EE5731 Visual Computing

Assignment 1: Panoramic Image Stitching

Student Name: Nie Hongtuo

Student ID: A0224712N

9 Oct 2020

Part 1: 2D Convolution

Implementation

In this section we should write our own 2D convolution function to replace the **`conv2`** function in MATLAB. The MATLAB function **`conv2 = (A,B,shape)`** has a parameter to set the subsection of convolution, which can be 'full', 'same' or 'valid'. By choosing 'same' it returns the central part of the convolution, which is the same size as A. This is what we need in this task. I implemented my convolution function in **`my_conv2`**. It uses the same zero padding method like **`conv2`** and gives the same result as **`conv2`**. This can be verified using this piece of code in the script:

```
%% verify the correctness of my_conv2 function
a = rand(4,5);
b = rand(3,3);
c = conv2(a,b,'same');
d = my_conv2(a,b);
diff_conv = sum(sum(abs(c-d))); % usually diff_conv < 1e-15
```

Also, I generated my Sobel and Gaussian kernel, and the 5 basic Haar-like masks. The Sobel vertical and horizonal kernel are both 3x3. The Gaussian kernel is built with the help of **`meshgrid`**. To verify my Gaussian kernel is correct, I wrote this piece of code for a test:

```
%% verify the correctness of our Gaussian kernel
% with MATLAB predefined 2-D filter function fspecial
g = fspecial('gaussian', [m n], sigma);
diff_gaussian = sum(sum(abs(gaussian-g)));
```

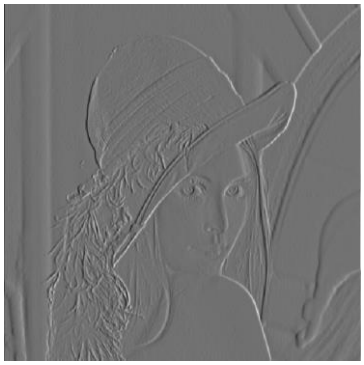
The Haar-like masks are scalable. Also, I wrote **`my_norm`** to normalize the matrix to form a uint8 image for display. The main script is **`p1_2d_convolution`**. Here are the results.

Results

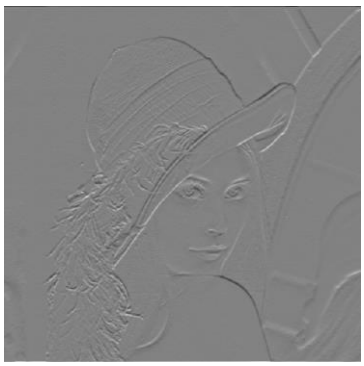
By enlarging the size of Gaussian kernel, it covers a boarder area on the image, thus make the image more blurring. Also, due to zero-padding, the edge becomes darker when we set a large Gaussian kernel, since it takes more zeros from the padding area outside the image.

Original image

Sobel vertical kernel



Sobel horizonal kernel



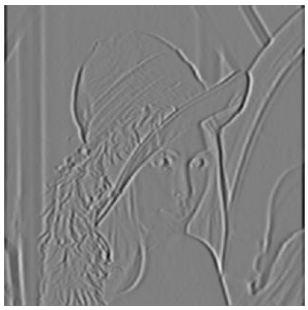
Gradient magnitude

Sobel kernel is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the resulting vertical and horizontal gradient approximations can be combined to give the gradient magnitude.

$haar_1$ scale=5

$haar_2$ scale=5

$haar_3$ scale=5

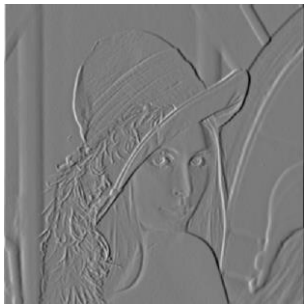


$haar_4$ scale=5

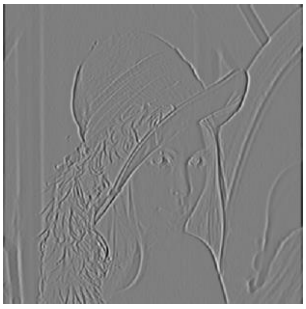
$haar_5$ scale=5

haar$_1$ scale=3

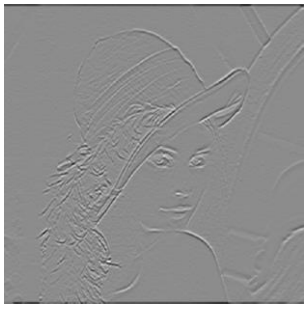
haar$_2$ scale=3



haar$_3$ scale=3



haar$_4$ scale=3



haar$_5$ scale=3

By changing the scale of Haar-like masks, we can notice the difference of ability to find out the features (edge, line) in the image. The smaller masks can indicate smaller/subtle features (like the details in Lena's hair), while the larger ones can spot the wider edges and lines.

Part 2: SIFT Features and Descriptors

Implementation

In this part, I used SIFT functions from both siftDemoV4 and VLFeat to generate the keypoints and descriptors, then show them with the original image. siftDemoV4 provides function **`sift`** and **`showkeys`**.

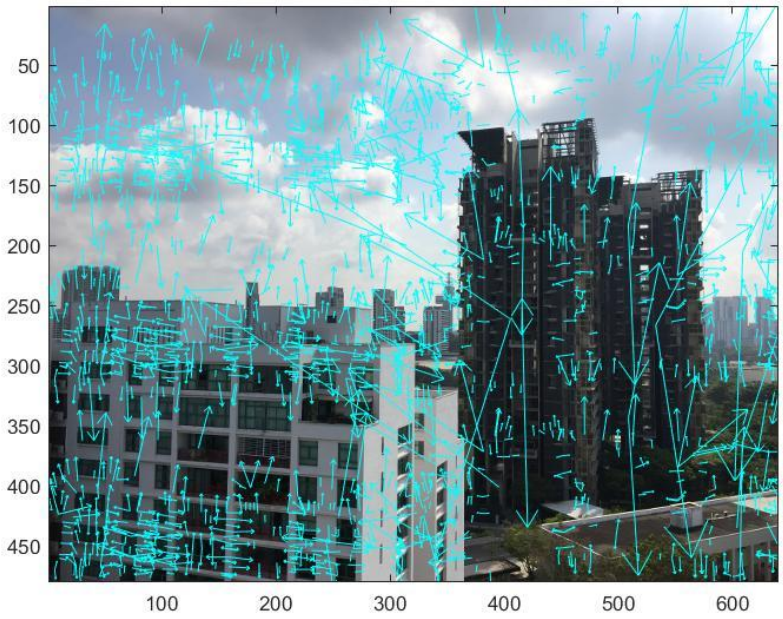
`sift` extracts the keypoint information, including keypoint location (x, y), scale and orientation, as well as the descriptors. Then we show the descriptors as arrows, indicating the location, scale and orientation of the keypoints. Run **`p2_sift_keypoints`** to show the keypoints.

This process is primitive and time consuming. For a better visualization of the descriptors, run **`p2_sift_descriptors`**. It uses VLFeat libraries to show the descriptors as well as the keypoints in an intuitive way.

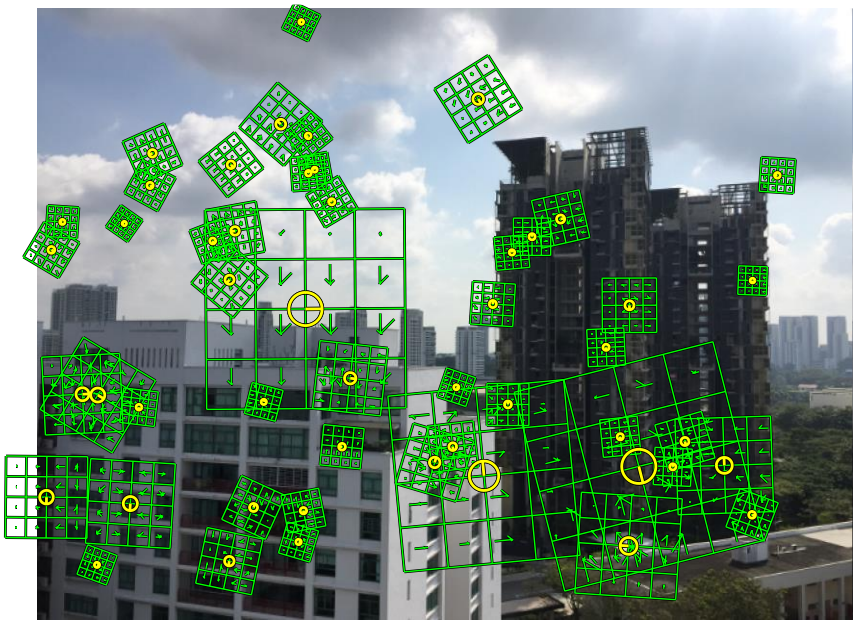
⚠ Please change **`rootpath`** to the project root path so that it can find the images.

Results

The result from siftDemoV4:



The result generated by VLFeat is more intuitive. It can randomly select n keypoints for descriptor visualization. In this case n=50:



