CS543/ECE549 Assignment 1

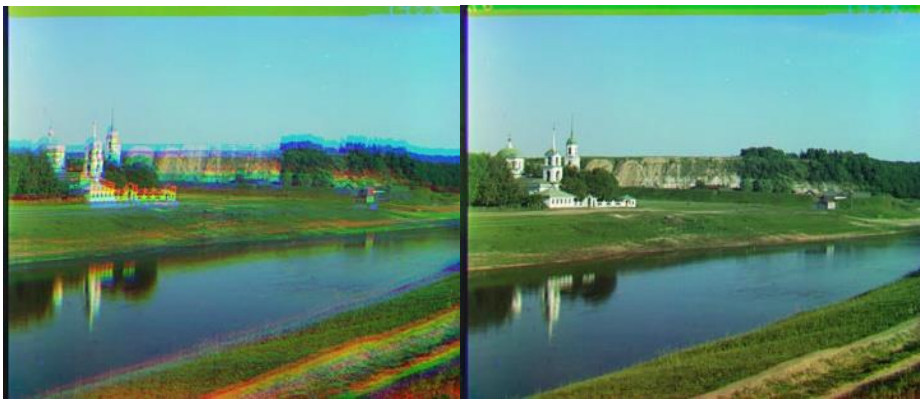
Name: Li-Kai Chuang
NetId: likaikc2

Part 1 : Implementation Description

Provide a brief description of your implemented solution, focusing especially on the more "non-trivial" or interesting parts of the solution.

- What implementation choices did you make, and how did they affect the quality of the result and the speed of computation?
- What are some artifacts and/or limitations of your implementation, and what are possible reasons for them?

We're given B, G, and R channels of a single image. Our goal is to perfectly align these three channels so that after we stack all of them together, the colorized image can exhibit a clear result, where there is no weird artifact like an illusion. The figures below illustrate what is considered a good or a bad result. (On the left is an example of bad result. On the right is a good result.)



To do this, we choose one of the channels as the base, say B channel. Then all we have to do is to find the displacement of G and R channels. So our ultimate goal boils down to finding the best displacement, which can lead to a perfect alignment.

An exhaustive search is a way to solve this problem. It works for a low-resolution image, but unfortunately would take too much time if we apply this brute force approach on a high-resolution image. Therefore, for the implementation of this assignment, there would be two methods. A normal_align() for low-resolution image, and the other pyramid_align() for high-resolution image.

First, to do a normal_align(), we search in a window [-15, 15] of x and y direction respectively, for each displacement pair (x, y), try to compute the similarity of two channels. A best alignment should give us a highest score among all displacement pairs. Furthermore, the way we compute the similarity is by using Zero Normalized Cross-Correlation method. The pseudo code is as follows:

```
normalAlignment(base_channel, channel2)
    // Initialization
    max_score = -1
    best_d = (0, 0)
    // Search best displacement
    for x = -15 to 15
       for y = -15 to 15
          shifted_channel2 = shift channel2 by (x, y)
          score = NCC(base_channel, shifted_channel2)
          if(score > max_score)
              max_score = score
              best_d = (x, y)
    return best_d
```

Something worth noting is that when we're computing the NCC of two channels, it's really a good idea not to score the pixels around the corner. Therefore, I crop out 10% of the image from both ends. Only the middle part of the image is scored. Furthermore, the reason we choose NCC over SSD is that the brightness of each channel is not necessarily similar, so a zero normalization before doing correlation might be a better idea.

```
NCC(image1, image2)
    // crop the image from both ends
    height, width = image1.shape
    image1 = image1[10% : 90%height, 10% : 90%width]
    image2 = image2[10% : 90%height, 10% : 90%width]
    // zero-mean & normalize
    image1 = image1 – image1.mean
    image2 = image2 – image2.mean
    image1 = image1 / norm(image1)
    image2 = image2 / norm(image2)
    return dot_product(image1, image2)
```

