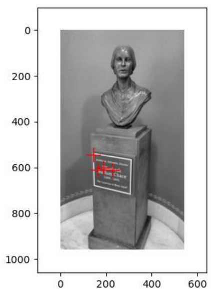
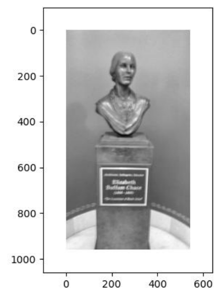
1.1

(a)

结果如下:

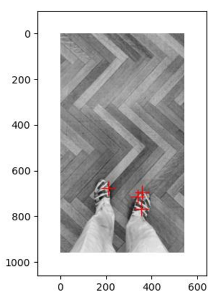


Chase1

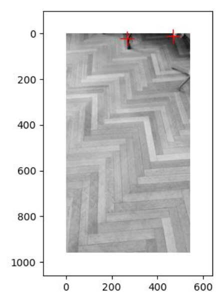


Chase2

图 1: Chase 组

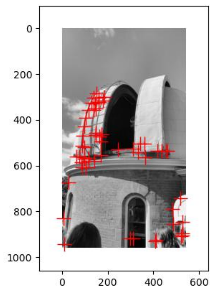


RISHLibrary1

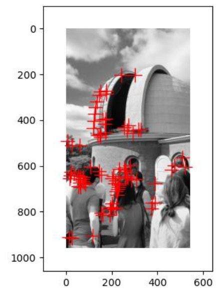


RISHLibrary2

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) $y = \lambda n + X$

$x = (u, v) \Rightarrow (u, v, 1)$

$n = (a, b)$

$$y = \lambda \begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \lambda a + u \\ \lambda b + v \end{pmatrix} \Rightarrow \begin{pmatrix} \lambda a + u \\ \lambda b + v \\ 1 \end{pmatrix}$$

$$y' = Hy = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} \lambda a + u \\ \lambda b + v \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} h_{11}(\lambda a + u) + h_{12}(\lambda b + v) + h_{13} \\ h_{21}(\lambda a + u) + h_{22}(\lambda b + v) + h_{23} \\ h_{31}(\lambda a + u) + h_{32}(\lambda b + v) + h_{33} \end{pmatrix}$$

(b) 当 ~~$h_{31}(\lambda a+u)+h_{32}(\lambda b+v)+h_{33}$ 无穷大,~~

即 ~~l与投影平面平行时,~~

当l平行于投影平面或沿投影方向时，l′收敛于1个消失点.

$$y'_v = \lim_{\lambda \to \infty} \left(\frac{h_{11}\lambda a + h_{12}\lambda b + h_{13}}{h_{31}\lambda a + h_{32}\lambda b + h_{33}}, \frac{h_{21}\lambda a + h_{22}\lambda b + h_{23}}{h_{31}\lambda a + h_{32}\lambda b + h_{33}} \right)$$

$$= \left(\frac{h_{11}a + h_{12}b}{h_{31}a + h_{32}b}, \frac{h_{21}a + h_{22}b}{h_{31}a + h_{32}b} \right).$$

(c) 当H不是有效的变换矩阵,

或 • l与图像平面的平行线相交时.

图 4: 手写题

2.3

(a)

实现如下:

```
def get_interest_points(image, feature_width):

# 设置参数
sigma = 1.0
k = 0.06
threshold = 0.1

# 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
Ix = filters.sobel_v(image=image)
Iy = filters.sobel_h(image=image)

Ixx = Ix**2
Iyy = Iy**2
Ixy = Ix * Iy

# 将高斯滤波器用于上面算子
Ixx = filters.gaussian(Ixx, sigma=sigma)
Iyy = filters.gaussian(Iyy, sigma=sigma)
Ixy = filters.gaussian(Ixy, sigma=sigma)

# 计算每个像素的 R= det(H) - k(trace(H))²。det(H)表示矩阵H的行列式，trace表示矩阵H的迹。通常k的取值范围为[0.04,0.16]。
detH = (Ixx * Iyy) - (Ixy ** 2)
traceH = Ixx + Iyy
R = detH - k * ((traceH) ** 2)

# 满足 R>=max(R) * 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
    # 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
    points = feature.peak_local_max(R, min_distance=feature_width, threshold_rel=threshold)

    xs = points[:, 1]
    ys = points[:, 0]

    return xs, ys
```

