CS543 Assignment 2

Your Name: Emmanuel Gallegos
Your NetId: eg11

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 <C1> as base channel):

Image	R(h,w) offset	G(h,w) offset
00125v.jpg	(13,398)	(7,399)

00149v.jpg	(11,1)	(5,1)
00153v.jpg	(16,3)	(5,2)
00351v.jpg	(15,0)	(5,0)
00398v.jpg	(14,396)	(7,396)
01112v.jpg	(8,2)	(0,1)

High-resolution images (using channel <C1> as base channel):

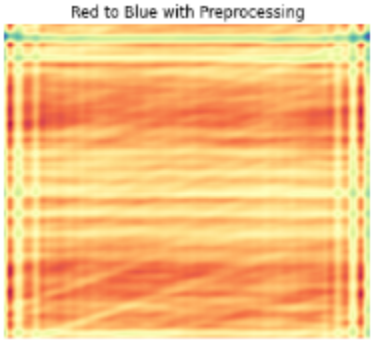
Image	R (h,w) offset	G (h,w) offset
01047u.tif	(74,32)	(26,20)
01657u.tif	(121,10)	(58,7)
01861a.tif	(148,61)	(71,36)

B: Output Visualizations

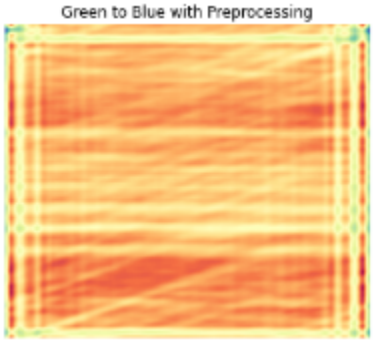
For each image, insert 5 outputs total (aligned image + 4 inverse Fourier transform visualizations) as described above. When you insert these outputs be sure to clearly label the inverse Fourier transform visualizations (e.g. “G to B alignment without preprocessing”).

