浙江大学

# 计算机视觉(本科)作业报告

作业名称： Image Stitching

姓　　名：

学　　号：

电子邮箱：

联系电话：

导　　师：

2019　年　12　月　21　日

# Image Stitching

## 一、 作业已实现的功能简述及运行简要说明

### 1.功能简述

本次实验的要求是编程实现全景图片的自动拼接功能。通过将图像列表分为左右两部分以最中央的照片为基准进行匹配，其中匹配的具体过程如下：①检测出两张图片的 SIFT 特征，并通过比率检测筛选出比较好的特征点进行匹配；②通过基于 RANSAC 的鲁棒算法得到单应性矩阵；③根据单应性矩阵进行图片的重构与拼接；④进行图片融合，消除因图像拼接造成的分割线；⑤调整图片大小，使尽可能好地呈现拼接出来的图片。

其中，在本实验过程中，没有调用 OPENCV 或其他 SDK 里与图像拼接相关的函数；最大特征值图，最小特征值图，R 图，原图上叠加检测结果图，图像的边、角点图等在实验结果中展示。

在本次实验中，最后可以得到中间的处理结果及最终的检测结果，输出成图像文件，包括原图像的特征点图，原图像的特征点连接图，拼接过程中生成图像的特征点图，拼接过程中生成图像的特征点连接图。

### 2.运行简要说明：

打开 Pycharm 运行程序，可以在运行程序过程中显示视频，并在./code 文件夹下生成 n 个 jpg 文件，分别为原图像的特征点图，原图像的特征点连接图（含所有特征点），原图像的特征点连接图（仅含匹配好的特征点），拼接过程中生成图像的特征点图，拼接过程中生成图像的特征点连接图（含所有特征点），拼接过程中生成图像的特征点连接图（仅含匹配好的特征点）。

## 二、 作业的开发与运行环境

Windows 10

Python 3.7.4

Opencv 3.4.2

## 三、 系统或算法的基本思路、原理、及流程或步骤等

程序分为以下七个步骤：

**1. 将图像列表分为左右两部分进行匹配拼接**

将图片列表分为 left_part 和 right_part，对这两部分分别进行匹配和连接。

**2. 检测出两张图片的 SIFT 特征，并通过比率检测筛选出比较好的特征点进行匹配**

将图片转换为灰度图片，匹配出其 SIFT 特征，通过比率检测得到比较好的特征点。

**3. 通过基于 RANSAC 的鲁棒算法得到单应性矩阵**

用上一步所得到的比较好的特征点，运用基于 RANSAC 的鲁棒算法进行运算得到单应性矩阵。

**4. 根据单应性矩阵进行图片的重构与拼接**

根据上一步得到的单应性矩阵，对两张图片建立对应关系，将两张图片拼接成一张图片。

**5. 进行图片融合，消除因图像拼接造成的分割线**

采用图片融合方式，在分割线范围内进行图像的融合以消除拼接造成的分割线。

**6. 调整图片大小，使尽可能好地呈现拼接出来的图片**

经过前几步之后，图像周围会出现一些明显的黑边，这里尽可能去除黑边，尽可能好地呈现拼接出来的图片。

## 四、 具体如何实现，例如关键（伪）代码、主要用到函数与算法等（为了表现清晰，在这里放置一组结果图，更多结果图会在实验结果中展示）

**1. 将图像列表分为左右两部分进行匹配拼接**

首先所有的文件名整合成一个列表，读入列表中每一张图片并重新定义其大小，读入图片运用了 `cv2.imread(filename,flags)`函数，filename 为导入图片的路径文件名，改变图像大小运用了 `cv2.resize(InputArray src, OutputArray dst, Size, fx, fy, interpolation)`函数，InputArray src 为输入图片的路径文件名，Size 为输出图片的尺寸，这里将图片缩放为 800*450 大小。在此过程后对图片进行特征点的标记和相邻两张图片的特征点匹配，具体算法会在之后描述。

然后我将图片列表分为 left_part 和 right_part，对于奇数张图片，左半部分为 n/2+1 张图片，右半部分为 n/2 张图片；对于偶数张图片，左半部分为 n/2+1 张图片，右半部分为 n/2-1 张图片，之后对这两部分分别进行匹配和连接。

