CSC320: Project 1

The Prokudin-Gorskii Colour Photo Collection

Ramaneek Gill

January 24, 2015

Part 0

Input data
The project's goal is to create a python program that automatically align the inverted blue, green, and red-filtered negatives into one colour photo. I will be using the Anaconda for a Python shell with NumPy, SciPy and MatPlotLib packages.

The images this project works with can be found here:
`http://www.cs.toronto.edu/ guerzhoy/320/proj1/images.zip`

The code **p1.py** is included at the end of the report. It is easy enough to understand on its own. I will leave it to you to read it and understand how it works. I have also included the runtimes for each image alignment and its displacemnt in each figure's description. The code is run on an intel i5-3337 CPU at 1.80 GHz laptop on Windows 8.1 set on high performance battery life and each runtime is rounded to the nearest even number for every 10 milliseconds.

The input is a jpg or png image that contains the blue green and red-filtered negatives in order from top to bottom, an example of this is shown Figure 1 below. Keep in mind that the png images are very large (about 10 times larger than the jpgs). The png images have are at a resolution of 3741×9656 pixels while the jpgs are at a resolution of 394×2014 pixels.

Some key points: Each image was cropped 10% *from each side to get rid of the black borders, NCC alignment is done by subtracting the average of the image, SSD was calculated using only the overlapping area (as suggested by Michael during his office hours), and aligning the large png images used a gaussian pyramid from levels 3 to 0 to come to a satisfactory and efficient alignment.*

Figure 1: '00757v.jpg', a jpg input image `imshow('00757v.jpg'); show()`

Figure 2 is a png picture, note that it is scaled to be a quarter of the size of Figure 1 in this report to show how big of an image the program will have to work with.

Figure 2: '00089u.png', a png input image `imshow('00089u.png'); show()`

Part 1

Applying an SSD or NCC image alignment on jpg images
In part 1 I will only be concerned with aligning jpg images. I do this because before aligning the larger (png) images I first need to experiment and see if I should align images with SSD or NCC algorithms.

In my program **p1.py** I first test each (jpg) image by manually assigning the image file name in the variable image_name and alternating the value of use_SSD between 1 and 0 this is because if use_SSD is 1 the program will align the image with SSD, if it is 0 it will use NCC. (lines 137-138 of p1.py).

Below are various figures, read their descriptions to see which algorithm was applied (SSD xor NCC) and the specific image.

(a) Input image

(b) SSD alignment
runtime: 0.57s
red offset:(8,-4)
green offset:(3,1)

(c) NCC alignment
runtime: 1.76s
red offset:(5,5)
green offset:(3,2)

Figure 3: 00757v.jpg

(a) Input image

(b) SSD alignment
runtime: 0.57s
red offset:(9,9)
green offset:(-1,1)

(c) NCC alignment
runtime: 01.69s
red offset:(1,0)
green offset:(-1,1)

Figure 4: 00911v.jpg

(a) Input image

(b) SSD alignment
runtime: 0.58s
red offset:(2,4)
green offset:(2,1)

(c) NCC alignment
runtime: 01.66s
red offset:(2,4)
green offset:(1,1)

Figure 5: 01031v.jpg

Part 1 continued on next page...

(a) Input image

(b) SSD alignment
runtime: 00.58s
red offset:(2,8)
green offset:(1,6)

(c) NCC alignment
runtime: 01.69s
red offset:(1,10)
green offset:(1,5)

Figure 6: 01657v.jpg

(a) Input image

(b) SSD alignment
runtime: 0.59s
red offset:(-10,5)
green offset:(2,5)

(c) NCC alignment
runtime: 01.94s
red offset:(1,10)
green offset:(2,6)

Figure 7: 01880v.jpg

These are only a couple of the output results. First just by analysing each image we can see that SSD's runtime is much quicker than NCC. However this speedy execution results in less than desirable alignments. NCC seems to offer optimal results at reasonable runtimes.

Figure 3 is a good example of NCC's longer computation resulting in better alignment. Figure 3's SSD alignment is clearly off which is shown by the 'blurry' image, but by looking closer you can actually tell that one of the red or green channel is misaligned by a significant amount of pixels. Comparing this to its NCC alignment we can conclude NCC was better in this case. Note that both of the images have an artifact on the left side (a vertical bar) this is most likely from the same vertical bar appearing on the blue filter section of the input image (the bottom one).