(b)

实现如下:

normalized patches:

```
def get_features(image, x, y, feature_width):
    h = image.shape[0]
    w = image.shape[1]

    offset = feature_width // 2
    num_points = x.shape[0]

    # 计算索引范围
    x_start = x - offset
    x_stop = x + offset
    y_start = y - offset
    y_stop = y + offset

    # Compute padding parameters
    x_min = x_start.min()
    x_max = x_stop.max()
```

```python
    y_min = y_start.min()
    y_max = y_stop.max()

    x_pad = [0, 0]
    y_pad = [0, 0]
    if x_min < 0:
        x_pad[0] = -x_min
    if y_min < 0:
        y_pad[0] = -y_min
    if x_max - w >= 0:
        x_pad[1] = x_max - w + 1
    if y_max - h >= 0:
        y_pad[1] = y_max - h + 1

    x_start += x_pad[0]
    x_stop += x_pad[0]
    y_start += y_pad[0]
    y_stop += y_pad[0]

    # 对图片进行pudding
    image = np.pad(image, [y_pad, x_pad], mode="constant")

    # 对每个维度窗口下建立索引
    cell_size = 4
    num_blocks = feature_width // cell_size

    x_idx = np.array([np.arange(start, stop) for start, stop in zip(x_start, x_stop)])
    y_idx = np.array([np.arange(start, stop) for start, stop in zip(y_start, y_stop)])

    # 转置之前 : (num_blocks, num_points, cell_size)
```

```python
    x_idx = np.array(np.split(x_idx, num_blocks, axis=1))
    y_idx = np.array(np.split(y_idx, num_blocks, axis=1))

    # 转置之后：(num_points, num_blocks, cell_size)
    x_idx = x_idx.transpose([1, 0, 2])
    y_idx = y_idx.transpose([1, 0, 2])

    # 对窗口中的每个像素建立索引
    x_idx = np.tile(np.tile(x_idx, cell_size), [1, num_blocks, 1]).flatten()
    y_idx = np.tile(np.repeat(y_idx, cell_size, axis=2), num_blocks).flatten()

    # 计算偏导数
    partial_x = filters.sobel_h(image)
    partial_y = filters.sobel_v(image)
    # 计算梯度模长
    magnitude = np.sqrt(partial_x * partial_x + partial_y * partial_y)
    # 计算梯度方向
    orientation = np.arctan2(partial_y, partial_x) + np.pi
    # 梯度近似为最近的角度
    orientation = np.mod(np.round(orientation / (2.0 * np.pi) * 8.0), 8)
    orientation = orientation.astype(np.int32)
    # 对梯度做高斯平滑
    magnitude = filters.gaussian(magnitude, sigma=offset)

    # 所有的块数据转为1维
```

```
    magnitude_in_pixels = magnitude[y_idx, x_idx]
    orientation_in_pixels = orientation[y_idx, x_idx]

    # 转为数组 (num_patches, cell_size, cell_size)
    magnitude_in_cells = magnitude_in_pixels.reshape((-1, cell_size * cell_size))
    orientation_in_cells = orientation_in_pixels.reshape((-1, cell_size * cell_size))

    # 对每个cel计算梯度方向加权和
    features = np.array(list(
        map(lambda array, weight: np.bincount(array, weight, minlength=8), orientation_in_cells, magnitude_in_cells)))

    # 每一行都是角点对应的一个特征向量
    features = features.reshape((num_points, -1))
    features = features / np.linalg.norm(features, axis=-1).reshape((-1, 1))
    features[features >= 0.2] = 0.2
    features = features / np.linalg.norm(features, axis=-1).reshape((-1, 1))

    return features
```

SIFT:

(c)

实现如下:

```
def match_features(im1_features, im2_features):