以下的具体步骤介绍为对左右两部分进行拼接时都用到的一些步骤，算法不同时会具体指出。

**2. 检测出两张图片的 SIFT 特征，并通过比率检测筛选出比较好的特征点进行匹配**

对于列表中相邻两张图片进行匹配，首先检测出两张图片的 SIFT 特征。

首先我们检测出两张图片的 SIFT 特征，首先初始化 sift 检测器对象，这里用到的函数为 `cv2.xfeatures2d.SIFT_create()`；然后将图像转为灰度图像，这里用到的函数为 cv2.`cvtColor(src, code, dst, dstCn)`函数，其中 src 为需要转换的图像，code 为颜色映射类型，在程序中我运用到的实例为：cv2.`cvtColor`(img, cv2.COLOR_BGR2

BGRA)；然后提取该灰度图像的关键点，这里用到的函数为 detectAndCompute(self, image, mask, descriptors=None, useProvidedKeypoints=None)，该函数中 image 指需要检测特征点的图片，返回值为关键点列表和形状为 Number_of_Keypoints×128 的 numpy 数组。这部分的相关代码如下：

```
# 得到图片的特征值点
def getFeatures(image):
    sift = cv2.xfeatures2d.SIFT_create()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2BGRA)  # 转为灰度图像
    kp, des = sift.detectAndCompute(gray, None)  # 检测关键点，计算描述符，kp 是关键点列表，des 是形状为 Number_of_Keypoints×128 的 numpy 数组
    return kp, des
```

在这里我们采用 FLANN 匹配算法（FLANN 匹配器只能使用 SURF 和 SIFT 算法来检测角点），FLANN (Fast_Library_for_Approximate_Nearest_Neighbors)快速最近邻搜索包，是一个对大数据集和高维特征进行最近邻搜索的算法的集合。我们首先创建 FLANN 匹配器，这一部分的相关代码如下：

```
index_params = dict(algorithm=0, trees=5)
search_params = dict(checks=50)  # 指定递归遍历的次数 checks，值越高越准确，消耗的时间也就越多
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

使用上面所得到的两张图片的 SIFT 特征，进行匹配搜索，这里我们所用到的函数为 knnMatch(self, queryDescriptors, trainDescriptors, k, mask=None, compactResult=None)，其中 queryDescriptors 为待匹配描述符，trainDescriptors 为训练描述符，k 表示如果查询描述符总共有少于 k 个可能的匹配项，则计算每个查询描述符找到的最佳匹配项的数量或更少，mask 表示掩膜，compactResult 为当掩码不为空时使用的参数。这里我所运用的实例为 matches = flann.knnMatch(des2, des1, k=2)，之后通过比率检测得到比较好的特征点，寻找距离近的放入 good 列表。这一部分的代码如下：

```
#进行匹配，得到单应性矩阵
matches = flann.knnMatch(des2, des1, k=2)
good = []
for i, (m, n) in enumerate(matches):
    if m.distance < 0.7 * n.distance:
        good.append((m.trainIdx, m.queryIdx))
```

在得到 good 列表和关键点列表后，我们可以得到图片的特征点图和两张图片的特征点匹配连接图。其中，得到图片的特征点图所用到的函数为 cv2.drawKeypoints (image, keypoints, outImage, color=None, flags=None)，其中 image 为输入的图片，keypoints 为特征点向量，outImage 为输出的图像；得到两张图片的特征点图所用到的函数为 drawMatchesKnn(img1, keypoints1, img2, keypoints2, matches1to2,

outImg, matchColor=None, singlePointColor=None, matchesMask=None, flags=None)，其中 img1 指图片 1，keypoints1 指图片 1 的关键点列表，img2 指图片 2，keypoints2 指图片 2 的关键点列表，matches1to2 指图片 1 与图片 2 的匹配。

### 3. 通过基于 RANSAC 的鲁棒算法得到单应性矩阵

首先通过距离比较近的描述符，找到两幅图片的关键点，分别存为 current_points 和 previous_points。

然后运用基于 RANSAC 的鲁棒算法进行运算得到单应性矩阵，这里运用到的函数为 findHomography(srcPoints, dstPoints, method=None, ransacReprojThreshold=None, mask=None, maxIters=None, confidence=None)其中 srcPoints 代表源平面中点的坐标矩阵，dstPoints 代表目标平面中点的坐标矩阵，method 表示计算单应性矩阵所使用的方法，这里我们使用基于 RANSAC 的鲁棒算法，ransacReprojThreshold 表示将点对视为内点的最大允许重投影错误阈值。这里我们的实例应用为 H, _ = cv2.findHomography(current_points, previous_points, cv2.RANSAC, 4)。该部分的具体代码如下：

