Computer Vision

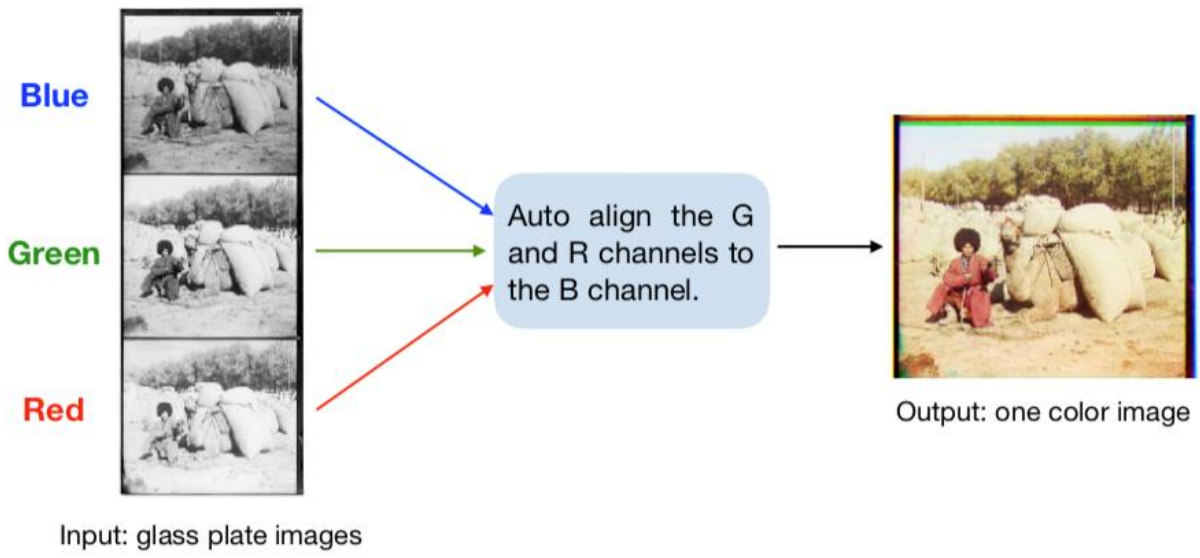
Group 12 | 0786031 廖俊凱 0516044 陳思妤 0856733 黃明翰

Introduction:

Hybrid images demonstrate how certain frequencies dominate at certain distances. Most notably, high frequencies dominate when the object is close to you, while lower frequencies dominate when it is further away. By applying a Gaussian blur to an image in order to produce a low-pass or high-pass filtered image, we can demonstrate the effect of hybrid image.

Image pyramid is a type of multi-scale signal representation where an image is subject to repeated smoothing and subsampling. Here we implement two pyramids: Gaussian pyramid and Laplacian pyramid. In a Gaussian pyramid, subsequent images are weighted down using a Gaussian blur and scaled down. A Laplacian pyramid is very similar to a Gaussian pyramid but saves the difference image of the blurred versions between each levels.

Sergei Mikhailovich Prokudin-Gorskii (1863-1944) conceived a method for recording color images before color photography. His method was simple, but practical. He took photos of various subjects across the Russian Empire. Each subject was photographed three times. Once with a Blue filter, once with a Green filter and finally once with a Red filter. These three photographs made up a set that was placed onto plate glass. His intention was to project each image on top of each other using Blue, Green and Red light to merge them into a color photograph. For this project, each input image consists of three separate images that were taken with different filters. The three separate images correspond to different color channels. For example, the top image corresponds to blue, the middle to green, and the bottom to red. The goal of this project is to automatically align the three images for the various plates and automatically produce a color image from the digitized Prokudin-Gorskii glass plate images with as few visual artifacts as possible.



Implementation Procedure:

- ## Hybrid Image

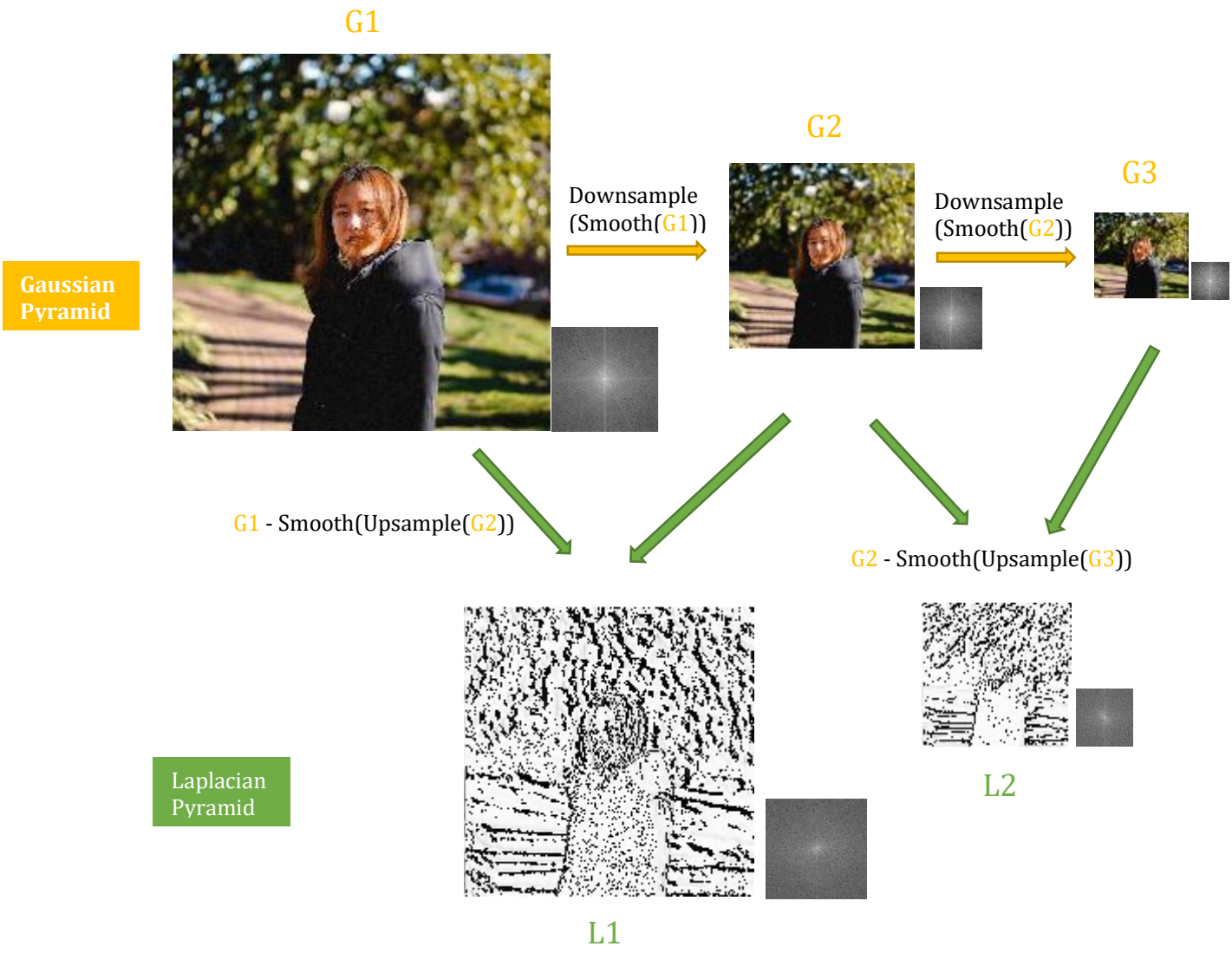
1. Load the image and convert it to grayscale image. To combine two images together, they need to have the same shape, so we reshape the smaller image to the size of the other one.
2. Create Gaussian filter kernel with the size of the certain image. Perform this formula to each and every pixel:

$$G(x, y) = e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

   By doing so, we can obtain a low-pass Gaussian filter. High-pass filter can be obtained by 1 minus low-pass kernel.

3. Compute Fourier transformation of the input image with function fft2(), and then perform fftshift() to shift the zero-frequency component to the center of the spectrum. And then multiply the output by the Gaussian filter we created earlier. After all this operations, compute the inverse Fourier transform by function ifft2() and shift the result back with function ifftshift()
4. We can obtain the final output image by adding up the pairing images and extract the real part
5. Perform the same procedure of step 3 and 4 with ideal lowpass / highpass filter.

- **Image Pyramid**



1. Use Gaussian filter smoothing G_i
2. Sub-sample (Downsample) $\text{smooth}(G_i)$ to get G_{i+1}.
3. Upsample G_{i+1} with 'nearest' interpolation, then smooth the result with Gaussian filter.

$$L_i = (G_i - \text{smooth}(\text{Upsampling}(G_{i+1}))$$

4. Get the magnitude spectrum of G_{i+1} and L_i.
5. Repeat 1~4 to get the Gaussian pyramid and Laplacian pyramid.

- # Colorizing the Russian Empire

A. Step 1: Dividing glass plate images into three images.

Using function PIL.Image.crop() can approach the goal easily. The function crop the image to get the region we want, and output the region as an image type. By this function, we simply divide the glass plate image into three pieces with different color channels which imply blue channel, green channel, and red channel respectively.

```
# crop(left, upper, right, lower)
# devide image into the three b g r channels
b = img.crop((0,                      0, oriWidth,              maxHeight))
g = img.crop((0,              maxHeight, oriWidth, oriHeight - maxHeight))
r = img.crop((0, oriHeight - maxHeight, oriWidth,              oriHeight))
```

B. Step 2: Defining the statement of matching.

A possible way to align two similar images is by exhaustive search for the best (x, y) displacement vector over a possible window of displacement. This approach assumes that an image is a translation of the other image, i.e. there is no rotation or any change in size. Here, we are looking to minimize the Mean Squared Error (MSE) between two channels. We determine whether two images are similar by mean squared error. For large mean squared error, it seems that these two images do not match perfectly to each other. For small mean squared error, it means these two images fit well to each other. The algorithm and code are described below.

$$MSE\ (imageA, image2) = \frac{\sum_y \sum_x (image1(x,y) - image2(x,y))^2}{image(x) \times image(y)}$$

```
def mse(imageA, imageB):
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])
    return err
```

To find the minimum the mean squared error, we can shift the image C gradually and evaluate the mean squared error between image A and image C (Fig.2). Eventually, we will get the best shift between image A and image C (Fig.3).

Here is what we done for matching. We first crop the image B into image C with a smaller size of image B, which means we slice the boundary of image B, and store the magnitude of cut (Fig.1). After finding the best shift between image A and image C by gradually evaluating mean squared error, we minus the magnitude of cut to get the best shift between image A and image B (Fig.4).