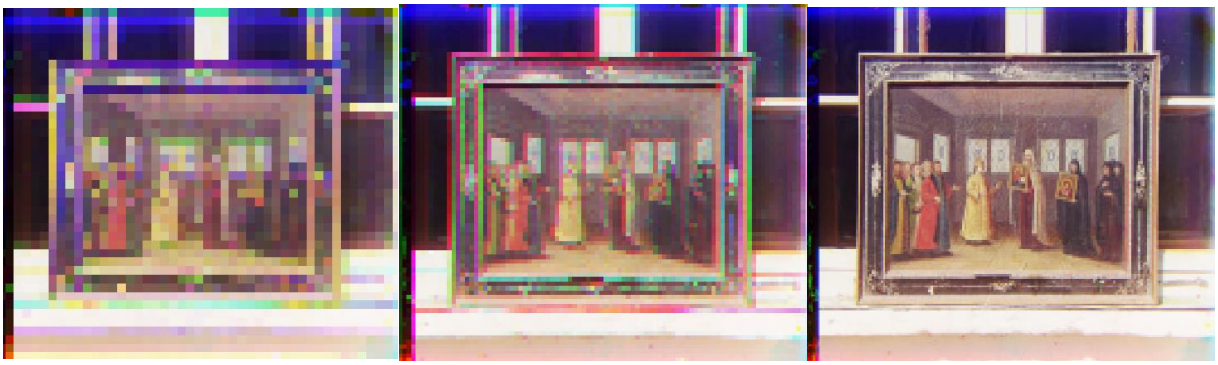
To optimize, I use np.roll() to shift the image, which simply moves the pixel from one side of the image to the other side. Therefore, the moved pixels remain in the original image, which makes the average value of the image stay the same. That way we don't need to re-calculate the mean value of image in the NCC function every time when we are doing new displacement search. We just compute it once and store it. It helps a lot to reduce the time we compute the NCC. The code is trivial so I'm not attaching the pseudo code again.

Now we can totally apply the normal alignment method to the high-resolution image, and that will give us a correct result as well. However, it'll cost too much running time. To handle this problem, we use image pyramid method to reduce the time.

More specifically, we first resample the image into smaller size to get multiple images with different scales, just like a pyramid. Since the resolution is not high, we can apply the exhaustive search on this small image to get an estimation of the best displacement. As we step into a higher resolution, we can search the best displacement in a smaller range since we already got a meaningful estimation from last step. We recursively do that until we find a best displacement on the original high-resolution image.

```
// global original image: img
pyramidAlignment(scale, x0, y0)
    if(scale < 1)
        merge_img() // reached the bottom. Best displacement found
        return
    image = img.resize(scale)
    best_x, best_y = nccAlign(x0, y0) // find best displacement on smaller image
    pyramidAlignment(scale/2, best_x, best_y) // Use local results as a good estimate
```

We can see the alignment at each layer.



Result:

The searching range will be smaller and smaller as the we go to bottom of the pyramid (the higher resolution). The way I do it is simply set the range inversely proportional to the scale, like this:

```
// base_channel, channel2
nccAlign(scale, x0, y0)
    // Initialization
    max_score = -1
    best_d = (0, 0)
    // Search best displacement
    for x = [2*x0 – scale, 2*x0 + scale]
        for y = [2*y0 – scale to 2*y0 + scale]
            shifted_channel2 = shift channel2 by (x, y)
            score = NCC(base_channel, shifted_channel2)
            if(score > max_score)
                max_score = score
                best_d = (x, y)
    return best_d
```

Another thing worth noting is that as we pass the estimate of displacement to the next layer, we need to multiply the x0 and y0 by the factor we scale (Here it's 2). The reason is obvious. We will need to move twice as far as we do in the low-resolution image in order to achieve the same alignment.

The limitation of this pyramid method is that it doesn't always perform well on low-resolution image. Since the image is already small, which means when we further resample this image, we will get an image without much information. In such case, the correlation method might not work as planned. To deal with this problem, we need to avoid the problem of missing information caused by resampling. We can achieve that by first using low pass filter or gaussian to blur the image, then we resample.

