CS543 Assignment 2

Your Name: Li-Kai Chuang
Your NetId: likaikc2

Part 1 Fourier-based Alignment:

You will provide the following for each of the six low-resolution and three high-resolution images:

- Final aligned output image
- Displacements for color channels
- Inverse Fourier transform output visualization for ***both*** channel alignments ***without*** preprocessing
- Inverse Fourier transform output visualization for ***both*** channel alignments ***with*** any sharpening or filter-based preprocessing you applied to color channels

You will provide the following as further discussion overall:

- Discussion of any preprocessing you used on the color channels to improve alignment and how it changed the outputs
- Measurement of Fourier-based alignment runtime for high-resolution images (you can use the python time module again). How does the runtime of the Fourier-based alignment compare to the basic and multiscale alignment you used in Assignment 1?

A: Channel Offsets

Replace <C1>, <C2>, <C3> appropriately with B, G, R depending on which you use as the base channel. Provide offsets in the **original image coordinates** and be sure to account for any cropping or resizing you performed.

Low-resolution images (using channel B as base channel):

Image	G (h,w) offset	R (h,w) offset
00125v.jpg	(5, 2)	(11, 1)

00149v.jpg	(4, 2)	(10, 2)
00153v.jpg	(7, 3)	(15, 5)
00351v.jpg	(4, 1)	(14, 1)
00398v.jpg	(5, 3)	(12, 4)
01112v.jpg	(0, 0)	(6, 1)

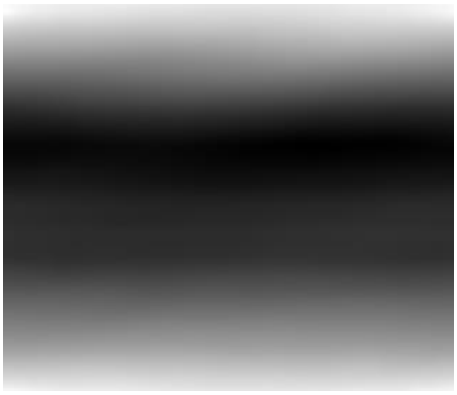
High-resolution images (using channel B as base channel):

Image	G (h,w) offset	R (h,w) offset
01047u.tif	(25, 19)	(72, 33)
01657u.tif	(56, 9)	(113, 12)
01861a.tif	(72, 39)	(148, 62)

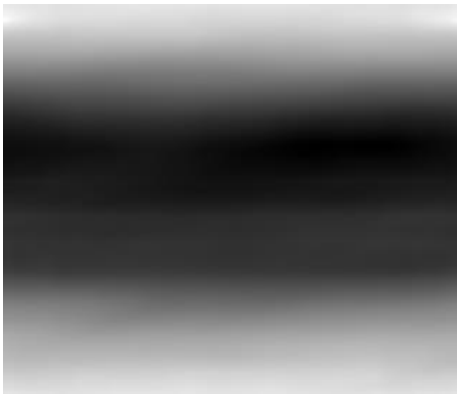
B: Output Visualizations

For each image, insert 5 outputs total (aligned image + 4 inverse Fourier transform visualizations) as described above. When you insert these outputs be sure to clearly label the inverse Fourier transform visualizations (e.g. “G to B alignment without preprocessing”).

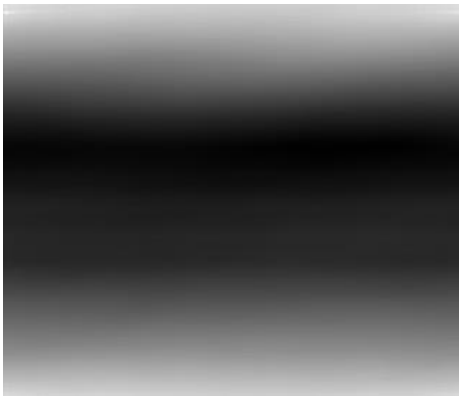
00125v.jpg



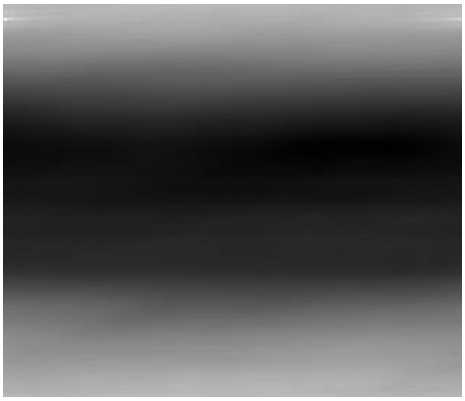
(G, raw)



(R, raw)

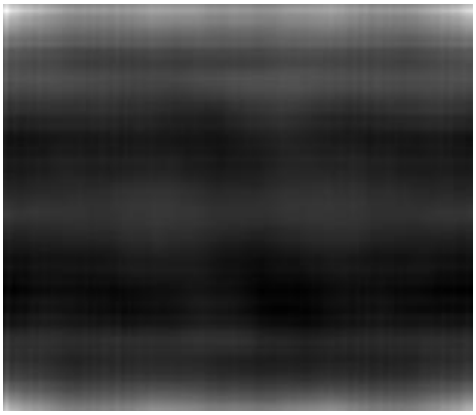


(G, sharpened)

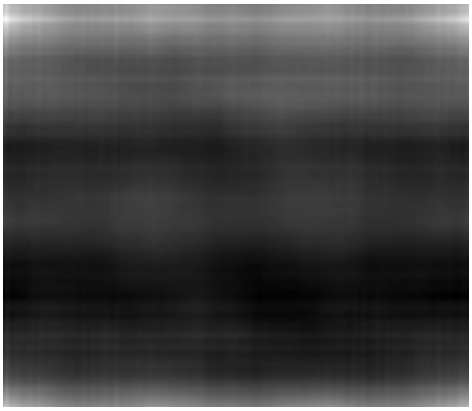


(R, sharpened)

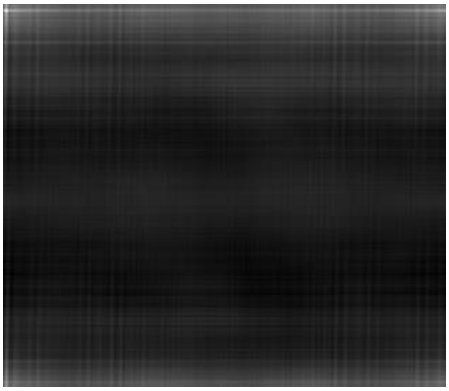
00149v.jpg



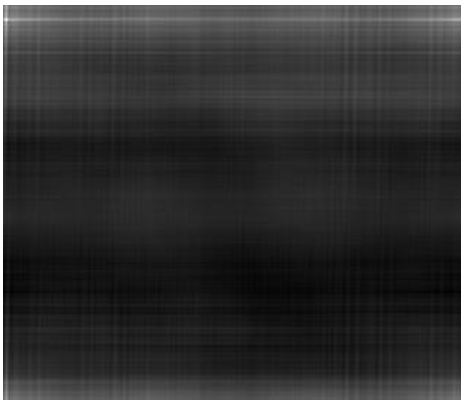
(G, raw)



(R, raw)