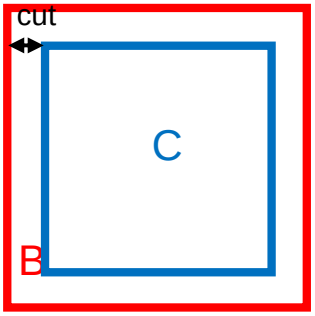


Fig.1

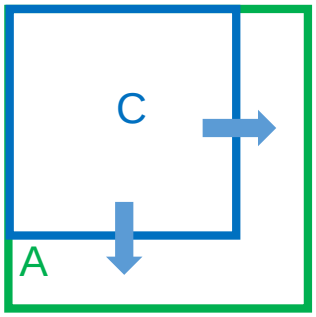


Fig.2

Fig.1

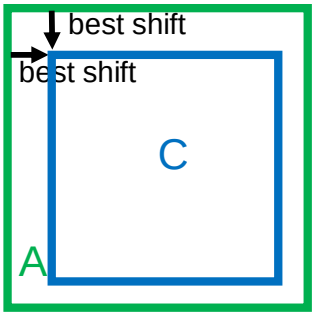


Fig.3

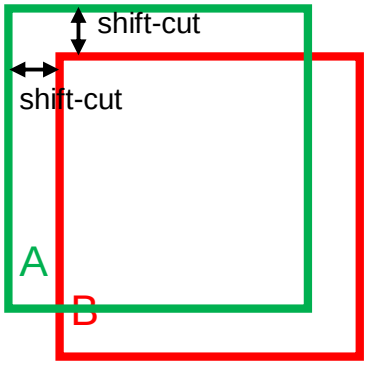


Fig.4

```
def findShift(basicImg, matchingImg):
    basicN, basicM = basicImg.shape   # N:height M:width
    matchN, matchM = matchingImg.shape # N:height M:width

    xShifted = 0
    yShifted = 0
    err = mse(basicImg[:matchN,:matchM], matchingImg)

    # find the smallest err and how much basicImg need to shift
    # such that the two images could best match
    for i in range(basicN-matchN):
        for j in range(basicM-matchM):
            temp = mse(basicImg[i:matchN+i, j:matchM+j], matchingImg)

            if(temp < err):
                err = temp
                xShifted = i
                yShifted = j

    return xShifted, yShifted, err

def findShiftWithShifted(basicImg, matchingImg, xShifted, yShifted, err):
    matchN, matchM = matchingImg.shape
    xShiftedReturn = xShifted
    yShiftedReturn = yShifted
    err = mse(basicImg[xShifted: xShifted + matchN, yShifted : yShifted + mat
chM], matchingImg)

    searchRangeX = min(xShifted, 2)
    searchRangeY = min(yShifted, 2)

    for i in range(-searchRangeX, searchRangeX):
        for j in range(-searchRangeY, searchRangeY):
            temp = mse(basicImg[xShifted + i : xShifted + matchN + i, yShifte
d + j : yShifted + matchM + j], matchingImg)

            if(temp < err):
                err = temp
                xShiftedReturn = xShifted + i
                yShiftedReturn = yShifted + j

    return xShiftedReturn, yShiftedReturn, err
```

C. Step 3: Coarse-to-fine registration.

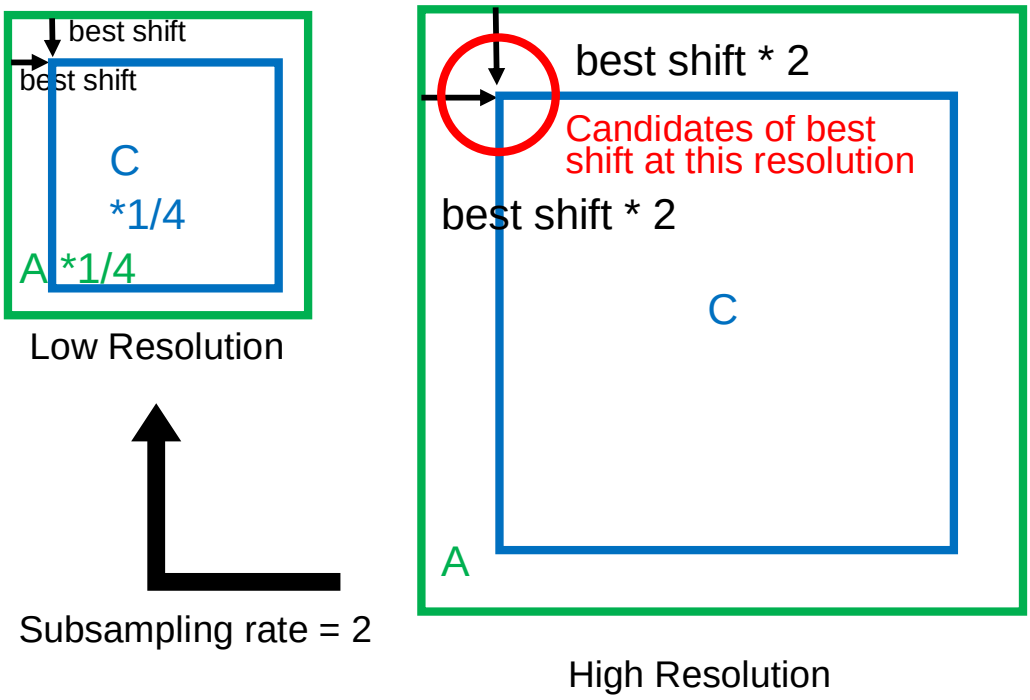
For high resolution negatives (~70MB), Minimizing MSE has too much computation to be efficient. Since it uses memory and time for computation propositional to the number of pixels, a big image takes too long, and sometimes cannot be completed due to the insufficiency of the computer's memory. Therefore, we use an image pyramid technique to help speeding up and reducing the memory needed. The concept of the pyramid is to do a computation on a low-resolution image first, then the higher

resolution can be done with a smaller offset, and hence reduce the number of computations needed.

If the image is too large, it is slow to find best shift by step 2, so we use the method, coarse-to-fine registration, to make the procedure fast and efficient. By the concept of coarse-to-fine registration, we first do the step 2 at low resolution stage, and we will find a best shift (Sx, Sy). Then we go to higher resolution stage, and multiply the (Sx, Sy) with the sampling rate to get the approximation of the best shift at this stage. That is, the equation between two best shifts shows below:

$$BestShift_{HighResolution} \approx BestShift_{LowResolution} \times SubsamplingRate$$

Once we get the approximation location of best shift at this stage, we can just evaluate the mean squared error surrounding this location, and this reduce the computation since we just consider the location and its surrounding as the best shift candidates.

