
# Harris Corner Detector & Image Stitcher

## RE4017 - Dr Colin Flanagan

Finn Hourigan 17228522

Ronan Reilly 18242367

Brendan Lynch 18227651

Barry Hickey 18243649

# Introduction

The project aims to find matching points between two images using Harris interest points (Hips) and normalized patch descriptors. The process involves thresholding Harris response images for image 1 and image 2 to detect Hips, and then iterating through various angles and scalings to find the best match. Normalized patch descriptor vectors are formed for all the Hips in both images, and matches are found using inner product operation and thresholding for strong matches. The resulting list of point correspondences is sorted by match strength, with the strongest matches appearing first. To filter out outliers, an exhaustive RANSAC algorithm is applied, and the best translation between the images is returned as the result. Finally, the translation is used to create a composite image, which is returned as the final output.

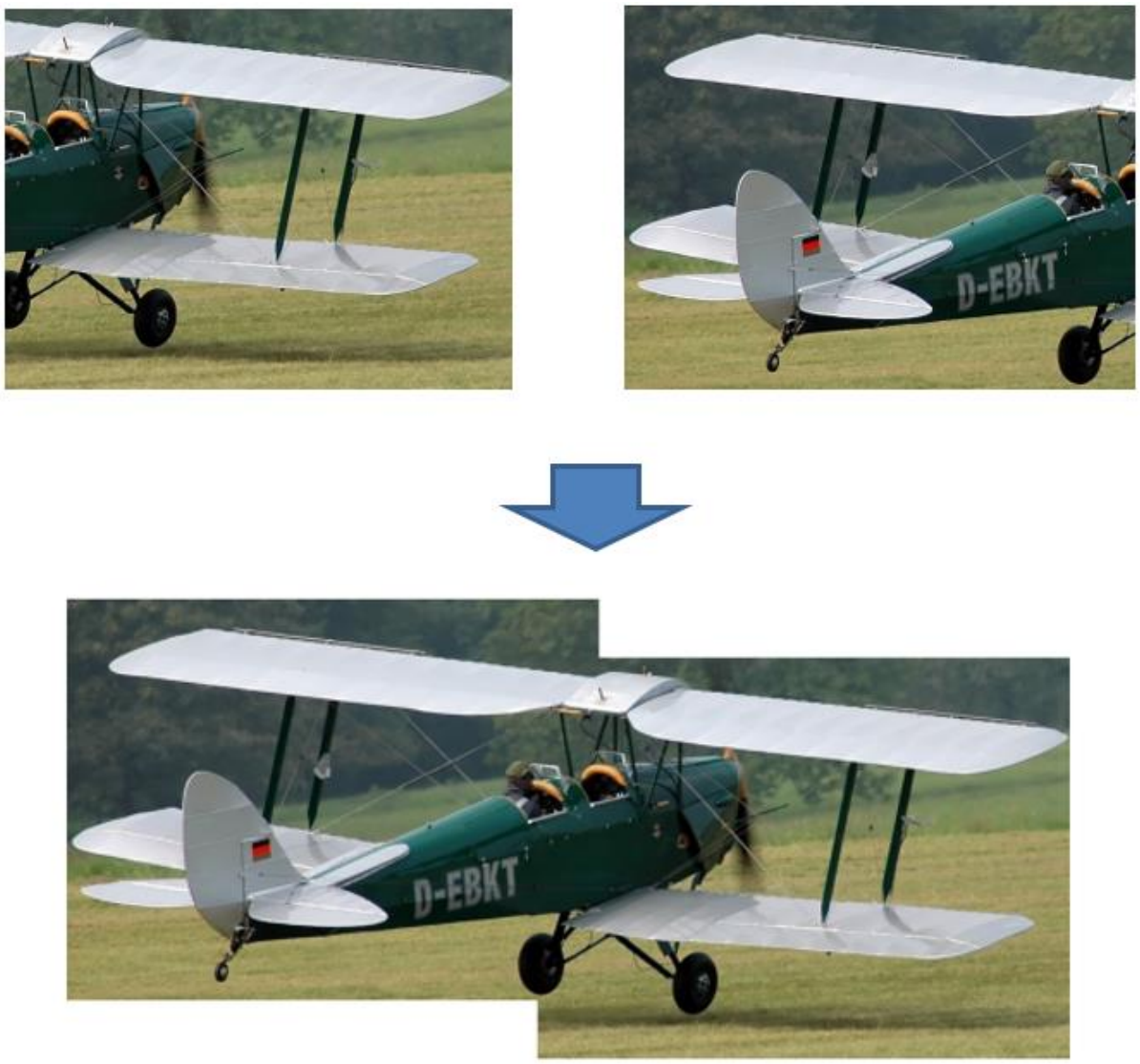


*Figure 1 - Image matching & stitching process*

# Operation of image matcher / stitcher

1) Open Anaconda prompt
2) Navigate to location of proj3.py
3) Enter command prompt, specifying the two images to be matched and stitched. Example:

```
(base) C:\RE4017\RE4017_Project_3>python proj3.py arch1.png arch2.png
```

4) The program will ask if you want to check for rotation or scaling. Reply Y/N

```
(base) C:\RE4017\RE4017_Project_3>python proj3.py arch1.png arch2.png
Do you want to check for rotation? (Y/N): n
Do you want to check for scaling? (Y/N): n
```

5) The program will now run and the resulting combined image (if generated) will appear in its own window.

*Figure 2 - Output of arch1.png and arch2.png*

# Methodology

The most logical way to present the methodology of this python program is to step through the main function and explain each the function of each line of code / function as we progress.

