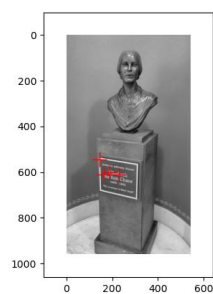


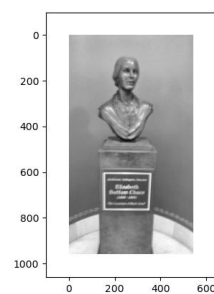
1.1

(a)

结果如下:

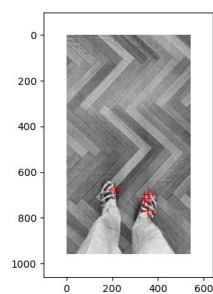


Chase1

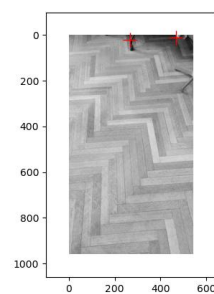


Chase2

图 1: Chase 组

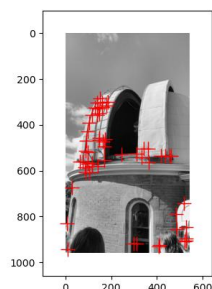


RISHLibrary1

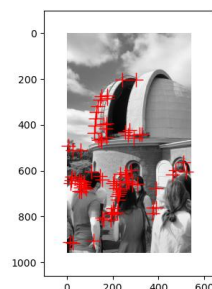


RISHLibrary2

图 2: RISHLibrary



LaddObservatory1



LaddObservatory2

图 3: LaddObservatory 组

(b)

对于 Chase 组, Chase1 找到了比较明显的几个 corners, Chase2 则没有找到, 从原图中可以看到 Chase2 比 Chase1 模糊了很多。

对于 RISHLibrary 组, RISHLibrary1 找到的是鞋子的 corners, RISHLibrary2 找到的是桌角的 corners, 他们都是原图中比较显眼的地方。同时两张原图的光线存在差异。

对于 LaddObservatory 组, 在建筑物上两个结果相差不是特别大, 对于 LaddObservatory2 中的人更多识别到的关于人的 corners 也更多。两张原图应该是不同视角拍摄下来的图片。

从三个对照组来看, 我认为匹配图像特征存在的挑战有:

1. 视角和尺度变化
2. 遮挡
3. 光线变化
4. 噪声问题

1.2

(a)

1. **欧氏距离**是指两个向量之间的直线距离，即它们在空间中的几何距离。在二维平面上，欧氏距离可以被视为连接两个点的直线的长度。在更高维度的空间中，欧氏距离可以看作是两个向量之间的“直线”距离。它主要衡量了向量之间的“接近程度”，距离越短表示向量越相似。当关注向量的绝对值大小和空间位置时，欧氏距离是更合适的选择。例如，在图像处理中，如果需要比较两幅图像的像素值之间的相似性，可以使用欧氏距离来衡量它们之间的距离。
2. **余弦相似度**是指两个向量之间的夹角的余弦值，它衡量了向量之间的方向相似度而不考虑它们的大小。如果两个向量的方向越接近，余弦相似度就越接近于 1；如果它们的方向正交或者相反，余弦相似度就越接近于 0。余弦相似度衡量了向量之间的“方向相似度”，而不受向量大小的影响。当关注向量之间的方向相似度而不考虑它们的绝对大小时，余弦相似度更适合。例如，在自然语言处理中，可以使用余弦相似度来比较文档之间的相似性，因为文档的长度可能不同，但它们的主题方向可能是相似的。

(b)

我认为给定距离度量方式后，一个比较好的特征描述符匹配方法是 nearest neighbor 算法。因为他比较简单，直观且易于理解，在欧氏距离和余弦相似度两种距离度量上都比较适用，还可以通过 KD-Tree 和 Ball-Tree 等数据结构来加速大规模数据集的匹配。

1.3

(a)

(b)

(c)