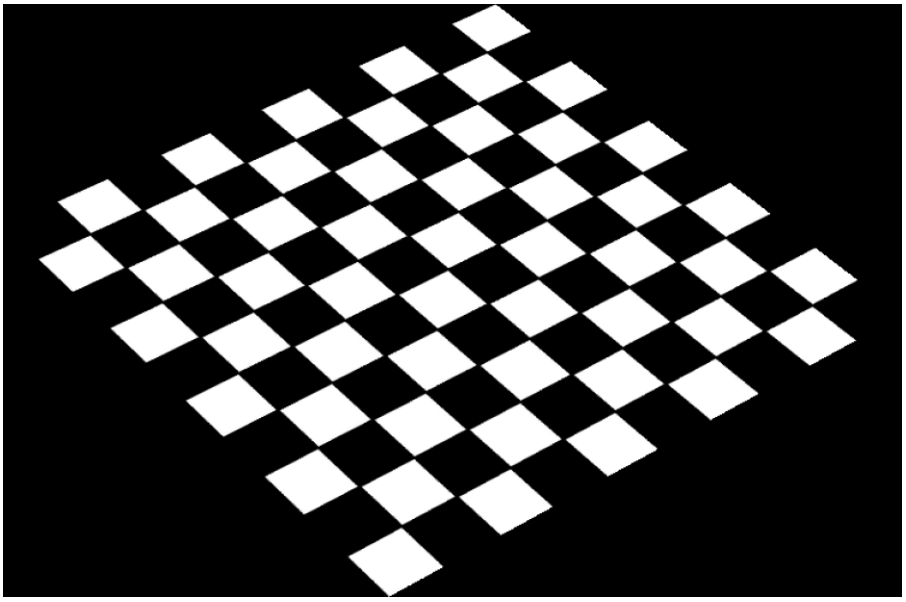
Part 3: Homography

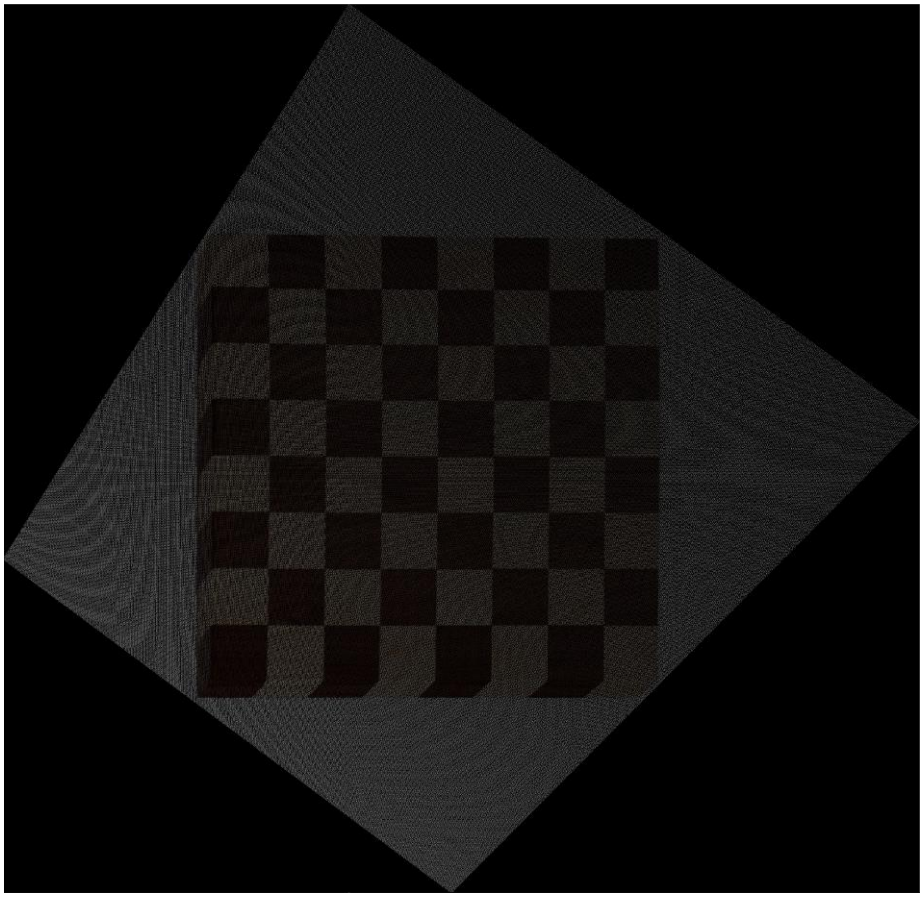
Implementation

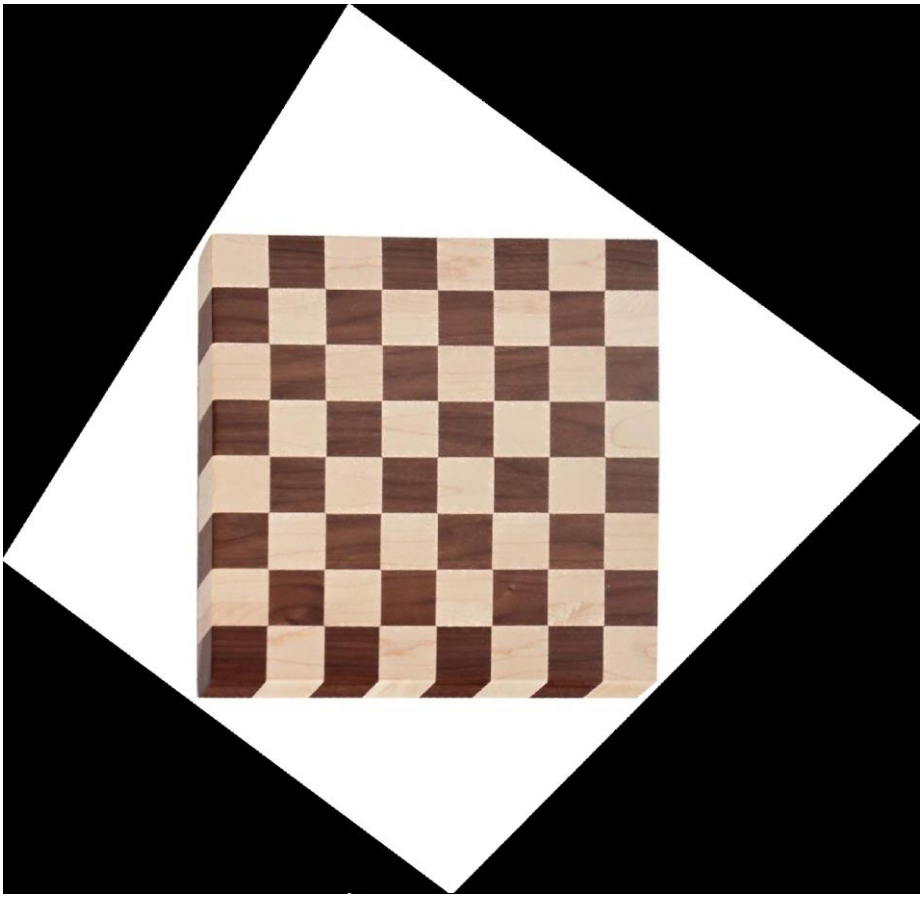
In this part, the script contains 6 main stages shown as follow:

1. Manual selection of the keypoints
   You can select the keypoints with cursor by single-clicking on the image and end with a double-click. For example, if you want to select 4 points, click 3 times and double-click on your 4th point.
   You can select more than 4 pairs of points, but please make sure the order you click in both images are the same.
2. Compute H12 in [x2,y2,1] = H12 * [x1,y1,1]
   First I construct matrix A in Ah = 0, then compute SVD over A to form H12.
3. Transform the image 1 to match image 2
   First I deploy the canvas and make sure the transformed coordinates will not fall outside the canvas by firstly transform the vertices of the image and set the size. Then I run the pixel-wise transformation.
4. H21 in [x1,y1,1] = H21 * [x2,y2,1]
   Here a easy way to get H21 is calculate the inverse matrix of H12.
5. Transform the image 2 to match image 1
   Same as step 3.
6. Avoid image tearing
   There are some ways to prevent small images from being tore up when transforming to large scale. One is interpolation, like I do in in the next part. Here I used a cheap but funny method called 2D order-statistic filtering, which is just copying the maximum from the pixels around the gaps. This works pretty good for simple images like this.

Results







Part 4: Manual Homography + Sticthing

Implementation

The main steps are same from part 3. By manually selecting points in both images, they are stitched and displayed in the same canvas. This need 2 steps: pixel-wise copy of the first image and pixel-wise transformation of the second one.

Another difference from part 3 is, interpolation is used on the original image to avoid tearing up. Also, in this part of the project, we need to automatically update the size of the canvas, by first transform the vertices of the image, then find the lower and upper bound of transformed pixel coordinates.

In some case there will be **'double edges'** effect in the overlapping area. Usually this is caused by the error when matching the two images. We can use some techniques like gain compensation and blending to make the overlapping parts merge with each other.

⚠Please change **`rootpath`** to the project root path so that it can find the images.

Results

Here we can see the transformation stretched the image and teared it up. By interpolating the image by a factor of about 4, we can almost eliminate this effect. But this has some shortcomings. We need to manually select the scaling factor. Also, a large image can significantly slow down the code.

Transform Interpolated Image 2 & Stitch to Image 1

Part 5: Homography + RANSAC

Implementation

There are 9 stages in this part. The time for running is ~60s. It can be written in a more compact way by packaging each part into functions, like I did in part 6. But here I chose to make it detailed, making all the processes and variables visible.

1. Load the images and extract the discriptors
   Function VL_SIFT is used in this script to extract keypoints and descriptors. It is the only high-level function used in this part.

2. Exhaustive search on matched descriptors
   Find the best match between the keypoints in both images. This process has exactly the same functionality and output as the function VL_UBCMATCH.

   Also, I exactly followed the paper by David Lowe, by setting a threshold to make sure only the pairs that satisfy the following condition are accepted:

   *THRESHOLD * CLOSEST_DISTANCE < SECOND_CLOSEST_DISTANCE*

   Only the best descriptor pairs, which are significantly closer to each other than to other descriptors, are accepted. The threshold is set to 1.5 by default, as mentioned in the paper Automatic Panoramic Image Stitching Using Invariant Features [1].

3. Show the keypoint pairs before RANSAC

4. RANSAC
   Some parameters are tunable in this part like iterations, epsilon and the number of selected keypoints.

5. Show the keypoint pairs after RANSAC

6. Stitching - transform img1 & stitch to img2

7. Stitching - transform img2 & stitch to img1

8. Stitching - transform interpolated img1 & stitch to img2

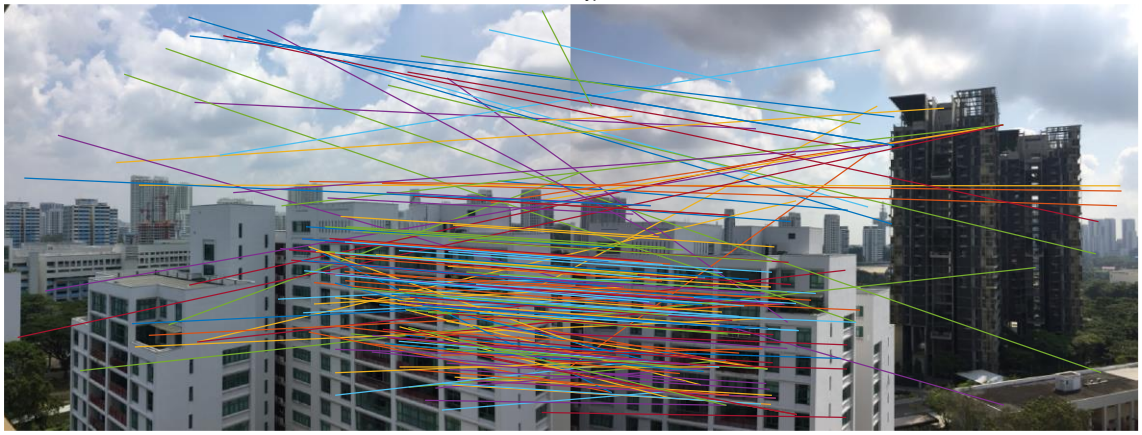
9. Stitching - transform interpolated img2 & stitch to img1

⚠ Please change **`rootpath`** to the project root path so that it can find the images and use the VLFeat libraries.

