Project 2: Image Feature Detection

Caleb Ehrisman

Instructions

- Download the datasets from D2L that contain images of overlapping information from several locations.
- D2L: Submit your PDF writeup answering questions, illustrating methods, and ample examples of feature detection and matching along with the processes you used to generate them.
- Insert all code snippets within your document.

Problem Overview

Develop your own Python functions to compute image pyramids (you should have done this in project 1), extract features and edges, and correspond said features between successive image frames. The features that you detect and descriptors you develop should be at least minimally invariant to rotation and illumination changes. The project has been broken into different steps for your convenience.

Project Work

First step in the project was to create a function that would visualize the feature descriptor. So to do this I am drawing a red square with a line show the gradient similarly to the example image in the project description. I calculate the corner points of the square given a center point and an angle(theta) for orientation and then draw a line from the center to mid point of point 1 and 4. I am using openCV function line to draw the square. Some examples are shown below.

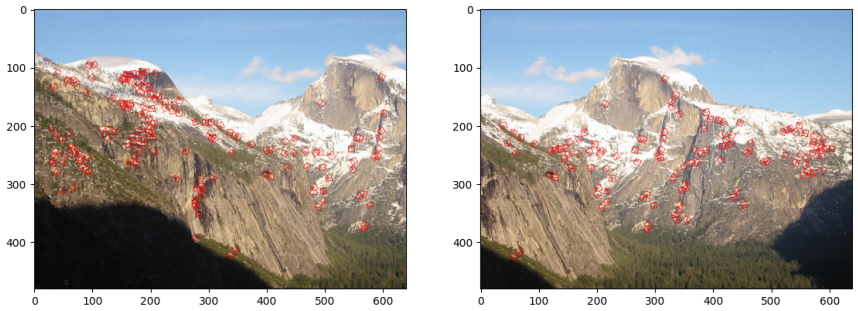


Figure 1: Yosemite with feature squares

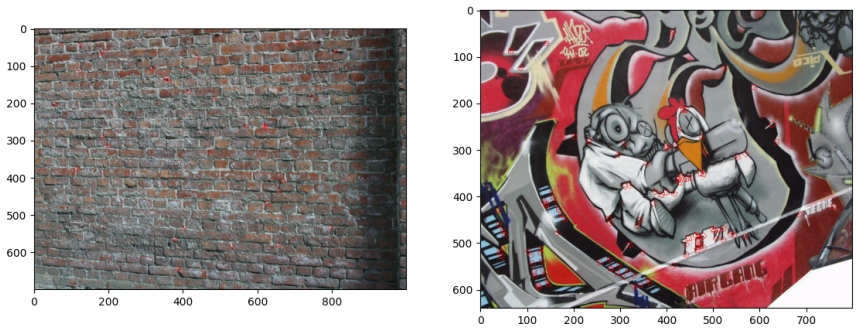


Figure 2: Graf and wall with feautre sqaures

Draws the features squares on the image.

```
#   Draws a feature square on a given image with the center
   point, scale and orientation angle given
def drawSquare(img, point, scale, theta):
    length = scale
    theta = theta * math.pi / 180.0

    p1 = (int(point[0] - 0.5 * length * (math.sin(theta) +
       math.cos(theta))),
    int(point[1] - 0.5 * length * (math.sin(theta) - math.
       cos(theta))))

    p2 = (int(point[0] + 0.5 * length * (math.sin(theta) -
       math.cos(theta))),
    int(point[1] - 0.5 * length * (math.sin(theta) + math.
       cos(theta))))

    p3 = (int(point[0] + 0.5 * length * (math.sin(theta) +
       math.cos(theta))),
    int(point[1] + 0.5 * length * (math.sin(theta) - math.
       cos(theta))))

    p4 = (int(point[0] - 0.5 * length * (math.sin(theta) -
       math.cos(theta))),
    int(point[1] + 0.5 * length * (math.sin(theta) + math.
       cos(theta))))


    cv2.line(img, p1, p2, (0, 0, 255), 1)
    cv2.line(img, p2, p3, (0, 0, 255), 1)
    cv2.line(img, p3, p4, (0, 0, 255), 1)
    cv2.line(img, p4, p1, (0, 0, 255), 1)

    midpoint = (int((p4[0] + p1[0]) / 2), int((p4[1] + p1
       [1]) / 2))
    y = int(point[1] + 10 * math.cos(theta))
    x = int(point[0] + 10 * math.sin(theta))
    # print([y, x])
    cv2.line(img, point, midpoint, (0, 0, 255), 1)

    return img
```

The second step for me was to actually identify and describe the features within the image. To do this I followed the steps given in the slides for the Harris Corner Detection as this is rotation and Illumination invariant. The reason why they are is that since you compute the gradient and get the orientation you can realign the images to determine similarity. And the magnitude is not affected by the brightness as the difference between the points would be the same ratio.