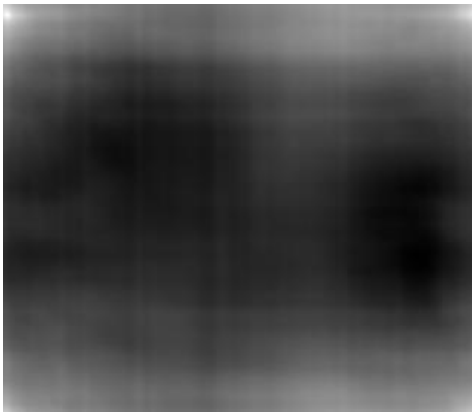


(G, sharpened)

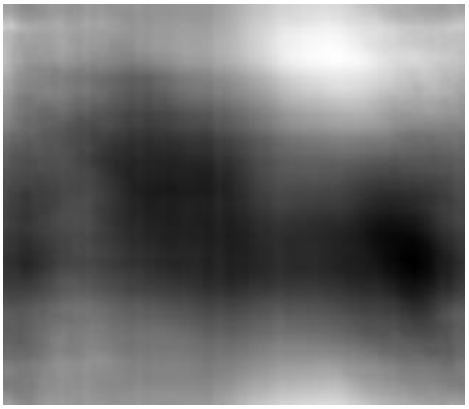


(R, sharpened)

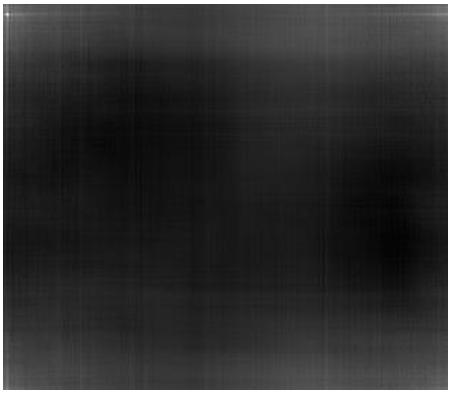
00153v.jpg



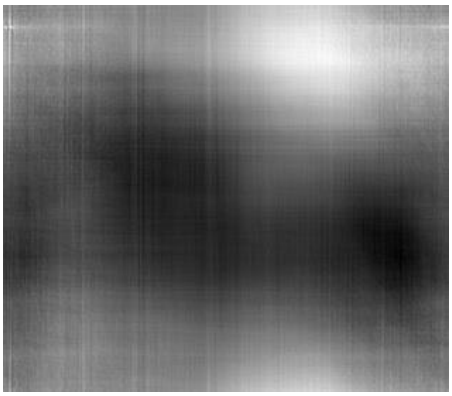
(G, raw)



(R, raw)

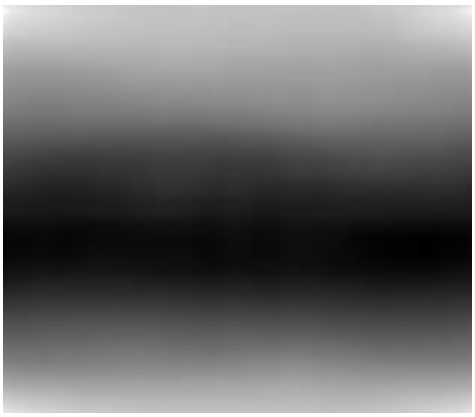


(G, sharpened)

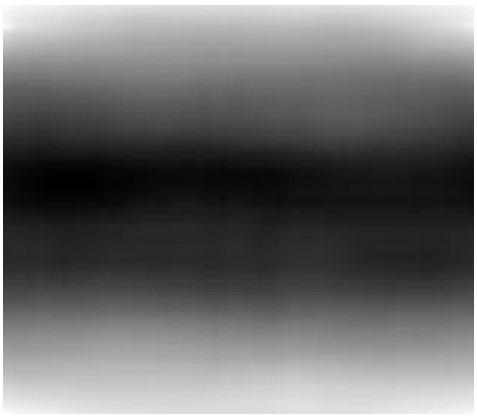


(R, sharpened)

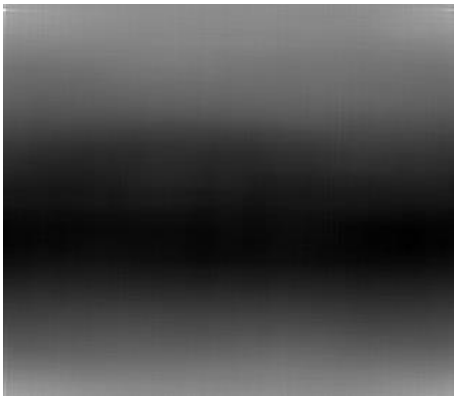
00351v.jpg



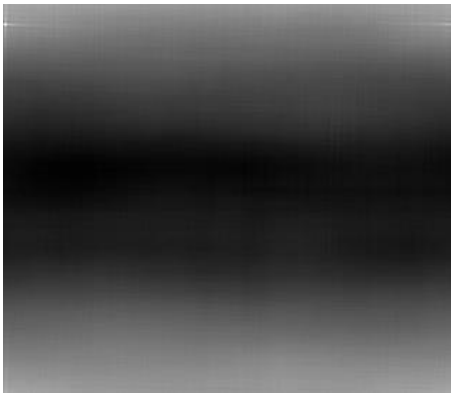
(G, raw)



(R, raw)

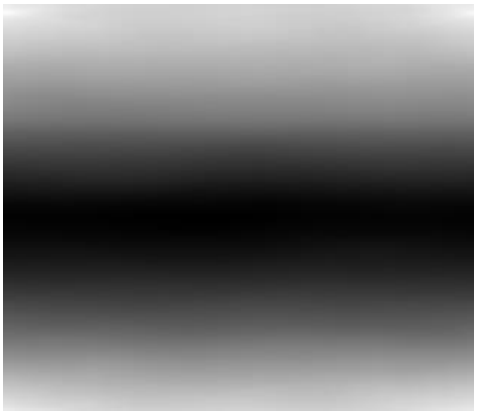


(G, sharpened)



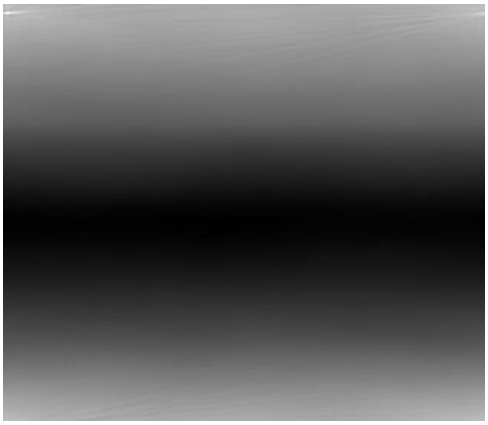
(R, sharpened)

