Image Stitching Project Report

October 12, 2015

Contents

1 Introduction

1.1 Compilation

Dependencies:

1. gcc $\geq$ 4.7

2. GNU make
3. Eigen[1]
4. CImg[2] (already included in the repository)

CImg is only used to read and write images, and it optionally depends on libpng, libjpeg.

Compilation:

```
$ cd src && make
```

1.2 Run

Various parameters are saved in `src/config.cfg`. Here I'm introducing some of them.

Two modes of stitching are available (set/unset the option `CYLINDER`)

cylinder mode When the following conditions meet, this mode usually yields better results:

- Images are taken with almost-pure rotation. (as common panoramas)
- Images are given in the correct order. (I might fix this in the future)
- Images are taken with the same camera, and a good `FOCAL_LENGTH` is set.

general mode This mode has no assumptions on input images. So it'll be slower.

Some other options users may care:

- `FOCAL_LENGTH`: focal length of your camera in 35mm equivalent[3]. Only used in cylinder mode.
- `STRAIGHTEN`: Only used in general mode. Whether to try straighten the result. Good to set when dealing with rotational panorama.
- `CROP`: whether to crop the final image to avoid black border.

Other parameters are quality-related. The default values are generally good for images with more than 0.7 megapixels. If your images are too small and cannot give satisfactory results, it might be better to resize your images rather than tune the parameters.

To run the stitcher, use

```
./image-stitching <file1> <file2> <file3> ...
```

Output file is `out.png`.

In cylinder mode, the input file names need to have the correct order.

[1] http://eigen.tuxfamily.org
[2] http://cimg.eu
[3] https://en.wikipedia.org/wiki/35_mm_equivalent_focal_length

2 Algorithms

2.1 Feature Detection and Matching

2.1.1 SIFT

Lowe's SIFT algorithm[4] is implemented in `feature/*.cc`. The procedure of the algorithm and some results are briefly described as followed.

1. **Scale Space** (`feature/dog.cc`)

   A Scale Space consisting of $S \times O$ grey images is built. The original image is resized in O different sizes (AKA. octaves), and each is then Gaussian-Blured by S different σ.

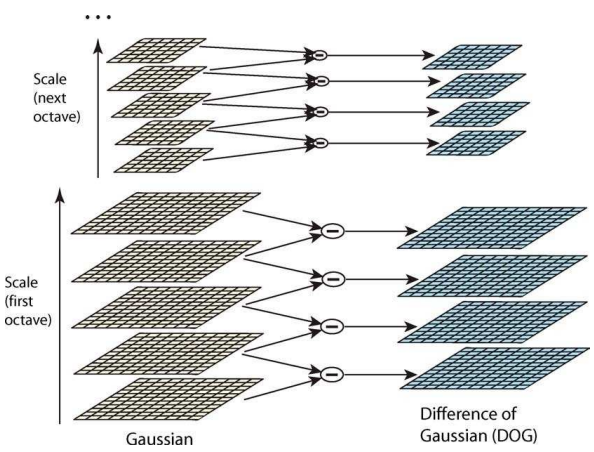

Figure 1: Scale Space and DOG Space

   Feature is detected on different resized version of the original image, to provide scale invariant feature.

   The gaussian blur here is implemented by applying two 1-D convolutions, rather than a 2-D convolution. This speeds up the computation significantly.

2. **DOG Space**(`feature/dog.cc`)

   In each octave, calculate the differences of every two adjacent blured images, to build a Difference-of-Gaussian Space. Therefore, DOG Space consists of $(S-1) \times O$ grey images. As shown in Fig.1.

3. **Extrema Detection**(`feature/extrema.cc`)

   In DOG Space, detect all the minimum and maximum by comparing a pixel with its 26 neighbors in **three directions**: x, y, σ. See Fig.2 and Fig.3

