Image Panorama Stitching with OpenCV

Thalles Silva Follow
Jul 26 · 8 min read ★

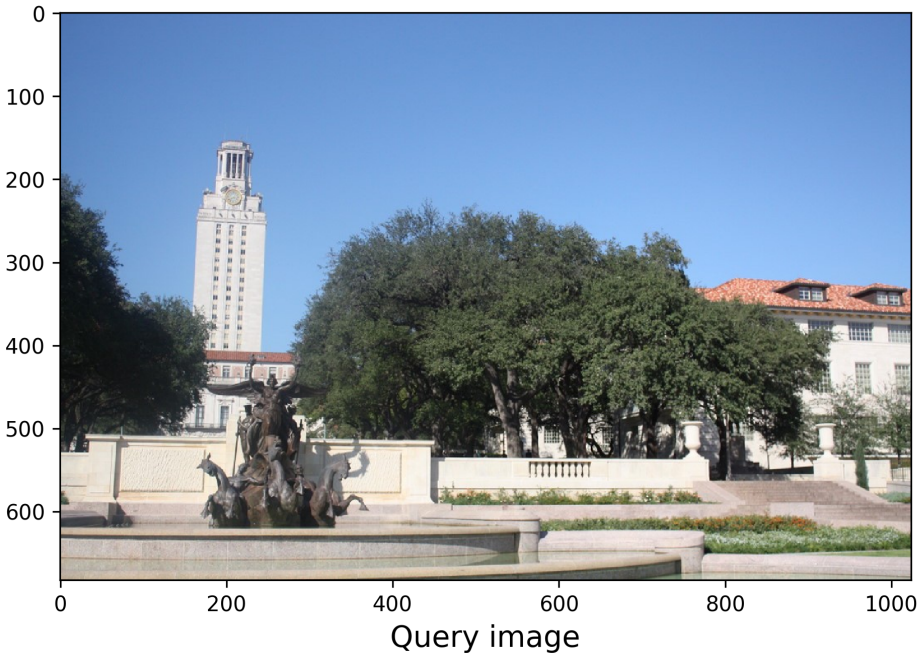
Image stitching is one of the most successful applications in Computer Vision. Nowadays, it is hard to find a cell phone or an image processing API that does not contain this functionality.

In this piece, we will talk about how to perform image stitching using Python and OpenCV. Given a pair of images that share some common region, our goal is to “stitch” them and create a panoramic image scene.

Throughout this article, we go over some of the most famous Computer Vision techniques. These include:

- Keypoint detection
- Local invariant descriptors (SIFT, SURF, etc)
- Feature matching
- Homography estimation using RANSAC

We explore many feature extractors like SIFT, SURF, BRISK, and ORB. You can follow along using this Colab notebook and even try it out with your pictures.



Feature Detection and Extraction

Given a pair of images like the ones above, we want to stitch them to create a panoramic scene. It is important to note that both images need to share some common region.

- Angle
- Spacial position
- Capturing devices

The first step in that direction is to extract some key points and features of interest. These features, however, need to have some special properties.

Let's first consider a simple solution.

Keypoints Detection

An initial and probably 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 based on some measure of similarity like Euclidean distance. As we know, corners have one nice property: **they are invariant to rotation**. It means that, once we detect a corner, if we rotate an image, that corner will still be there.

However, what if we rotate then scale an 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!

Keypoints and Descriptors.

Methods like SIFT and SURF try to address the limitations of corner detection algorithms. Usually, corner detector algorithms use a fixed size kernel to detect regions of interest (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 differently scaled versions of the same image. It also uses the neighboring pixel information to find and refine key points and corresponding descriptors.

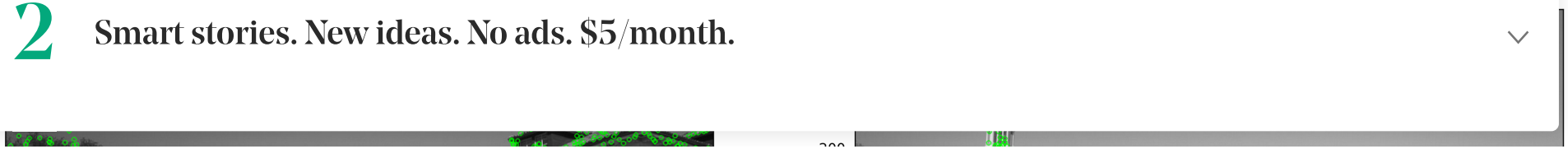
To start, we need to load 2 images, a query image, and a training image. Initially, we begin by extracting key points and descriptors from both. We can do it in one step by using the OpenCV *detectAndCompute()* function. Note that in order to use *detectAndCompute()* we need an instance of a keypoint detector and descriptor object. It can be ORB, SIFT or SURF, etc. Also, before feeding the images to *detectAndCompute()* we convert them to grayscale.