00398v.jpg

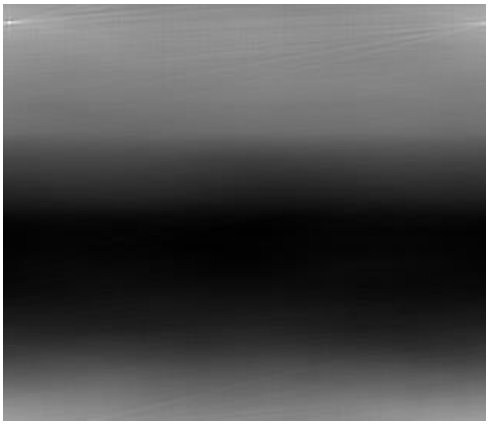


(G, raw)

(R, raw)

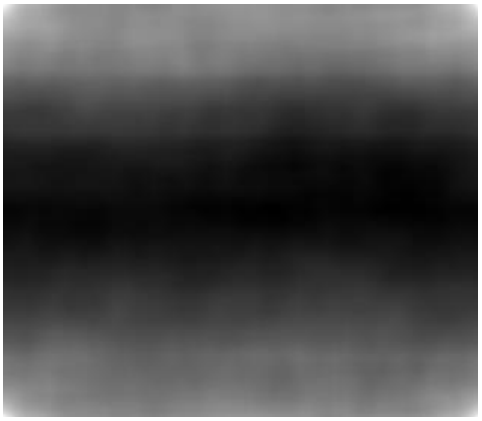


(G, sharpened)

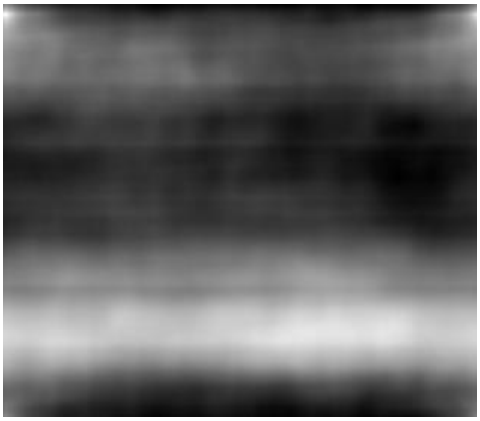


(R, sharpened)

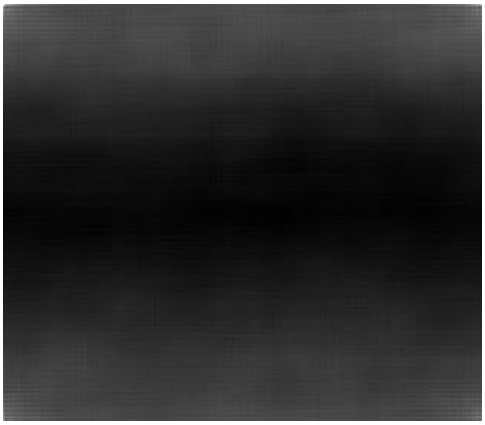
01112v.jpg



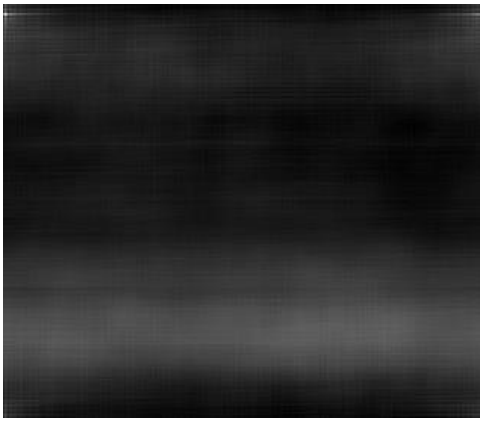
(G, raw)



(R, raw)

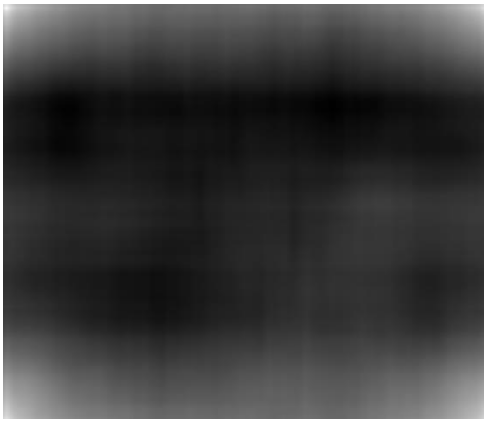


(G, sharpened)

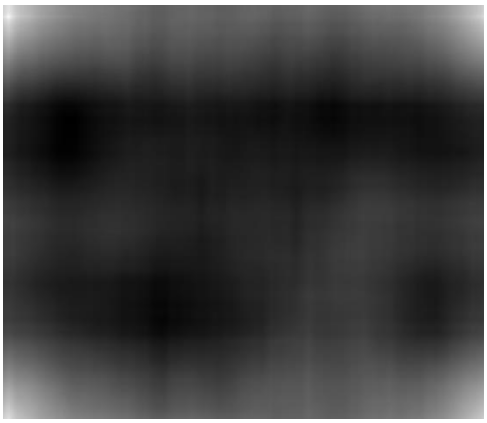


(R, sharpened)

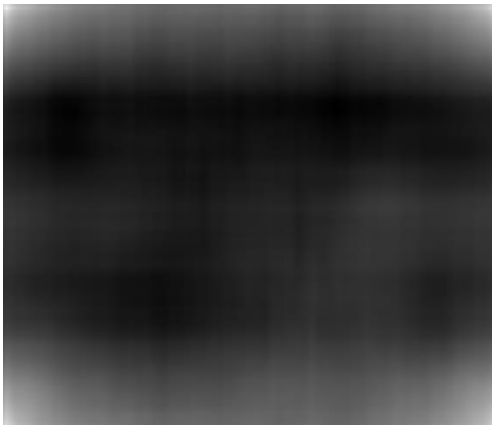
01047u.tif



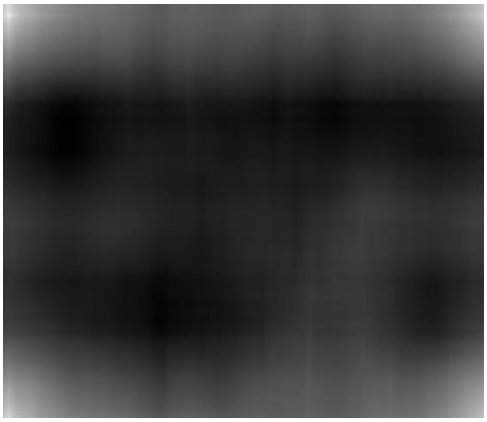
(G, raw)



(R, raw)

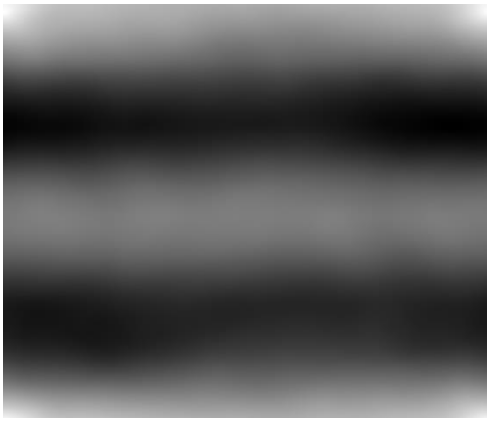


(G, sharpened)