Figure 4 shows the same result as above.

Figure 5 has almost the exact same displacement vectors for both SSD and NCC alignments. This is most likely because the intensity in each of the red, blue, and green filters are similar thus producing a similiar for SSD compared to NCC. This implies that images that have similar intensities across the rgb channels can probably be aligned using SSD instead of NCC for a faster alignment, unfortunately this is usually not the case.

Figure 6 shows similar results to Figure 3 and Figure 4.

Figure 7 is an 'strange' result when compared to all the other input images that were processed. Both SSD and NCC did a poor job of alignment, this is probably due the different lettering on the building for the red and green filter. Since the blue filter was used as a base for the offsetting of red and green both the NCC and SSD resulted in a poor offset for the red filter. This is because the red filter is a 'more' different image compared to the blue filter than the green filter. Even with such a strange input NCC edges out SSD yet again for being more accurate.

After analysing the images I can conclude that aligning images using NCC is far more superior in accuracy when compared to SSD in most results. This is the conclusion reached from the input images given by Sergei Prokudin-Gorskii for this project.

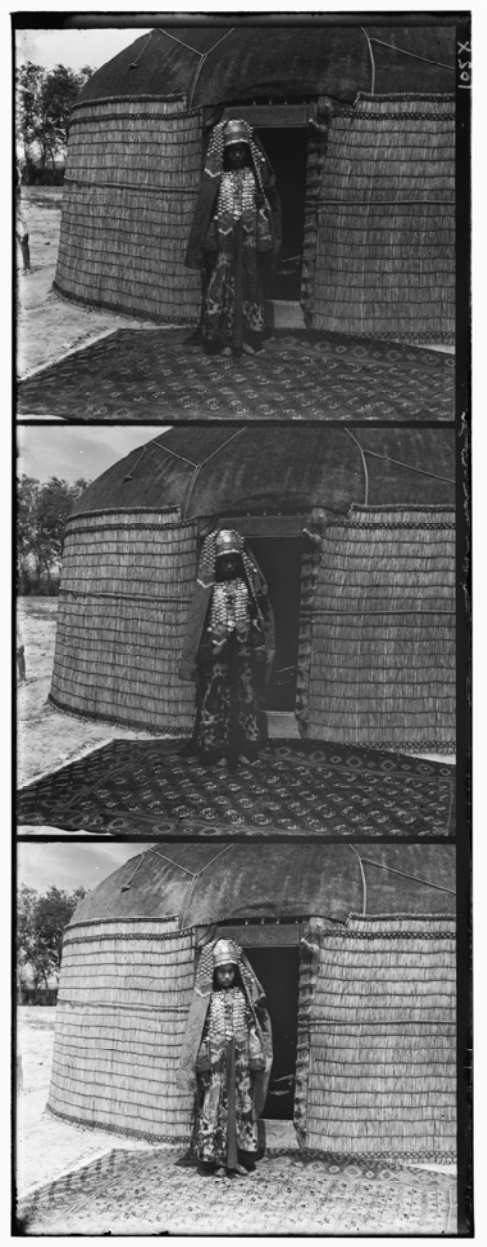
Note that the vertical bars on the left and right and the horizontal bars on the top and bottom of each image is a result of a circular shifting of the image matrix with the roll() function. This can be discarded as the image is being processed but the black bars left over from deleting this data would have skewed the alignment even further and potentially would have cut out some important image context information.

Part 2

Aligning the large (png) input images.
Aligning the large images using the same algorithm for Part 1 would result in a very poor result. This is because we are only looking for a displacement from -10 to +10 pixels for both the red and green filter. In order to align it well I have consecutively aligned each image 4 times at gaussian pyramid levels from 3 to 0. To clarify; I first aligned an image at 1/8th its size and found an offset between -80 and +80, then aligned that modified image at 1/4th its size to an offset of -40 to +40, then again at on half its size to an offset of -20 to +20 and finally one more time at its full size to an offset of -10 to +10.

I aligned each image using NCC, this is because Part 1 showed NCC was a superior algorithm for alignment than SSD. Below are a couple results: *Note: the maximum offset for each red and green filter is -150 to +150 pixels, this means that if one rgb filter from an input image was very misaligned it will always produce a bad result with this program, fortunately enough Michael supplied us with tame test inputs.*

Part 2 continued on next page...