```

```python
    assert im1_features.shape[1] == im2_features.shape[1]
    # 设置confidence
    confidence = 1

    matches = []
    confidences = []

    for i in range(im1_features.shape[0]):
        distances = np.sqrt(
            ((im1_features[i, :] - im2_features) ** 2).sum(axis=1))
        indexes = np.argsort(distances)
        min_distance = distances[indexes[0]]
        second_min_distance = distances[indexes[1]]

        ratio = min_distance / second_min_distance if second_min_distance != 0 else 0
        if ratio < confidence:
            matches.append([i, indexes[0]])
            confidences.append(1 - ratio)

    matches = np.asarray(matches)
    confidences = np.asarray(confidences)

    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:

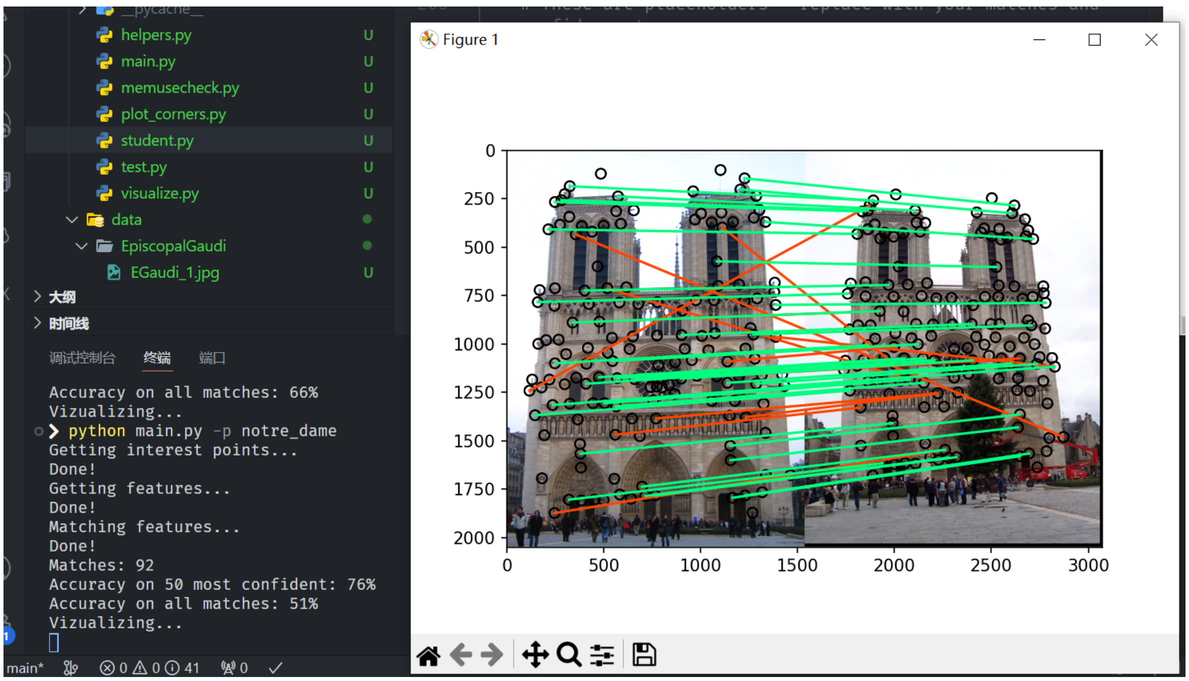


图 5: normalized patches 结果

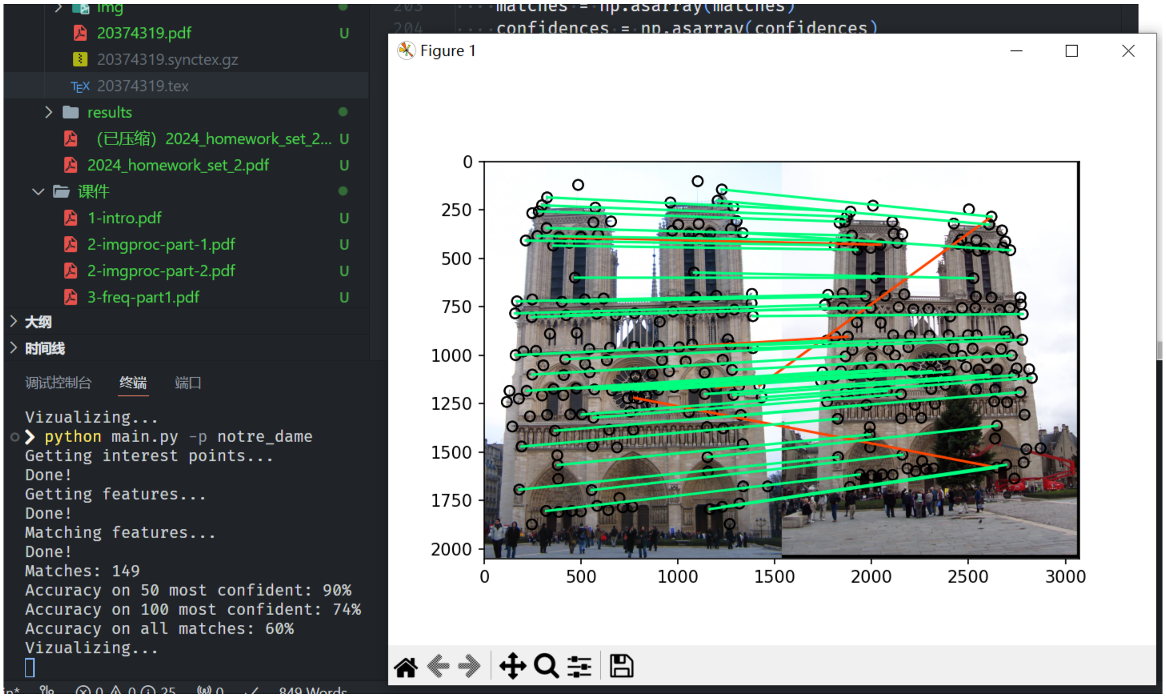
