Harri Corner Detection & Image Stitcher

Team members:
Jufeng Yang ID: 20125011
Xingda Zhou ID: 19107471
Zhongen Qin ID: 19107579

Table of Content

1. Introduction

The aim of this project is to stitch together two images with a small number of identical elements to make a complete and unblemished image. Generally this identical element is positioned at the edge of the image, and the code is used to process the image and match the identical elements, get the position of this element separately and move one of the images to complete the stitching and finally form a single image.

The first step is to use a Gaussian filter to process the image, leaving the more obvious points (greater than a threshold) and marking them as interest points. All points are then regularised. The most important part of the data processing is the dot product of the array matrix of interest points. The dot product operation of the matrix strengthens the description vectors with the same values, by obtaining the corresponding rows and columns and marking the positions of the same description vectors to form matching pairs.

The third step uses the difference in coordinates of each pair to find the most appropriate offset value, seeking an average value to determine the photo offset. The final stitching is done by moving the photos according to the offset. When the photographs are of the same size and orientation, then the average match will give very good results. The same size means that the difference between the coordinates of each pair of points of interest is similar. This gives excellent results when the stitching is done.

2. Interesting Points Detection & Matching

2.1 Import Relative Libraries

Import imutils, which contains image reading functions, image display functions, channel reading functions, etc. numpy is used for array data. scipy.ndimage is used for gaussian_filter. Import PIL to show pictures.

```
import numpy as np
from scipy.ndimage import filters
from PIL import Image
import imutils
import matplotlib.pylab as plt
```

Figure.1: Useful libraries.

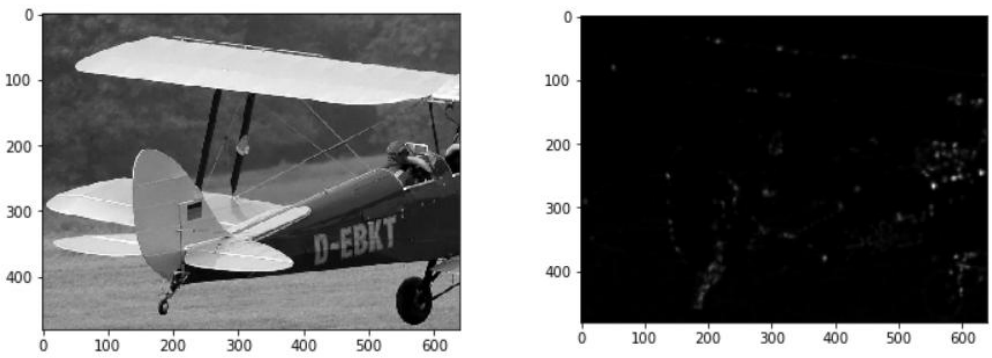
2.2 Harris Corner Detection

The original image is processed using a Gaussian filter to construct a filter matrix so that the corners of the image have relatively large pixel values. Output a matrix of pixel-by-pixel grey-scale image. Code implementation as shown in Figure.2. Algorithmic principles design from mathematical design in project guidance.

```
# Get gaussian filters.
def Harris_Matrix(image, sigma = 2): #sigma is always 2
    Lx = np.zeros(image.shape)
    filters.gaussian_filter(image, 1, (0,1), output = Lx)
    Ly = np.zeros(image.shape)
    filters.gaussian_filter(image, 1, (1,0), output = Ly)

    # Compute the components of the Harris matrix - used to find Trace and Determinant below
    A = filters.gaussian_filter(Lx * Lx, sigma)
    B = filters.gaussian_filter(Lx * Ly, sigma)
    C = filters.gaussian_filter(Ly * Ly, sigma)

    # Find the Trace and Detminant - used to calculate R
    determinantM = (A * C) - (B ** 2)
    traceM = (A + C)

    return determinantM / traceM

```

Figure.2: Construct a method to generate a Harri corner image.

Comparison of the original image with the output after Harri Corner Detection. This is shown in Figure 3.



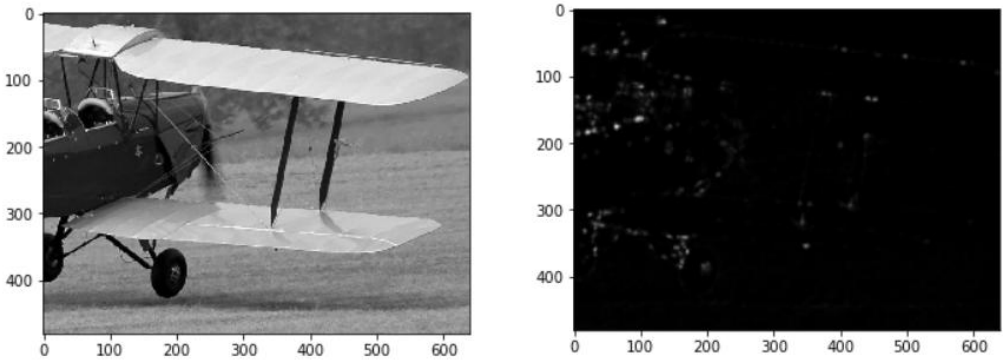


Figure.3: Original picture (Left), After Harris Matrix processing (Right).