当直线平行于投影平面或者直线位于投影平面内的时候。

2.3

(a)

实现如下:

```
1     def get_interest_points(image, feature_width):
2
3     # 设置参数
4     sigma = 1.0
5     k = 0.06
6     threshold = 0.1
7
8     # 从main.py中可以知道传入的图片都是灰度图，可以直接
        计算Ix, Iy,求得海森矩阵
9     Ix = filters.sobel_v(image=image)
10    Iy = filters.sobel_h(image=image)
11
12    Ixx = Ix**2
13    Iyy = Iy**2
14    Ixy = Ix * Iy
15
16    # 将高斯滤波器用于上面算子
17    Ixx = filters.gaussian(Ixx, sigma=sigma)
18    Iyy = filters.gaussian(Iyy, sigma=sigma)
19    Ixy = filters.gaussian(Ixy, sigma=sigma)
20
21    # 计算每个像素的  $R = \det(H) - k(\text{trace}(H))^2$ 。 $\det(H)$ 
        表示矩阵H的行列式， $\text{trace}$ 表示矩阵H的迹。通常k的
        取值范围为[0.04,0.16]。
22    detH = (Ixx * Iyy) - (Ixy ** 2)
23    traceH = Ixx + Iyy
24    R = detH - k * ((traceH) ** 2)
25
26    # 满足  $R \geq \max \times th$  的像素点即为角点。th常取0.1，
```

```

    不知道peak_local_max的threshold_rel是不是就是在
    做这个
27  # 参数中的feature_width好像是没用的，又或者说他与
    min_distance有关？我感觉它只是在get_features中
    会用到
28  points = feature.peak_local_max(R, min_distance=
    feature_width, threshold_rel=threshold)
29
30  xs = points[:, 1]
31  ys = points[:, 0]
32
33  return xs, ys

```

(b)

实现如下:

normalized patches:

```

1  def get_features(image, x, y, feature_width):
2  h = image.shape[0]
3  w = image.shape[1]
4
5  offset = feature_width // 2
6  num_points = x.shape[0]
7
8  # 计算索引范围
9  x_start = x - offset
10 x_stop = x + offset
11 y_start = y - offset
12 y_stop = y + offset
13
14 # Compute padding parameters
15 x_min = x_start.min()
16 x_max = x_stop.max()

```

```

17     y_min = y_start.min()
18     y_max = y_stop.max()
19
20     x_pad = [0, 0]
21     y_pad = [0, 0]
22     if x_min < 0:
23         x_pad[0] = -x_min
24     if y_min < 0:
25         y_pad[0] = -y_min
26     if x_max - w >= 0:
27         x_pad[1] = x_max - w + 1
28     if y_max - h >= 0:
29         y_pad[1] = y_max - h + 1
30
31     x_start += x_pad[0]
32     x_stop += x_pad[0]
33     y_start += y_pad[0]
34     y_stop += y_pad[0]
35
36     # 对图片进行padding
37     image = np.pad(image, [y_pad, x_pad], mode="
        constant")
38
39     # 对每个维度窗口下建立索引
40     cell_size = 4
41     num_blocks = feature_width // cell_size
42
43     x_idx = np.array([np.arange(start, stop) for start
        , stop in zip(x_start, x_stop)])
44     y_idx = np.array([np.arange(start, stop) for start
        , stop in zip(y_start, y_stop)])
45
46     # 转置之前 : (num_blocks, num_points, cell_size)

