# CS543/ECE549 Assignment 1

**Name:** Derek Yang

**NetId:** mcy3

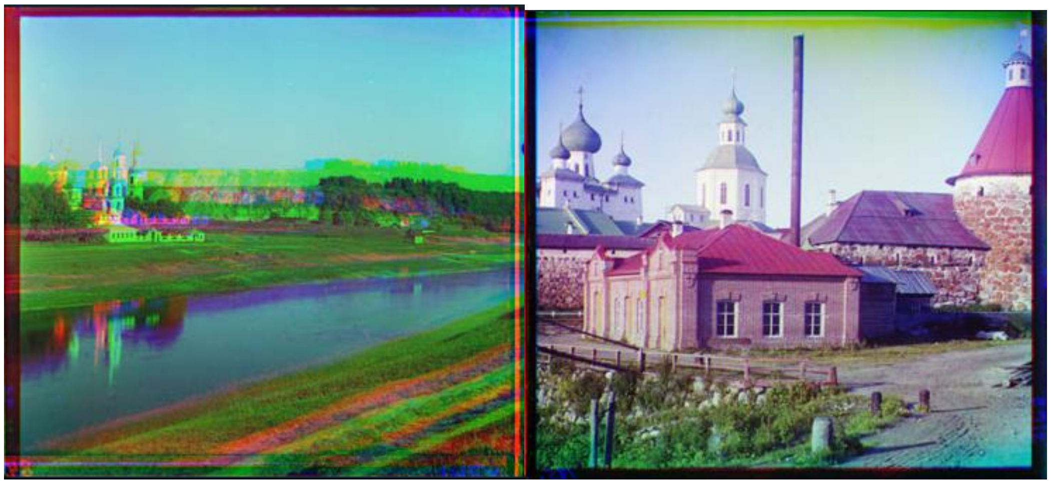
## Part 1 : Implementation Description

*All the time record is timed with jupyter notebook cell timer

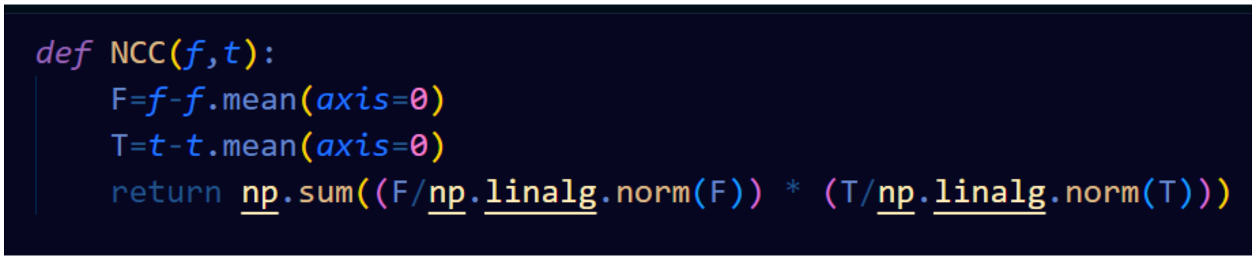
I cropped the white border by deleting any row/col with a mean value larger than 210.

For the alignment I implemented both *Sum of Square Difference* and Zero-*Normalized Cross Correlation,* after testing ZNCC yield a significantly better results overall so I end up using ZNCC for all the problem.

(a) the results of SSD

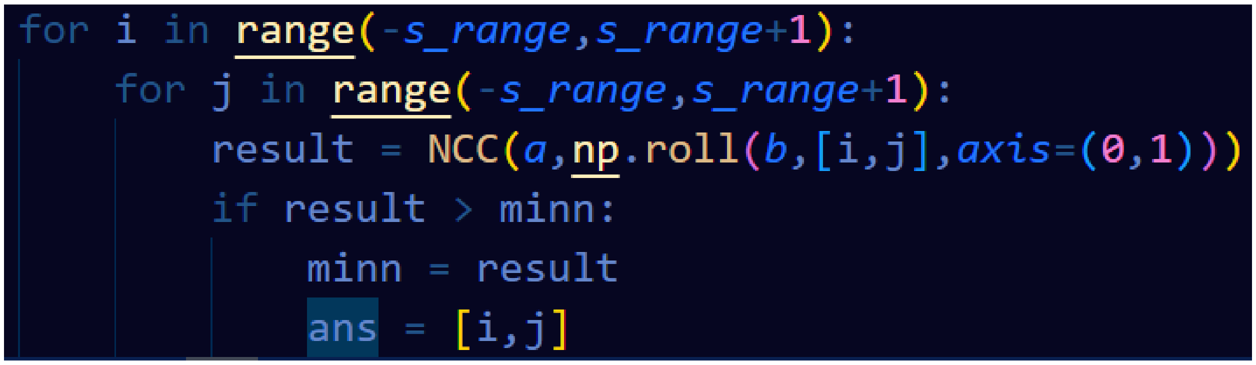


The ZNCC implementation is fairly simple with $< \frac{F}{||F||}, \frac{T}{||T||} >$



```
def NCC(f,t):
    F=f-f.mean(axis=0)
    T=t-t.mean(axis=0)
    return np.sum((F/np.linalg.norm(F)) * (T/np.linalg.norm(T)))
```

The basic aligning is done with looping for a given range, shifting the image with np.roll and calling the NCC function and save the offset for the best outcome.

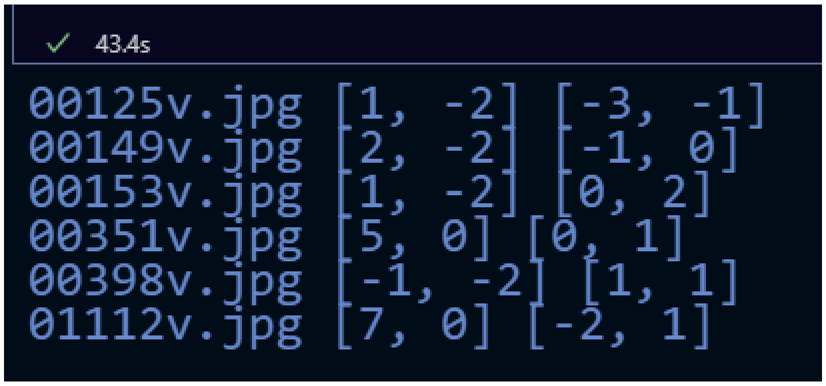


```python
for i in range(-s_range,s_range+1):
    for j in range(-s_range,s_range+1):
        result = NCC(a,np.roll(b,[i,j],axis=(0,1)))
        if result > minn:
            minn = result
            ans = [i,j]
```

After stacking different channels with the other two, setting green channel as the base channel yield the best result for all of the images especially “00153v.jpg”, my assumption is that the amount of blue in the picture is causing difficulties for the metric to compute the optimal offset.

