CS543 Assignment 2

Your Name: <luoyan li>
Your NetId: <luoyanl2>

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	(-1,2)	(-4,1)
00149v.jpg	(-3,2)	(-5,2)
00153v.jpg	(-1,2)	(-1,4)
00351v.jpg	(-5,0)	(-5,1)
00398v.jpg	(0,2)	(1,4)
01112v.jpg	(-7,0)	(-9,1)

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

Image	<G> (h,w) offset	<R> (h,w) offset
01047u.tif	(-5,21)	(10,32)
01657u.tif	(-13,8)	(-26,11)
01861a.tif	(-13,37)	(-20,62)

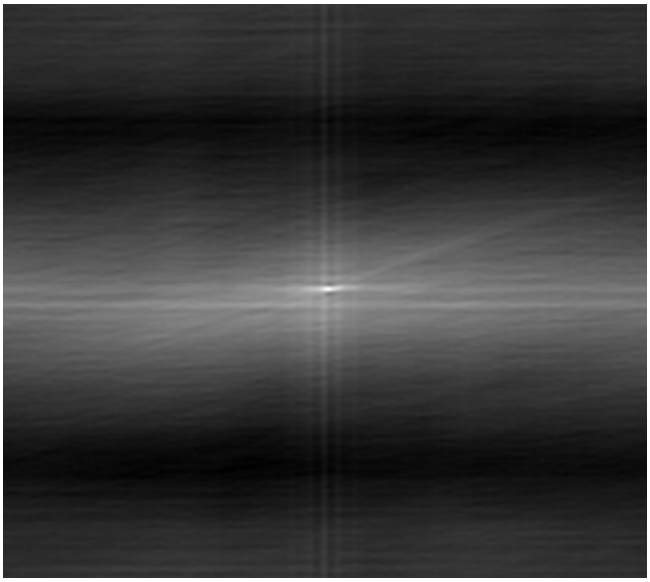
B: Output Visualizations

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”).

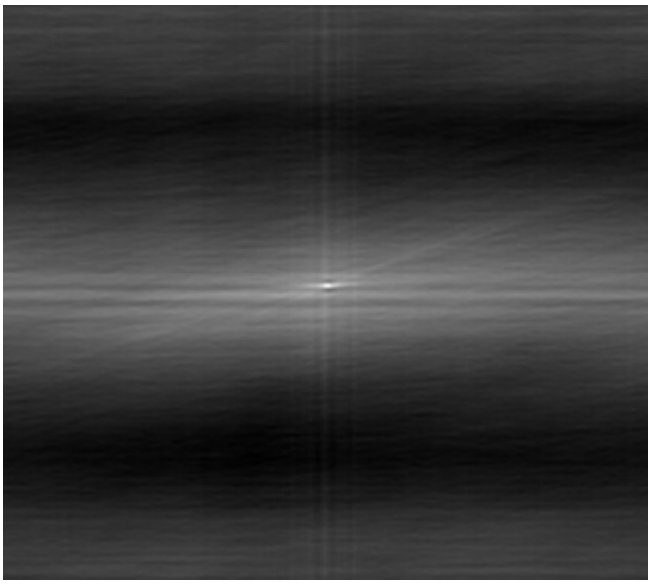
00125v.jpg

Aligned image:

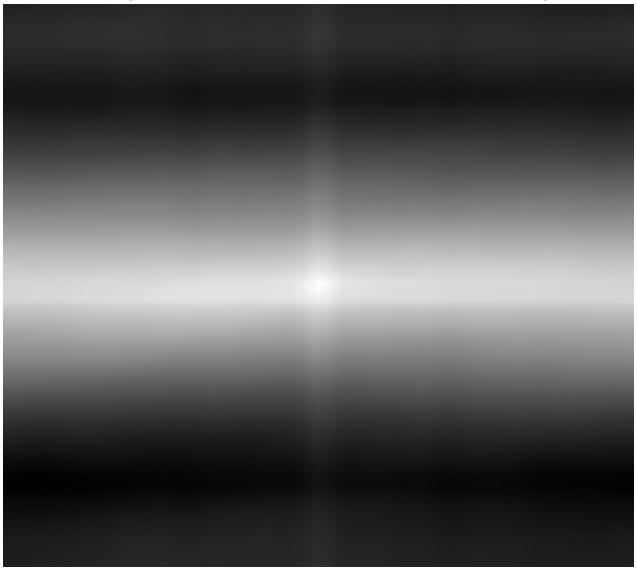
G to B alignment ifft:



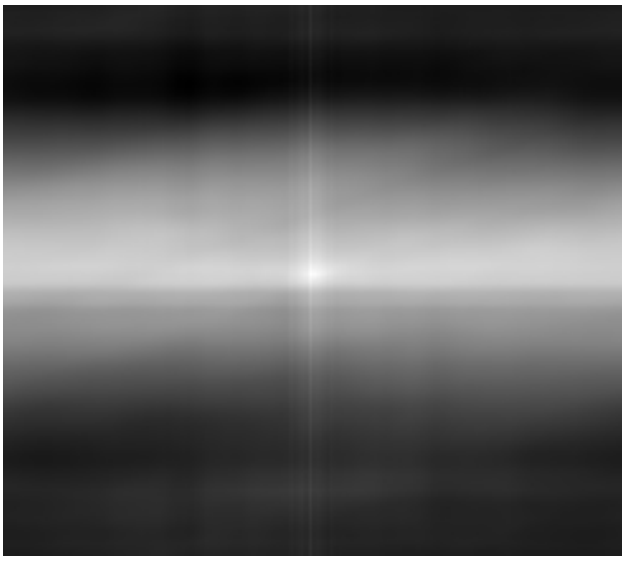
R to B alignment ifft:



G to B alignment ifft without preprocessing:

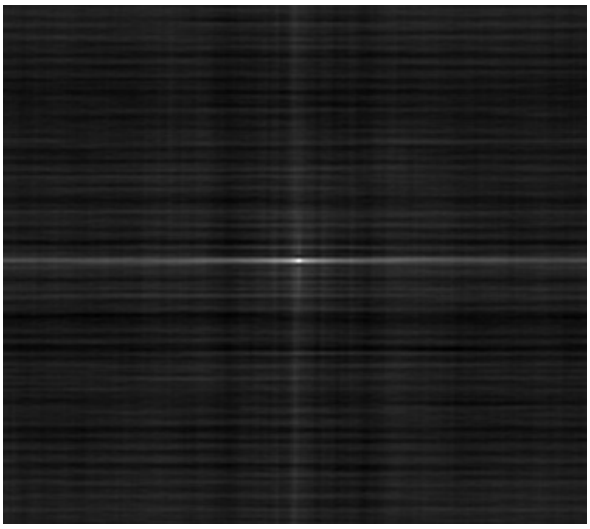


R to B alignment ifft without preprocessing:

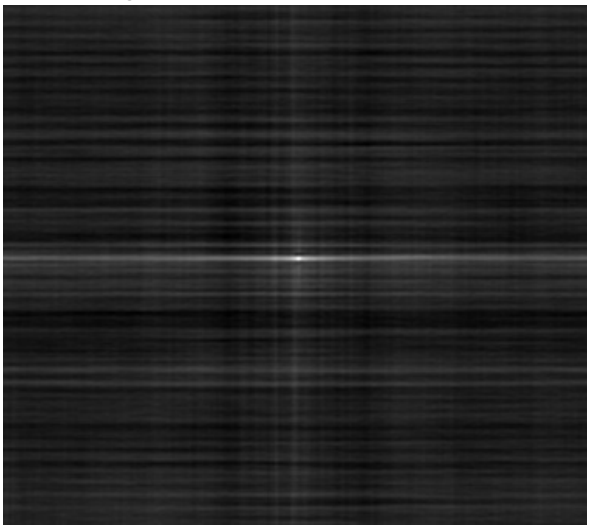


00149v.jpg

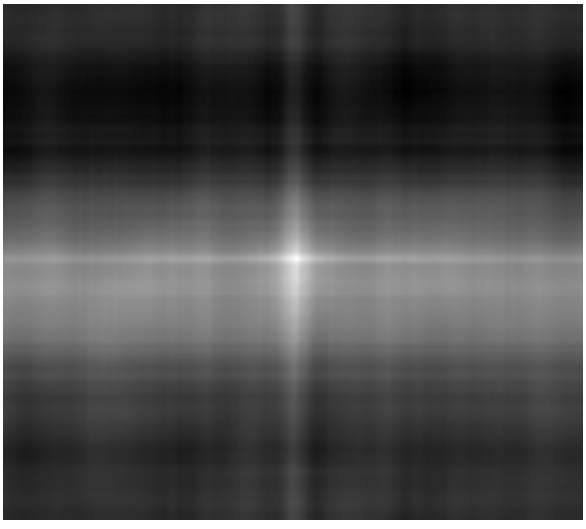
G to B alignment ifft:



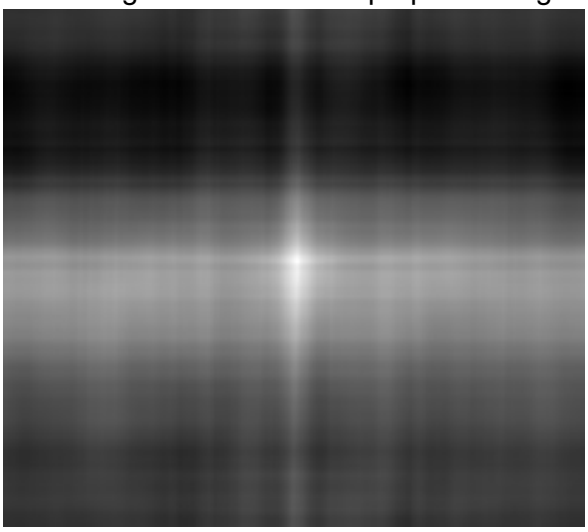
R to B alignment ifft:



G to B alignment ifft without preprocessing:

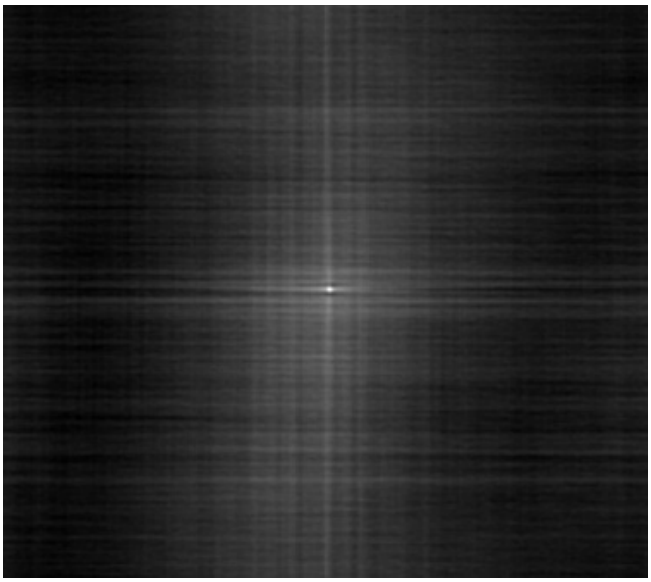


R to B alignment ifft without preprocessing:

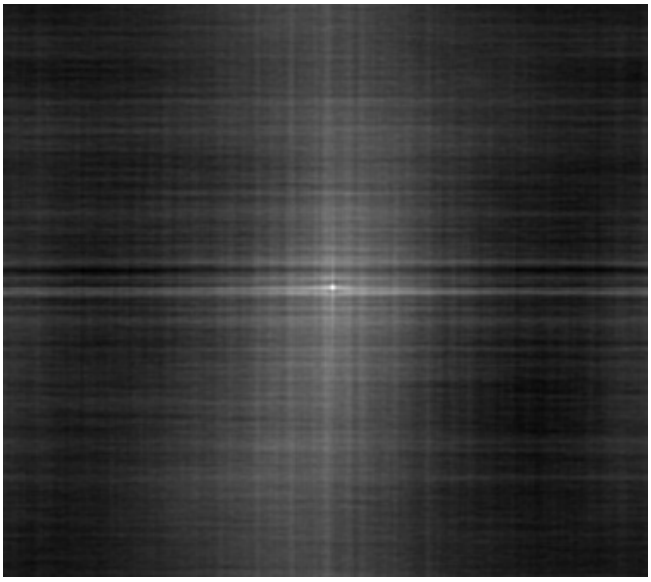


00153v.jpg

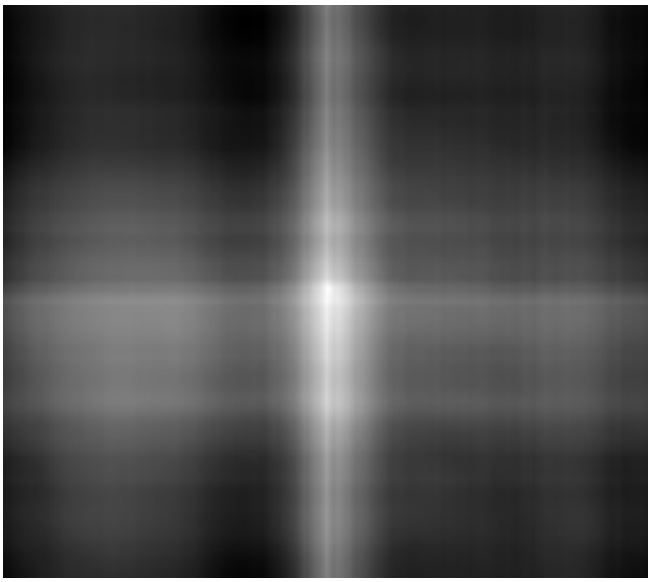
G to B alignment ifft:



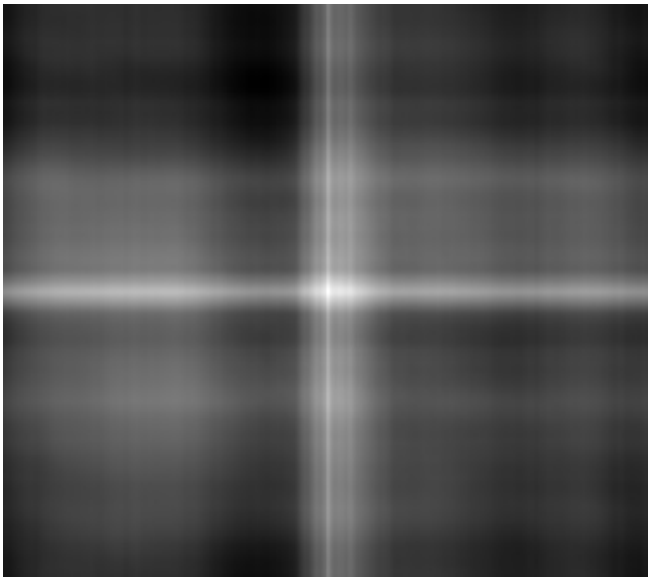
R to B alignment ifft:



G to B alignment ifft without preprocessing:

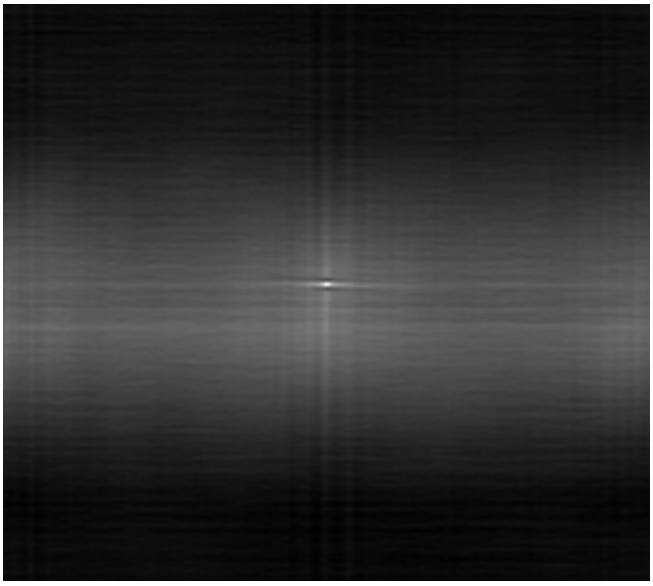


R to B alignment ifft without preprocessing:

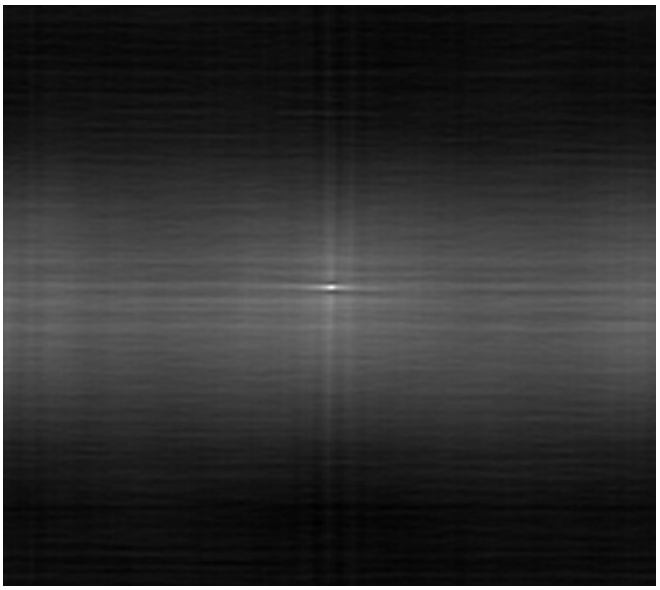


00351v.jpg

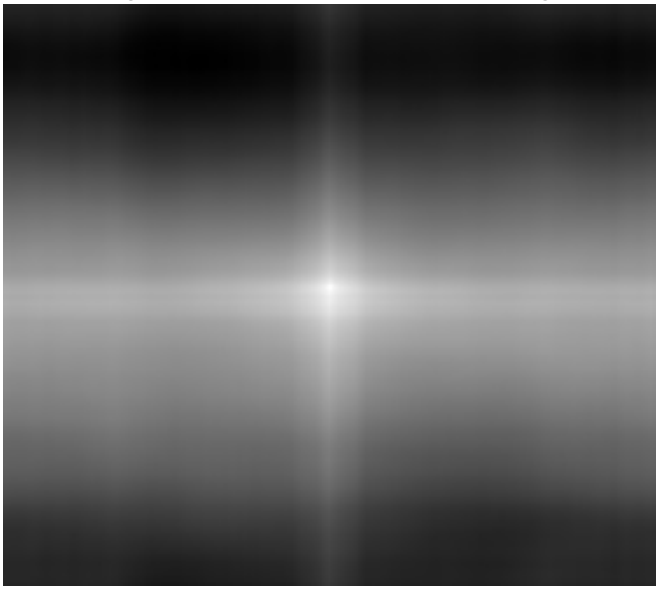
G to B alignment ifft:



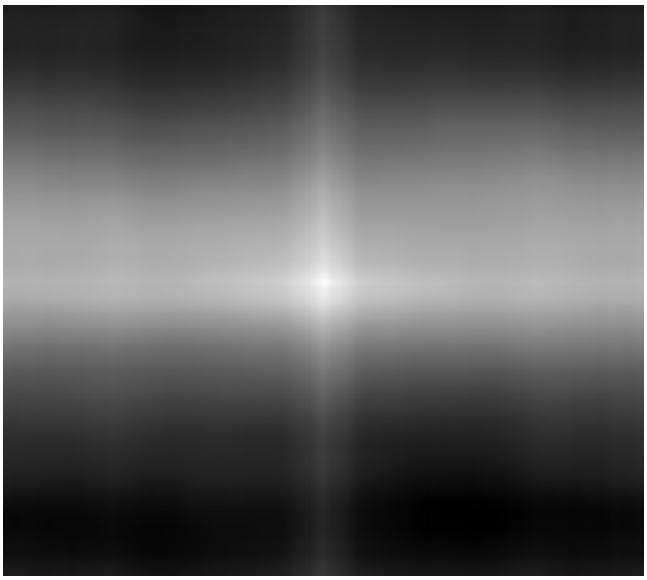
R to B alignment ifft:



G to B alignment ifft without preprocessing:

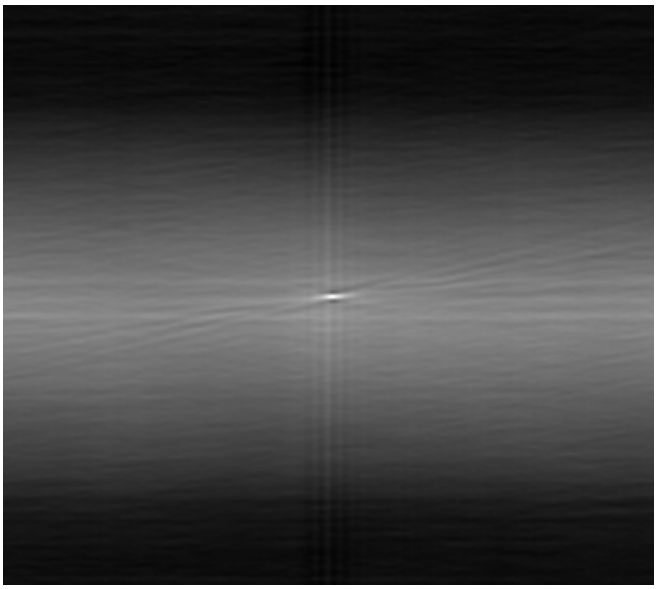


R to B alignment ifft without preprocessing:

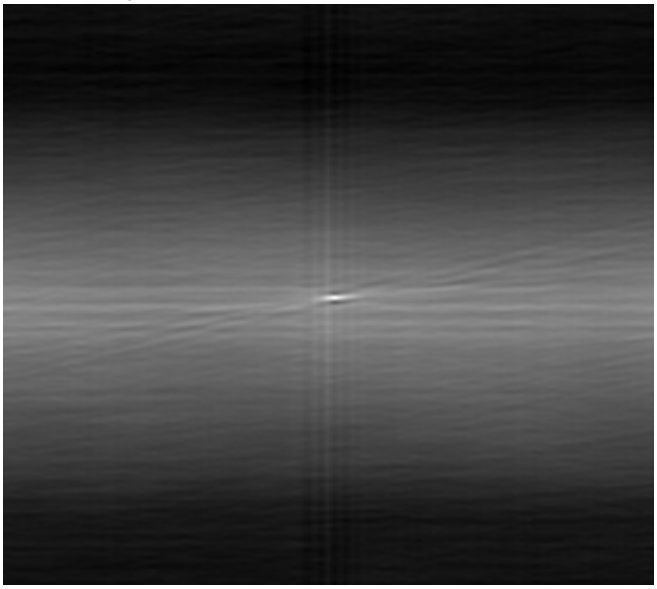


00398v.jpg

G to B alignment ifft:

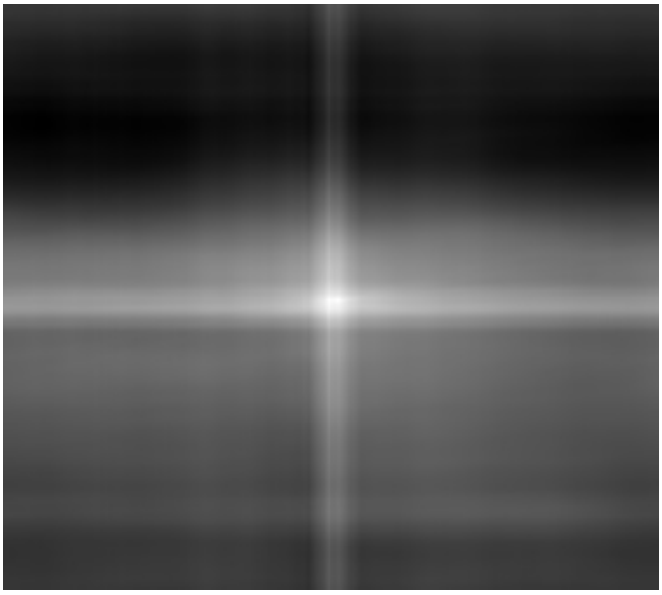


R to B alignment ifft:



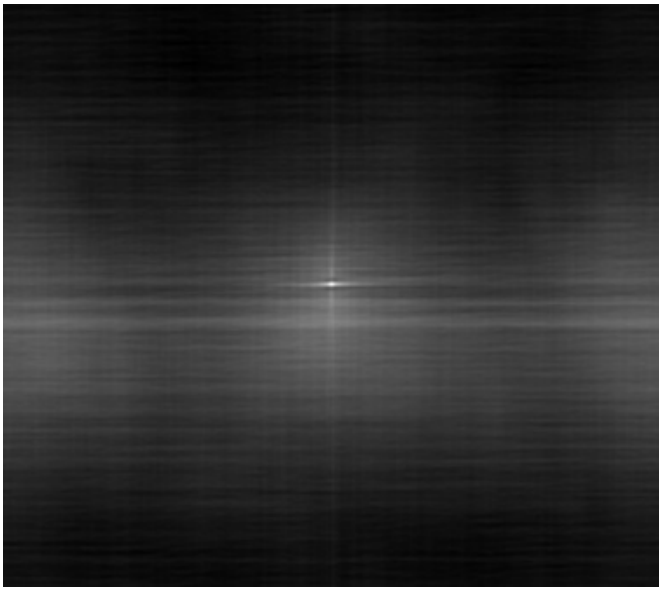
G to B alignment ifft without preprocessing:

R to B alignment ifft without preprocessing:

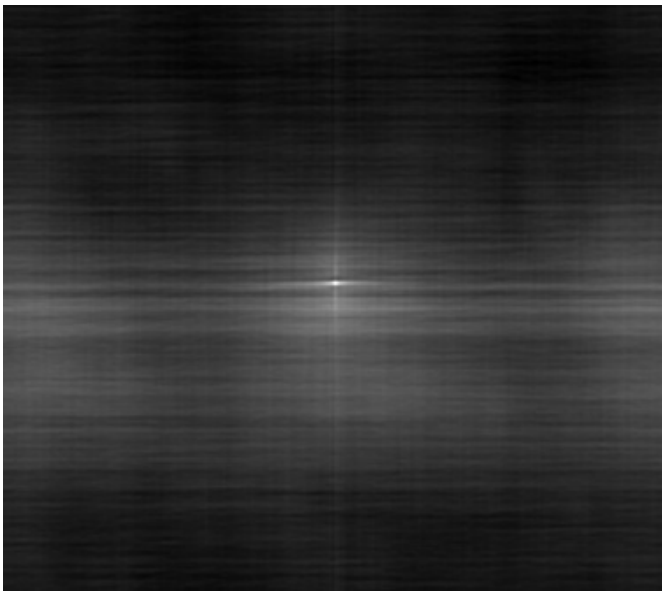


01112v.jpg

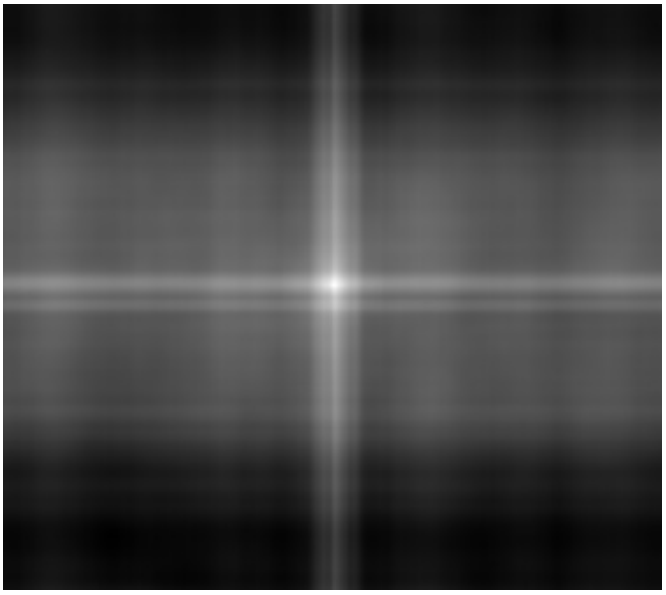
G to B alignment ifft:



