Assignment 02: Image Stitching

Maheep[†], Lim Yuam Qing[†], Ye Weijian[†]

Matriculation Number: G2303665G, G2101988B, G2204819E

Computer Science and Engineering
Nanyang Technological University
Singapore
September, 14, 2023

Image Stitching

Maheep[†], **Lim Yuam Qing**[†], **Ye Weijian**[†]
Nanyang Technological University
{maheep001, ylim170, ye0023an}@e.ntu.edu.sg
[†] equal contribution

https://github.com/AiRiFiEd/AI6121_ASSIGNMENT_2.git

Abstract

This assignment is centered around achieving image stitching to generate a panoramic view from multiple images. We employ various OpenCV functions in the process, such as SIFT for robust feature detection and FlannBasedMatcher for feature matching, along with its alternative versions. The final step involves stitching the images using blending and warping techniques. We conducted extensive experiments by adjusting blending values and implementing diverse approaches for feature matching and homography. These include using LMED and RANSAC for homography, as well as employing FLANN, Bruteforce, and different distance metrics such as L1, L2, and L2SQR for feature matching. We found feature points detected using SIFT, matched using Flann-based matcher and warped with homography found using RANSAC algorithm generated good stitching results. Linear blending operator also worked well for all but 1 of the image pairs, with weights found by empirical trial and error.Furthermore, this report also provide the implementation of the code, which could be easily accessed using the link: https://github.com/AiRiFiEd/AI6121_ASSIGNMENT_2.git.

1 Introduction

Image Stitching, also known as panoramic stitching or panorama creation, is a process frequently associated with creating panoramic images. It seamlessly merges images to provide a wider field of view than what a single photograph can capture independently. This technique finds applications in various domains, including photography, virtual reality, and medical imaging. This report focuses on generating panoramic images through image stitching using image pairs. While the task may appear straightforward, it involves aligning images that may share common features. Additionally, variations in brightness or scaling due to different shooting conditions can further complicate the process, presenting a challenge in Image Stitching. The task can be broken down into 4 main steps: *Feature Detection, Image Mapping*, and *Computing Homography*. Feature Detection involves identifying distinctive features in different images that remain consistent under various transformations, such as rotation. These features encompass elements like blobs, corners, and edges, which are crucial for aligning and blending the images, serving as the initial step in the process. The second step involves matching these identified features in images of the same scene, which will eventually be stitched together. These matched features are robust to changes and signify areas or pixels that are common in both images, serving as a reference for the stitching process. The third step encompasses computing the homography for the images, which involves transforming the images to correct distortions and achieve precise alignment. These transformations are achieved through linear operations that perform geometric adjustments. Finally, the transformed images are warped and blended to create a cohesive stitched image. The warping process involves generating a new image by projecting vectors from the original image onto the new one. The blending process seamlessly combines the two images, ensuring a smooth transition from one image to the other in the final stitched result. This is achieved by combining pixels using a weighted average, with weights determined by a blending mask that prioritizes pixels in the center of the overlap. We have used different functions to accomplish each step in order to achieve the most sophisticated image stitching. This report mainly discuss all the major implementations and results in the "Implementation" (Section 2), while concluding about the paper in the "Conclusion" (Section **??**). The Implementation section delves into the various steps taken to achieve image stitching, encompassing Feature Detection, Feature Matching, Computing Homography, and Image Stitching. In addition

to implementing the basic of accomplishing the task we go thorough various experimentation's from each perspective, including blending, homography, and feature matching. This allows for a comprehensive evaluation of the techniques, offering insights into the impact of various functions on image stitching. Readers interested in the improvements and additional functions can directly refer to Section 3. Finally, Section **??** provides an outlook of the authors on the paper and what might be some future directions and limitation of the implemented techniques.

2 Implementation

This section aims to provide readers with a comprehensive understanding of each module used in image stitching. We commence by delving into the "Feature Detection" module, where we explain the intricate workings of feature detection. This includes a detailed explanation of the underlying mechanism, accompanied by the presentation of key code snippets and results. Similarly, in the "Feature Matching" module, we address the process of how features identified in the feature detection module are matched to facilitate image stitching. Additionally, given that different images may possess varying properties, such as brightness, despite having the same content, we dedicate a section to "Computing Homography" to account for these variations. Finally, in the concluding section on Image Stitching, we elucidate the functionalities of warping and subsequently blending the images. We also explore alternative operations and present their corresponding outcomes.

2.1 Feature Detection

The task of Feature Detection is focused on the detecting the features of the images that are robust to various changes, like rotation, scale changes, and changes in lighting conditions. The features mostly consist of corners, edges or other other distinctive landmarks in the image. The feature detection returns the keypoints and descriptors. The keypoints describe the specific points in the image, whereas the descriptors are the numerical representation of the image, defining the image features. There are various methods that could be used to execute the feature detection. We have implemented the SIFT method, which stands for Scale-Invariant Feature Transform. It uses a Gaussian filter with varying standard deviations (σ) to create multiple scale pyramids of different blurred images. These are then subtracted from each other to obtain the difference-of-Gaussian (DoG) images, producing potential keypoints. Furthermore, to improve the localization accuracy it fits a quadratic function to the nearby pixels to find the exact location of the keypoint. To make the extracted keypoints invariant, it assigns an orientation to each keypoint based on the local gradient directions. After applying the feature detection, we drew the different detected keypoints in the image.

