Harris Corner Detection

RE4017 Project 2

Ryan Jennings - 15152324

Instructions

Research and implement Harris Corner Detection using Python/Numpy. Investigate the behaviour of the algorithm. Is it rotation invariant? Is it scale-invariant?

Research an appropriate region descriptor for the Harris detector and use it to find correspondence points between pairs of images. Use these correspondences to align the image pairs and generate a composite image.

Report on the performance of your detector using some test image pairs. Suitable test images pairs can be found in the Resources section.

Provide your Python/Numpy code, all input images used, and a short (3-5 page) report in pdf showing the behaviour of your code, with the aligned images shown clearly. Submit your work as a zip file (no .rar files please).

You may work by yourself or in a group with up to two others (max group size is three).

Imports

Standard imports for working with images. CV2 and PIL for image handling and matplotlib, numpy and scipy for functions to work on the image data

In [2]:

```
import cv2

import matplotlib.pyplot as plt
import numpy as np

from PIL import Image
from scipy.ndimage import filters
```

Read images into array format

In [3]:

```
image1 = cv2.imread('arch1.png')
image2 = cv2.imread('arch2.png')
```

Images

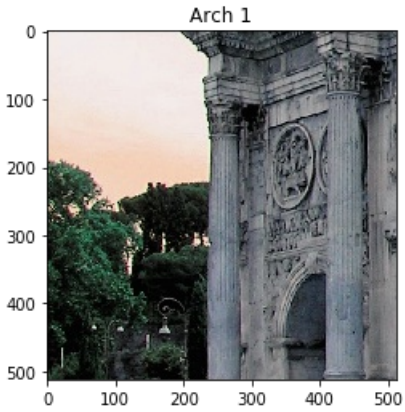
The first images, two images of the same arch are displayed below. Our objective will be to line up the images then analyse lining up the images with respect to changing scale and rotation.

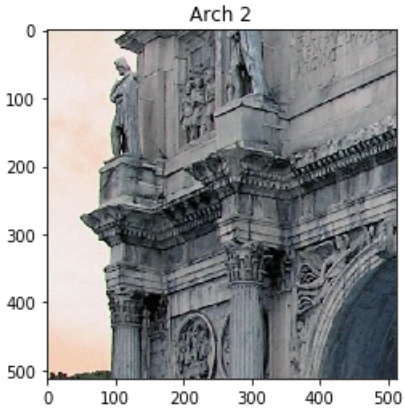
In [4]:

```
plt.figure()
plt.imshow(image1)
plt.title('Arch 1')

plt.figure()
plt.imshow(image2)
plt.title('Arch 2')
```

Out[4]:

```
Text(0.5, 1.0, 'Arch 2')
```





Harris Corner Detection Algorithm

Find R for thresholding

1. Find edges in image I(x,y) by convolving with derivative of Gaussian x & y kernels (σ = 1) to give Ix(x, y) & Iy(x, y).
2. Form Ix(x, y)2, Ix(x, y) Iy(x, y) and Iy(x, y)2 for each point.
3. Convolve these with the patch filter Gσ2 (typically a Gaussian, σ = 2), to give: A = Gσ2 *Ix2, B* = *Gσ2* IxIy, C = Gσ2 * Iy2.
4. Form: R = det(M)–κ(tr(M)2) = (AC–B2)–κ(A+C)2.

In [32]:

```
def harris(im, sigma=3):
    # Performs the Harris corner detector response function
    # derivatives
    imx = np.zeros(im.shape)
    filters.gaussian_filter(im, (1, 1), (0, 1), imx)
    imy = np.zeros(im.shape)
    filters.gaussian_filter(im, (1, 1), (1, 0), imy)

    # compute components of the Harris matrix
    A = filters.gaussian_filter(imx * imx, sigma=2)
    B = filters.gaussian_filter(imx * imy, sigma=2)
    C = filters.gaussian_filter(imy * imy, sigma=2)

    # determinant and trace
    R_determinant = A*C - B**2
    R_trace = A + C

    # Return harmonic mean
    return R_determinant / R_trace
```

