Imaging Stitching

Digital Visual Effects(Spring 2023)

Group 5

B09201049 - Lu Guan Ting

B09705024 - Liu Chih Hsin

---

1. Image Collection

The original photos are below

2. Cylinder Projection

In **Cylinder Projection**, we adopt the following formula to derive the new coordinate for every pixel.

$$x' = f \arctan \frac{x}{f} \quad \text{and} \quad y' = f \frac{y}{\sqrt{x^2 + f^2}}$$

The f here is the **focal length**, we calculate it via the ration of image width and sensor width and get approximately 5146.67. Hence, the right image was yielded.

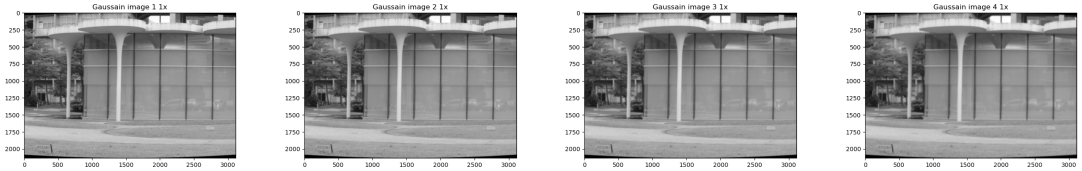
Note that relating codes are in **cylinder.py**.

3. Features(Scale-Invariant Feature Transform)

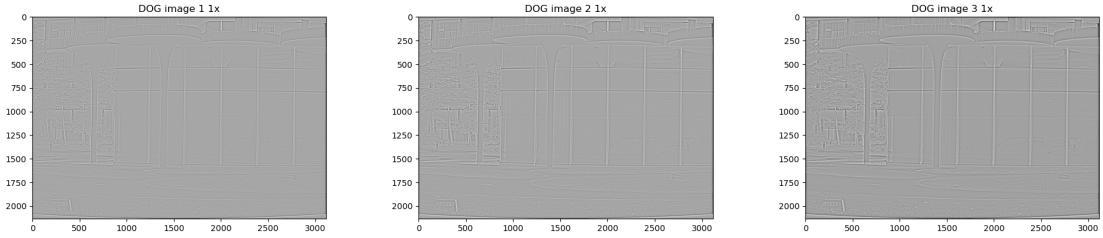
In **Features(Scale-Invariant Feature Transform)**, we refer to the matlab implementation to implemented our SIFT in python. The main steps are as below.

1. **Scale-space extrema detection**
   This involves identifying key points in the image that are invariant to scale and rotation.

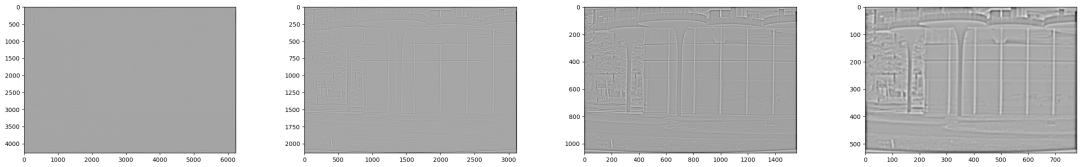
This is done by creating a Difference of Gaussian (DoG) scale space, and identifying local extrema in this space.
For parameter setting, we set 4 octaves in total, which are 2x, 1x, 0.5x, 0.25x scale of the original image respectively. Each scale makes 4 layers of gaussian, and hence obtains 3 layers of DoG.
In order to get the DoG, we first need to apply gaussian filter on the input images with different $\sigma = 2^{\frac{1}{s}i}a$ where s is 3 and $i \in [0, 3]$, and a is 1.6 respectively. After that, we can generate DoG by subtracting every neighbor. Take the first input image as example, the Gaussian images under different σ is generated as below.



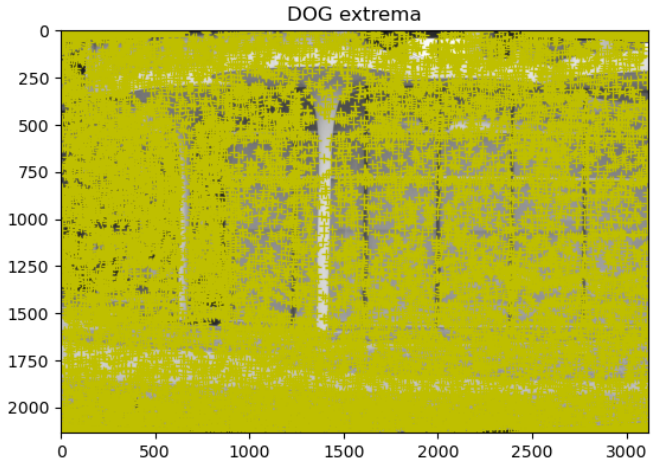
The Difference of Gaussian (DoG) images with different σ are as below.



