中山大学计算机学院

模式识别实验报告

第一次作业

（2024学年春季学期）
课程名称：

模式识别	第一次作业	专业(方向)	计算机科学与技术
学号	21307185	**姓名**	张礼贤
Email	zhanglx78@mail2.sysu.edu.cn	**完成日期**	2024.5.5

1. 实验目的

1. 熟悉Harris 角点检测器的原理和基本使用
2. 熟悉RANSAC 抽样一致方法的使用场景
3. 熟悉HOG 描述子的基本原理

---

2. 实验要求

1. 提交实验报告，要求有适当步骤说明和结果分析，对比
2. 将代码和结果打包提交
3. 实验可以使用现有的特征描述子实现

---

3. 实验内容

1. 使用 Harris 焦点检测器寻找关键点。

2. 构建描述算子来描述图中的每个关键点，比较两幅图像的两组描述子，并进行匹配。
3. 根据一组匹配关键点，使用RANSAC 进行仿射变换矩阵的计算。
4. 将第二幅图变换过来并覆盖在第一幅图上，拼接形成一个全景图像。
5. 实现不同的描述子，并得到不同的拼接结果。

---

4. 实验过程

4.1 Harris 角点算法

1. **算法思想：**
   哈里斯角点检测算法通过计算图像局部区域的特征值来判断该区域是否为角点。对于平坦区域，其灰度值在各个方向上变化较小；对于边缘区域，其灰度值在某个方向上变化较大；而对于角点区域，则在多个方向上都有较大的灰度变化。

2. **实现步骤：**

   1. **计算图像灰度梯度：** 对输入的灰度图像进行梯度计算，常用的方法是使用Sobel算子或Prewitt算子。

   2. **计算结构矩阵：** 对每个像素点，根据其局部邻域内的梯度值，计算结构矩阵M，即每个像素点的梯度矩阵。

   结构矩阵M定义为：

$$M = \sum_{i,j} w(i,j) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

   其中，I_x 和 I_y 分别表示像素点处的水平和垂直方向的梯度值，$w(i,j)$ 是一个权重函数，通常为高斯窗口函数，用于加权求和以降低噪声对角点检测的影响。

