EECS 442: Computer Vision

HW1: Color Images

Chuan Cen / Jan 20th 2018

Part I: Coloring Prokudin-Gorskii images

● Basic Implementation: Dot Product

In our basic implementation, we use the sum of dot product of two layers as the only metric to decide the shift. The algorithm searched over displacements in the range [-15, 15] both horizontally and vertically and pick the shift that maximize the dot product value among 961 shifts (=31*31 shifts).

This basic implementation was not stable as there's a chance that *ip.jpg* and *candy.jpg* fail to be matched correctly. It was largely improved once the "gradient domain alignment" technique was implemented. See below.

● **Extra credit 1:** Gradient domain alignment

Use *edge* function gave large improvements. But we noticed that *candy.jpg* could fail with some chance if we only use the *edge* function's output to do alignment. Finally we came up with an idea that we first combine (i.e. add) the output of *edge* with the original image and then do the alignment. By doing this, we got the right results 100%.

Specifically, the key line of code that do the combination is as follows:

```
for i=1:3
    im(:,:,i) = (edge(im(:,:,i),'canny'))+
  (edge(im(:,:,i),'prewitt'))*0.8+(edge(im(:,:,i),'log'))*0.6;
end
```

Note that we use two edge functions and the original image together with different weight on them. This is the best combination we found after tons of experiments.

● **Extra credit 2:** Faster alignment

A coarse-to-fine-style faster alignment was implemented. The steps are as follows:

1. **Coarse alignment**: We first resize the image to the quarter of original size, make an alignment and obtain a shift result.

2. **Fine alignment**: The shift result at Coarse alignment step is then used as the the input of the Fine alignment which align two full size images(layers to be specific), but with smaller search region.

In other words, the first alignment gives an estimate shift and the second search in a small region under the guide of the first result.

By doing this we got 2.58X speed-up. Detailed experiment data is shown in the "Result" section below.

● **Extra credit 3:** Boundary effects

The algorithm we used below cut the ugly boundaries automatically

1. Transform the original RGB image into gray scale by using function *rgb2gray* so that a single layer image is obtained.

2. Cut off 4 grayscale border regions (top, bottom, left, right) which are the pixels within 10% of image height/width of the edge.

3. For each border region, use *edge* function with "Canny" method.

4. Do dot product between a white line (an all-one array) and every column(if it's left&right) or row(if top&bottom). This step produce an array of dot product values.

5. Select out those columes/rows whose dot product value is larger than a specific threshold, and then pick up the one that is closest to the center of the image.

In "Result" section we give the output of this boundary-cut algorithm on Prokudin-Gorskii image set.

- Results

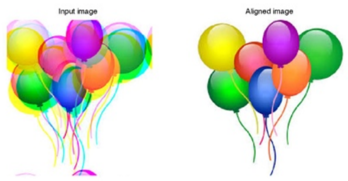
Basic outputs

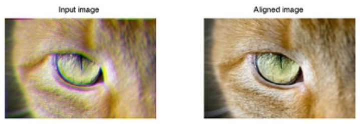
The ouput of 10 test images and 6 Prokudin-Gorskii images are shown below.

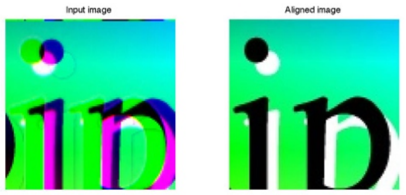
- Output of test images

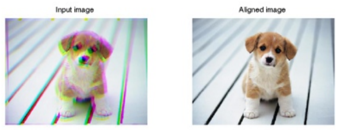
> - <u>Code-running Instruction:</u>
>
> 1. Run the script “evalAlignment”.
> 2. If you want to run the faster version, change the function call “alignChannels” into the function “alignChannels_fst” in the script “evalAlignment”.

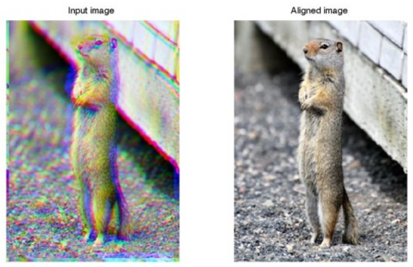
```
>> evalAlignment
Evaluating alignment ..
 1 balloon.jpeg
        gt shift: ( 4,-7) (-12, 1)
      pred shift: (-4, 7) (12,-1)
 2 cat.jpg
        gt shift: (14,-11) (14,15)
      pred shift: (-14,11) (-14,-15)
 3 ip.jpg
        gt shift: (14, 9) ( 0,-11)
      pred shift: (-14,-9) ( 0,11)
 4 puppy.jpg
        gt shift: (-2, 9) (13,14)
      pred shift: ( 2,-9) (-13,-14)
 5 squirrel.jpg
        gt shift: ( 5,11) (-14,13)
      pred shift: (-5,-11) (14,-13)
 6 candy.jpeg
        gt shift: ( 6, 8) ( 8,-3)
      pred shift: (-6,-8) (-8, 3)
 7 house.png
        gt shift: ( 5, 6) (-10,-15)
      pred shift: (-5,-6) (10,15)
 8 light.png
        gt shift: (-7,-12) (-14,10)
      pred shift: ( 7,12) (14,-10)
 9 sails.png
        gt shift: ( 6,14) (-6,-14)
      pred shift: (-6,-14) ( 6,14)
10 tree.jpeg
        gt shift: (-2, 8) (-4, 9)
      pred shift: ( 2,-8) ( 4,-9)
```