## Step 1. Ask user if rotation or scaling checks required

This program can accommodate matching and stitching images which may require slight adjustments in angle and scale. The program iterates over a range of angles and scales for the second image and then choses the values which provided the best match. This iteration process can be slow however so the choice is provided.

The function **rotation_or_scaling_required** carries out this process with a simple while-True loop for checking user input validity.

If **check_scale** is True, the array **scales** will contain values from 0.5 to 1.5 which will be iterated through using a for-loop. Otherwise, a scale value of 1 is used.

If **check_angle** is True, the array **angles** will contain values from -45 to +45 which will be iterated through using a for-loop. Otherwise, an angle value of 0 is used.

Once **scales** and **angles** have been defined, an iteration loop is set up, with the intention of finding the best value of **scale** and **angle** for the given images.

```
for scale in scales:
    for angle in angles:
```

*Figure 3 - Iteration loop*

## Step 2. Find strong agreements at given angle and scale

The function **find_translation** returns three values, **dr,dc** and **most_agreements.** In Step 2, only the **most_agreements** value will be needed.

### Step 2.1. Rotating and scaling images as required

To begin, **image_1** and **image_2** are defined as normalized greyscale numpy arrays of the two input images on the command line.

Each iteration uses a different combination of values for **scale** and **angle**. In this project for the sake of demonstration **image_2** is always the rotated / scaled image.

Rotation through **angle** is achieved using ndimage.rotate, and scaling is achieved using custom function **scale_array** which uses Pillow to change the height and width of the image accordingly. The result is **image_2** rotated by **angle** degrees and scaled by **scale.**