00125v.jpg

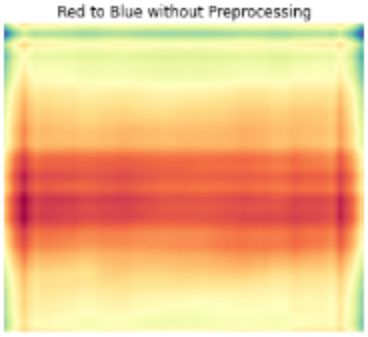
Red to Blue with Preprocessing



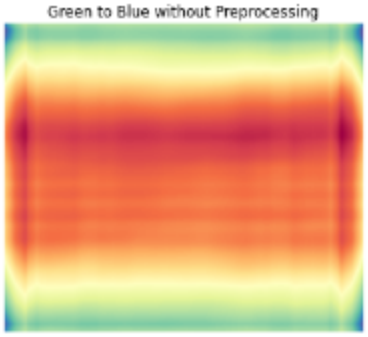
Green to Blue with Preprocessing



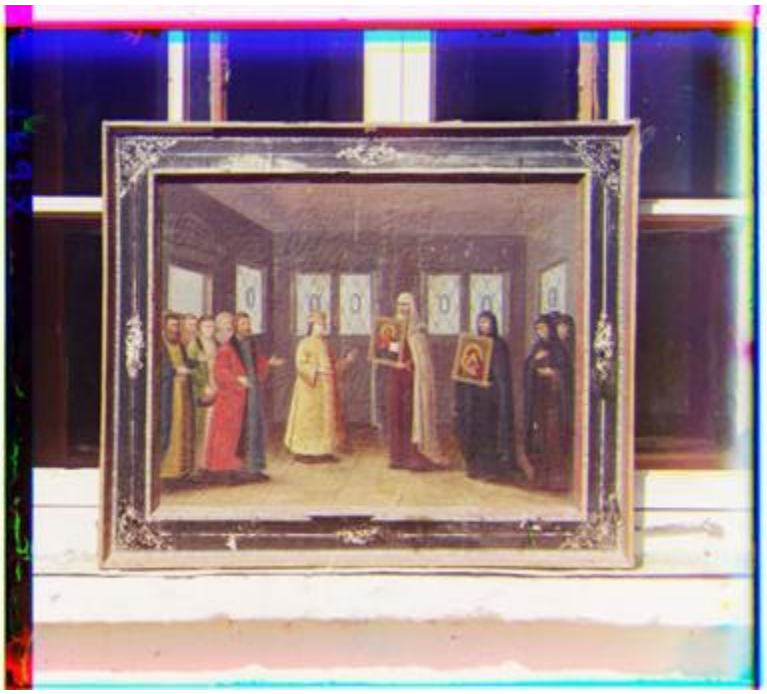
Red to Blue without Preprocessing



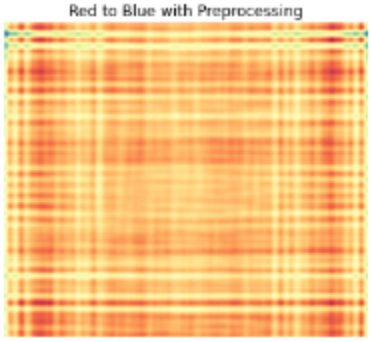
Green to Blue without Preprocessing



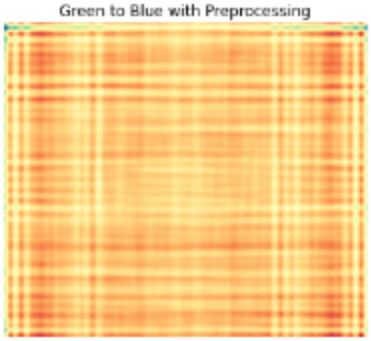
00149v.jpg



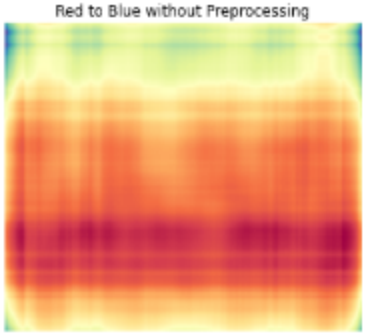
Red to Blue with Preprocessing



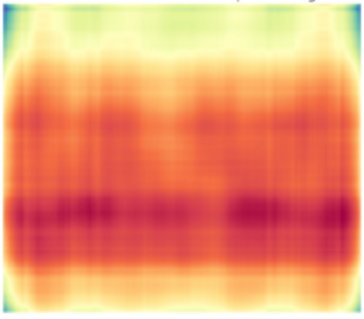
Green to Blue with Preprocessing



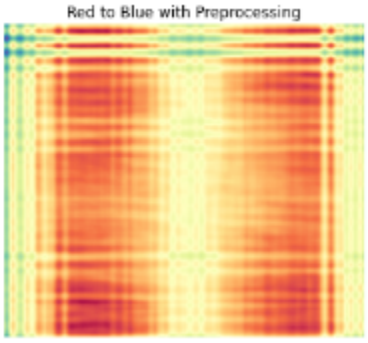
Red to Blue without Preprocessing

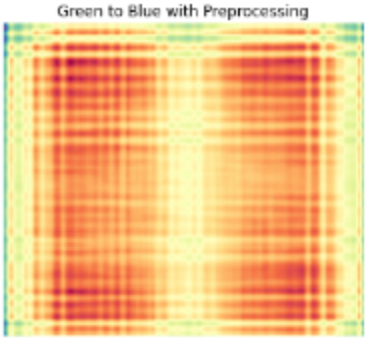


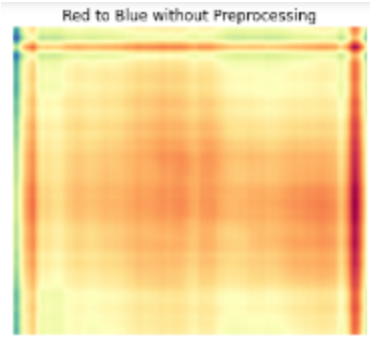
Green to Blue without Preprocessing

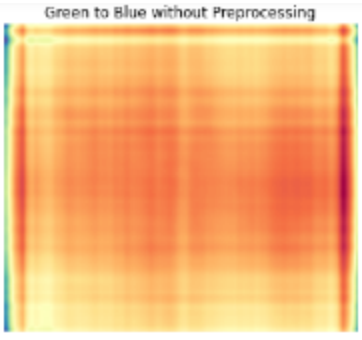


00153v.jpg


Red to Blue with Preprocessing

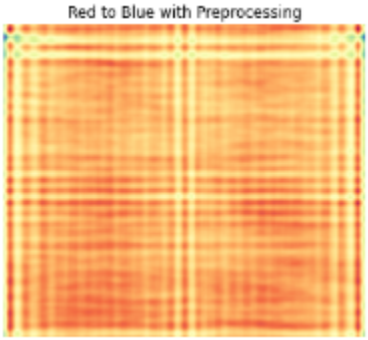

Green to Blue with Preprocessing


Red to Blue without Preprocessing

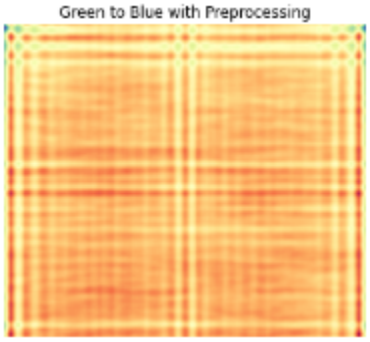