   3. **计算角点响应函数：** 对于每个像素点，利用其对应的结构矩阵M，计算角点响应函数R。

   角点响应函数R定义为：

$$R = det(M) - k \times trace^2(M)$$

其中，$det(M)$ 是矩阵M的行列式，$trace(M)$ 是矩阵M的迹，k是一个常数（通常取0.04 - 0.06），用于调节角点的敏感度。

4. **角点检测：** 对于每个像素点，比较其对应的角点响应函数R与设定的阈值，如果R大于阈值，则将该像素点标记为角点。

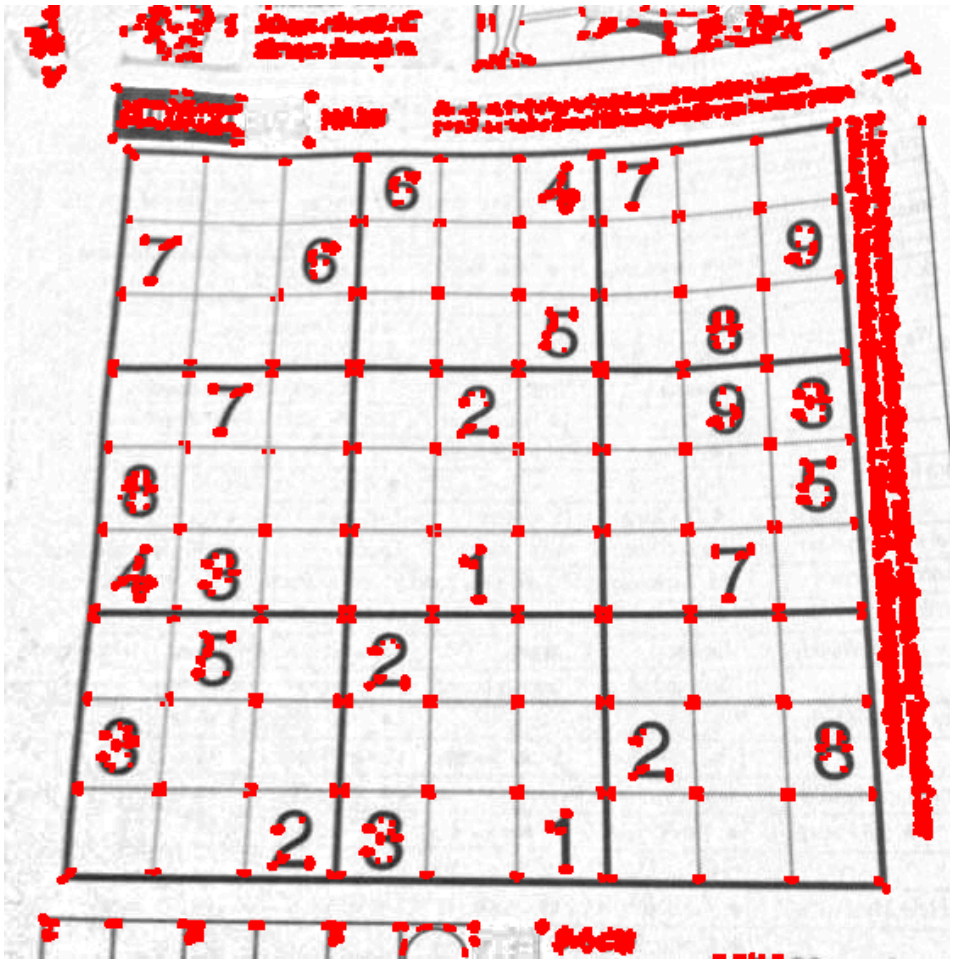
5. **非极大值抑制：** 对检测到的角点进行非极大值抑制，保留局部最大的角点响应值，并抑制其余的非最大值。

3. **代码实现：**

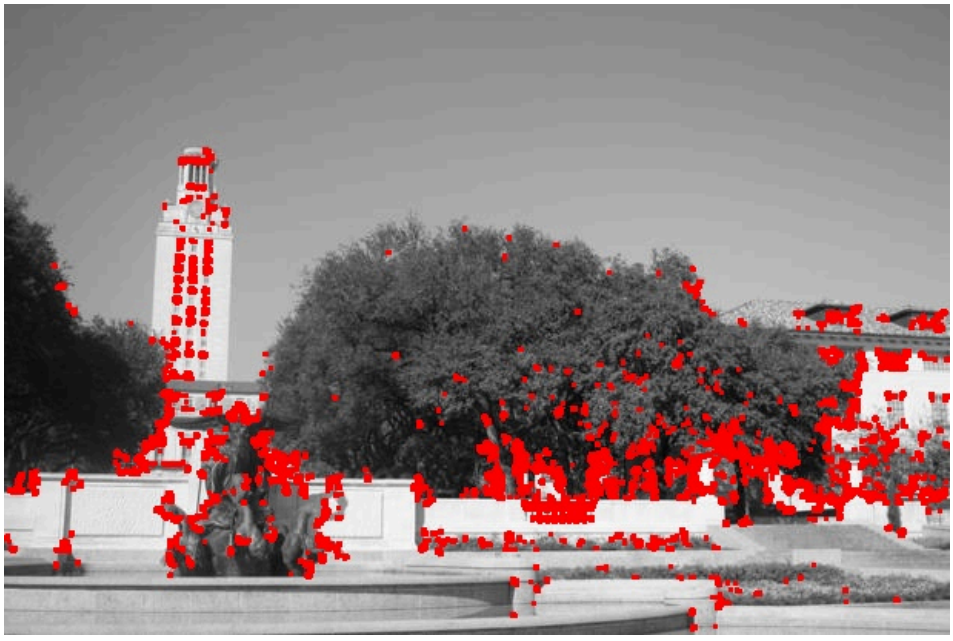
```python
def harris_corner_detection(image, threshold=0.01,
window_size=3, k=0.04):
# 1. 计算图像的梯度
dx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
dy = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

# 2. 计算 Harris 角点响应函数
Ixx = dx ** 2
Ixy = dx * dy
Iyy = dy ** 2

height, width = image.shape
offset = window_size // 2
corner_response = np.zeros((height, width))

for y in range(offset, height - offset):
    for x in range(offset, width - offset):
        window_Ixx = Ixx[y - offset : y + offset + 1,
x - offset : x + offset + 1]
        window_Ixy = Ixy[y - offset : y + offset + 1,
x - offset : x + offset + 1]
        window_Iyy = Iyy[y - offset : y + offset + 1,
x - offset : x + offset + 1]

        # 计算局部窗口内的梯度协方差矩阵的特征值
        Sxx = np.sum(window_Ixx)
        Sxy = np.sum(window_Ixy)
        Syy = np.sum(window_Iyy)

        # 计算角点响应函数值
        det = Sxx * Syy - Sxy ** 2
        trace = Sxx + Syy
        corner_response[y, x] = det - k * trace ** 2

# 3. 对角点响应函数进行阈值处理
corner_response[corner_response < threshold *
np.max(corner_response)] = 0

# 4. 非极大值抑制
corner_response = cv2.dilate(corner_response, None)

return corner_response
```