```
#进行匹配，得到单应性矩阵
matches = flann.knnMatch(des2, des1, k=2)
good = []
for i, (m, n) in enumerate(matches):
    if m.distance < 0.7 * n.distance:
        good.append((m.trainIdx, m.queryIdx))

if len(good) > 4:
    current_points = np.float32(
        [kp2[i].pt for (__, i) in good]
    )
    previous_points = np.float32(
        [kp1[i].pt for (i, __) in good]
    )
    H, _ = cv2.findHomography(current_points, previous_points, cv2.RANSAC, 4
)
```

### 4. 根据单应性矩阵进行图片的重构与拼接

根据上一步得到的单应性矩阵，对两张图片建立对应关系，将两张图片拼接成一张图片。

（1）left_part：

我们得到的 H 矩阵事实上是图片 2 映射到图片 1 上的矩阵，所以我们要得到图片 1 映射到图片 2 上的矩阵就需要调用 numpy.linalg.inv(a)得到 H 的逆矩阵 IH，并将其规范化。之后我们改变 IH 矩阵使图片 1 能够显示在之后连接起来后的画布上。

通过我们估算的位置（运用 IH 矩阵进行估算左上角和右下角的位置，计算位置的原理如 Fig-1 所示，由矩阵运算可以得到相应的位置坐标）设置后之后拼接起来的图片

的大小 dsize，运用 cv2.warpPerspective(src, M, dsize, dst=None, flags=None, borderMode=None, borderValue=None)函数来进行对于图片 1 的透视变换，其中 src 表示原图像，M 表示变换矩阵，dsize 表示输出图像的大小，borderMode 采用 cv2.BORDER_TRANSPARENT。透视变换后，将图片 2 放置在合适的位置即可。

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Fig-1

该部分相关的代码如下：

```
H = getHomography(previous_image.copy(), current_image.copy(), i+1)
IH = np.linalg.inv(H)  # 矩阵求逆

offset = np.dot(IH, np.array([0, 0, 1]))  # 矩阵积
offset = offset / offset[-1]  # 规范化

# 改变矩阵，使其可以显示在画布上
IH[0][-1] += abs(offset[0])
IH[1][-1] += abs(offset[1])

ds = np.dot(IH, np.array([previous_image.shape[1], previous_image.shape[0], 1]))
offset_y = abs(int(offset[1]))
offset_x = abs(int(offset[0]))
dsize = (offset_x + current_image.shape[1], int(ds[1]) + offset_y)

temp = cv2.warpPerspective(previous_image, IH, dsize, borderMode=cv2.BORDER_TRANSPARENT)
# print(temp.shape)
result = temp.copy()
result[offset_y: current_image.shape[0] + offset_y, offset_x: current_image.shape[1] + offset_x] = current_image
```

(2) right_part

对于 right_part 的操作我们可以直接运用上面步骤得到的 H 矩阵，通过我们估算的位置设置后之后拼接起来的图片的大小 dsize，运用 cv2.warpPerspective(src, M, dsize, dst=None, flags=None, borderMode=None, borderValue=None)函数来进行对于图片 2 的透视变换，其中 src 表示原图像，M 表示变换矩阵，dsize 表示输出图像的大小，borderMode 采用 cv2.BORDER_TRANSPARENT。透视变换后，将图片 1 放置在合适的位置即可，这里我们所运用的代码如下：

```
H = getHomography(previous_image, current_image, num + i)
```

```
ds = np.dot(H, np.array([current_image.shape[1], current_image.shape[0], 1])
)
ds = ds / ds[-1]
dsize = (int(ds[0]), max(int(ds[1]), previous_image.shape[0]))
temp = cv2.warpPerspective(current_image, H, dsize)
result = mix_and_match(previous_image, temp.copy())
```

在图片拼接的过程中，对拼接时所遇到的一些情况有一些处理，当拼接好的图像有黑色边缘时，将之前的图像的点补充在拼接好的图像上，当之前的图像像素点不是黑色时，将该点赋值在拼接好的图像上。

与该部分相关的代码如下：