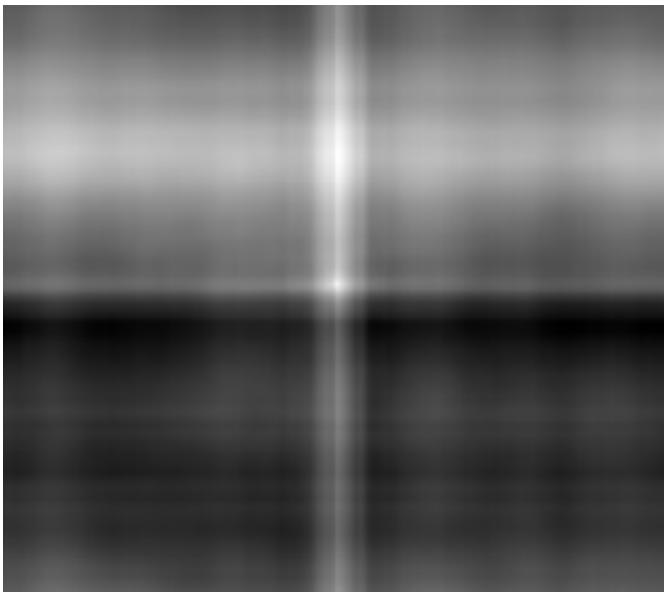
R to B alignment ifft:



G to B alignment ifft without preprocessing:

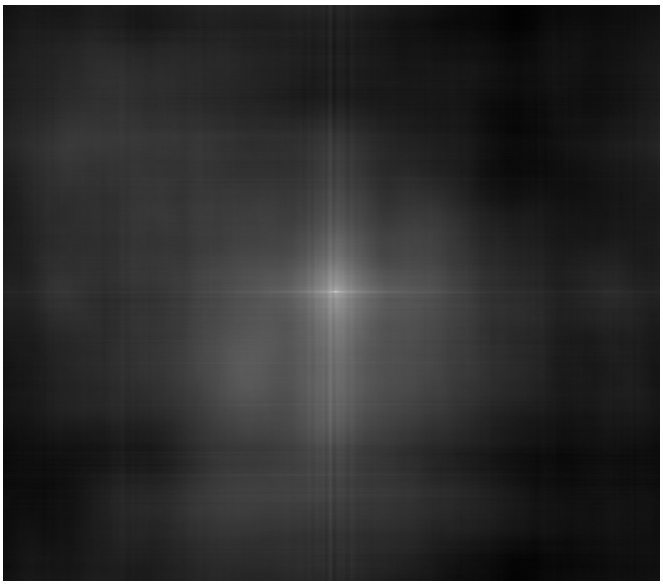


R to B alignment ifft without preprocessing:

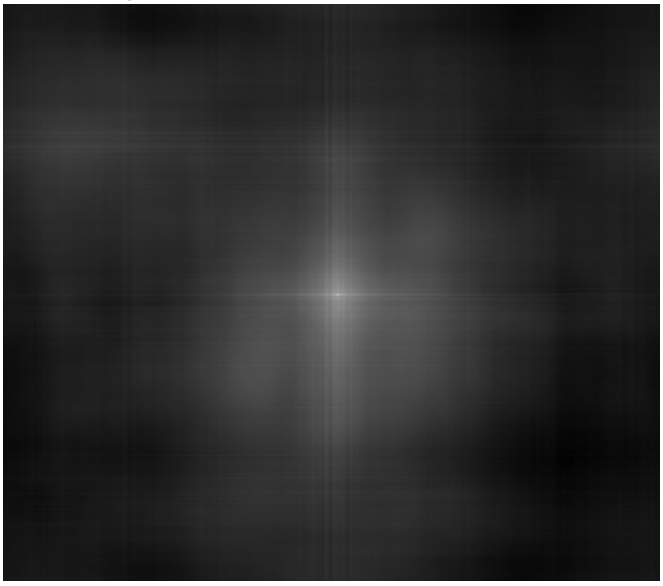


01047u.tif

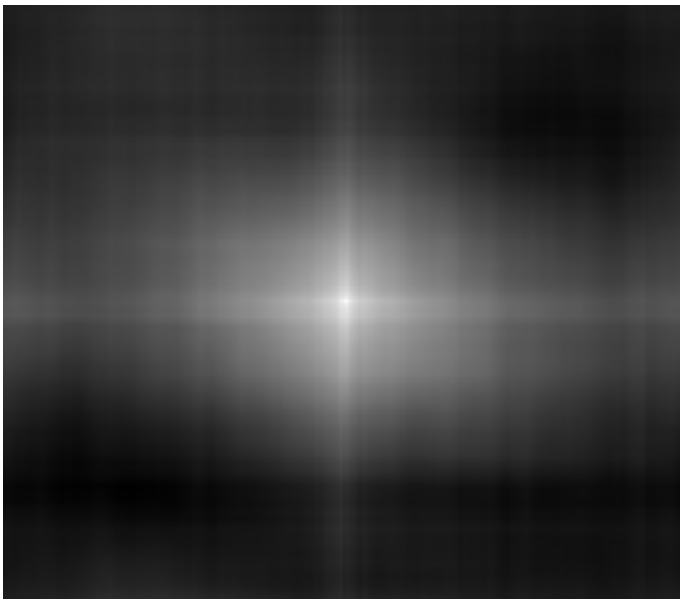
G to B alignment ifft:



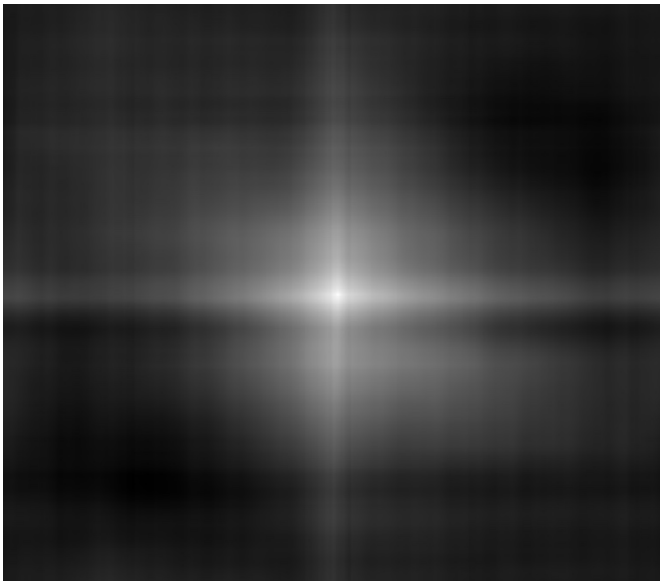
R to B alignment ifft:



G to B alignment ifft without preprocessing:

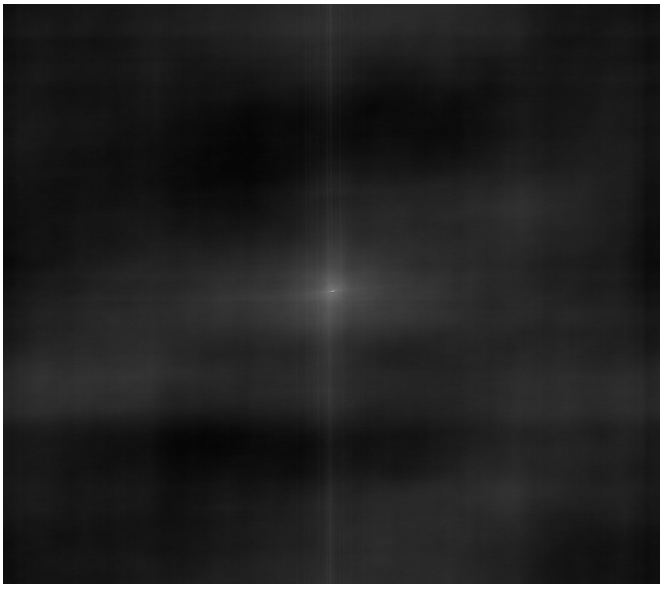


R to B alignment ifft without preprocessing:

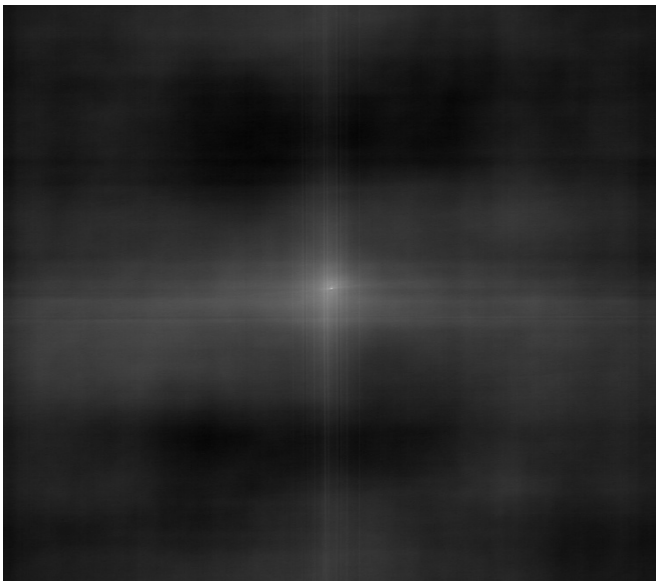


01657u.tif

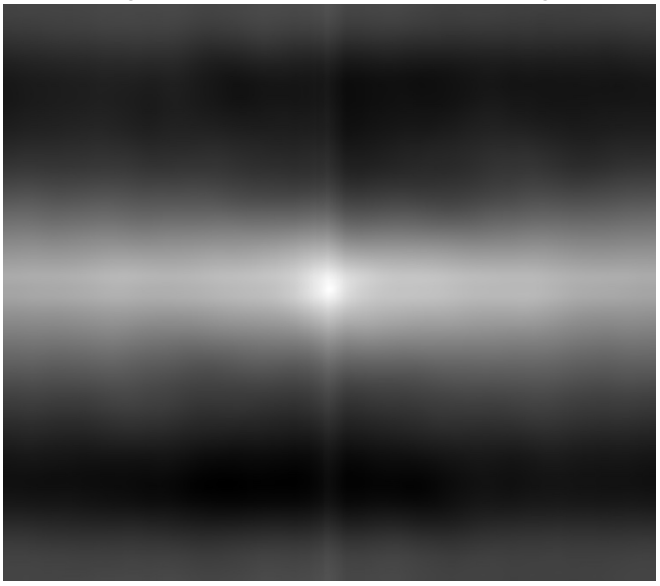
G to B alignment ifft:



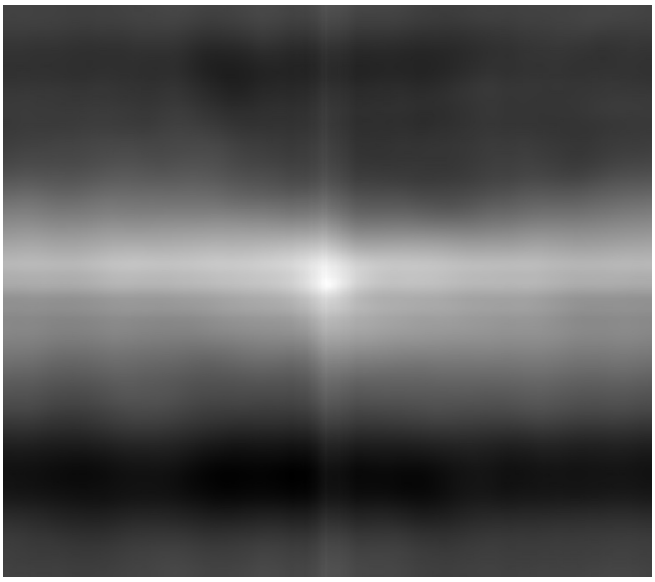
R to B alignment ifft:



G to B alignment ifft without preprocessing:

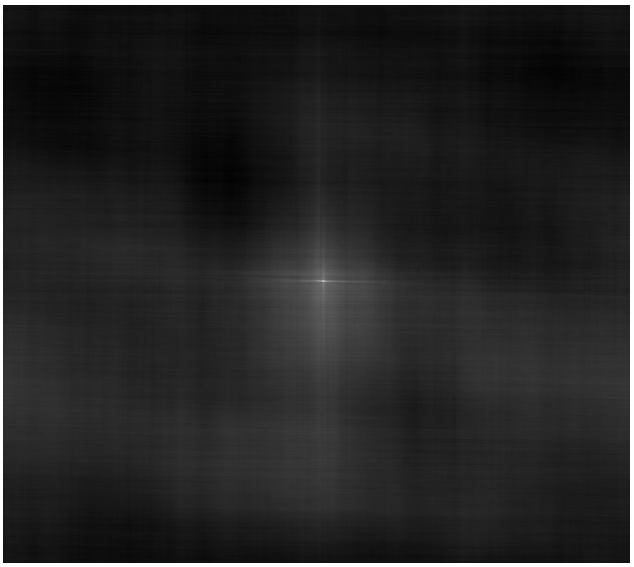


R to B alignment ifft without preprocessing:

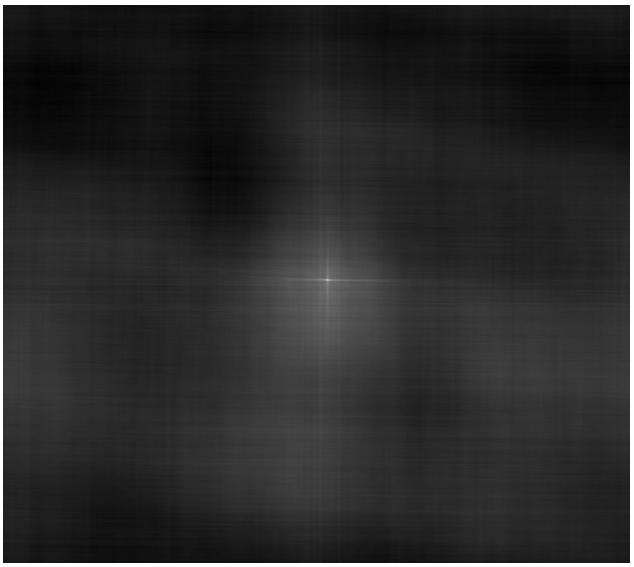


01861a.tif

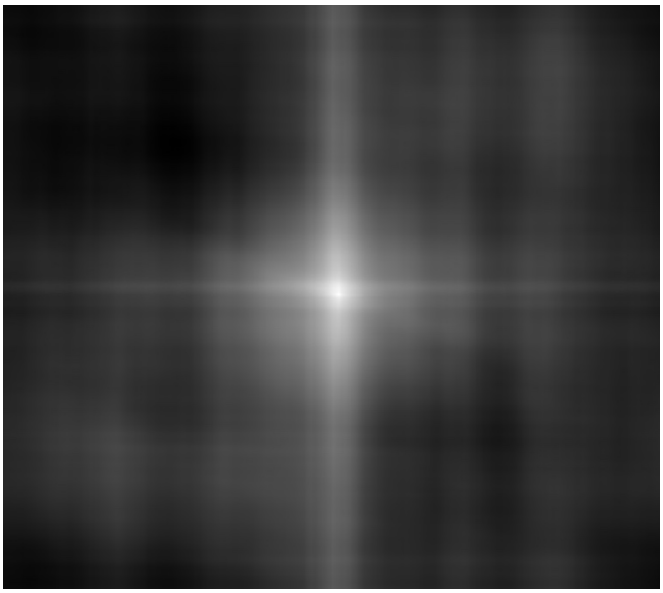
G to B alignment ifft:



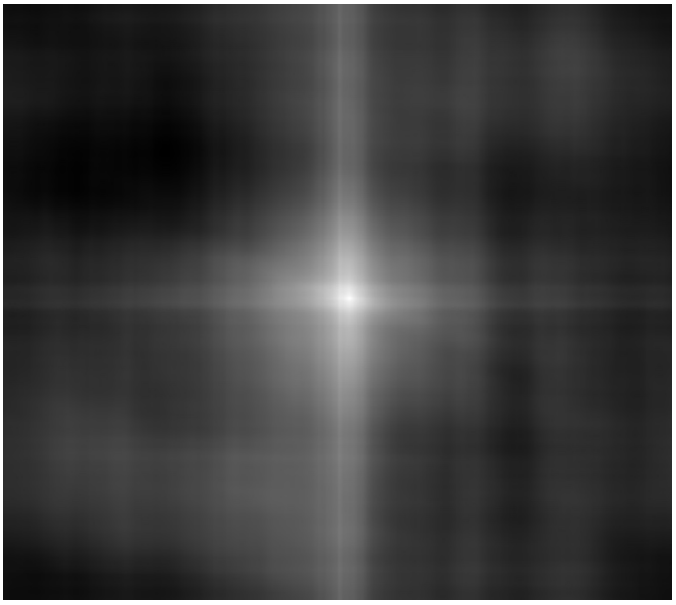
R to B alignment ifft:



G to B alignment ifft without preprocessing:



R to B alignment ifft without preprocessing:



C: Discussion and Runtime Comparison

a. I have cropped the white the borders of the input photos. I also apply Laplacian Gaussian filter to sharpen the photo before doing the alignments. After the preprocessing, I have got better alignment, i.e., compared to those without preprocessing, the result with preprocessing looks clearer.

```
Time for find G channel offset for01047u.tif is 2.8867979049682617
Time for find R channel offset for01047u.tif is 2.74965500831604
Time for find G channel offset for01657u.tif is 2.8197121620178223
Time for find R channel offset for01657u.tif is 2.834439992904663
Time for find G channel offset for01861a.tif is 1.8213908672332764
Time for find R channel offset for01861a.tif is 1.849052906036377
```

b. Results are shown on above. Compared to assignment 1, assignment 2 method has much shorter time to find the offset. In assignment 1, it costs nearly 30 seconds but for assignment 2, it only costs 2-3s each offset.

Part 2 Scale-Space Blob Detection:

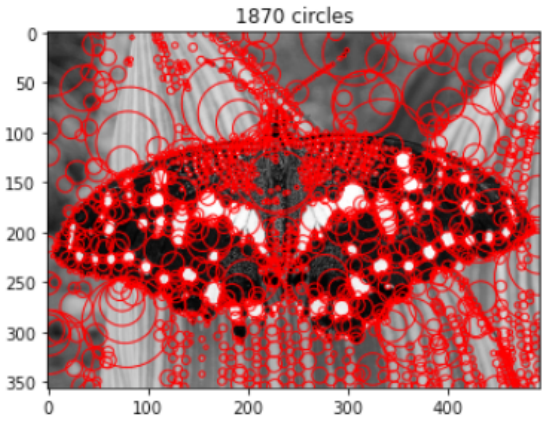
You will provide the following for ***8 different examples*** (4 provided, 4 of your own):

- 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

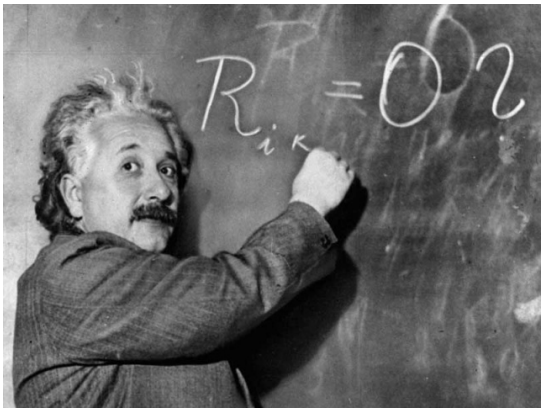
Example 1:

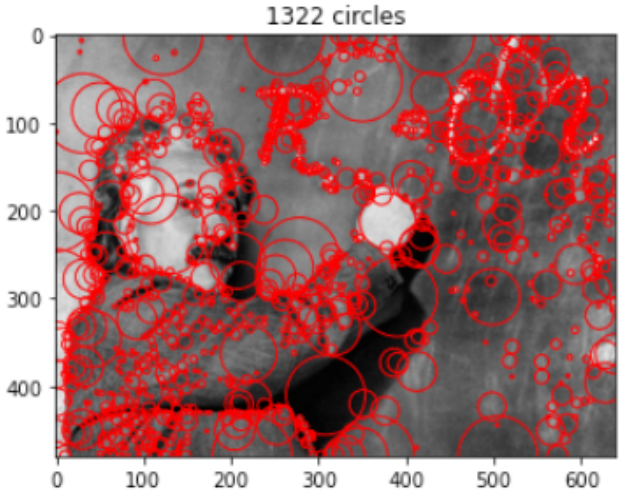


Running time for efficient implementation: 1.007s

Running time for inefficient implementation: 1.110s

Example 2:

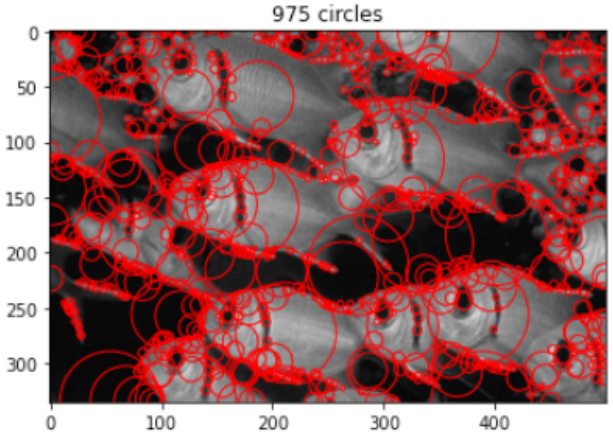




Running time for efficient implementation: 1.697s

Running time for inefficient implementation: 1.847s

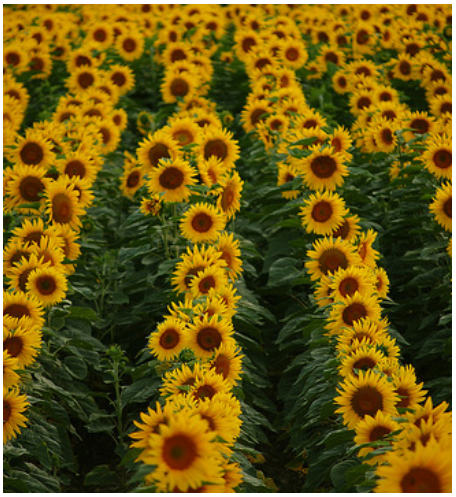
Example 3:

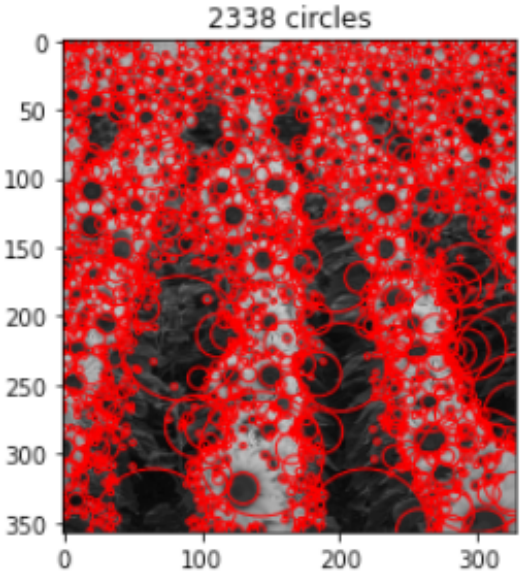


Running time for efficient implementation: 0.932s

Running time for inefficient implementation: 1.017s

Example 4:

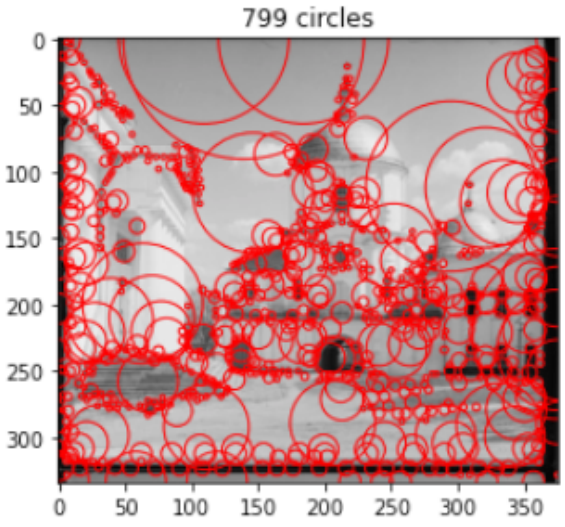




Running time for efficient implementation: 0.653s

Running time for inefficient implementation: 0.715s

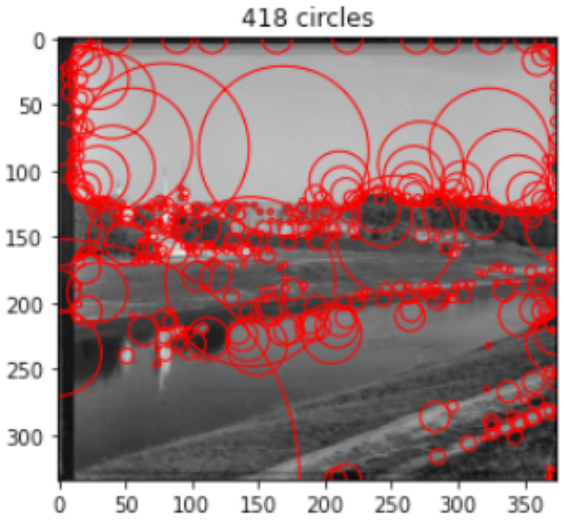
Example 5:



Running time for efficient implementation: 0.874s

Running time for inefficient implementation: 1.674s

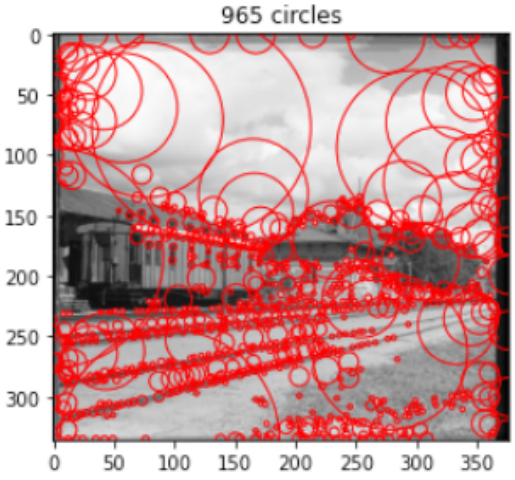
Example 6:



Running time for efficient implementation: 0.901s

Running time for inefficient implementation: 1.702s

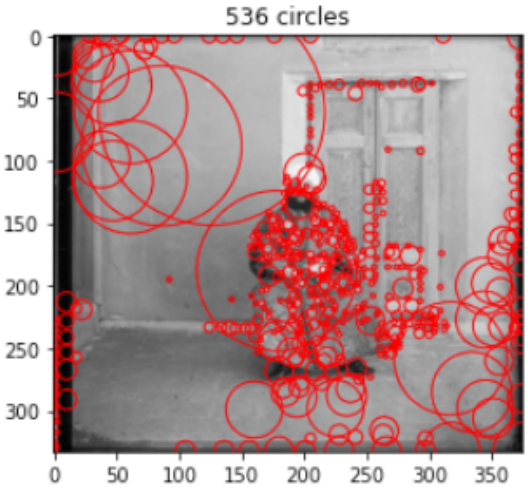
Example 7:



Running time for efficient implementation: 0.874s

Running time for inefficient implementation: 1.703s

Example 8:



Running time for efficient implementation: 0.871s

Running time for inefficient implementation: 1.663s

Discussion:

1. For all implementation, I have chosen K= sqrt (2) and initial sigma= sqrt (2). For the first four examples, I choose iteration times= 10, and for the last four, I choose iteration times= 15. The threshold for NMS is dependence with different photos, ranging from 0.003 to 0.01, but actually they do not have many differences.
2. I have tried different k and sigma in testing, and find that k goes higher, the scale of the circles increasing more rapidly, and have some unreasonable big circles. With sigma goes higher, we will lose many small blobs and the smallest circles become larger.
3. With increasing iteration level, we will get more circles and some very big circles will cover some small circles. Some unreasonable big circles occur as well.
4. The downsampling method gets faster than the increasing filter method, especially in those photos with high resolution rate.
5. The most difficult part for me is writing the 3d NMS part. I set a threshold to remove some noise in our blob detection.