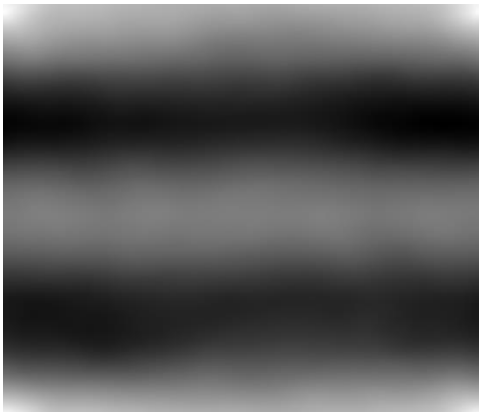
(R, sharpened)

01657u.tif

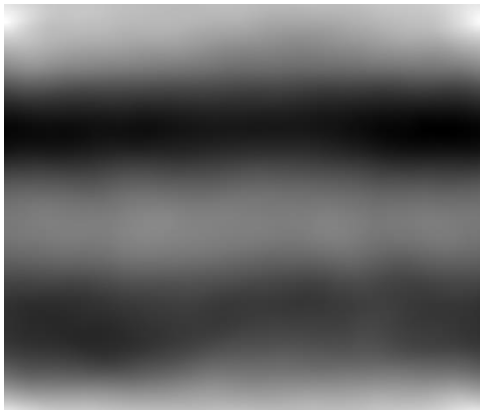


(G, raw)

(R, raw)

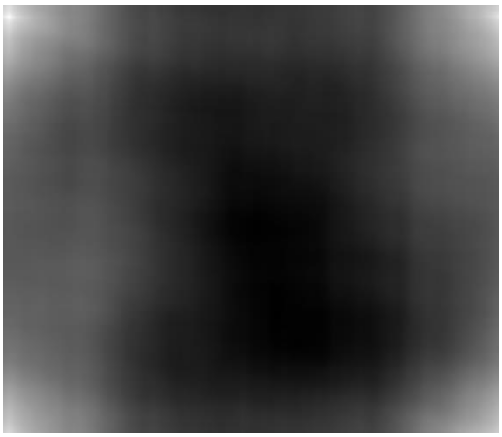


(G, sharpened)

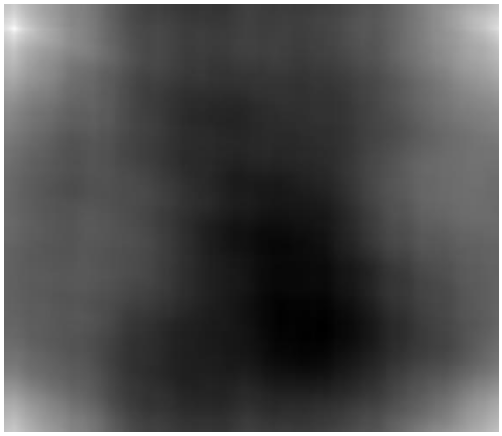


(R, sharpened)

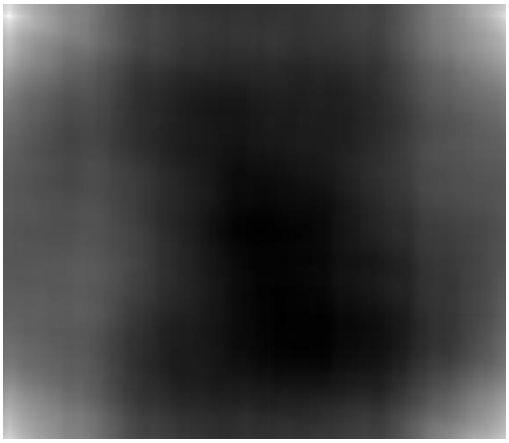
01861a.tif



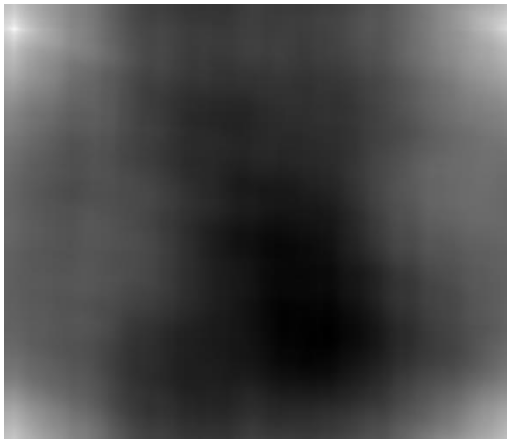
(G, raw)



(R, raw)



(G, sharpened)



(R, sharpened)

C: Discussion and Runtime Comparison

For the preprocessing, I first do image sharpening before fourier transform. That way we can get a clearer point in the inverse fourier. We can compare the raw and preprocessed versions of inverse fourier. We can clearly see that the preprocessed version is generally darker and we can see our desired point would look much brighter than it does in the raw version. This way there would be less ambiguity and we can find the point more easily.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Unit: seconds

Image	Normal Alignment	Pyramid Alignment	Fourier Alignment
01047u.tif	Over 3 minutes	9.418	2.633
01657u.tif	Over 3 minutes	9.077	3.146
01861a.tif	Over 3 minutes	9.038	2.247

For the runtime, theoretically, it's faster when we use fourier alignment, since we only need to do pixel-wise multiplication in frequency domain to perform convolution in the original image, according to the convolution theorem. This could significantly decrease the time complexity from O(n^2*m^2) to O(n^2). From the table above, we can see that the results verify our thoughts that fourier alignment is much faster than a normal and pyramid alignment.

Part 2 Scale-Space Blob Detection:

You will provide the following for ***8 different examples*** (4 provided, 4 of your own):

- original image
- output of your circle detector on the image
- running time for the "efficient" implementation on this image
- running time for the "inefficient" implementation on this image

You will provide the following as further discussion overall:

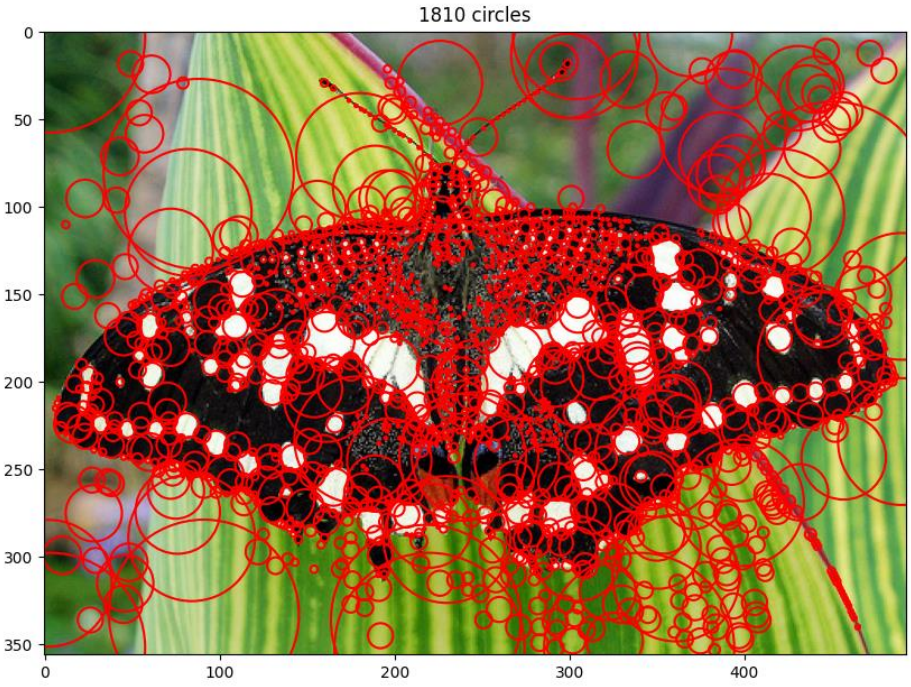
- Explanation of any "interesting" implementation choices that you made.
- Discussion of optimal parameter values or ones you have tried

Used sigma:
[0.1, 0.6, 1, 1.5, 2.25, 3.375, 5.0625, 7.59375, 11.390625, 17.0859375, 25.62890625, 38.443359375, 57.6650390625]

Nonmaximum Suppression: using 3x3xh, where h is the number of layers in the scale space.

