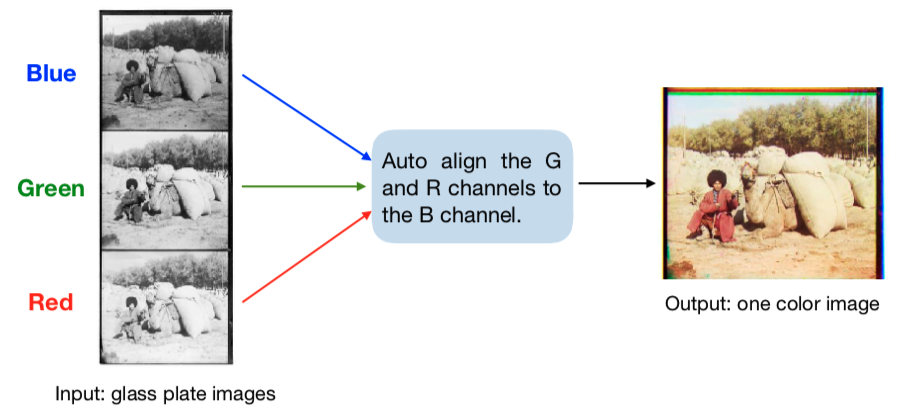
Homework 2

Group 12

0786031廖俊凱 0516044陳思妤 0856733黃明翰

1. Introduction

[Sergei Mikhailovich Prokudin-Gorskii](http://en.wikipedia.org/wiki/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.



1. Implementation Procedure
   1. 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.

1. # crop(left, upper, right, lower)
2. # devide image into the three b g r channels
3. b = img.crop((0,                     0, oriWidth,             maxHeight))
4. g = img.crop((0,             maxHeight, oriWidth, oriHeight - maxHeight))
5. r = img.crop((0, oriHeight - maxHeight, oriWidth,             oriHeight))
   1. 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.
6. **def** mse(imageA, imageB):
7. err = np.sum((imageA.astype("float") - imageB.astype("float")) \*\* 2)
8. err /= float(imageA.shape[0] \* imageA.shape[1])
9. **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).

A

C

Fig.2

A

best shift

Fig.3

best shift

C

B

A

shift-cut

shift-cut

Fig.4

B

C

cut

Fig.1

1. **def** findShift(basicImg, matchingImg):
2. basicN, basicM = basicImg.shape   # N:height M:width
3. matchN, matchM = matchingImg.shape # N:height M:width
5. xShifted = 0
6. yShifted = 0
7. err = mse(basicImg[:matchN,:matchM], matchingImg)
9. # find the smallest err and how much basicImg need to shift
10. # such that the two images could best match
11. **for** i **in** range(basicN-matchN):
12. **for** j **in** range(basicM-matchM):
13. temp = mse(basicImg[i:matchN+i, j:matchM+j], matchingImg)
15. **if**(temp < err):
16. err = temp
17. xShifted = i
18. yShifted = j
20. **return** xShifted, yShifted, err
22. **def** findShiftWithShifted(basicImg, matchingImg, xShifted, yShifted, err):
23. matchN, matchM = matchingImg.shape
24. xShiftedReturn = xShifted
25. yShiftedReturn = yShifted
26. err = mse(basicImg[xShifted: xShifted + matchN, yShifted : yShifted + matchM], matchingImg)
28. searchRangeX = min(xShifted, 2)
29. searchRangeY = min(yShifted, 2)
31. **for** i **in** range(-searchRangeX, searchRangeX):
32. **for** j **in** range(-searchRangeY, searchRangeY):
33. temp = mse(basicImg[xShifted + i : xShifted + matchN + i, yShifted + j : yShifted + matchM + j], matchingImg)
35. **if**(temp < err):
36. err = temp
37. xShiftedReturn = xShifted + i
38. yShiftedReturn = yShifted + j
40. **return** xShiftedReturn, yShiftedReturn, err
    1. 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 computation 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:

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.

A

best shift \* 2

A \*1/4

best shift

Low Resolution

best shift

C \*1/4

best shift \* 2

Candidates of best shift at this resolution

C

Subsampling rate = 2

High Resolution

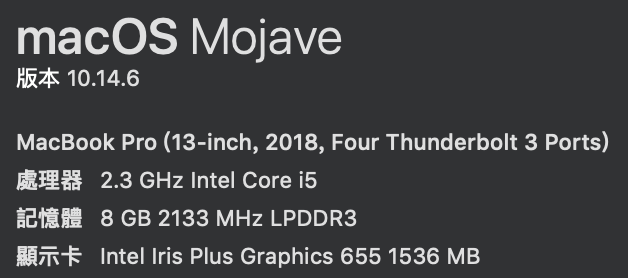
To summarize, we first get best shift value at low resolution by step 2, then we find best shift value at medium resolution by step 3, and keep finding best shift value at higher resolution by step 3 until we get the best shift at original resolution.

1. **def** downSearch(basicImg, matchingImg, sub\_rate, iteration):
2. xShifted = 0
3. yShifted = 0
4. basicPyramid = []
5. matchingPyramid = []
7. basicPyramid.append(np.array(basicImg))
8. matchingPyramid.append(np.array(matchingImg))
10. **for** i **in** range(iteration):
11. n, m = basicImg.size
12. basicImg = basicImg.resize((int(n/sub\_rate), int(m/sub\_rate)))
13. basicPyramid.append(np.array(basicImg))
15. n, m = matchingImg.size
16. matchingImg = matchingImg.resize((int(n/sub\_rate), int(m/sub\_rate)))
17. matchingPyramid.append(np.array(matchingImg))
19. xShifted, yShifted, err = findShift(basicPyramid[iteration], matchingPyramid[iteration])
20. #print "after findShift: ", xShifted, yShifted, err
22. **for** i **in** range(iteration-1, -1, -1):
23. xShifted \*= sub\_rate
24. yShifted \*= sub\_rate
25. xShifted, yShifted, err = findShiftWithShifted(basicPyramid[i], matchingPyramid[i], xShifted, yShifted, err)
26. #xShifted, yShifted, err = findShift(basicPyramid[i], matchingPyramid[i])
27. #print "iteration:", i, ":", xShifted, yShifted, err
29. **return** xShifted, yShifted, err
30. Experimental Result

|  |  |
| --- | --- |
|  | Result |
| cathedral.jpg |  |
| monastery.jpg |  |
| nativity.jpg |  |
| tobolsk.jpg |  |
| icon.tif |  |
| lady.tif |  |
| melons.tif |  |
| onion\_  church.tif |  |
| three\_  generations.tif |  |
| train.tif |  |
| village.tif |  |
| workshop.tif |  |
| emir.tif |  |

1. Discussion
   1. 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 |

* 1. 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))
    1. 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 luminances. 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 image as HDR enhancement example. The result is as follow.

|  |
| --- |
| Result |
| cathedral.jpg |
| monastery.jpg |
| nativity.jpg |
| tobolsk.jpg |

1. Conclusion

In this assignment, we learned that 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.

1. Reference
   1. Hybrid Image
   2. Image Pyramid
   3. Colorizing the Russian Empire
      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>
2. Work Assignment Plan Between Team Members

|  |  |
| --- | --- |
| 0786031廖俊凱 | Colorizing the Russian Empire |
| 0516044陳思妤 | Hybrid Image |
| 0856733黃明翰 | Image Pyramid |