Green to Blue without Preprocessing

00351v.jpg

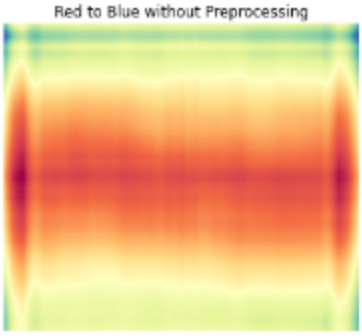
Red to Blue with Preprocessing



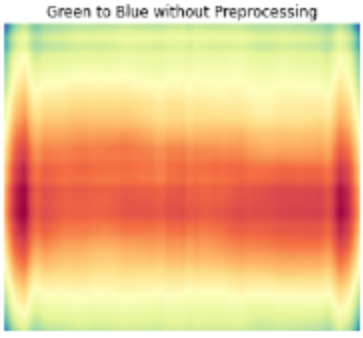
Green to Blue with Preprocessing



Red to Blue without Preprocessing

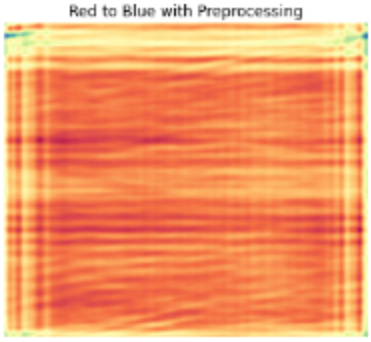


Green to Blue without Preprocessing

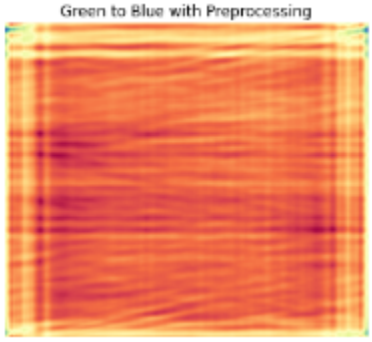


00398v.jpg

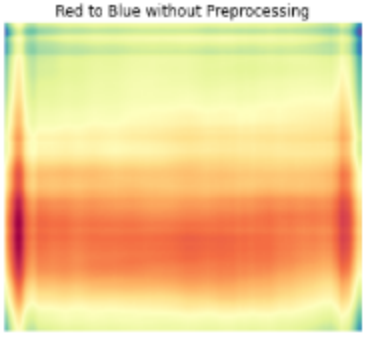
Red to Blue with Preprocessing



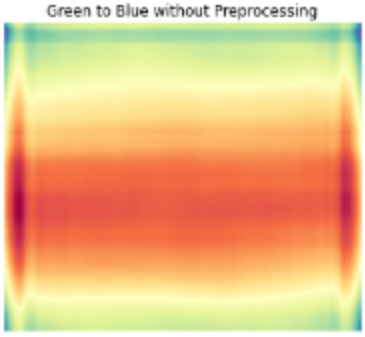
Green to Blue with Preprocessing



Red to Blue without Preprocessing

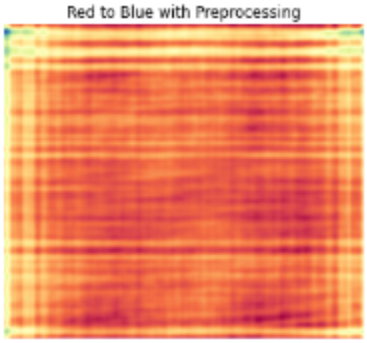


Green to Blue without Preprocessing

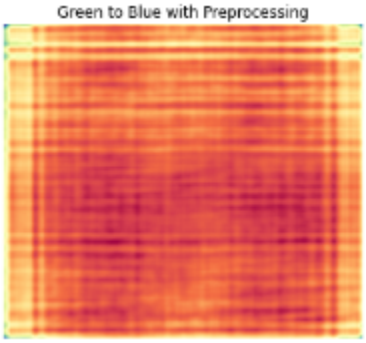


01112v.jpg

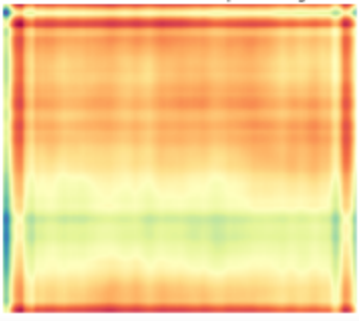
Red to Blue with Preprocessing



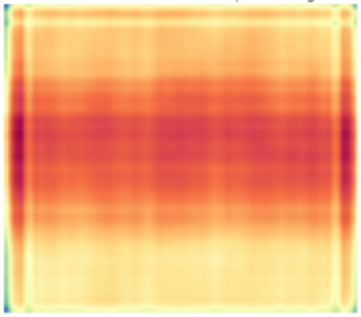
Green to Blue with Preprocessing



Red to Blue without Preprocessing

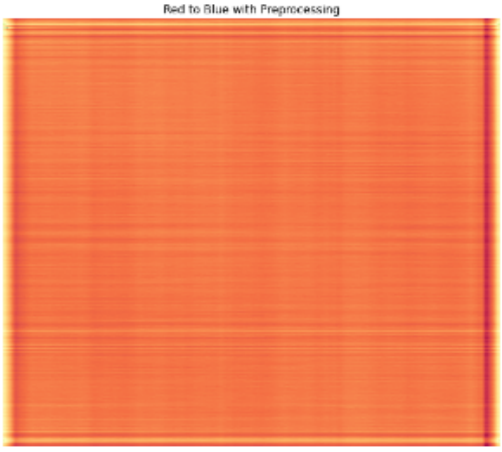


Green to Blue without Preprocessing

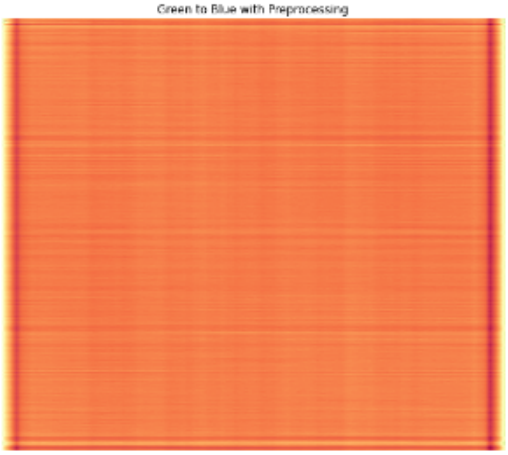