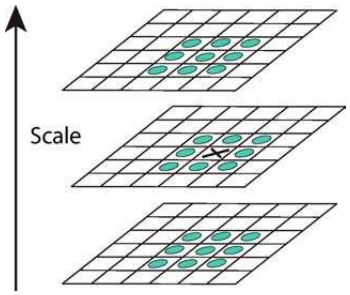

Figure 2: Extrema Detection

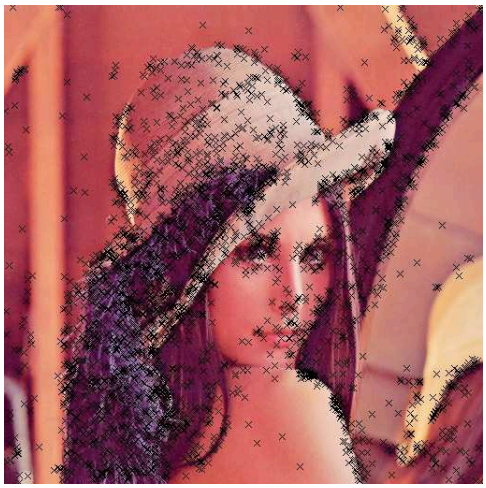

Figure 3: Extrema Example

4. **Keypoint Localization**(`feature/extrema.cc`)

   Use **parabolic interpolation** to look for the accurate location of the extrema. Then reject the points with **low contrast**(by thresholding pixel value) or **on the edge**(by thresholding principle curvature), to get more distinctive features. See Fig.4, Fig.5, Fig.6.

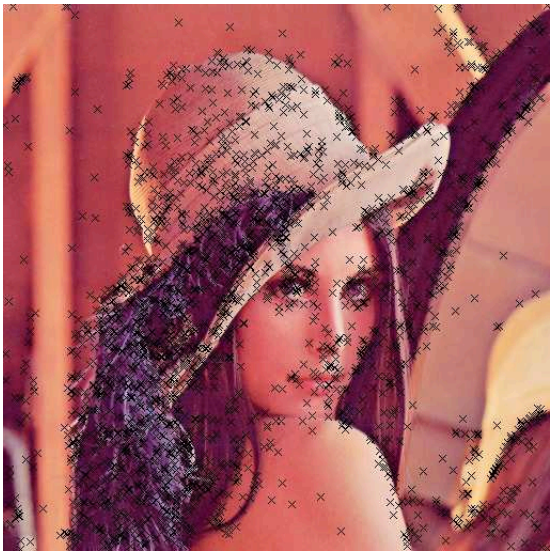

Figure 4: After Localization

Figure 5: After Rejecting Low Contrast

5. **Orientation Assignment**(`feature/orientation.cc`)

   First, we calculate **gradient and orientation** for every point in the Scale Space. For each keypoint detected by the previous procedure, the orientations of its nearby

Figure 6: After Eliminating Edge Point

Figure 7: After Assigning Orientation

points will be collected and used to build an **orientation histogram**, weighted by the magnitude of the gradient at each point, together with a gaussian kernel centered at the keypoint. The peak in the histogram is chosen to be the major orientation of the keypoint, as shown by the arrows in Fig.7.

6. **Descriptor Representation**(`feature/sift.cc`)

   D. Lowe suggested choosing 16 points around the keypoint, to build orientation histograms for each point, similar to the method in Sec.5. Each histogram uses 8 different possible values(bins). Therefore the final SIFT feature is a **128-dimensional** floating point vector. Since the major orientation of the keypoint is known, by using relative orientation to the keypoint, this feature is roatation-invariant.

7. **Feature Matching**(`feature/{feature,match}.cc`)

   Euclidean distance of the 128-dimensional descriptor is the criteria for feature matching between two images. A match is considered not convincing and therefore rejected, if the distances from it to its closest neighbor and second-closest neighbor are similar. A result is shown in Fig.8.

