Image Panorama Stitching

Thalles Santos Silva*
Hélio Pedrini†

1. Introduction

This piece presents a method to perform image stitching using Python and OpenCV. Given a pair of images that share some common region, our goal is to "stitch" them to create panoramic images.

Our method uses compute vision techniques such as keypoint detection, local invariant descriptors, keypoint matching, hoography estimation using RANSAC and perspective warping. Moreover, we explore many feature extractor algorithms like SIFT, SURF and ORB.

2. Setup

This project was developed using Python version 3 and Jupyter Notebooks. The libraries used across the project are NumPy [2] and OpenCV [3] to do image-based computation and matplotlib [4] for visualization. The Notebook is divided into subsections analogous to the sessions contained in this document.

3. Feature Detection and Extraction

Given a pair of images like in Figure 1, the first step to stitch them in a panoramic image is to extract some key points and features of interest.

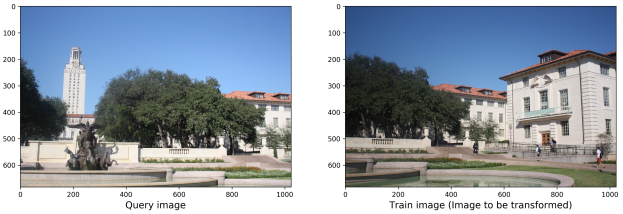

Figure 1. Input images pair.

These features, however, need to have some special properties. An initial and naive approach would be to extract key points using an algorithm such as Harris Corners. Then, we could try to match the corresponding key points from both images. As we know, corners have one nice property: they are invariant to rotation. In other words, once we detect a corner, if we rotate the image, the corner will still be there.

However, what if we rotate then scale the image? In this situation, we would have a hard time because corners are not invariant to scale. That is to say, if we zoom-in to an image, the previously detected corner might become a line!

In summary, we need features that are invariant to rotation and scaling. That is where more robust methods such as SIFT, SURF, and ORB come in.

In summary, methods like SIFT and SURF try to address the limitations of corner detection algorithms. As we know, corners are not invariant to scale. That is because corner detector algorithms use a fixed size kernel to spot corners on images. It is easy to see that when we scale an image, this kernel might become too small or too big. To address this limitation, methods like SIFT uses Difference of Gaussians (DoD). The idea is to apply DoD on different scaled versions of the same image. It also uses the neighboring pixel information to find and refine key points and corresponding descriptors.

To start, we need to load 2 images, a query image, and a training image. Initially, we begin by extracting key points and descriptors from both. We can do it in one step by using the OpenCV detectAndCompute() function. Note that in order to use detectAndCompute() we need an instance of a keypoint detector and descriptor object. It can be ORB, SIFT or SURF, etc. Also, before feeding the images to detectAndCompute() we convert it to grayscale. We run detectAndCompute() on both, the query and the train image. At this point, we have a set of key points and descriptors for both images. If we use SIFT as the feature extractor, it returns a 128 dimensional feature vector for each key point. If SURF is chosen, we get a 64-dimensional feature vector. Figures 2, 3, 4, and 5 show features extracted using SIFT, SURF, BRISK, and ORB.

3.1. Feature Matching

As we can see, we have a large number of features in both images. The next step is to compare the 2 sets of features and stick with the pairs that show more similarity. This

*Is with the Institute of Computing, University of Campinas (Unicamp). **Contact**: thalles753@gmail.com
†Is with the Institute of Computing, University of Campinas (Unicamp). **Contact**: helio@ic.unicamp.br

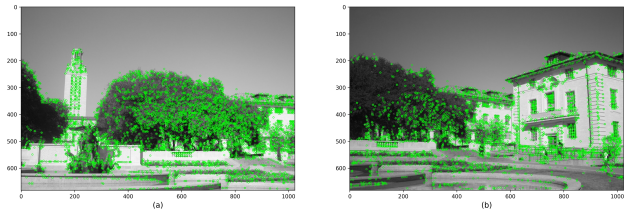

Figure 2. Detection of key points and descriptors using SIFT.