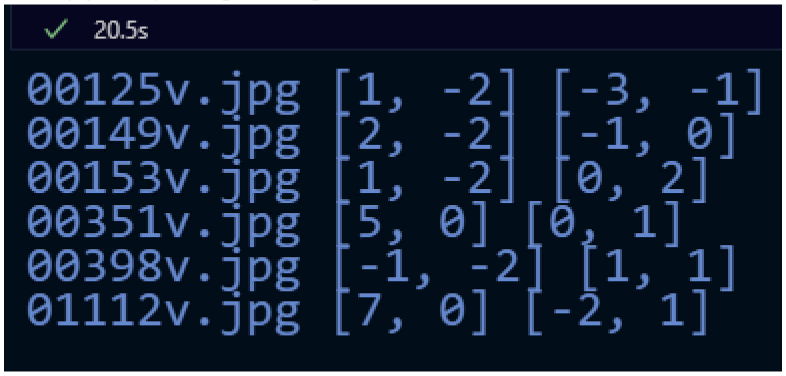
Using NCC means it results in a longer computation time comparing to other method since it needs to calculate the norm of all the two images for the given range. By lowering the offset range by 5 cuts the full computation time down to half.

(a) range of [-15,15]



```
✓ 43.4s
00125v.jpg [1, -2] [-3, -1]
00149v.jpg [2, -2] [-1, 0]
00153v.jpg [1, -2] [0, 2]
00351v.jpg [5, 0] [0, 1]
00398v.jpg [-1, -2] [1, 1]
01112v.jpg [7, 0] [-2, 1]
```

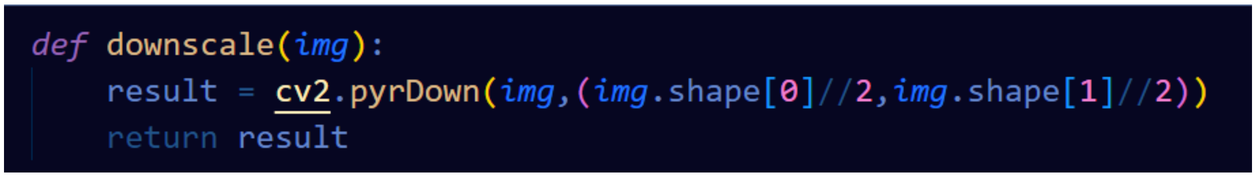
(b) range of [-10,10]



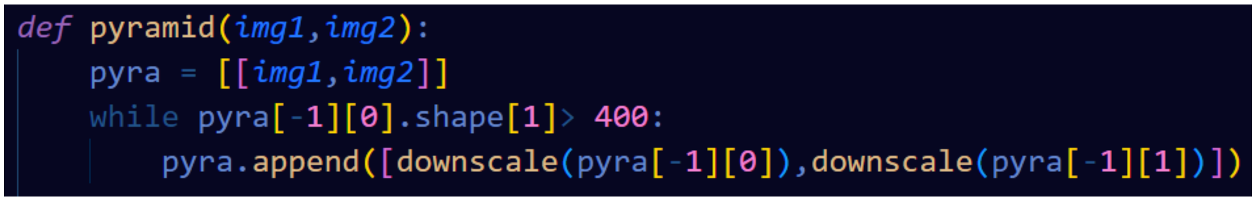
```
✓ 20.5s
00125v.jpg [1, -2] [-3, -1]
00149v.jpg [2, -2] [-1, 0]
00153v.jpg [1, -2] [0, 2]
00351v.jpg [5, 0] [0, 1]
00398v.jpg [-1, -2] [1, 1]
01112v.jpg [7, 0] [-2, 1]
```

The pyramid for larger images is done with an iterative approach, instead of a recursive one since it will be faster.

I scale the input images down by a factor of 2 using cv2.pyrDown until the image size is less than 400 and store all of them in a list

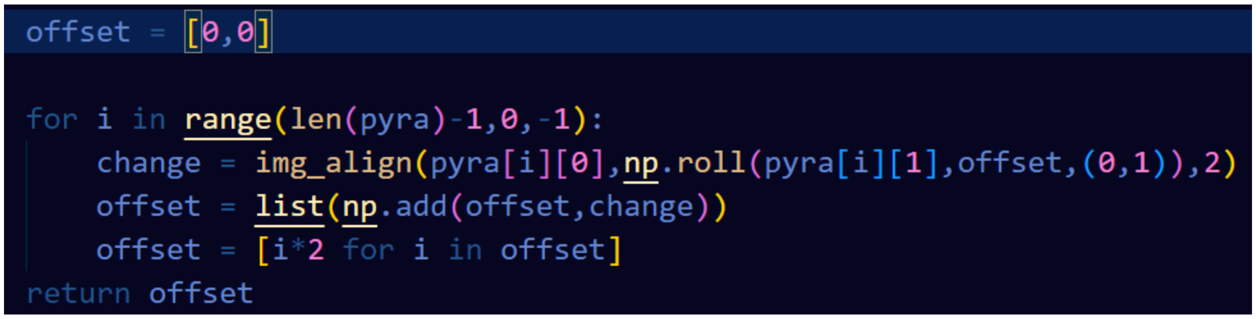


```
def downscale(img):
    result = cv2.pyrDown(img,(img.shape[0]//2,img.shape[1]//2))
    return result
```



```
def pyramid(img1,img2):
    pyra = [[img1,img2]]
    while pyra[-1][0].shape[1]> 400:
        pyra.append([downscale(pyra[-1][0]),downscale(pyra[-1][1])])
```

By calculating the offset of the downscale image, add to the current offset, multiply it by 2 and move upwards until the final image.



```
offset = [0,0]

for i in range(len(pyra)-1,0,-1):
    change = img_align(pyra[i][0],np.roll(pyra[i][1],offset,(0,1)),2)
    offset = list(np.add(offset,change))
    offset = [i*2 for i in offset]
return offset
```