```

```

47 x_idx = np.array(np.split(x_idx, num_blocks, axis
48                             =1))
49 y_idx = np.array(np.split(y_idx, num_blocks, axis
50                             =1))
51
52 # 转置之后: (num_points, num_blocks, cell_size)
53 x_idx = x_idx.transpose([1, 0, 2])
54 y_idx = y_idx.transpose([1, 0, 2])
55
56 # 对窗口中的每个像素建立索引
57 x_idx = np.tile(np.tile(x_idx, cell_size), [1,
58                             num_blocks, 1]).flatten()
59 y_idx = np.tile(np.repeat(y_idx, cell_size, axis
60                             =2), num_blocks).flatten()
61
62 # 计算偏导数
63 partial_x = filters.sobel_h(image)
64 partial_y = filters.sobel_v(image)
65 # 计算梯度模长
66 magnitude = np.sqrt(partial_x * partial_x +
67                       partial_y * partial_y)
68 # 计算梯度方向
69 orientation = np.arctan2(partial_y, partial_x) +
70                   np.pi
71 # 梯度近似为最近的角度
72 orientation = np.mod(np.round(orientation / (2.0 *
73                                   np.pi) * 8.0), 8)
74 orientation = orientation.astype(np.int32)
75 # 对梯度做高斯平滑
76 magnitude = filters.gaussian(magnitude, sigma=
77                               offset)
78
79 # 所有的块数据转为1维

```

```

72     magnitude_in_pixels = magnitude[y_idx, x_idx]
73     orientation_in_pixels = orientation[y_idx, x_idx]
74
75     # 转为数组 (num_patches, cell_size, cell_size)
76     magnitude_in_cells = magnitude_in_pixels.reshape
77         ((-1, cell_size * cell_size))
78     orientation_in_cells = orientation_in_pixels.
79         reshape((-1, cell_size * cell_size))
80
81     # 对每个cel计算梯度方向加权和
82     features = np.array(list(
83         map(lambda array, weight: np.bincount(array,
84             weight, minlength=8), orientation_in_cells,
85             magnitude_in_cells)))
86
87     # 每一行都是角点对应的一个特征向量
88     features = features.reshape((num_points, -1))
89     features = features / np.linalg.norm(features,
90         axis=-1).reshape((-1, 1))
91     features[features >= 0.2] = 0.2
92     features = features / np.linalg.norm(features,
93         axis=-1).reshape((-1, 1))
94
95     return features

```

SIFT:

(c)

实现如下:

```

1     def match_features(im1_features, im2_features):
2

```



```

3      assert im1_features.shape[1] == im2_features.shape
        [1]
4      # 设置 confidence
5      confidence = 1
6
7      matches = []
8      confidences = []
9
10     for i in range(im1_features.shape[0]):
11         distances = np.sqrt(
12             ((im1_features[i, :] - im2_features) ** 2)
13             .sum(axis=1))
14         indexes = np.argsort(distances)
15         min_distance = distances[indexes[0]]
16         second_min_distance = distances[indexes[1]]
17
18         ratio = min_distance / second_min_distance if
            second_min_distance != 0 else 0
19         if ratio < confidence:
20             matches.append([i, indexes[0]])
21             confidences.append(1 - ratio)
22
23     matches = np.asarray(matches)
24     confidences = np.asarray(confidences)
25
26     return matches, confidences

```

(d)

(i)

我在 main.py 中看到传入的图像已经做了转化为灰度图的处理，所以我在实现时没有考虑彩色图。实现时主要就是按照 harris corner, SIFT, Nearest neighbor 算法的描述用 numpy 和 skimage 提供的 api 进行实现。

实现 Nearest neighbor 时 comment 提到要设置一个 confidence，我设置为 0.9。

(ii)

使用 cheat_interest_points 时, 实现 get_features 两种方案的结果:

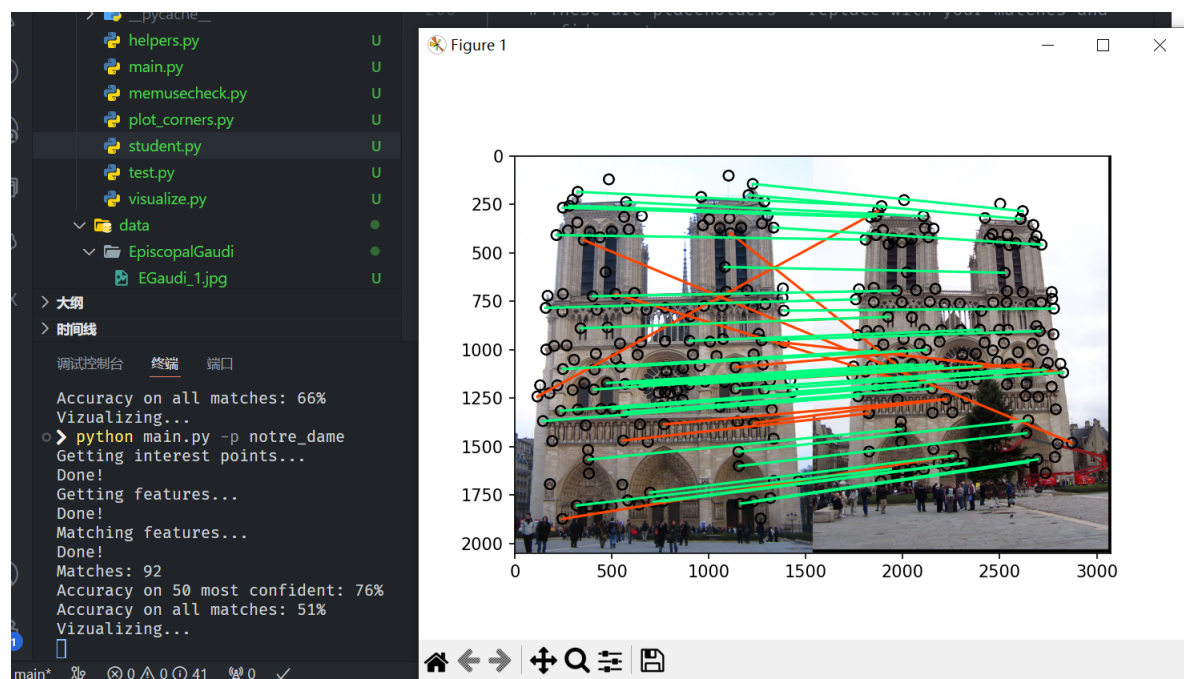


图 4: normalized patches 结果

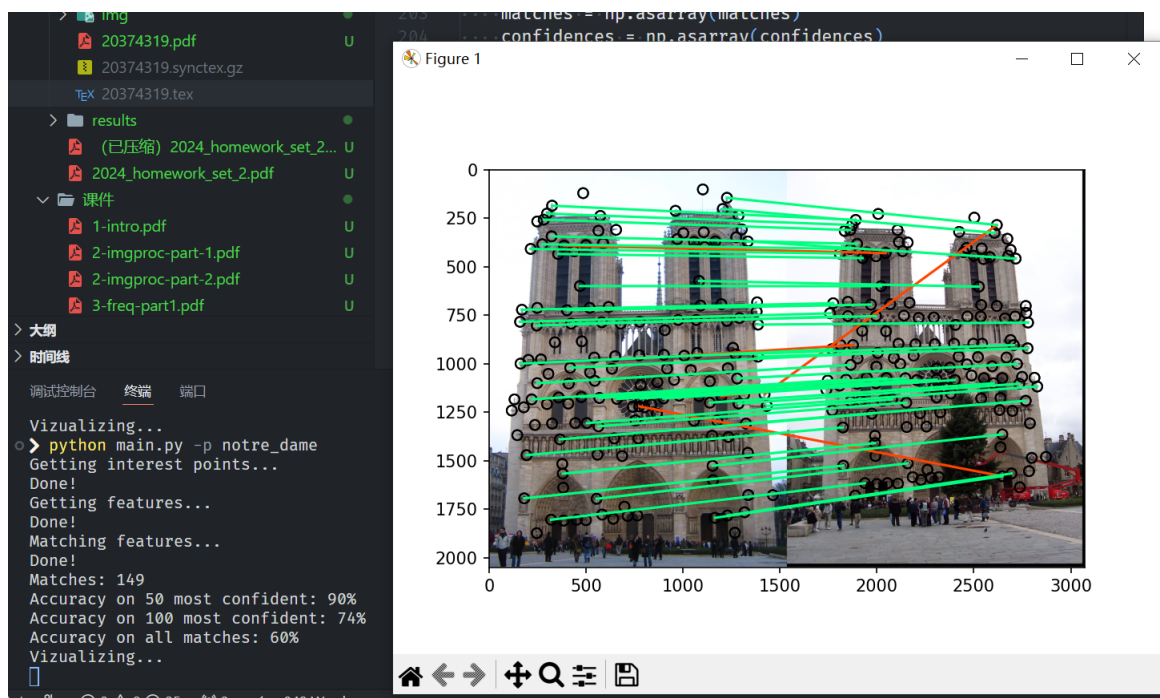


图 5: sift 结果

从图片中可以看到，使用 sift 相比于 normalized patches 大概能提升 10%。

(iii)

结果如下:

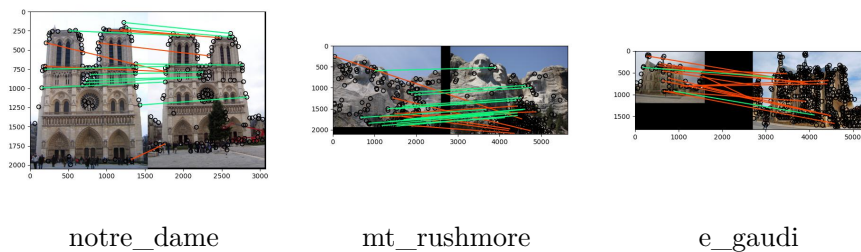
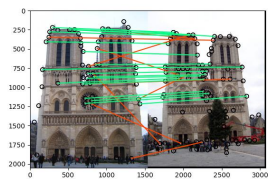


图 6: 结果展示

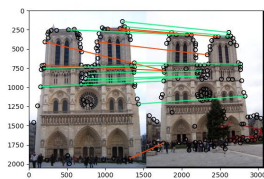
(e)

(i)

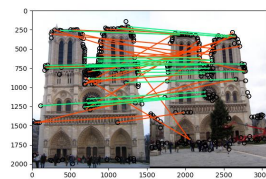
不同 scale 下的结果:



notre_dame_0.3

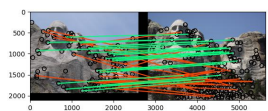


notre_dame_0.5

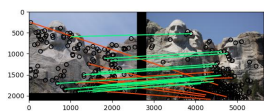


notre_dame_0.8

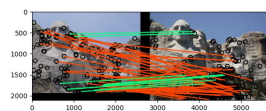
图 7: notre_dame



mt_rushmore_0.3

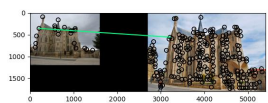


mt_rushmore_0.5

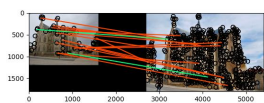


mt_rushmore_0.8

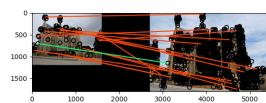
图 8: mt_rushmore



e_gaudi_0.3



e_gaudi_0.5



e_gaudi_0.8

图 9: e_gaudi

(ii)