2.3 Interesting Points Detection

After acquiring the image after the Harris Corner Matrix, the part with the values is not very discrete. Some of the pixel values are so small that it is not worthwhile to divide them up as a point of interest. It is therefore necessary to check the size and extent of the points. The larger values are then used as a meaningful point of interest.

The code implementation is shown in Figure 4. This part of the code comes from the hints in the experimental guide and will not be explained much here.

```
# Find Harris corner points above a threshold and perform nonmax suppression in the region +/- minimumDistance.
def Get_Harris_Points(Harris_image):
    # Define a minimum distance between the points
    Minimum_distance = 10
    # Define a threshold to filte the points below 0.9 max value
    threshold = 0.1

    # To get top points above a threshold(0.9) times max value
    cornerThreshold = Harris_image.max() * threshold
    Image_thresholded = (Harris_image > cornerThreshold)

    # Convert list value into array and delete all nonzero value
    # Return the location(position) of the Harri points in orihinal image
    coordinates = np.array(Image_thresholded.nonzero()).T
    candidateValues = np.array([Harris_image[c[0],c[1]] for c in coordinates])

    # Return the indeces and sort array as increasing order
    indices = np.argsort(candidateValues)

    # Store the allowed point locations in a Boolean Image and guarantee the the minmum distance of a points and reduce the nonmax
    # value effect
    Pure_locations = np.zeros(Harris_image.shape, dtype = 'bool')
    Pure_locations[Minimum_distance:-Minimum_distance, Minimum_distance:-Minimum_distance] = True

    # Select the best points using nonmax suppression based on the allowedLocations array
    filteredCoordinates = []
    for i in indices[::-1]:
        r, c = coordinates[i]
        if Pure_locations[r, c]:
            filteredCoordinates.append((r, c))
            Pure_locations[r-Minimum_distance:r+Minimum_distance, c-Minimum_distance:c+Minimum_distance] = False

    return filteredCoordinates

```

Figure.5: The implementation of interesting points.

Construct a way to display all points of interest on the original map. Check that the function of the method is correctly implemented. The code implementation is shown in Figure 6. Use

for loops to display points of interest one by one

```
# Plot all red Harris interesting points
def Plot_Harris_Interest_Points(image, interestPoints):
    plt.figure('Harris points/corners')
    plt.imshow(image, cmap='gray')
    plt.plot([p[1] for p in interestPoints], [p[0] for p in interestPoints], 'ro')
    plt.axis('off')
    plt.show()
```

Figure.6: Method used to display all interesting points.

The code blocks shown in 2.1 and 2.2 are in the form of methods. The code to call the method to implement the image processing is then shown in Figure 7. First convert the image to an array and call the Harris method to output a greyscale image. The processed image is then fed into the method that detects the interest points. Finally, the method that displays the points of interest is called. Display the interest point information.

```
# Reading Images from memory.
harrisImage1 = (np.array(Image.open('image-pairs/tigermoth1.png').convert('L'), dtype=np.float32))
harrisImage2 = (np.array(Image.open('image-pairs/tigermoth2.png').convert('L'), dtype=np.float32))

# Shown the images in memeory.
print("Images shown:")
imutils.imshow(harrisImage1)
imutils.imshow(harrisImage2)

# Use the threshold to generate filter and store the bright points
print("Harris interesting points:")
image1 = Harris_Matrix(harrisImage1, 2)
image2 = Harris_Matrix(harrisImage2, 2)

# Show the gray interesting points
print("Shown printing Harris interesting points:")
imutils.imshow(image1)
imutils.imshow(image2)

# Make the bright points into red
print("Getting Interest Points for both images")
interestPoints1 = Get_Harris_Points(image1)
interestPoints2 = Get_Harris_Points(image2)

# Use the interesting points length to get the points number
print("Found " + str(len(interestPoints1)) + " interest points in image 1.")
print("Found " + str(len(interestPoints2)) + " interest points in image 2.")

# Show the red interesting points in images
Plot_Harris_Interest_Points(harrisImage1, interestPoints1)
Plot_Harris_Interest_Points(harrisImage2, interestPoints2)
```

Figure.7: Call method to process pictures and display all interesting points.

