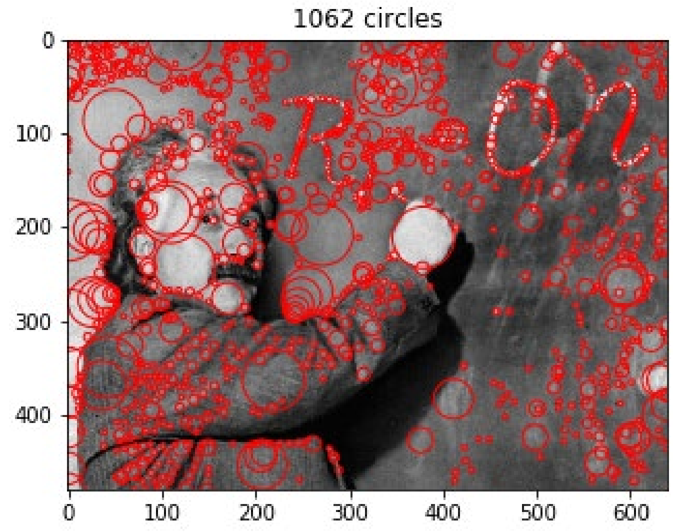


(Inefficient)

Provided 2:

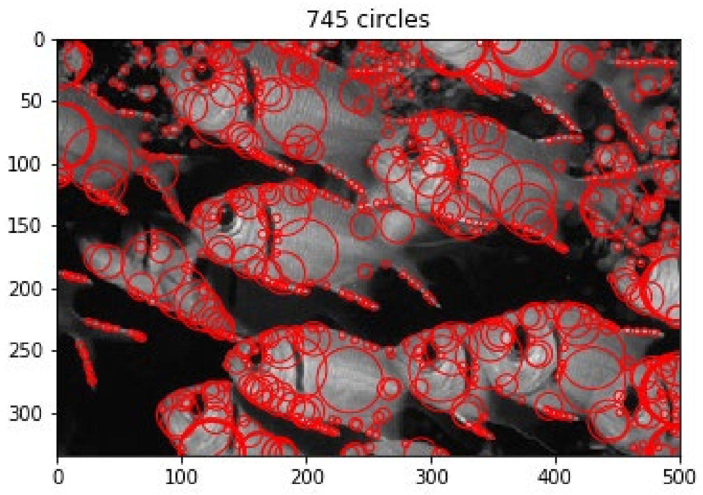


(Efficient)

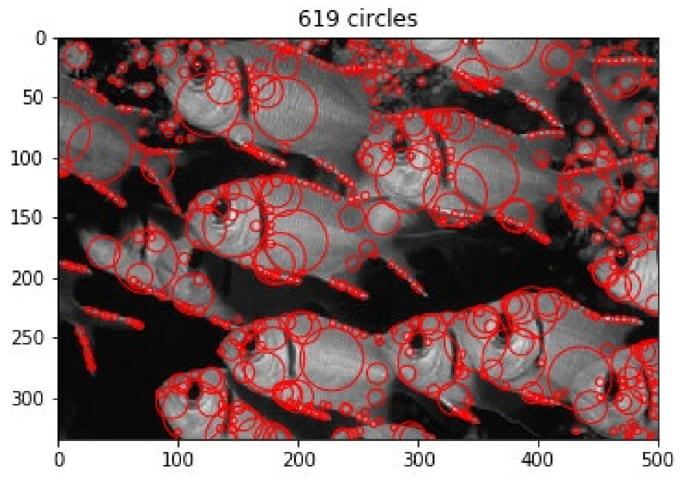


(Inefficient)

Provided 3:

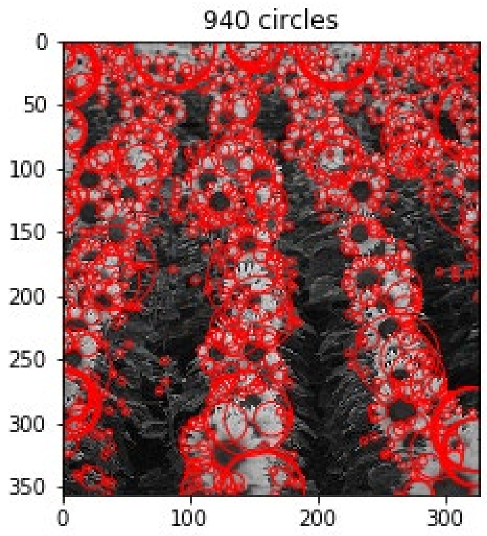


(Efficient)