Part 2: Basic Alignment Outputs

For each of the 6 images, include channel offsets and output images. Replace <C1>, <C2>, <C3> appropriately with B, G, R depending on which you use as the base channel.

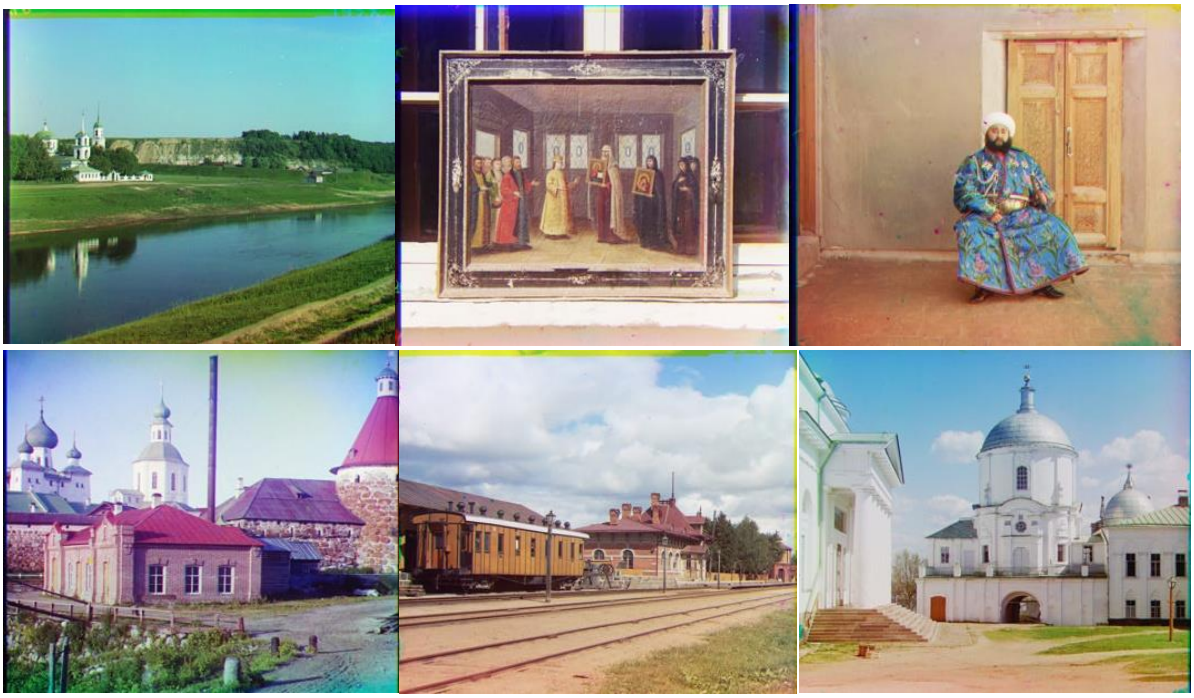
A: Channel Offsets

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)	(14, 5)
00351v.jpg	(4, 1)	(14, 1)
00398v.jpg	(5, 3)	(12, 4)
01112v.jpg	(0, 0)	(6, 1)

B: Output Images

Insert the aligned colorized outputs for each image below (in compressed jpeg format):



Part 3: Multiscale Alignment Outputs

For each of the 3 high resolution images, include channel offsets and output images. Replace <C1>, <C2>, <C3> appropriately with B, G, R depending on which you use as the base

channel. You will also need to provide an estimate of running time improvement using this solution.