4. **实验结果：**

- **数独：**



- **uttower1：**



- **uttower2：**

可以看到，Harris 角点检测算法能够有效地检测出图像中的角点，对于数独图像，角点主要集中在数字的交叉点处；对于建筑物图像，角点主要集中在建筑物的边缘和拐角处。

4.2 HOG 描述子与拼接

1. **算法思想：**

    1. **图像梯度计算**：通过使用 Sobel 算子计算图像中每个像素点的梯度幅值和方向。

    $$G_x = \frac{\partial I}{\partial x}, \quad G_y = \frac{\partial I}{\partial y}$$

    2. **梯度方向直方图统计**：将图像划分为多个小区域（cells），对每个区域内的梯度方向进行统计，形成梯度方向直方图。

    $$H(\theta) = \sum_{(x,y) \in \text{cell}} \text{magnitude}(x,y) \cdot \delta(\theta - \text{angle}(x,y))$$

    3. **梯度直方图归一化**：对每个区域内的梯度方向直方图进行归一化，增强对光照变化的鲁棒性。

    $$\text{histogram}_{\text{norm}} = \frac{\text{histogram}}{\sqrt{\text{sum of squares} + \epsilon}}$$

4. **特征块描述子生成**：将归一化的梯度方向直方图组合起来，形成最终的特征块描述子。

$$\text{HOG descriptor} = \left[\text{histogram}_{\text{norm}}^{(1)}, \text{histogram}_{\text{norm}}^{(2)}, \ldots, \text{histogram}_{\text{norm}}^{(N)}\right]$$

2. **实现步骤：**

1. **图像预处理：** 对输入的图像进行预处理，通常包括图像灰度化、图像归一化、图像梯度计算等操作。

2. **计算梯度直方图：** 将图像划分为若干个小的局部区域（cell），对每个局部区域计算梯度直方图。

3. **梯度直方图归一化：** 对每个局部区域的梯度直方图进行归一化，以降低光照和阴影等因素对特征提取的影响。

4. **梯度直方图拼接：** 将所有局部区域的梯度直方图拼接起来，形成整个图像的特征描述。

5. **使用ransac算法进行拼接：** 使用ransac算法对两幅图像的特征描述子进行匹配，找到两幅图像之间的对应关系，然后通过对应关系计算两幅图像之间的仿射变换矩阵，将第二幅图像变换到第一幅图像的坐标系中，然后将两幅图像拼接在一起。