## <u>Part 2: Basic Alignment Outputs</u>

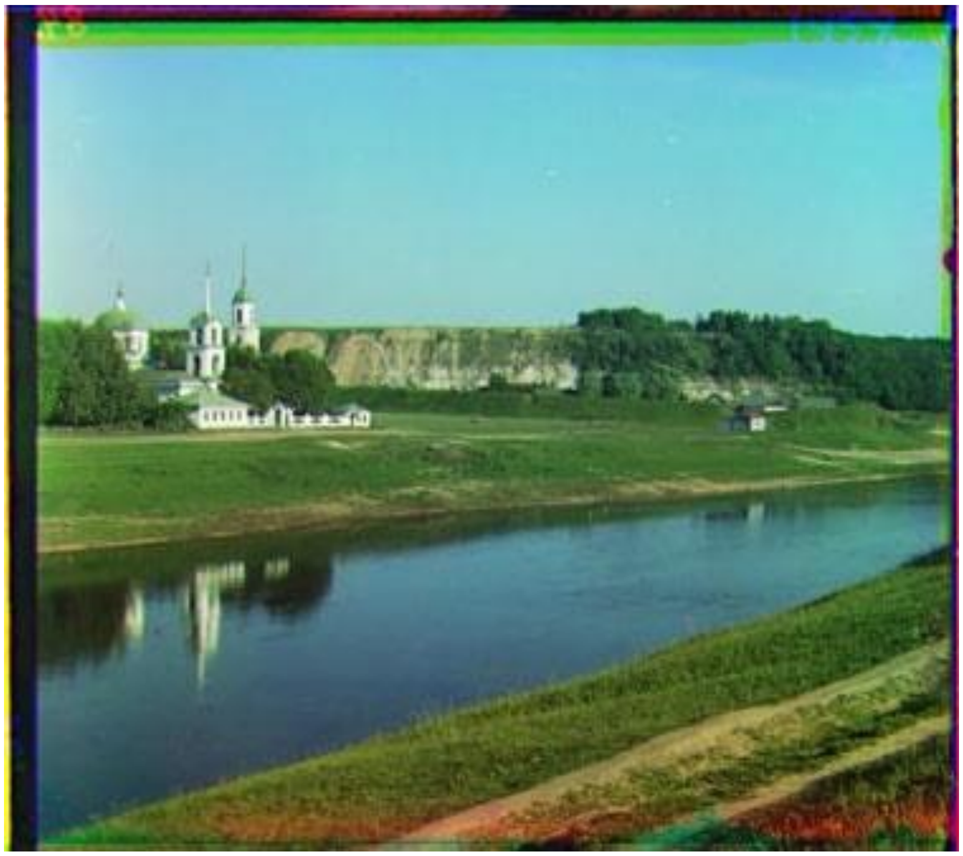
### A: Channel Offsets

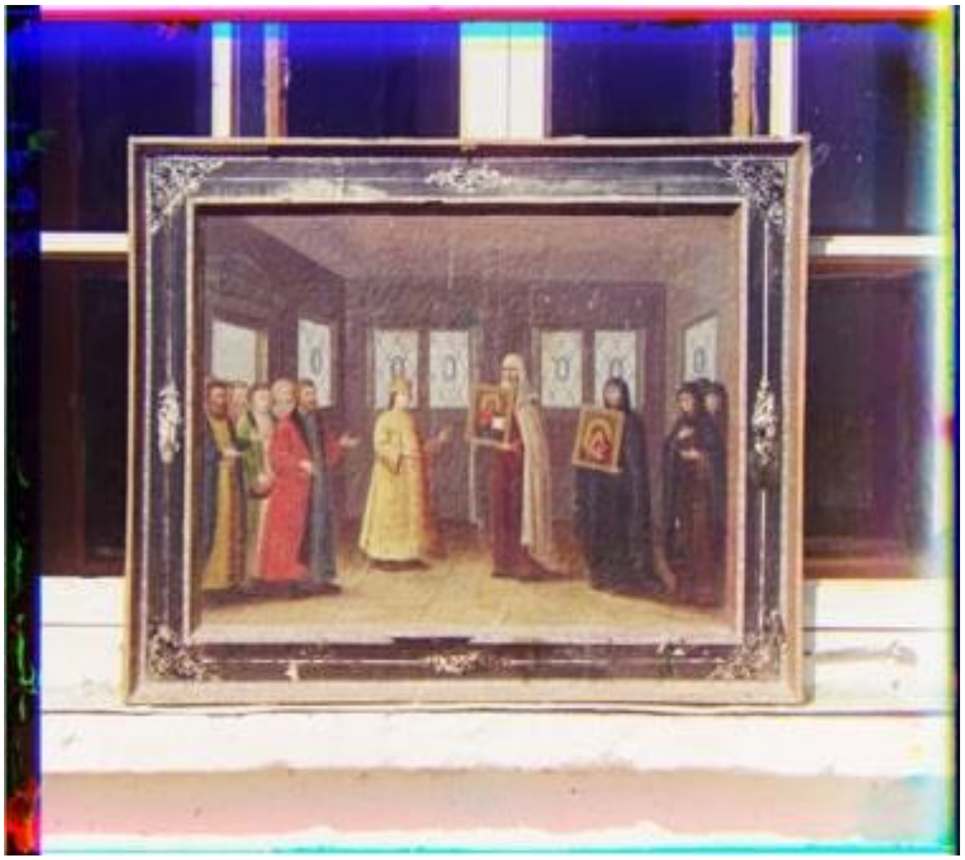
Using channel green as base channel:

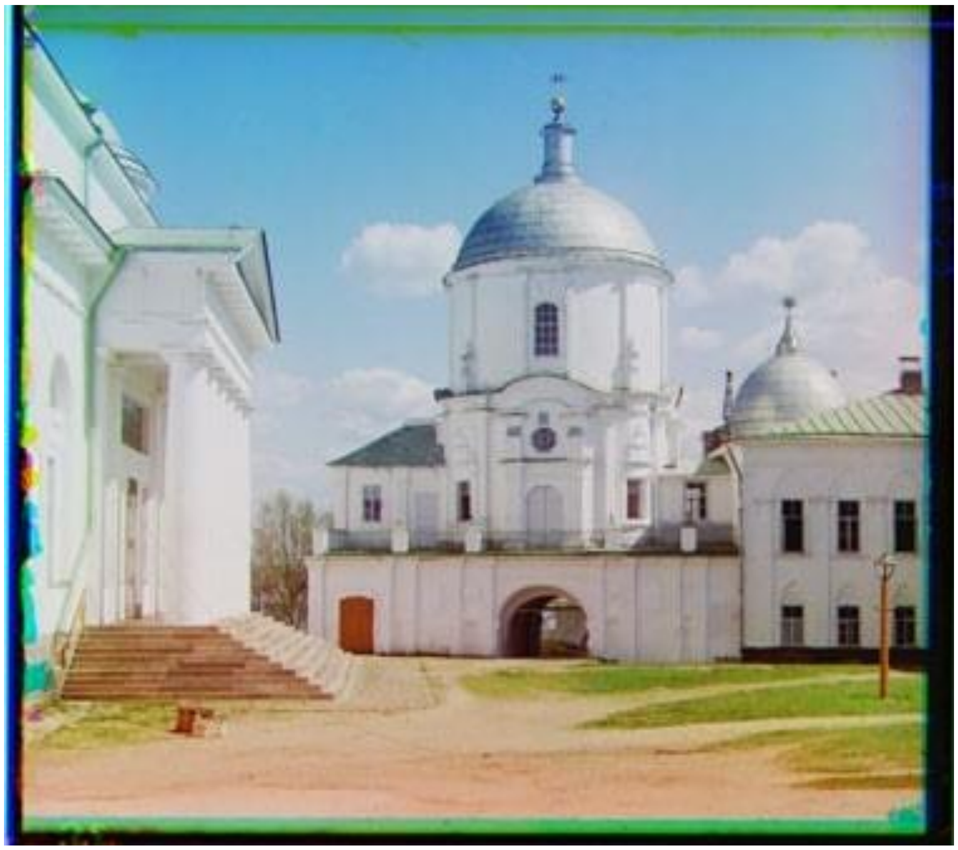
| Image | Blue (h,w) offset | Red (h,w) offset |
|---|---|---|
| 00125v.jpg | ( 1 , -2 ) | ( 3 , -1 ) |
| 00149v.jpg | ( 2 , -2 ) | ( -1 , 0 ) |
| 00153v.jpg | ( 1 , -2 ) | ( 0 , 2 ) |
| 00351v.jpg | ( 5 , 0 ) | ( 0 , 1 ) |
| 00398v.jpg | ( -1 , -2 ) | ( 1 , 1 ) |