2.4 Description Vector Generation and Match Vectors

The coordinate information of the previously generated points of interest is recalled and the part of the image that goes through the Harris matrix that corresponds to the coordinates is turned into a one-dimensional vector using a flattening operation and stored in an array variable. One row of data represents one point of interest. As shown in Figure.8.

```python
# Convert all points into vectors.
def Vectors_Descriptors(image, interestPoints):
    # Define a width to frame a small region
    width = 5
    # Define a descriptor to store all interesting points
    descriptors = []

    # Use the width return a small region and store it in list
    # Normalize all vector points
    for point in interestPoints:
        vector = image[point[0] - width:point[0] + width + 1,point[1] - width:point[1] + width + 1].flatten()
        vector -= np.mean(vector)
        vector /= np.linalg.norm(vector)
        descriptors.append(vector)

    return descriptors
```

Figure.8: Generate a description vector.

This is one of the more important processes in the whole project, using matrix dot products to identify pairs of points of interest. Input description vector, convert the description vector to a 32-bit floating point array. Use the matrix of two description vectors to perform a dot product operation, where the dot product of two vectors that are very similar is relatively large. In general, the dot product of two vectors that are not identical is smaller than 90% of the dot product of the identical vectors. Therefore, after filtering by 0.9 times the maximum value, only the mostly identical vector pairs will remain. This operation achieves the function of filtering out valid vector pairs. Stores information about the ranks of all the same vectors. The code implementation is shown in Figure 9,10.

```python
# Match all points in both pictures.
def Match_Descriptors(descriptors1, descriptors2):
    # Set a 0.95 threshold to filte most of useless points
    threshold = 0.95

    # Define 2 array to store 2 insteresting regions
    descriptors1 = np.array(descriptors1).astype('float32')
    descriptors2 = np.array(descriptors2).astype('float32')

    # Calculate the dot of two descript array.
    # Find the maximum values of array1, array2 and the dot product
    Response_matrix = np.dot(descriptors1, descriptors2.T)
    max1 = descriptors1.max()
    max2 = descriptors2.max()
    Max_elem_martix = Response_matrix.max()

    # Initial, non-thresholded dot product - compared with the thresholded version below
    originalMatrix = Image.fromarray(Response_matrix * 255)

```

Figure.9: Upper half part of description vector method.

```
    # Initial, non-thresholded dot product - compared with the thresholded version below
    originalMatrix = Image.fromarray(Response_matrix * 255)

    # Setting a pair array and stores all matrix position
    Points_pairs = []
    for r in range(Response_matrix.shape[0]):
        First_colum_value = Response_matrix[r, 0]
        for c in range(Response_matrix.shape[1]):
            if (Response_matrix[r, c] > threshold) and (Response_matrix [r, c] > First_colum_value):
                Points_pairs.append((r,c))
            else:
                Response_matrix[r, c] = 0

    # Compare the above matrix with the new, thresholded matrix
    Thresholded_matrix = Image.fromarray(Response_matrix * 255)

    # In order: Maximum of array1, maximum of array2, maximum of Dot Product,
    # Image before thresholding, Image after thresholding and Pairs list
    return max1, max2, Max_elem_martix, originalMatrix, Thresholded_matrix, Points_pairs
```

Figure.9: Down half part of description vector method.

Construct a method to draw matching pairs "Plot_Matches". After generating a description vector match pair, input the match pair information into the method, use a for loop, read out the coordinate information from the match pair variable, draw the point of interest and connect the two point of interest pairs using a green line.

```
# Plot all points and lines
def Plot_Matches(image1, image2, interestPoints1, interestPoints2, pairs):
    rows1 = image1.shape[0]
    rows2 = image2.shape[0]

    if rows1 < rows2:
        image1 = np.concatenate((image1, np.zeros((rows2 - rows1, image1.shape[1]))), axis=0)
    elif rows2 < rows1:
        image2 = np.concatenate((image2, np.zeros((rows1 - rows2, image2.shape[1]))), axis=0)

    # create new image with two input images appended side-by-side, then plot matches
    image3 = np.concatenate((image1, image2), axis=1)


    # note outliers in this image - RANSAC will remove these later
    plt.imshow(image3, cmap="gray")
    column1 = image1.shape[1]

    # plot each line using the indexes recovered from pairs
    for index in range(len(pairs)):
        index1, index2 = pairs[index]
        plt.scatter([interestPoints1[index1][1], interestPoints2[index2][1] + column1],
                    [interestPoints1[index1][0], interestPoints2[index2][0]], color = 'r', s = 2)
        plt.plot([interestPoints1[index1][1], interestPoints2[index2][1] + column1],
                    [interestPoints1[index1][0], interestPoints2[index2][0]], color = 'g', linewidth = 0.5)
    plt.axis('off')
    plt.show()
```

Figure.10: Code implementation of plotting matched interesting points.