Find harris interest points

Threshold the points and perform nonmax suppresion

In [6]:

```
def harris_points(harris_im, min_d=10, threshold=0.1):
    # Python/Numpy code for selecting Harris corner points above a threshold and performing nonmax

    # suppression in a region +/- min_d. “min_d” is the nonmax supression “1/2 width”.

    # (1) Find top corner candidates above a threshold.
    corner_threshold = harris_im.max() * threshold
    harris_im_th = (harris_im > corner_threshold)

    # (2) Find the co-ordinates of these candidates, and their response values
    coords = np.array(harris_im_th.nonzero()).T
    candidate_values = np.array([harris_im[c[0],c[1]] for c in coords])

    # (3) Find the indices into the ‘candidate_values’ array that sort it in order of increasing r
esponse strength.
    indices = np.argsort(candidate_values)

    # (4) Store allowed point locations in a Boolean image.
    allowed_locations = np.zeros(harris_im.shape, dtype='bool')
    allowed_locations[min_d:-min_d,min_d:-min_d] = True

    # (5) Select the best points, using nonmax suppression based on the array of allowed
locations.
    filtered_coords = []
    for i in indices[::-1]: # Note reversed indices, strongest values are at end of index array.
        r,c = coords[i] # If a point is chosen, blank around it in the ‘allowed_locations’ array.
        if allowed_locations[r,c]:
            filtered_coords.append((r,c))
            allowed_locations[r-min_d:r+min_d+1,c-min_d:c+min_d+1] = False

    return filtered_coords
```

Get normalised image patch descriptors

We take a small square around the image of pixel size 2(wid) + 1 and flatten it

In [ ]:

```
def get_descriptors(image, filtered_coords, wid=3):
    #return pixel value
    desc = []
    for coords in filtered_coords:
        # Step 1: Align rows into one vector
        patch = image[coords[0]-wid:coords[0]+wid+1,
                      coords[1]-wid:coords[1]+wid+1].flatten()
        # Step 2: Subtract mean value from all elements
        # Step 3: Divide all values of row vector by its norm
        patch = (patch - np.mean(patch))/np.std(patch)
        desc.append(patch)
    return desc
```

Matching

Find the pairwise distances and if the value is greater than the threshold add it to the matches array to be returned

In [88]:

```
def match(desc1, desc2, threshold=1.0):
    n = len(desc1[0])

    # Find pairwise distances
    matches = []
    for i in range(len(desc1)):
        for j in range(len(desc2)):
            ncc_value = sum(desc1[i] * desc2[j]) / (n - 1)
            if ncc_value > threshold:
                matches.append((i, j))
    return matches
```

Plot image matches

Plot image matches

Attach the images together and draw lines between their matching points

In [90]:

```
def append_images(im1, im2):
    # The appended images displayed side by side for image mapping
    # select the image with the fewest rows and fill in enough empty rows
    rows1 = im1.shape[0]
    rows2 = im2.shape[0]

    # if elseif statement
    if rows1 < rows2:
        im1 = np.concatenate((im1,zeros((rows2-rows1,im1.shape[1]))), axis=0)
    elif rows1 > rows2:
        im2 = np.concatenate((im2,zeros((rows1-rows2,im2.shape[1]))), axis=0)


    return np.concatenate((im1,im2), axis=1)

def plot_matches(im1, im2, locs1, locs2, matchscores, show_below=True):
    # Plots the matches between both images
    im3 = append_images(im1,im2)
    if show_below:
        im3 = np.vstack((im3,im3))

    plt.figure()
    plt.imshow(im3)

    cols1 = im1.shape[1]
    for i, m in matchscores:
        if m > 0:
            plt.plot([locs1[i][1], locs2[m][1]+cols1], [locs1[i][0], locs2[m][0]], 'c')
    plt.axis('off')
```

In [67]:

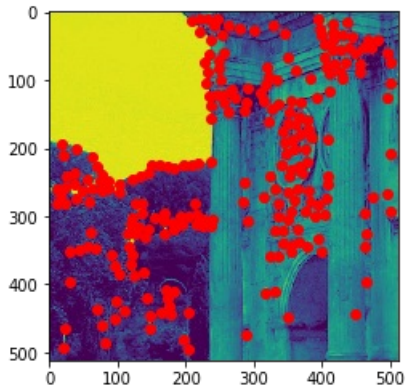
```
def plot_harris_points(im, points):
    plt.figure()
    plt.imshow(im)
    plt.plot([i[1] for i in points], [i[0] for i in points], 'ro')
```