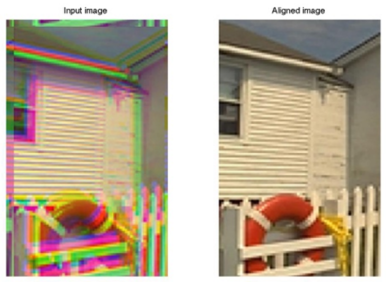


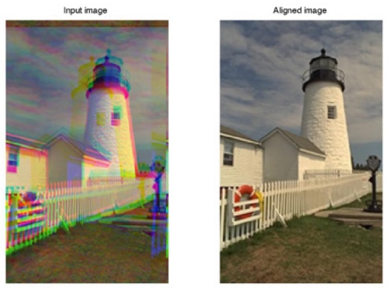


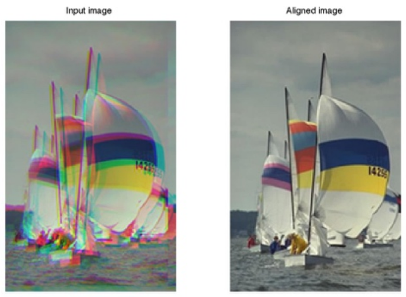


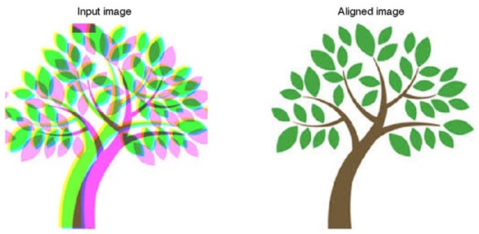










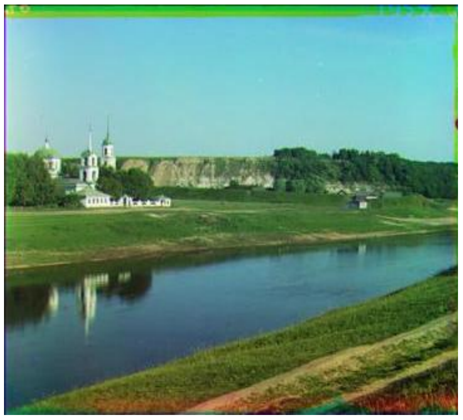
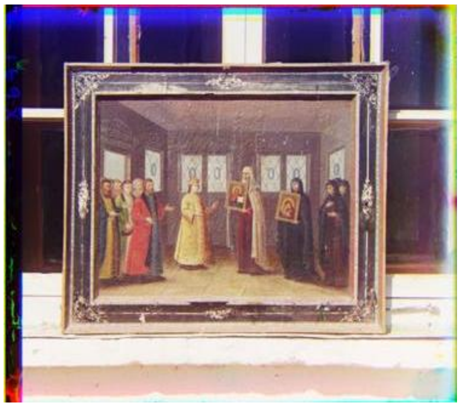
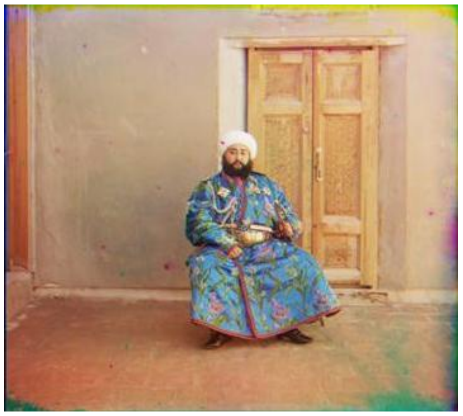
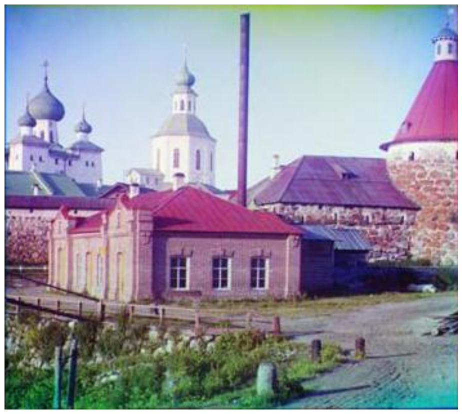
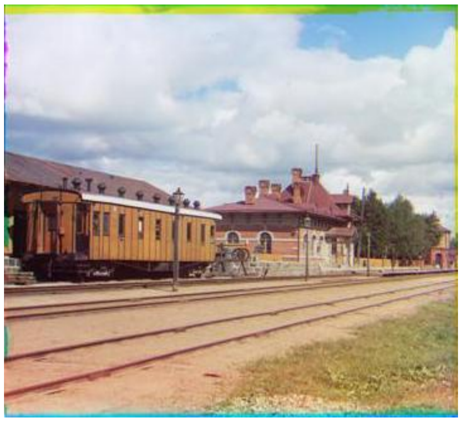
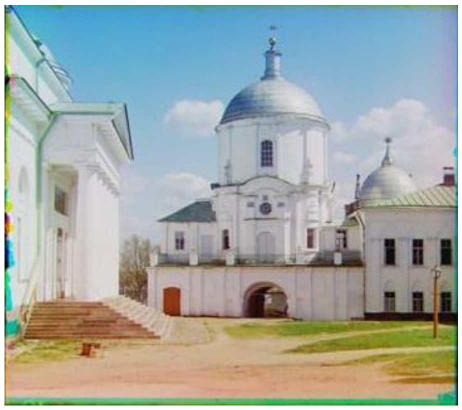


➢ Output of Prokudin-Gorskii images

➢ <u>Code-running Instruction:</u>

1. Run the script "alginProkudin" first.
2. Then run the script "cutBoundaryScript" to reduce the boundary effects. The output images are saved in the directory "../output/prokudin-gorskii-cutedge". Please make sure this directory exists before you run.

#		B Shift	R Shift
1	**00125v.jpg**	(-4, 1)	(-10, -1)
2	**00153v.jpg**	(-7, -1)	(-14, -3)
3	**00398v.jpg**	(-6, -1)	(-11, -4)
4	**00149v.jpg**	(-5, 0)	(-9, -2)
5	**00351v.jpg**	(-9, -1)	(-13, -1)
6	**01112v.jpg**	(-5, -1)	(-5, -1)

◆ **Faster alignment performance**

➢ Code-running Instruction:

1. Change the function call "alignChannels" into "alignChannels_fst" within the script file "evalAlignment.m"

2. Then run the script "evalAlignment".

The comparison between basic implementation and faster version is shown below.