|  |  |  |
|---|---|---|
| 01112v.jpg | ( 7 , 0 ) | ( -2 , 1 ) |

**B: Output Images**







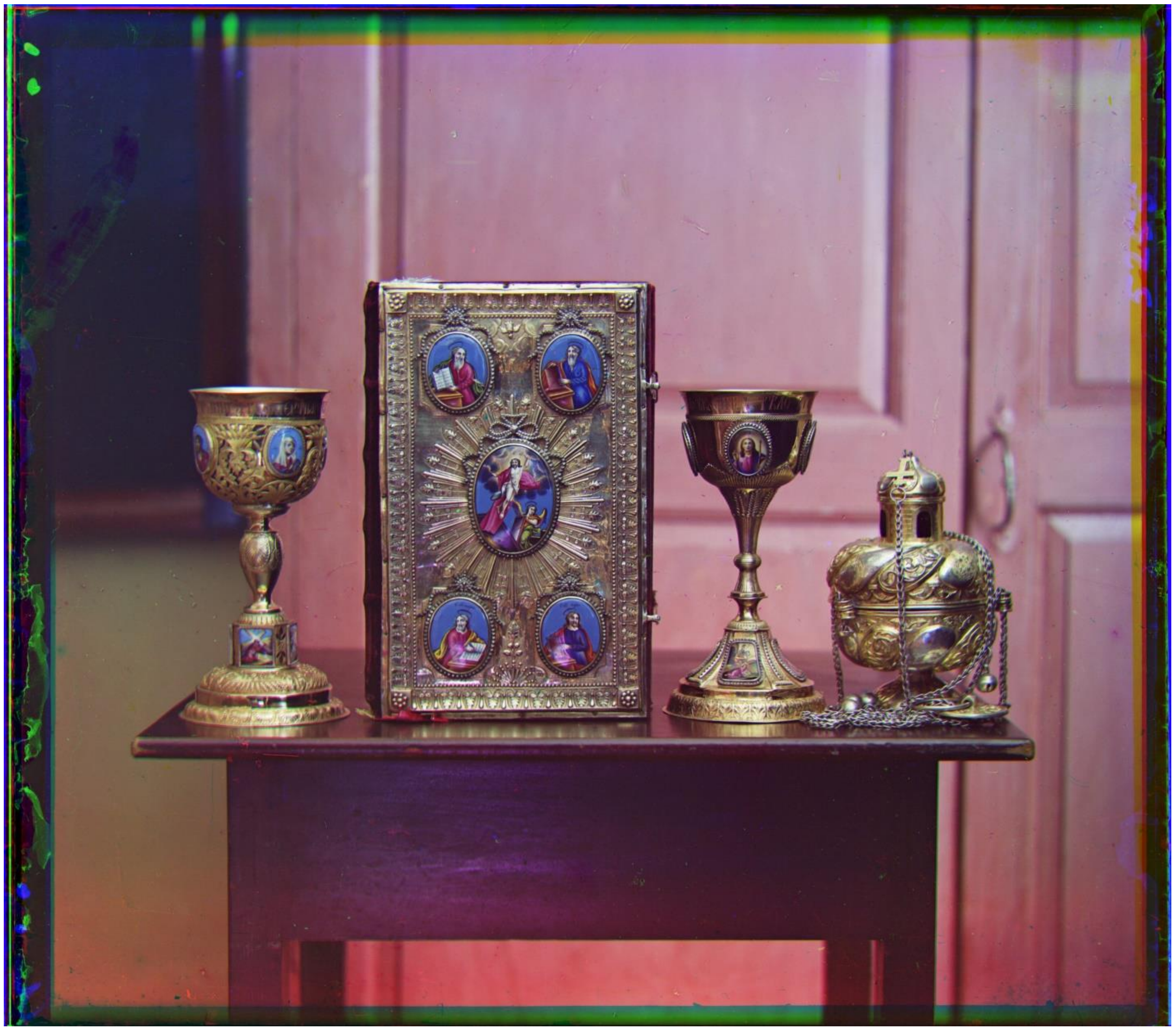
## Part 3: Multiscale Alignment Outputs

### A: Channel Offsets

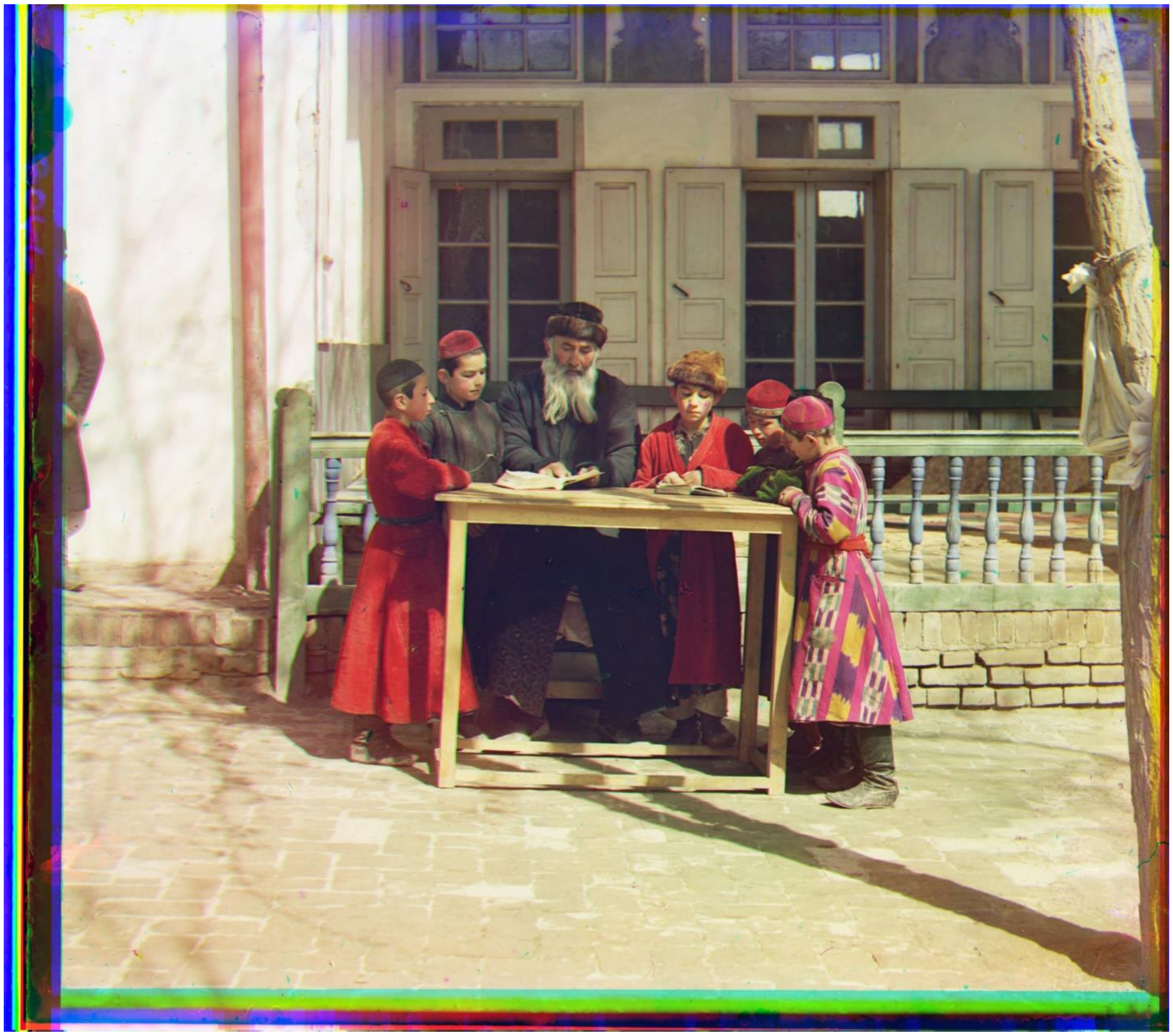
Using channel Blue as base channel:

| Image | Green (h,w) offset | Red (h,w) offset |
|---|---|---|
| 01047u.tif | ( -4 , 20 ) | ( 16 , 34 ) |
| 01657u.tif | ( -16 , 6 ) | ( -32 ,12 ) |
| 01861a.tif | ( -6 , 40 ) | ( -6 , 60 ) |