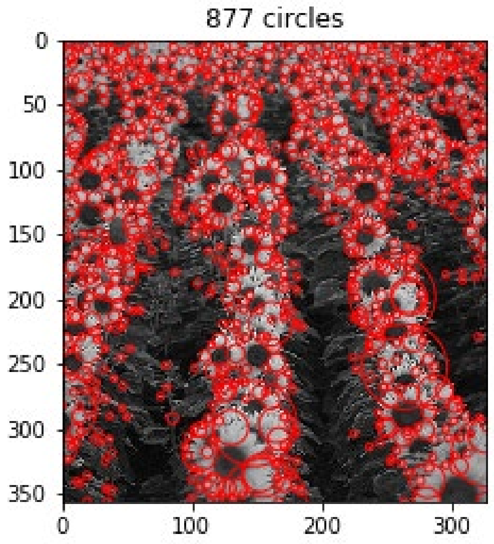


(Inefficient)

Provided 4:

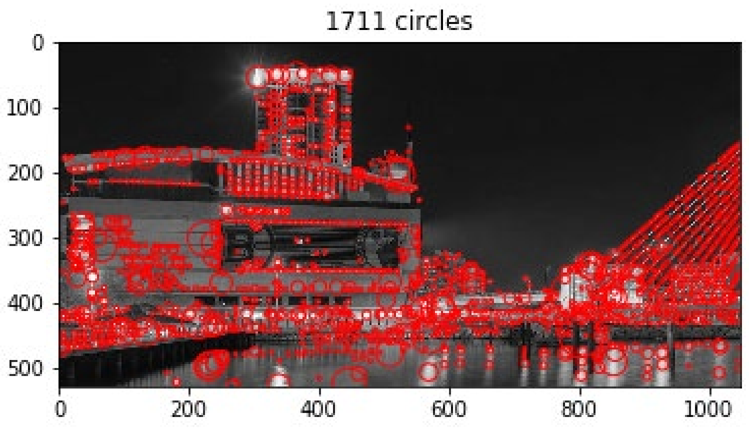


(Efficient)

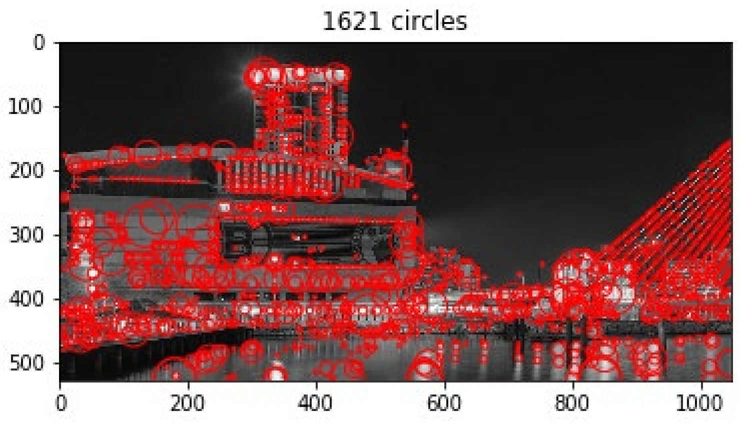


(Inefficient)

Test 1:

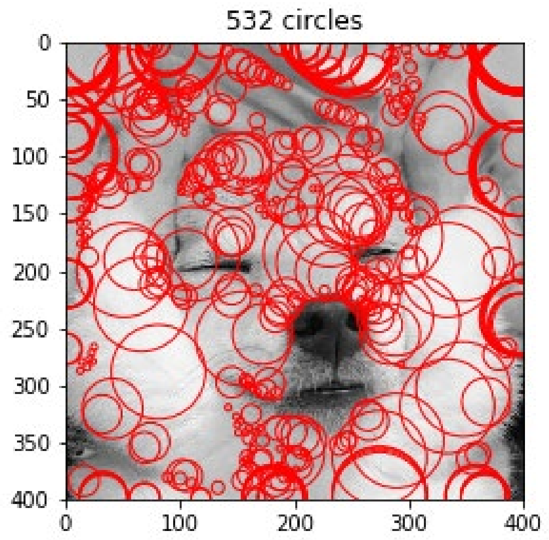


(Efficient)

