Homework 4 Writeup

Code Explanation: get_interest_points

First, if image is color or 3D, it is converted to black and white 2D.

```
if image.ndim == 3 and image.shape[2] == 1:
  image = image[:,:,0]
elif image.ndim == 3 and image.shape[2] == 3:
  image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
```

The following is the process of finding partial derivatives.

```
H = image.shape[0]
W = image.shape[1]

Ix = np.zeros((H, W))
Iy = np.zeros((H, W))

for x in range(W-1):
  Ix[:,x] = image[:, x+1] - image[:, x]

for y in range(H-1):
  Iy[y,:] = image[y+1, :] - image[y, :]
```

The following is the process of finding second derivatives.

```
Ixx = Ix * Ix
Iyy = Iy * Iy
Ixy = Ix * Iy
```

Ones created above are blurred with Gaussian filter.

```
gIxx = cv2.GaussianBlur(Ixx, (descriptor_window_image_width-1,
    descriptor_window_image_width-1), 0)
gIyy = cv2.GaussianBlur(Iyy, (descriptor_window_image_width-1,
    descriptor_window_image_width-1), 0)
gIxy = cv2.GaussianBlur(Ixy, (descriptor_window_image_width-1,
    descriptor_window_image_width-1), 0)
```

C is calculated.

```
C = gIxx * gIyy - gIxy**2.0 - 0.04 * (gIxx + gIyy)**2.0
```

The case where C is greater than 0.000005 is selected. Clumping is eliminated by selecting the one with the largest C among closely clustered ones. And it returns x, y.

```
x = []
y = []

for i in range(int(W/descriptor_window_image_width)):
  for j in range(int(H/descriptor_window_image_width)):
    index =
       np.unravel_index(np.argmax(C[j*descriptor_window_image_width:(j+1)*descrip
       i*descriptor_window_image_width:(i+1)*descriptor_window_image_width],
       axis=None), (descriptor_window_image_width,
       descriptor_window_image_width))
    index_x = i*descriptor_window_image_width+index[1]
    index_y = j*descriptor_window_image_width+index[0]
    max_c = C[index_y, index_x]
    if max_c > 0.000005:
      x.append(index_x)
      y.append(index_y)

x = np.array(x)
y = np.array(y)
```

Code Explanation: get_descriptors

If image is color or 3D, it is converted to black and white 2D.

```
if image.ndim == 3 and image.shape[2] == 1:
  image = image[:,:,0]
elif image.ndim == 3 and image.shape[2] == 3:
  image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
```

Frequently used variables are defined. At this time, zero padding is applied to image to work well in edges and corners.

```
half_w = int(descriptor_window_image_width/2.0)
quat_w = int(descriptor_window_image_width/4.0)
image = np.pad(image, ((half_w,half_w),(half_w,half_w)),
    'constant', constant_values=0)
features = np.zeros((x.shape[0], 128))
```

For loop is used to calculate a point obtained from get_interest_points function. First, define some variables and create local area(local_img) around the point. And calculate derivative, magnitude and direction in it.

```
for i in range(len(x)):
  index_x = int(x[i] + half_w)
  index_y = int(y[i] + half_w)
```

```
  f = []

  local_img = image[index_y-half_w: index_y+(half_w+1),
      index_x-half_w: index_x+(half_w+1)]
  Ix = local_img[:descriptor_window_image_width, 1:] -
      local_img[:descriptor_window_image_width,
      :descriptor_window_image_width]
  Iy = local_img[1:, :descriptor_window_image_width] -
      local_img[:descriptor_window_image_width,
      :descriptor_window_image_width]
  mag_grad = (Ix**2 + Iy**2)**0.5
  theta_grad = np.arctan2(Iy, Ix)
```

Each area is made by dividing local area into 16 equal parts.

```
for m in range(4):
  for n in range(4):
    local_theta = theta_grad[quat_w*m:quat_w*(m+1),
        quat_w*n:quat_w*(n+1)]
    local_mag = mag_grad[quat_w*m:quat_w*(m+1),
        quat_w*n:quat_w*(n+1)]
```