## B: Output Images



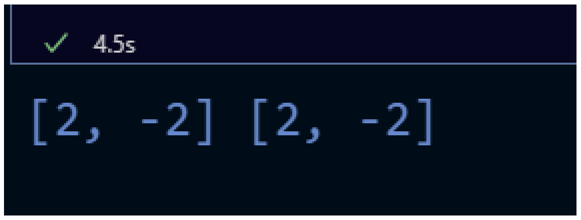
7293



**C: Multiscale Running Time improvement**

*All the time record is timed with jupyter notebook cell timer

time spent on range 2 with no pyramid on one image



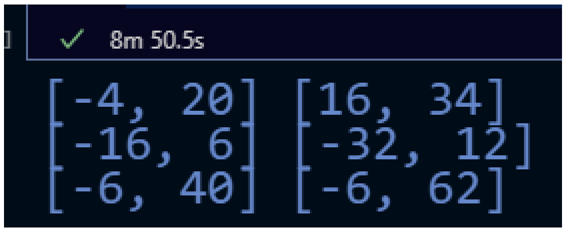
```
✓ 4.5s

[2, -2] [2, -2]
```

4(range for horizontal) * 4 (range for vertical) * 2 (2 channels) * 1 (1 image) = 32
Which means 32 matches cost 4.5 seconds so on average one match costs 0.140625 second.
The most offset on the three pictures is 60, the amount of matches need to be done on it will be

86400 matches (120*120*2* 3). It will at least take 3 hours and 22 minutes to compute 3 images with no pyramid when knowing the result.

By doing the image scaling, even with range 15 with 3 images can be done in under 9 minutes. Which is a 99.92% decrease.

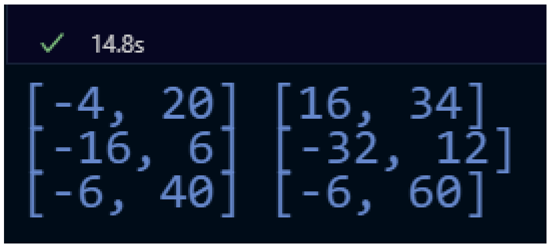
time spent on range 15 with pyramid on 3 images



```
✓ 8m 50.5s
[-4, 20] [16, 34]
[-16, 6] [-32, 12]
[-6, 40] [-6, 62]
```

We can further decrease the time spent by lowering the range on the alignment

time spent on range 2 with pyramid on 3 images

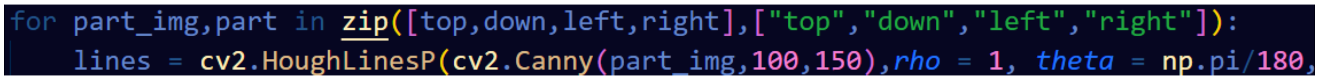


```
✓ 14.8s
[-4, 20] [16, 34]
[-16, 6] [-32, 12]
[-6, 40] [-6, 60]
```

## **<u>Part 4 : Bonus Improvements</u>**

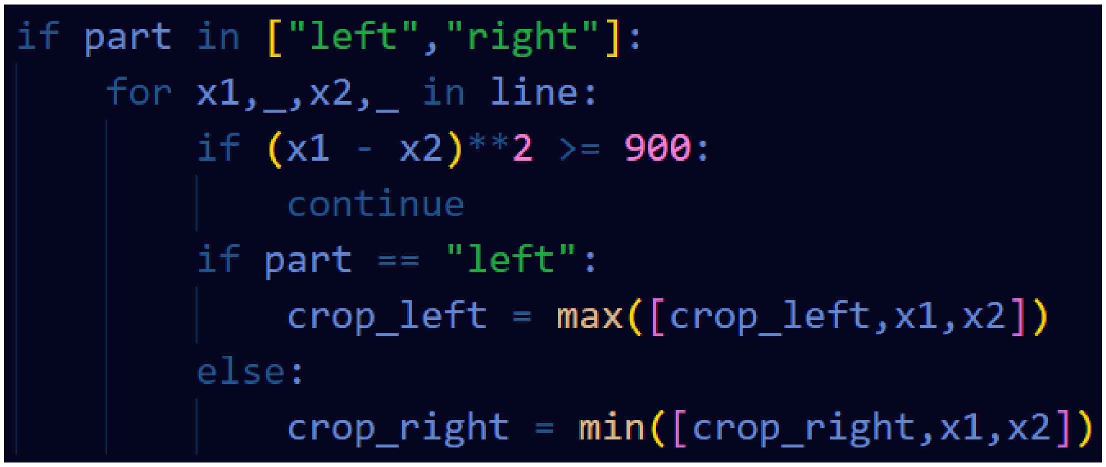
I implemented an auto crop for all the strange colors on the side of the picture caused by aligning the channels with Probabilistic Hough Transform and Canny edge detection.

I took 10% image from all sides, went through cv2.Canny and cv2.HoughLinesP to find the possible edge on 4 image sides.

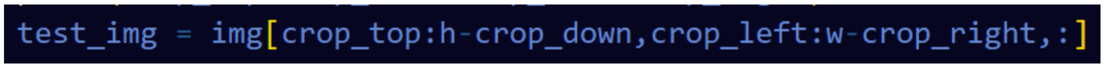


```
for part_img,part in zip([top,down,left,right],["top","down","left","right"]):
    lines = cv2.HoughLinesP(cv2.Canny(part_img,100,150),rho = 1, theta = np.pi/180,
```

Find the closest line towards the center for 4 sides and crop the image with the value with it.



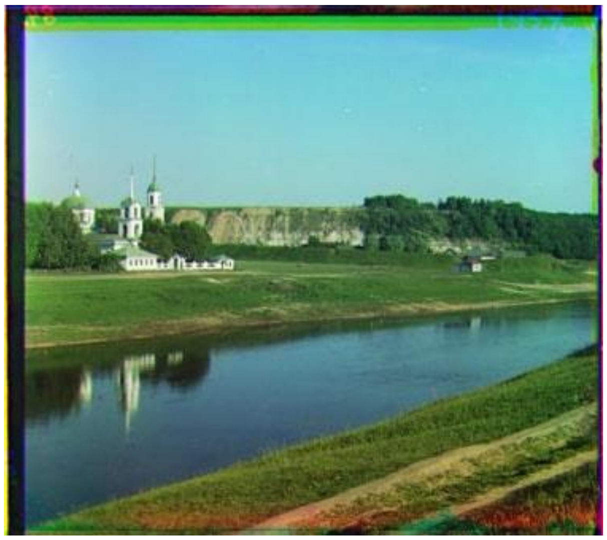
```
if part in ["left","right"]:
    for x1,_,x2,_ in line:
        if (x1 - x2)**2 >= 900:
            continue
        if part == "left":
            crop_left = max([crop_left,x1,x2])
        else:
            crop_right = min([crop_right,x1,x2])
```