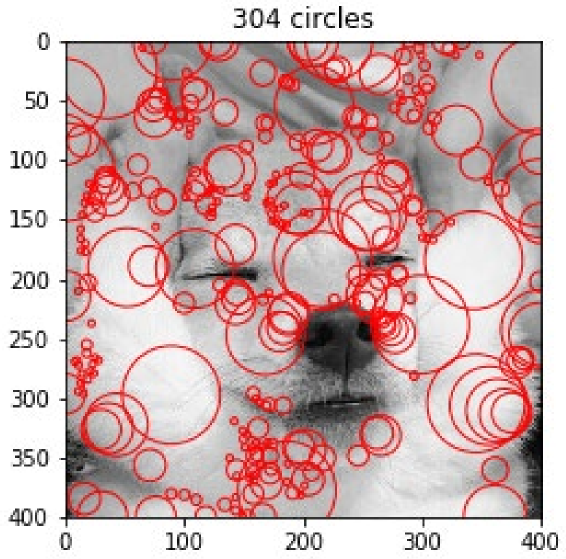


(Inefficient)

Test 2:

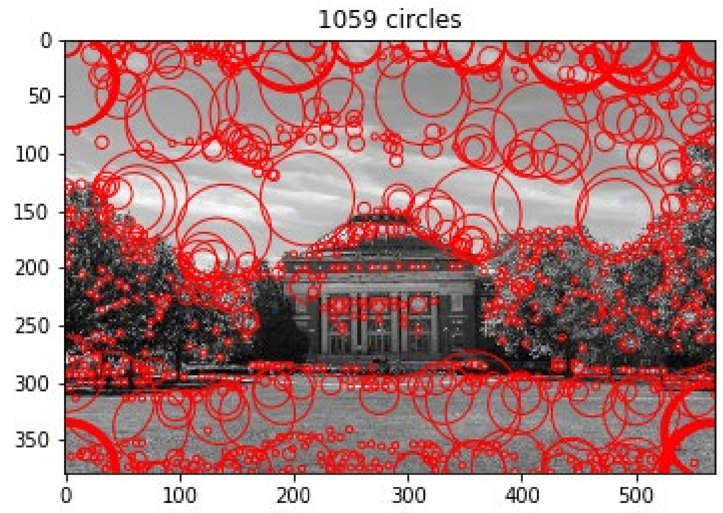


(Efficient)

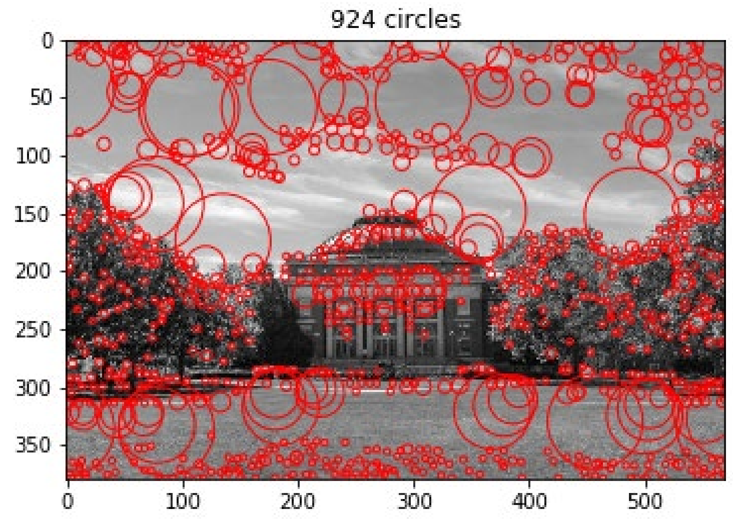


(Inefficient)

Test 3:

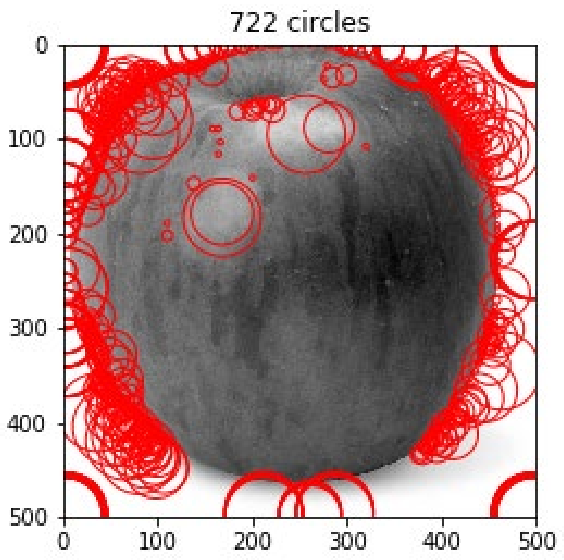


(Efficient)