It is implemented using the customized function by this study, as given below:

```
def sift(self) -> None:
  sifter = cv2.xfeatures2d.SIFT_create()
  self.keypoints, self.descriptors = sifter.detectAndCompute(self.original_image,
                                                             None)
  self.annotated_image = np.copy(self.original_image)
  cv2.drawKeypoints(
      self.gray, self.keypoints, self.annotated_image
  )
```

We obtain the following images for keypoints detected after feature detection, using the Figure 1, 2, 3, and 4. These are presented as gray-scaled images to better show the keypoints that were detected by the algorithm.

As can be seen from the figures, SIFT managed to detect multiple scale-invariant keypoints in each of the sample images that can be used for matching subsequently. While all the sample images have reasonable resolution (without obvious blurring) the density of feature points detected for each pair seems to vary visually. Hence, we computed the average number of feature points detected per pixel (Table 1) out of curiosity to explore if that would affect the performance of subsequent matching and stitching of the respective image pairs.

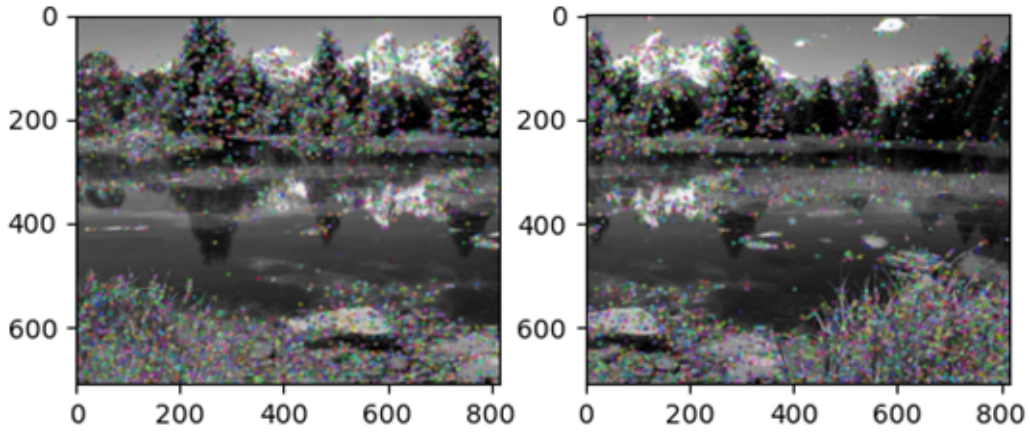


Figure 1: Feature Detection for Sample Image image pairs_01

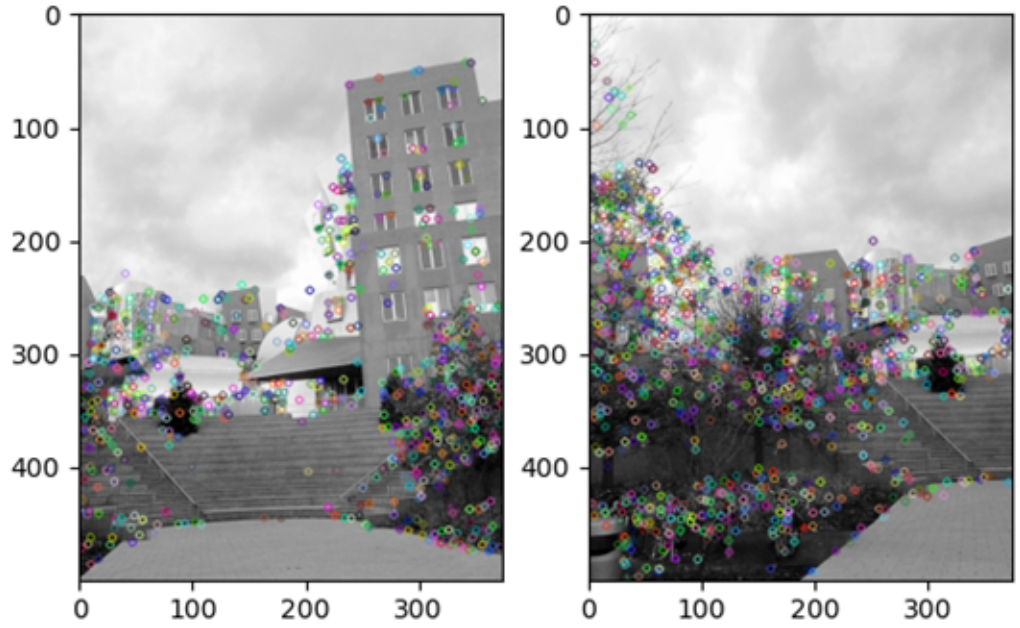


Figure 2: Feature Detection for Sample Image image pairs_02

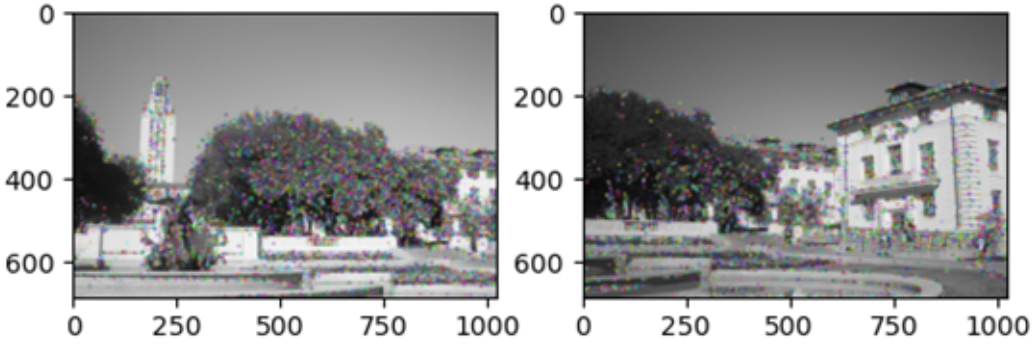


Figure 3: Feature Detection for Sample Image image pairs_03

2.2 Feature Matching

Feature Matching is the process of identifying the same pixels in images. These images usually consist of images capturing the same view with some commonality in them. The matched points are identified

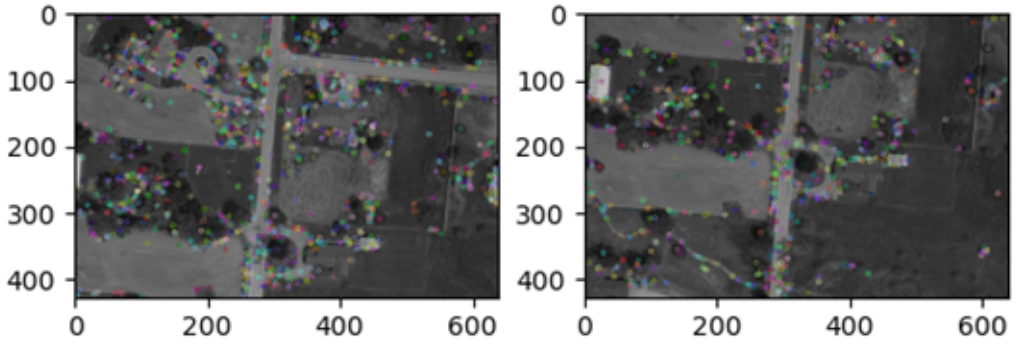


Figure 4: Feature Detection for Sample Image image pairs_04

Image Name	**Shape**	**Size**	**# of keypoints Detected**	**# of Keypoints Detected Per Pixel**
image_pairs_01_01	(710, 818, 3)	1742340	5911	0.00339
image_pairs_01_02	(709, 816, 3)	1735632	5425	0.00313
image_pairs_02_01	(500, 375, 3)	562500	805	0.00143
image_pairs_02_02	(500, 375, 3)	562500	1127	0.00200
image_pairs_03_01	(683, 1024, 3)	2098176	5172	0.00246