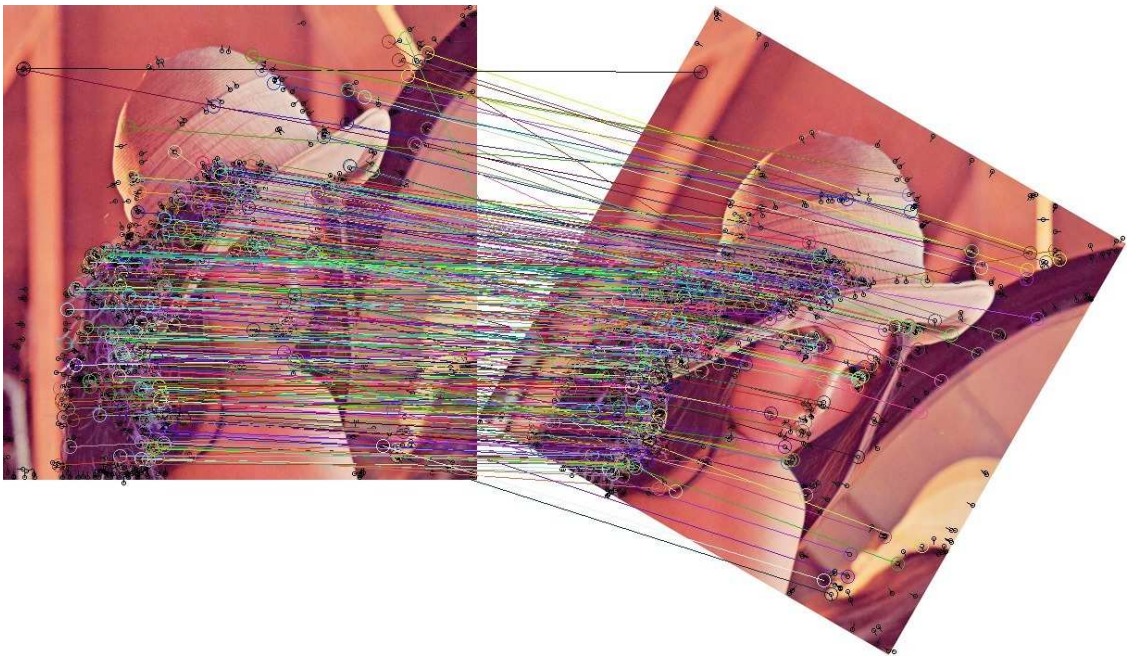

Figure 8: Matching Result

To calculate Euclidean distance of two arrays, I used Intel SSE3 intrinsics to speed up. KDTree could be used in the future to search for nearest neighbor.

2.1.2 BRIEF

BRIEF is a simple and robust descriptor, details can be seen in [2].

1. **Keypoint**

   Keypoint localization in BRIEF feature can be done in different ways. I choosed to use the same routine as in SIFT feature.

2. **Descriptor Representation**(`feature/brief.cc`)

   BRIEF is computed by comparing 256 pairs of pixels around the keypoint. These pairs are pre-computed and fixed. Therefore the result would be a bit vector of 256 length.

3. **Feature Matching**(`feature/{match,feature}.cc`)

   Hamming Distance is used to compare two bit vectors. Similarly, a match is rejected if the distance to its nearest neighbor is similar to the distance to its second-nearest neighbor. The hamming distance is implemented with gcc `__builtin_popcount()` function to speed up.

2.2 Transformation

For any two images taken from a camera at some fixed point, they can be related by a homography matrix H, such that for a pair of corresponding point $p = (x, y, 1), q = (u, v, 1)$, we have

$$p \sim Hq$$

The homogrpahy matrix has the following two fomulations:

1. Homography I: $H = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{bmatrix}$
2. Homography II: $H = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ and $\left\|H\right\| = 1$

When two images are taken with only camera translation and rotation aligned with the image plane (no skew), then they can be related by an affine matrix of the following form:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

RANSAC (Random Sample Consensus) algorithm[3] is used to calculate a transformation matrix. In every iteration, several matched pairs is randomly chosen to calculate a best-fit transformation matrix, and the pairs which agree with the matrix are taken as inliers. It's implemented in `stitch/transform_estimate.cc`.

Given a set of matches, affine and the first kind of homography transformation can be solved by least-squares fitting an over-determined linear system. The second kind of homography transformation can be solved by finding eigenvectors. These estimation are implemented in `lib/imgproc.cc`.

After each matrix estimation, I performed a "health check" to the matrix, to avoid mal-formed transformation estimated from false matches. This includes two checks: (see `stitch/transform.hh`)

1. The skew factor in homography $(H_{3,1}, H_{3,2})$ cannot be too large. Their absolute values are usually less than 0.002.
2. Flipping cannot happen in image stitching. A homography is rejected if it flips either x or y coordinate.

For a matched pair of images, a confidence for this match is assigned by taking the ratio of inliers among all the feature matches. This is suggested by [1], and is useful in the construction of global structure.

2.3 Cylinder Mode

2.3.1 Necessity of Projection

If a rotational input is given, as most panoramas are built, using planar homography leads to **vertical distortion**, such as Fig.9.

Figure 9: Vertical Distortion with Homography on a Planar

This is because a panorama is essentially an image taken with a cylindrical or spherical lens, not a planar lens any more. Under this circumstance, a circle around the camera (in Fig.9) should become a line.

There are two ways to handle with this problem: to warp before or after transform estimation. [4] These are the two modes used in this system. In this section we will introduce the algorithms used in cylinder mode.