Histogram is created within created area and included in feature.

```
hist = np.zeros(8)
for j in range(quat_w):
  for k in range(quat_w):
    if (local_theta[j,k] > 0) and (local_theta[j,k] <=
        np.pi/4):
      hist[0] = hist[0] + local_mag[j,k]
    elif (local_theta[j,k] > np.pi/4) and (local_theta[j,k]
        <= np.pi/2):
      hist[1] = hist[1] + local_mag[j,k]
    elif (local_theta[j,k] > np.pi/2) and (local_theta[j,k]
        <= np.pi/4*3):
      hist[2] = hist[2] + local_mag[j,k]
    elif (local_theta[j,k] > np.pi/4*3) and (local_theta[j,k]
        <= np.pi):
      hist[3] = hist[3] + local_mag[j,k]
    elif (local_theta[j,k] > -1*np.pi) and (local_theta[j,k]
        <= -1*np.pi/4*3):
      hist[4] = hist[4] + local_mag[j,k]
    elif (local_theta[j,k] > -1*np.pi/4*3) and
        (local_theta[j,k] <= -1*np.pi/2):
      hist[5] = hist[5] + local_mag[j,k]
    elif (local_theta[j,k] > -1*np.pi/2) and
        (local_theta[j,k] <= -1*np.pi/4):
      hist[6] = hist[6] + local_mag[j,k]
    elif (local_theta[j,k] > -1*np.pi/4) and
        (local_theta[j,k] <= 0):
      hist[7] = hist[7] + local_mag[j,k]
```

```
f = f + list(hist)
```

Features are normalized, made 0.2 that are greater than 0.2, and renormalized. And it returns features.

```
f = np.array(f)
f = f/np.linalg.norm(f)
f[f > 0.2] = 0.2
f = f/np.linalg.norm(f)
features[i,:] = f
```

Code Explanation: match_features

First, declare variables.

```
matches = []
confidences = []
d = {}
```

Find two points (first and second minimum distance points) that correspond to feature in features1.

```
for i in range(features1.shape[0]):
  sec_dis = float("inf")
  min_dis = float("inf")

  for j in range(features2.shape[0]):
    distance = np.linalg.norm(features1[i]-features2[j])
    if min_dis >= distance:
      sec_dis = min_dis
      min_dis = distance
      min_j = j
    elif sec_dis >= distance:
      sec_dis = distance
```

Find distance ratio and filter points less than 0.85.

```
ratio = min_dis/sec_dis
if ratio <= 0.85:
  d[(i, min_j)] = 1 - ratio
```

Do the same as above for feature2.

```
for i in range(features2.shape[0]):
  sec_dis = float("inf")
```

```
  min_dis = float("inf")

  for j in range(features1.shape[0]):
    distance = np.linalg.norm(features1[j]-features2[i])
    if min_dis >= distance:
      sec_dis = min_dis
      min_dis = distance
      min_j = j
    elif sec_dis >= distance:
      sec_dis = distance

  ratio = min_dis/sec_dis
  if ratio <= 0.85:
    d[(min_j, i)] = 1 - ratio
```

d ditionary(Corresponding points and confidences) is divided into variables matches and confidences, respectively. And it returns matches and confidences

```
for key, value in d.items():
  matches.append(key)
  confidences.append(value)

matches = np.array(matches)
confidences = np.array(confidences)
```

Result: NotreDame

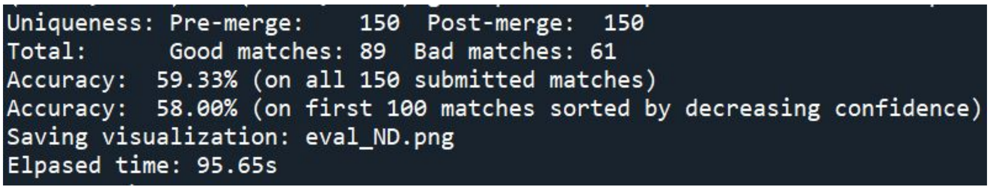
```
Uniqueness: Pre-merge:    150  Post-merge:  150
Total:      Good matches: 89  Bad matches: 61
Accuracy:  59.33% (on all 150 submitted matches)
Accuracy:  58.00% (on first 100 matches sorted by decreasing confidence)
Saving visualization: eval_ND.png
Elapsed time: 95.65s
```