After all the methods have been completed, the methods need to be called in order to generate and output the output in turn.

```
# Use to get Normalised Image Patches (Image Descriptors) for both images
descriptors1 = Vectors_Descriptors(harrisImage1, interestPoints1)
descriptors2 = Vectors_Descriptors(harrisImage2, interestPoints2)

# Use to matches between Descriptors
Maxvalue_image1, Maxvalue_image2, Max_dotproduct, Original_matrix, Thresholded_matrix, Pairs_list = Match_Descriptors(descriptors1, de
# Output the maximum value of images
print("Maximum of Image1: " + str(Maxvalue_image1))
print("Maximum of Image2: " + str(Maxvalue_image2))
print("Maximum of Dot Product: " + str(Max_dotproduct))

# Print the response matrix binary image
print("\nResponse matrix before and after thresholding: ")
plt.subplot(121)
plt.imshow(Original_matrix)
plt.subplot(122)
plt.plot([0,])
plt.imshow(Thresholded_matrix)
plt.show()

# Output the match image
print("\nPlot the matches between the two images:")
result = Plot_Matches(harrisImage1, harrisImage2, interestPoints1, interestPoints2, Pairs_list)

```

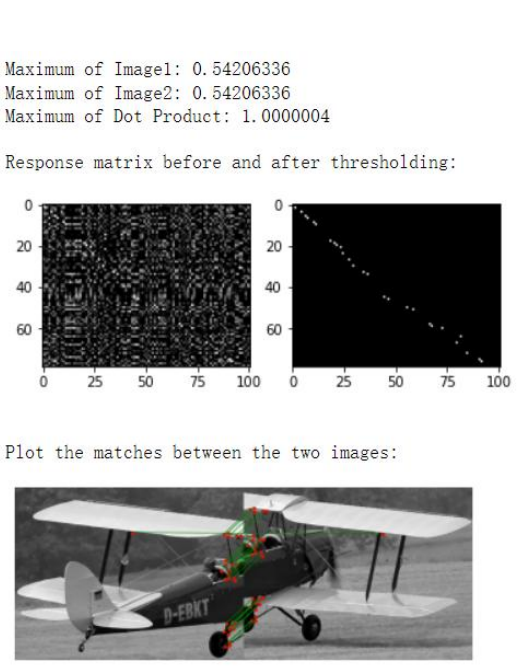
Figure.11: Call method to display implementation output.

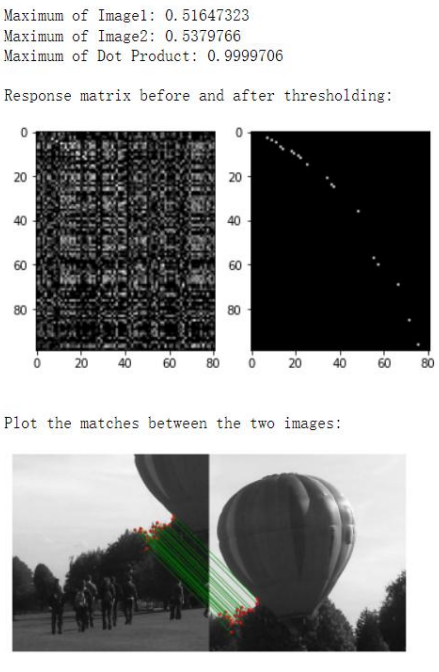
Outputs the maximum value of image 1, the maximum value of image 2, and the maximum value of the dot product. as shown in Figure 12.

```
Finding Normalised Image Patches (Image Descriptors) for both images
OK

Finding matches between Descriptors
Maximum of Image1: 0.54206336
Maximum of Image2: 0.54206336
Maximum of Dot Product: 1.0000004
```

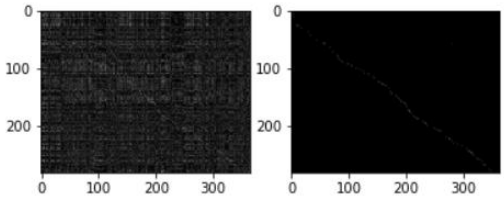
Figure.12: Maximum value Images and Dot product.