2.3.2 Warp

One way of doing this is to project each image to a cylinder surface at the beginning, by the following formula:

$$\begin{cases} x' = \arctan \dfrac{x - x_c}{f} \\ y' = \dfrac{y - y_c}{\sqrt{(x - x_c)^2 + f^2}} \end{cases}$$

where f is the focal length of the camera, and x_c, y_c is the center of image. See `stitch/warp.cc`

After projecting all images to the cylinder surface, images can be simply stitched together by an affine transformation. The result is in shown in Fig.10. Note that the white line on the ground is straightened.

[4] A comprehensive app revealing the reason of this projection can be seen at http://graphics.stanford.edu/courses/cs178-12/applets/projection.html

Figure 10: Stitching Result After Projection

2.3.3 Straightening

Figure 11: Bended Panorama

Since the tilt angle of camera is unknown, the projection to the cylinder could still lead to distortion. Then the output panorama would be bended as shown in Fig.11. Instead of using the first image as the pivot, and calculating all the other transformation relative to it, using the image in the middle as the pivot can help reduce the tilt effect. Apart from that, I found a method to reduce the effect, by searching for y_c in the above warping formula.

Since the effect is caused by camera tilt, y_c could differ from $\frac{height}{2}$. The algorithm works by changing y_c and find a value which leads to the most straight result. See `Stitcher::update_h_factor()`. After this processes, the result is like Fig.12.

Figure 12: After bend correction

The change of y_c is not enough, since we are still warping images to a vertical cylinder. The error can be seen at the left and right edges in Fig.12. To account for this error, I used a perspective transform to align the left and right edges (`Stitcher::perspective_correction`). The result is in Fig.13

Figure 13: After perspective correction

2.4 General Mode

2.4.1 Intuition

The cylinder mode assume a known focal length and pure rotational panorama, to warp them on cylinder surface at the beginning. We want to get rid of these assumptions.

One method I tried is to estimate pairwise homographies, and directly stitch them together. To avoid distortion like Fig.9, I performed a cylindrical warp **after** the transformation. Then I get something like this:

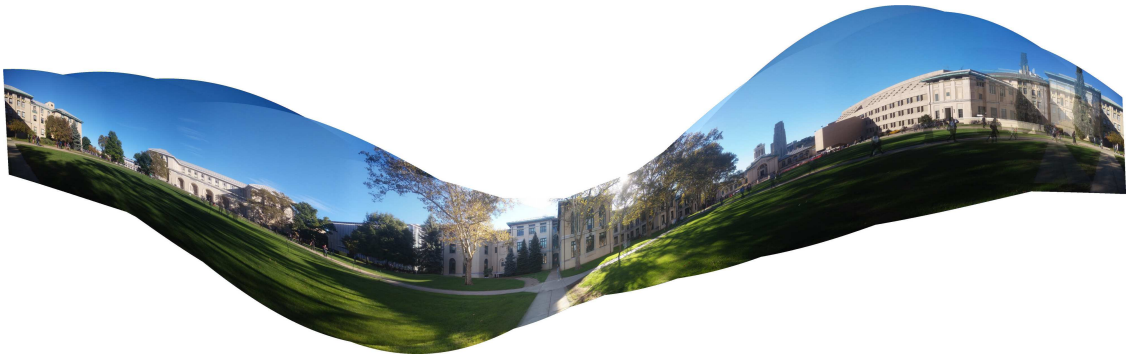

Figure 14: CMU campus, directly stitched and then warped to cylinder

This is likely caused by our weak assumption on homography matrix. A homography is estimated as a 3 by 3 matrix with the only constraint being the scaling factor. However, a homography is actually formed as:

$$H_{1,2} = K_1 R_1 R_2^{-1} K_2^{-1}$$

where K, R are intrinsic and rotation matrix of each camera. From this formulation there should be a geometric constrain on each H, which is not built into our initial estimation. That's why the overall geometric structural is broken, although the matching pairs are still well overlapped.

To overcome this problem, I estimated initial K, R for each camera, from pairwise homographies. Then, a global optimization is used to refine the parameters in K, R. Finally, pairwise homographies are rebuilt from our refined K, R.

2.4.2 Initial Estimation

As suggested by [5], focal length can be roughly estimated from pairwise homographies. This is implemented in `stitch/camera.cc`. I used it to produce initial estimation of focal length and assign it to each camera.