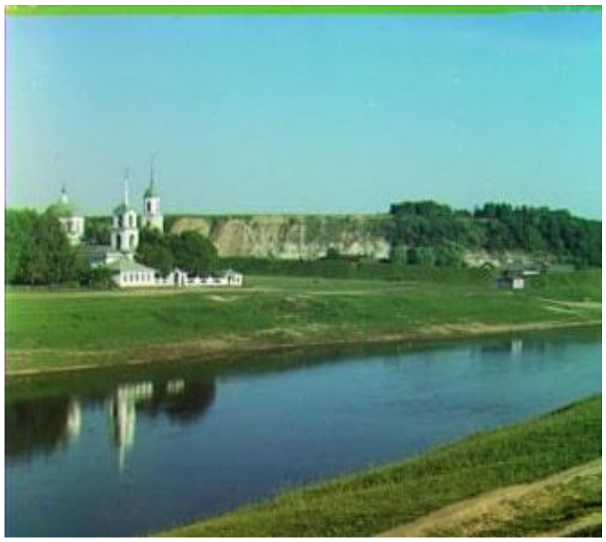
```
test_img = img[crop_top:h-crop_down,crop_left:w-crop_right,:]
```

Some parameters might still need some fine tuning as you can see some borders can’t be completely cropped, but generally works well.