Figure 1: NotreDame Accuracy and Elapsed time

The accuracy on first 100 matches sorted by decreasing confidence is 58% and elapsed time is 95.65s.

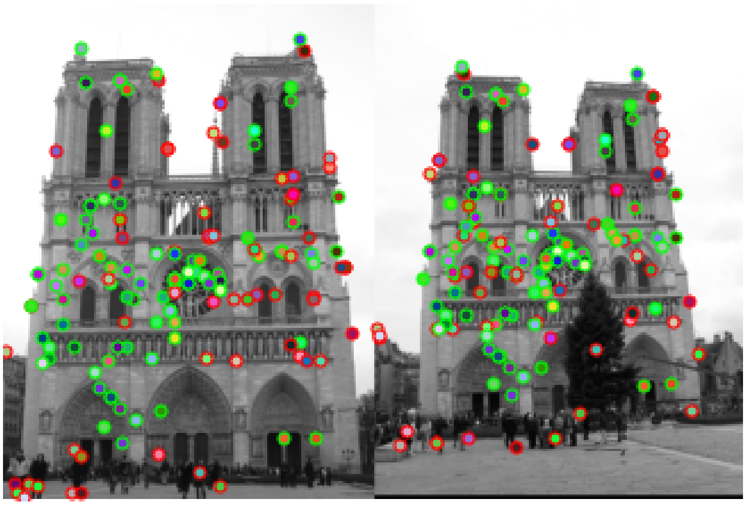

Figure 2: NotreDame Corresponding points

Result: MountRushmore

```
Uniqueness: Pre-merge:    175  Post-merge:  174
Total:      Good matches: 133  Bad matches: 41
Accuracy:  76.44% (on all 174 submitted matches)
Accuracy:  86.00% (on first 100 matches sorted by decreasing confidence)
Saving visualization: eval_MR.png
Elapsed time: 131.85s
```

Figure 3: MountRushmore Accuracy and Elapsed time

The accuracy on first 100 matches sorted by decreasing confidence is 86% and elapsed time is 131.85s.

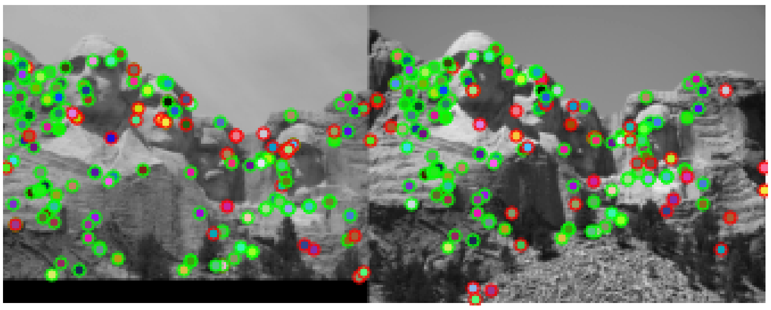

Figure 4: MountRushmore Corresponding points

Result: EpiscopalGaudi

```
Uniqueness: Pre-merge:    30  Post-merge:  30
Total:      Good matches: 5  Bad matches: 25
Accuracy:   16.67% (on all 30 submitted matches)
Accuracy:   5.00% (on first 100 matches sorted by decreasing confidence)
Saving visualization: eval_EG.png
Elapsed time: 86.90s
```

Figure 5: EpiscopalGaudi Accuracy and Elapsed time

The accuracy on first 100 matches sorted by decreasing confidence is 5% and elapsed time is 86.90s.

Figure 6: EpiscopalGaudi Corresponding points