Outputs the result of the dot product of the two images, outputting the result after a threshold filter.





```
Maximum of Image1: 0.5254465
Maximum of Image2: 0.34587705
Maximum of Dot Product: 1.0000006

Response matrix before and after thresholding:
```




```
Plot the matches between the two images:
```

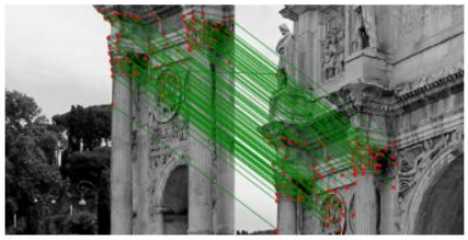


Figure.13: The output of Max Dot Product and the Interesting pairs in 3 images.

3. Image stitching

3.1 Built RANSAC & Find Best Offset

Construct a method to compare the position differences of all pairs of interest points. The final offset is found by summing the deviations between each pair of deviating similar interest points and eventually dividing by the number of total similar interest differences. This approach can ignore the pairs that are incorrectly matched and improve the correct matching rate.

```
# To Get the offset to conbine pictures
def RANSAC(matches, coordinates1, coordinates2 ):
    # Define a distance as a condition to judgement distance and jusify the offset
    matchDistance=1.6
    d2 = matchDistance ** 2

    # Build a list of offsets from the lists of matching points for the 2 images.
    # Use two array to store 2 offset in colum and row.
    offsets = np.zeros((len(matches), 2))
    for i in range(len(matches)):
        index1, index2 = matches[i]
        offsets[i, 0] = coordinates1[index1][0] - coordinates2[index2][0]
        offsets[i, 1] = coordinates1[index1][1] - coordinates2[index2][1]

    # Use the for loop to clean noise and comparison each pairs offset and return a average offset.
    # If the images are parallel, The offset of each pair are similar, so the offset will be strongly useful.
    # Sometimes, If the images is rotated and scalared, the offset maybe fail.
    # Run the comparison.  best_match_count keeps track of the size of the
    # largest consensus set, and (best_row_offset,best_col_offset) the
    # current offset associated with the largest consensus set found so far.
    best_match_count = -1
    best_row_offset, best_col_offset = 1e6, 1e6
    for i in range(len(offsets)):
        match_count = 1.0
        offi0 = offsets[i, 0]
        offi1 = offsets[i, 1]
        # Use j loop looking for consensus if this point hasn't
        # been found and folded into a consensus set earlier.  Just improves
        # efficiency.
        if (offi0 - best_row_offset) ** 2 + (offi1 - best_col_offset) ** 2 >= d2:
            sum_row_offsets, sum_col_offsets = offi0, offi1
            for j in range(len(matches)):
                if j != i:
                    offj0 = offsets[j, 0]
                    offj1 = offsets[j, 1]
                    if (offi0 - offj0) ** 2 + (offi1 - offj1) ** 2 < d2:
                        sum_row_offsets += offj0
                        sum_col_offsets += offj1
                        match_count += 1.0
            if match_count >= best_match_count:
                best_row_offset = sum_row_offsets / match_count
                best_col_offset = sum_col_offsets / match_count
                best_match_count = match_count

    return best_row_offset, best_col_offset, best_match_count
```

Figure.14 Implementation of RANSAC method.

Once the optimal horizontal and vertical offsets have been found, the images are moved and cropped accordingly to obtain the final stitching result. The code implementation is shown in Figure 15.

```python
# Conbine 2 pictures
def Append_Images(image1, image2, rowOffset, columnOffset):
    # Convert floats to ints
    rowOffset = int(rowOffset)
    columnOffset = int(columnOffset)

    # create new 'canvas' image with calculated dimensions
    canvas = Image.new(image1.mode, (image1.width + abs(columnOffset), image1.width + abs(
        rowOffset)))
    canvas.paste(image1, (0, canvas.height - image1.height))  # paste image1
    canvas.paste(image2, (columnOffset, canvas.height - image1.height + rowOffset))  # paste image2

    # plot final composite image
    plt.figure('Final Composite Image')
    plt.imshow(canvas)
    plt.axis('off')
    plt.show()

    return canvas
```

Figure.15: Image stitching.

Call RANSAC and the stitch image method to stitch the initial two original images.

```python
# Output the images aligmnet
# Use RANSAC to clean the noise
rowOffset, columnOffset, bestMatches = RANSAC(Pairs_list, interestPoints1, interestPoints2)
print('Number of agreements (best match count): ' + str(bestMatches))
print('Row offset: ' + str(rowOffset))
print('Column offset: ' + str(columnOffset))

print("Final Image Reconstruction:")
colourImage1 = Image.open('image-pairs/tigermoth1.png')
colourImage2 = Image.open('image-pairs/tigermoth2.png')
final = Append_Images(colourImage1, colourImage2, rowOffset, columnOffset)
```

Figure.16: Call method to implement image stitching.