The steps are as followed

- Convert to gray scale
- Apply Gaussian Blur
- Apply Sobel Operator to compute derivatives
- For each pixel in image, compute the corner strength function for a 7x7 window.
- Normalize the values and then identify the local maxima of all points that exceed a user defined threshold.

The formula for the corner strength function is given as $det - k(trace^2)$

The code below implements the harris corner detection.

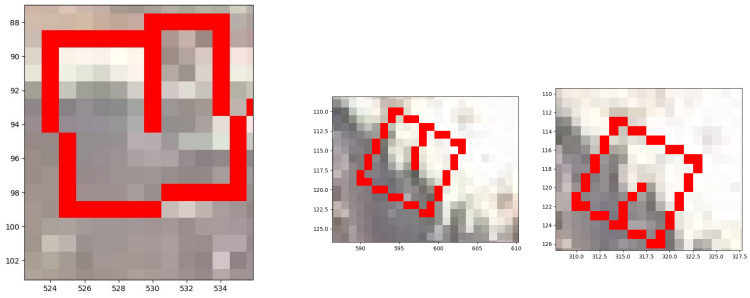
```
def harris(img, k, threshold):
    points = []
    height = img.shape[0]
    width = img.shape[1]
    offset = 3
    # Convert image to grayscale
    gray_img = rgb2gray(img)

    # plt.imshow(gray_img, cmap='gray')
    # plt.show()
    gray_img = signal.convolve2d(gray_img, gauss_mask)

    dx = signal.convolve2d(gray_img, sobel_x)
    dy = signal.convolve2d(gray_img, sobel_y)

    # plt.imshow(dxy)
    # plt.show()
    dx2 = np.square(dx)
    dy2 = np.square(dy)
    dxy = np.multiply(dx, dy)

```

```
    r_val_matrix = np.zeros((height, width))

    for r in range(offset, height - offset):
        for c in range(offset, width - offset):
            Sum_dx2 = np.sum(dx2[r - offset:r + offset + 1,
               c - offset:c + offset + 1])
            Sum_dy2 = np.sum(dy2[r - offset:r + offset + 1,
               c - offset:c + offset + 1])
            Sum_dxy = np.sum(dxy[r - offset:r + offset + 1,
               c - offset:c + offset + 1])

            det = (Sum_dx2 * Sum_dy2) - (Sum_dxy ** 2)
            trace = Sum_dxy * 2
            response = det - k * (trace ** 2)
            r_val_matrix[r][c] = response

    cv2.normalize(r_val_matrix, r_val_matrix, 1, 0, cv2.
       NORM_MINMAX)

    for y in range(int(height * 0.08), int(height - height *
        0.08)):
        for x in range(int(width * 0.08), int(width - width
           * 0.08)):
            if r_val_matrix[y, x] > threshold:
                vals = r_val_matrix[y - offset:y + 1 +
                   offset, x - offset:x + 1 + offset]

                if r_val_matrix[y, x] == np.max(vals):
                    points.append((x, y))

    return points
```

The next step was to define a descriptor for each feature point identified. I went with a basic one of computing the gradient magnitude and orientation for each point shown below. The figures may not seem accurate as it computed by the direct neighboring values from the chosen point.

The below code shows how the descriptors were made.

```
def computeGradient_Mag(img, pts):
    img = rgb2gray(img)
    grad_mag = np.zeros(len(pts))
    theta = np.zeros(len(pts))
```



```
5
6       for i in range(len(pts)):
7           x_gradient = img[pts[i][1], pts[i][0] + 1] - img[pts
                [i][1], pts[i][0] - 1]
8           y_gradient = img[pts[i][1] + 1, pts[i][0]] - img[pts
                [i][1] - 1, pts[i][0]]
9
10          grad_mag[i] = math.sqrt((x_gradient**2) + (
                y_gradient**2))
11          theta[i] = math.atan2(y_gradient, x_gradient)
12
13      return grad_mag, (theta * 180/math.pi)
```