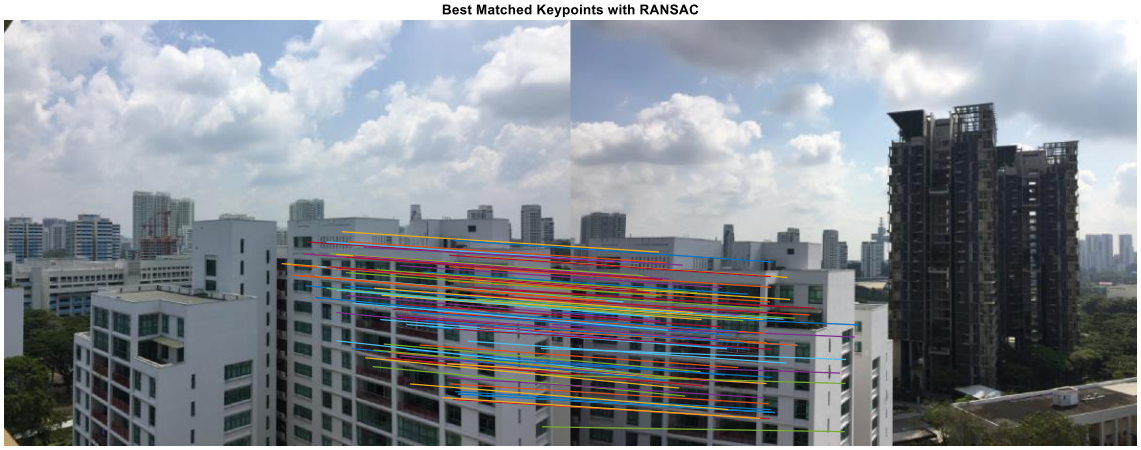
Results

Original Images

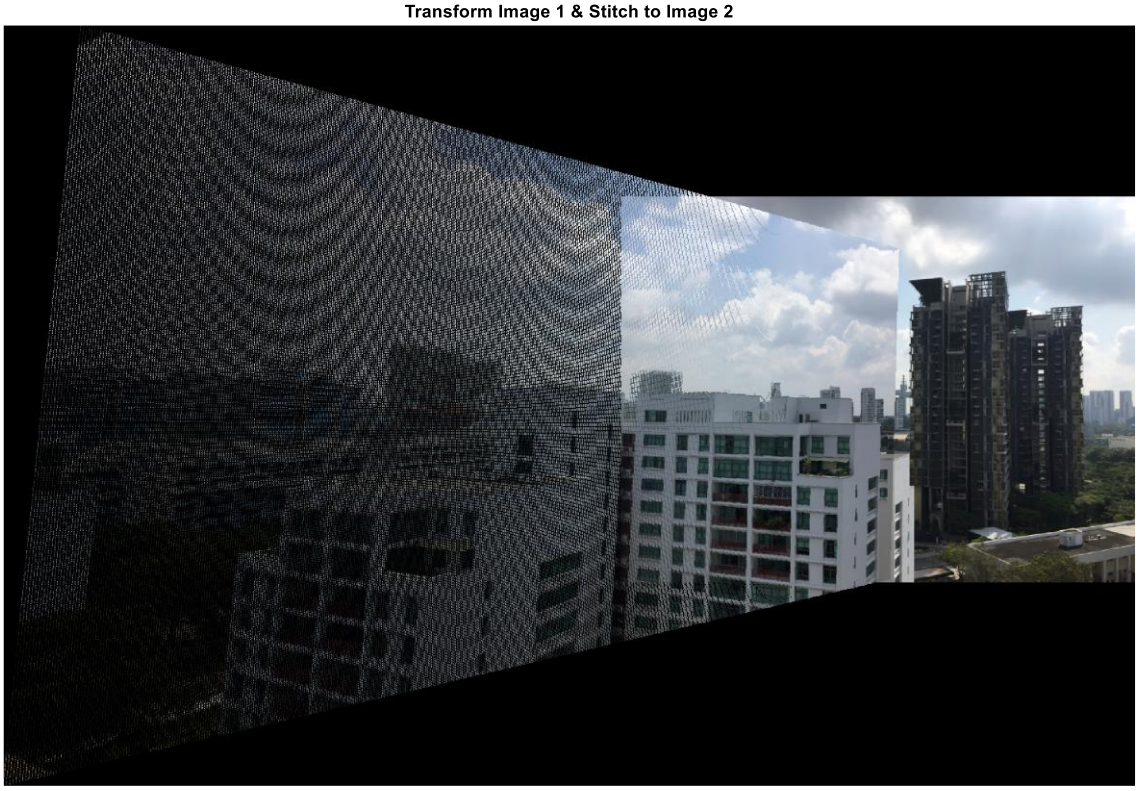
Matched Keypoints



Best Matched Keypoints with RANSAC



Transform Image 1 & Stitch to Image 2



Transform Image 2 & Stitch to Image 1

Transform Interpolated Image 1 & Stitch to Image 2

Transform Interpolated Image 2 & Stitch to Image 1

Part 6: Basic Panoramic Image

Implementation

There are 7 stages in this part. The time for running will be longer if larger interpolation parameters are chosen. The running progress will be updated in the command window.

1. Load the 5 images and extract the descriptors
   Function VL_SIFT is used in this script to extract keypoints and descriptors. It is the only high-level function used in this part.

2. Exhaustive search on matched descriptors
   Find the best match between the keypoints in neighbouring images. This part is placed in **`my_match`** function, which has exactly the same functionality and output as the function VL_UBCMATCH from VLFeat.

   Also, I exactly followed the paper by David Lowe, by setting a threshold to make sure only the pairs that satisfy the following condition are accepted:

   *THRESHOLD * CLOSEST_DISTANCE < SECOND_CLOSEST_DISTANCE*

   Only the best descriptor pairs, which are significantly closer to each other than to other descriptors, are accepted. The threshold is set to 1.5 by default, as mentioned in the paper Automatic Panoramic Image Stitching Using Invariant Features [1].

3. RANSAC
   This part is placed in **`my_ransac`** function. It outputs the best matches and their inliers for the further calculation of homography matrix. Some parameters can be changed in this part like iterations, epsilon and the number of selected keypoints.

4. Apply interpolation to the images
   I chose img3 to be the canvas, which means that img2 and img4 should be interpolated to a larger size, and img1 and img5 should be interpolated to a much larger scale. By setting the parameter vector **`scale`** we can achieve respective scaling on each image. For example, the default **`scale`** is [8 4 1 4 8].

5. Calculate homography matrices
   The script in **`my_homography`** function is used to compute H12, H23, H34 and H45. Then I plan to map img1, 2, 4, 5 directly to the canvas (img3), so H13, H23, H43 and H53 are calculated.

6. Prepare the canvas

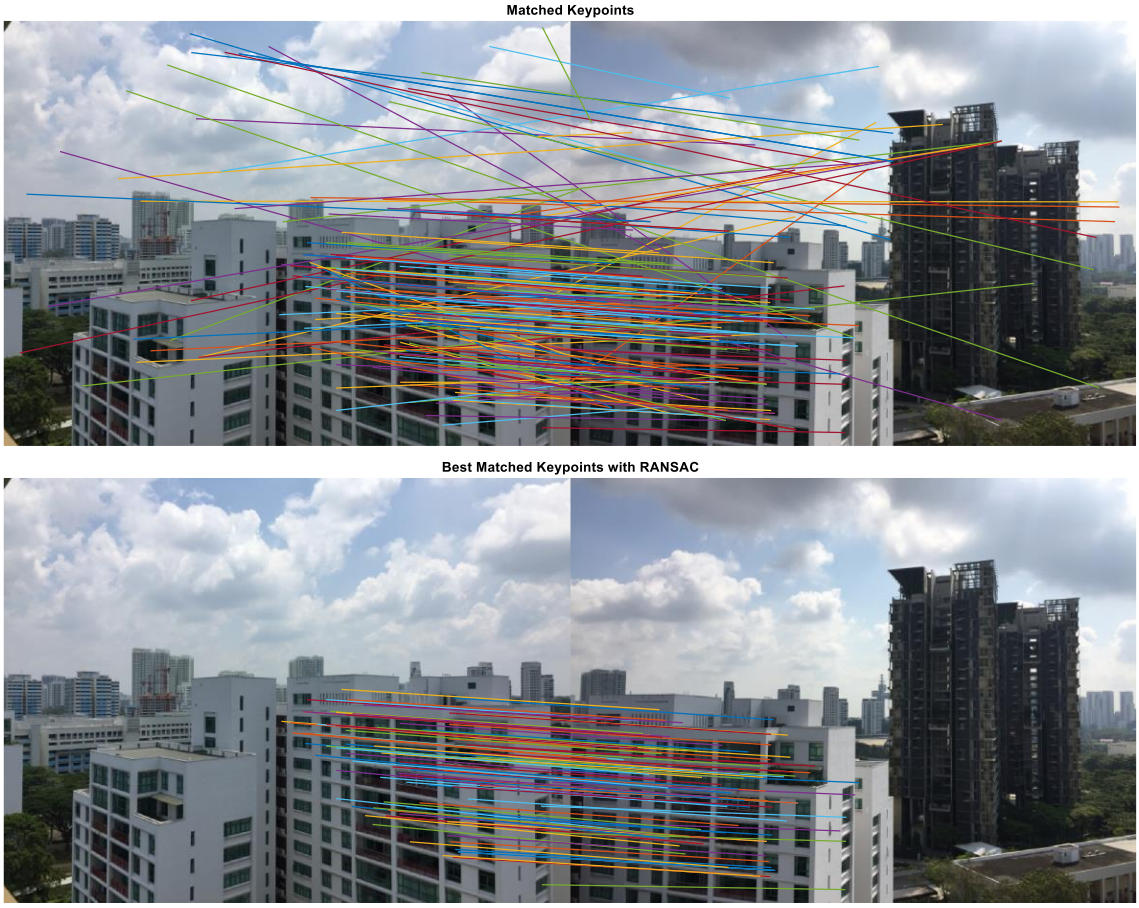
7. Stitching - Transform interpolated img1, 2, 4, 5 & stitch to img3

⚠Please change **`rootpath`** to the project root path so that it can find the images and use the VLFeat libraries.