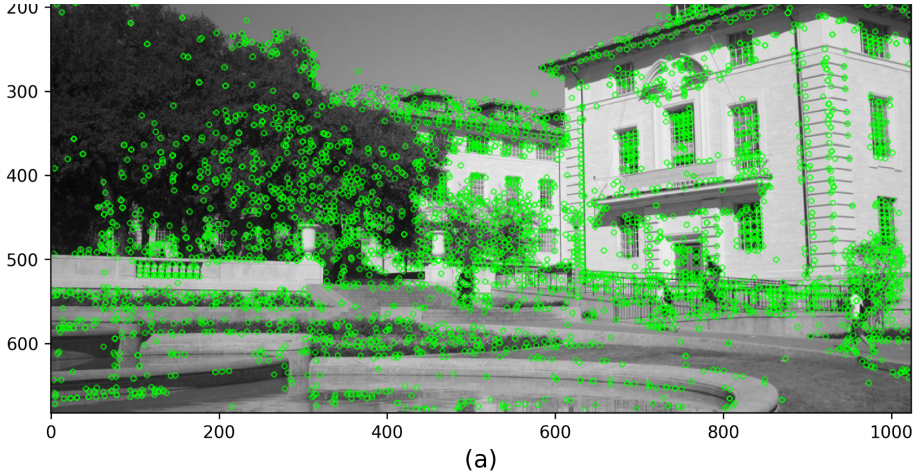
```
1  def detectAndDescribe(image, method=None):
2      """
```

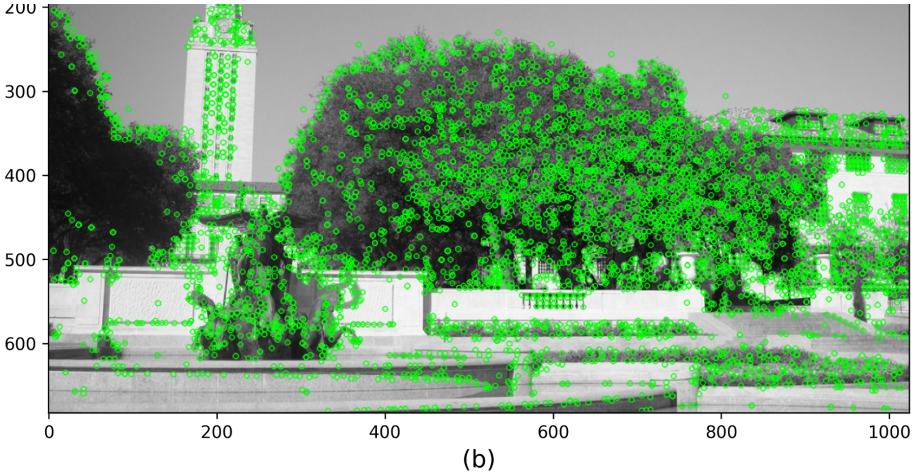
```
    # detect and extract features from the image
    if method == 'sift':
        descriptor = cv2.xfeatures2d.SIFT_create()
    elif method == 'surf':
        descriptor = cv2.xfeatures2d.SURF_create()
    elif method == 'brisk':
        descriptor = cv2.BRISK_create()
    elif method == 'orb':
        descriptor = cv2.ORB_create()

    # get keypoints and descriptors
    (kps, features) = descriptor.detectAndCompute(image, None)

    return (kps, features)
```

detectAndDescribe.py hosted with ❤ by GitHub view raw

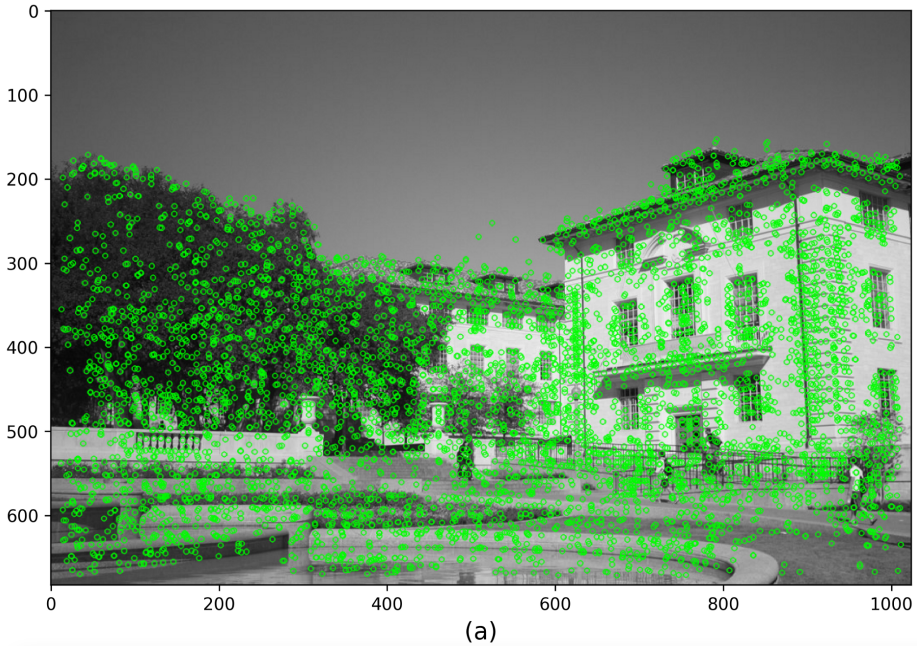
We run *detectAndCompute()* on both, the query and the train image. At this point, we have a set of key points and descriptors for both images. If we use SIFT as the feature extractor, it returns a 128-dimensional feature vector for each key point. If SURF is chosen, we get a 64-dimensional feature vector. The following images show some of the features extracted using SIFT, SURF, BRISK, and ORB.

