# Project 2 Writeup

## Project Overview

In this project, feature matching is implemented in the following steps:

- Implement function `get_features()` to get SIFT like features
- Implement function `match_features()` for matching features
- Implement function `get_interest_points()` to get interest points using Harris cornes detection.

## Implementation Detail

The implementation order is follwing the implementation strategy part on project website. The accuracy results will be discussed in the Results section.

### Explaination for implementation of get_features()

At first, an error message will be raised if the feature width is not a multiple of 4. And then to improve the accuracy, especially for EpiscopalGaudi case, the image is rescaled if the input image is too small. The corresponding interest points are also rescaled to the same factor. The same operation has also been done in `get_interest_points()` function. This one operation can increase accuracy on EpiscopalGaudi from 10% to 80%.

For each interested points, a box with the shape of [feature_width, feature_width] around the interest point is cut out for feature construction. Interested ponts that are close to the image edge will be skipped. The window is then splitted into cells with the shape of [4, 4]. Each cell will be reprensented by a one dimension feature with length 8, the value of which is calculated by summing up magnitude of the gradients of pixels in specific orientation. Then one feature for the whole window is obtained by concatenating all cell features, resulting in a one dimension vector with length as 128 for one interest point. The feature vector is normalized and clipped by a threshold and then renormalized before collected by `features`.

Two different descriptors are also implemented and their implementation and performance are discussed in the Extra Credits section.

### Explaination for implementation for match_features()

The feature vectors for interest points from two images are compared similarity by calculating Euclidean distances among all pairs. The shortest the distance is, the more similar the two points are. The confidence value is defined as 1 - nddr. To improve the accuracy, the matches has been performed from image2 to image1 as well as from image1 to image2. And the common pairs are extracted for the final matches. This can increase accuracy from 80% to 90% or so. PCA analysis has been implemented for test purpose and are discussed in the Extra Credit section.

### Explaination for implementation for get_interest_points()

At first, the image is also rescaled the same as it stated in `get_features()` function. However, the returned x and y coordinates are rescaled back to the original input shape to avoid conflicting the `main.py`. Then, the image is filtered with gaussian filter for better detection. The gradients for each pixels are calculated using `np.gradient()`. And the $R$ values are calculated using $R = det(M) - \alpha(trace(M))$, where $\alpha = 0.04$. The $R$ is then normalized to unit length and clipped the smallest 15% value to 0. Interest points are picked using `feature.peak_local_max()` function, which pick the largest value among a window size of `feature_width//2` * `feature_width//2`.

## Result

Results for NotreDame, MountRushmore and EpiscopalGaudi pairs are presented below. The program is run in local with the same python environment on the department machine. The results are obtained with the optimal hyperparameters, SIFT-like descriptor and rescaling operation for images that are too small. The However, I found that there might be 1% accuracy difference from the results on Gradescope.

1. NotreDame:
    - Matches: 380
    - Accuracy on 50 most confident: 100%
    - Accuracy on 100 most confident: 100%
    - Accuracy on all matches: 68%
2. MountRushmore:
    - Matches: 737
    - Accuracy on 50 most confident: 100%
    - Accuracy on 100 most confident: 100%
    - Accuracy on all matches: 56%