	Basic Implementation	**Faster Implementation**
Balloon	0.943	0.3584
Cat	1.0127	0.3718
Ip	0.2165	0.1505
Puppy	1.0927	0.3719
Squirrel	1.0788	0.3763
Candy	0.9094	0.3436
House	0.5589	0.2525
Light	0.5569	0.2491
Sails	0.5554	0.2332
Tree	0.9419	0.3379
Average	0.7866	0.3045

From above we can see that the faster version achieved 2.58(=0.7866/0.3945) times speed-up.

Notice that we can choose a larger "scale ratio" to gain a much larger speed-up. Now the scale ratio for the faster implementation is 2, which means the coarse alignment phase resize the image into the one with 1/2 height and 1/2 width. If, say, we choose the scale ratio to be 3, the speed-up in theory can be up to 8 instead. But the problem to increase the scale ratio is that the correctness of the output will decrease. With scale ratio 2 we can get correct output on the 10 test images with approximately 95% chance.

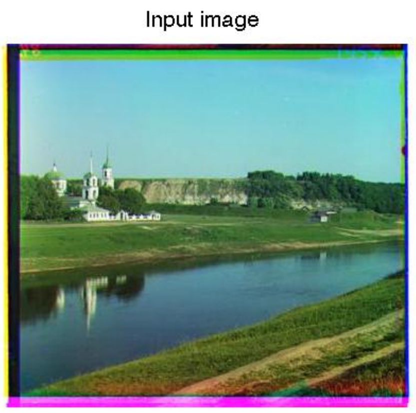
The reason why coarse-to-fine style faster implementation will decrease the correctness is that, when at coarse phase, a lot of information of the image has lost so the alignment result at this phase can't deliver a reliable estimate to the fine phase(to make a further searching). We believe scale ratio 2 is a good trade off between correctness and speed.

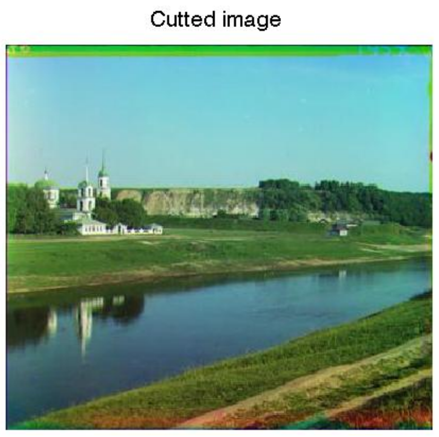
■ Boundary effects

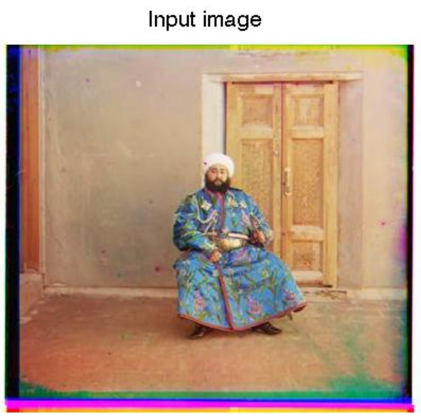
➢ <u>Code-running Instruction:</u>

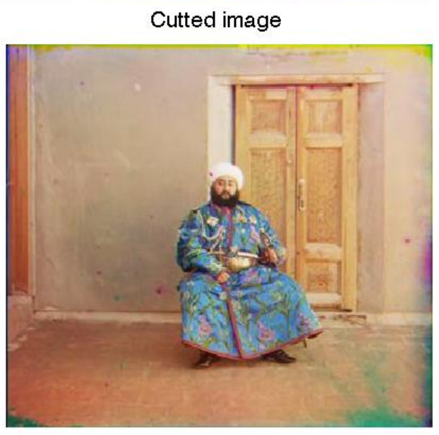
1. Run the script "alginProkudin" first.
2. Then run the script "cutBoundaryScript" to reduce the boundary effects. The output images are saved in the directory "../output/prokudin-gorskii-cutedge". Please make sure this directory exists before you run.

The comparison between "aligned image" and "aligned and cutted image" is shown below:

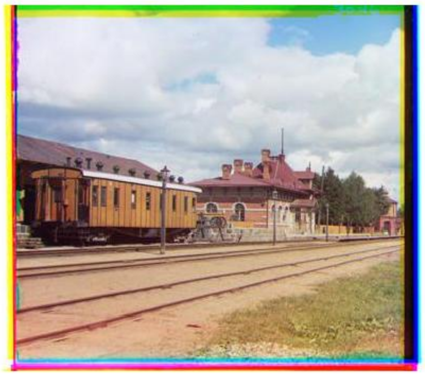




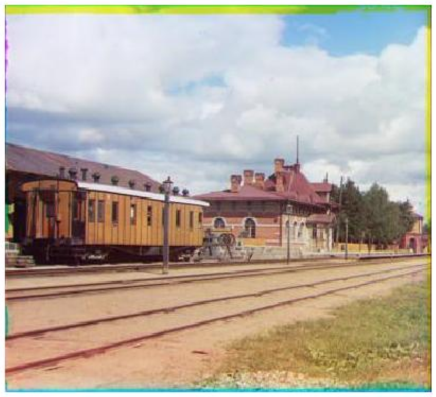




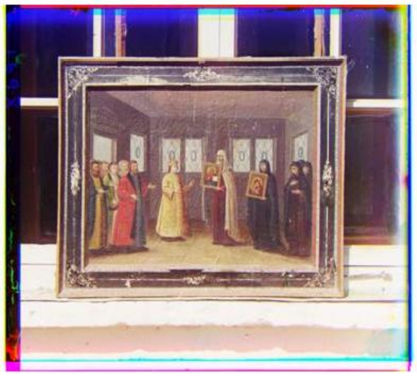
Input image



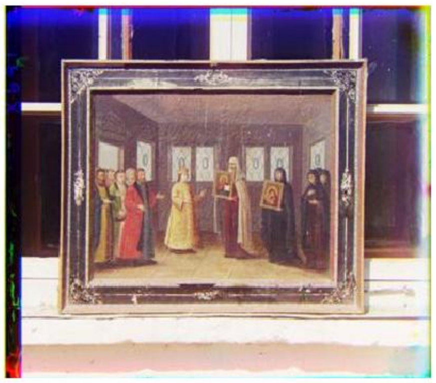
Cutted image



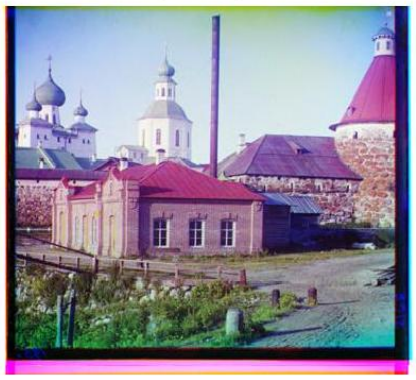
Input image



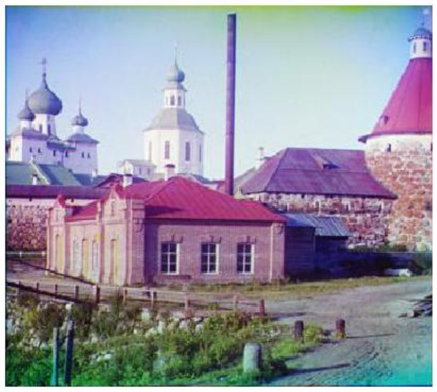
Cutted image



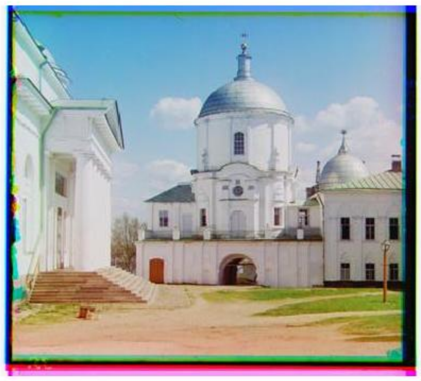
Input image



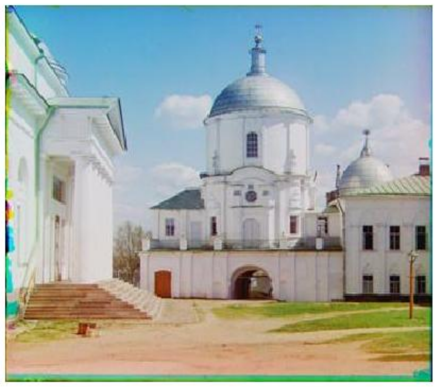
Cutted image



Input image



Cutted image



From above we can see the boundary-cut algorithm works really well. 5 images out of the 6 got near-perfect results except for the 4th image that still has the left border uncut. This exception may be due to the fusion of the dark part of the original image and the dark border.

Part II: Color image demosaicing

● Basic Implementation

■ Methods Explanation

◆ Nearest Neighbor

1. Green Layer:
   a) Every empty pixel that corresponds to "red" location get the pixel value from the right.

   b) Every empty pixel that corresponds to "blue" location get the pixel value from the left.
2. Red Layer:
   a) First, every empty pixel gets the value from the left. (even if the left is empty).

   b) Second, every empty pixel gets the value from the above.
3. Blue Layer:
   a) First, every empty pixel gets the value from the right. (even if the left is empty).

   b) Second, every empty pixel gets the value from below row.

The border and corner has to be handled after doing all the above steps.

◆ Linear Interpolation

1. Green Layer: Every empty pixel be set as the mean value of its top, bottom, left and right pixels.
2. Red Layer:
    a) First, for those empty pixel that has two sides non-zero, set them as the mean value of the two sides.
    b) Second, every empty pixel remained in the layer be set as the mean value of its top, bottom, left and right pixels.
3. Blue Layer: follow the same steps of Red Layer.

◆ Adaptive Gradient

The adaptive gradient method is basically to first compare the two differences of top/bottom and left/right. If the difference of top/bottom pixels is larger than left/right, then set the empty pixel as the mean value of the left and right pixels, and vise-versa. But for the red and blue layer we can't use this method directly because some empty pixels only have two sides or even no side filled with value. So for red and blue layer we need to first take the step which is the same as 2.a) in Linear Interpolation, then do the adaptive gradient interpolation.