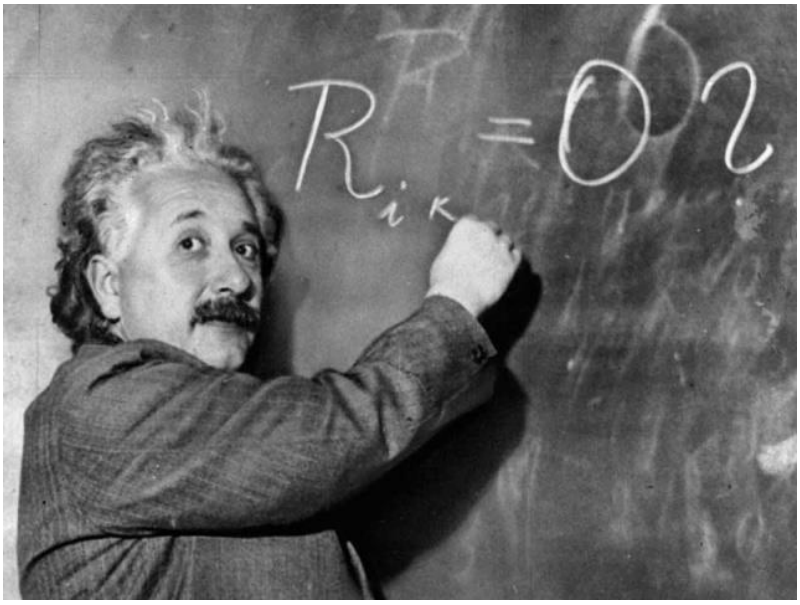
The downsample version of output images are not attached since the result is pretty much the same as the normal version, it's just it's more efficient. So only the running time and the discussions would be presented here.