01047u.tif

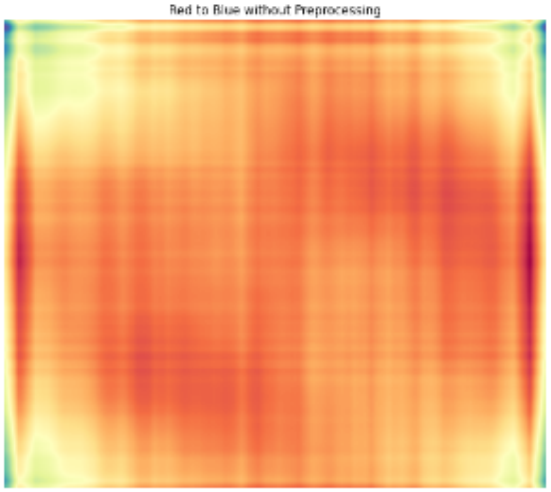
Red to Blue with Preprocessing



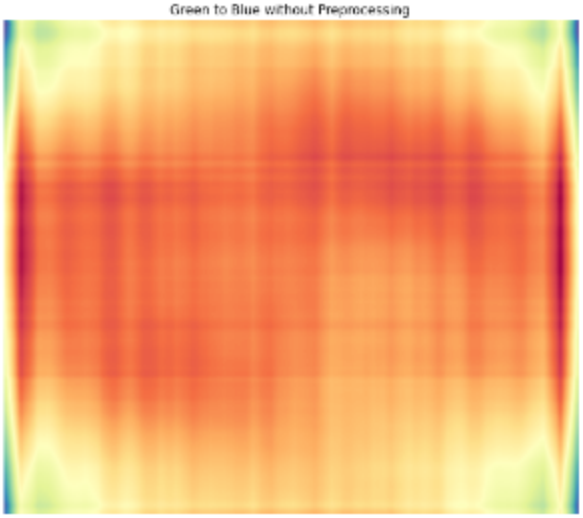
Green to Blue with Preprocessing



Red to Blue without Preprocessing

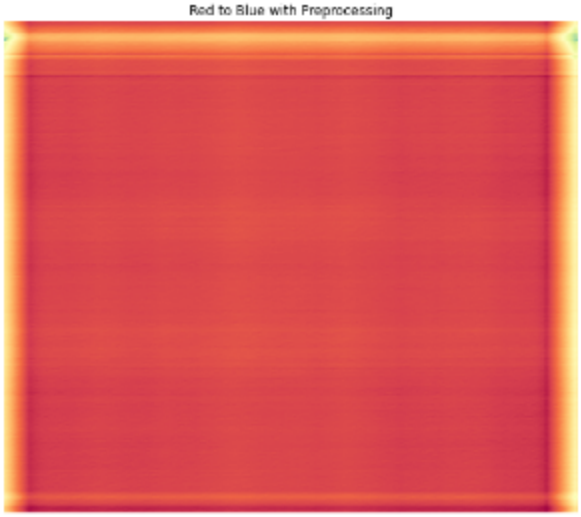


Green to Blue without Preprocessing

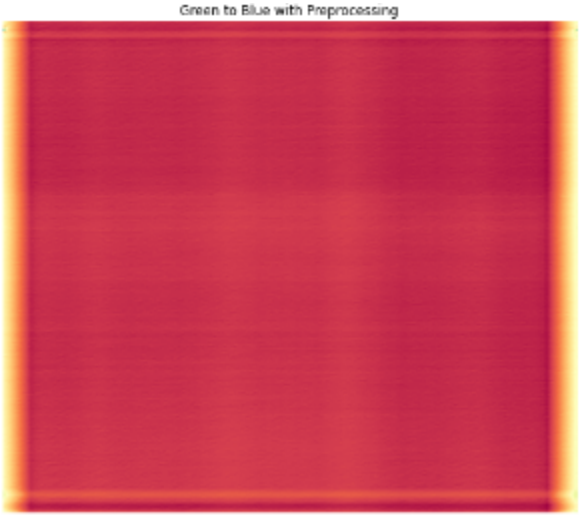


01657u.tif

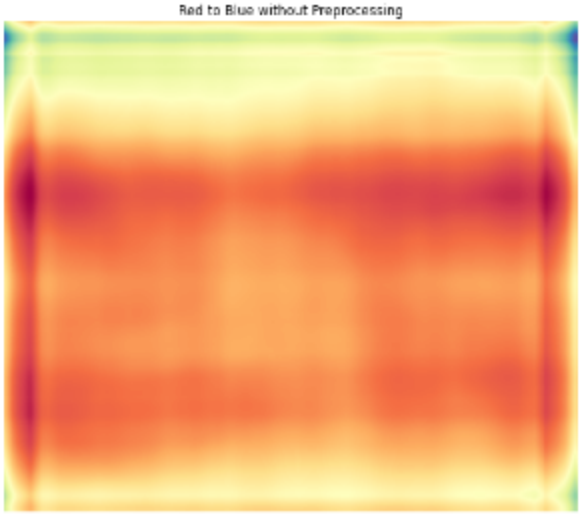
Red to Blue with Preprocessing



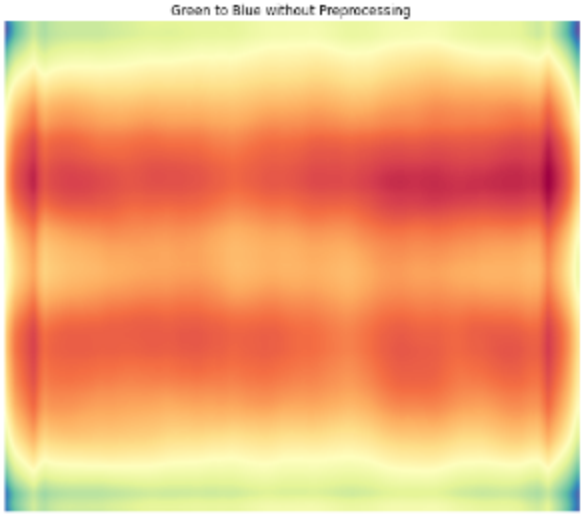
Green to Blue with Preprocessing



Red to Blue without Preprocessing

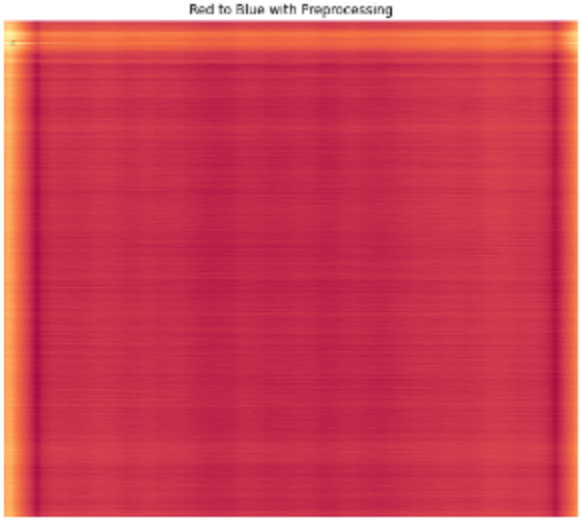


Green to Blue without Preprocessing

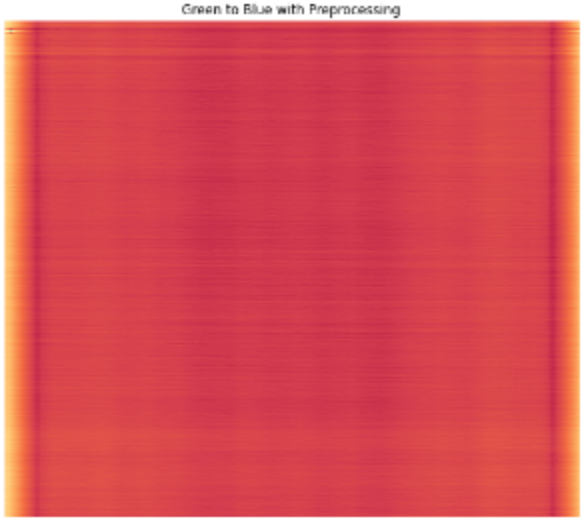


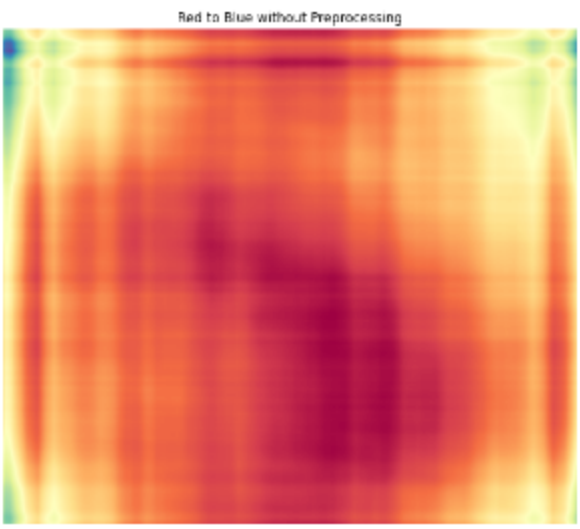
01861a.tif

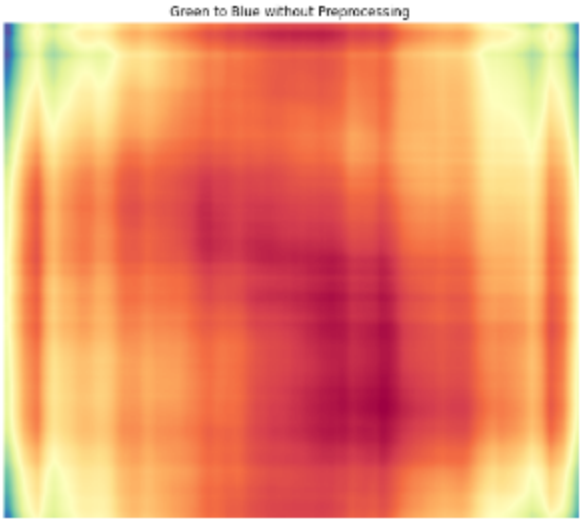
Red to Blue with Preprocessing



Green to Blue with Preprocessing







C: Discussion and Runtime Comparison