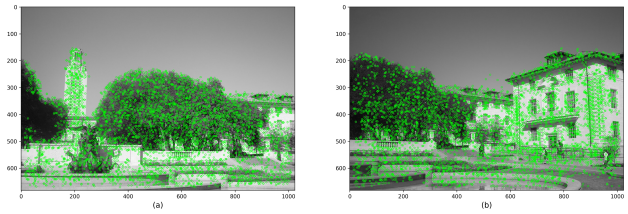

Figure 3. Detection of key points and descriptors using SURF.

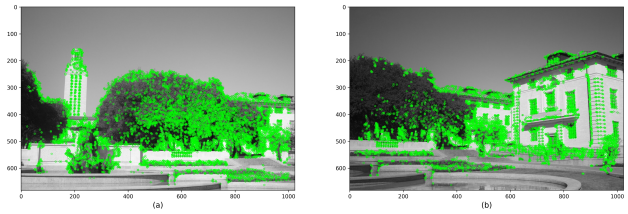

Figure 4. Detection of key points and descriptors using BRISK and Hamming distances.

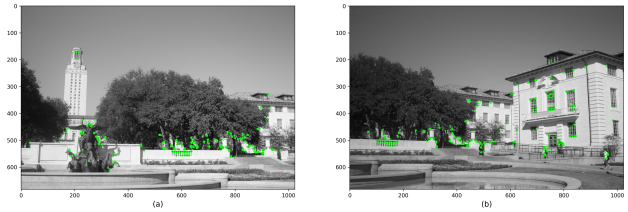

Figure 5. Detection of key points and descriptors using ORB and Hamming distances.

process is called feature matching. With OpenCV, feature matching requires a Matcher object. Here, we explore two flavors: KNN (k-nearest neighbors) and Brute Force.

The BruteForce (BF) Matcher does exactly what its name suggests. Given 2 sets of features (from image A and image B), each feature from image A is compared against all features from image B. By default, BF Matcher computes the Euclidean distance from two points. Thus, for every feature in set A, it returns the closest feature from the set B. For SIFT and SURF OpenCV recommends using Euclidean distance. For other feature extractors like ORB and BRISK, Hamming distance is suggested.

To create a BruteForce Matcher using OpenCV we only

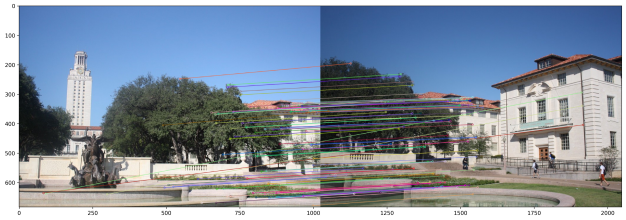

Figure 6. Feature matching using Brute Force Matcher on SIFT features.

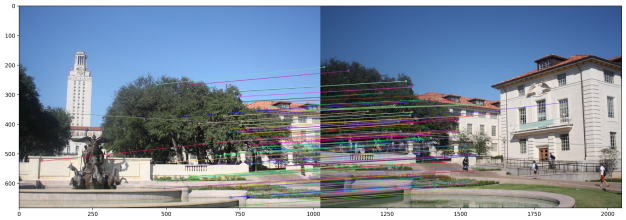

Figure 7. Feature matching using KNN and Ration Testing on SIFT features.

need to specify 2 parameters. The first, is the distance measurement. The second, is the crossCheck boolean parameter. It indicates whether the two features have to match each other to be considered valid matches. In other words, for a pair of features (f1, f2) to considered valid, f1 needs to match f2 and f2 has to match f1 as the closest match as well. This procedure ensures a more robust set of matching features and is described in the original SIFT paper.

However, for the case where we want to consider more than one good match, we can use the KNN Matcher. Instead of returning the only best match for a given feature, KNN returns the k best matches. Note that the value of k has to be pre-defined by the user.

As we expect, KNN provides a larger set of candidate features. However, we need to ensure that all these matching pairs are robust before going further.

3.2. Ratio Testing

D. Lowe, the author of the SIFT paper, suggests a technique called ratio test. Basically, we iterate over each of the pairs returned by KNN and perform a distance test. For each pair of features (f1, f2), if the distance between f1 and f2 is within a certain ratio, we keep it, otherwise, we through it away. Also, the ratio value must be chosen manually.