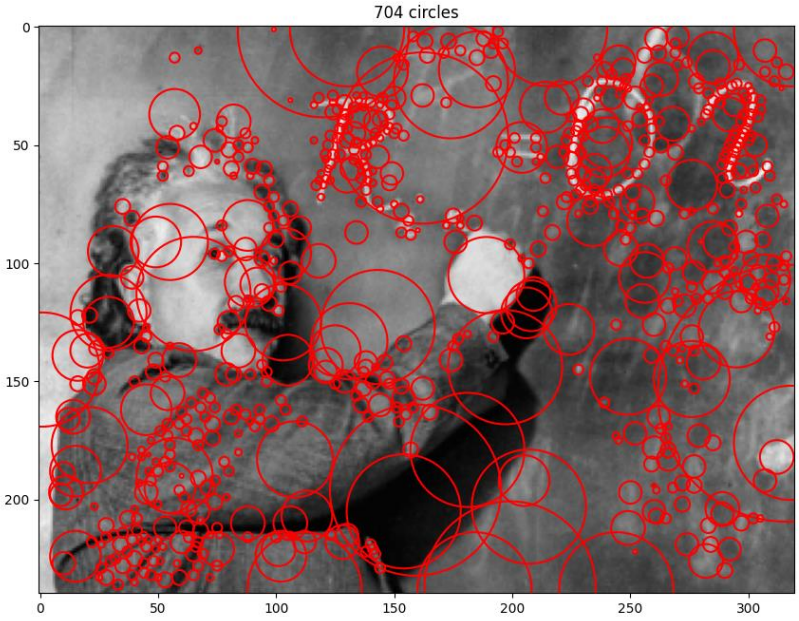
Example 1:butterfly



Efficient: 3.953 s
Inefficient: 4.196 s

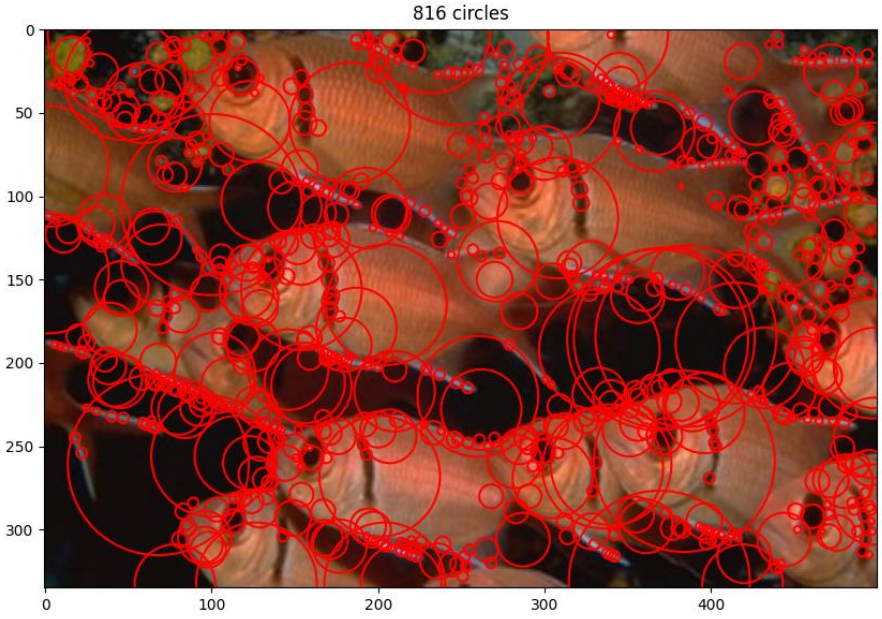
Example 2: einstein





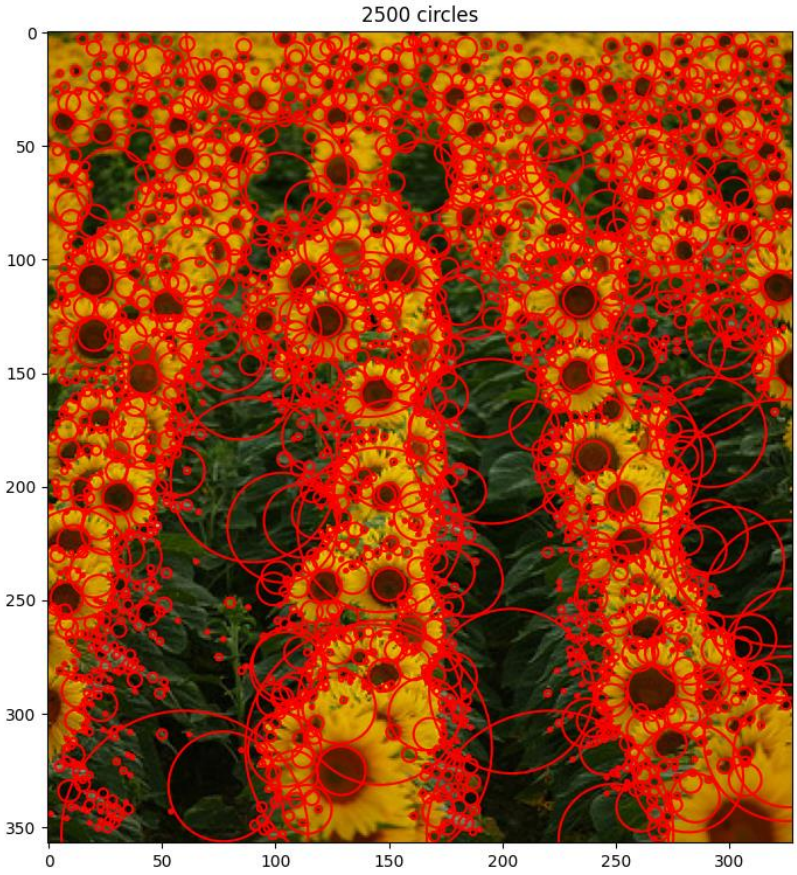
Efficient: 7.127 s
Inefficient: 7.388 s

Example 3: fishes



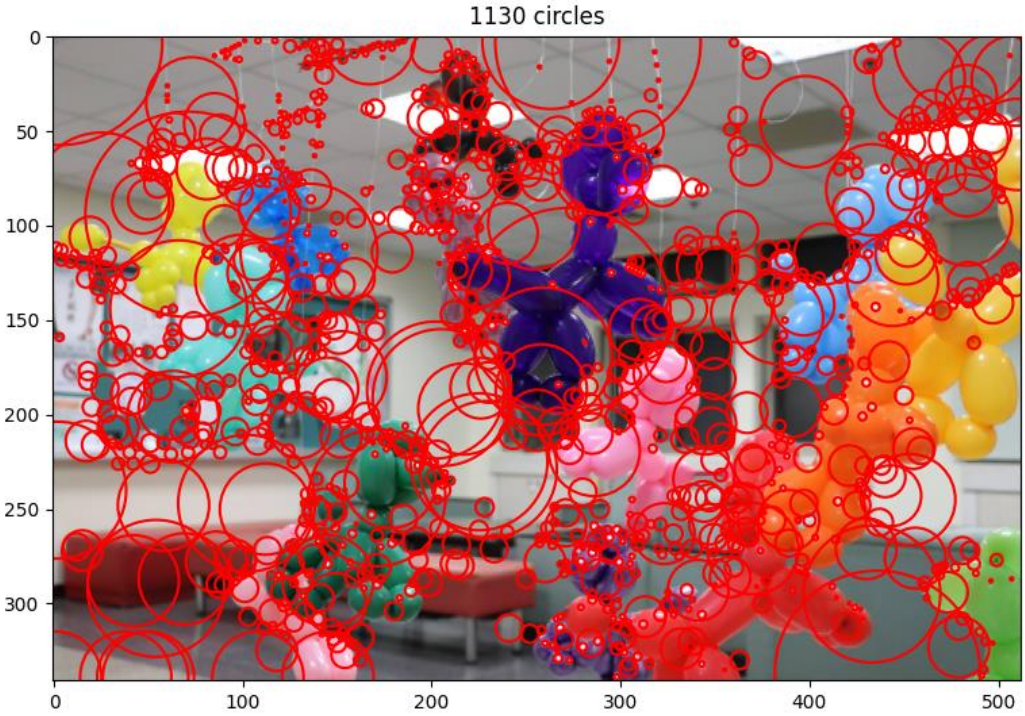
Efficient: 3.666 s
Inefficient: 4.362 s

Example 4: sunflowers



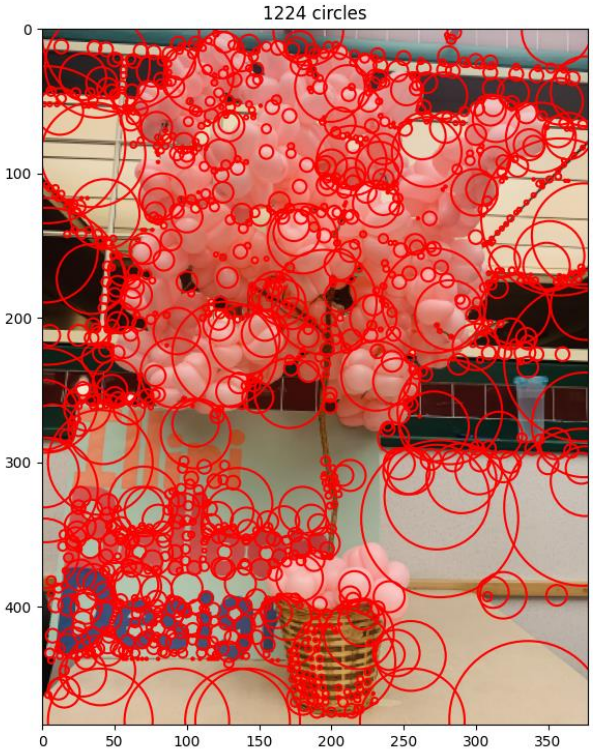
Efficient: 2.616 s
Inefficient: 2.898 s

Example 5: dog



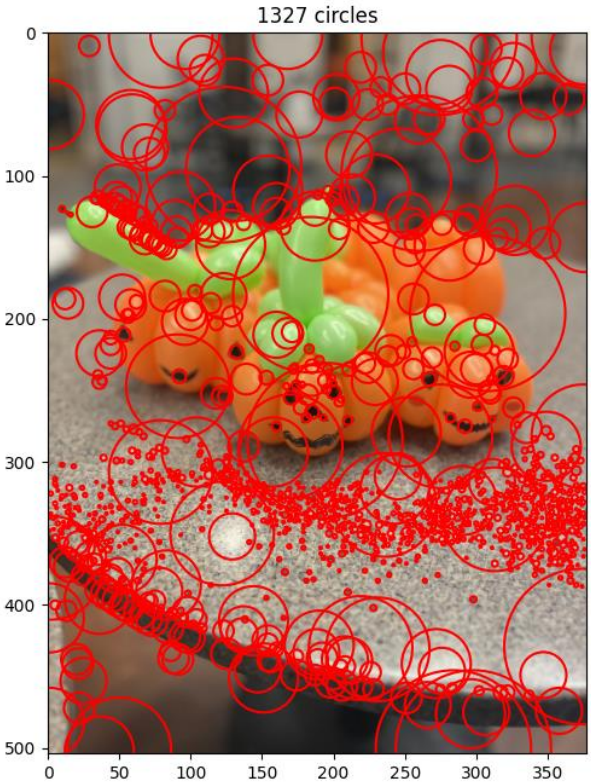
Efficient: 3.973 s
Inefficient: 4.440 s

Example 6: cherry



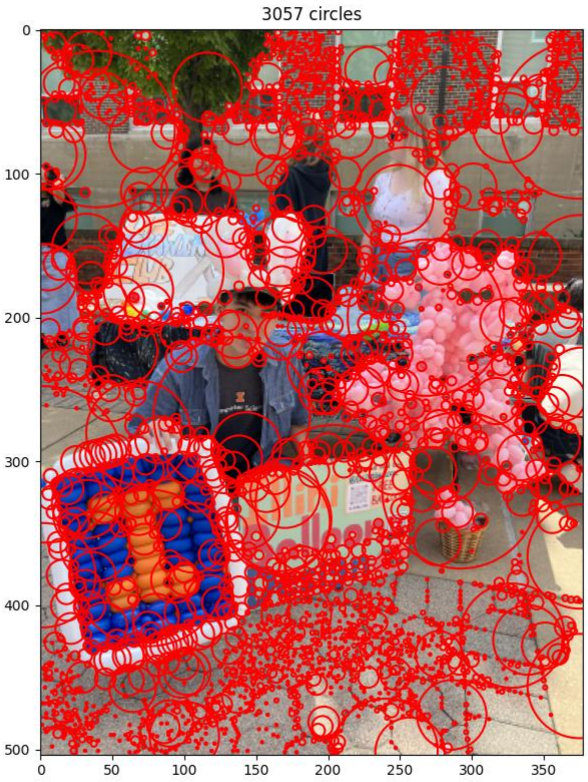
Efficient: 15.828 s
Inefficient: 17.405 s

Example 7: pumpkin



Efficient: 4.205
Inefficient: 4.806

Example 8: QuadDay



Efficient: 4.354 s
Inefficient: 4.569 s

Discussion:

To detect the proper size of blobs in the image, we need to set a series of decent value of sigma, I chose :
[0.1, 0.6, 1, 1.5, 2.25, 3.375, 5.0625, 7.59375, 11.390625, 17.0859375, 25.62890625, 38.443359375, 57.6650390625]
This way, we can detect small dots such as the dots on the body of the butterfly, we're also able to detect big blobs since our sigma is big enough.

Also, to avoid appearing too many overlapping blob detection, I use 3x3xh neighbor instead of 3x3x3 neighbor to do nonmaximum suppression, where h is the number of layers in the scale space. So basically a point would only be marked as an interest point when it's the largest among all his neighbors.

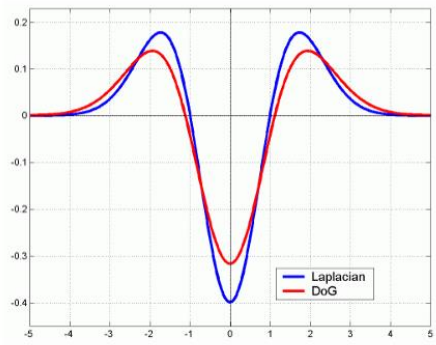