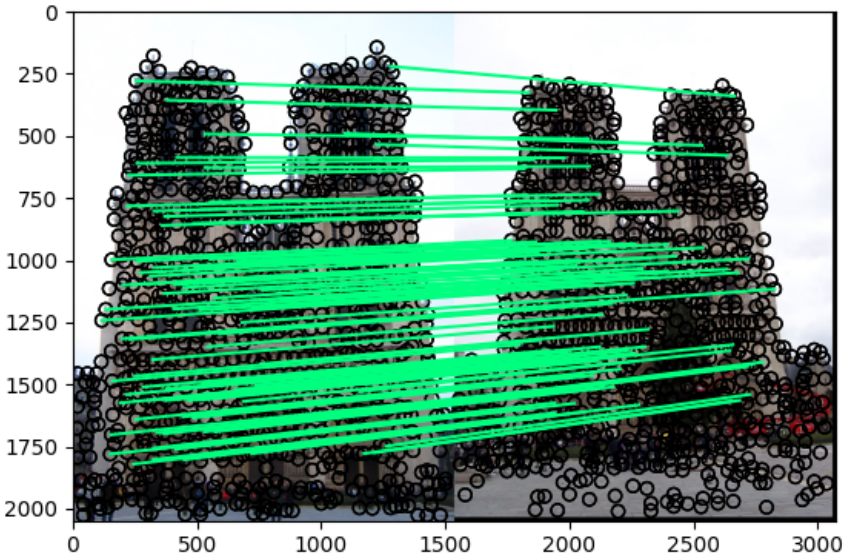


Figure 1: Mathces on 100 most confident pairs of NotreDame

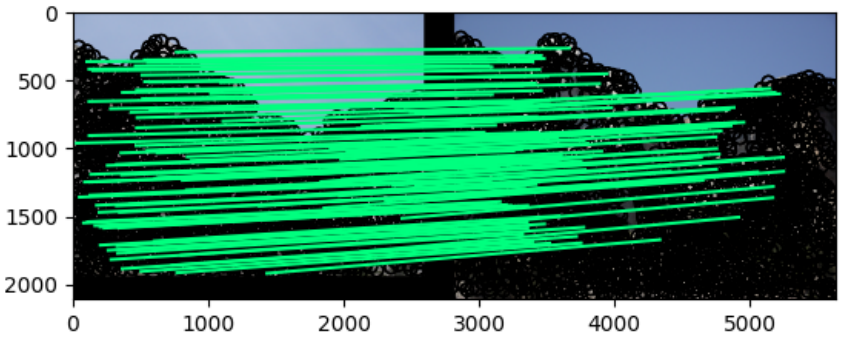


Figure 2: Mathces on 100 most confident pairs of MountRushmore

3. EpiscopalGaudi:
    - Matches: 495
    - Accuracy on 50 most confident: 94%
    - Accuracy on 100 most confident: 88%
    - Accuracy on all matches: 44%

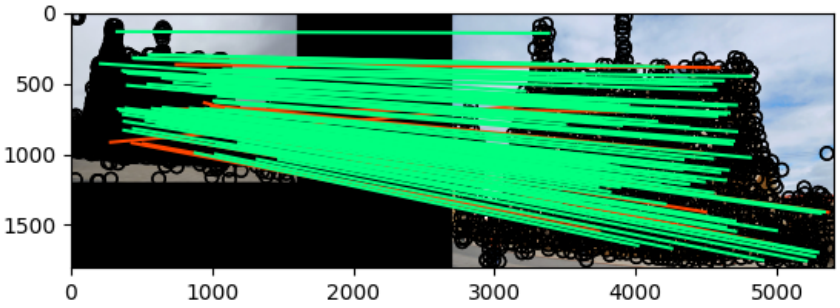


Figure 3: Mathces on 100 most confident pairs of EpiscopalGaudi

## Extra Credit (Optional)

1. **Detection**

    Adaptive non-maximum suppression (ANMS) implementation The ANMS is implemented in the following manner: distances between the point $A$ and the nearest point of which 90% R value is larger than the point $A$ are calculated and sorted. Only the first 500 points with the largest distance value are extracted as interest points. In this way, we could suppress points that close the the peak we pick but with small R value.

    The results with ANMS implementation are presented below. Compared to the results in Result section, we found the accuracy for all three cases decreases. The decrease here might be the number of points we pick here is only 500 while before

ANMS, the number of points varies from 700 to 2000. To sum up, this method might be useful in some case but it doesn't show better performance here.

- NotreDame:
  - Matches: 206
  - Accuracy on 50 most confident: 100%
  - Accuracy on 100 most confident: 93%
  - Accuracy on all matches: 57%
- MountRushmore:
  - Matches: 211
  - Accuracy on 50 most confident: 96%
  - Accuracy on 100 most confident: 65%
  - Accuracy on all matches: 36%
- MountRushmore:
  - Matches: 165
  - Accuracy on 50 most confident: 68%
  - Accuracy on 100 most confident: 40%
  - Accuracy on all matches: 26%

Here is the corresponding part code in the implementation.

```
def anms(points, R, top=500):
    l, x, y = [], 0, 0
    threshold = np.mean(R)
    while x < len(points):
        minpoint = float("inf")
        xi, yi = points[x][0], points[x][1]
        while y < len(points):
            xj, yj = points[y][0], points[y][1]
            if (
                xi != xj
                and yi != yj
                and R[points[x][0], points[x][1]] > threshold
                and R[points[y][0], points[y][1]] > threshold
                and R[points[x][0], points[x][1]]
                < R[points[y][0], points[y][1]] * 0.9
            ):
                dist = math.sqrt((xj - xi) ** 2 + (yj - yi) ** 2)
                if dist < minpoint:
                    minpoint = dist
            y += 1
        l.append([xi, yi, minpoint])
        x += 1
        y = 0
    l.sort(key=lambda x: x[2], reverse=True)
    return np.array(l[:top])
```