By assuming a rotation matrix for certain pivot image being identical, other rotation matrices can be estimated one by one using pairwise homography:

$$R_1 = K_1^{-1} H_{1,2} K_2 R_2$$

To reduce accumulated error, I first construct a max-spanning tree by using confidence of pairwise matches calculated in Sec.2.2 (`Stitcher::max_spanning_tree()`). Then each time the most confidence pair is used to estimate an unknown rotation matrix from a known one.

After obtaining a guess of K and R for each image, pairwise homographies can be reconstructed and used to stitch image together. A result is given below. Note that the distortion is gone due to our constraints on the formulation of pairwise homographies, but the quality is not satisfactory.

Figure 15: Initial guess of camera parameters

2.4.3 Bundle Adjustment

I implemented bundle adjustment in `stitch/bundle_adjuster.cc`, to globally optimize all the camera parameters. The object is to minimize the sum of error of every matching pair of pixels. The object function can be optimized in Levenberg-Marquardt algorithm [5]. For simplicity, I used numerical differentiation to calculate the Jacobian, although a closed-form formula is available, as shown in [1].

During the optimization, the best params so far is kept. The optimization ended when the error doesn't decrease in the last 5 iterations.

After the optimization, the above panorama looks much better:

[5] https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm

Figure 16: After Bundle Adjustment

I noticed that in some hard cases, bundle adjustment cannot converge to a good result. This is usually due to false matches or low-quality matches between images. [1] suggests incrementally adding images to the bundle adjuster, which sounds reasonable but I don't have time to implement that for now. Also, I think it'll also help to reject some matches if it is inconsistent with the current bundle.

2.4.4 Straightening

As suggested by [1], the result of bundle adjustment can have wavy effect, due to the unknown tilt angle. By assuming all cameras have their X vectors lying on the same plane, we can estimate a Y vector perpendicular to that plane to account for the tilt and fix the wavy effect. This part is implemented in `stitch/camera.cc`.

See the following two images and notice the straight line on the grass is corrected (it is actually a circle in the center of a soccer field).

Figure 17: Unstraightened and straightened result

2.5 Blending

The size of the final result is determined after having all the transformations. And the pixel value in the result image is calculated by an inverse transformation and bilinear interpolation with nearby pixels, in order to reduce alias effect.

For overlapped regions, the distance from the overlapped pixel to each image center is used to calculate a weighted sum of the pixel value. I only used the distance along the x axis to calculate the weight, to get better result in panoramic image. The result is almost seamless. Compare Fig.13 and Fig.17 to see the effect of blending.

2.6 Cropping

When the option `CROP = 1` is set, the program simply manage to find a rectangular with largest area within the original result.

A $O(n \times m)$ algorithm is used, where n, m is the height and width of the original result. The algorithm works like this:

For each row i, the update

$$h[j] = \begin{cases} 0, & if\ i = 0\ or\ A[i][j]\ is\ outside\ the\ area \\ h[j] + 1, & otherwise \end{cases}$$
$$r[j] = \max\{k \in [0, m) \cap \mathbf{N} : h[t] \geq h[j], \forall j \leq t \leq k\}$$
$$l[j] = \min\{k \in [0, m) \cap \mathbf{N} : h[t] \geq h[j], \forall k \leq t \leq j\}$$

for every $j \in [0, n)$ can be done in amortized $O(1)$ time, and the corresponding area is $(r[j] - l[j] + 1) \times h[j]$. After the iteration, the rectangular $[i - h[j] + 1, i] \times [l[j], r[j]]$ with maximum area is our result.

The cropped final result is shown in Fig.18

Figure 18: Cropped Result

3 More Results

Figure 19: Zijing Apartment in Tsinghua University

Figure 20: Carnegie Mellon University

Figure 21: Newell Simon Hall at CMU (this is hard since objects are close to camera)

Figure 22: me being silly

In general mode, no translation is the only requirement on camera. So we can even build wider-angle panorama:

Figure 23: CMU stitched with 38 images.

By adding a polar transformation on the above result, we can simply turn it into this, apple-like stuff:

4 References

[1] Matthew Brown and David G Lowe. "Automatic panoramic image stitching using invariant features". In: *International journal of computer vision* 74.1 (2007), pp. 59–73.

[2] Michael Calonder et al. "Brief: Binary robust independent elementary features". In: *Computer Vision–ECCV 2010* (2010), pp. 778–792.

[3] Martin A Fischler and Robert C Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (1981), pp. 381–395.

[4] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[5] H-Y Shum and Richard Szeliski. "Construction of panoramic image mosaics with global and local alignment". In: *Panoramic vision*. Springer, 2001, pp. 227–268.