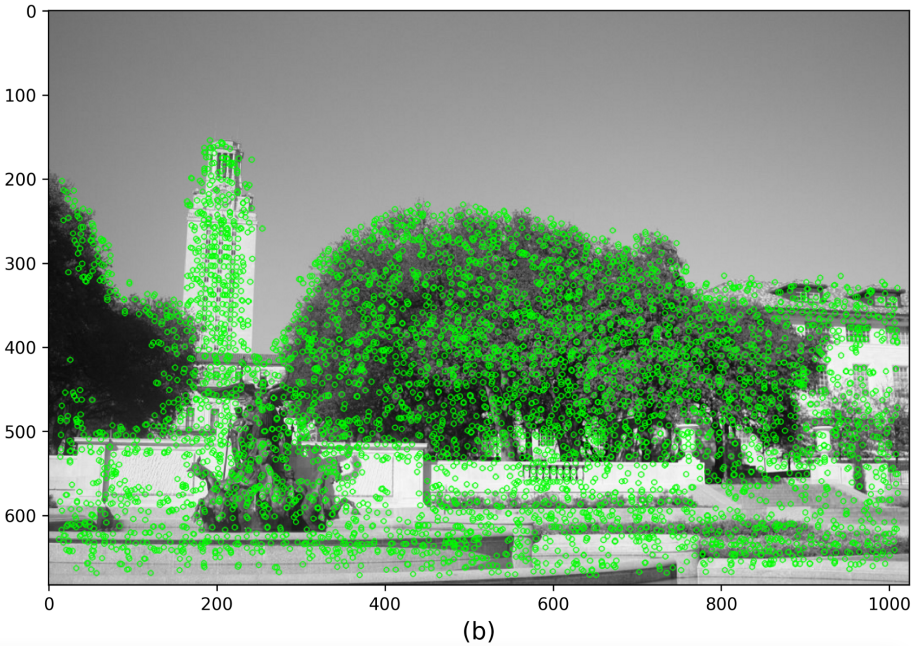


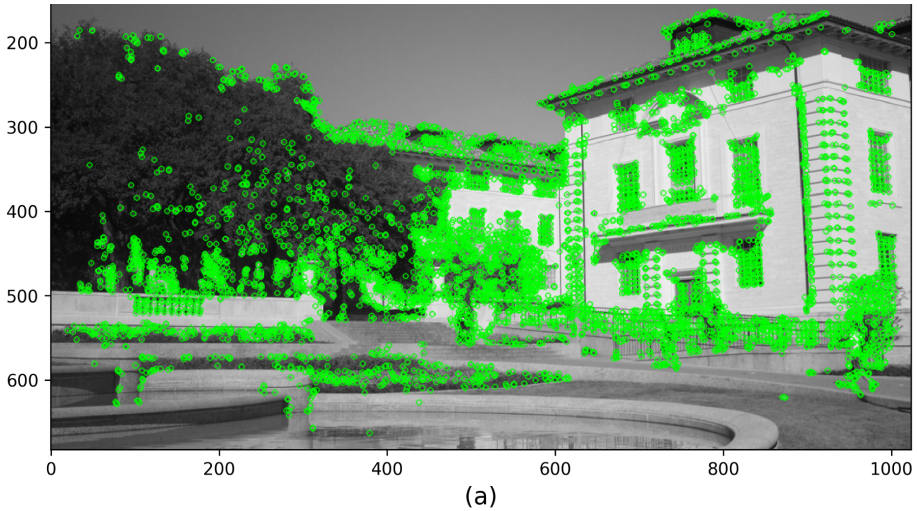


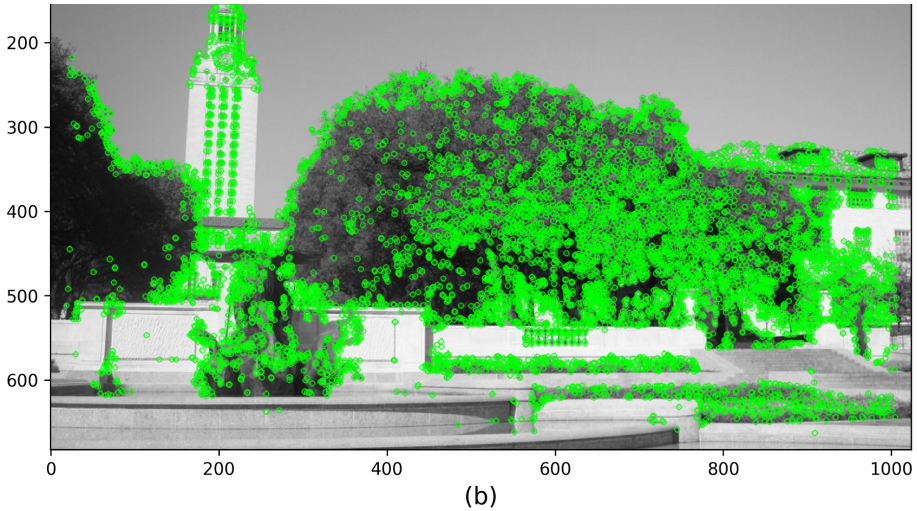


Detection of key points and descriptors using SIFT

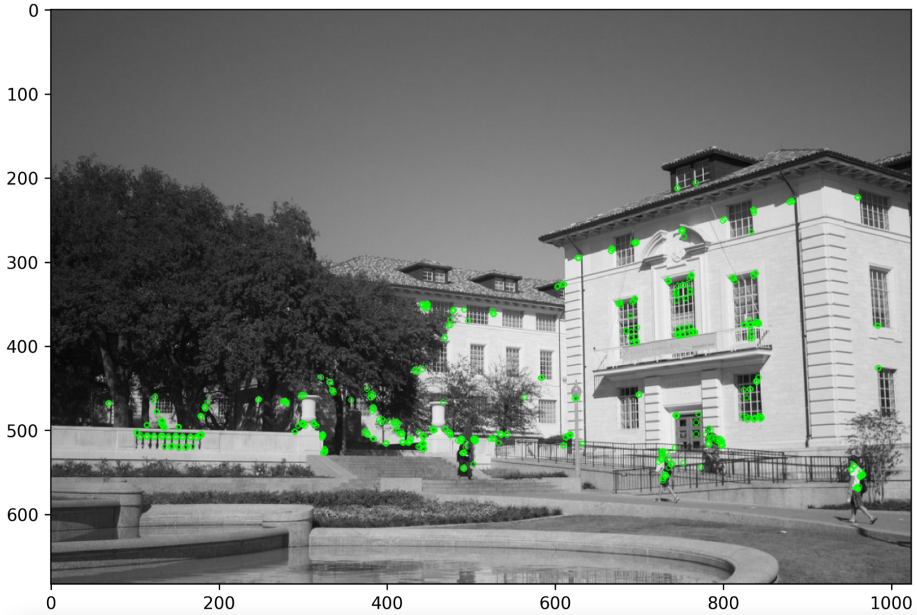


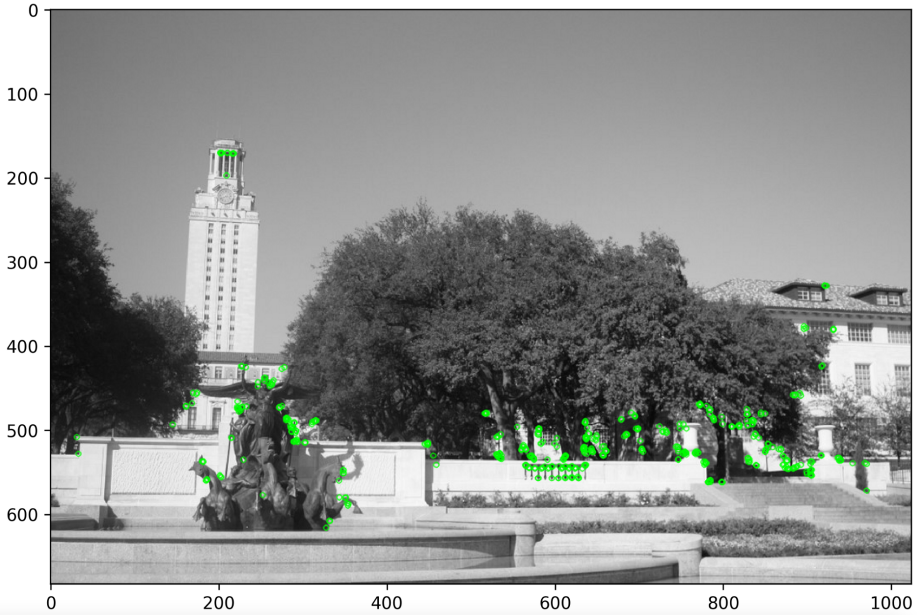






Detection of key points and descriptors using BRISK and Hamming distances.





Feature Matching

As we can see, we have a large number of features from both images. Now, we would like to compare the 2 sets of features and stick with the pairs that show more similarity.

With OpenCV, feature matching requires a Matcher object. Here, we explore two flavors:

- Brute Force Matcher
- KNN (k-Nearest Neighbors)

The BruteForce (BF) Matcher does exactly what its name suggests. Given 2 sets of features (from image A and image B), each feature from set A is compared against all features from set B. By default, BF Matcher computes the **Euclidean distance between two points**. Thus, for every feature in set A, it returns the closest feature from 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 need to specify 2 parameters. The first is the distance metric. The second is the *crossCheck* boolean parameter.

```
def createMatcher(method,crossCheck):
```

```
    elif method == 'orb' or method == 'brisk':
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=crossCheck)
    return bf
```

createMatcher.py hosted with ❤️ by GitHub view raw

The *crossCheck* bool parameter indicates whether the two features have to match each other to be considered valid. 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.