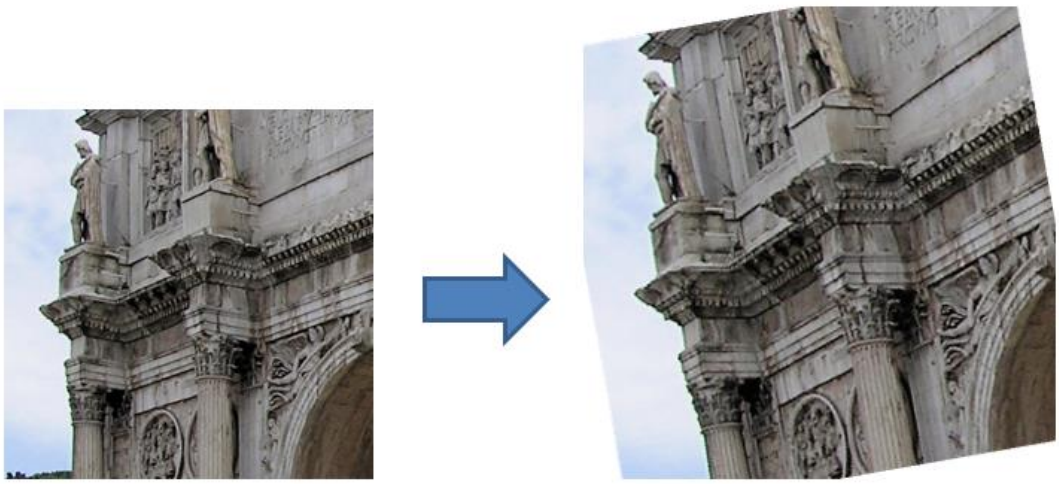

*Figure 4 - Rotation and scaling of image arch2.png*

## Step 2.2 Form Harris responses & find Harris interest points (HIPs)

The Harris corner detection algorithm is used to find corners in each image and then identify a list of Harris interest points. This is achieved using **harris_response()** and **get_harris_points()** as follows:

1) In **harris_response,** edges found by convolving with derivative of Gaussian x and y kernels, taking **sigma** as 1.
2) Next, values for $I_X^2(x,y)^2$ , $I_X(x,y)$ , $I_Y(x,y)$ and $I_Y(x,y)^2$ are determined.
3) Find elements **A, B** & **C** of the Harris matrix using the Gaussian filter with **sigma**2 = **sigma2** * 2.5 as follows:

   $A = G_{\sigma 2} * I_x^2,\ B = G_{\sigma 2} * I_x I_y,\ C = G_{\sigma 2} * I_y^2.$

   In python:

```
# Convolve with patch filter to find A,B,C as follows
A = gaussian_filter(I_x * I_x, sigma2 )
B = gaussian_filter(I_x * I_y, sigma2 )
C = gaussian_filter(I_y * I_y, sigma2 )
```

*Figure 5 - Find A,B and C*

4) Find determinant and trace of Harris matrix as follows:

```
# Find determinant and trace
det_M = (A * C) - (B * B)
tra_M = A + C + 0.00001     #addition of 0.00001 avoids possible zero division error
```

*Figure 6 - Find det_M and tra_M*

5) Harris image is found by dividing **det_M** by **tra_M.**
6) Using **get_harris_points(),** sample code given for this function achieves the following:
   - Find all HIP candidates above given threshold * max value
   - Find the co-ordinates of these candidates and their values using **argsort()**
   - Use co-ordinates to extract these candidate values in the greyscale image
   - Select the best points, using nonmax suppression based on the array of allowed locations
   - Return **filtered_coords,** the Harris interest points in each image.
7) Back in the **find_translation** function these values are stores in the numpy arrays **HIPs_image_1** and **HIPs_image_2**.

*Figure 7 - HIPs in arch2.png*

*Figure 8 - HIPS in arch1.png*

## Step 2.3 Form normalized 121-element patch descriptor vectors

The function **get_descriptor_vectors** forms normalised 121-elemnt patch descriptor vectors for all Harris point locations in the given image. This is achieved using the following steps:

1. Form an image patch around the first Harris interest point (11x11 pixels)
2. Flatten it into a 121 element vector.
3. Subtract the mean of that vector from the vector itself, creating a zero-mean vector
4. Normalize the vector so values range from -1 to +1 bu diving by **np.linalg.norm** of vector
5. Repeat for all interest points, stacking the vectors each time in **descriptors** using **.append** routine.

Returns **descriptors** , an array where the first element is the descriptor vector of the patch surrounding the first Harris point, and so on.

Repeat for image 2.

## Step 2.4. Generate response matrix from descriptor vectors and filter

Using **get_response_matrix** ,the response matrix **r12** for the two given sets of descriptor vectors **m1** and **m2** is achieved by finding the “outer product” of the two sets of descriptor vectors as follows:

```
#The response matrix r12 is generated as the "outer product" of m1 and m2 transposed
r12 = np.dot(m1, m2.T)
```

*Figure 9 - Finding response matrix*

The response matrix elements are indicative of the “match strength” between the descriptor vectors. When visualised, white pixels (or close to white) indicate matches between corresponding Harris points.