2. **Description**:

   - It have been tested with **different number of bins**. The accuracy with different number of bins is presented below. (The accuracy on 100 most confident matches for NotreDame is run in local, which might have less than 1 % difference from results on Gradescope. All other parameters are values shown in code.) From the table, we could see that tt can harm the accuracy if the number of bins is too small but won't change the accuracy too much if the number is enough. However, the excution time will increase with the increase of the number of bins. Thus, 8 is suitable for descriptor construction considering both accuracy and excution time.

| Number of bins | 4 | 6 | 8 | 10 | 12 | 30 |
|---|---|---|---|---|---|---|
| Accuracy | 94 | 97 | 100 | 99 | 99 | 99 |
| Excution Time (s) | 5.58 | 5.91 | 6.13 | 6.74 | 8.30 | 14.93 |

Table 1: Accuracy changes w.r.t the number of bins on NotreDame case. (Excution time is measured in local.)

   - **Different spatial layouts for feature**: There are two other types of descriptors have been implemented. The type can be chosen through one parameter `descriptor_type` in `get_features()` function.
     - **HOG-like descriptor**: the main idea is that the gradient closer to the center orientation of the corresponding bins will contribute more to the bin value. Thus, the contribution is calculated using

$$value = \sum_i magnitude_i * \frac{orientation_i - center}{bin\_width}$$

       However, though the construction of feature has taken the orientation of gradient into consideration, the accuracy on 100 most confident matches for NotreDame decreases from 100% to 99%. And for MountRushmore is the same. However, for EpiscopalGaudi, the accuracy increase from 88% to 90%.
       Here is the corresponding part code in the implementation.

```
1  for subwindow_i in range(len(patch_window_magnitudes)):
2      inds = np.digitize(patch_window_orientations[
                                          subwindow_i],
                                          bins)
3      for inds_i in range(num_bins):
4          mask = np.array(inds == inds_i)
5          feature[subwindow_i * num_bins + inds_i] = np.
                                              dot(
6          patch_window_magnitudes[subwindow_i].flatten()[
                                              mask],
7          np.cos(
8              np.abs(
9              patch_window_orientations[subwindow_i].
                                                  flatten
                                                  ()[mask]
```

```
                    - (bins[inds_i] + bin_width / 2)
                )
            ),
        )
```

– **GLOH-like descriptor**: borrows the idea from the GLOH descriptor. In the cell, pixels are firstly grouped based upon their orientation and then grouped pixels are grouped based upon their magnitude again, which results in 24 bins for one cell. Thus, the total feature length is 384. This is GLOH-like descriptor as I have not fully understand how to implement the true GLOH descriptor. This descriptor also increase the accuracy on EpiscopalGaudi to 90%.
Here is the corresponding part code in the implementation.

```
mag_bin = np.array([0,2,4])
for subwindow_i in range(len(patch_window_magnitudes)):
    inds = np.digitize(patch_window_orientations[
                                        subwindow_i],
                                        bins)
    for inds_i in range(num_bins):
        mask = np.array(inds == inds_i)
        sub_inds = np.digitize(
        patch_window_magnitudes[subwindow_i][mask],
                                            mag_bins
            )
        for sub_inds_i in range(num_mag_bins):
            sub_mask = np.array(sub_inds == sub_inds_i)
            feature.append(
            np.dot(
            patch_window_magnitudes[subwindow_i].flatten
                                                ()[mask][
                            sub_mask],
            np.cos(
            np.abs(
            patch_window_orientations[subwindow_i].
                                                flatten()
                                                [
                    mask][
                            sub_mask]
                    - (bins[inds_i] + bin_width / 2)
                            )
                        ),
                    )
                )
```

3. **Matching**: Principle component analysis (PCA) implementation

   The PCA method has been implemented as following. The idea of PCA is analyzing the weight of each feature component and project it onto those dimensions with larger wights. In default of current implementation, the first 32 components are extracted for matching. However, the accuracy decreases to 0 - 2%. (I have checked with `sklearn.decomposition.PCA()`, which shows the same result.)

```
def PCA(features, m=32):
    C = features - np.mean(features, axis=0)
    cov_matrix = np.cov(C.T)
    eigenvals, eigenvecs = np.linalg.eig(cov_matrix)
    i = np.argsort(-1 * np.abs(eigenvals))
    eigenvecs = eigenvecs[i]
    eigenvals = eigenvals[i]
    pca_features = np.dot(eigenvecs.T, C.T)
    return pca_features[:m].T
```