image_pairs_03_02	(683, 1024, 3)	2098176	4201	0.00200
image_pairs_04_01	(427, 640, 3)	819840	1094	0.00133
image_pairs_04_02	(427, 640, 3)	819840	787	0.00096

Table 1: Average Number of Feature Points Detected-per-Pixel

such that they are robust to change, especially consisting of features consisting of corners, edges, or other distinctive landmarks. They are robust in sense of being invariant under different transformations like rotation, scale changes, and changes in lighting conditions. There are various feature matching variants, we have particularly focused on four kinds of feature matching: FLANN, and various variant of BruteForce−L1, L2, L2SQR.

BruteForce Matcher takes the descriptor as inputs, which are array of numbers describing the feature of the image. These are used to compare the images to match the same features. The method can be implemented using the code. We have implemented different variants of the BruteForce Matcher, including the L1, L2 and L2SQR distances, which are defined more extensively below.

```
class MatcherBruteForce(Matcher):
    def __init__(self,
                 norm_type: str = 'norm_l2',
                 cross_check: bool = False,
                 k: int = 2) -> None:
        self.norm_type = norm_type
        self.cross_check = cross_check
        self.k = k

        self.matcher = cv2.BFMatcher(self._enum, self.cross_check)
        self.matches = tuple()

    @property
    def _enum(self) -> int:
        if self.norm_type == 'norm_l2':
            return cv2.NORM_L2
        elif self.norm_type =='norm_inf':
            return cv2.NORM_INF
        elif self.norm_type =='norm_l1':
            return cv2.NORM_L1
```

```
        elif self.norm_type =='norm_l2sqr':
            return cv2.NORM_L2SQR
        elif self.norm_type =='norm_hamming':
            return cv2.NORM_HAMMING
        elif self.norm_type =='norm_hamming2':
            return cv2.NORM_HAMMING2

    def match(self,
              descriptor_1: np.ndarray,
              descriptor_2: np.ndarray) -> Tuple:
        self.matches = self.matcher.knnMatch(descriptor_1,
                                             descriptor_2,
                                             self.k)
        return self.matches
```