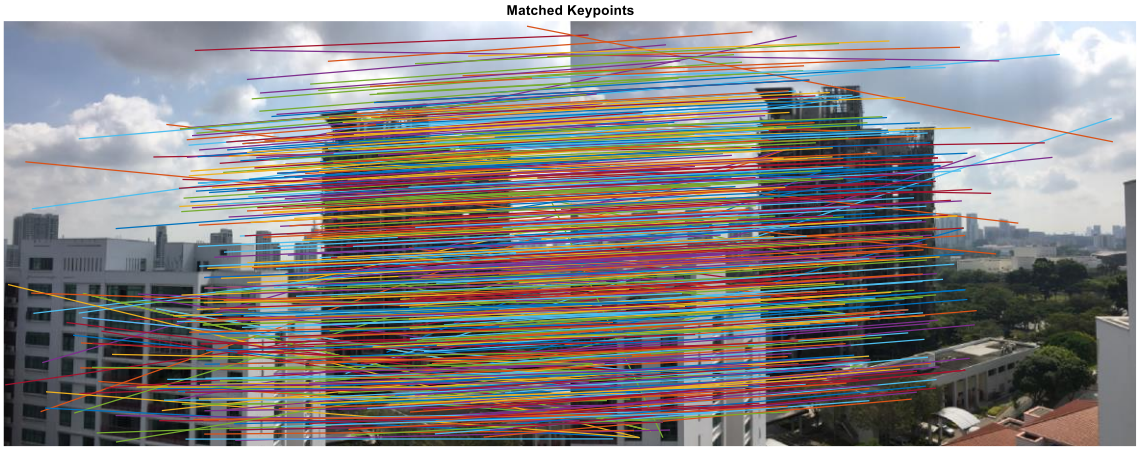
Results

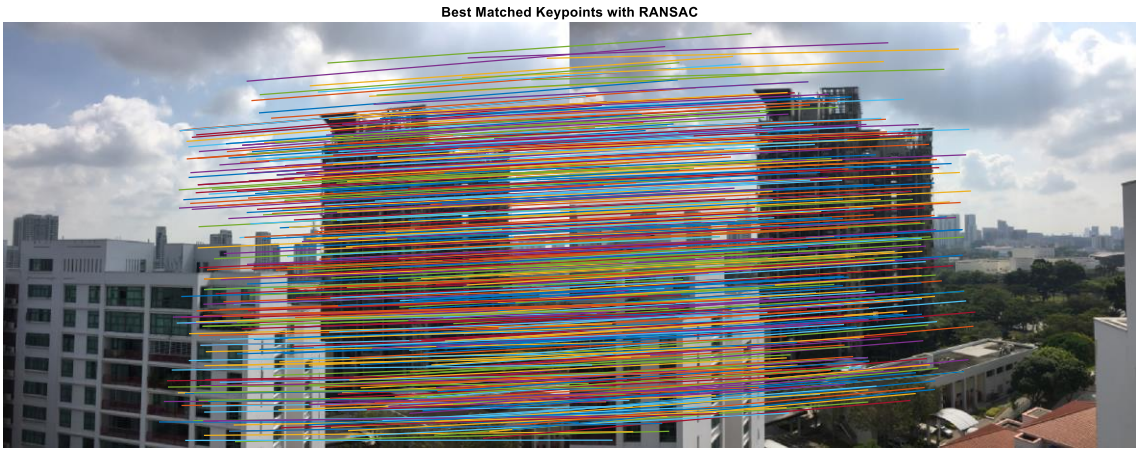
By setting the scaling factor for img1~5 to **`scale`** = [8 4 1 4 8], we have the following result:

The matched keypoints before RANSAC and the best matches after RANSAC between im01 and im02:

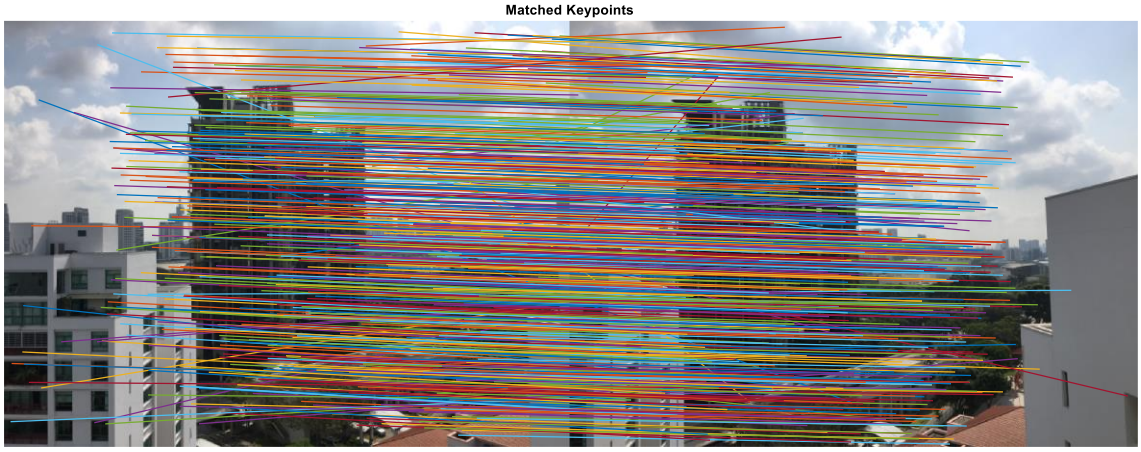


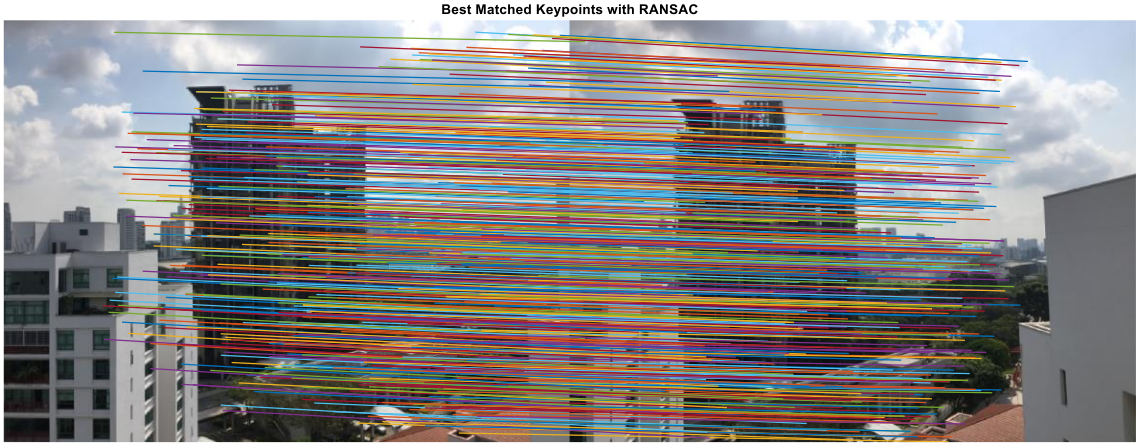
The matched keypoints before RANSAC and the best matches after RANSAC between im02 and im03:





The matched keypoints before RANSAC and the best matches after RANSAC between im03 and im04:



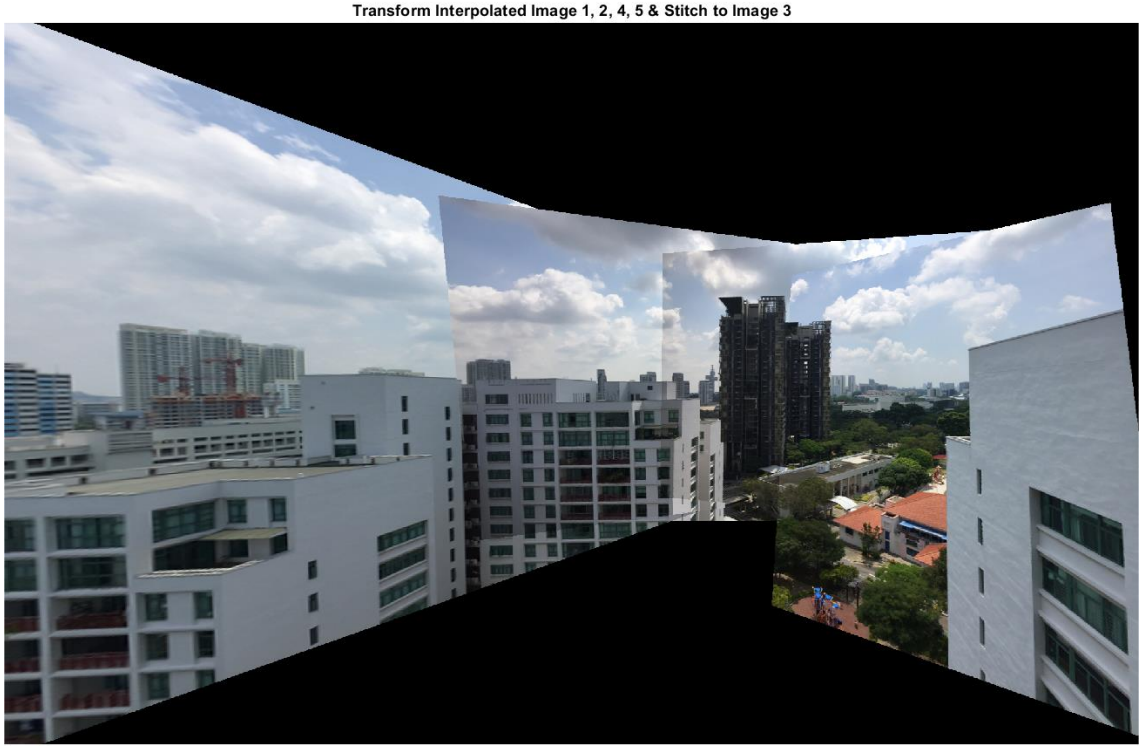


The matching between im4 and im5 are the same. The stitching result is:

By setting the scaling factor for img1~5 to **`scale`** = [16 4 1 4 8], we have the result below. Note that this may take a longer time.

Here is a detailed display of the stitched part. You can also take a look at the result.fig file.

Transform Interpolated Image 1, 2, 4, 5 & Stitch to Image 3



Transform Interpolated Image 1, 2, 4, 5 & Stitch to Image 3

Part 7: Advanced Panoramic Image

1. Handling Unordered Images

In this part, the only input is a set of images. The program can automatically detect the relationship between each pair of images, link the related images and form a graph with undirected edges, weighing the edges with the correlation *R*, which is designed and introduced in David Lowe's paper [1][3]. I then turn the graph into a maximum spanning tree using Minimum Spanning Tree (MST) algorithm. After selecting the root of the tree by finding out the node with the least distance to the leaves of the tree, I wrote a function **`my_crazyfunction`** to recursively calculate the homography matrix, move the canvas, and do the transformation, from the root to each leaves.

Theory

1) Image Matching Verification

There is more than one method to achieve unordered image matching. Most of the new methods are machine learning based [2]. In my project, I used the method proposed in *Automatic Panoramic Image Stitching using Invariant Features* [1] by Matthew Brown and David Lowe in 2007. Actually, this method was first proposed separately in their paper *Recognizing panoramas* [3] in 2003. The only change is some tuning on the parameters. Here is the core idea of the method.

After matching the descriptors and then using RANSAC to filter the best matches, we got the inliers and outliers. The inliers are the features that lies close to their targets after transformation, while the outliers are those who lies inside the overlapped area but are far from their targets. The sum of inliers and outliers are the total features in the overlapped area. We denote the sum of features as n_f, and the number of inliers as n_i. Only when they follow the inequation below can these two images be considered as relevant images:

$$n_i > 8.0 + 0.30n_f$$

The inequation in the original paper [3] is:

$$n_i > 5.9 + 0.22n_f$$

The first one works pretty well in my experiments. The detailed implementation is discussed in next section.