```
if (np.array_equal(warped_image[j, i], [0, 0, 0])):
    warped_image[j, i] = previous_image[j, i]
else:
    if not np.array_equal(previous_image[j, i], [0, 0, 0]):
        bl, gl, rl = previous_image[j, i]
        warped_image[j, i] = [bl, gl, rl]
```

## 5. 进行图片融合，消除因图像拼接造成的分割线

在分割线范围内进行图像的融合以消除拼接造成的分割线，这里的处理思路是加权融合，在重叠部分由前一幅图像慢慢过渡到第二幅图像，即将图像的重叠区域的像素值按一定的权值相加合成新的图像。其中权重与当前处理点距重叠区域的边界的距离成正比，即 `alpha = (w - (j - start_x)) / w`，其中 `alpha` 为左边图像的权重（`(1-alpha)` 为右边图像的权重），`w` 为要处理的重叠部分的宽度，`j` 指现在的横坐标，`start_x` 指边界的横坐标。以处理列表左半部分的所有图像为例，相关代码如下：

```
alpha = (w - (j - start_x)) / w
if (direction == 'left'):
    result[i, j, 0] = alpha * temp[i, j, 0] + (1 - alpha) * result[i, j, 0]

    result[i, j, 1] = alpha * temp[i, j, 1] + (1 - alpha) * result[i, j, 1]