The only preprocessing I did on the images was to pad the original image with empty rows along the bottom to make its rows a multiple of 3, and then applied ndimage.gaussian_laplace with a sigma value of 3 to all color channels before doing Fourier analysis.

In assignment one, my average runtime for the multiscale alignment for the basic method was about 67 seconds per photo, which also included cropping the images significantly before performing alignment to decrease the image sizes. After implementing the image pyramid, the running time was an average of 6.265 seconds per photo, or about 13 times faster than my naive implementation. With Fourier analysis to perform the alignment, the average time to align a hi-res image was 7.873 seconds. This is actually a bit slower than my image-pyramid alignment method, though both produced equally excellent alignments.

Part 2 Scale-Space Blob Detection:

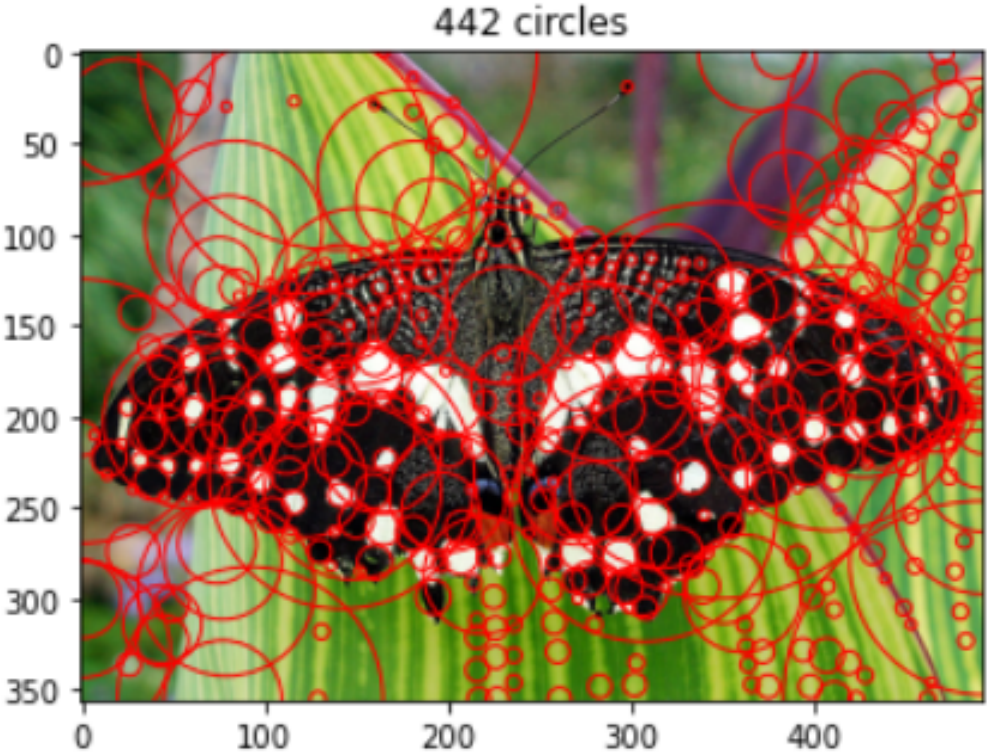
You will provide the following for ***8 different examples*** (4 provided, 4 of your own):

- 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:

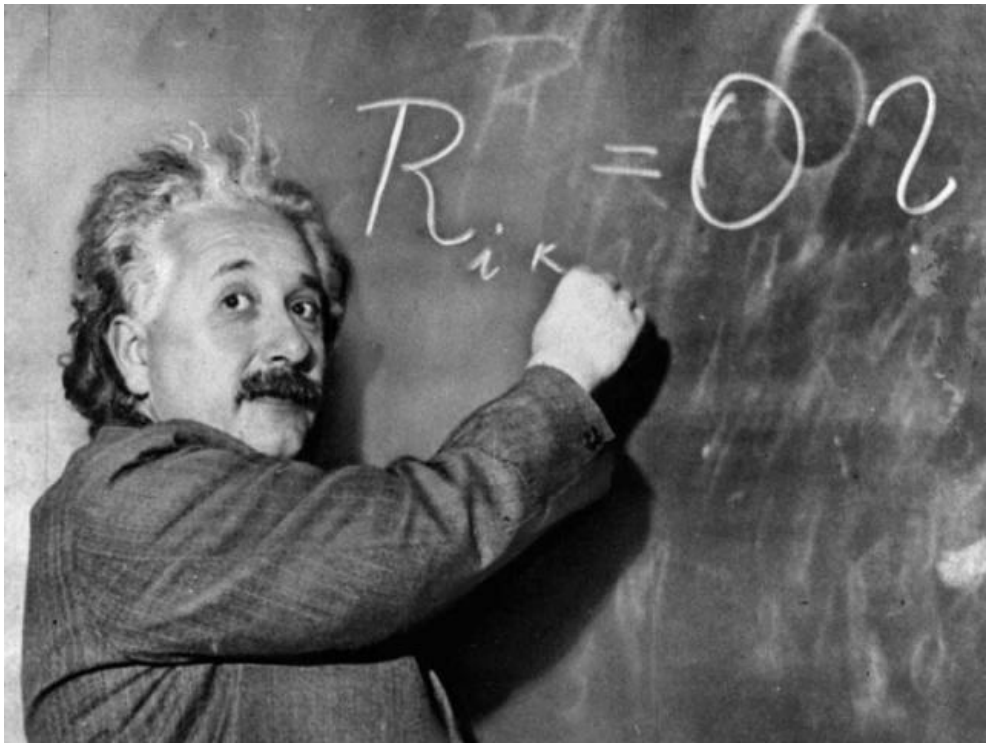
- Explanation of any "interesting" implementation choices that you made.
- Discussion of optimal parameter values or ones you have tried

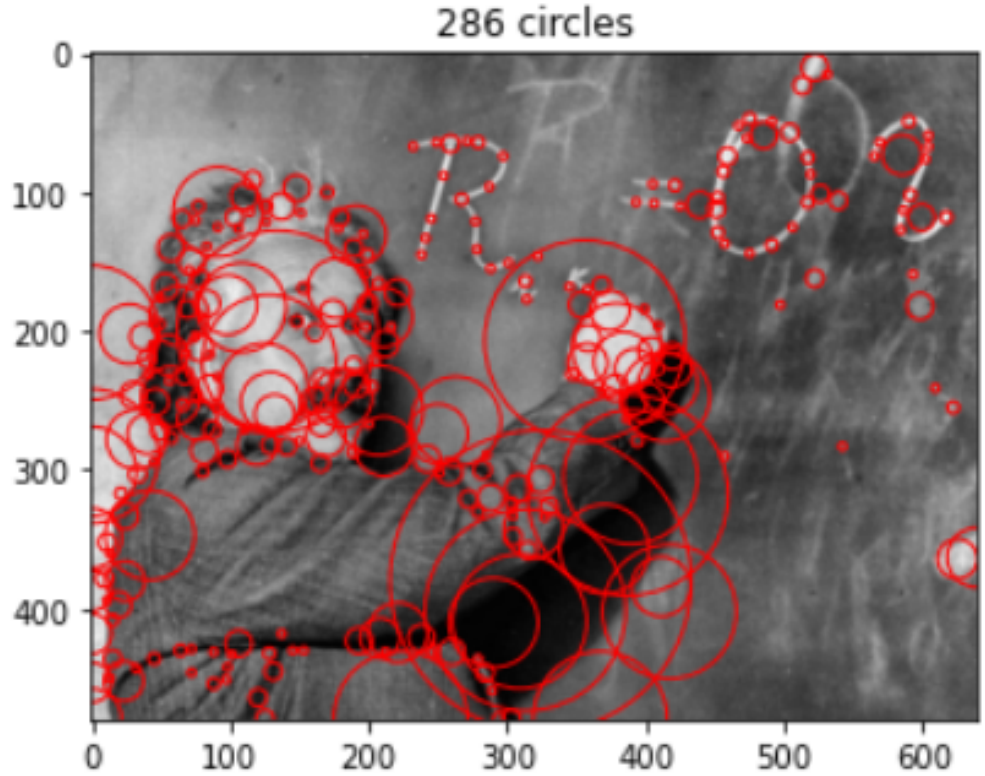
Example 1:



Efficient Run Time: 19.32s
Inefficient Run Time: 19.96s

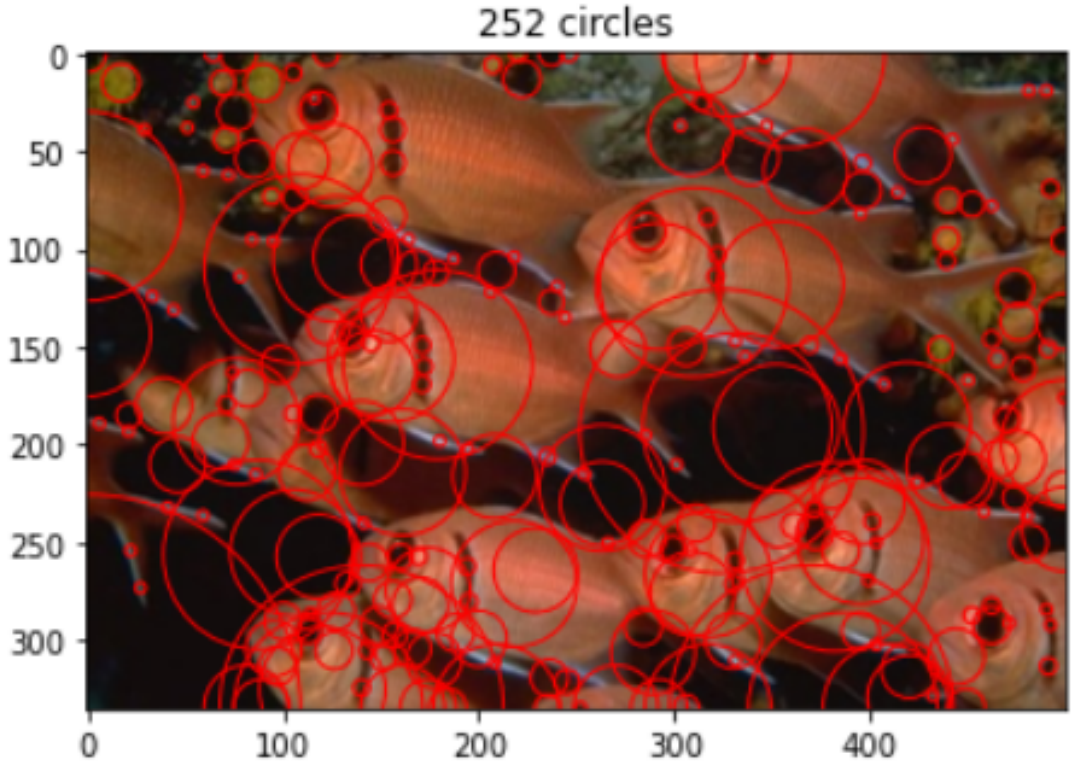
Example 2:





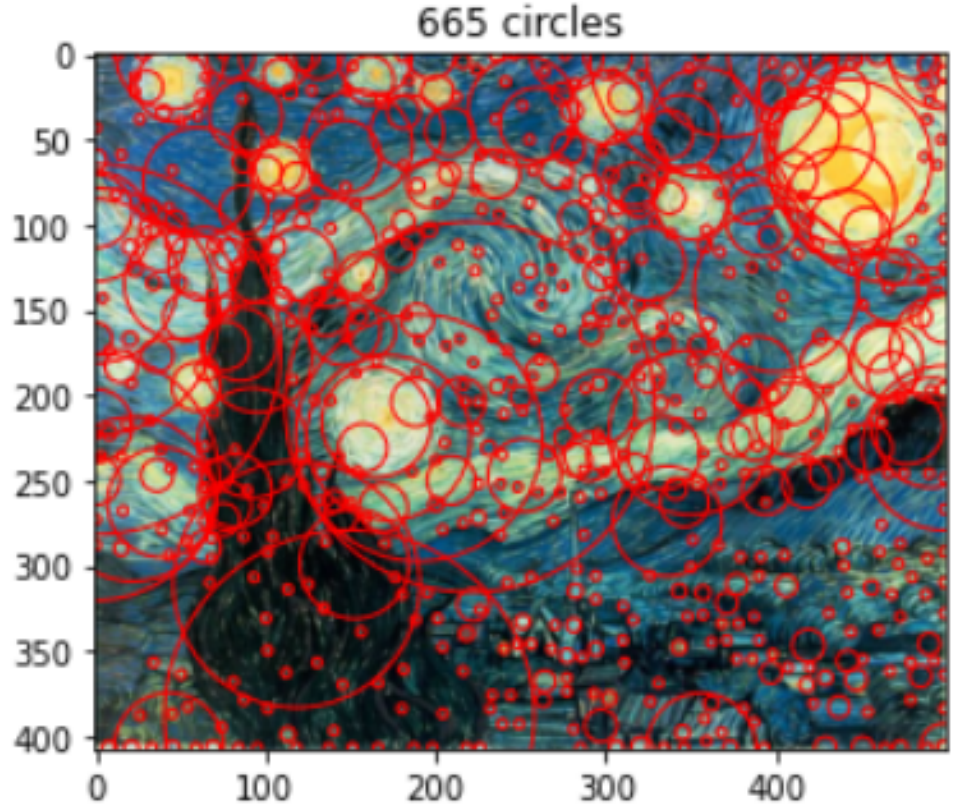
Efficient Run Time: 37.43s
Inefficient Run Time: 34.75s

Example 3:



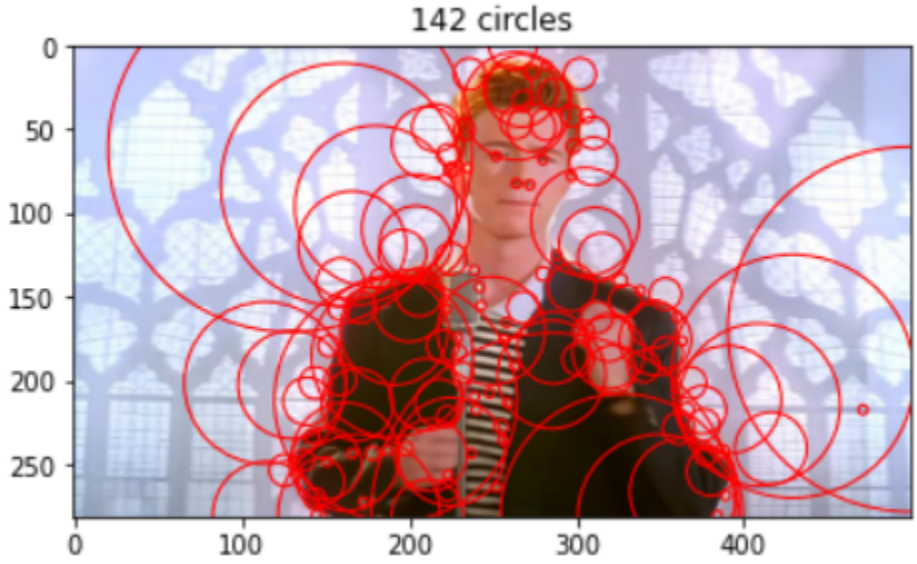
Efficient Run Time: 18.66s
Inefficient Run Time: 19.22s