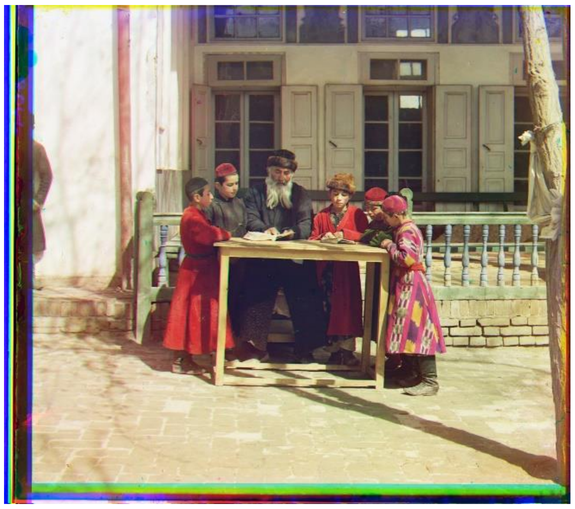
original image



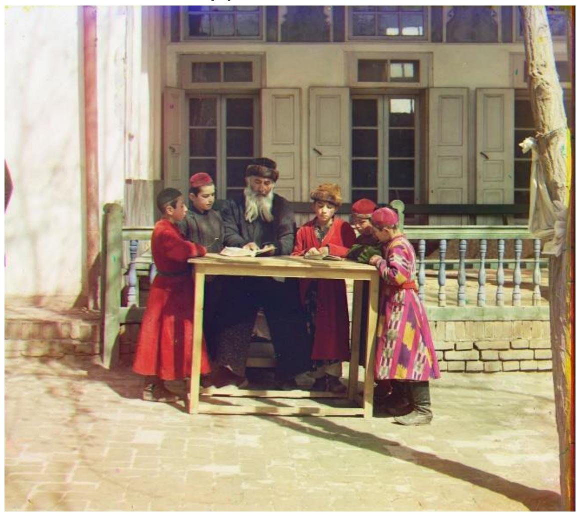
cropped



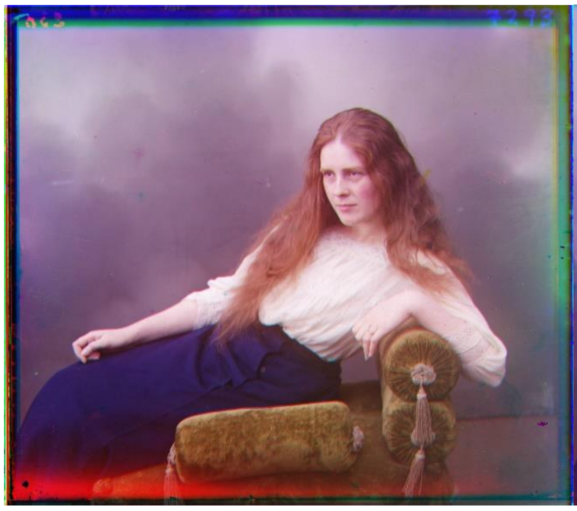
original image



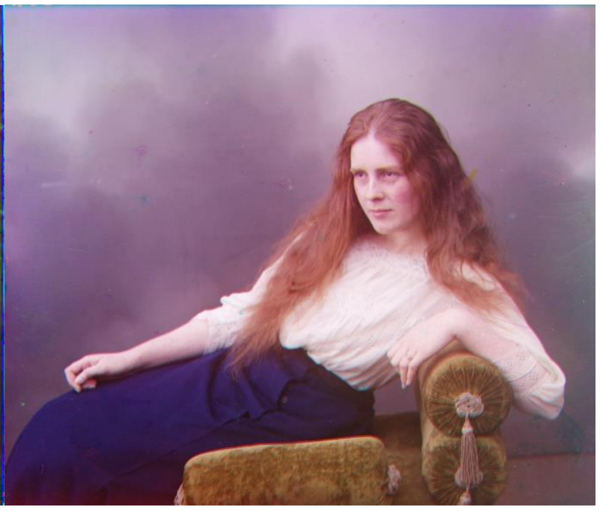
cropped



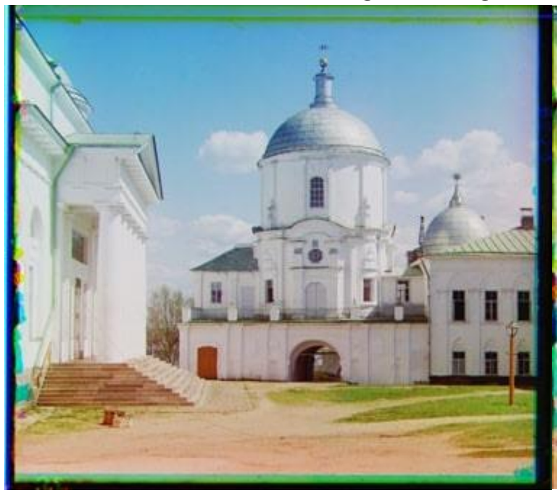
original image



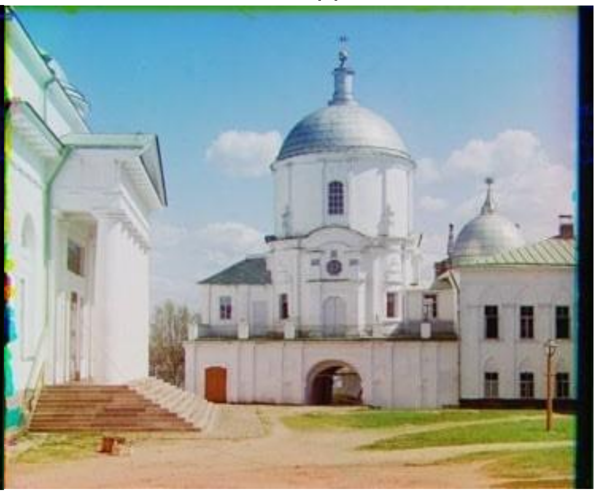
cropped



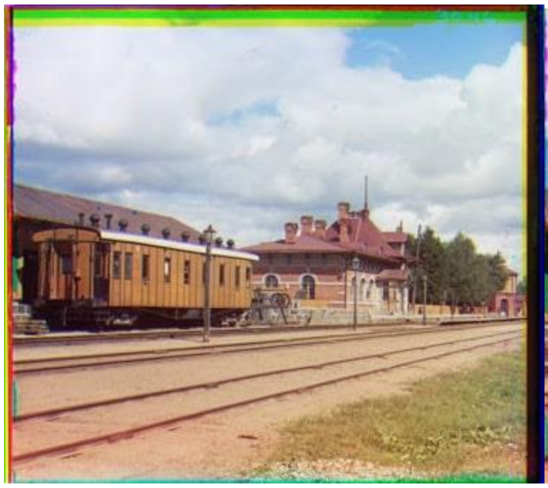
original image



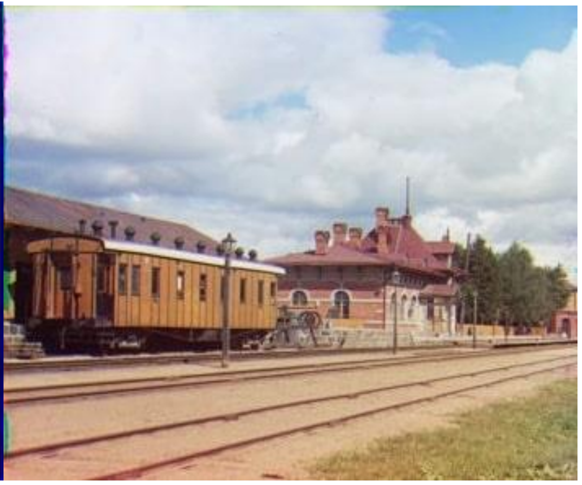
cropped



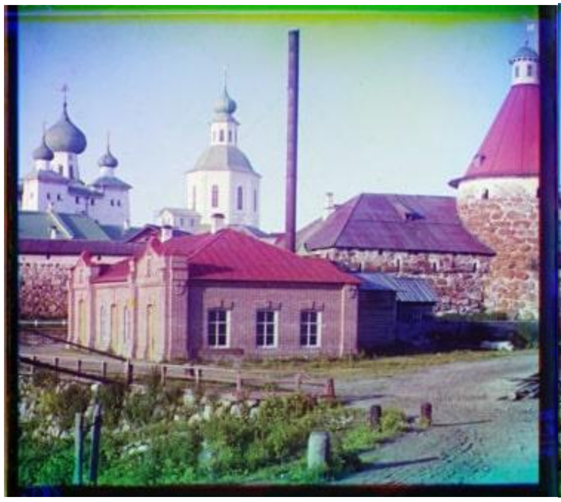
original image



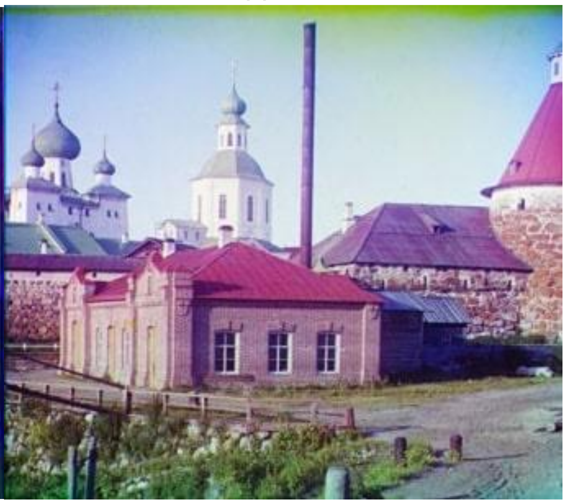
cropped



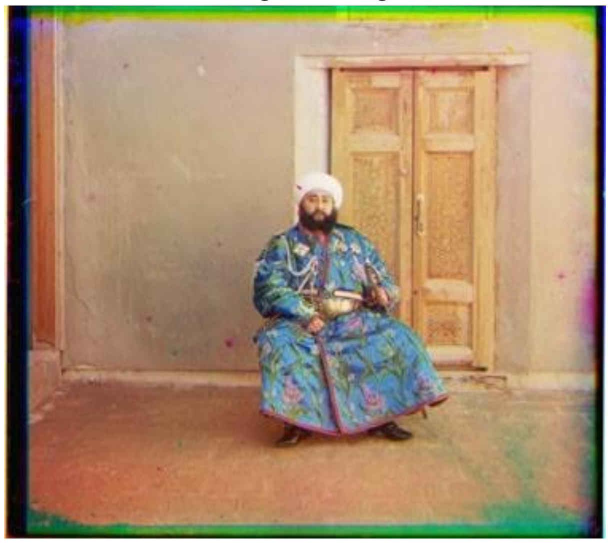
original image



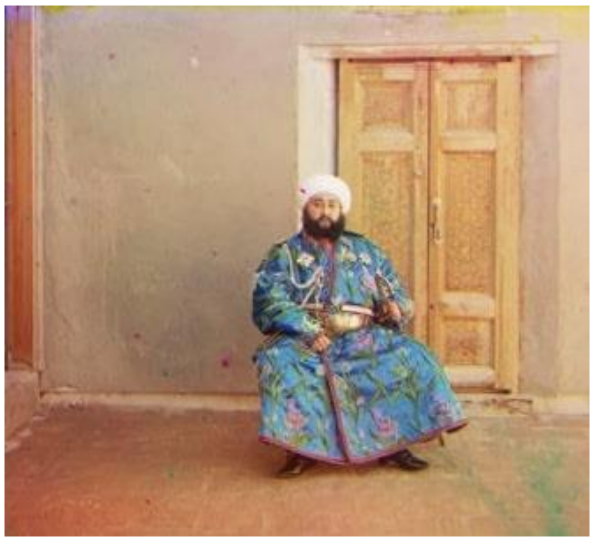
cropped



original image



cropped

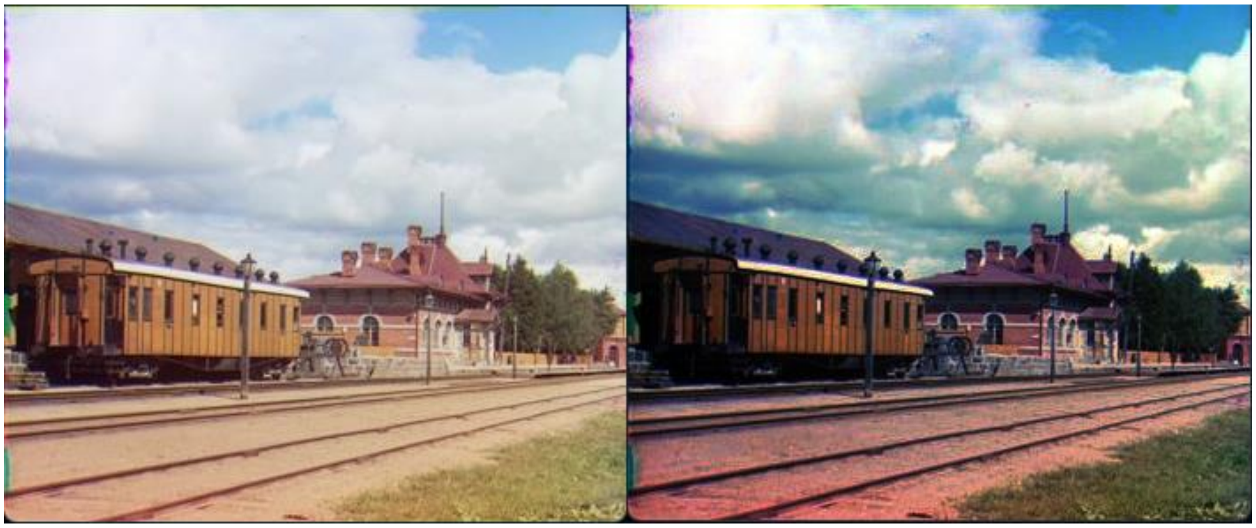


original image

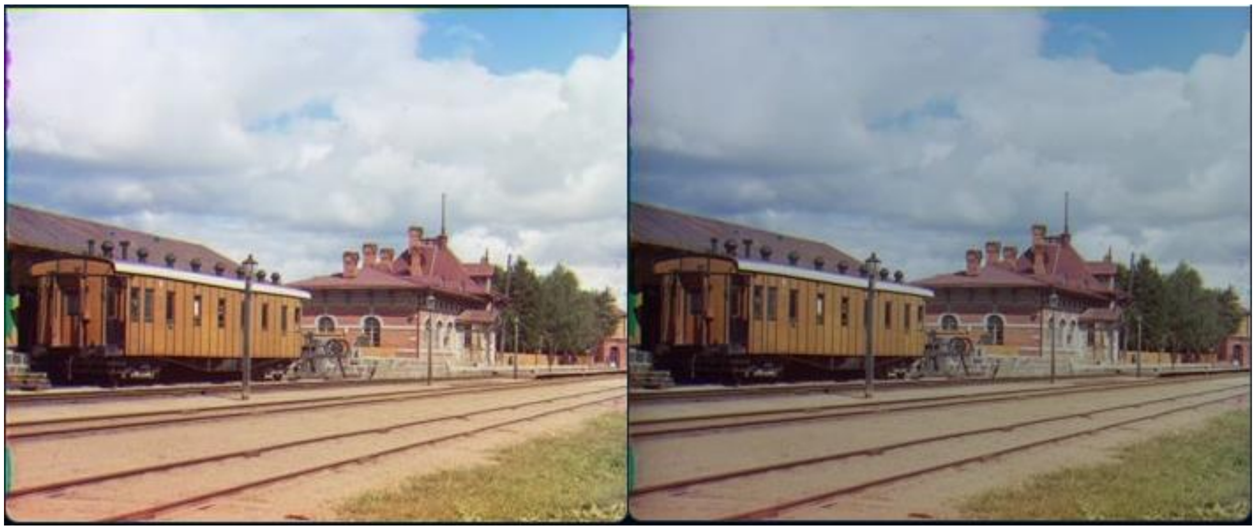
cropped