    result[i, j, 2] = alpha * temp[i, j, 2] + (1 - alpha) * result[i, j, 2]
```

图片融合前后的图片对比如下：

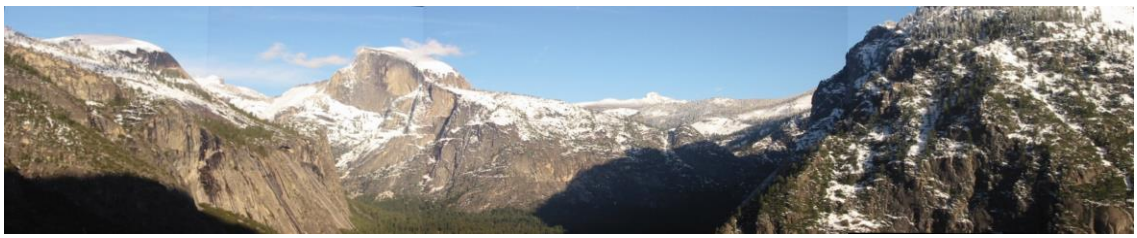


Fig-2 图片融合前

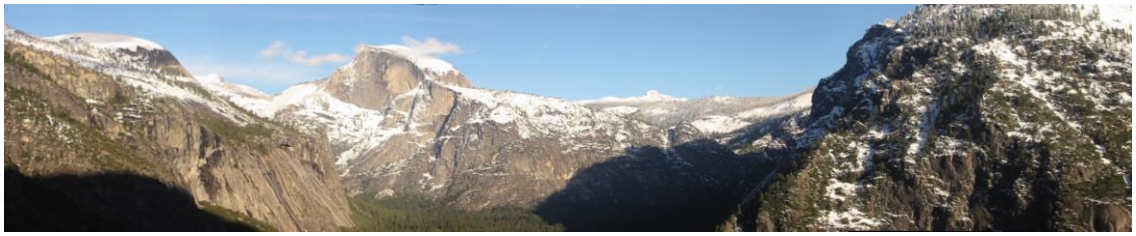


Fig-3 图片融合后

### 6. 调整图片大小，使尽可能好地呈现拼接出来的图片

经过前几步之后，图像周围会出现一些明显的黑边，这里尽可能去除黑边，尽可能好地呈现拼接出来的图片。

这里的左边界由最左边图像和最中间图像共同决定，上边界由最中间的图像决定，下边界和右边界由最右边的图像决定。其原理为：上边界和左边界为最大的首先不为 0 的像素点的坐标值，下边界和有边界为最小的首先为 0 的像素点的坐标值。

举例来说，如图所示：