实现一个 Multi-scale descriptor, 我通过在 params 中接受一个 scales 数组实现:

```
1     def get_features(image_origin, x, y, feature_width
2         , scales):
3     features_final = []
4     for scale in scales:
5         image = np.float32(rescale(image_origin, scale
6             ))
7         h = image.shape[0]
8         w = image.shape[1]
9
10        offset = feature_width // 2
11        num_points = x.shape[0]
12
13        # 计算索引范围
14        x_start = x - offset
15        x_stop = x + offset
16        y_start = y - offset
17        y_stop = y + offset
18
19        # Compute padding parameters
20        x_min = x_start.min()
21        x_max = x_stop.max()
22        y_min = y_start.min()
23        y_max = y_stop.max()
24
25        x_pad = [0, 0]
26        y_pad = [0, 0]
27        if x_min < 0:
28            x_pad[0] = -x_min
29        if y_min < 0:
30            y_pad[0] = -y_min
```

```

29         if x_max - w >= 0:
30             x_pad[1] = x_max - w + 1
31         if y_max - h >= 0:
32             y_pad[1] = y_max - h + 1
33
34         x_start += x_pad[0]
35         x_stop += x_pad[0]
36         y_start += y_pad[0]
37         y_stop += y_pad[0]
38
39         # 对图片进行padding
40         image = np.pad(image, [y_pad, x_pad], mode="
            constant")
41
42         # 对每个维度窗口下建立索引
43         cell_size = 4
44         num_blocks = feature_width // cell_size
45
46         x_idx = np.array([np.arange(start, stop) for
            start, stop in zip(x_start, x_stop)])
47         y_idx = np.array([np.arange(start, stop) for
            start, stop in zip(y_start, y_stop)])
48
49         # 转置之前 : (num_blocks, num_points,
            cell_size)
50         x_idx = np.array(np.split(x_idx, num_blocks,
            axis=1))
51         y_idx = np.array(np.split(y_idx, num_blocks,
            axis=1))
52
53         # 转置之后: (num_points, num_blocks, cell_size
            )
54         x_idx = x_idx.transpose([1, 0, 2])

```

```

55     y_idx = y_idx.transpose([1, 0, 2])
56
57     # 对窗口中的每个像素建立索引
58     x_idx = np.tile(np.tile(x_idx, cell_size), [1,
59                             num_blocks, 1]).flatten()
60     y_idx = np.tile(np.repeat(y_idx, cell_size,
61                             axis=2), num_blocks).flatten()
62
63     # 计算偏导数
64     partial_x = filters.sobel_h(image)
65     partial_y = filters.sobel_v(image)
66     # 计算梯度模长
67     magnitude = np.sqrt(partial_x * partial_x +
68                             partial_y * partial_y)
69     # 计算梯度方向
70     orientation = np.arctan2(partial_y, partial_x)
71     + np.pi
72     # 梯度近似为最近的角度
73     orientation = np.mod(np.round(orientation /
74     (2.0 * np.pi) * 8.0), 8)
75     orientation = orientation.astype(np.int32)
76     # 对梯度做高斯平滑
77     magnitude = filters.gaussian(magnitude, sigma=
78     offset)
79
80     # 所有的块数据转为1维
81     magnitude_in_pixels = magnitude[y_idx, x_idx]
82     orientation_in_pixels = orientation[y_idx,
83     x_idx]
84
85     # 转为数组 (num_patches, cell_size, cell_size)
86     magnitude_in_cells = magnitude_in_pixels.
87     reshape((-1, cell_size * cell_size))

```

```

80         orientation_in_cells = orientation_in_pixels.
            reshape((-1, cell_size * cell_size))
81
82     # 对每个cel计算梯度方向加权和
83     features = np.array(list(
84         map(lambda array, weight: np.bincount(
            array, weight, minlength=8),
            orientation_in_cells,
            magnitude_in_cells)))
85
86     # 每一行都是角点对应的一个特征向量
87     features = features.reshape((num_points, -1))
88     features = features / np.linalg.norm(features,
            axis=-1).reshape((-1, 1))
89     features[features >= 0.2] = 0.2
90     features = features / np.linalg.norm(features,
            axis=-1).reshape((-1, 1))
91     features_final.append(features)
92
93     return features_final

```

(iii)

使用 Ball-Tree 来加速匹配过程:

```

1     def match_features(im1_features, im2_features):
2
3         matches = []
4         confidences = []
5         threshold = 0.9
6         from sklearn.neighbors import KDTree
7         kd_tree = KDTree(im1_features, leaf_size=3)
8
9         dist, ind = kd.query(im2_features, 2)

```



```

10
11     for i in range(len(ind)):
12         index = ind[i]
13         distances = dist[i]
14
15         if distances[0] / distances[1] < threshold:
16             matches.append([index[0], i])
17             confidences.append(1 - distances[0])
18 matches = np.array(matches)
19 confidences = np.array(confidences)
20 return matches, confidences

```