There are various distances that can be used to compute the bruteforce matcher, such as Manhattan Distance or L1, Euclidean Distance L2 and L2SQR. The Manhattan Distance is the distance between two points, in an assumed grid-based structure. It is also known as L1 distance and can be computed using the Equation 1 between the two points: $P(x_1, y_1)$, and $Q(x_2, y_2)$.

$$\text{Manhattan Distance } (L1) = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

Similarly, the euclidean distance can be interpreted as the line distance between two points. It can be interpreted as the shortest distance between two points. For deriving it extensively, we will use the above points, i.e. $P(x_1, y_1)$, and $Q(x_2, y_2)$ using the Equation 2.

$$\text{Euclidean Distance } (L2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2}$$

Moreover, the L2SQR distance can be defined as the square of the above defined euclidean distance. It is often used in computational tasks for various reasons as it can simplify certain mathematical operations, making computations more efficient. In other words, it avoids computationally expensive square root operation involved in calculating the actual Euclidean distance. It can be defined using the Equation 3, for the two points taken above, i.e. $P(x_1, y_1), Q(x_2, y_2)$ or more generally using Equation 4.

$$L2SQR = (x_1 - x_2)^2 + (y_1 - y_2)^2 \tag{3}$$

$$L2SQR = (x_1 - x_2)^2 + (y_1 - y_2)^2 + \cdots \tag{4}$$

FLANN stands for Fast Library for Approximate Nearest Neighbors. We used FlannBasedMatcher in opencv using the function `MatcherFlannBased`$(\cdot,\cdot)$ for matching features of one image to another image. It uses Nearest Neighbors Approach and usually runs faster than Brute-Force-Matcher for various datasets. The `matcher.knnMatch(descriptor_1, descriptor_2, self.k)`$(\cdot,\cdot,\cdot)$ function is used to perform a k-nearest neighbor search to find the two nearest neighbors. The matches are then filtered using a distance ratio test. The implementation of FLANN is initialised by building data set index, leveraging the supported indexing methods. The nearest neighbour can then be performed by the used by querying the index, returning the index for nearest neighbors to the query point, along with their distances. Below we present the implemented code for FLANN based matcher:

```
class MatcherFlannBased(Matcher):
    def __init__(self,
                 index_params: Dict,
                 search_params: Dict,
                 k: int = 2) -> None:
        super().__init__('flann')
        self.index_params = index_params
        self.search_params = search_params
        self.k = k
```

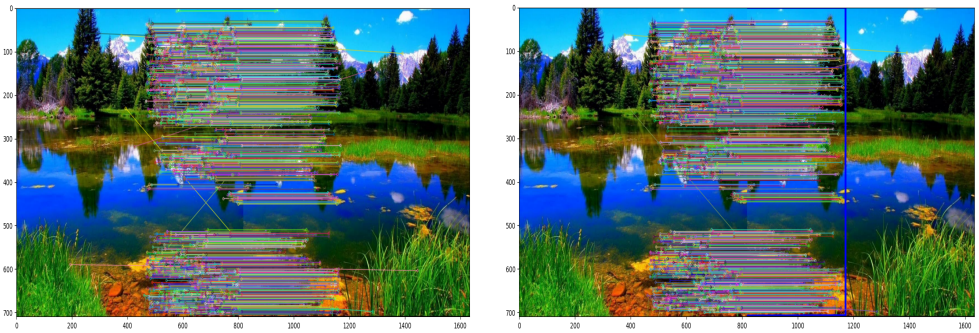
```
        self.matcher = cv2.DescriptorMatcher_create(
                       cv2.DescriptorMatcher_FLANNBASED)
        self.matches = tuple()

    def match(self,
              descriptor_1: np.ndarray,
              descriptor_2: np.ndarray) -> Tuple:
        self.matches = self.matcher.knnMatch(descriptor_1,
                                             descriptor_2,
                                             self.k)
        return self.matches
```

Good Matches	**FlannMatcher**	**BruteForceMatcher NormL1**	**BFMatcher NormL2**	**BFMatcher NormL2Sqr**
image_pairs_01	1982	2006	1983	2101
image_pairs_02	132	133	130	173
image_pairs_03	889	903	872	1165
image_pairs_04	265	273	264	323

Table 2: Different matchers and number of feature points

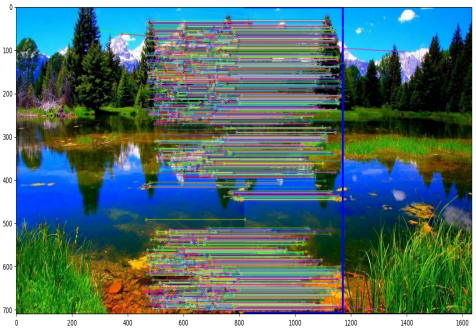
In implementing knnMatch for both the Flann-based and Brute Force matchers, we have chosen k to be equals to 2 in order to obtain the top 2 matches for each feature vector. In addition, we have applied a Lowe's ratio of 0.7 in an attempt to filter out only good quantity matches. Comparing the number of good matches obtained from each of the matching algorithm (with its corresponding distance measure), it can be seen that Brute Force matcher with NormL1 and NormL2 distance measures achieved around the same number of good matches as with Flann-based matcher. This is consistent with OpenCV's documentation recommending L1 and L2 norms for SIFT and SURF descriptors. It is interesting to note that Brute Force matcher with NormL2Sqr distance measure achieved a much higher number of good matches relative to the others.