3. **代码实现：**

- hog描述子提取：

```
def compute_hog_descriptor(patch, cell_size=(8, 8),
num_bins = 9, bin_range = 20):
    # 计算梯度
    grad_x = cv.Sobel(patch, cv.CV_64F, 1, 0,
ksize=4)
    grad_y = cv.Sobel(patch, cv.CV_64F, 0, 1,
ksize=4)
    mag = np.sqrt(grad_x ** 2 + grad_y ** 2)
    angle = np.arctan2(grad_y, grad_x) * (180 /
np.pi)

    # 划分 cells
    mag_cells =
skimage.util.shape.view_as_blocks(mag, cell_size)
    angle_cells =
skimage.util.shape.view_as_blocks(angle, cell_size)

    rows, cols = mag_cells.shape[:2]
    cells = np.zeros((rows, cols, num_bins))

    # 计算直方图
    for row in range(rows):
        for col in range(cols):
            for y in range(cell_size[0]):
                for x in range(cell_size[1]):
                    bin_index =
int(angle_cells[row, col, y, x] / bin_range)
                    cells[row, col, bin_index] +=
mag_cells[row, col, y, x]

    # 归一化
    cells = (cells - np.mean(cells)) /
np.std(cells)
    hog_descriptor = cells.reshape(-1)

    return hog_descriptor

# 计算关键点处图像块的 HOG 特征描述子
def calculate_keypoints_hog_descriptor(image,
keypoints, patch_size=8):
    image = image.astype(np.float32)
    descriptors = []
    for kp in keypoints:
```

```
        y, x = kp
        half_patch_size = patch_size // 2
        start_x = max(0, x - half_patch_size)
        end_x = min(image.shape[1], x +
half_patch_size)
        start_y = max(0, y - half_patch_size)
        end_y = min(image.shape[0], y +
half_patch_size)
        patch = image[start_y:end_y, start_x:end_x]

descriptors.append(calculate_hog_descriptor(patch))

    return np.array(descriptors)
```

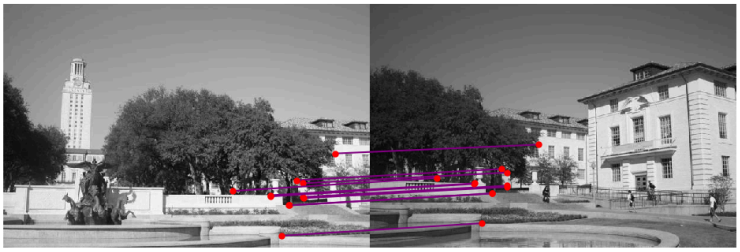
- **拼接:**

```
def estimate_homography_ransac(keypoints1,
keypoints2, matches):
num_iterations = 2000
tolerance = 5.0
best_model = None
best_inliers = 0

# 将关键点转换为点
src_pts = np.float32([keypoints1[i.queryIdx].pt
for i in matches])
dst_pts = np.float32([keypoints2[i.trainIdx].pt
for i in matches])

# RANSAC 迭代
for _ in range(num_iterations):
    # 随机选择4个点对应
    indices = np.random.choice(len(src_pts), 4,
replace=False)
    src_sample = src_pts[indices]
    dst_sample = dst_pts[indices]

    # 计算该随机样本的单应性矩阵
    M, _ = cv.findHomography(src_sample,
dst_sample)

    # 使用估计的单应性矩阵计算投影点
    projected_pts =
cv.perspectiveTransform(src_pts.reshape(-1, 1, 2),
M)

    # 计算投影点与实际目标点之间的欧几里得距离
    distances =
np.linalg.norm(projected_pts.squeeze() - dst_pts,
axis=1)

    # 统计在容差范围内的内点数量
    inliers = np.sum(distances < tolerance)

    # 如果发现更多内点，则更新最佳模型
    if inliers > best_inliers:
        best_inliers = inliers
        best_model = M
```

```
    # 将匹配列表中对应于内点的索引提取出来
    inliers_idx = np.where(distances < tolerance)[0]
    # 通过内点索引提取出匹配列表中的内点
    inliers_matches = np.array(matches)[inliers_idx]

    return best_model, inliers_matches
```