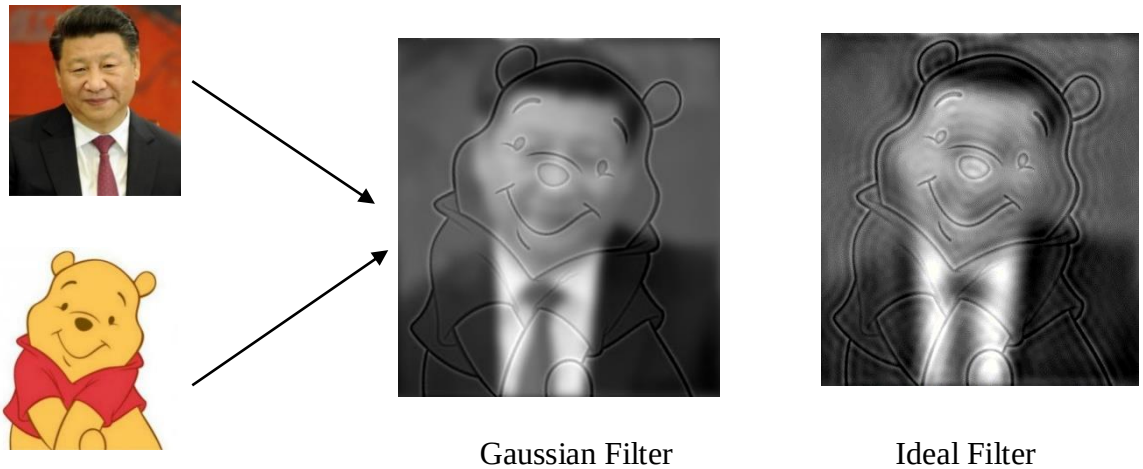


To summarize, we first get best shift value at low resolution by step 2, then we find best shift value at medium resolution by step 3, and keep finding best shift value at higher resolution by step 3 until we get the best shift at original resolution.

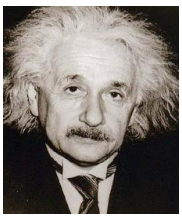
```
def downSearch(basicImg, matchingImg, sub_rate, iteration):
    xShifted = 0
    yShifted = 0
    basicPyramid = []
    matchingPyramid = []

    basicPyramid.append(np.array(basicImg))
    matchingPyramid.append(np.array(matchingImg))

    for i in range(iteration):
        n, m = basicImg.size
        basicImg = basicImg.resize((int(n/sub_rate), int(m/sub_rate)))
        basicPyramid.append(np.array(basicImg))

        n, m = matchingImg.size
        matchingImg = matchingImg.resize((int(n/sub_rate), int(m/sub_rat
e)))
        matchingPyramid.append(np.array(matchingImg))

    xShifted, yShifted, err = findShift(basicPyramid[iteration], matchin
gPyramid[iteration])
    #print "after findShift: ", xShifted, yShifted, err

    for i in range(iteration-1, -1, -1):
        xShifted *= sub_rate
        yShifted *= sub_rate
        xShifted, yShifted, err = findShiftWithShifted(basicPyramid[i],
matchingPyramid[i], xShifted, yShifted, err)
        #xShifted, yShifted, err = findShift(basicPyramid[i], matchingPy
ramid[i])
        #print "iteration:", i, ":", xShifted, yShifted, err

    return xShifted, yShifted, err
```

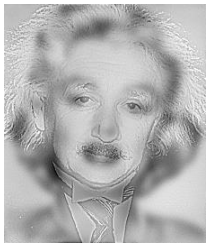
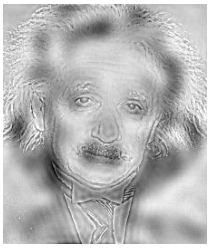
Experiment Result