We can also compare DoG on different sacle. We can see that as we zoom in the image, we can see more details. The scale from left to right are 2x, 1x, 0.5x, and 0.25x respectively.



Since we already got the DoG, each point on DoG is compared with the 26 adjacent points, the larger or smaller one is labeled as extrama. Take the first input image as example, all the extremas we found here are shown in below.



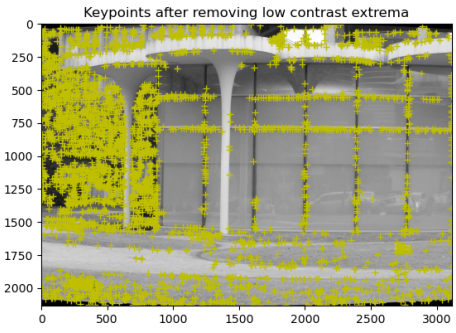
Note that relating codes are in **sift_detector.py** and in the functions **generate_pyramid** and **get_keypoints** in **utilis.py**.

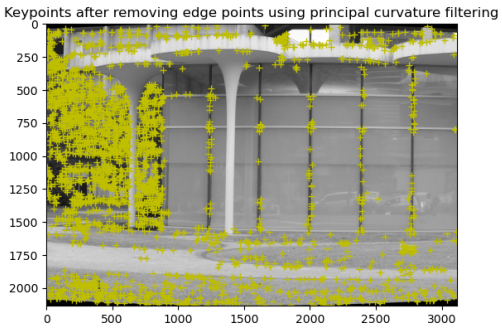
2. **Keypoint localization**
We can see that there are too many keypoints found in previous step. The key points identified in step 1 are not necessarily precise, so there are two tests these keypoints need to pass. First, we set the threshold to be 3.0. Only those keypoints with absolute value larger that 3.0 will remain. Second, we compute the ratio of the principal curvatures.sub-pixel accuracy. This is done by fitting a quadratic function to the DoG scale space in the vicinity of each keypoint. we need to compute the trace and the determinant of the Hessian to get the principal curvatures.

$$Trace\ of\ Hessian = D_{xx} + D_{yy}$$

$$Determinant\ of\ Hessian = D_{xx}D_{yy} - D_{xy}^2$$

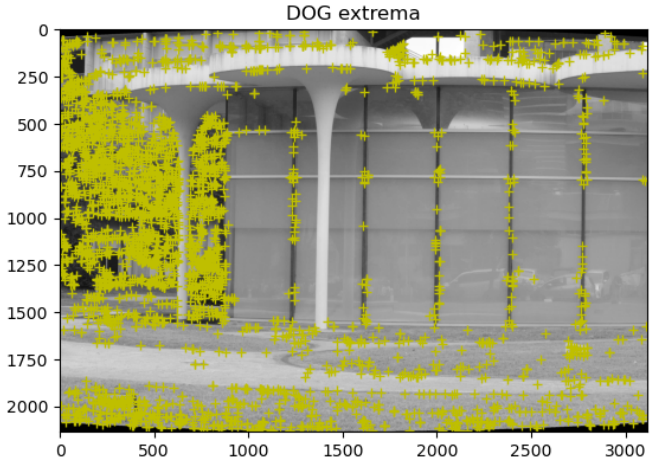
Take the first input image as example, all the key points we found after removing contrast extrema and using the principal curvature filtering are shown here.





Note that relating codes are in **sift_detector.py** and in the function **get_keypoints** in **utilis.py**.

3. **Orientation assignment and Keypoint descriptor**
   Here, we want to get keypoint descriptors for feature matching. By assigning a consistent orientation, the keypoints descriptor can be orientation invariant. Hence, A descriptor is computed for each keypoint based on the gradient magnitude and orientation in a local patch around the keypoint through taking 16×16 pixels for each keypoint, cutting them into 4×4 cells on average, and taking the gradient and angle value of each cell as a Histogram of 8 bins. These information is merged into a 128-dimensional data, and finally we also normalize the feature descriptor to a unit vector to make the descriptor invariant to affine changes in illumination.