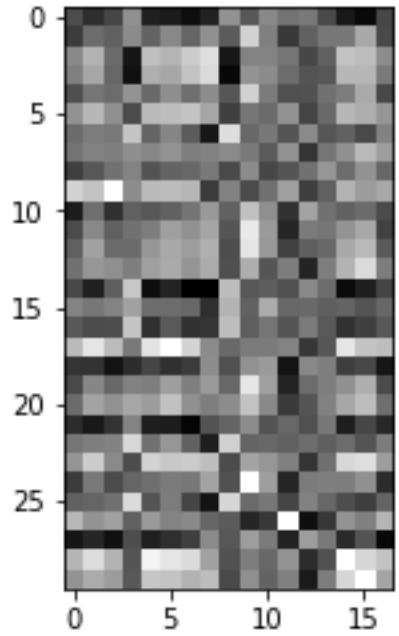


*Figure 10 - Harris response matrix (unfiltered)*

The function **find_matches** is then used to filter out all matches that are below a given threshold. A threshold of 0.95 is used in this case.

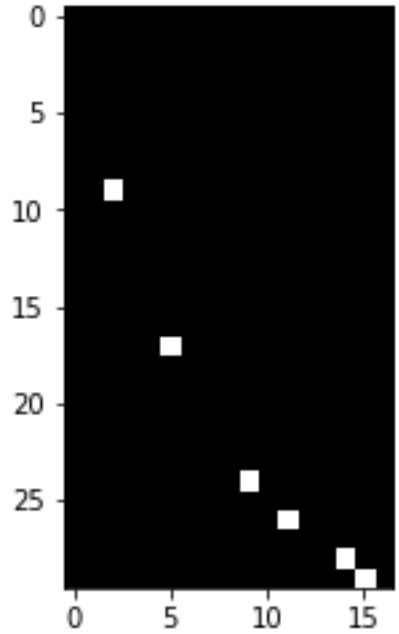


*Figure 11 - Harris response matrix, filter = 0.95*

All matching pairs are returned in the array **matches.** Back in the **find_translation** function, this array is stored as **matched_HIP_pairs**.

## Step 2.5. Use RANSAC algorithm for finding best translation and number of strong agreements

**get_best_translation_RANSAC** function is used to find the best translation between points using an exhaustive RANSAC algorithm. The following image indicates corresponding points:

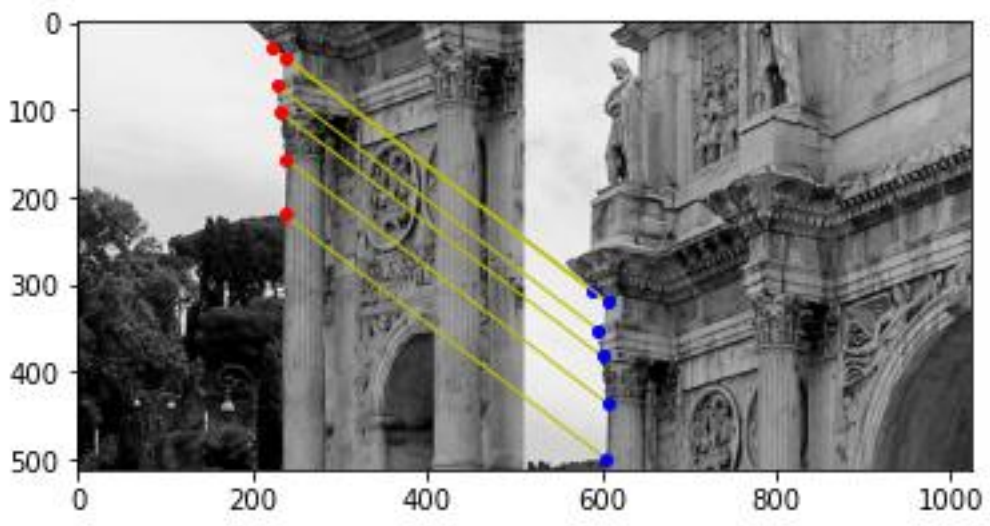


*Figure 12 - Plotting of mappings between matching points*

The parameter **corresponding_HIP_pairs** can be used to generate an array of translations using the co-ordinates in **HIPs_image_1** and **HIPs_image_2.** This array is named as **translations** and contains the row and column translation for every pair of points that map to each other. It is determined as follows:

```
for i in range(len(corresponding_HIP_pairs)):

    p1,p2 = corresponding_HIP_pairs[i]
    # p1 identifies an interest point location in the array HIPs_image_1 that
    # corresponds with the interest point identified by p2 in HIPs_image_2

    # In other words, the calculations thus far suggest that these two
    # points should 'overlap' when the final combined image is generated

    # find translation of corresponding points:

    # find row translations for translation i
    translations[i][0] = HIPs_image_1[p1][0] - HIPs_image_2[p2][0]
    #find column translation for translation i
    translations[i][1] = HIPs_image_1[p1][1] - HIPs_image_2[p2][1]
```

*Figure 13 - Finding array of translations when mapping from p1 to p2*

All these translations should be similar to one another, but there will always exist a possibility of outliers. Thus an exhaustive RANSAC algorithm is employed, with a threshold of 1.6 pixels, to determine which translation has the most agreements with the other translations in the list.

This is achieved by comparing the Euclidian distance between each translation and every other translation. Any instance in which the Euclidian distance is less than the threshold counts as an agreement i.e. a very similar translation, and increments the **agreements** variable by one. The translation with the highest number of **agreements** (number stored as **most_agreements)** is chosen

as **best_translation**. From **best_translation,** values for **dr** and **dc** are extracted and are returned as well as **most_agreements.**

Steps 2.1 – 2.5 repeated for however many iterations is required to use all combinations of **scale** and **angle** arguments.

## Step 3. Storing of iteration results

Each iteration, at a given **angle** and **scale**, results in a value for **most_agreements**. This value, the **angle** value and the **scale** value are stored in their respective storage arrays as follows:

```
agreements_at_given_angle_and_scale.append(most_agreements)
angles_list.append(angle)
scales_list.append(scale)
```

*Figure 14 - Storing iteration values*

## Step 4. Determining iteration which produced the best match

First, it is determined that at least one iteration resulted in at least one strong match. This is carried out using the following code snippet:

```
for num in agreements_at_given_angle_and_scale:

    if num > 0:
        valid_match = True
        break
```

*Figure 15 - Check that at least one iteration resulted in at least one match*

If **valid_match** is True, then the index of the best match is determined by the iteration with the largest number of strong mathes and is stored as **best_match_index**. If no matches are found in any iteration (**valid_match** is False), then the program ends at this point with the message *"No matches found!"*.

**best_match_index** is used to find **best_angle** and **best_scale** from **angles_list** and **scales_list** arrays. These values are printed to the console.

```
Best rotation = 0 degrees

Best scale = 1
```

*Figure 16 - Console output for arch1.png and arch2.png*

As expected for arch1.png and arch2.png, the best angle of rotation is 0 degrees and best scale is 1.

## Step 5. Find best translation.

It has been determined that the best translation between the two images occurs when **image_2** undergoes rotation through **best_angle** and is scaled by a factor equal to **best_scale**. Thus, to find the row translation **dr** and column translation **dc** that correspond to the best translation, we must use the function **find_translation** once more with **best_scale** and **best_angle** as arguments.

```
dr, dc, most_agreements = find_translation(best_scale,best_angle)
```

*Figure 17 - Finding dc and dr for the best translation*

This function incorporates all functions listed in Step 2.1 – 2.5. The values for **dr** and **dc** will be used when combining the final images.

## Step 6. Transform image objects as required

Image objects **final_image_1** and **final_image_2** are created using PIL's **Image.open()** routine, opening the images from file.

**final_image_2** undergoes rotation and scaling using the values found for **best_angle** and **best_scale**.

```
# Step 6. Transform image objects as required
final_image_1 = Image.open(f"./Test Images/{img1_name}")

final_image_2 = Image.open(f"./Test Images/{img2_name}")
final_image_2 = final_image_2.rotate(best_angle)
final_image_2 = scale_image(final_image_2, best_scale)
```

*Figure 18 - Transformation of final_image_2*

**scale_image()** is a custom function which takes the image and required scaling as arguments, converts the image object to an array, finds the width and height of the array and scales them by the given **scale** value to find **new_height** and **new_width**. The image object is then resized to **new_height** and **new_width** using PIL **.resize()** routine.