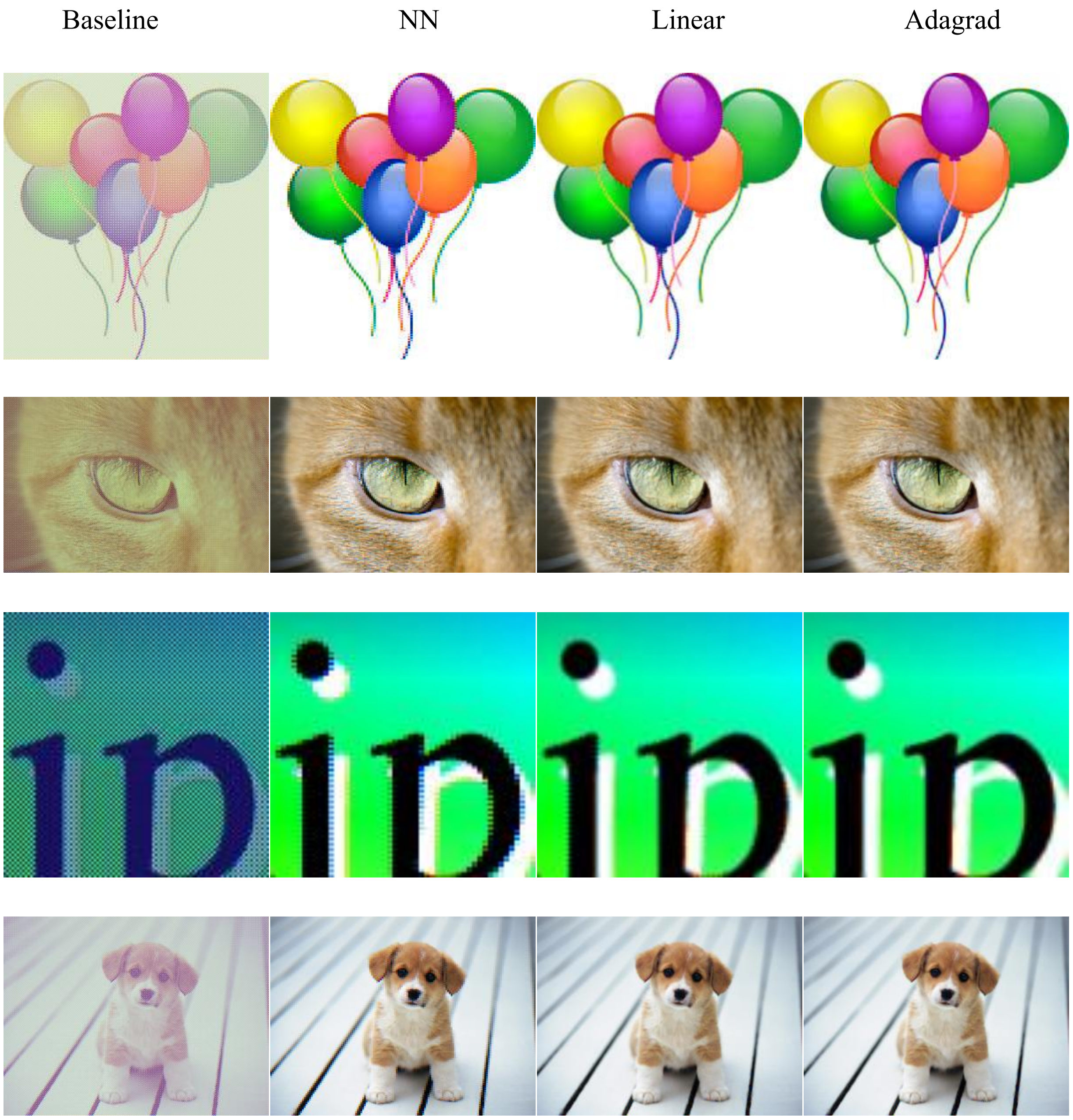
■ Results

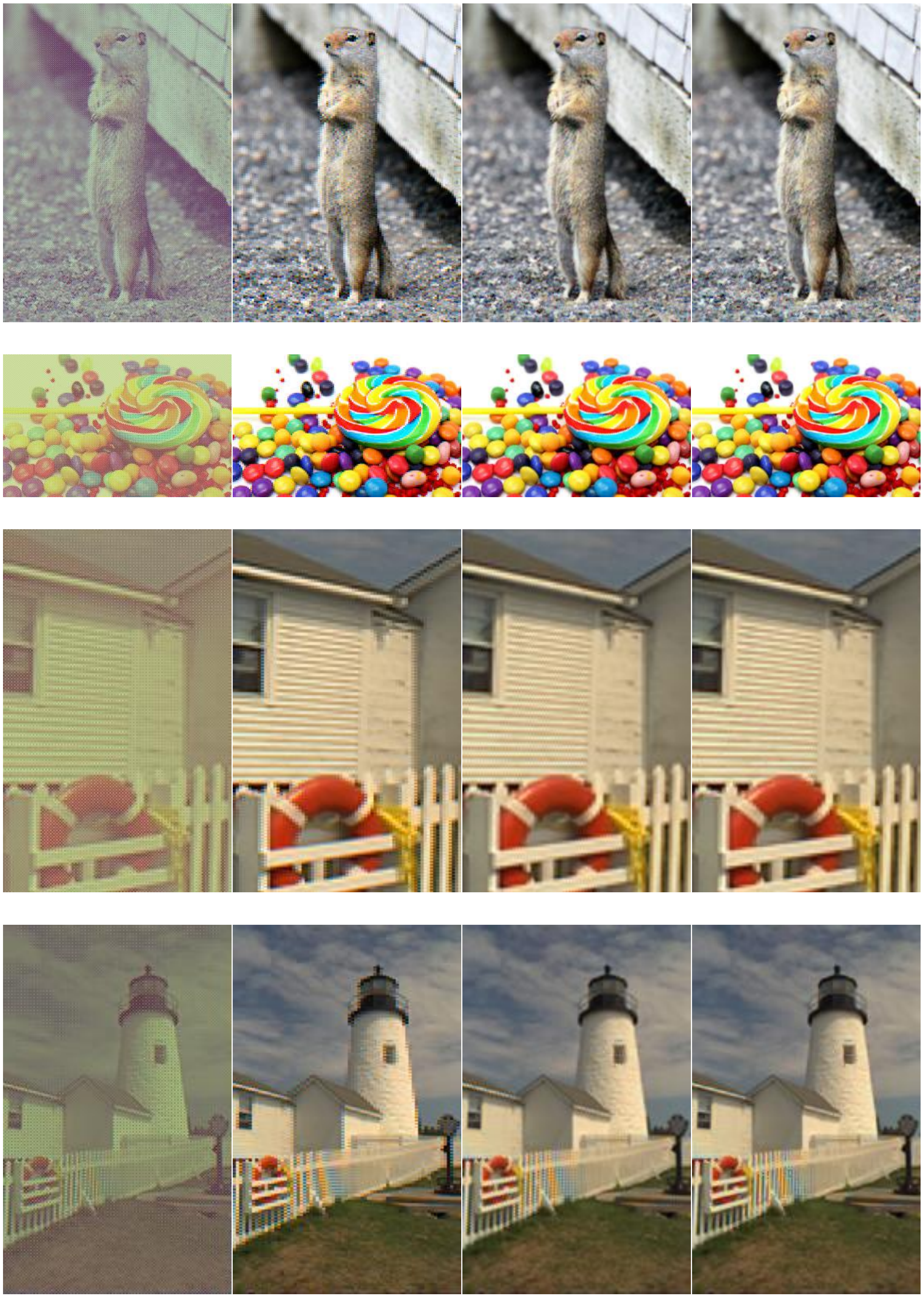
➢ Code-running Instruction:

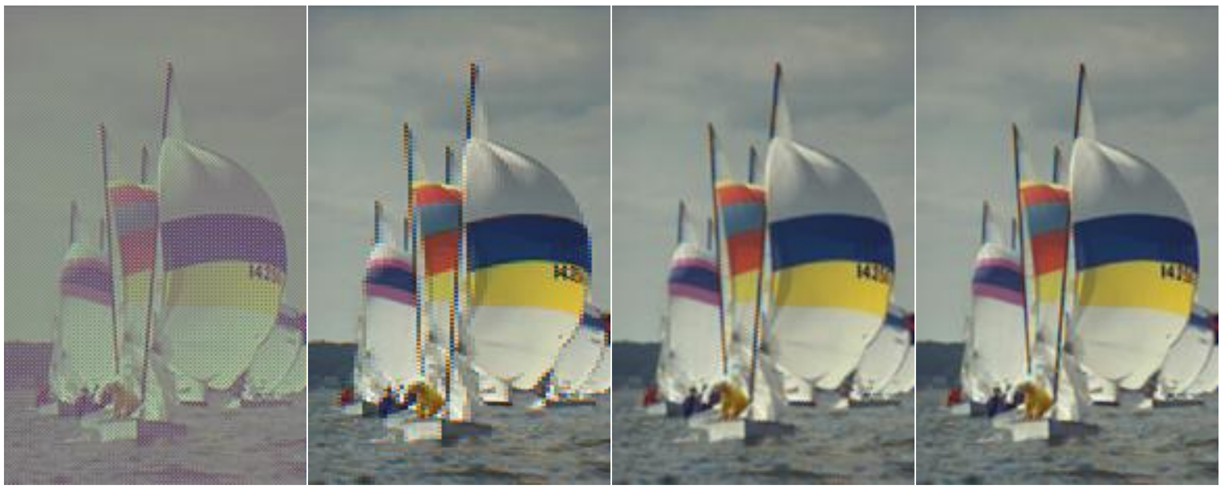
1. Run the script "evalDemosicing".

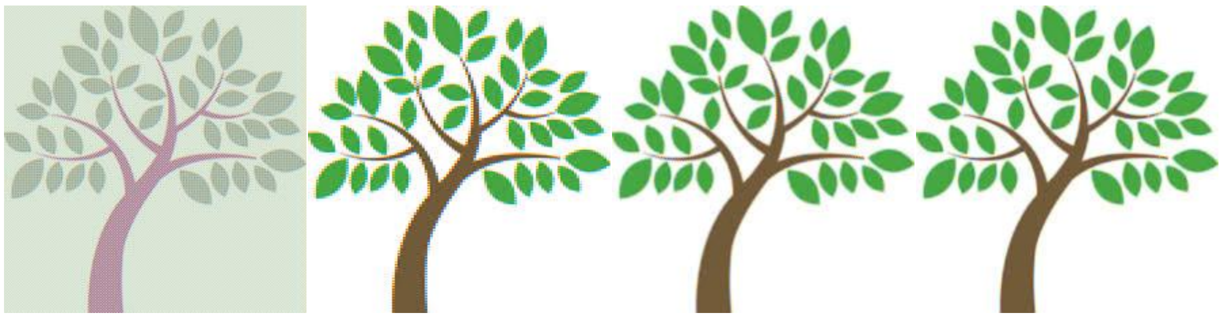
#	image	baseline	nn	linear	adagrad
1	balloon.jpeg	0.179239	0.018572	0.012879	0.011973
2	cat.jpg	0.099966	0.021616	0.013407	0.01342
3	ip.jpg	0.231587	0.023371	0.014693	0.012841
4	puppy.jpg	0.094093	0.01367	0.006126	0.005961
5	squirrel.jpg	0.121964	0.037839	0.023109	0.023542
6	candy.jpeg	0.206359	0.03697	0.022487	0.021636
7	house.png	0.117667	0.026267	0.01724	0.015653
8	light.png	0.097868	0.026159	0.017305	0.016561

9	sails.png	0.074946	0.020215	0.01364	0.012891
10	tree.jpeg	0.167812	0.024825	0.015211	0.014294
	Average	0.13915	0.02495	0.01561	0.014877









From the table above we can see that all of three methods implemented achieved significant improvement compared with the baseline. And the adaptive gradient method produced the best result with the average error being 0.014877.

The reason why adaptive performed well is that the adaptive interpolation always choose the mean of the two sides whose difference is smaller, making the image smoother. And smoothness is regarded as an intrinsic property of the nature. From another point of view, adaptive method is a way to detect and connect lines and edges in the image, and other methods such as NN and Linear may blur the edges and lines.

● **Extra Credit 1:** Transformed color spaces

■ **Methods and Results**

We experiment with 2 methods to transform the color spaces:

1. Color Ratio

Color ratio method is to first interpolate the green channel and then transform the red and blue channels to R/G and B/G, i.e., dividing by the green channel and then transforming them back after interpolation.

The main problem when we implement this was the 0s in the interpolated green channels, which cause errors since 0 can't be a divisor. We solve this problem by first traverse through every pixel in the green channel and every time we meet a 0 we fill it with the mean of its top, bottom, left and right pixels. And when we transform them back, what we did is to multiply the green channel with 0s (rather than the the one whose 0s has been filled).