In [66]:

```
r1 = harris(image1[:,:,0])
filtered_coords1 = harris_points(r1)
d1 = get_descriptors(image1[:,:,0], filtered_coords1, 3)
```

In [68]:

```
plot_harris_points(image1[:,:,0], filtered_coords1)
```



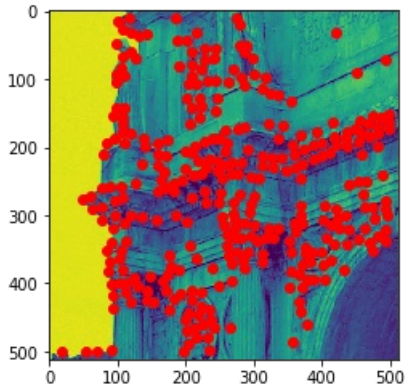
In [69]:

```
r2 = harris(image2[:, :, 0])
filtered_coords2 = harris_points(r2)
```

```
d2 = get_descriptors(image2[:, :, 0], filtered_coords2, 3)
```

In [70]:

```
plot_harris_points(image2[:,:,0], filtered_coords2)
```



In [64]:

```
matches = match(d1, d2)
```

In [65]:

```
plot_matches(image1[:,:,0], image2[:,:,0], filtered_coords1, filtered_coords2, matches)
```

In [84]:

```
def ransac(points1, points2, matches, max_dist=1.0):
    # Square of max distance needed for match calculations
    max_squared = max_dist**2
    best_matches = 0
    offsets = np.zeros((len(matches), 2))
    row_offset, col_offset = 1 * 10**6, 1 * 10**6

    for i in range(len(matches)):
        index1, index2 = matches[i]
        offsets[i, 0] = points1[index1][0] - points2[index2][0]
        offsets[i, 1] = points1[index1][1] - points2[index2][1]

    for i in range(len(offsets)):
        match_count = 1.0
        if(offsets[i, 0] - row_offset)**2 + (offsets[i, 1] - col_offset)**2 >= max_squared:
            row_offsets = offsets[i, 0]
            col_offsets = offsets[i, 1]
            for j in range(len(matches)):
                if j != i:
                    if(offsets[i, 0] - offsets[j, 0])**2 + (offsets[i, 1] - offsets[j, 1])**2 <
max_squared:
                        row_offsets += offsets[j, 0]
                        col_offsets += offsets[j, 1]
                    match_count += 1.0
```

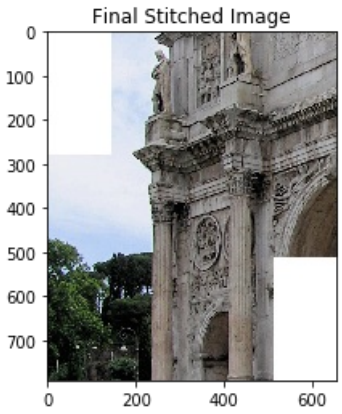
```
        if match_count >= best_matches:
            row_offset = row_offsets / match_count
            col_offset = col_offsets / match_count
            best_matches = match_count

    return row_offset, col_offset

def join_images(im1, im2, row_offset, col_offset):
    width, height = im1.width + abs(int(col_offset)), im1.width + abs(int(row_offset))
    stitched_im = Image.new(im1.mode, (width, height))

    stitched_im.paste(im1, (0, stitched_im.height - im1.height))

    stitched_im.paste(im2, (int(col_offset), stitched_im.height - im1.height + int(row_offset)))

    plt.figure()
    plt.imshow(stitched_im)
    plt.title('Final Stitched Image')
```

In [85]:

```
row_offset, col_offset = ransac(filtered_coords1, filtered_coords2, matches)
```

In [86]:

```
i1 = Image.open('arch1.png')
i2 = Image.open('arch2.png')
join_images(i1, i2, row_offset, col_offset)
```