   We can see that we really removed a lot of keypoints compared to initial one. Note that relating codes are in **sift_detector.py** and in the functions **assign_orientation** and **generate_descriptor** in **utilis.py**.

4. Feature matching

In **Feature matching**, features are matched between each pair of images by comparing their descriptors. Here, we use the KDTree and knn library from **sklearn.neighbors** to help us speed up matching. After that, we go through each pair of keypoints. And we set the distance threshold to be 0.25. Only the pair of keypoins with distance smaller than the threshold will remain.
Note that relating codes are in **feature_matching.py**.

Figure 1: The features matched between 1st image ans 2nd image

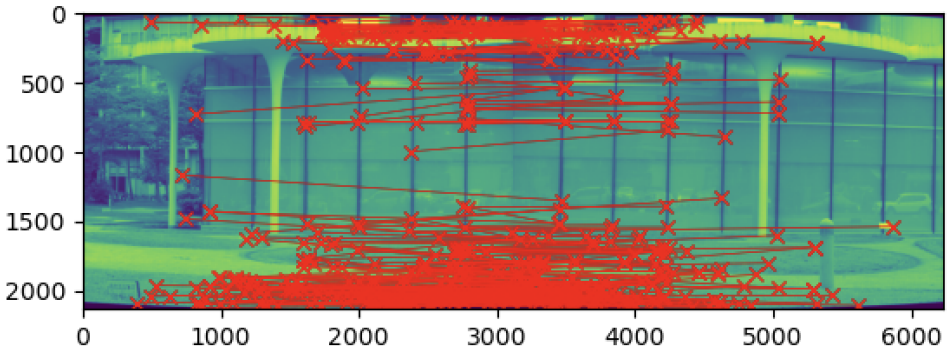


Figure 2: The features matched between 2nd image ans 3rd image

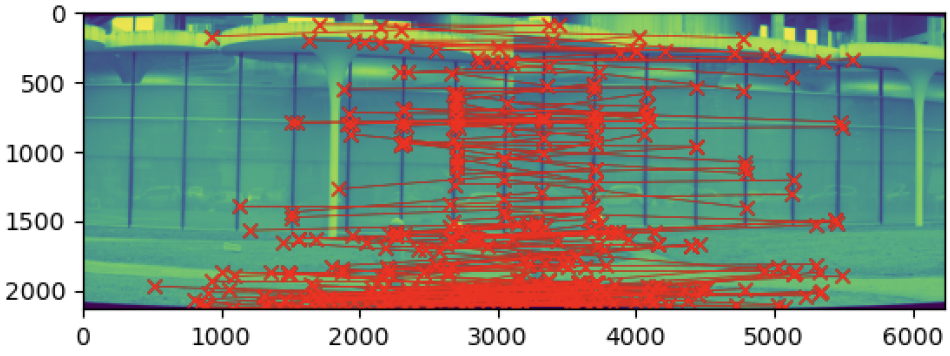


Figure 3: The features matched between 3rd image ans 4th image

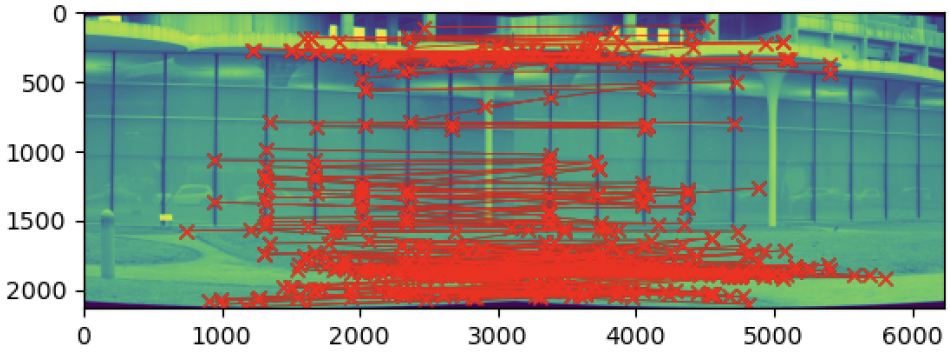
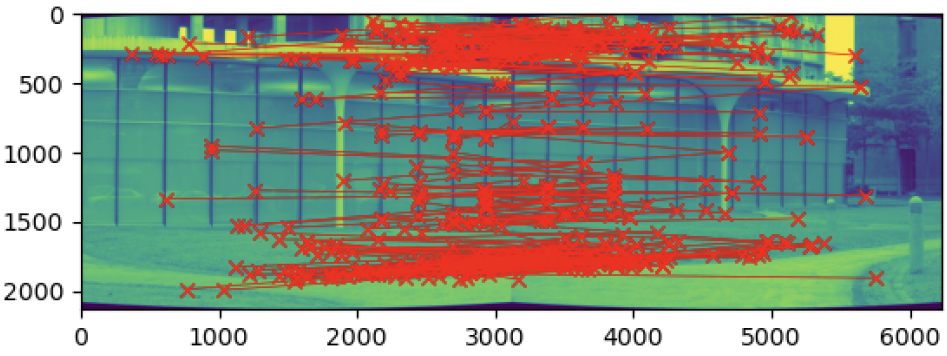