- ## Hybrid Image

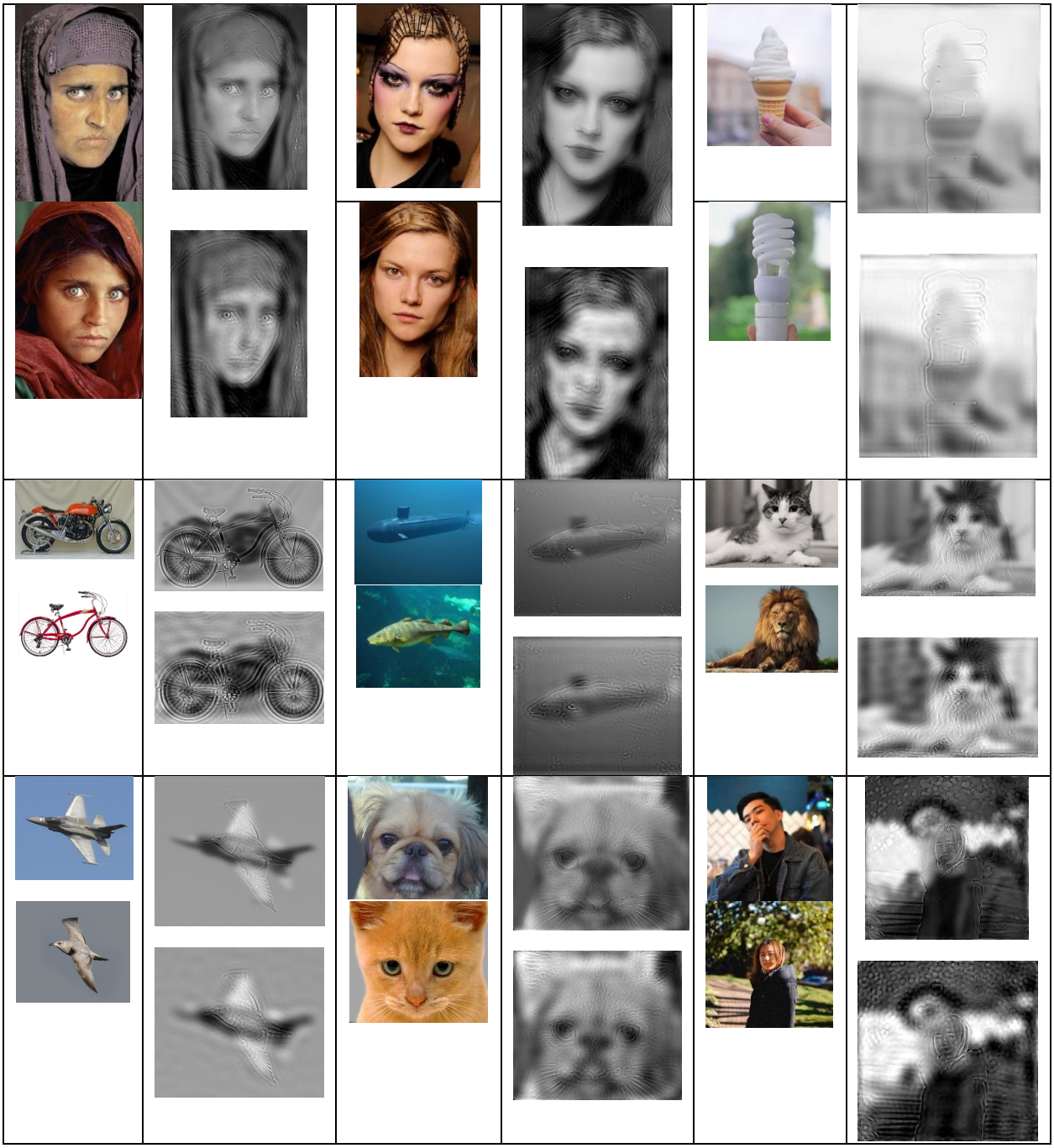


Gaussian Filter Ideal Filter



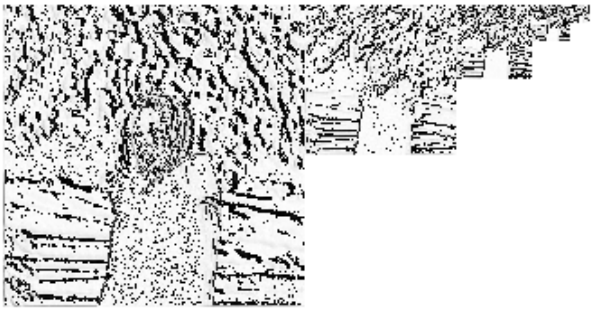
- **Other Hybrid Image Results**

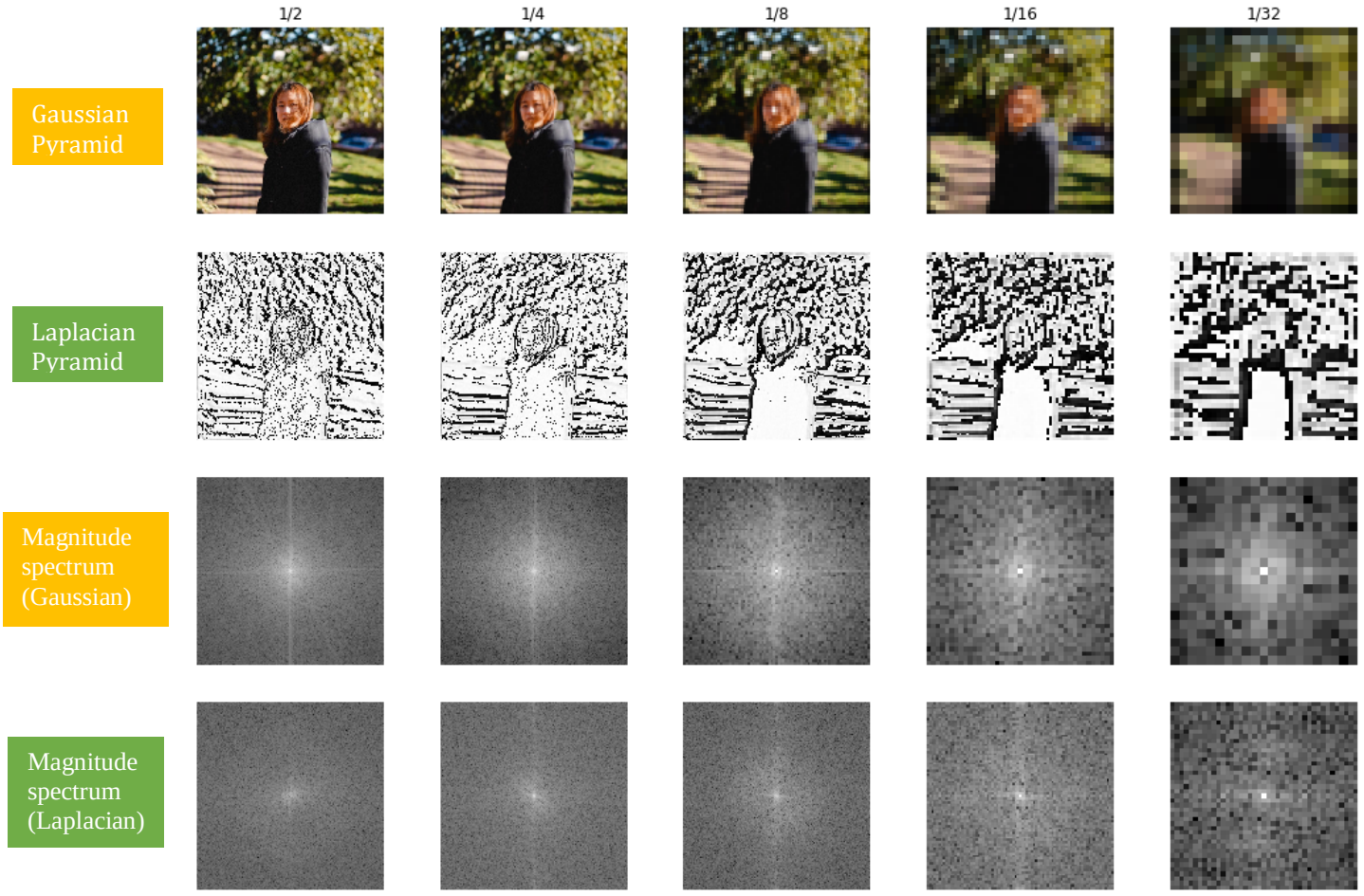


- **Image Pyramid**

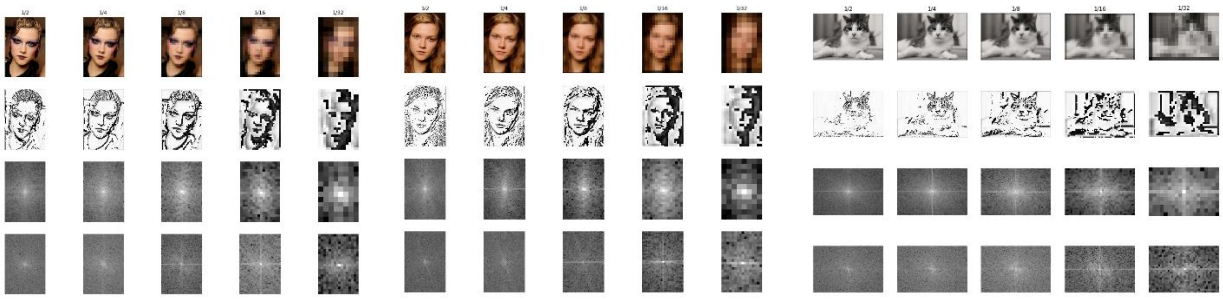
Gaussian Pyramid

Laplacian Pyramid

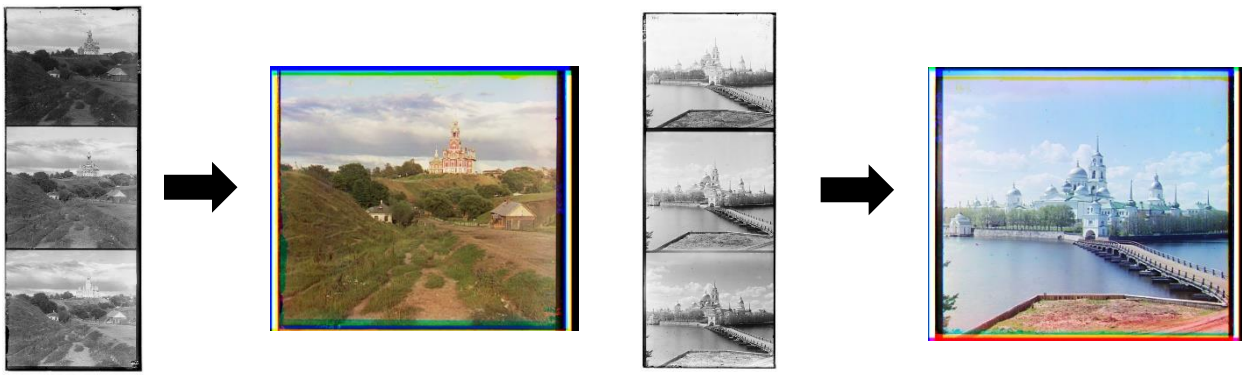




Other Image Pyramid Results



- **Colorizing the Russian Empire**



Other Images

Discussion

1. Hybrid Image

A. Comparing the output image using ideal filter and Gaussian filter, we can observe that the one using ideal filter have the edges that are not supposed to exist in the original image. The reason why it happened is because when transforming the ideal low pass filter from the frequency domain to the special domain, which is