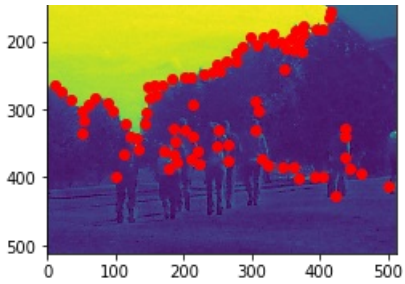


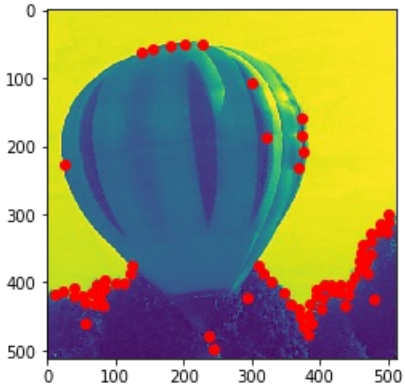
The image stitching finally produces a joint image that perfectly lines up the two images. The speed of the program is also suitably fast as higher thresholds allowed for less noise and less points in general to test in the Ransac function

Testing on Balloon images

In [92]:

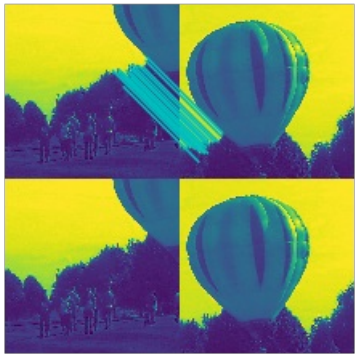
```
balloon1 = cv2.imread('balloon1.png')
balloon2 = cv2.imread('balloon2.png')

br1 = harris(balloon1[:,:,0])
bfiltered_coords1 = harris_points(br1)
bd1 = get_descriptors(balloon1[:,:,0], bfiltered_coords1, 3)
plot_harris_points(balloon1[:,:,0], bfiltered_coords1)

br2 = harris(balloon2[:,:,0])
bfiltered_coords2 = harris_points(br2)
bd2 = get_descriptors(balloon2[:,:,0], bfiltered_coords2, 3)
plot_harris_points(balloon2[:,:,0], bfiltered_coords2)
```





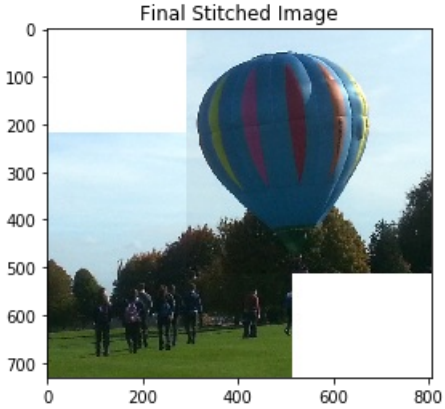
In [93]:

```
bmatches = match(bd1, bd2)
plot_matches(balloon1[:,:,0], balloon2[:,:,0], bfiltered_coords1, bfiltered_coords2, bmatches)
```



In [94]:

```
brow_offset, bcol_offset = ransac(bfiltered_coords1, bfiltered_coords2, bmatches)
bi1 = Image.open('balloon1.png')
bi2 = Image.open('balloon2.png')
join_images(bi1, bi2, brow_offset, bcol_offset)
```



Similar to the arch images the test on the balloon images also produces a suitable result that visually looks correct. Again this program runs without large delays from having to test numerous points

Testing for Rotation Invariance

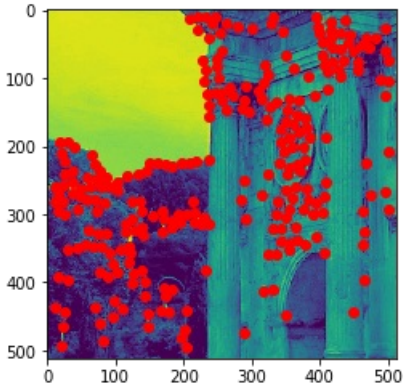
In [109]:

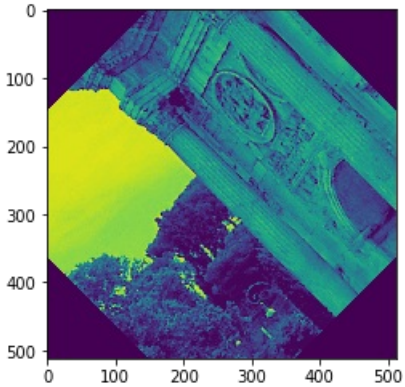
```
def test_point_finding(im1, im2):
    im_r1 = harris(im1)
    coords1 = harris_points(im_r1)
    plot_harris_points(im1, coords1)

    im_r2 = harris(im2)
    coords2 = harris_points(im_r2)
    plot_harris_points(im2, coords2)
```

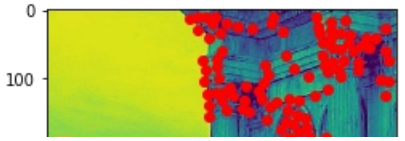
In [111]:

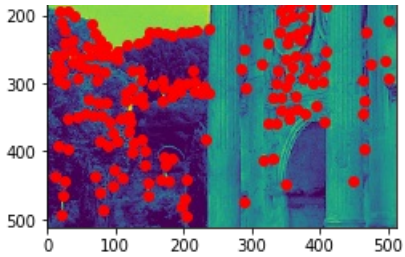
```
arch1 = np.array(Image.open('arch1.png').convert('L'))
rotated_arch1_45 = np.array(Image.open('arch1.png').convert('L').rotate(45))
rotated_arch1_60 = np.array(Image.open('arch1.png').convert('L').rotate(60))
rotated_arch1_90 = np.array(Image.open('arch1.png').convert('L').rotate(90))
rotated_arch1_180 = np.array(Image.open('arch1.png').convert('L').rotate(180))

test_point_finding(arch1, rotated_arch1_45)
test_point_finding(arch1, rotated_arch1_60)
test_point_finding(arch1, rotated_arch1_90)
test_point_finding(arch1, rotated_arch1_180)
```

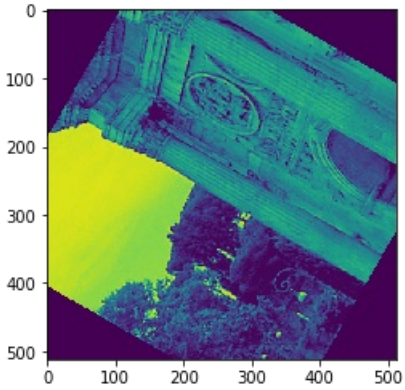
```
/Users/ryan.jennings/.pyenv/versions/3.6.6/lib/python3.6/site-packages/ipykernel_launcher.py:19: R
untimeWarning: invalid value encountered in true_divide
/Users/ryan.jennings/.pyenv/versions/3.6.6/lib/python3.6/site-packages/ipykernel_launcher.py:7: Ru
ntimeWarning: invalid value encountered in greater
  import sys
```