Fig-4 调整图片大小前

最后结果图如下：

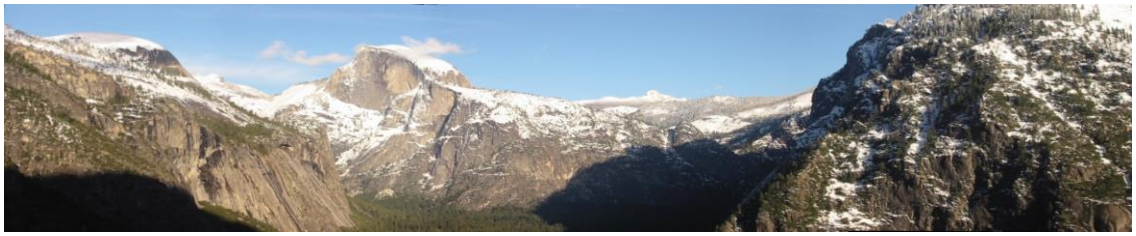


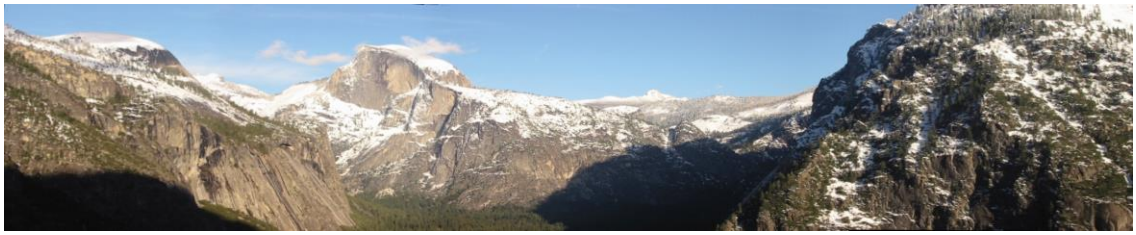
Fig-5 调整图片大小后

## 五、 实验结果与分析

我所选用的三组图像分别为题目要求的山脉、树木和曹光彪楼，其中树木和曹光彪楼为本人拍摄。

### 1.结果展示

### （1）山脉



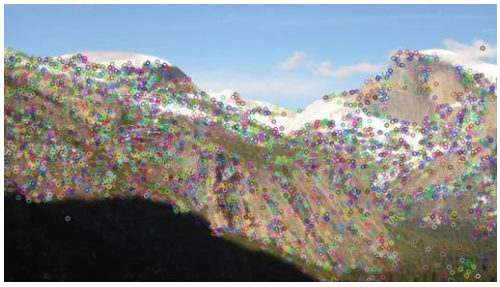
（2）树木

（3）曹光彪楼

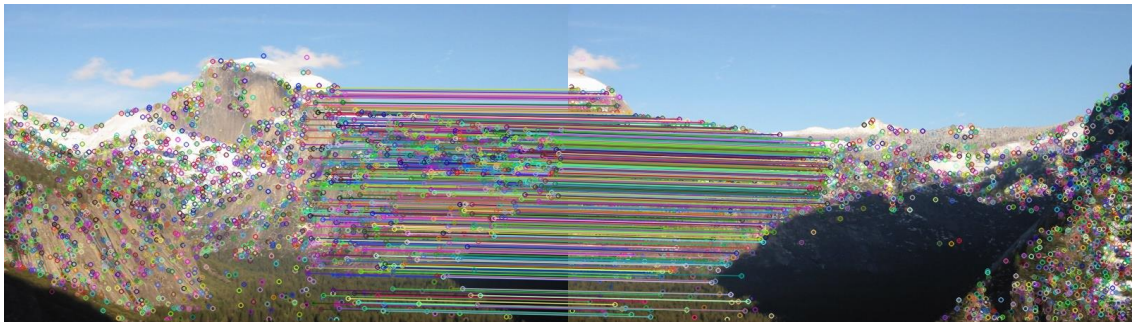
2.中间过程展示（由于图片过多，这里只选取一部分图片进行展示）

（1）山脉

①特征值点图



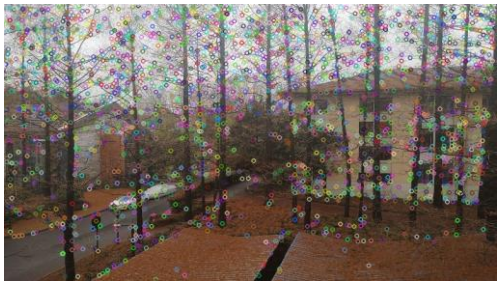
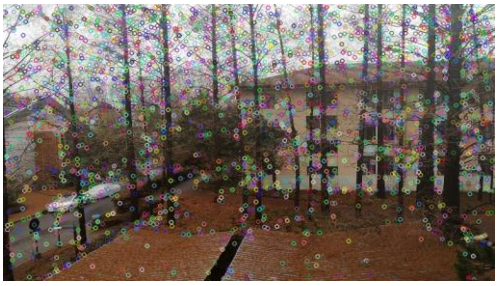