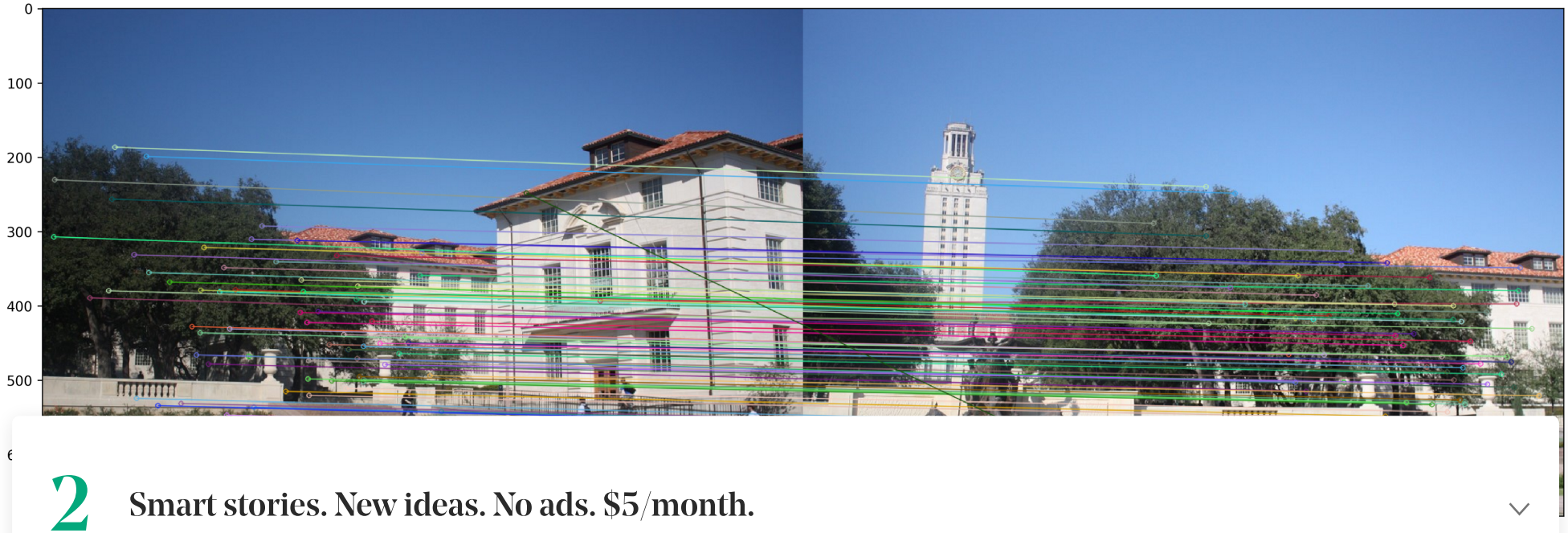
However, for cases where we want to consider more than one candidate match, we can use a KNN based matching procedure.

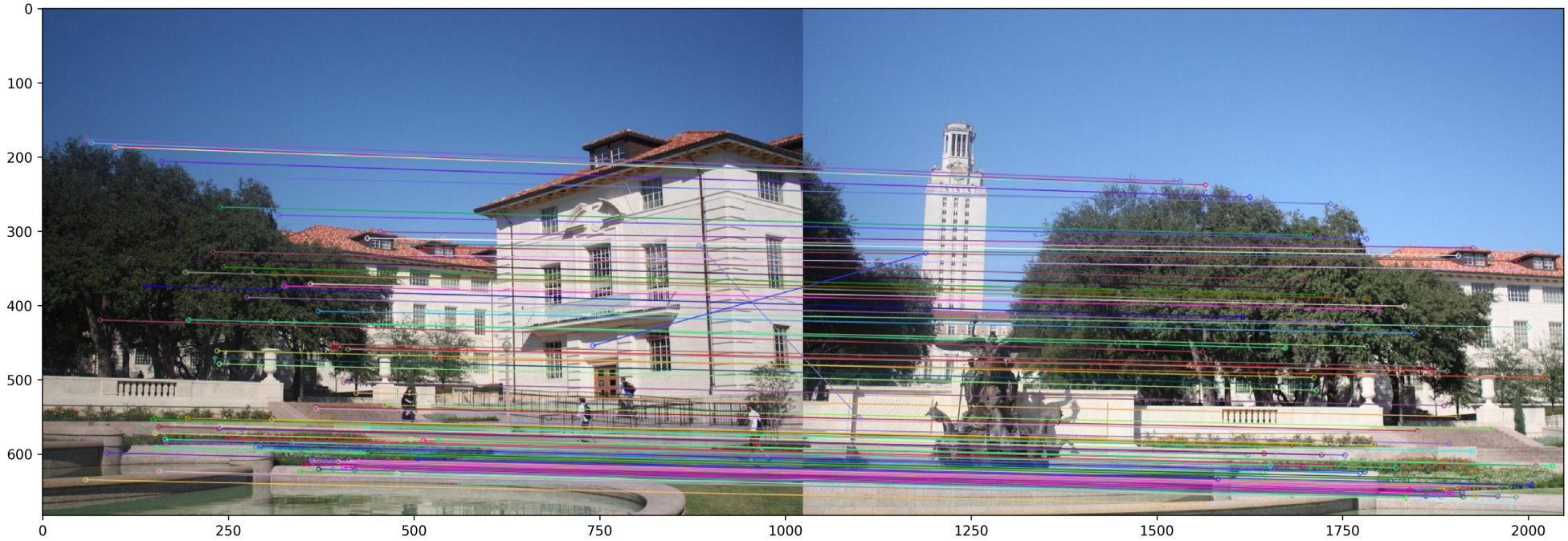
> Instead of returning the single best match for a given feature, KNN returns the k best matches.

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

To make sure the features returned by KNN are well comparable, the authors 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 throw it away. Also, the ratio value must be chosen manually.

In essence, ratio testing does the same job as the cross-checking option from the BruteForce Matcher. Both, ensure a pair of detected features are indeed close enough to be considered similar. The 2 figures below show the results of BF and KNN Matcher on SIFT features. We chose to display only 100 matching points to clear visualization.





Feature matching using Brute Force Matcher on SIFT features

Note that even after cross-checking for Brute force and ratio testing in KNN, some of the features do not match properly.

Nevertheless, 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.

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.