(a) Input image

(b) NCC alignment
runtime: 342.29s
red offset:(55,107)
green offset:(38,47)

Figure 8: 00087u.png

Part 2 continued on next page...

(a) Input image

(b) NCC alignment
runtime: 344.07s
red offset:(32,87)
green offset:(5,42)

Figure 9: 00458u.png

Part 2 continued on next page...

(a) Input image

(b) NCC alignment
runtime: 348.90s
red offset:(14,41)
green offset:(6,14)

Figure 10: 00737u.png

(a) Input image

(b) NCC alignment
runtime: 348.62s
red offset:(33,125)
green offset:(25,57)

Figure 11: 00822u.png

Part 2 continued on next page...

(a) Input image

(b) NCC alignment
runtime: 346.19s
red offset:(17,11)
green offset:(10,-6)

Figure 12: 00822u.png

As you can see this algorithm produces amazingly accurate results (try and zoom in on the pictures, the pdf doesn't do it justice!) and in a reasonable amount of time. Once again the artifacts on the borders of the image have to do with the roll() function in python, read the previous note to understand why it occurs and why I left it that way. You can verify the accuracy of the results by looking at small patterns in the images, ie. the foilage in the forest/trees, the wheel spokes of the train, the reflection of the water.

p1.py:

```
from pylab import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import random
import time
from scipy.misc import imread
from scipy.misc import imresize
import matplotlib.image as mpimg
import os

os.chdir('C:/Users/Ramaneek/SkyDrive/Documents/University/Third Year/CSC320/project 1/')

matplotlib
gray()

#########################################################################

#function to align img1 to img2, returns shifted img1, if use_SSD is > 0 then
#img1 is aligned using SSD, NCC if use_SSD < 0
def align(img1, img2, use_SSD):
   print 'aligning ', image_name

   #so we don't overwrite the original images
   new_img1 = img1
   new_img2 = img2

   if use_SSD > 0:
      x_offset, y_offset = ssd_offset(new_img1, new_img2)
   else:
      x_offset, y_offset = ncc_offset(new_img1, new_img2)

   #circurally shifts the matrix, this makes last elements wrap around
   #since only affects a very small edge of the image just leave it
   print 'total offset is (',x_offset,',',y_offset,')'
   new_img1 = roll(new_img1, x_offset, 1)
   new_img1 = roll(new_img1, y_offset, 0)

   return new_img1