Figure 4: The features matched between 4th image ans 5th image



5. Image matching

Once feature matches have been identified, a robust estimation technique **RANSAC** is used to estimate the offsets between given pair of images. Practically, we set the threshold as 2.5 pixels, and since we don't really have a large number of feature pair to be matched, we just go through all the pairs. Then, we use the number of other offsets whose distance are lower than the threshold as a score, and the final offset will be determined via this score in the end.
In the end, we will get a list of offsets representing the offset for each image. The result in our image sets is [[0, 0], [-1639, -11], [-2104, -52], [-1076, -2], [-2179, -70]]
Note that relating codes are in **image_matching.py**.

6. Panorama

To form the ideal Panorama, we need to tackle with the rugged frame of the final image and the raw overlapping area. The following images are raw version of panoramas.

1. **Cropping**
   To get a complete rectangle as our final panorama, we picked

   $$\min\{\textit{ceiling of i-th image}\} \text{ and } \max\{\textit{floor of i-th image}\}$$

   to be the upper bound and lower bound to crop the image, so we can maximize the size of panorama. After cropping, we derived the image below.

2. **Linear Blending**
   We naively chose the distance from the frame of overlapping area as their weight to blend pixels. For example, consider pixels p_a and p_b in the exactly same location from image A and image B respectively, under the condition that the width of the overlapping is w-pixels and p_a, p_b are d-pixels from image A, the result pixel is going to be

   $$p_{result} = \frac{w-p}{w}p_a + \frac{p}{w}p_b$$

   In the end, we get the image in section **Result**.

3. **Example Images Demonstration**
   We also applied our algorithms to these two sample images. After blending, we have

   After cropping, we get

Note that relating codes about constructing panorama are in **stitch.py**.

7. Result

We have the result shown below.

8. Review

Lu Guan Ting
This project is absolutely much more trickier than the previous one, and I do learned a lot from it. For example, I applied the coordinate changing techniques to cylinder projection visually. However, several algorithms we adopted are difficult to be implemented, but my partner still conquer numerous issues stemming from the implementation. Hence, I realize that applying our mathematical and coding skill to a project is not easy at all.
Liu Chih Hsin
In this porject, the part of the implementation of **SIFT** was really troublesome, and I did not go further to improve it. Hence, feature points were chosen in efficiently. However, this project do bring me a lot of valuable experiences. For example, to implement high-quality stitching, steadies of photos and the parameter of focal length really matter. If qualities of photos vary a lot, the result will be terrible. Furthermore, if the overlapping area contains information from three or more image, there might be some errors during phases of stitching and blending.

References

[1] Cylinder projection
Digi VFX lecture06 https://www.csie.ntu.edu.tw/~cyy/courses/vfx/23spring/lectures/handouts/lec07_stitching.pdf

[2] SIFT
SIFT matlab implementation http://ftp.cs.toronto.edu/pub/jepson/teaching/vision/2503/SIFTtutorial.zip
pySIFT https://github.com/SamL98/PySIFT

[3] Kdtree matching
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html

[4] RANSAC
Digi VFX lecture06 https://www.csie.ntu.edu.tw/~cyy/courses/vfx/23spring/lectures/handouts/lec07_stitching.pdf

[5] Alignment(Crop)
Digi VFX lecture06 https://www.csie.ntu.edu.tw/~cyy/courses/vfx/23spring/lectures/handouts/lec07_stitching.pdf