The makeSqaure function is what is used to draw the square with the orientation line shown to allow the gradient to be seen.

```
1   #   Draws a feature square on a given image with the center
        point, scale and orientation angle given
2   def drawSquare(img, point, theta):
3       length = 10
4       theta = theta * math.pi / 180.0
5
6       p1 = (int(point[0] - 0.5 * length * (math.sin(theta) +
            math.cos(theta))),
7               int(point[1] - 0.5 * length * (math.sin(theta) -
                    math.cos(theta))))
8
9       p2 = (int(point[0] + 0.5 * length * (math.sin(theta) -
            math.cos(theta))),
10              int(point[1] - 0.5 * length * (math.sin(theta) +
                    math.cos(theta))))
11
12      p3 = (int(point[0] + 0.5 * length * (math.sin(theta) +
```

```
        math.cos(theta))),
13          int(point[1] + 0.5 * length * (math.sin(theta) -
                math.cos(theta))))
14
15      p4 = (int(point[0] - 0.5 * length * (math.sin(theta) -
            math.cos(theta))),
16          int(point[1] + 0.5 * length * (math.sin(theta) +
                math.cos(theta))))
17
18      # print(p1)
19      cv2.line(img, p1, p2, (255, 0, 0), 1)
20      cv2.line(img, p2, p3, (255, 0, 0), 1)
21      cv2.line(img, p3, p4, (255, 0, 0), 1)
22      cv2.line(img, p4, p1, (255, 0, 0), 1)
23
24      midpoint = (int((p2[0] + p3[0]) / 2), int((p4[1] + p1
            [1]) / 2))
25      y = int(point[1] + 10 * math.cos(theta))
26      x = int(point[0] + 10 * math.sin(theta))
27      # print([y, x])
28      cv2.line(img, point, midpoint, (255, 0, 0), 1)
29
30      return img
```

Then to match the features I have kept a list of points of the interest points gathered from the harris corner detector along with the computed orientation. I go to the given point in a image and slice a 18x18 section around that pixel, then rotates it negative the given orientation to set it to angle 0. Then this is repeated for the other image and given point. Then the two samples are stacked on each other and a sum of differences of each pixel value and the same index in the other sample are then squared to find the overall absolute difference between the two images given. And this process is done with all feature points in one image while comparing to all in another. If the the SSD does not exceed a threshold then it is not saved as it generally not a strong match.

Code below shows how the comparison is being done.

```
1  def match(img1, img2, pts1, pts2):
2      img1 = rgb2gray(img1)
3      img2 = rgb2gray(img2)
4      best_match = []
5      for i in range(len(pts1)):
6          curr_best = None
7          best_SSD = 9999999
8          for j in range(len(pts2)):
9              SSD = 0
10
11             # get 18x18 sample
12             sample1 = img1[pts1[i][1] - 9: pts1[i][1] + 10,
                   pts1[i][0] - 9: pts1[i][0]+10]
13             sample2 = img2[pts2[j][1] - 9: pts2[j][1] + 10,
                   pts2[j][0] - 9: pts2[j][0] + 10]
14
15             x_gradient = sample1[9, 10] - sample1[9, 8]
16             y_gradient = sample1[10, 9] - sample1[8, 9]
17
18             theta1 = math.atan2(y_gradient, x_gradient)
19
20             x_gradient = sample2[9, 10] - sample2[9, 10]
21             y_gradient = sample2[10, 9] - sample2[10, 9]
22
23             theta2 = math.atan2(y_gradient, x_gradient)
24
25             # rotate the sample negative the gradient to
                   align it to 0
26             rot1 = skimage.transform.rotate(sample1, -theta1
                   )
27             rot2 = skimage.transform.rotate(sample2, -theta2
```

```
                )

            # grab center 7x7 from rotated sample
            compare_samp1 = rot1[6:13, 6:13]
            compare_samp2 = rot2[6:13, 6:13]

            # Compute difference between all pixels and then
                square then sum
            diff = np.subtract(compare_samp1, compare_samp2)
            Square = np.square(diff)
            SSD = np.sum(Square)
            if SSD < .1:
                if SSD < best_SSD:
                    best_SSD = SSD
                    # print(best_SSD, SSD, (i, j))
                    curr_best = (i, j)

        best_match.append(curr_best)
    return best_match
```

Results

The Harris detector does correctly find corners in the image and the feature descriptors are shown on the image. The feature matching will match features correctly on translation most of the time e.g. Fig 1. But I did notice some issues that I was not able to find the root cause of. First being that when I calculate the points of the corner and then proceed to draw them the Squares seem to be off by a small margin. The problem persisted if I used a plain circle based on the point e.g. if I had a corner computed at (55, 95) and then the center of the square would draw at 59, 99 which can make a big difference since the window is only 18x18. Otherwise the matching and corner calculations are not off by that same margin.