2) Maximum Spanning Tree

After we verified the potential relationship between the unordered images, a graph with undirected edges is established. We need to remove the redundant edges and turn the graph into a tree with only one root. Also, there is only one path to each node. The root is then set as the canvas. If we set the weight of each edge as the ratio:

$$R = n_i \,/\, (8.0 + 0.30n_f)$$

, then the weight can indicate how much they correlate to each other. We only need to keep the edges with the largest weight. In other words, we hope our tree has the largest weights among all the candidates.

To solve the maximum spanning tree problem, we can just multiply -1 to the edges to turn it into a

minimum spanning tree (MST). There are a bunch of cool algorithms on MST.

3) Finding the root

To decide which node should be the root of the tree, we calculate the maximum distance to the leaves, then select the one with shortest maximum distance:

$$root = \underset{i}{\operatorname{argmin}}(max(distance(node(i), node(j)))$$

Finally, we carry out the transformation one-by-one from the root (image in the center) to each node (neighboring images) by calling **`my_crazyfunction`**. This part was not covered in Lowe's paper, but mentioned in some blogs on the internet. I then implemented it with MATLAB.

Implementation

To achieve unordered image stitching using arbitrary image set, I used MATLAB structure to represent images and their attributes like below:

```
% load image
num_img = 4;

for i = 1:num_img
    fprintf(['Loading image ' num2str(i) '/' num2str(num_img) '... '])
    img(i).img                    = imread(['yosemite' num2str(i) '.jpg']);
   [img(i).h, img(i).w, ~]        = size(img(i).img);
   [img(i).f, img(i).d]           = vl_sift(im2single(rgb2gray(img(i).img)));
    img(i).d                      = double(img(i).d);
    fprintf('Done.\n')
end
```

All the variables are stored in different structures. It is convenient especially when we need to use the homography matrices and their inverse (H_{ij} and H_{ji}). We can just call `H(i,j).h` for H_{ij} and `H(j,i).h` for H_{ji}. More details can be found in **`my_crazyfunction`**.

Results

One image set is the one given by the professor. The 5 images are captures from left to right. We human can easily figure out the best relations among them, which is 1-2-3-4-5. Let's see what my script will do to find out the tree.

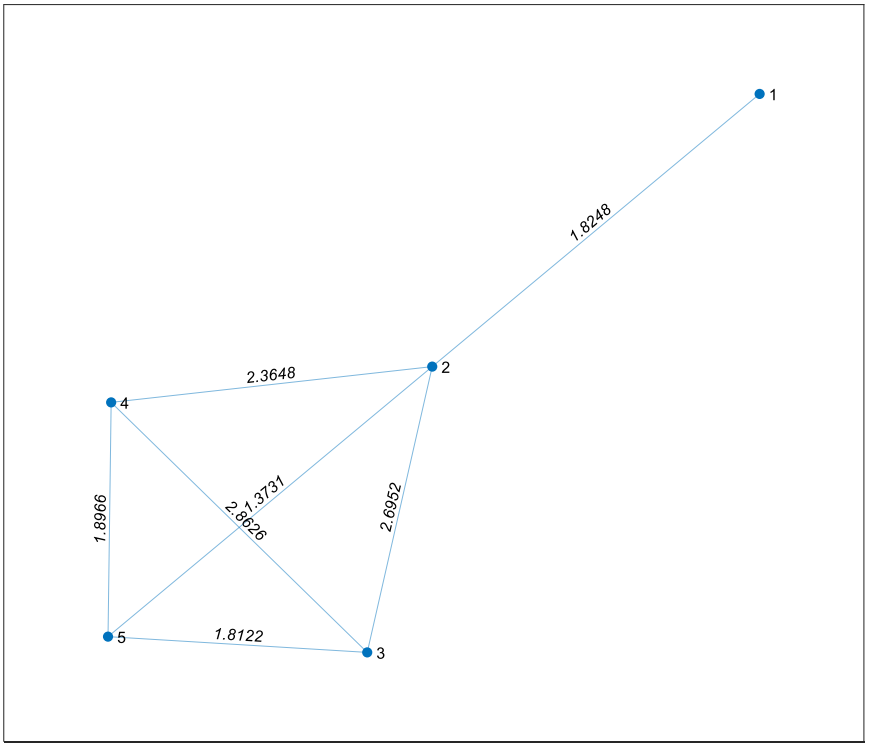




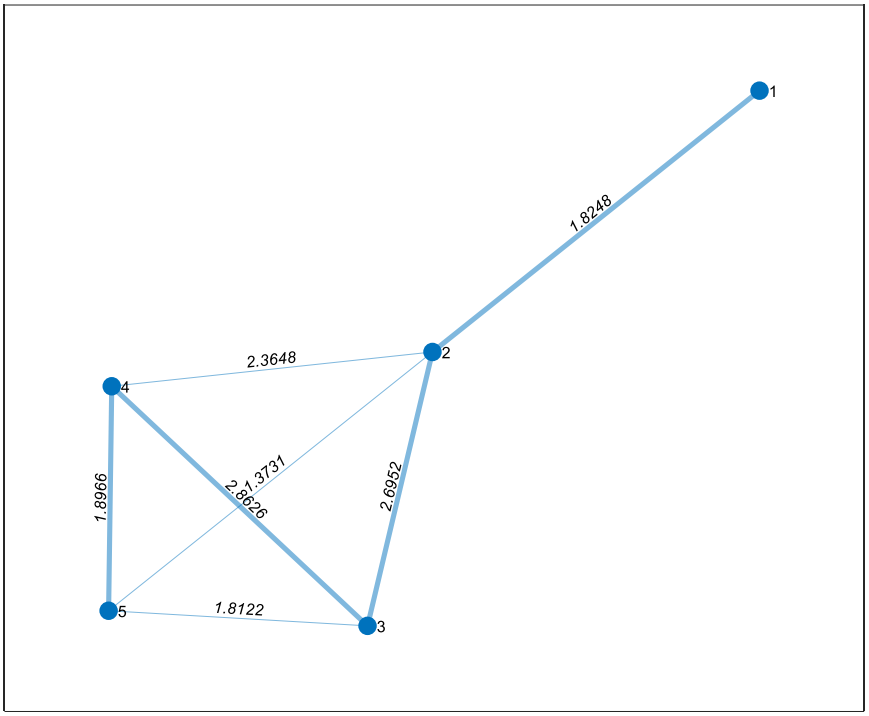
After running the code, we can see the output in the command window:

```
>> p7_unordered_stitching
Loading image 1/5... Done.
Loading image 2/5... Done.
Loading image 3/5... Done.
Loading image 4/5... Done.
Loading image 5/5... Done.
Checking image 1 & 2... w 1.8248 n 100N 156 Matched.
Checking image 1 & 3... w 0.43269 n 18N 112 Denied.
Checking image 1 & 4... w 0.22059 n 6N 64   Denied.
Checking image 1 & 5... w 0.33613 n 4N 13   Denied.
Checking image 2 & 3... w 2.6952 n 459N 541 Matched.
Checking image 2 & 4... w 2.3648 n 258N 337 Matched.
Checking image 2 & 5... w 1.3731 n 46N 85   Matched.
Checking image 3 & 4... w 2.8626 n 527N 587 Matched.
Checking image 3 & 5... w 1.8122 n 83N 126  Matched.
Checking image 4 & 5... w 1.8966 n 88N 128  Matched.
```

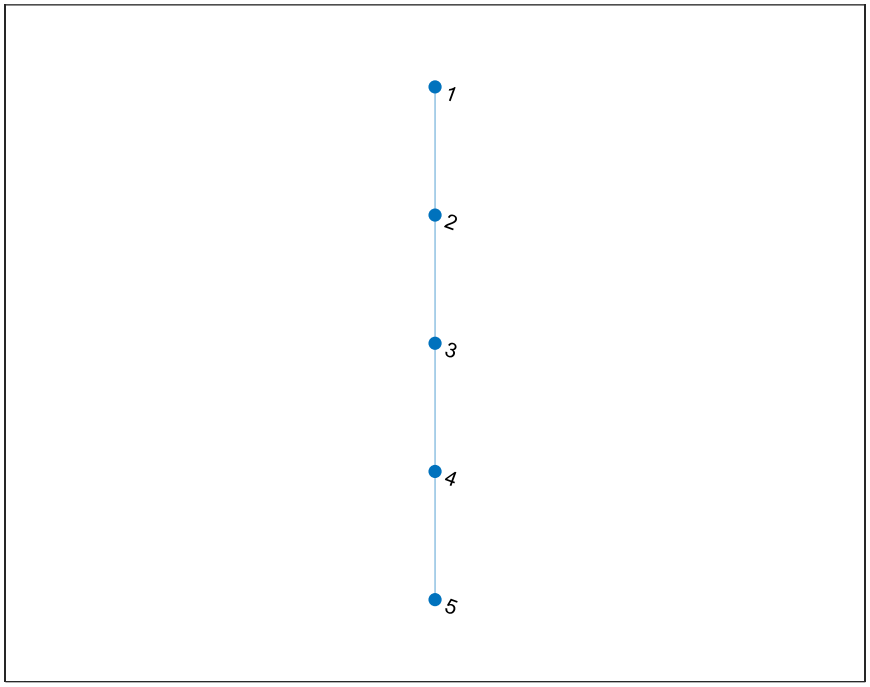
This outputs the weight, inliers, and features in the overlapped area, after 'w', 'n' and 'N' respectively. The original graph G looks like below.