perform inverse Fourier transform, it turns into a sinc wave. It would result in the edges we see in the output image. For Gaussian filter, the shape basically stay the same in frequency and spatial domain, so it can give us the smooth outcome we wanted.

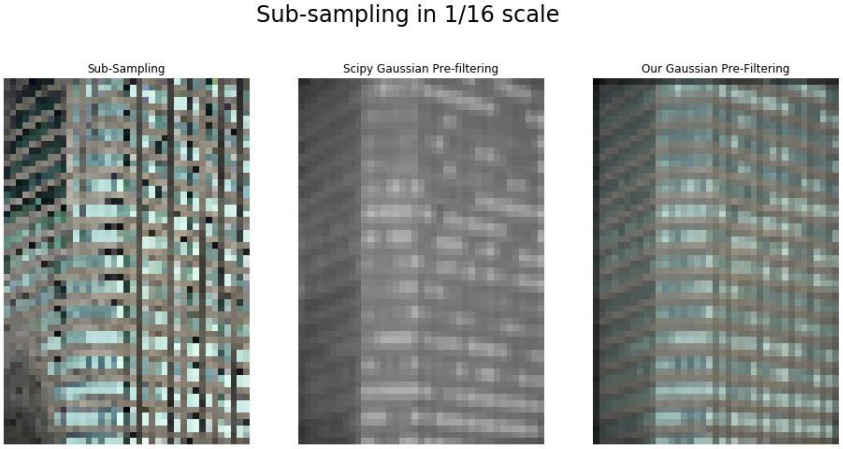
B. We use openCV function cv2.imread to load image. For this function, a colored image would be represented in a three-dimension numpy array, with RGB represented in three different dimensions. To simplify the algorithm, we choose to transform the colored image into grayscale image. In the visual aspect, we think that grayscale hybrid images have better effect.

C. To decide the cutoff frequency, we simply use our own eyes to distinguish the difference between the output of different output image.