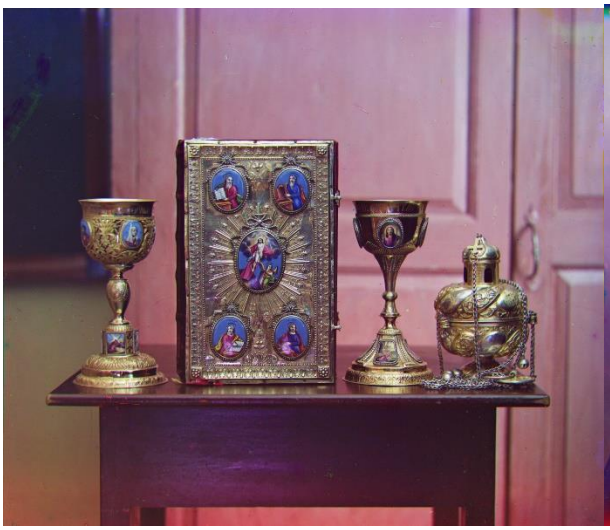
A: Channel Offsets

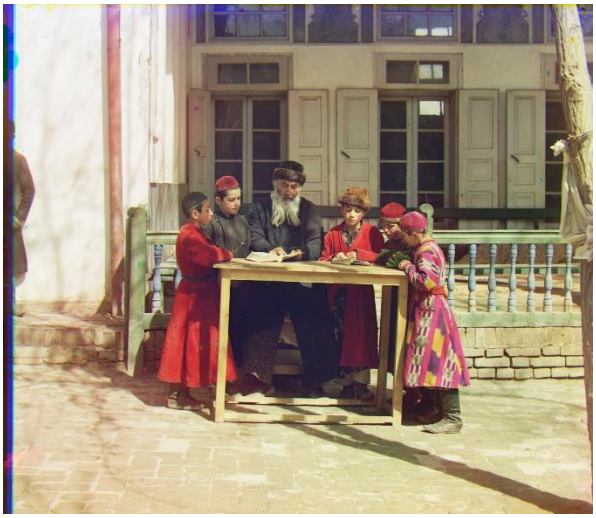
Using channel B as base channel:

Image	G (h,w) offset	R (h,w) offset
01047u.tif	(25, 20)	(72, 33)
01657u.tif	(55, 8)	(118,11)
01861a.tif	(72, 39)	(149, 63)

B: Output Images

Insert the aligned colorized outputs for each image below (in compressed jpeg format):





C: Multiscale Running Time improvement

Report improvement for the multiscale solution in terms of running time (feel free to use an estimate if the single-scale solution takes too long to run). For timing, you can use the python `time` module, as described in the assignment instructions.

Multiscale factor: 2
Pyramid height: 3 layers
Unit: seconds

Image	Normal Alignment	Pyramid Alignment
01047u.tif	Over 3 minutes	9.418
01657u.tif	Over 3 minutes	9.077
01861a.tif	Over 3 minutes	9.038
00125v.jpg	2.774	0.483
00149v.jpg	2.545	0.458
00153v.jpg	2.256	0.420 (Not perfectly aligned)
00351v.jpg	2.417	0.403
00398v.jpg	2.674	0.401
01112v.jpg	2.480	0.433

Part 4: Bonus Improvement

Using the above settings, 00153v.jpg will have some problem aligning. A possible reason is that since the image is already small, which means when we further resample this image, we will get an image without much information. In such case, the correlation method might not work as planned. Therefore, I tune the parameters a little bit and test those images again. It runs successfully when I reset the searching range to be smaller, which avoids the channels being displaced too much in a low-resolution image. Also, when computing NCC, I crop the borders even more to improve the correlation. Besides, decreasing the layer of pyramids also help to retain information in low-resolution image.

To crop out the unwanted border of the stacked image, we can directly crop out 5% of the image from both ends and that will give us a good result. However, we can use the displacement that we got from the alignment to further have a better estimate on the border. A intuitive approach would be first look at the displacement is positive or negative. If it’s positive,

then we know the top-left part should be cropped out more, whereas the negative values tell us the bottom-right should be cropped out more.

Another approach would be to look at the gradient of the stacked image. The unwanted order will exhibit abrupt changes in pixel values, which we can tell from the gradient image. Then we can simply search from the border toward the middle, to find that abrupt changes. That will give us information about whether it's the border we want or not.

I tried both above approaches and I think the first approach is enough for us to handle the situation now, since we know in advance that the three channels are obtained from a single image divided by three parts. That gives us an idea of the way to crop the image. However if it's an image obtained randomly from elsewhere, the displacement is not meaningful anymore. In such case, the second approach would be better.