In essence, the ratio test does the same job as the cross-checking procedure from the BruteForce Matcher. Both, ensure a pair of detected features are indeed close enough to be considered similar. Figures 6 and 7 show the results of BF and KNN Matcher on the SIFT features. We chose to displays only 100 matching points to clear understanding.

Note that even after cross-checking for Brute force and

ratio testing in KNN, some of the features do not match properly.

The Matcher algorithm will give us the best (more similar) set of features from both images. Now, we need to take these points and find the transformation matrix that will stitch the 2 images together based on their matching points.

Such a transformation is called the Homography matrix. In short, it is a 3x3 matrix that can be used in many applications such as camera pose estimation, perspective correction, and image stitching. The Homography is a 2D transformation. It maps points from one plane (image) to another. Let's see how we get it.

3.3. Estimating the Homography

RANdom SAmple Consensus or RANSAC is an iterative algorithm to fit linear models. Different from other linear regressors, RANSAC is designed to be robust to outliers. Basically, models like Linear Regression uses least squares estimation to fit the best model to the data. However, ordinary least squares is very sensitive to outliers. As a result, it might fail if the number of outliers is significant.

As an answer, RANSAC solves this problem by estimating parameters only using a subset of inliers in the data. Figure 8 shows a comparison between Linear Regression and RANSAC. First, note that the dataset in Figure 8 contains a fairly high number of outliers.

We can see that the Linear Regression model gets easily influenced by the outliers. That is because it is trying to reduce the average error. Thus, it tends to favor models that minimize the overall distance from all data points to the model itself. That includes the outliers.

On the contrary, RANSAC only fits the model on the subset of points identified as the inliers. This characteristic is very important to our use case. For panorama stitching, we are going to use RANSAC to estimate the Homography matrix. It turns out that the Homography is very sensitive to the quality of data we pass to it. Hence, it is important to have an algorithm (RANSAC) that can separate inliers from outliers.

To estimate the homography using RANSAC we can use OpenCV findHomography() function. It takes in, the keypoints from the query and train images, and the a reprojection threshold. This threshold defines the maximum allowed reprojection error to treat a point pair as an inlier. For our experiments, the default value of 3.0 worked fine. Figure 9 shows an example of an estimated homography matrix using RANSAC.

Once we have the Homography between the 2 images estimated (using RANSAC) we need to warp one of the images to a common plane.

Here, we are going to apply a perspective transformation to one of the images. Basically, a perspective transform may combine one or more operations like rotation, scale, translation, or shear. The idea is to transform one of the images so that both images merge as one. To do this, we can use the OpenCV warpPerspective() function. It takes an image and the homography as input. Then, it warps the source image to the destination based on the homography.

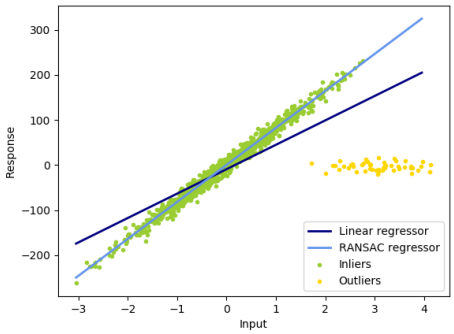

Figure 8. RANSAC model fitting.

```
[[ 7.66380921e-01  3.65186796e-02  4.46627996e+02]
 [-1.35000633e-01  9.11358967e-01  7.61410429e+01]
 [-2.10164883e-04 -3.32000479e-05  1.00000000e+00]]
```

Figure 9. Homography matrix estimation using RANSAC.

Some resulting panoramas are shown in Figures 10, 11, 12 and 13. As we see, there are a couple of artifacts in the result. More specifically, we can see some problems related to lighting conditions and edge effects at the image boundaries. Ideally, we can perform post-processing techniques to normalize the intensities like histogram matching. These would likely make the result look more realistic.

4. Conclusion

...

References

[1] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016. 1

[2] Oliphant Travis E. A guide to numpy, 2006. [Online; accessed 15/04/2018]. 1

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 1

[4] John D. Hunter. Matplotlib: A 2d graphics environment, 2007. [Online; accessed 15/04/2018]. 1

Figure 10. .

Figure 11. .

Figure 12. .

Figure 13. .