The final horizontal and vertical offsets are shown in Figure 17. The final stitching result of the image is shown in Figure 18.

```
RANSAC Operation to clean up the noisy mapping above:
Number of agreements (best match count): 26.0
Row offset: 70.0
Column offset: 565.9615384615385
```

Figure.17: The row offset and column offset.

```
Final Image Reconstruction:
```

Figure.18: The final output of Image stitching

For the other 2 images. The output is:

```
Number of agreements (best match count): 99.0
Row offset: -279.0
Column offset: 144.0

Final Image Reconstruction:
```

Figure.19: The image stitching in arch1.png and arch2.png

```
Number of agreements (best match count): 19.0
Row offset: -219.0
Column offset: 293.0

Final Image Reconstruction:
```

Figure.20: The image stitching in balloon1.png and balloon2.png

3.2 Rotation and Scaling Adaptability

Only the stitching of images in normal conditions was discussed previously, and obviously, as shown above, the results are excellent. But next we will discuss the matching of the images in the case of rotation and scaling. where the stitching is good or bad and fails completely. In my opinion, the stitching can tolerate not being perfectly matched, but it can fail.

I will be testing rotation angles from 0-20° and no greater than 20°, and I feel that the match will fail when the rotation angle is greater than 20 degrees. So 20° or less is sufficient for testing. In addition I will test scaling between 0.5 and 1.5. Test when it will fail.

The rotation and scaling of the image is implemented first and the code implementation is shown in Figure 21.

```
# Investigation for scalar and rotation
Image1 = Image.open('image-pairs/arch1.png').convert('L')
Image2 = Image.open('image-pairs/arch2.png').convert('L')

# Roaate
rotates = 4
print("Rotating angle: ", rotates)
RotateImage2 = Image2.rotate(4)
RotateImage2 = np.array(RotateImage2, dtype=np.float32)
#print(RotateImage2.shape)
imutils.imshow(RotateImage2)

# Scalar
#ScalarImage2 = Image2.HAMMING
scalar = 1.25
print("Scaling multiplier: ", scalar)
W, H = Image2.size
ScalarImage2 = Image2.resize((int(W*scalar), int(H*scalar)))
ScalarImage2 = np.array(ScalarImage2, dtype=np.float32)

imutils.imshow(ScalarImage2)
print("Original image: ")
Image_array = np.array(Image2, dtype=np.float32)
imutils.imshow(Image_array)
```

Figure.21: Code implementation of image rotation and scalar.

The images are scaled for comparison, as shown in Figure 22. The scaling is not easy to spot here and can be found by comparing the image's axes to find out how it is scaled.

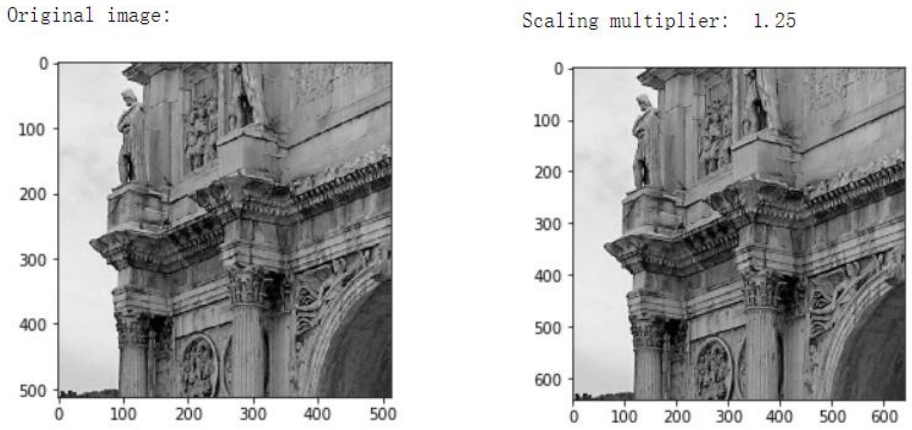


Figure.22: Scalar comparison.

The images are rotated for comparison, as shown in Figure 23.