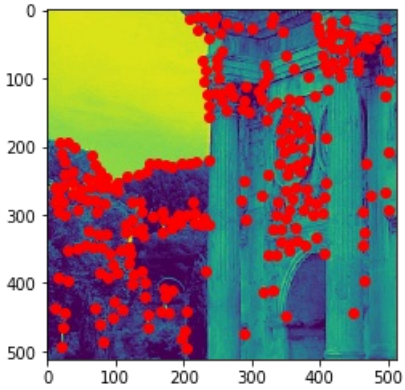


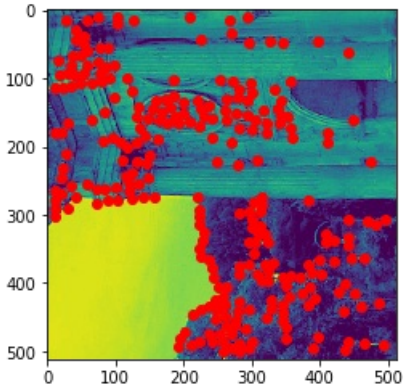


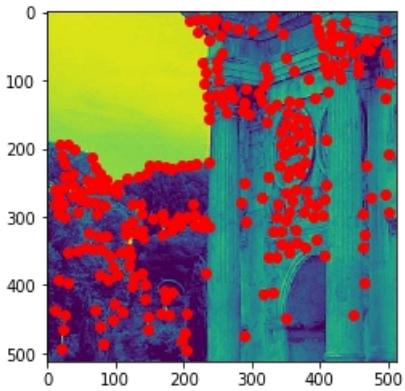


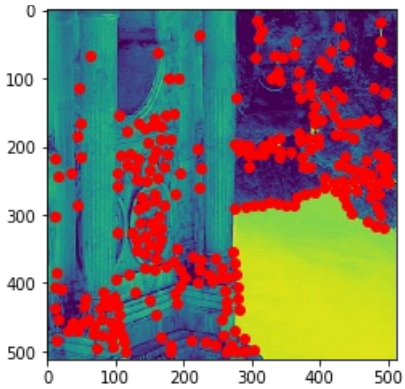

200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500



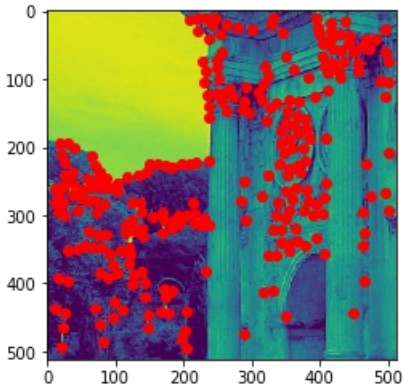
When testing for rotation invariance rotating to angles that didn't produce square images were not able to find points. This could be due to facts such as aliasing of the image. The program had no problem with square images such as at 90 and 180 degrees. The points found at 90 and 180 appear to be the exact same despite being upside down or on its side. To stitch these images that are at different angles would go beyond this program as homographic techniques to find how best to scale or wrap the image would be required

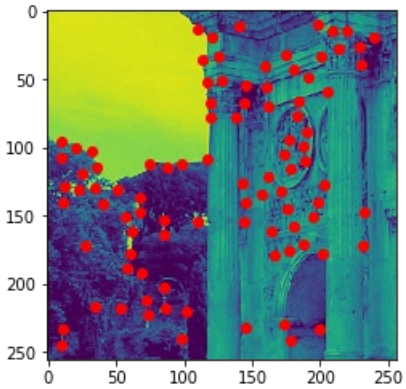
Testing for scale invariance

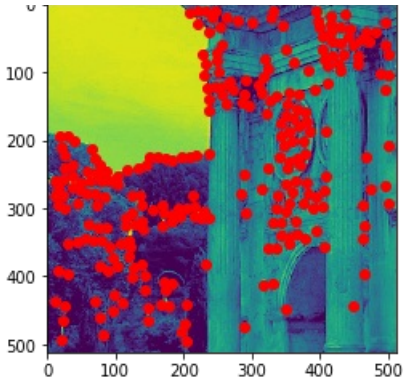
In [117]:

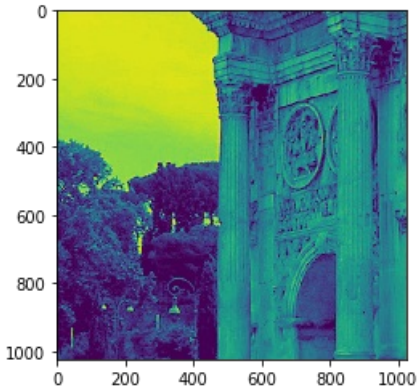
```
img1 = Image.open('arch1.png').convert('L')
larger_arch1 = np.array(img1.resize((int(img1.width*2), int(img1.height*2))))
smaller_arch1 = np.array(img1.resize((int(img1.width*0.5), int(img1.height*0.5))))

test_point_finding(arch1, smaller_arch1)
test_point_finding(arch1, larger_arch1)
```

```
/Users/ryan.jennings/.pyenv/versions/3.6.6/lib/python3.6/site-packages/ipykernel_launcher.py:19: R
untimeWarning: invalid value encountered in true_divide
/Users/ryan.jennings/.pyenv/versions/3.6.6/lib/python3.6/site-packages/ipykernel_launcher.py:7: Ru
ntimeWarning: invalid value encountered in greater
  import sys
```









When testing for scale invariance scaling to a smaller image produced less interest points which is understandable given there is less of the image to find suitable corners in. Unexpectedly the larger image returned no interest points, this could possibly be that the increased size was filled in with blank pixels which threw off the program's ability to find sharp corners