DigiVFX project #2: Image Stitching

R08922079 洪浩翔

About this project

In this project, I implement panoramas creation, a famous image stitching application, with python and reference of Brown and Lowe's ICCV2003 paper called Recognising Panoramas. The panoramas creation process I implemented can be separated into four parts, including feature detection, feature matching, image matching and blending.

Pre-processing

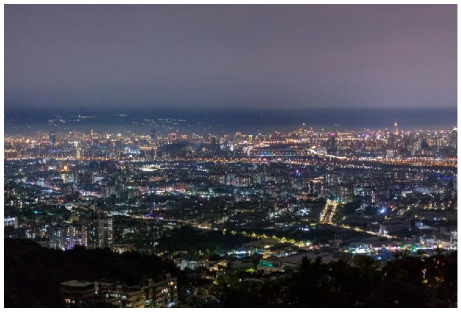
Before start to run through the whole process, pre-processing of images data should be done first, such as cylinder coordinate transformation. First, I collect the focal length in pixels of each input image. The formula we used is:

$$\mathrm{fp} = \mathrm{w} \cdot \frac{f}{Wccd}$$

where w is the width of image, f is the focal length in mm, and Wccd is the width of CCD size of camera. In my case, 50D's CCD size is 22.3 mm, so I set Wccd as 22.3 here. Then use the formula under to warp the input images to cylinder coordinate.

$$\mathrm{Newx} = \mathrm{fp} \cdot \frac{\mathrm{i}}{(\sqrt{j^2+fp^2})}; \quad \mathrm{Newy} = \mathrm{fp} \cdot \mathbf{Tan^{-1}} \frac{j}{fp}$$

In order to make the cropping process at the end of stitching easier, I also crop the resultant images to rectangle. In the end, I can get input images with warping to cylinder projection coordinate. Results are shown in Figure 1.



Original image

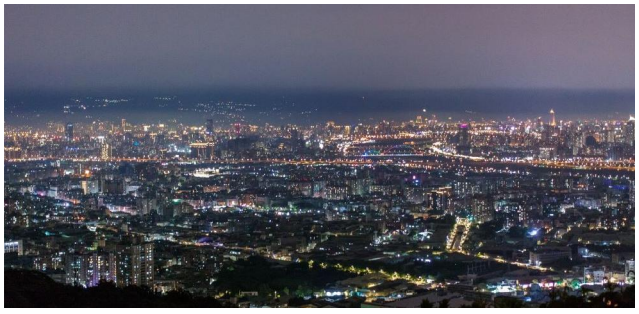
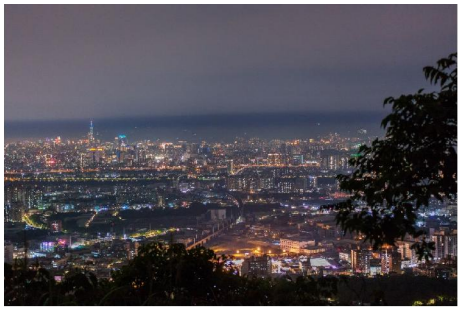
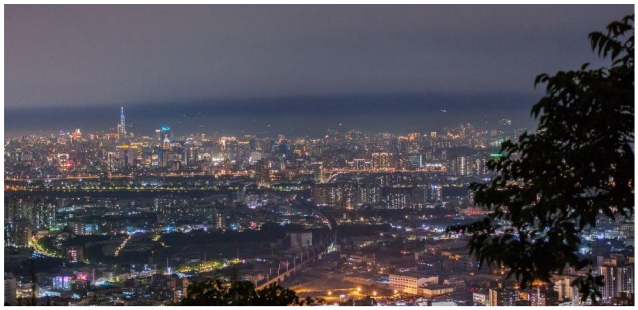


Image after warping

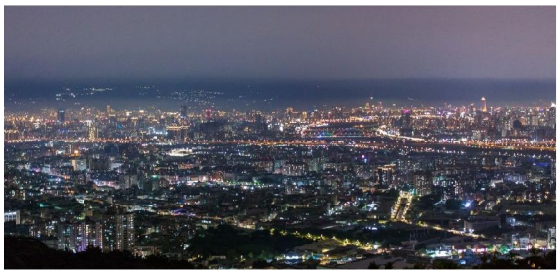



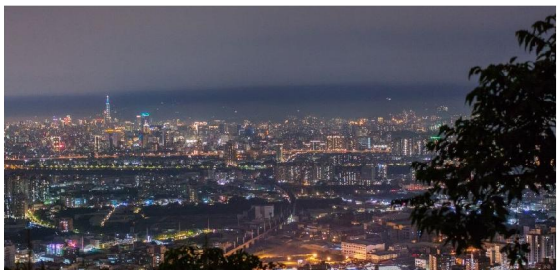
Original image Image after warping

Figure 1. Original input images and resultant images after warping to cylinder coordinate.

The next step is to transform the images into grayscale. Originally, I use OpenCV to transform the images. However, there are some unknown errors occurring when transforming. As a result, I do it by myself. I use formula below to transform RGB images to grayscale, and results are shown in Figure 2. This part can be done in 8.8 seconds with my testing images.

$$\text{Gray} = 0.2989 \cdot \text{R} + 0.587 \cdot \text{G} + 0.114 \cdot \text{B}$$






Warped images Grayscale images

Figure 2. Warped images and corresponding grayscale images

Feature detection

I first try to implement SIFT as my feature detection method. However, SIFT is too slow, and I also face some problems when implementing. Therefore, I discard

SIFT in the end and choose Harris corner detector as my feature detection method. After I compute the Harris algorithm and get corner points, I collect neighbored pixels in 30x30 regions of each corner points and flatten them as the feature vector. Figure 3 shows the feature points on the corresponding images.