However, visually inspecting the images (Figure 5) shows the use of NormL2Sqr distance measure resulted in extremely noisy matches despite satisfying Lowe's ratio test, and hence, does not seem to be approriate for the purpose of this assignment.

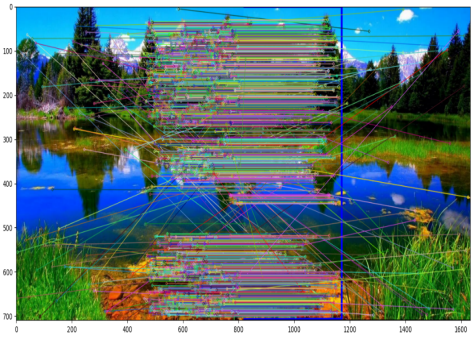


(a) flann

(b) norm L2



(c) norm L1



(d) norm L2sqr

Figure 5: Feature match of different matchers

In Figure 5, we can find that before homography, norm L2sqr matcher has more wrong feature matching though it has more feature points, and flann matcher has higher accuracy. Flann and norm L2 have better reusults, and norm L2sqr is not a good choice.

2.3 Computing Homography

Homography is computed to transform an image with respect to the points in another image, to align the two images. It is computed by forming a 3×3 transformation matrix, focusing to transform the features that are invariant such as the corner, edges and other features. Below have stated the algorithm used to compute it.

```
class Homographer(object):
    def __init__(self, method: str, minimum_match_count: int) -> None:
        self.method = method
        self.minimum_match_count = minimum_match_count

    def has_minimum_match(self, good_matches: List) -> bool:
        return len(good_matches) > self.minimum_match_count

    def find_homography(self, source_points: np.ndarray,
        destination_points: np.ndarray) -> Tuple:
        pass
```

This can account for different kind of transformation such as rotation, translation, scaling, skewing, and perspective distortion. To compute the homography, one needs at the minimum four pair of paints in the two images. These points are matched based on their visual features, such as corners or other distinctive landmarks. The homography matrix is then calculated using different techniques, such as LMEDS, RANSAC, etc. LMeDS stands for Least-Median of Squares and focuses to find a

transformation (homography) that aligns two sets of corresponding points, even in the presence of outliers or incorrect correspondences. It is executed using the code below:

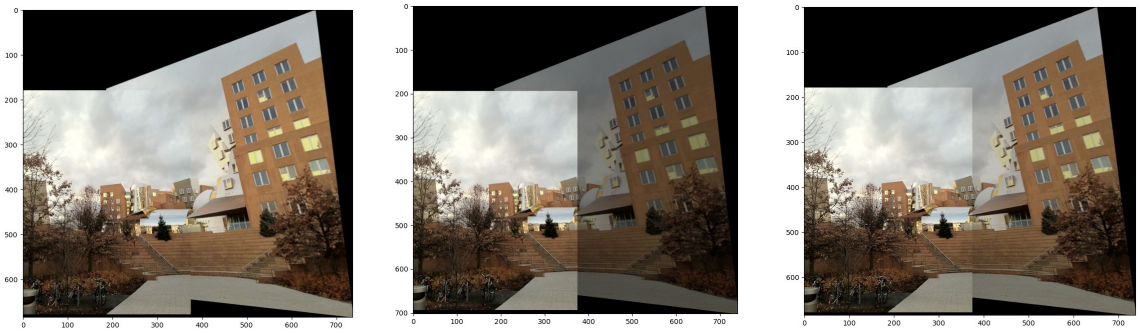
```
class HomographerLMeDS(Homographer):
    def __init__(self, minimum_match_count: int = 10) -> None:
        super().__init__('lmeds', minimum_match_count)
        self.matrix = np.empty((0,0))
        self.mask = np.empty((0,0))

    def find_homography(self, source_points: np.ndarray,
        destination_points: np.ndarray) -> Tuple:
        self.matrix, self.mask = cv2.findHomography(source_points,
                                    destination_points, cv2.LMEDS)
        return self.matrix, self.mask
```

In addition to image LMeDS, we also test our algorithm by computing the RANSAC, which stands for Random Sample Consensus. It is mostly to deal to refine the outlier points detected in the feature detection, which are labelled as same but are not same, creating erroneous data.

```
class HomographerRansac(Homographer):
    def __init__(self, minimum_match_count: int = 10,
            reprojection_threshold: int=3) -> None:
        super().__init__('ransac', minimum_match_count)
        self.reprojection_threshold = reprojection_threshold
        self.matrix = np.empty((0,0))
        self.mask = np.empty((0,0))

    def find_homography(self, source_points: np.ndarray,
        destination_points: np.ndarray) -> Tuple:
        self.matrix, self.mask = cv2.findHomography(source_points,
                                    destination_points, cv2.RANSAC,
                                    self.reprojection_threshold)
        return self.matrix, self.mask
```