4. **实现结果：**

   - HOG描述子提取结果：



   - HOG拼接结果：

这里颜色出了问题，故采用灰度图像，但是拼接效果还是可以看出来的，但是在连接处有虚影存在。

4.3 sift描述子与拼接

1. **算法思想：**

    1. **尺度空间极值检测**：通过高斯差分金字塔检测图像中的关键点。
    2. **关键点定位**：在检测到的极值点周围，利用 Hessian 矩阵计算关键点的位置和尺度。
    3. **关键点方向分配**：为每个关键点分配主方向，以增强特征的旋转不变性。
    4. **局部图像描述**：以关键点为中心，提取局部区域的梯度信息，构建关键点的描述子。

    数学表达：

    - 高斯差分金字塔：

    $$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y)$$

    - Hessian 矩阵：

    $$H = \begin{bmatrix} L_{xx} & L_{xy} \\ L_{xy} & L_{yy} \end{bmatrix}$$

    - 关键点位置和尺度计算：

    $$(x, y, \sigma) = \arg\max |D(x, y, \sigma)|$$

    - 关键点方向分配：

    $$\theta(x, y) = \arg\max H(x, y, \sigma)$$

    - 局部图像描述：

    $$\text{SIFT descriptor} = \left[\text{gradient}_{\text{hist}}^{(1)}, \text{gradient}_{\text{hist}}^{(2)}, \ldots, \text{gradient}_{\text{hist}}^{(N)}\right]$$

2. **实现步骤：**

    1. **检测关键点：** 使用SIFT算法检测图像中的关键点，通常使用高斯差分金字塔来检测关键点。

    2. **计算描述子：** 对每个关键点，计算其周围的局部特征描述子，通常使用关键点周围的梯度直方图来描述关键点的特征。

3. **匹配关键点：** 对比两幅图像的关键点描述子，找到两幅图像之间的对应关系。

4. **使用ransac算法进行拼接：** 使用ransac算法对两幅图像的关键点进行匹配，找到两幅图像之间的对应关系，然后通过对应关系计算两幅图像之间的仿射变换矩阵，将第二幅图像变换到第一幅图像的坐标系中，然后将两幅图像拼接在一起。

这里实现有点困难，因此采用调用库函数进行实现。

3. **代码实现：**

   - **sift描述子提取：**

```
def sift_feature_matching(img1, img2):
# 初始化 SIFT 特征提取器
sift = cv2.SIFT_create()

# 检测关键点并计算描述子
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

# 创建暴力匹配器
bf = cv2.BFMatcher()

# 使用欧几里得距离进行特征匹配
matches = bf.knnMatch(des1, des2, k=2)

# 筛选好的匹配点
good_matches = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good_matches.append(m)

# 绘制匹配结果
img_matches = cv2.drawMatches(img1, kp1, img2,
kp2, good_matches, None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# 保存匹配结果图像
cv2.imwrite('Homework 1\\\\uttower_match_sift.png',
img_matches)
```

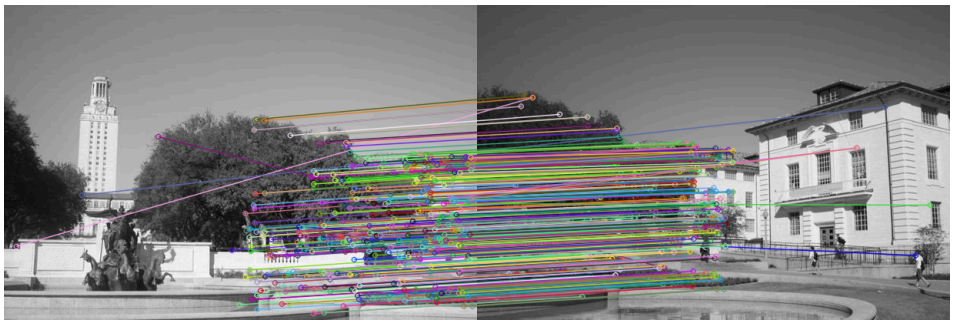
- **拼接:**

  ransac函数同上。这里展示混合策略拼接

```
# 图像拼接：尝试不同的混合策略
res = np.zeros([rows, cols, 3], np.uint8)
for row in range(0, rows):
    for col in range(0, cols):
        if not LeftImg[row, col].any():
            res[row, col] = warpImg[row, col]
        elif not warpImg[row, col].any():
            res[row, col] = LeftImg[row, col]
        else:
            LeftImgLen = float(abs(col - left))
            RightImgLen = float(abs(col - right))
            # 尝试使用不同的权重计算混合值
            if LeftImgLen < RightImgLen:
                alpha = LeftImgLen / RightImgLen
                res[row, col] = LeftImg[row, col]
* (1-alpha) + warpImg[row, col] * alpha
            else:
                alpha = RightImgLen / LeftImgLen
                res[row, col] = LeftImg[row, col]
* alpha + warpImg[row, col] * (1-alpha)
```