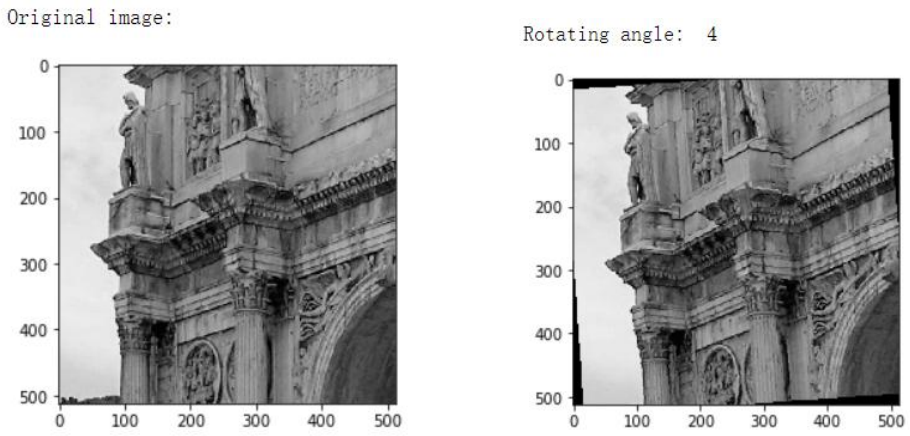


Figure.23: Rotated comparison.

When scaled 1.3x, and 1.4x. At 1.4x, it can basically already be considered invalid.

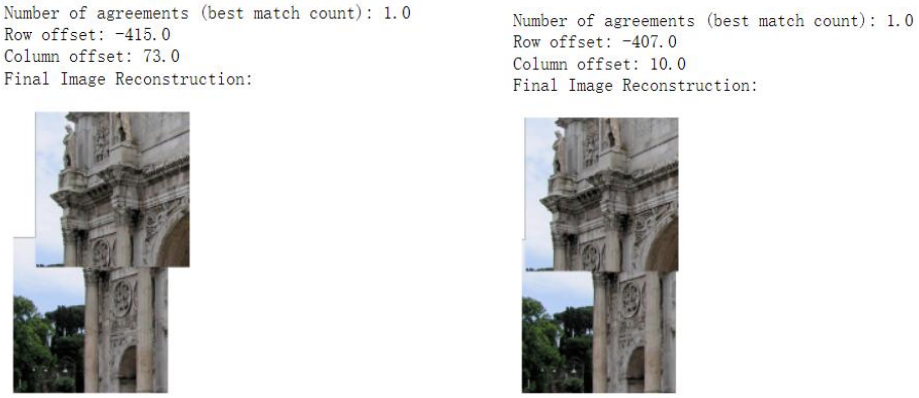


Figure.24: Scalar 1.3 (Left), 1.4 (Right).

When scaled 0.8x, and 0.75x. At 0.7x an error occurs on the run. Direct failure. There are no longer any points of interest right. As can be seen in Figure 23, 0.8x is less effective than 0.75x. This is because 0.8x has more interest point matching pairs and there is the effect of having a lot of interest point pairs in the calculation of the offsets. However, at 0.75 there is only one pair of interest points, so the offset comes from this one pair only, and the results are better understood.

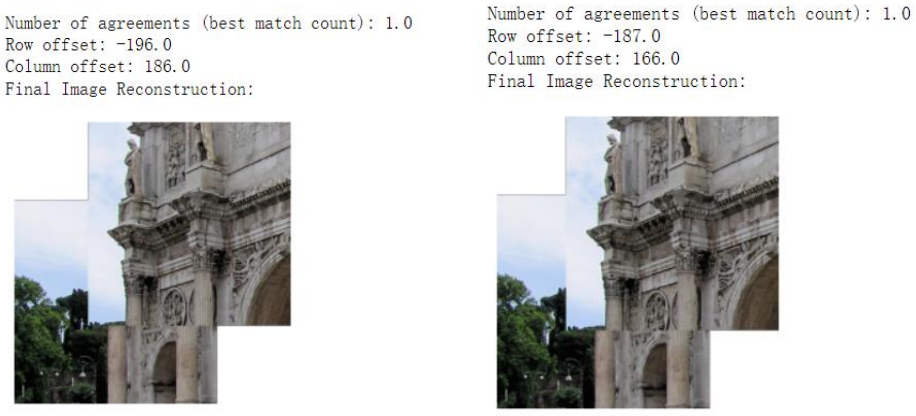


Figure.25: Scalar 0.8 (Left), 0.75 (Right).

When the image is rotated, the code fails after a rotation of 4 degrees without changing the sigma parameter when forming the Harris matrix. The code fails at 15 degrees when the value of sigma increases as the rotation is read. The output at 4 and 14 degrees is shown in Figure 26.

```
Number of agreements (best match count): 24.0
Row offset: -278.2083333333333
Column offset: 137.83333333333334
Final Image Reconstruction:
```

```
Number of agreements (best match count): 1.0
Row offset: -219.0
Column offset: 122.0
Final Image Reconstruction:
```

Figure.26: Rotation degree is 4 (Left), 14(Right).