2. Gaussian Pre-filtering

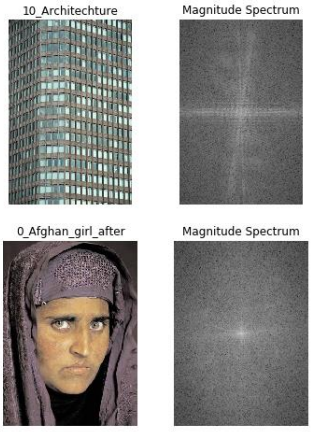
We use an image with lots of windows to show the **aliasing problem**. The following result shows that directly remove rows and columns (sub-sampling) will cause the aliasing problem

(leftest img). We can avoid that by first smoothing the image, then conduct sub-sampling after (rightest img).

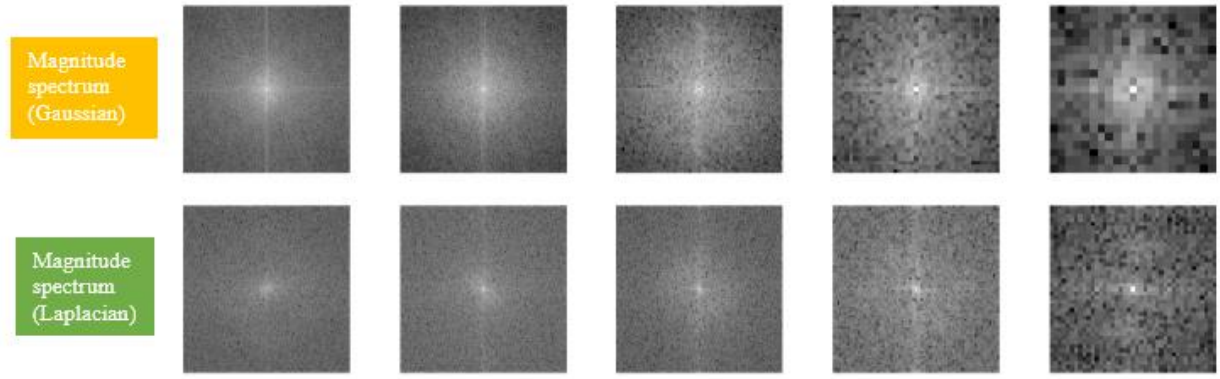


3. Magnitude spectrum

A. The magnitude spectrum of the image with lots of vertical and horizontal lines and the magnitude spectrum of the image with less vertical and horizontal lines are very different.



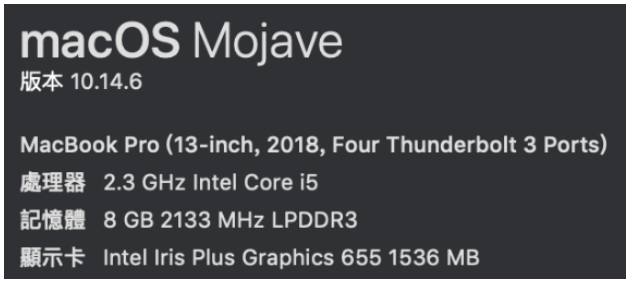
B. We can see the difference between the magnitude spectrum of Gaussian pyramid and the magnitude spectrum of Laplacian pyramid. Since the Laplacian pyramid remains the edges of the image, the magnitude spectrum of Laplacian pyramid lost lots of information in low frequencies.



4. Colorizing the Russian Empire

A. Deal with big input image

For high resolution negatives (~70MB), Minimizing MSE has too much computation to be efficient. Since it uses memory and time for computation propositional to the number of pixels, a big image takes too long, and sometimes cannot be completed due to the insufficiency of the computer's memory. Without coarse-to-fine registration, the process is slow when the input image is large, and we solve this problem by using coarse-to-fine registration. The following form shows the execution time among the program with and without coarse-to-find registration.



system environment

	Size (MB)	Execution time without Coarse-to-fine registration (s)	Execution time with Coarse-to-fine registration (s)
cathedral.jpg	0.169	0.62793	0.07616
monastery.jpg	0.156	0.66097	0.08365
nativity.jpg	0.18	0.62645	0.06939
tobolsk.jpg	0.177	0.62564	0.06884
icon.tif	72.8	Unable to calculate	3.78823
lady.tif	72.5		3.64694
melons.tif	73.3		2.55228
onion_church.tif	73		3.71711
three_generations.tif	71.5		3.55428

train.tif	72.7		3.63135
village.tif	75		3.86830
workshop.tif	72		3.83347
emir.tif	71.3		3.81530