2. Color Difference

The second method we tried is color difference, where we first interpolate the green channel, and then transform the red and blue channels to R-G and B-G. This method perform well in some specific circumstances, which will be analized in later section.

We didn't present "Logarithm of the ratio" method here because through experiments we found it performed very bad, or it might be my fault that I didn't found a better way to implement it.

➢ Code-running Instruction:

1. To use the "Color Ratio" transform method, change the function call "demosaicImage" to "demosaicImage_color_ratio" in the script "runDemosaicing.m". Then run "evalDemosaicing".
2. To use the "Color Difference" transform method, change the function call "demosaicImage" to "demosaicImage_color_diff" in the script "runDemosaicing.m". Then run "evalDemosaicing".

◆ Color Ratio

#	image	baseline	nn	linear	adagrad
1	balloon.jpeg	0.179239	0.045952	0.019543	0.029224
2	cat.jpg	0.099966	0.023136	0.010484	0.010617

3	ip.jpg	0.231587	0.043029	0.013933	0.010132
4	puppy.jpg	0.094093	0.021167	0.005028	0.004761
5	squirrel.jpg	0.121964	0.049096	0.01709	0.018914
6	candy.jpeg	0.206359	0.073765	0.032823	0.040146
7	house.png	0.117667	0.025594	0.013453	0.0088
8	light.png	0.097868	0.032428	0.013746	0.011352
9	sails.png	0.074946	0.024975	0.010956	0.008178
10	tree.jpeg	0.167812	0.027199	0.012194	0.009955
	Average	0.13915	0.036634	0.014925	0.015208

◆ Color Difference

#	image	baseline	nn	linear	adagrad
1	balloon.jpeg	0.179239	0.036255	0.026256	0.024253
2	cat.jpg	0.099966	0.022076	0.010229	0.010315
3	ip.jpg	0.231587	0.084335	0.067686	0.062201
4	puppy.jpg	0.094093	0.016349	0.004827	0.004376
5	squirrel.jpg	0.121964	0.042284	0.016864	0.018469
6	candy.jpeg	0.206359	0.05071	0.029685	0.027984
7	house.png	0.117667	0.02285	0.012847	0.007432
8	light.png	0.097868	0.029523	0.013288	0.010527
9	sails.png	0.074946	0.023241	0.01076	0.007932
10	tree.jpeg	0.167812	0.027186	0.011532	0.008992
	Average	0.13915	0.035481	0.020397	0.018248

■ Analysis

◆ Observations

1. In color ratio method experiment, we can see that it only got better result for linear interpolation method, which decreased the error value from 0.01561 to 0.014925. But in terms of nn and adgrad method the performance got worse than before.

2. In color difference method, all the experiments got a worse result both than the color ratio method and the original one.

◆ Why the color ratio might be a good idea in theory?

Color ratio method is based on the assumption that there is a strong dependency among the pixel values of different color planes, especially in areas wit high spatial frequencies. In other words, the differences or ratios between the pixel values in two color planes are likely to be constant within a specific image region. By doing ratio before interpolating the Red and Blue, it'll reduce the color artifacts and make the image more smoother.

◆ Why the color difference might be a good idea in theory?

The color difference method is derived from the same principle as the color ratio which is the pixel value dependency between color layers, but in a slightly different way. The color difference performs better than color ratio when the pixels in green layer have low values. The low values in the green layer makes the ratio R/G, B/G very sensitive to the small variations in the red/blue layer, thus make the interpolation result more artifact-prone.

◆ Why both of them don't work in our experiments?

We came up with the following possible reason to explain why the color ratio and color difference didn't work in our experiments:

1. The way we do green layer interpolation might be wrong or bad. How we interpolate the green layer determine how well the Red and Blue layer is reconstructed.

2. Test image set size is too small. As we can see later in the Extra Credit 2 where we do the demosaic for Prokudin-Gorskii image set, we can observe some improvement when using the color difference method.

3. Most of the images are low-frequency in most of area, which can't exploit the advantages of the color space transformation.

● **Extra Credit 2:** Digital Prokudin-Gorskii

■ **Results**

In this section we run demosaic script on Prokudin-Gorskii image sets. We run the basic demosaic algorithm without the color transformation and paste the picture outputs. Then we also test the two color space transformation methods to make a comparison.

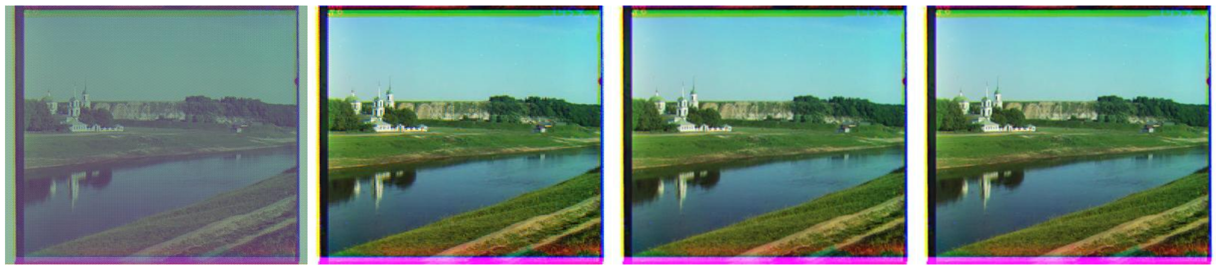
➢ Code-running Instruction:

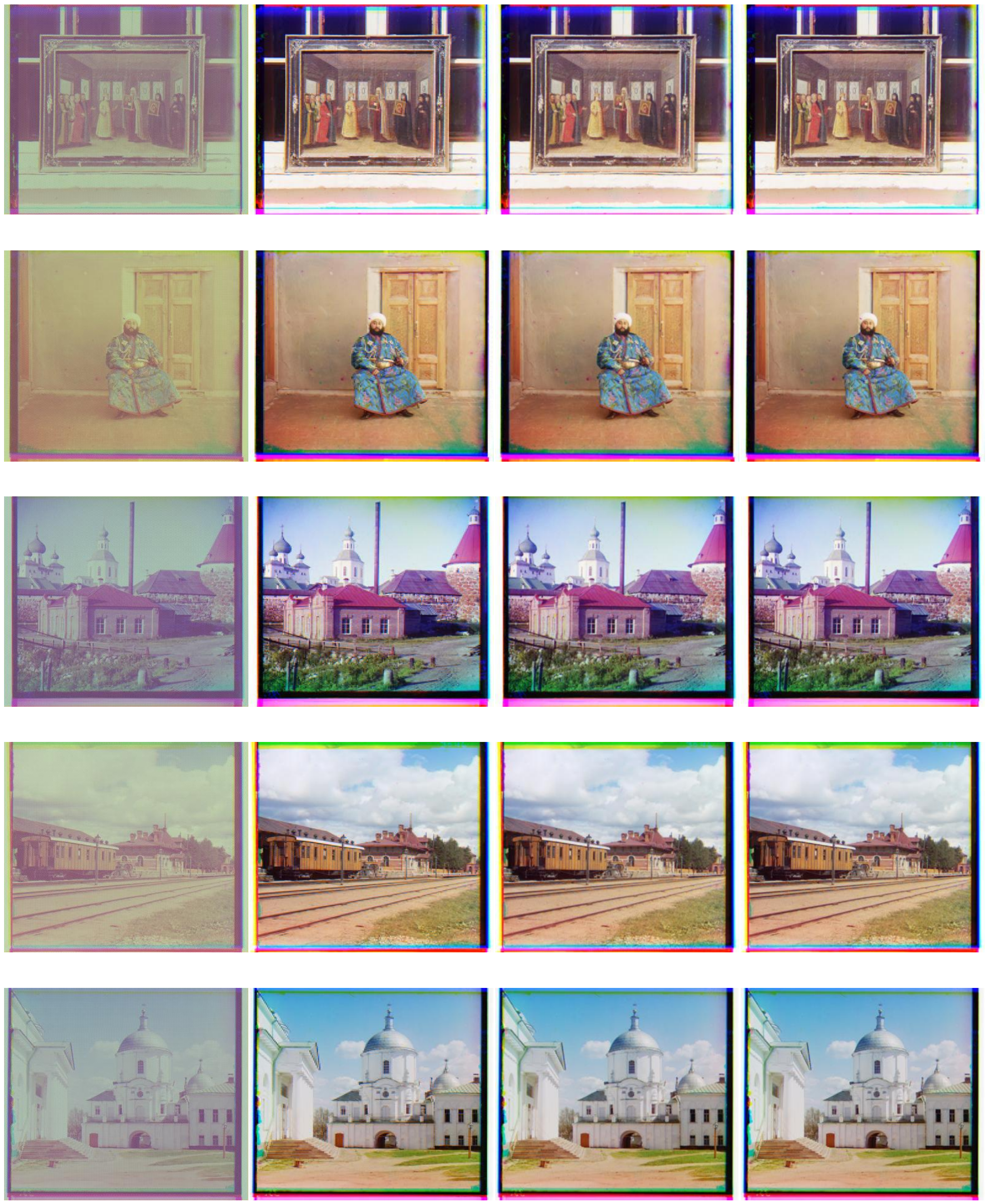
1. Run the script “evalDemosaicing_prokudin” to get the basic results (without color space transformation).
2. To use the “Color Ratio” transform method, change the function call “demosaicImage” to “demosaicImage_color_ratio” in the script “runDemosaicing.m”. Then run “evalDemosaicing_prokudin”.
3. To use the “Color Difference” transform method, change the function call “demosaicImage” to “demosaicImage_color_diff” in the script “runDemosaicing.m”. Then run “evalDemosaicing_prokudin”.

◆ Basic results (without color space transformation)

Below is the errors when running our demosaic algorithm on Prokudin-Gorskii image set. We also post out the output images.

#	image	baseline	nn	linear	adagrad
1	00125.jpg	0.155592	0.017759	0.010663	0.010224
2	00149.jpg	0.172048	0.027727	0.016648	0.015185
3	00153.jpg	0.11025	0.016386	0.009544	0.00898
4	00351.jpg	0.158936	0.027791	0.016589	0.016082
5	00398.jpg	0.143134	0.022644	0.014297	0.013382
6	01112.jpg	0.122246	0.019313	0.011607	0.010848
	Average	0.143701	0.021937	0.013225	0.01245





◆ Color Ratio

We also tried color ratio on this experiment. Unfortunately color ratio still make it worse.

#	image	baseline	nn	linear	adagrad
1	00125.jpg	0.155592	0.0257	0.01092	0.011303
2	00149.jpg	0.172048	0.039501	0.017801	0.017219
3	00153.jpg	0.11025	0.033294	0.012328	0.01618
4	00351.jpg	0.158936	0.045018	0.015881	0.017425
5	00398.jpg	0.143134	0.033034	0.015369	0.014912
6	01112.jpg	0.122246	0.025651	0.011235	0.010856
	Average	0.143701	0.0337	0.013922	0.014649

◆ Color Difference

We also tried color difference here and finally the color difference made a significant improvements when using linear and adaptive gradient interpolations. For linear interpolation, using color difference decreased the error by 10.13%. And for adaptive gradient this number is 21.6%.

#	image	baseline	nn	linear	adagrad
1	00125.jpg	0.155592	0.020134	0.00955	0.008618
2	00149.jpg	0.172048	0.032155	0.014866	0.010528
3	00153.jpg	0.11025	0.02119	0.00891	0.007458
4	00351.jpg	0.158936	0.032151	0.014047	0.012833
5	00398.jpg	0.143134	0.024103	0.013173	0.010574
6	01112.jpg	0.122246	0.022789	0.010763	0.008564
	Average	0.143701	0.025420	0.011885	0.009762