Estimating the Homography

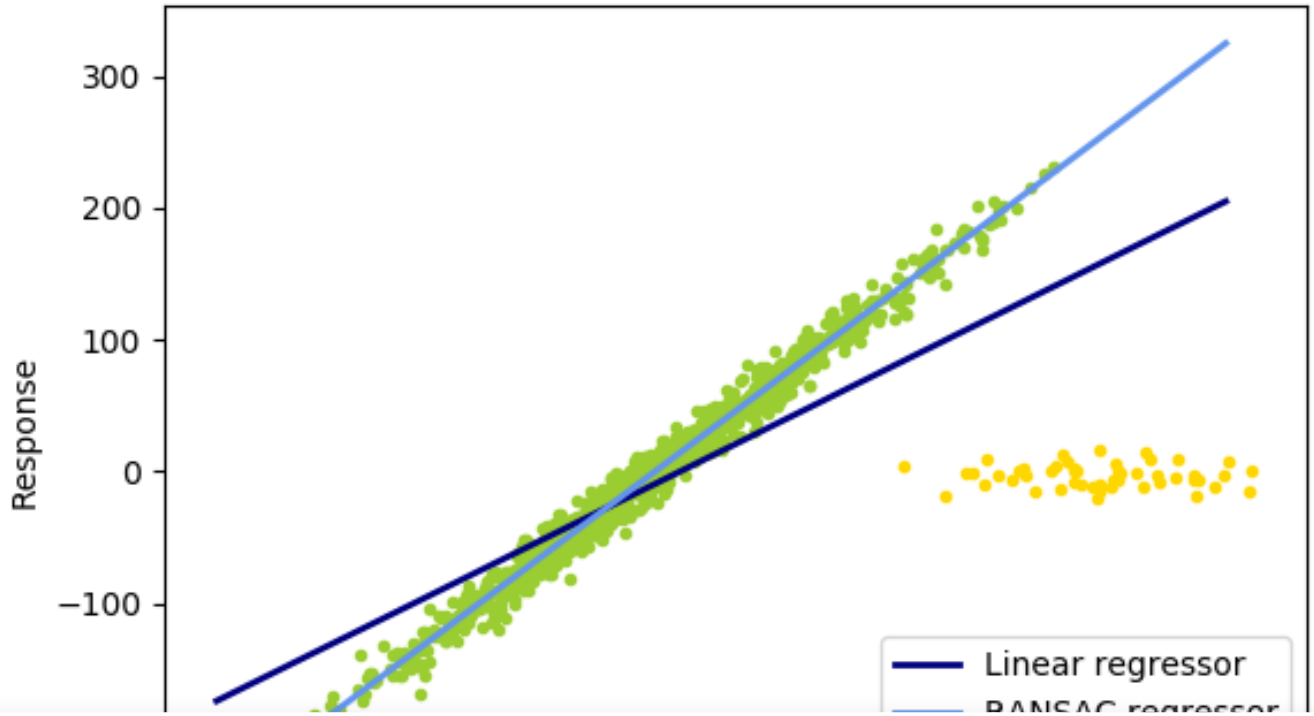
RANdom SAmple Consensus or RANSAC is an iterative algorithm to fit linear models. Different from other linear regressors, RANSAC is designed to be robust to outliers.

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.

RANSAC solves this problem by estimating parameters only using a subset of **inliers** in the data. The figure below shows a comparison between Linear Regression and RANSAC. First, note that the dataset 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. And that includes outliers.

This characteristic is very important to our use case. Here, 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 filter points that clearly belong to the data distribution from the ones which do not.



Comparison between Least Squares and RANSAC model fitting. Note the substantial number of outliers in the data.

Once we have the estimated Homography, 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.

```
# Apply panorama correction
width = trainImg.shape[1] + queryImg.shape[1]
height = trainImg.shape[0] + queryImg.shape[0]

result = cv2.warpPerspective(trainImg, H, (width, height))
result[0:queryImg.shape[0], 0:queryImg.shape[1]] = queryImg

plt.figure(figsize=(20,10))
```

panorama_correction.py hosted with ❤ by GitHub view raw

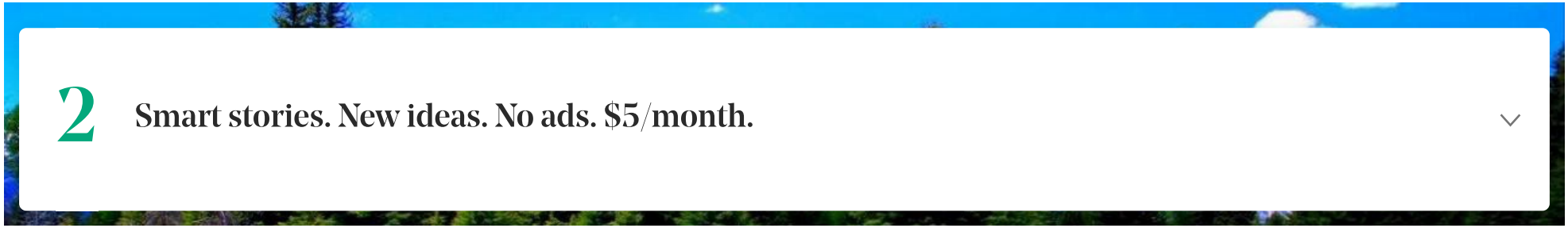
The resulting panorama image is shown below. As we see, there are a couple of artifacts in the result. More specifically, we can see some problems related to lighting conditions and edge effects at the image boundaries. Ideally, we can perform post-processing techniques to normalize the intensities like **histogram matching**. This would likely make the result look more realistic.