2.4 Image Stitching

In attempting to stitch each image pair together, we first obtain a warped perspective of the left image relative to the right image, before placing both the warped and right image onto the same canvas. As can be expected, any resolution differences between the left and right image (such as brightness) would result in a artificial cut-over between the left and right image. We attempt to apply a linear blend operator to the warped perspective of the left image and the right image that uses a weighted sum of elements in each array. We first applied a weight of 0.5 to each of the 4 image pairs−this means to use 0.5×elements in the warped perspective + 0.5×elements in the right image. As can be seen from Figure 6, the choice of weight to be used depends on the actual difference in resolution between the 2 images of that particular image pair.



(a) FLANN, RANSAC, Blending $\alpha = 0.5$ (b) FLANN, RANSAC, Blending $\alpha = 0.75$ (c) FLANN, RANSAC, Blending $\alpha = 0.9$

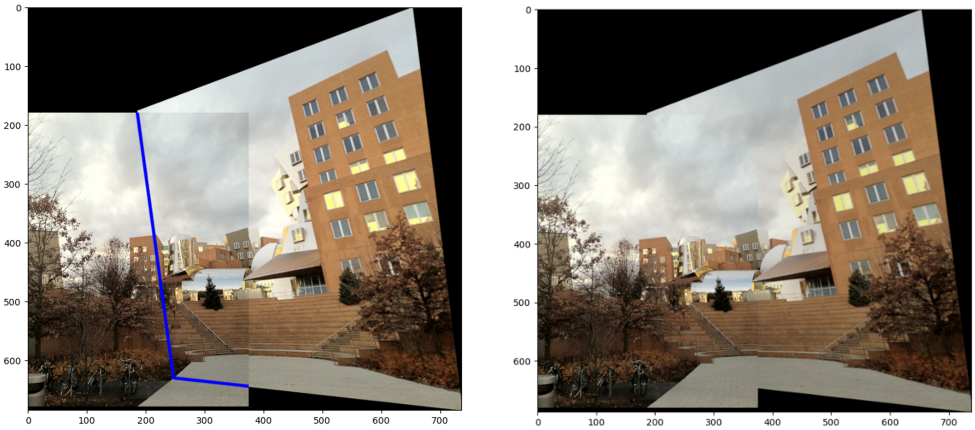
Figure 6: Comparison of different blending values without DoG, and in presence of FLANN, RANSAC

By trial and error, we have found the following weights to be appropriate for their respective image pair. We illustrate this using image pair 02 in Figure 7.

Image Name	**Warped Perspective**	**Right Image**
image_pairs_01	1.000	0.000
image_pairs_02	0.905	0.095
image_pairs_03	0.800	0.200
image_pairs_04	0.900	0.100

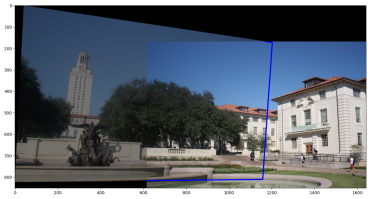
Table 3: Blending Weights

However, it is worth noting that changing the weights could not completely blend the warped perspective of the left image with that of the right image for image pair 03 (Figure 8). Visual inspection of the source image suggests this could be due to the fact that image pair 03 consists of images that differ not only in brightness but also in contrast. The application of a linear blending operator could likely blend image pairs that differ in brightness but may prove to be of limited effectiveness once images differ in contrast as well since the multiplication of a constant to the entire image matrix can only change its brightness. In the case of image pair 03, more complex blending methods such as multi-band may be necessary.



(a) FLANN, RANSAC, Blending $\alpha = 0.500$ (b) FLANN, RANSAC, Blending $\alpha = 0.905$

Figure 7: Results from using Optimal Alpha Found with Trial and Error for Image Pair 02



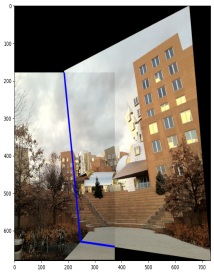
(a) FLANN, RANSAC, Blending $\alpha = 0.500$

(b) FLANN, RANSAC, Blending $\alpha = 0.800$

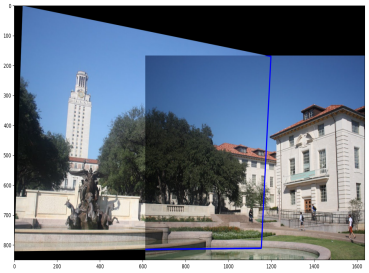
Figure 8: Results from using Optimal Alpha Found with Trial and Error for Image Pair 03