An interesting thing when I implement downsampling version is that we don't need to do scale normalization since we're performing LoG with a fixed sigma (scale). Additionally, the running time of downsampling is indeed faster than the normal verison. The reason is obvious because we don't need to do lots of convolution computation. The difference of running time is not obvious when it's a low-resolution image (from my results, the difference is about 0.2~0.5 seconds), but when it comes to high-resolution image or image that has lots of blobs in it, the time difference becomes larger and can reach roughly 2 seconds.

Another cool thing about downsampling is that when we upsamples the images, I think the cubic interpolation performs the best and is the most similar to the output of LoG in the normal version. Bilinear is fine but not as good, in my opinion.

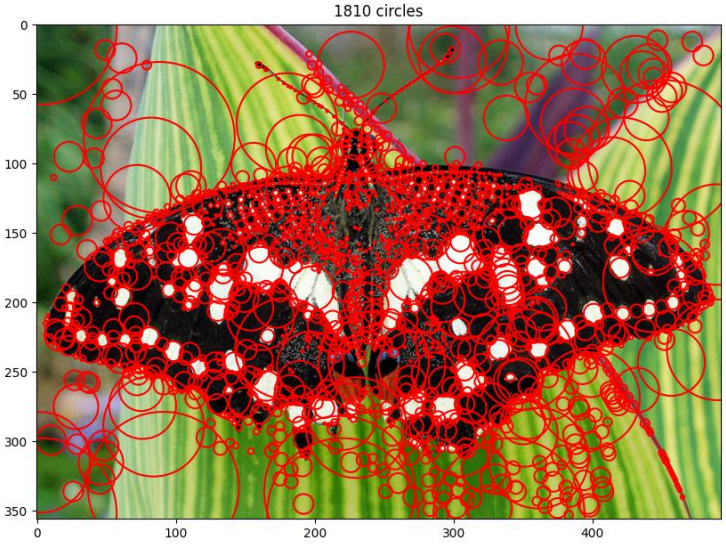
Bonus:

Blob-Detection Extra Credit

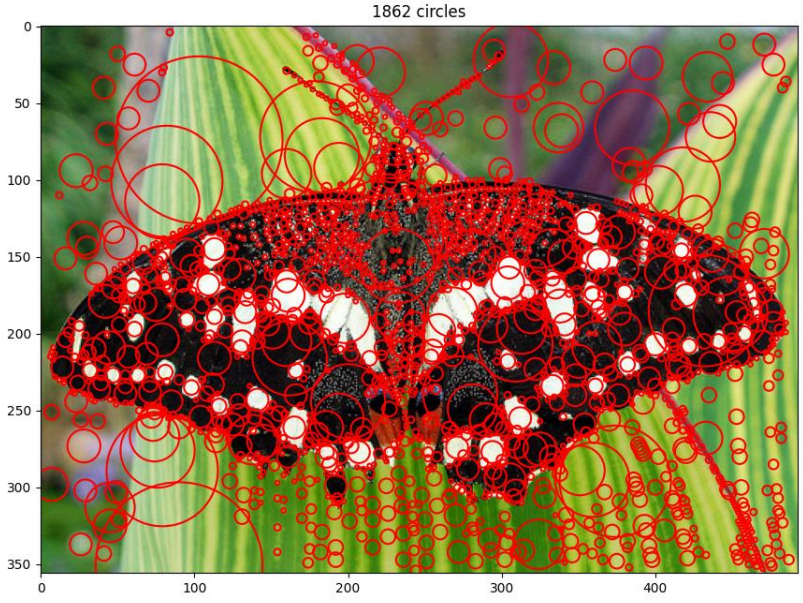
- Approximate LoG with a *difference of Gaussians* (DoG)
  - Laplacian: $\sigma^2\left(G_{xx}(x,y,\sigma) + G_{yy}(x,y,\sigma)\right)$
  - DoG: $G(x,y,k\sigma) - G(x,y,\sigma)$



I implemented the DoG version. Using the difference of gaussian to approximate the shape of laplacian of gaussain. Since we're just approximating the shape, we don't need to do scale normalization but we still need square response to detect both black and white blobs. However, the radius of scale seems to have a slight difference with the LoG version:



(LoG)



(DoG)