Thanks for reading!

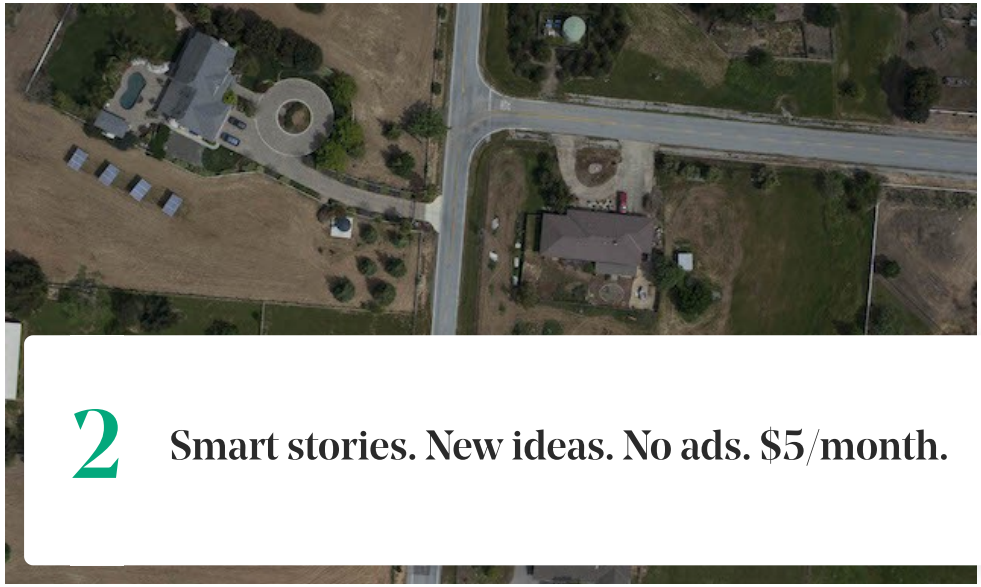
Input image pair

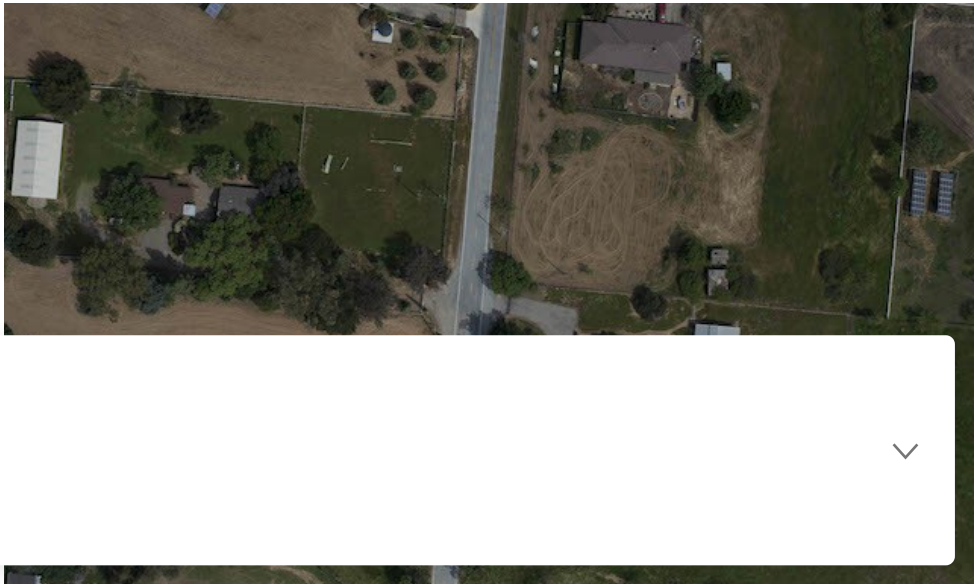
Panoramic Image

Input image pair



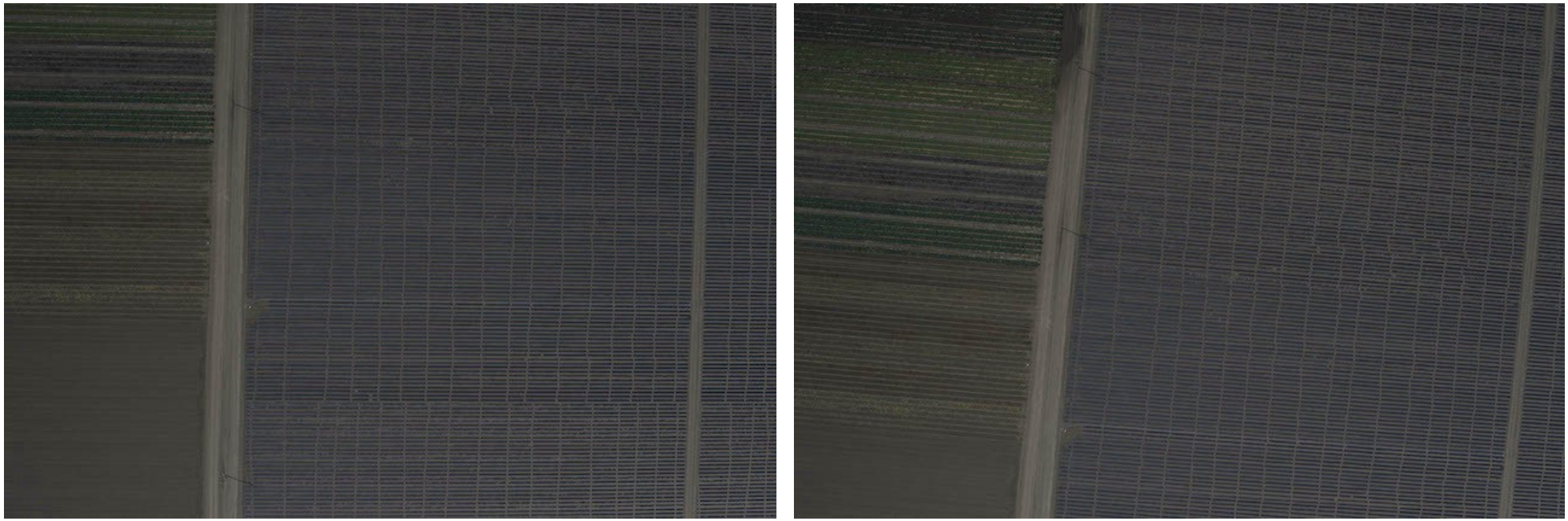
Panoramic Image