3 Experimentation & Results

(a) img1



(b) img2

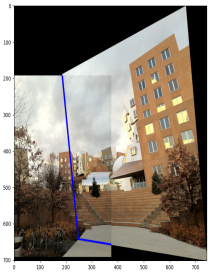


(c) img3

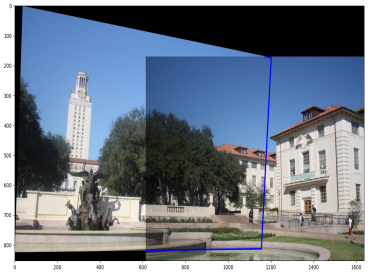
(d) img4

Figure 9: FLANN, RANSAC, No Blending

(a) img1



(b) img2

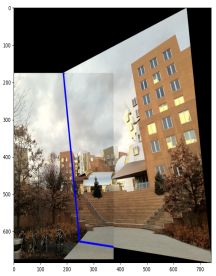


(c) img3

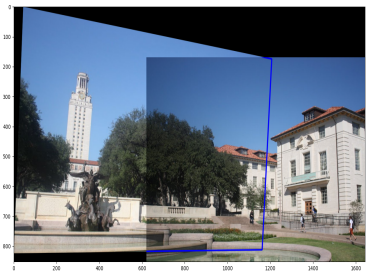
(d) img4

Figure 10: FLANN, LMeDS, No Blending

(a) img1



(b) img2

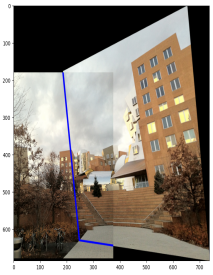


(c) img3

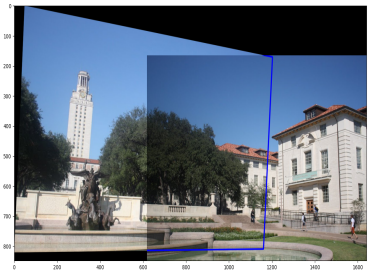
(d) img4

Figure 11: BruteForceNormL1, RANSAC, No Blending

(a) img1



(b) img2

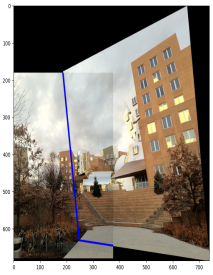


(c) img3

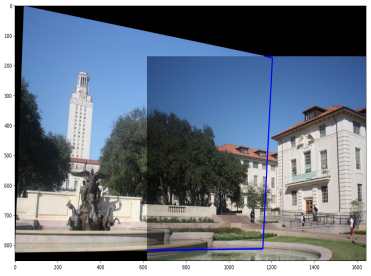
(d) img4

Figure 12: BruteForceNormL2, RANSAC, No Blending

(a) img1



(b) img2

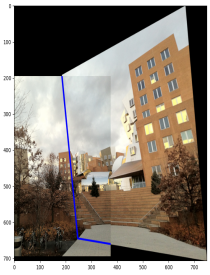


(c) img3

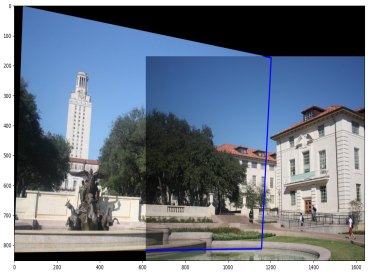
(d) img4

Figure 13: BruteForceNormL2Sqr, RANSAC, No Blending

(a) img1



(b) img2