I’ve also implemented gamma correction, grey world white balcancing and image equalization to try improve the image result. However none of it seems to actually improve it too well.

Equilization

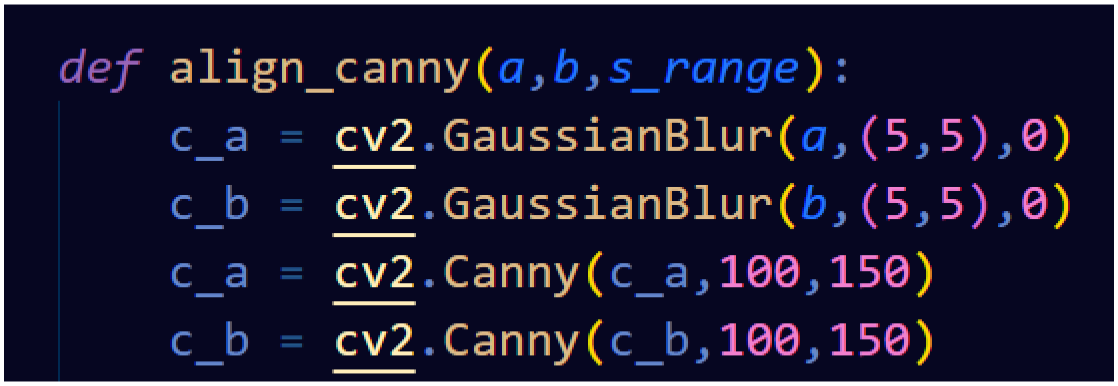


White Balancing



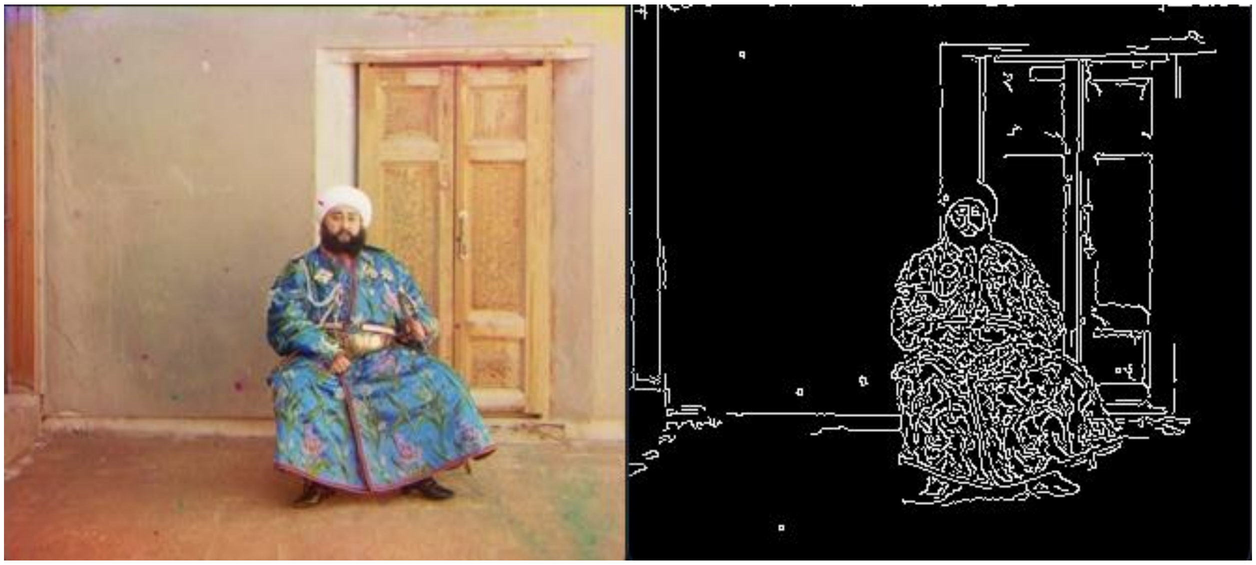
The Canny edge detection can also help with image aligning. The problem with the image “00153v” is that the pixel value has large difference between channels, by doing edge detection and match with only the edge mask image eliminates the brightness difference and focus the matching into aligning shapes.

By doing Gaussian Blur before edge detection, helps the metric to focus on the significant edges and not the small edges.



```
def align_canny(a,b,s_range):
    c_a = cv2.GaussianBlur(a,(5,5),0)
    c_b = cv2.GaussianBlur(b,(5,5),0)
    c_a = cv2.Canny(c_a,100,150)
    c_b = cv2.Canny(c_b,100,150)
```

Without Blur



With Blur