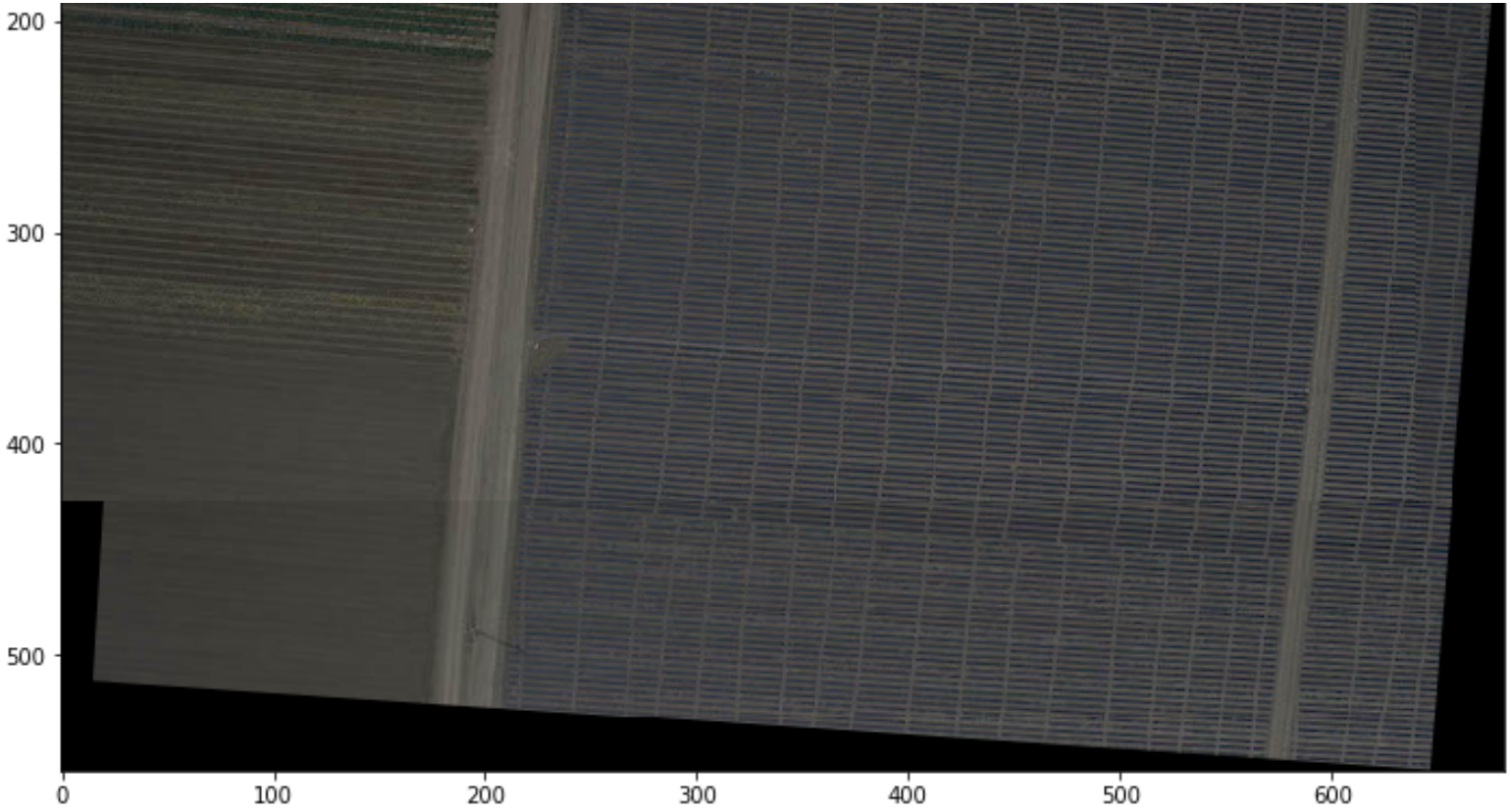


Input image pair

Panoramic Image



Input image pair.



Panoramic Image

Machine Learning | Computer Vision | Python | Image Processing | Towards Data Science