In addition, to speed up the whole detection process, I parallelize the detection process to compute feature vector of every image at the same time. This is done by using multiprocessing package of python, and as a result this part can be done in 3.46s.

Figure 3. Feature points on images. Because I find corners as the feature points, smoothy region such as dark sky may not contain any feature point

Feature matching

I simply calculate the distance of two vectors to represent the similarity of two key points. That is:

$$\text{dist} = \sqrt{sum((v1 - v2)^2)}$$

The smaller the distance of two vectors is, the more likely that they are the matching points. In addition, I also add a constraint to pick out the best matching. I only collect key point whose minimum distance is smaller than the second minimum distance multiply a ratio. That is:

$$\text{MIN}_{\text{first}} < \text{MIN}_{second} \cdot ratio$$

I set ratio as 0.8 in my case. Figure 4 presents the matching key points of two images. I find that most of the resultant matchings are correct, while there are still some outliers. As a result, I handle such a condition carefully in next section, image matching.

Besides, I also use parallelization here to speed up the processing speed. There are totally n sub-processes to deal with n comparing pairs of images. Thus, this part can be done in 43s.

Figure 4. Matching keypoints on two neighbored image

Image matching

To deal with image matching, I first create a large black rectangle image as the template of my final output, and put the first image on it just like shown in figure 5.

Figure 5. Template of final output image

The second step is to find the affine transform to transfer the image to the right place on the template. I use RANSAC to estimate the affine transformation, and set outlier rate to 0.5 as the parameter. The result of this part is a 3x3 matrix, with the first 2x3 presenting the transforming details.

After finding the best affine transformation, the next step is image warping. New pixel places can be calculated by dotting original place matrix with the affine transformation. That is:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = M_{affine} \cdot \begin{bmatrix} x_{old} \\ y_{old} \\ \mathbf{1} \end{bmatrix}$$

With new pixel places, I can transform pixels on original images to the big template.

The final step of this section is putting images on the big template with new places. I record origin points of each image pairs after transformation, and paste them on the template one by one. This method is easy to implement, but when the number of images is large, error of transforming origin points may distort the last few images, making them weird.

Blending

Though I get the places to paste the images on the template, artifacts are very obvious if I paste the image directly, especially at the boundary of two images. As a result, blending is necessary. The blending method I use is constant blending. That is, only blend the constant size at the overlapping region. Figure 6 gives a better demonstration of constant blending, and figure 7 displays the process of image matching and blending. The blending region size I set is 30 pixels. Image matching and blending costs 71s, which is the bottleneck of this process.

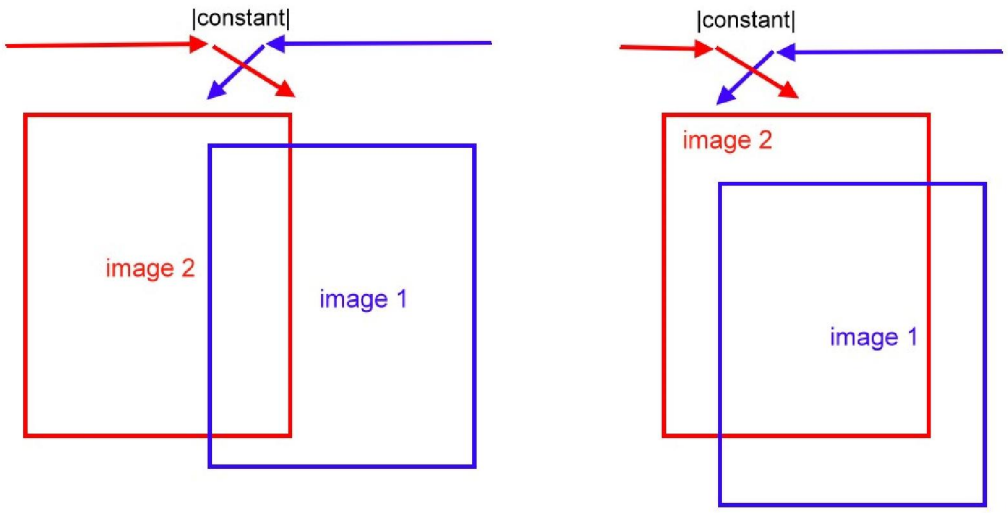


Figure 6. Example figure of constant blending. No matter how large the overlapping region is, I only blend contant size. Such a method can suppress or avoid ghost-like artifacts casued by moving objects.

Initial template

Stitching second image

Stitching third image

Stitching the last image

Figure 7. Process of image matching and blending

Post-process

In the last step, I remove redundant black places on the template. Instead of distributing error to images, I directly crop the template with recorded pixel places. Figure 8 displays the resultant images with different input. The time cost from starting to this section is about 139s.

Figure 8. Night view panorama created by my implemented process

What I have learned

First, I learn that SIFT is very time consuming that even with wrong output, it still costs 5 minutes for an image. On the other hand, Harris corner detector can be done in seconds with all images, which is very fast comparing to SIFT. However, I find that the key points detected by Harris are not very robust, or, can't detect key points

well in some extreme cases. For example, as shown in figure 9, night view with lack of obvious details can’t be stitched well.

Figure 9. Panorama with failure stitching

Files

main.py	Main program of panorama creation
util.py	Implemented function
Imgs4 & modified	Testing data set including images and text file recording focal length