B. Normalization of TIFF image

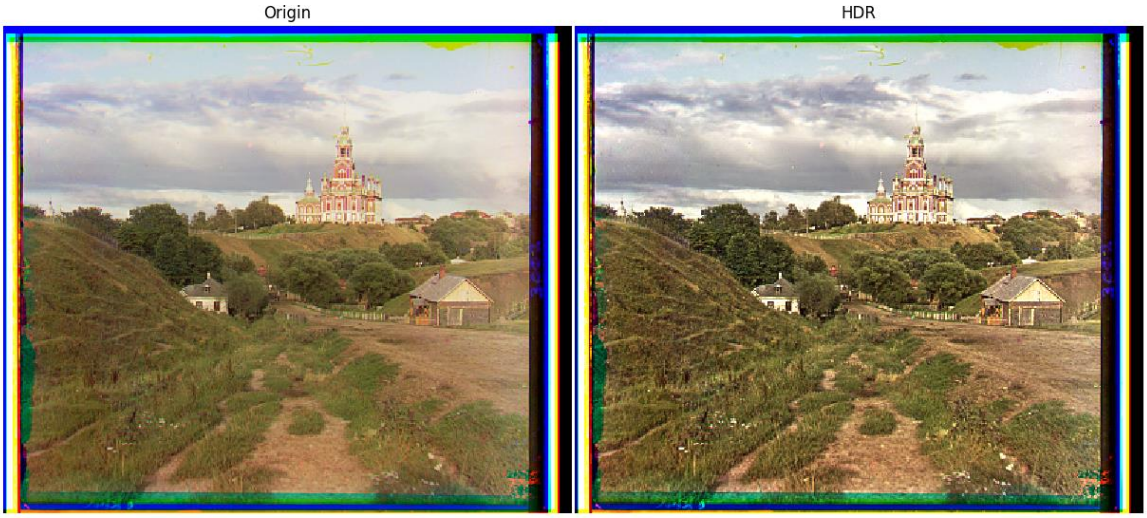
Input images have jpeg type and tiff type. At first, our code can handle the jpeg type input, but fail with tiff type input. The problem is that tiff type input use 16 bits per pixel to store the information, and we cannot put a 16 bits information into a RGB channel, which takes 8 bits per pixel to store information. We solve this problem by normalizing 16 bits and extending it into 8 bits.

```
1.  if img.format == "TIFF": # check the format of input dataset
2.          _, depth = img.mode.split(';') # seperated by ';' (I;depth)
3.          depth = float(depth) # 16
4.          # normalize 16 bits and extend it into 8 bits
5.          newRimg.paste(Image.fromarray(np.array(r) / (2 ** depth - 1) * 255.
    0).convert("L"), (ryShifted, rxShifted))
6.          #print "r: ",r
7.          newGimg.paste(Image.fromarray(np.array(g) / (2 ** depth - 1) * 255.
    0).convert("L"), (gyShifted, gxShifted))
8.          #print "g: ",g
9.          newBimg.paste(Image.fromarray(np.array(b) / (2 ** depth - 1) * 255.
    0).convert("L"), (byShifted, bxShifted))
10.         #print "b: ",b
11.     else:
12.         # for other img format
13.         newRimg.paste(r, (ryShifted, rxShifted))
14.         newGimg.paste(g, (gyShifted, gxShifted))
15.         newBimg.paste(b, (byShifted, bxShifted))
```

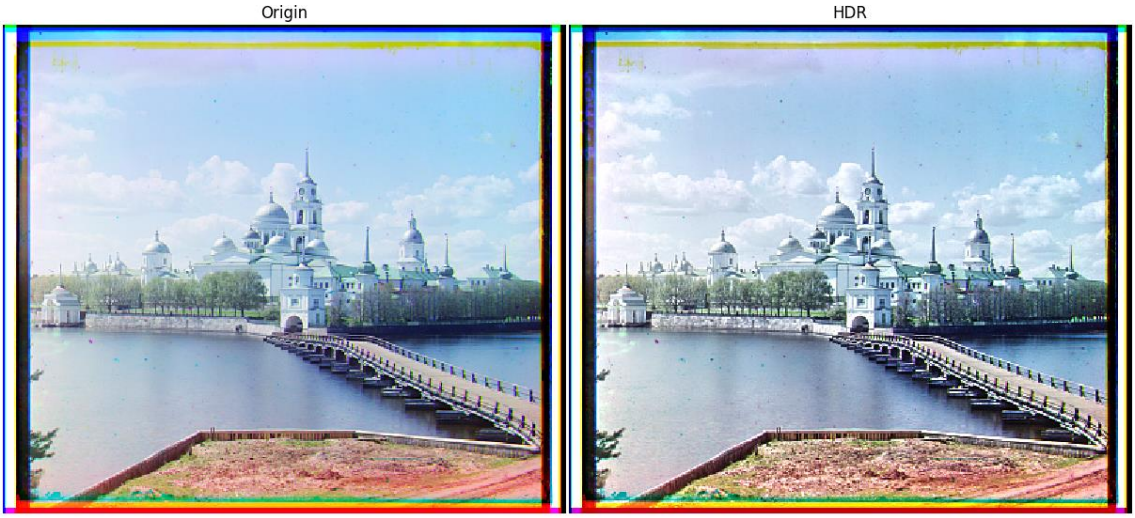
C. HDR enhancement of output image

From the result, we could suspect that some of images do not align well due to two reasons: First, the input channels have very different luminance. The bottom channel (Red) is relatively brighter than the others. And second, the image is very colorful and textured. This causes minimizing MSE to break down as a metric, when it prefers to align whites with whites and blacks with blacks. One possibility of improving an image is using High-dynamic-range imaging (HDRI) by improving the color balance of the images. HDR is a high dynamic range technique used in imaging and films to reproduce a greater dynamic range of luminosity than what is possible with standard digital imaging or photographic techniques. Note that Resolution of input image should not be too large (it depends on machine's memory) since using SciPy to solve a large-scale linear system may cause memory exhausted. Hence, we take four small-sized images as HDR enhancement example. The result is as follow.

Result



cathedral.jpg



monastery.jpg

Conclusion

In this assignment, we learned that

1. Gaussian Filtering
2. Demonstrate the effect of hybrid image

3. Correct way to down-scale an image
4. How to construct a Gaussian pyramid and a Laplacian pyramid
5. A color image can be reconstructed from a three-channel negative by manual alignment and careful color adjustment

Besides, we study how to align the three images for the various plates and automatically produce a color image. After finishing this project, we are more familiar with image processing. We know how powerful Python is in image processing.

Reference

1. http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/f07/proj1/www/wwedler/
2. https://andrewdcampbell.github.io/colorizing-the-prokudin-gorskii-photo-collection
3. https://inst.eecs.berkeley.edu/~cs194-26/fa17/upload/files/proj1/cs194-26-aec/
4. http://vision.gel.ulaval.ca/~jflalonde/cours/4105/h19/tps/tp1/index.html?lang=en
5. https://www.researchgate.net/publication/249873740_Automatic_Digicromatography_Colorizing_the_Images_of_the_Russian_Empire

Work Assignment Plan Between Team Members

0516044 陳思妤	Hybrid Image
0856733 黃明翰	Image Pyramid
0786031 廖俊凱	Colorizing