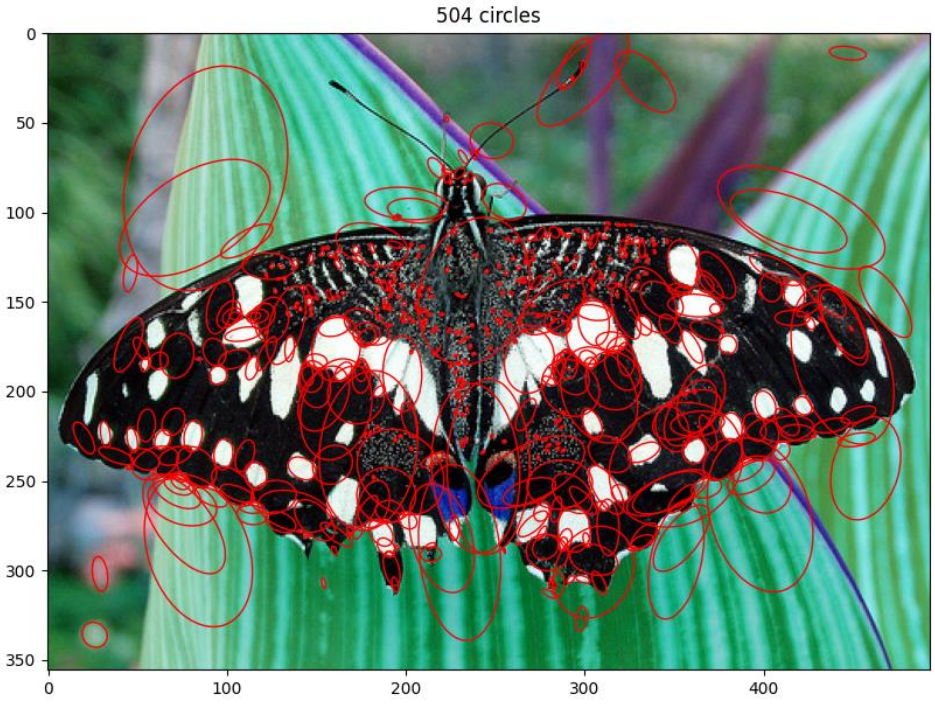
We can see that two outputs look pretty much the same. They detect the blobs at the same location and have almost the same number of detection. However, we can see in the DoG version that the circle is not tightly covering the region it's supposed to cover. I think a possible reason is that DoG detects a larger circle compared to LoG when they are using same sigma. Therefore, for a larger circle if we still apply sigma*sqrt(2) to be its radius, that'd be too smaller. That can account for the reason of such artifacts.

For the running time, both are similar and don't show an obvious difference. I think it's reasonable since both of them are doing filtering. One is laplacian of gaussain, and the other is simply gaussian. The DoG method will need to do subtraction additionally but that would not be the dominating time. Therefore, I think there is not computational advantage of one over the other.

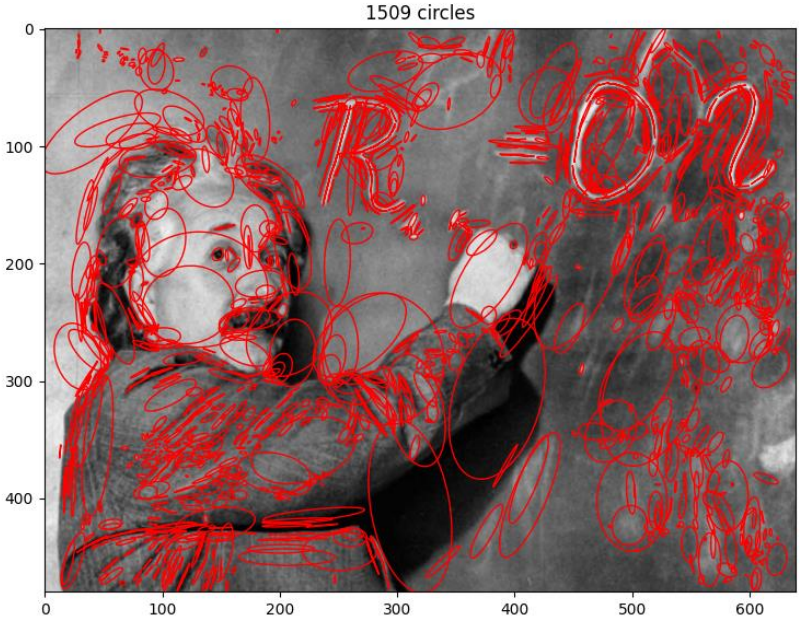
I also implemented the ellipse version:

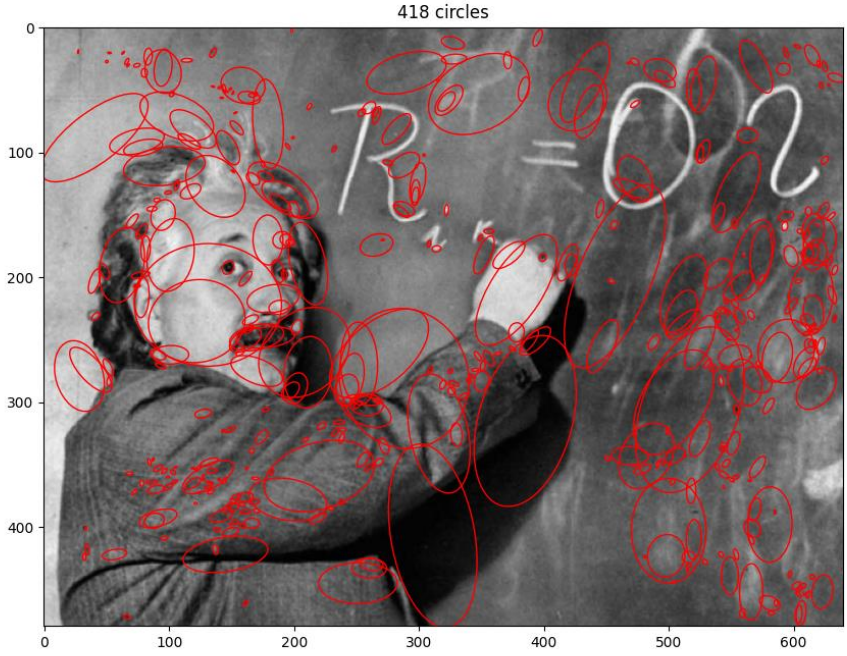
For each circle, we construct a square window with a length of 2r. Then we compute the second moment matrix R. Diagonalize R and we can get a rotation matrix and a diagonal matrix which we can derive the eigenvalues from. Then we scale the two eigenvalues to make their average equal to r, which will give us the information of the length of two axes of the ellipse. The heading of the ellipse is determined by the rotation matrix.

We can see that for the region that looks like an edge, the ellipses are very thin. We want to eliminate those ellipses, so we set an threshold to check if the two eigenvalues have significant difference. If yes, we eliminate the ellipse, which will give us the result below.



We can see that the edges significantly disappear compared to previous pictures. Let’s see another examples.


1509 circles
0
100
200
300
400
0
100
200
300
400
500
600


418 circles
0
100
200
300
400
0
100
200
300
400
500
600