However, it does OK at best for the change in perspective. It will correctly match some of the features but then also have some that are mismatched. This is shown in the wall image in figure 2 and in the graffiti images Fig 3. and Fig 4.

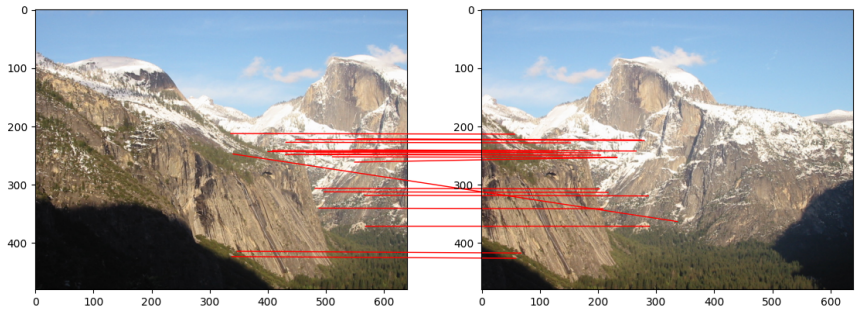


Figure 3: Example of matching some features on Yosemite.

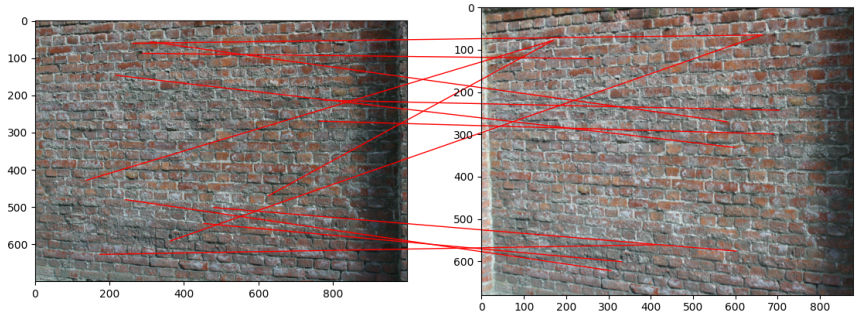


Figure 4: Example of matching some features on the wall.

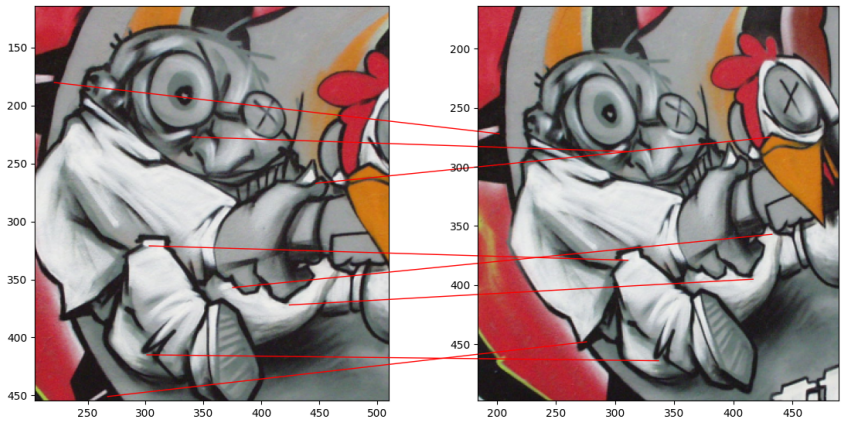
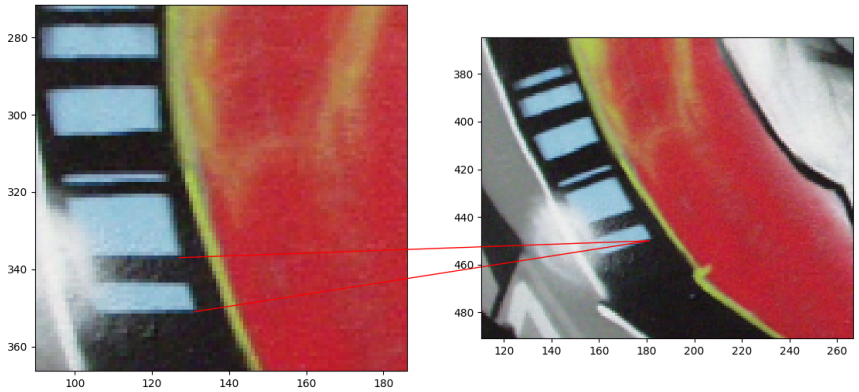


Figure 5: Example of matching some features on the graffiti.



]

Figure 6: Caption