Example 4:



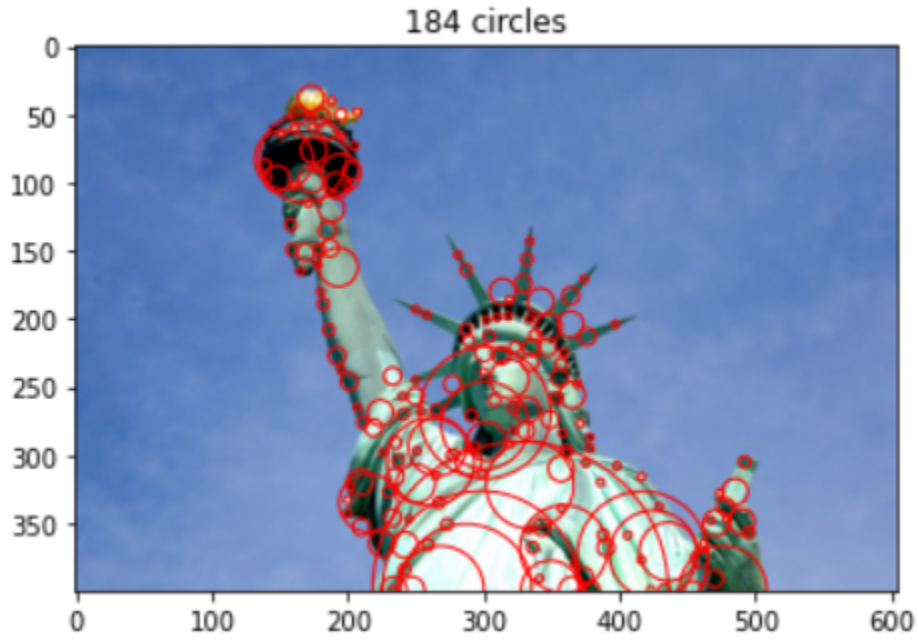
Efficient Run Time: 22.45s
Inefficient Run Time: 22.93

Example 5:



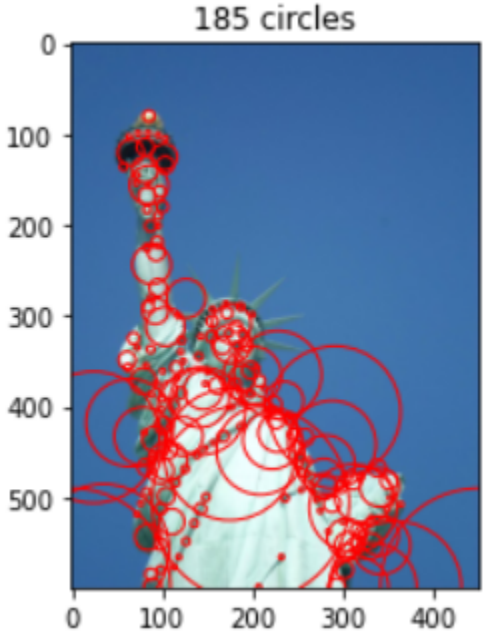
Efficient Run Time: 17.15s
Inefficient Run Time: 15.81s

Example 6:



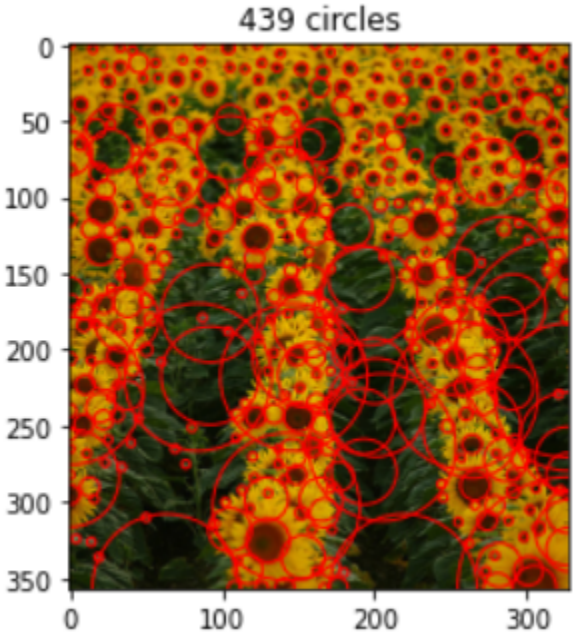
Efficient Run Time: 26.96s
Inefficient Run Time: 27.24s

Example 7:



Efficient Run Time: 30.25s
Inefficient Run Time: 30.69s

Example 8:



Efficient Run Time: 14.18s
Inefficient Run Time: 13.03s

Discussion:

For both methods in part 2, I followed basically the same steps and parameters. I used an initial scale of 2, a scaling factor k of 1.5, 10 scaling levels, a 2d NMS filter size of 15x15, and 3d NMS filter size of 15x15x5. All of these were discovered after a LOT of tinkering (hours worth). I tried scale factors between 1.2 and 2, between 9 and 15 scaling levels, filter sizes from (3x3) to (15x15), and both methods of NMS filtering (rank filter and generic filter). For the NMS filter, I ultimately used the generic filter method, and made my own custom filter method that simply checked if the pixel in question (the one in the middle) was the maximum of all the pixels in the neighborhood, and left the value untouched if it was, otherwise setting it to 0. This ensured that only true maximums in that pixel range were left ‘alive’ after the filtering.

I also used 2 different thresholds for thresholding the response function to choose the final number of circles for the 2 methods respectively. If I tried using the same threshold for both methods, I was left with drastically fewer circles in the second method. I suspect this was necessary because while the two methods functioned nearly identically, the downsampling I had to do in order to properly perform the second technique ended up with some ‘lost’ information that had to be reconstructed when the image was upscaled.