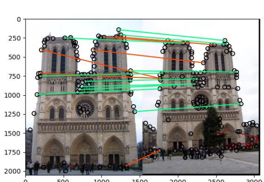


图 6: sift 结果

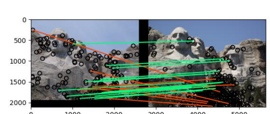
从图片中可以看到，使用 sift 相比于 normalized patches 大概能提升 10%。

(iii)

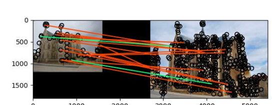
结果如下:



notre_dame



mt_rushmore



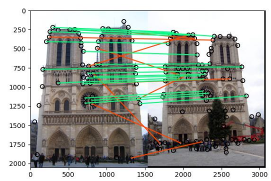
e_gaudi

图 7: 结果展示

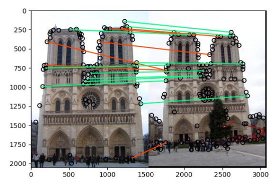
(e)

(i)

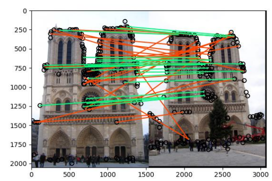
不同 scale 下的结果:



notre_ dame_0.3

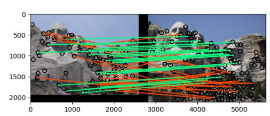


notre_ dame_0.5

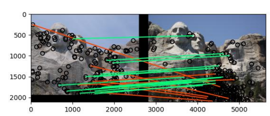


notre_ dame_0.8

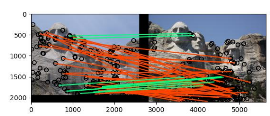
图 8: notre_dame



mt_ rushmore_0.3

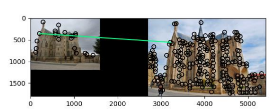


mt_ rushmore_0.5

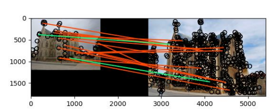


mt_ rushmore_0.8

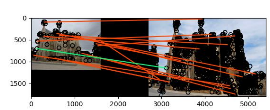
图 9: mt_rushmore



e_ gaudi_0.3



e_ gaudi_0.5



e_ gaudi_0.8

图 10: 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:

```
def get_features(image_origin, x, y, feature_width
    , scales):
features_final = []
for scale in scales:
    image = np.float32(rescale(image_origin, scale
        ))
    h = image.shape[0]
    w = image.shape[1]

     offset = feature_width // 2
    num_points = x.shape[0]

    # 计算索引范围
    x_start = x - offset
    x_stop = x + offset
    y_start = y - offset
    y_stop = y + offset

    # Compute padding parameters
    x_min = x_start.min()
    x_max = x_stop.max()
    y_min = y_start.min()
    y_max = y_stop.max()

    x_pad = [0, 0]
    y_pad = [0, 0]
     if x_min < 0:
        x_pad[0] = -x_min
     if y_min < 0:
        y_pad[0] = -y_min
```

```python
    if x_max - w >= 0:
        x_pad[1] = x_max - w + 1
    if y_max - h >= 0:
        y_pad[1] = y_max - h + 1

    x_start += x_pad[0]
    x_stop += x_pad[0]
    y_start += y_pad[0]
    y_stop += y_pad[0]

    # 对图片进行pudding
    image = np.pad(image, [y_pad, x_pad], mode="constant")

    # 对每个维度窗口下建立索引
    cell_size = 4
    num_blocks = feature_width // cell_size

    x_idx = np.array([np.arange(start, stop) for start, stop in zip(x_start, x_stop)])
    y_idx = np.array([np.arange(start, stop) for start, stop in zip(y_start, y_stop)])

    # 转置之前 : (num_blocks, num_points, cell_size)
    x_idx = np.array(np.split(x_idx, num_blocks, axis=1))
    y_idx = np.array(np.split(y_idx, num_blocks, axis=1))

    # 转置之后: (num_points, num_blocks, cell_size)
    x_idx = x_idx.transpose([1, 0, 2])
```

```
        y_idx = y_idx.transpose([1, 0, 2])

        # 对窗口中的每个像素建立索引
        x_idx = np.tile(np.tile(x_idx, cell_size), [1, num_blocks, 1]).flatten()
        y_idx = np.tile(np.repeat(y_idx, cell_size, axis=2), num_blocks).flatten()

        # 计算偏导数
        partial_x = filters.sobel_h(image)
        partial_y = filters.sobel_v(image)
        # 计算梯度模长
        magnitude = np.sqrt(partial_x * partial_x + partial_y * partial_y)
        # 计算梯度方向
        orientation = np.arctan2(partial_y, partial_x) + np.pi
        # 梯度近似为最近的角度
        orientation = np.mod(np.round(orientation / (2.0 * np.pi) * 8.0), 8)
        orientation = orientation.astype(np.int32)
        # 对梯度做高斯平滑
        magnitude = filters.gaussian(magnitude, sigma=offset)

        # 所有的块数据转为1维
        magnitude_in_pixels = magnitude[y_idx, x_idx]
        orientation_in_pixels = orientation[y_idx, x_idx]

        # 转为数组 (num_patches, cell_size, cell_size)
        magnitude_in_cells = magnitude_in_pixels.reshape((-1, cell_size * cell_size))
```

```
        orientation_in_cells = orientation_in_pixels.reshape((-1, cell_size * cell_size))

        # 对每个cel计算梯度方向加权和
        features = np.array(list(
            map(lambda array, weight: np.bincount(array, weight, minlength=8),
                orientation_in_cells,
                magnitude_in_cells)))

        # 每一行都是角点对应的一个特征向量
        features = features.reshape((num_points, -1))
        features = features / np.linalg.norm(features, axis=-1).reshape((-1, 1))
        features[features >= 0.2] = 0.2
        features = features / np.linalg.norm(features, axis=-1).reshape((-1, 1))
        features_final.append(features)

    return features_final
```

(iii)

使用 Ball-Tree 来加速匹配过程:

```
def match_features(im1_features, im2_features):

matches = []
confidences = []
threshold = 0.9
from sklearn.neighbors import KDTree
kd_tree = KDTree(im1_features, leaf_size=3)

dist, ind = kd.query(im2_features, 2)
```

```python

    for i in range(len(ind)):
        index = ind[i]
        distances = dist[i]

        if distances[0] / distances[1] < threshold:
            matches.append([index[0], i])
            confidences.append(1 - distances[0])
    matches = np.array(matches)
    confidences = np.array(confidences)
    return matches, confidences
```