For example, node 5 will choose to link to node 4, since the weight on edge (4,5) is larger than those on (3,5) and (2,5). And 4 will link to 3 because the weight on (3,4) is larger than that on (2,4). We can remove all the redundant branches and find the optimized tree. The generated tree looks like below:



To make it prettier:

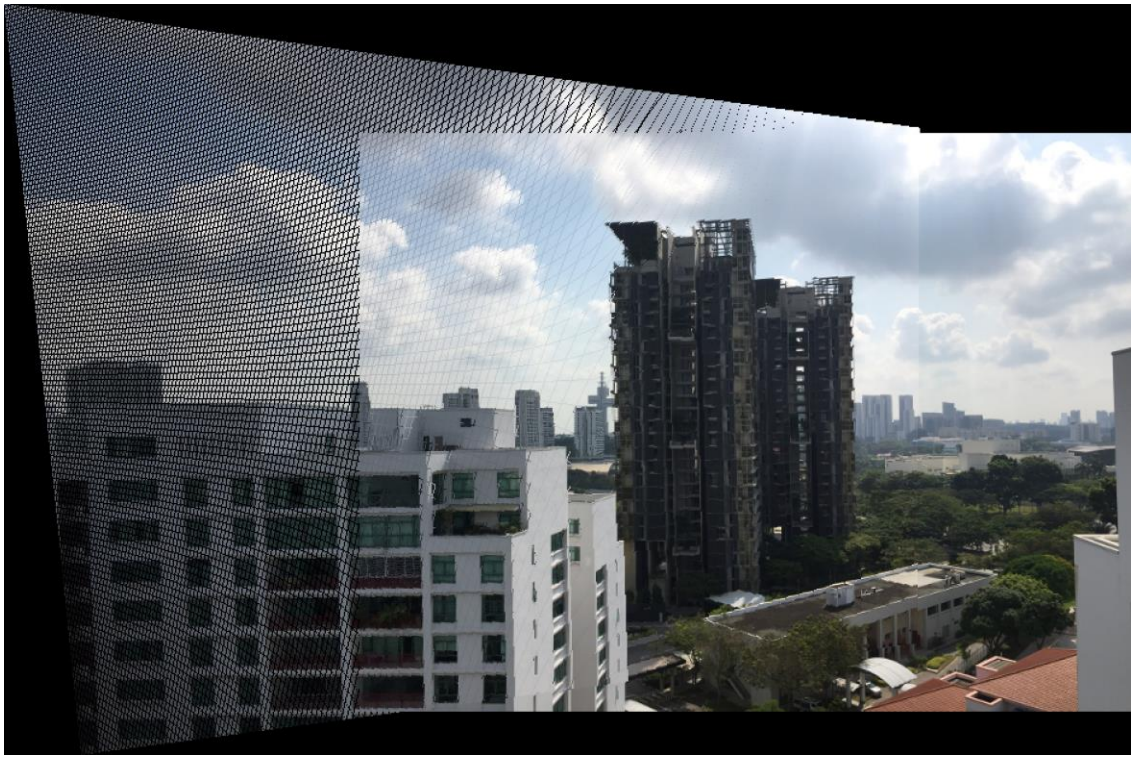


Of course, the root will be 2. This message can be seen in the command window.

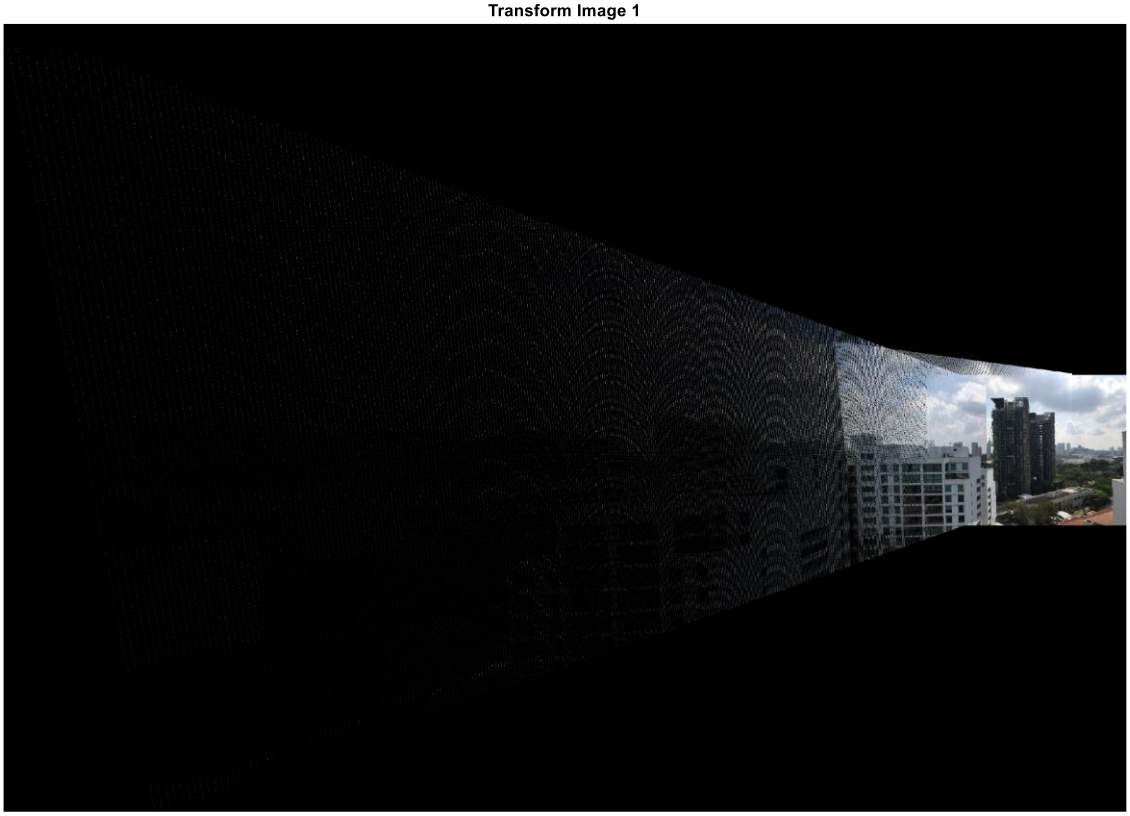
```
The root is node 3.
The farthest distance from root is 2.
```

Then H13 H23 H43 and H53 are computed and used to run the transform one-by-one:

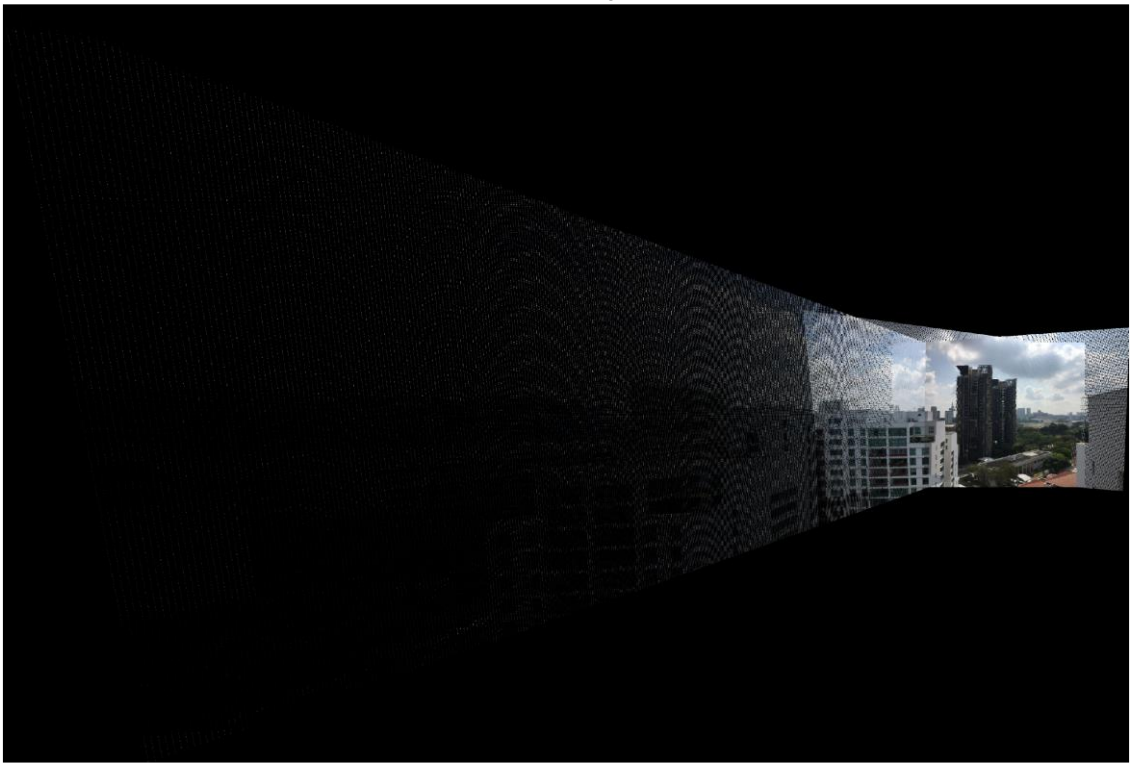
Transform Image 2



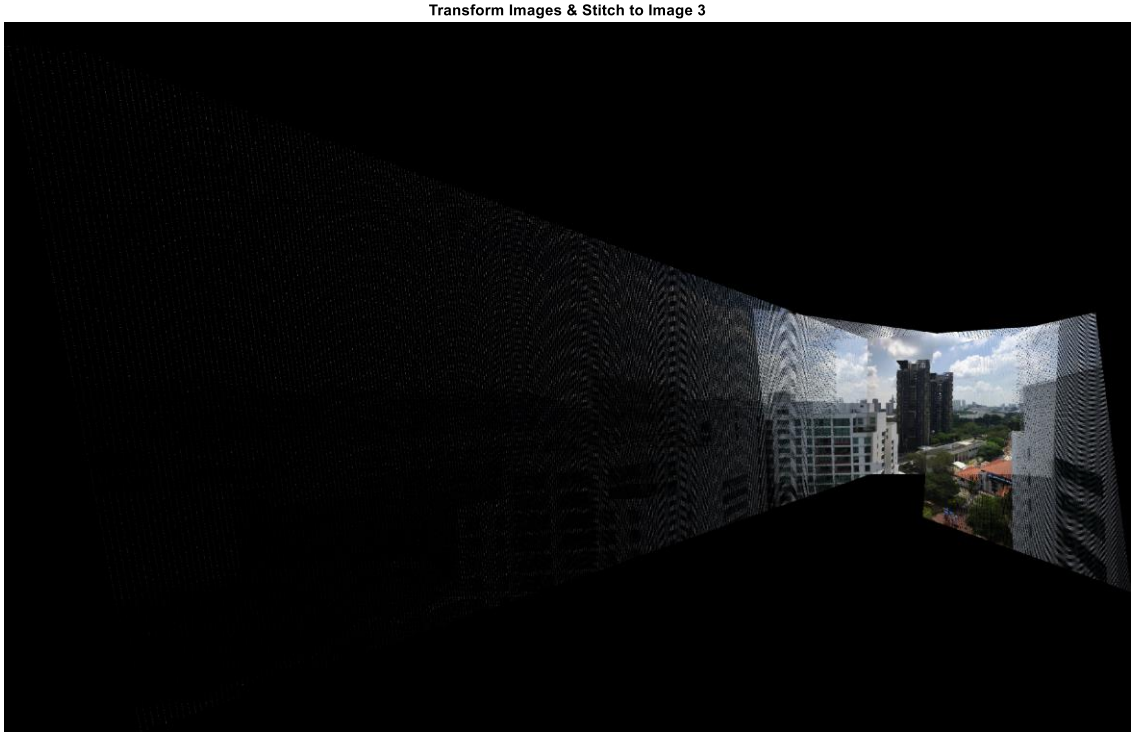
Transform Image 1



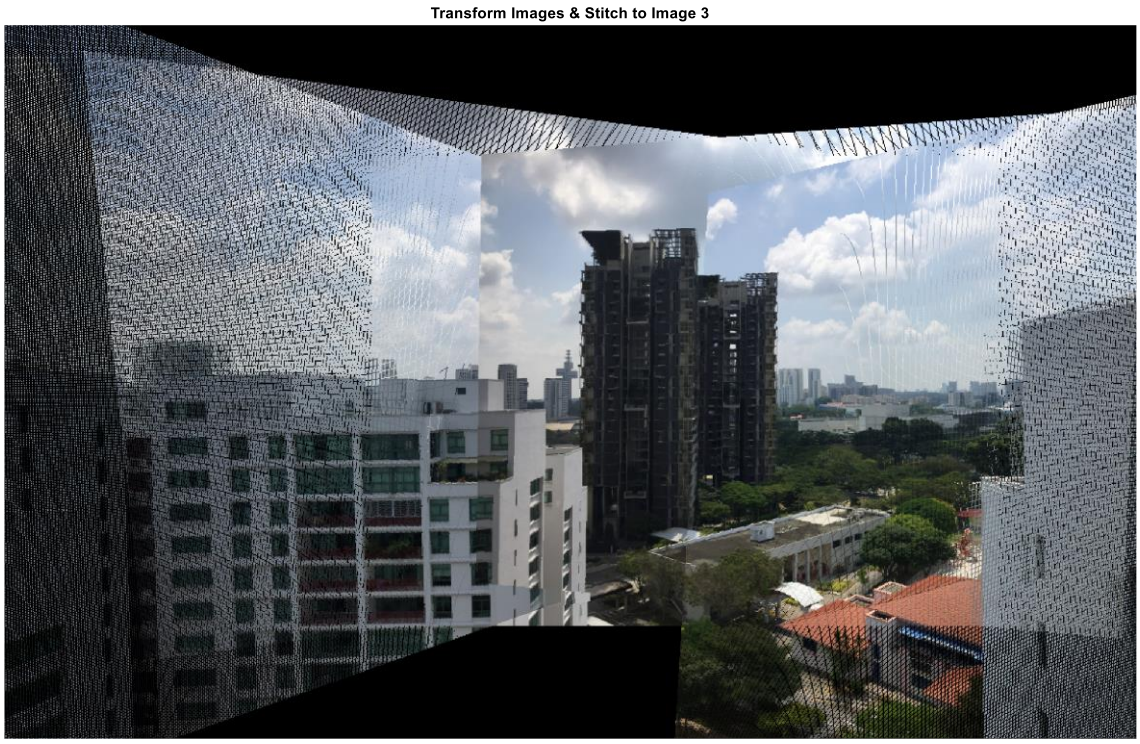
Transform Image 4



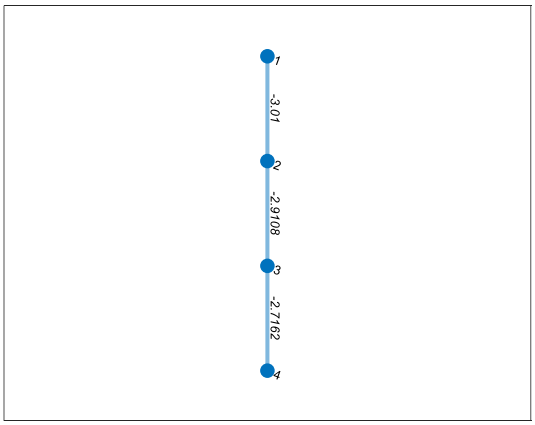
Transform Images & Stitch to Image 3



Interpolation is not used in this demo, so it doesn’t look so good as the one in part 6. Take a close look at the stitched part:

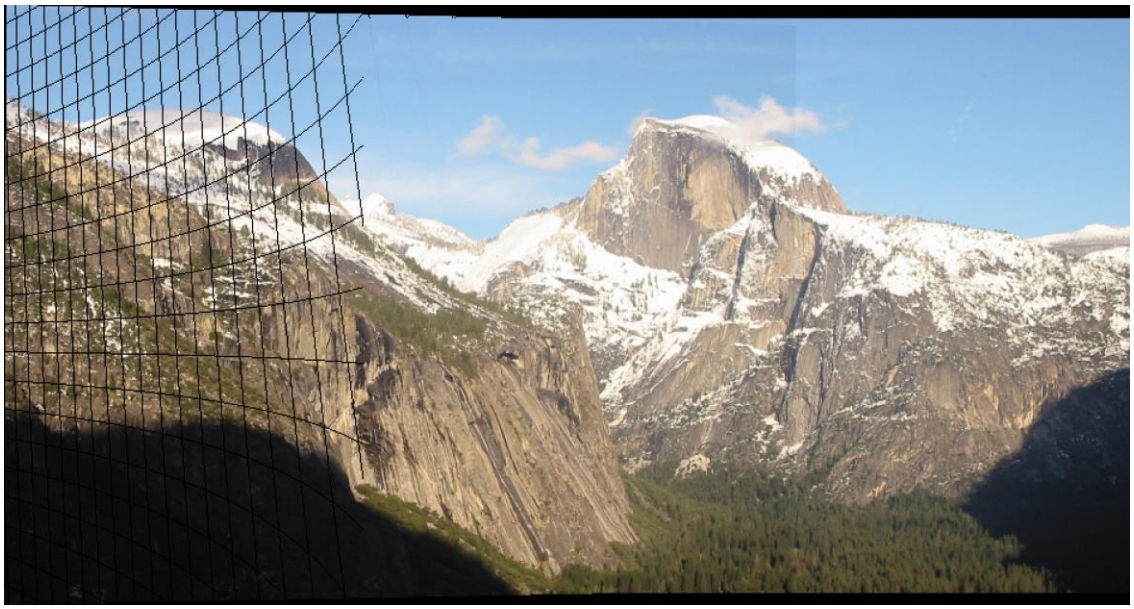


Another example on Yosemite image set:

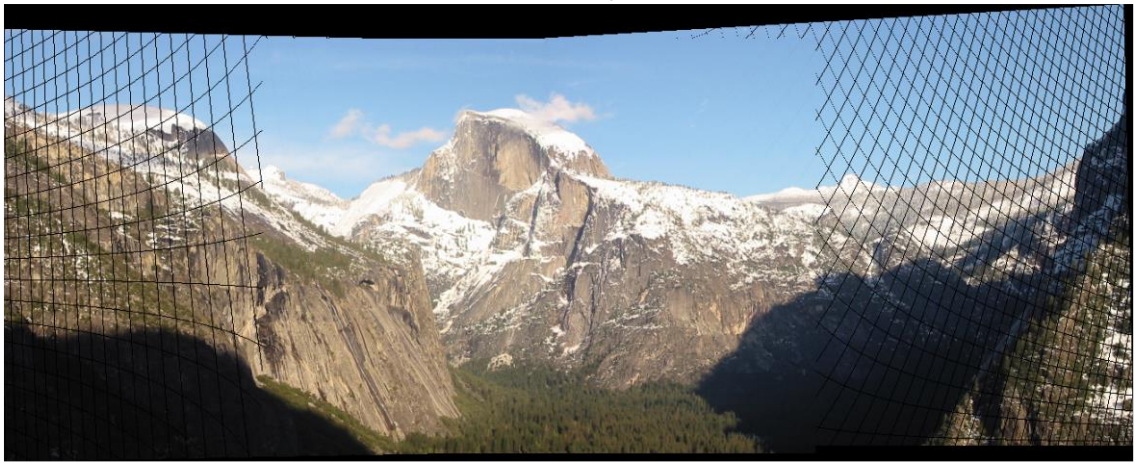


The root is found to be 2. Let's transform image 1, 3 and 4:

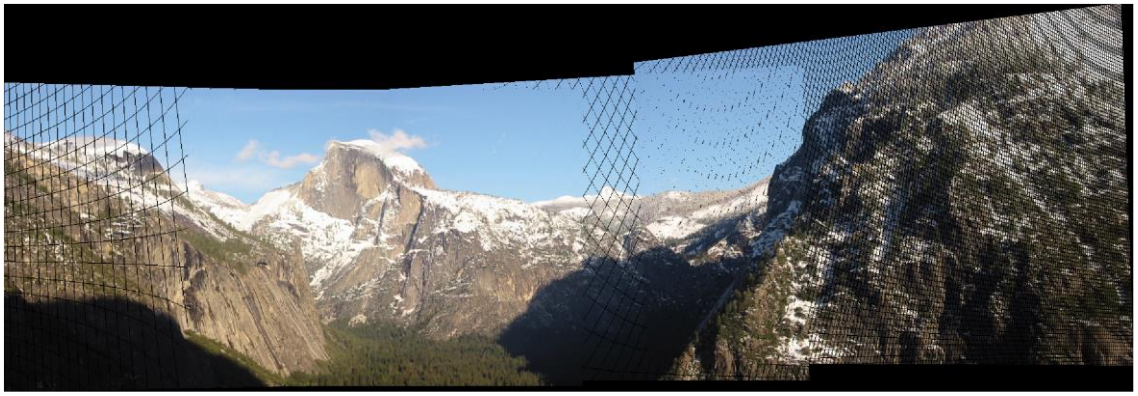
Transform Image 1



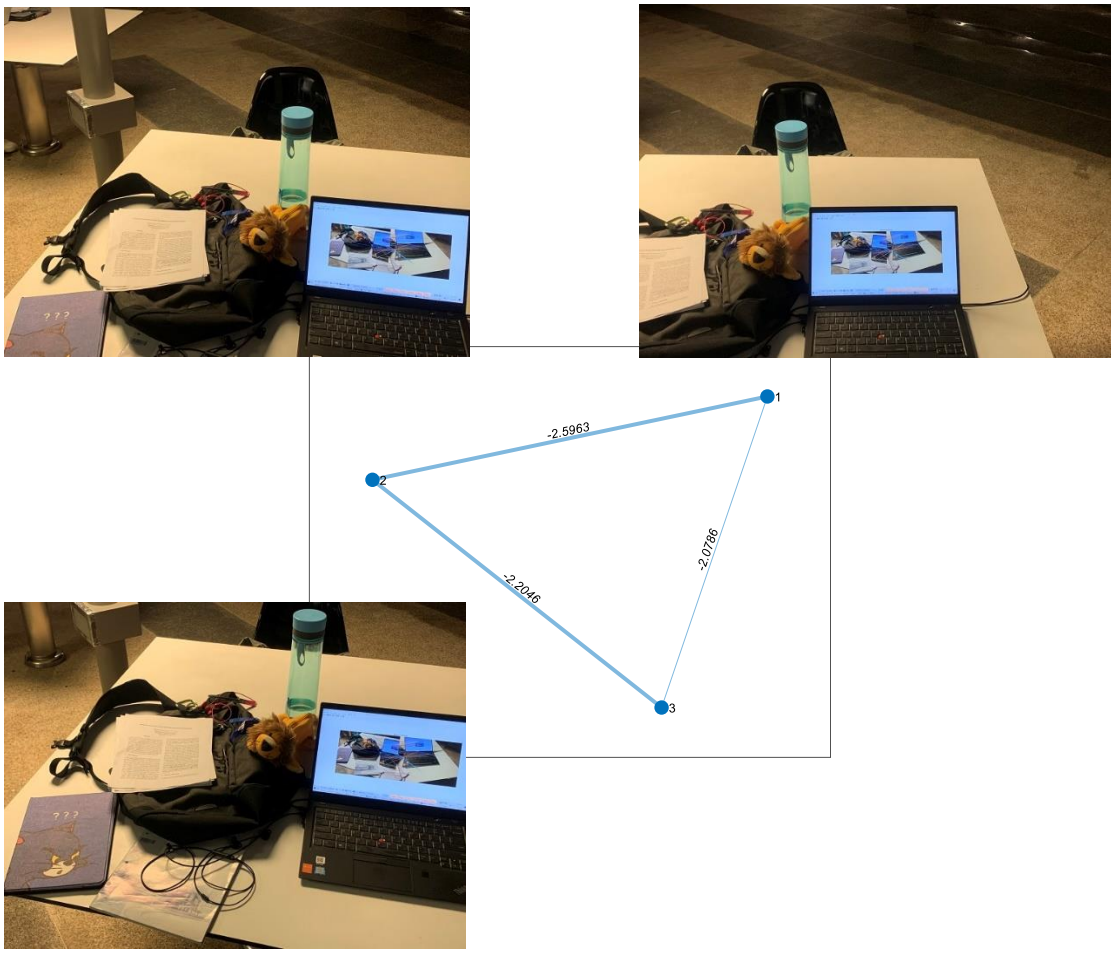
Transform Image 3

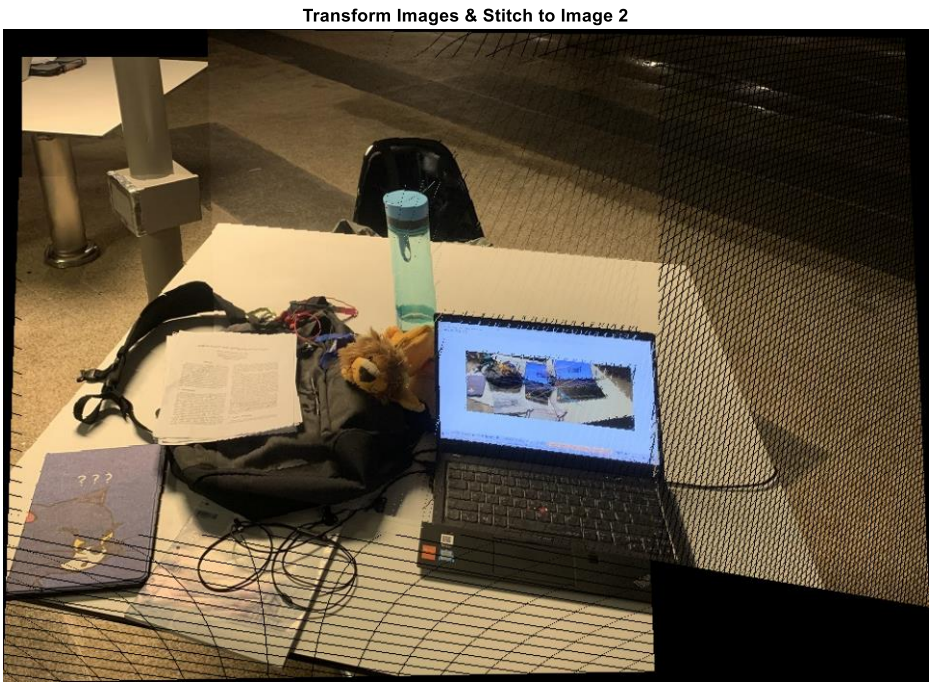


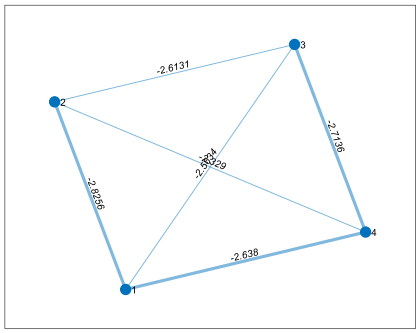
Transform Image 4

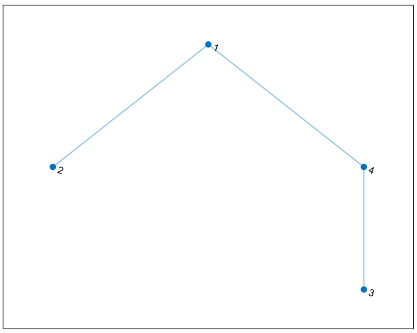


Here are another two examples using the beautiful scenes in UTown.



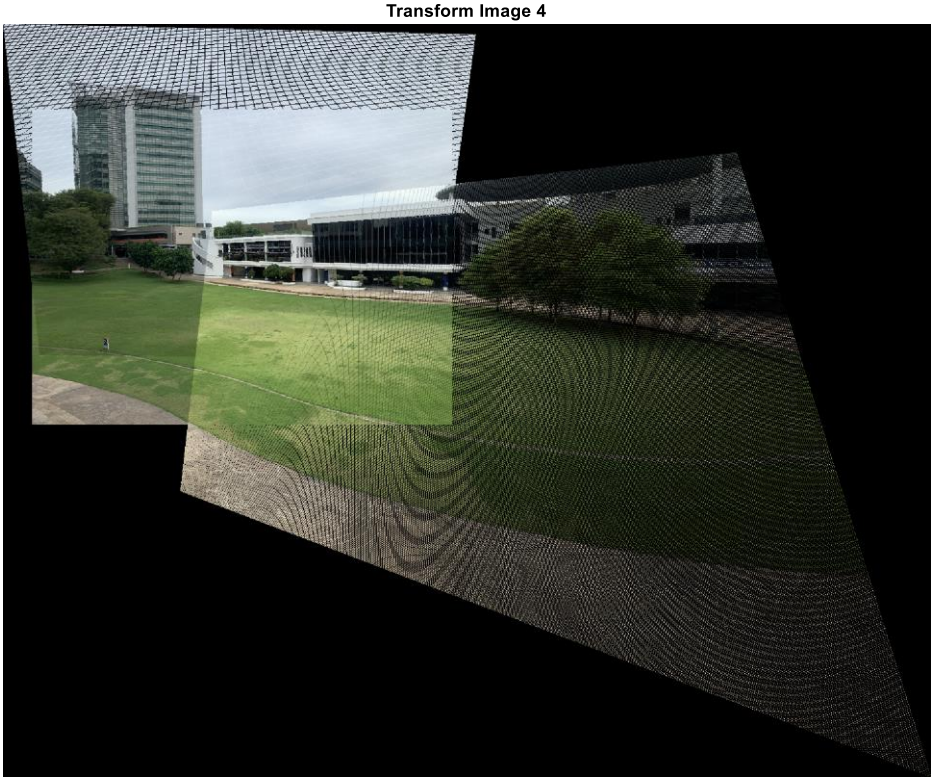




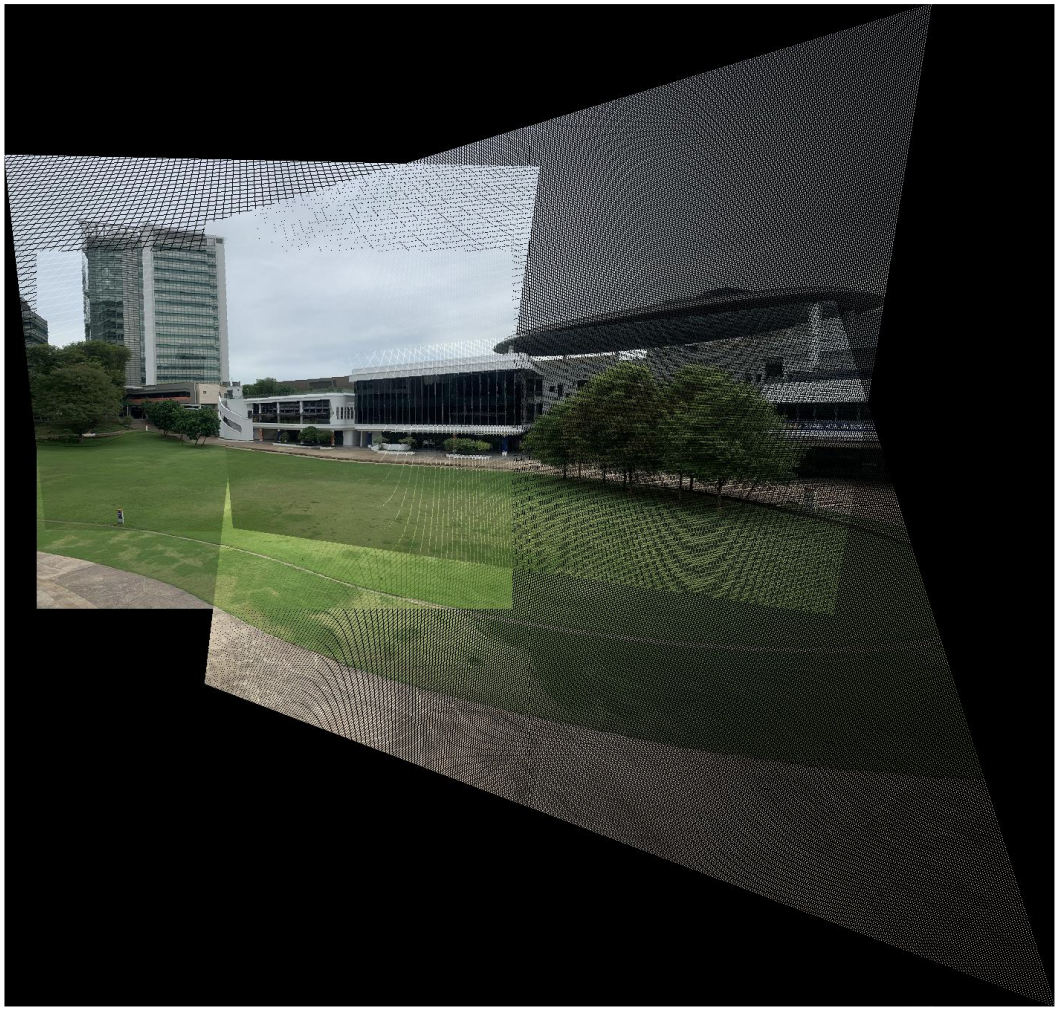


Transform Image 2

Transform Image 4



Transform Image 3



[1] Brown, M., Lowe, D.G. Automatic Panoramic Image Stitching using Invariant Features. *Int J Comput Vision* 74, 59–73, 2007.

[2] Z. Qu, J. Li, K. Bao and Z. Si, "An Unordered Image Stitching Method Based on Binary Tree and Estimated Overlapping Area," in *IEEE Transactions on Image Processing*, vol. 29, pp. 6734-6744, 2020.

[3] M. Brown and D. Lowe, “Recognizing panoramas,” in Ninth International Conference on Computer Vision (ICCV’03), (Nice, France), pp. 1218–1225, October 2003.