COMP9517 Assignment Report

Author : Yunqiu Xu (z5096489)

Environment

- Ubuntu 16.04 LTS, Python 2.7.13, IPython 5.1.0, numpy 1.12.1, opencv 2.4.11, Pillow(PIL) 3.1.2, matplotlib 2.0.2

Task 1

```
task1matrix.task1(img_name, h_factor = 0.5, w_factor = 0.5, window = 50, threshold = 15, step = 1)
```

- Step 1 : Divide the image as three parts with single channel (Blue, Green, Red)
- Step 2 : In order to deal with different sizes, besides of setting window size and step length manually, I use a size-based scale facter to adjust these parameters

```
scale_factor = max(int(min(height / 350, width / 400)), 1) # Treat (350, 400) as the base size of image
window *= scale_factor
step *= scale_factor
```

- Step 3 : We can just cut a part of image to match, which can be achieved by choosing suitable start position (h_factor, w_factor), window size (window), maximum steps (threshold) and step length (step)
- Step 4 : To make 3 parts more similar (some of them may be too bright or dark), we perform normalization for each of them.
- Step 5 : Find best match offset of Blue&Green, Blue&Red, this can be achieved via 2 methods:
  - Method 1: Moving the window by loop, this is easy to compute
  - Method 2: Matrix manipulation. First find the combination (Cartisian product) of all windows, then compute similarity and find minimum. By using numpy this can be very fast.
  - SSD is used as similarity measurement. Although it's slower than SAD, it's less sensitive to noise.
  - The best match is represented as the start position of two windows (e.g. [1,2,3,4] means img1[1:1+window, 2:2+window] matches img2[3:3+window, 4:4+window])
- Step 6 : Move G and R to match B, then merge them
- This method can deal with small (350, 400) and larger images (1600, 1800) accurately with rather fast speed. However if the image is extreme large (3200, 3742) , it's time consuming.

Task 2

`task2.task2(img_name, sharpen_factor, h_factor, w_factor, window, threshold, step)`

- To make image alignment faster, a 3-level Gaussian pyramid is built after splitting and normalizing the image.
- For the level with lowest resolution, images are sharpened (to make them clearer) and cut, then the method in task1 is reused to get best match.
- Suppose the best match we get from lowest-resolution level is (x_1, y_1, x_2, y_2), we then map it to higher-resolution level as $(2x_1, 2y_1, 2x_2, 2y_2)$. We treat this position as well as its 8 neighbours as candidates, e.g. $(2x_1 + step, 2y_1, 2x_2 + step, 2y_2)$, then find best match among them.
- Repeat above process to map best match to original images and find best match in this level. Then same with Task 1, we can move Green and Red to match Blue and merge them.
- Compared with Task 1, method in Task 2 is more suitable for larger images, even for very large images (3200, 3742) it can achieve decent performance in both accuracy and speed. However if the original image is too small that even the change of several pixels matters, the lowest-resolution level of image pyramid will loss too much information. Following are some examples treated by task1 or task2:

Task1 350 * 400

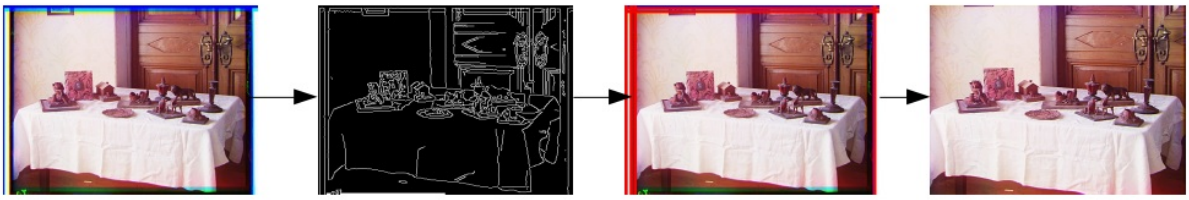
Task1 350 * 400

Task2 3200 * 3742

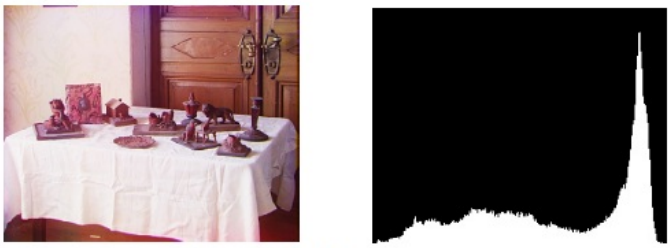
Task2 1600 * 1800

Task 3

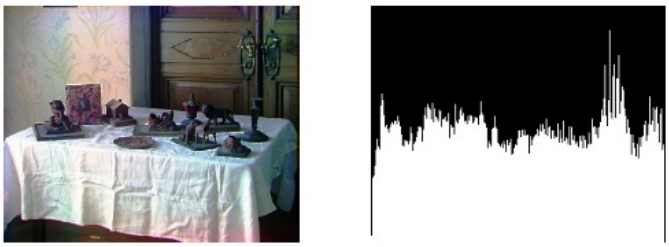
- Adjustment 1 : remove border
  - Perform Gaussian smoothing on image with gray scale
  - Find edge via Canny edge detection
  - Find straight line via Hough Transform
  - Cut the image, some intervals can be used to make sure all borders can be cut



- Adjustment 2 : histogram equalization
- Adjustment 3 : white balance using gray world algorithm



Before



After

- Adjustment 4 : brightness / contrast / saturation / sharpness adjustment
- Following is the image before and after all adjustments