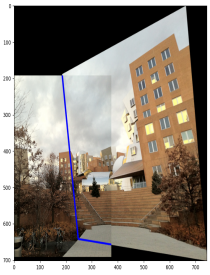


(c) img3

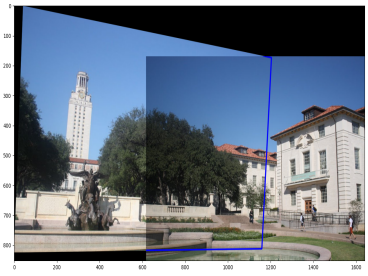
(d) img4

Figure 14: BruteForceNormL1, LMeDS, No Blending

(a) img1



(b) img2

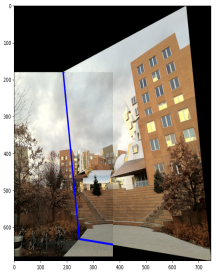


(c) img3

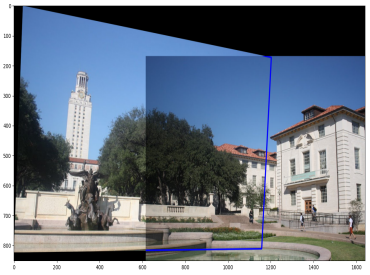
(d) img4

Figure 15: BruteForceNormL2, LMeDS, No Blending

(a) img1



(b) img2



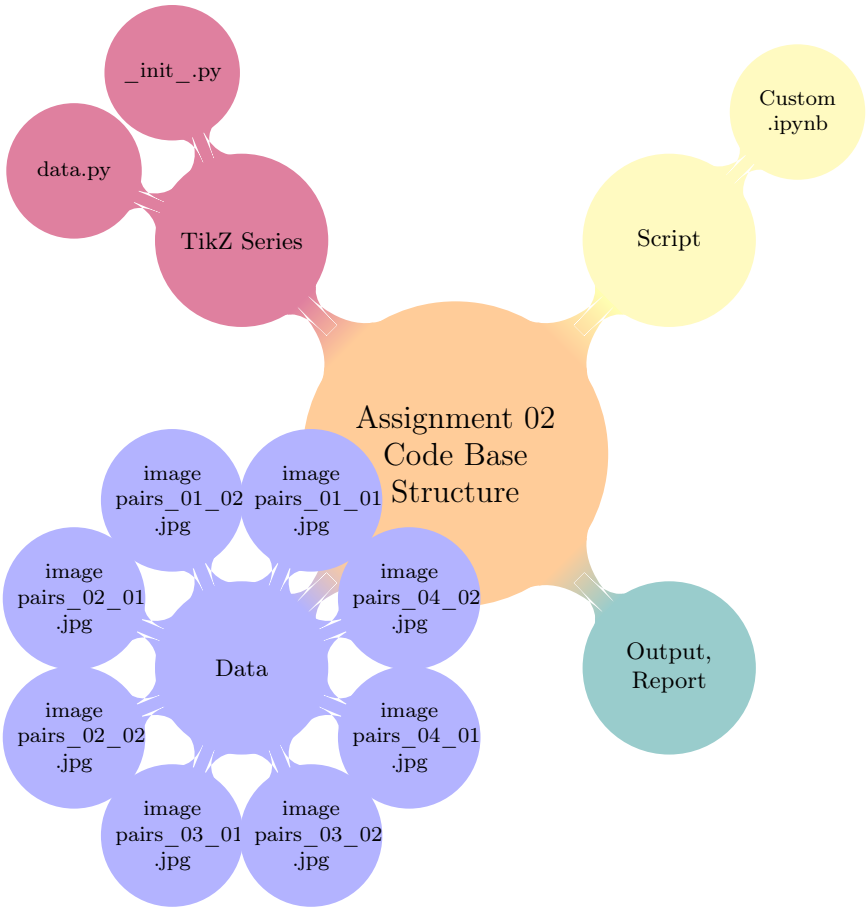
(c) img3

(d) img4

Figure 16: BruteForceNormL2Sqr, LMeDS, No Blending

4 Appendix

```
assignment_2
    data
       image pairs_01_01.jpg
       image pairs_01_02.jpg
       image pairs_02_01.jpg
       image pairs_02_02.jpg
       image pairs_03_01.jpg
       image pairs_03_02.jpg
       image pairs_04_01.jpg
       image pairs_04_02.jpg
    output
    report
    script
       custom.ipynb
    src
        _init_.py
        data.py
```



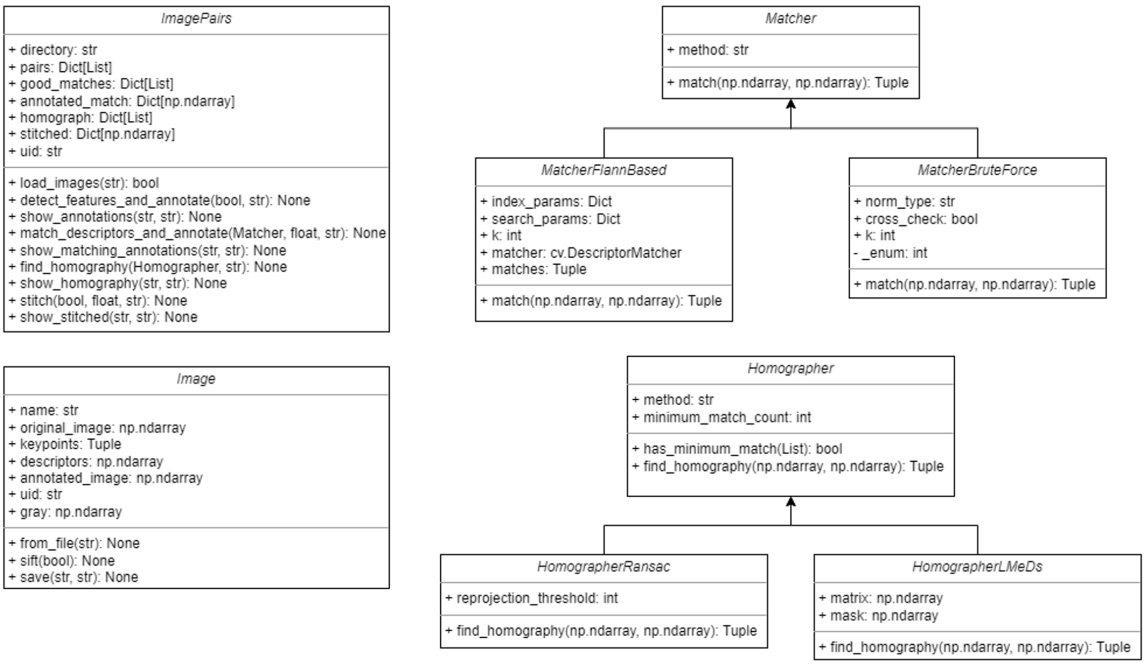


Figure 17: Detailed Structure of the Code