For the image Balloon, scaling fails below 0.75 and can be considered to fail at 1.35.

```
Number of agreements (best match count): 1.0
Row offset: -117.0
Column offset: 303.0
Final Image Reconstruction:
```

```
Number of agreements (best match count): 1.0
Row offset: -366.0
Column offset: 177.0
Final Image Reconstruction:
```

Figure.27: Scalar 0.75 (left), 1.35 (Right)

For the image Balloon, the rotation works fine at 8 degrees and fails at 9 degrees and the code runs with an error.

```
Number of agreements (best match count): 1.0
Row offset: -236.0
Column offset: 274.0
Final Image Reconstruction:
```

Figure.28: Rotation degree at 8.

For the image Tigermoth, the scaling multiplier 0.8 already shows a clear error, and the multiplier at 1.25 shows a serious error. The graph below shows the match at the limit value of 0.8 and 1.2

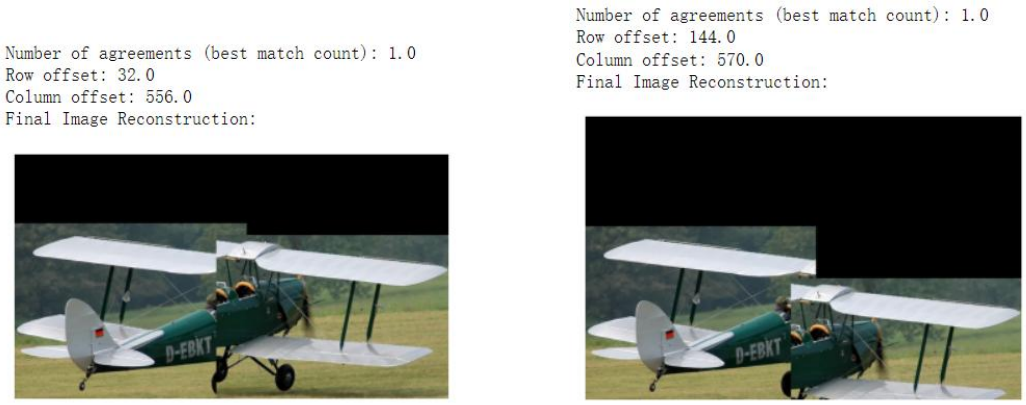


Figure.27: Scalar 1.2 (left), 0.8 (Right)

For the image Tigermoth, the rotation is fine at 6 degrees and fails at 7 degrees, giving a severe mis-match.

Figure.30: Rotation degree at 6.

I have found that when the stitching is done to different images, the limits when it fails are different. This depends on the number of matched pairs when the result is optimal. The higher the number of matched pairs, the better the resistance to rotation and scaling. The matching limit is reached when there is only one matching pair left, so when the rotation or scaling is such that there is only one matching pair left, and the degree of change is increased, there will be a false match or the code will run with an error, showing a value of 0.

4. Conclusion

The entire project process has been clearly presented. All images have been successfully stitched together. During the development process, the algorithm provided by Colin for finding points of interest provided a very clever way of doing this. Using Boolean arrays and determining the range of points of interest to suppress obscure pixel values increases the efficiency of extracting valid values. This is a very efficient way of processing. If this processing is not used in this section, it will significantly reduce the performance of the code. In addition, it is extremely sensible to use dot product operations after the description vector has been formed and to use thresholding to filter the matching pairs. The coordinates of the leftover dot product values are returned to obtain the match pair results. In addition, a matching distance parameter is used in the RANSAC process to filter out matches that cause significant deviations from the overall offset calculation. This approach enhances the resistance to incorrect pairings. These are a few of the subtle ways in which the entire code was handled during development.

Throughout the development process, the ability to cope with code related to machine vision has been greatly enhanced by training on the first two projects. As a result, there were no major problems in principle compared to the previous two projects. Of course, there are some problems that the team's own ideas would not be able to deal with in the way that the current code does. This is due in large part to the guidance provided by Colin.

This is the final project of the course and I would like to thank Colin for the excellent course content and the challenging and practical project design. By taking this course, we have gained a clear understanding of machine vision and a solid foundation for our future studies. Once again, I would like to thank Colin for his efforts in this course and hope that we can stand shoulder to shoulder with you in our understanding of AI in the future. We also hope to have the opportunity to work with you again in the future.

5. Reference

[1] C. Flanagan. 03-project-harris.4up [PDF]. Available: https://sulis.ul.ie/access/content/group/82edf4ed-f03d-4ab6-bf99-2773fa7cd801/Test%20Images/Harris%20Corner%20Detection/project3-harris.4up.pdf