## Step 7. Compose images

Finally, the resulting "stitched" image is generated using the **compose_images** function. The image objects **final_image_1** and **final_image_2** (post-transformation) are combined on top of a white background image object initialised as **canvas**.

The positioning of images on the canvas in relative to one another is decided based on the given values for **dr** and **dc**. There are 4 possible cases, depending on if **dr** and/or **dc** are above or below 0. A simple if-elif statement was used to account for all 4 cases as follows:

```
# Case: 1
if dr > 0 and dc > 0:
    canvas.paste(image2, (int(dc), int(dr)))
    canvas.paste(image1, (0,0))

# Case: 2
elif dr < 0 and dc > 0:
    canvas.paste(image2, (int(dc), 0))
    canvas.paste(image1, (0, int(abs(dr))))

# Case: 3
elif dr > 0 and dc < 0:
    canvas.paste(image2, (0, int(dr)))
    canvas.paste(image1, (int(abs(dc)), 0))

# Case: 4
else:
    canvas.paste(image2, (0,0))
    canvas.paste(image1, (int(abs(dc)), int(abs(dr))))
```

*Figure 19 - Relative positioning of images*

As a result, the images are “pasted” in such a way that the best translation has been used, with all string matches now “on top of” each other. Note that **image_2** is pasted first because it may have gained blank areas around the edge due to rotation, which can be covered by **image_1** if it is pasted first.

The final combined image is then shown:

*Figure 20 - Resultant composed image for arch1.png and arch2.png*

# Conclusion

The program which we wrote proved to be very accurate in identifying the correct rotation, scaling and translation required to match two given test images. It was successful in matching pairs in which one image required rotation, in which one image required scaling, and in which one image required both rotation and scaling. All tested pairs and the resulting composed images can be found in the Appendix.

This proved to be a challenging but rewarding project. Difficulties were encountered especially in understanding the theory behind Gaussian filters and how they could be used to identify corners in images. In addition, it took quite some time to design an algorithm replicating an exhaustive RANSAC.

Overall, we feel that this project benefited us greatly in learning image processing techniques which may be beneficial to us in our future as engineers.

# Appendix

## tigermoth1.png & tigermoth2.png

Command line:

```
python proj3.py tigermoth1.png tigermoth2.png
```

Result:


Best rotation = 0 degrees

Best scale = 1

## arch1.png & arch2.png

Command line:

```
python proj3.py arch1.png arch2.png
```

Best rotation = 0 degrees

Best scale = 1

## arch1.png & arch2_rotated.png

Command line:

```
python proj3.py arch1.png arch2_rotated.png
```

Best rotation = -10 degrees

Best scale = 1

## arch1.png & arch2_scaled.png

Command line:

```
python proj3.py arch1.png arch2_scaled.png
```

Best rotation = 0 degrees

Best scale = 0.8

## arch1.png & arch2_rotated_and_scaled.png

Command line:

```
python proj3.py arch1.png arch2_rotated_and_scaled.png
```

Best rotation = -10 degrees

Best scale = 0.8

## balloon1.png & balloon2.png

Command line:

```
python proj3.py balloon1.png balloon2.png
```

Best rotation = 0 degrees

Best scale = 1

## balloon1.png & balloon2_rotated.png

Command line:

```
python proj3.py balloon1.png balloon2_rotated.png
```

Best rotation = -10 degrees

Best scale = 1

## balloon1.png & balloon2_scaled.png

Command line:

```
python proj3.py balloon1.png balloon2_scaled.png
```

Best rotation = 0 degrees

Best scale = 0.8

# balloon1.png & balloon2_rotated_and_scaled.png

Command line:

```
python proj3.py balloon1.png balloon2_rotated_and_scaled.png
```

Best rotation = -10 degrees

Best scale = 0.8