```

```
#aligning images for png files (the big images)
#first match is done on 1/8th of img, then another time on 1/4th
def align_png(img1, img2, use_SSD):
    print 'aligning', image_name
    new_img1 = img1 #refers to resized images that are passed to ncc_offset
    new_img2 = img2
    aligned_img1 = new_img1 #refers to the most up to date full size aligned img
    totalx = 0; totaly = 0

    for ratio in reversed(range(0,4)):
        #aligning gaussian pyramid at level 3->2->1->original img size
        new_img1 = imresize(aligned_img1, 0.5**ratio)
        new_img2 = imresize(img2, 0.5**ratio)

        if use_SSD > 0:
            x_offset, y_offset = ssd_offset(new_img1, new_img2)
        else:
            x_offset, y_offset = ncc_offset(new_img1, new_img2)

        #resize offsets to coordinate with the original image size
        #I do 2**ratio because the image was reduced 0.5**ratio
        x_offset = int(x_offset*(2**ratio))
        y_offset = int(y_offset*(2**ratio))
        totalx += x_offset
        totaly += y_offset

        print ratio,' level of gaussian pyramid offset is (',x_offset,',',y_offset,')'
        #now make the new_img1 aligned and update aligned_img1 to point to latest update
        aligned_img1 = roll(aligned_img1, x_offset, 1)
        aligned_img1 = roll(aligned_img1, y_offset, 0)

        img1=img1

    print 'total offset is (',totalx,',',totaly,')'

    return aligned_img1


#function to compute SSD offset of img1 onto img2, returns tuple (x,y)
def ssd_offset(img1, img2):
    print 'using ssd'
    w_height, w_width = img1.shape
    x_final = 0; y_final = 0
    min = Infinity
    for x in range(-10,11): #goes from -10 to 10
        for y in range(-10, 11):
            img1_copy = img1 #so we can reset the image displacement every iteration

            #translate images
            img1_copy = roll(img1_copy, x, 1)
            img1_copy = roll(img1_copy, y, 0)
```

```
            #calculate the ssd with respect to the overlapping area (prof said this was
                good)
            overlapping_area = (img1_copy.shape[0] - abs(x))*(img1_copy.shape[1] - abs(y))
95          ssd = sum((img2 - img1_copy)**2) - overlapping_area

            if ssd < min:
               min = ssd
               x_final = x
100            y_final = y
    print 'finished ssd'
    return (x_final, y_final)


105 def ncc_offset(img1, img2):
    print 'using ncc'
    w_height, w_width = img1.shape
    x_final = 0; y_final = 0
    max = -Infinity
110 for x in range(-10,11): #goes from -10 to 10
        for y in range(-10, 11):
            img1_copy = img1

            img1_copy = roll(img1_copy, x, 1)
115         img1_copy = roll(img1_copy, y, 0)

            a_prime = img2 - mean(img2)
            b_prime = img1_copy - mean(img1_copy)

120         dot = np.dot(a_prime.flatten(), b_prime.flatten())
            ncc = dot/(norm(a_prime.flatten())*norm(b_prime.flatten()))

            if ncc > max:

125             max = ncc
                x_final = x
                y_final = y
    print 'finished ncc'
    return (x_final, y_final)
130

#####################################################################

#TO USE THIS PROGRAM INPUT THE FILE NAME BELOW
#TO USE SSD CHANGE THE 'use_SSD' variable to > 0
135 #TO USE NCC CHANGE THE 'use_SSD' variable to <= 0

start_time = time.time() #for displaying time the program takes to run

image_name = '01657v.jpg'
140 use_SSD = 1 # >0 to use SSD, <= 0 to use NCC

i = imread(image_name)

if image_name[-4:] == '.png':
```

```
    i = i/float(max(i.flatten())) #change values to 0to1
else:
    i = i.astype(uint8) #convert the image to uint8 just in case

height, width = i.shape
height = height/3 # each r,g,b image's height

# #get rid of the common left and right borders of i, 5% should be enough
# i = i[:, width*0.05:-width*0.05]

#get the rgb channels from the photo, only need to manipulate rows
b_i = i[:height]
g_i = i[height:2*height]
r_i = i[2*height:3*height]

#update the height and width value to correspond with individual rgb imgs
height, width = b_i.shape

#now lets 'crop' each rgb image a little more, 10% should be enough
b_i = b_i[width*0.1:-width*0.1, height*0.1:-height*0.1]
g_i = g_i[width*0.1:-width*0.1, height*0.1:-height*0.1]
r_i = r_i[width*0.1:-width*0.1, height*0.1:-height*0.1]

# figure(2); imshow(b_i)
# figure(3); imshow(g_i)
# figure(4); imshow(r_i)
# show()

#aligned image
if image_name[-3:] == 'png':
    new_g_i = align_png(g_i, b_i, use_SSD)
    new_r_i = align_png(r_i, b_i, use_SSD)
else:
    new_g_i = align(g_i, b_i, use_SSD)
    new_r_i = align(r_i, b_i, use_SSD)

# figure(2); imshow(b_i)
# figure(3); imshow(g_i)
# figure(4); imshow(r_i)
# figure(5); imshow(b_i)
# figure(6); imshow(new_g_i)
# figure(7); imshow(new_r_i)
# show()



#combine the rgb images into one, the 3 is for adding the 3rd dimension
size = b_i.shape

if image_name[-4:] == '.png':
    result = zeros((size[0], size[1], 3)) #because png will operate on floats
else:
    result = zeros((size[0], size[1], 3)).astype(uint8)
```

```
result[:, :, 0] = new_r_i
result[:, :, 1] = new_g_i
result[:, :, 2] = b_i

#save the result and then show it
if use_SSD > 0:
    if image_name[-4:] == '.png':
        image_name = image_name[:-4] + '_SSD.png'
    else:
        image_name = image_name[:-4] + '_SSD.jpg'
else:
    if image_name[-4:] == '.png':
        image_name = image_name[:-4] + '_NCC.png'
    else:
        image_name = image_name[:-4] + '_NCC.jpg'

imsave(image_name, result)

print 'Algorithm took', -1*(start_time - time.time()),'to run'

figure(1); imshow(result)
show()
```

‘