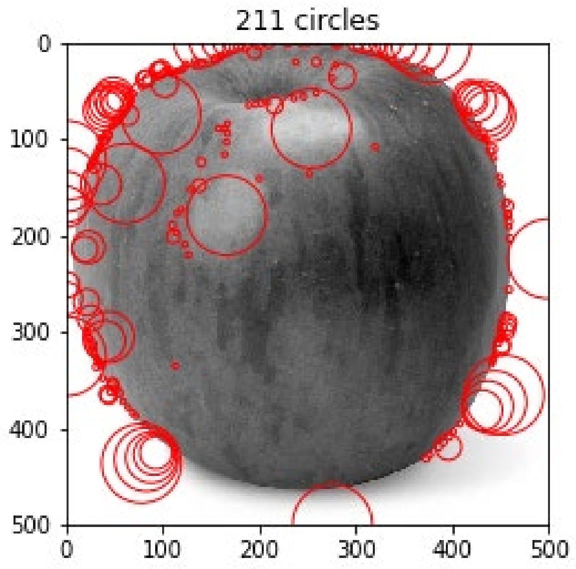


(Inefficient)

Test 4:



(Efficient)



(Inefficient)

Running time table:

| | Inefficient | Efficient |
|---|---|---|
| Butterfly | 0.5395763 seconds | 0.4158503999999996 |
| Einstein | 1.1284292999999996 seconds | 0.3876027999999998 seconds |
| Fishes | 0.3988560000000003 seconds | 0.3402718999999994 seconds |
| Sunflowers | 0.3525051999999995 seconds | 0.2575648000000008 seconds |
| Test1 | 1.724176700000001 | 0.842165399999999 seconds |
| Test2 | 0.6038616000000019 seconds | 0.24288419999999888 seconds |
| Test3 | 0.9878256000000007 seconds | 0.3139846999999989 seconds |
| Test4 | 0.8378399999999999 seconds | 0.42073119999999875 seconds |

2. An explanation of any "interesting" implementation choices that you made.
    a. I used a lot of scipy functions to generate gaussian filters, which speeds up my implementation.
    b. Parameters vary from images to images. I adjusted them based on my implementation results. I also attached my values selection table in Q3.
    c. I used down sample method to speed up the implementation. After comparing the results calculated by two methods, I found the method of increasing kernel size will generate less circles but is more time-consuming.
    d. For non-maximum suppression, I firstly find local maximum value using a (3x3) filter. Then performe 3D task.

3. An explanation of parameter values you have tried and which ones you found to be optimal.

| | Threshold | Initial sigma | Factor k | Levels | |
|---|---|---|---|---|---|
| Butterfly | 0.45 | 2 | 1.25 | 12 | |
| Einstein | 0.4 | 2 | 1.25 | 12 | |
| Fishes | 0.25 | 2 | 1.25 | 11 | |
| Sunflowers | 0.3 | 2 | 1.25 | 12 | |
| Test1 | 0.4 | 2 | 1.25 | 12 | |
| Test2 | 0.5 | 2 | 1.25 | 13 | |
| Test3 | 0.5 | 2 | 2.25 | 13 | |
| Test4 | 0.6 | 2 | 1.25 | 13 | |

Threshold:

Threshold varies from images to images. It decides the number of circles. After several trials, I choose the best value for each image.

Initial sigma:

Sigma determines what the size of features will be detected. Larger sigma would detect larger features. On the contrary, smaller value would care more about smaller features. I chose 2 as my initial sigma according to instruction.

Factor k:

I used 1.25 as my multiple-factor. It depends on the largest scale at which I want regions to be detected.

Levels (Iteration):

The number of levels affects the running time of my implementation. According to instructions, 10-15 is reasonable for implementation. I played around with these values and find the optimal value for each image.

4. Discussion and results of any extensions or bonus features you have implemented.