4. **实现结果：**

   - SIFT描述子提取结果：



- SIFT拼接结果：

可以看到，SIFT描述子提取和拼接效果较好，能够有效地匹配两幅图像之间的对应关系，并实现图像的拼接。但是在边界的处理问题上还有点点问题，但是不影响整体的观感

4.4 hog与sift的对比

1. **HOG**：

   - **设计原理**：HOG算法主要用于目标检测和图像分类任务。它将图像划分为小的局部区域（cells），计算每个区域内像素的梯度和方向，然后将这些信息组织成直方图。最终，通过对这些直方图进行归一化和连接，生成全局特征描述子。
   - **优点**：简单易懂，计算速度快，对光照变化和几何变换具有一定的鲁棒性。
   - **缺点**：对图像中的局部变化较为敏感，不具备旋转和尺度不变性。
   - **颜色问题原因**：HOG特征通常只捕获图像的梯度信息，而不包含颜色信息。因此，如果将HOG特征直接可视化，往往会导致颜色的问题，通常采用灰度渲染或伪彩色渲染的方式来表示。

2. **SIFT**：

   - **设计原理**：SIFT算法主要用于图像配准和目标识别任务。它通过检测图像中的关键点，并计算这些关键点的描述子（通常是128维的向量），从而构建出图像的特征集合。

- **优点**：具有尺度不变性和旋转不变性，对于光照变化和视角变化有一定的鲁棒性。
- **缺点**：计算量大，特征维度较高，对图像的变换和扭曲不具备很好的鲁棒性。
- **颜色问题原因**：SIFT特征的描述子通常是基于图像局部区域的梯度和方向信息，与具体颜色关系不大，因此在可视化时可能会表现为灰度图或伪彩色图。

4.5 SIFT + RANSAC 多图拼接

1. **算法思想：**
   - SIFT + RANSAC 多图拼接是一种用于多图像拼接的算法，通过对多幅图像的关键点进行匹配，并使用RANSAC算法计算多幅图像之间的仿射变换矩阵，从而实现多幅图像的拼接。
2. **实现步骤：**
   与 SIFT + RANSAC 单图拼接类似，只是在多图拼接时需要对多幅图像进行两两匹配，并计算多幅图像之间的仿射变换矩阵，然后将所有图像进行拼接。
3. **代码实现：**
   ransac和拼接代码与上面的类似，这里不再给出，这里展示多图拼接的逻辑

```
img1 = cv.imread('Homework 1\\images\\yosemite1.jpg')
img2 = cv.imread('Homework 1\\images\\yosemite2.jpg')
match(img1, img2, 1)

img1 = cv.imread('Homework 1\\images\\yosemite3.jpg')
img2 = cv.imread('Homework 1\\images\\yosemite4.jpg')
match(img1, img2, 2)

img1 = cv.imread('Homework
1\\\yosemite_stitching_sift_1.png')
img2 = cv.imread('Homework
1\\\yosemite_stitching_sift_2.png')
match(img1, img2, 3)
```

4. **实现结果：**

   - 多图拼接结果：

可以看到，SIFT + RANSAC 多图拼接算法能够有效地将多幅图像拼接在一起，实现全景图像的生成。但是拼接的结果会有一些黑色的矩阵块，通过mask掩模去除之后会导致不会呈现长方形展示，但是不影响拼接结果的观感。

---

5. 实验参考

https://github.com/fengyang95/CS131_homework/blob/master/

https://blog.csdn.net/hujingshuang/article/details/47337707