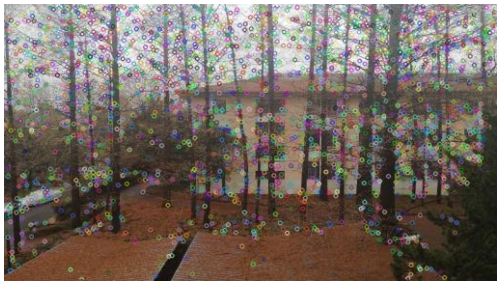
②匹配连线图（单张图片之间）



③匹配连线图（实际操作过程中）

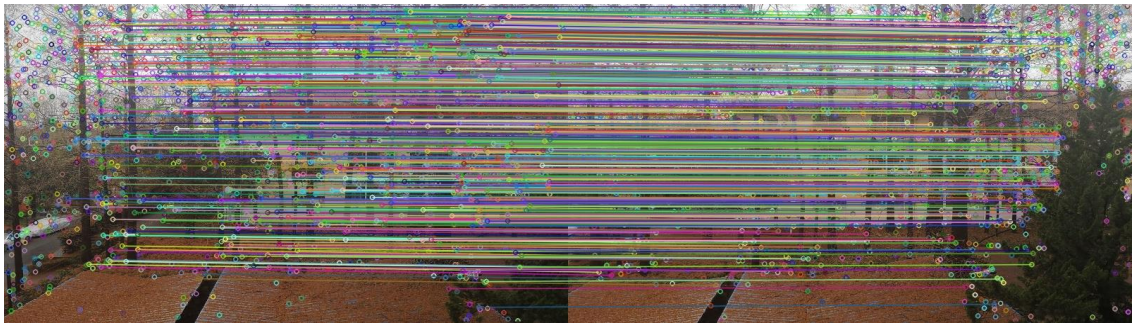
（2）树木

①特征值点图

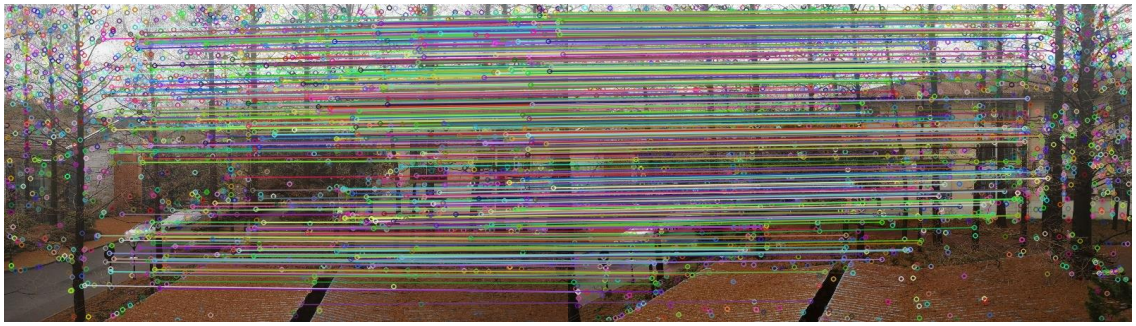




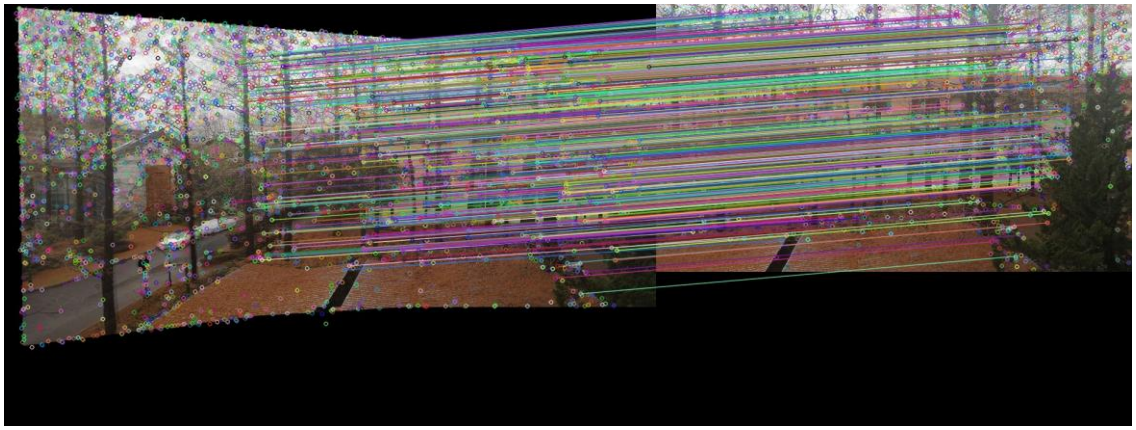

②匹配连线图（单张图片之间）





③匹配连线图（实际操作过程中）

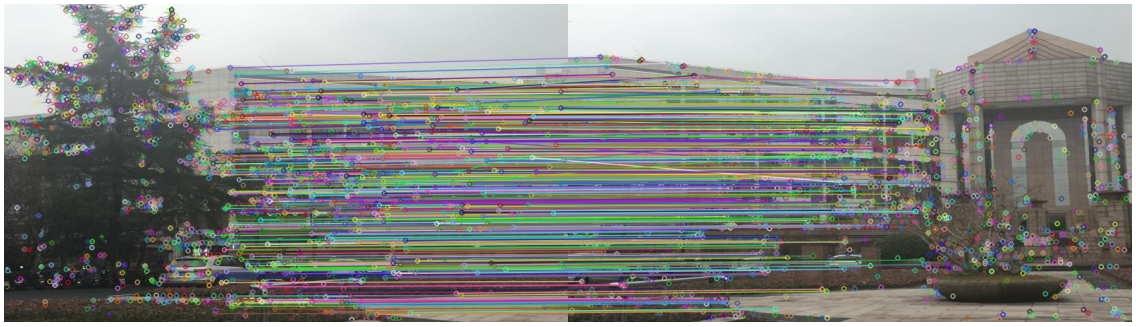


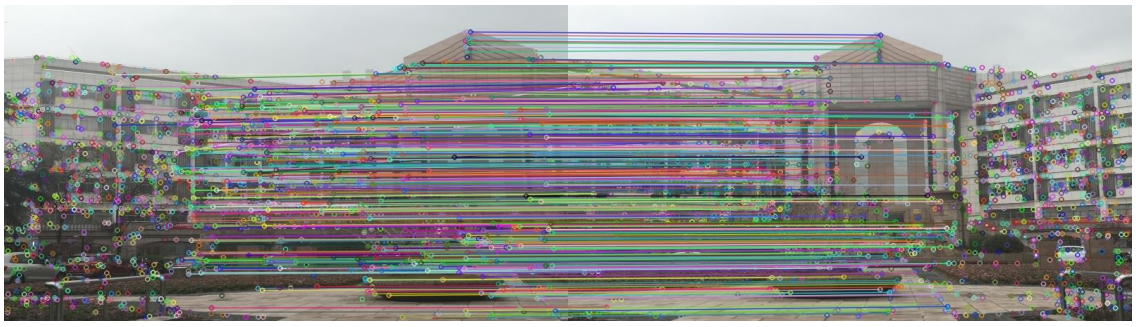


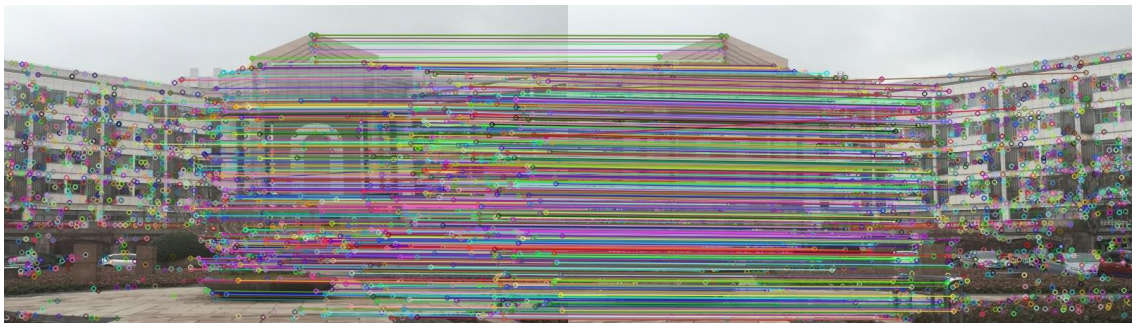
**（3）曹光彪楼**

**①特征值点图**

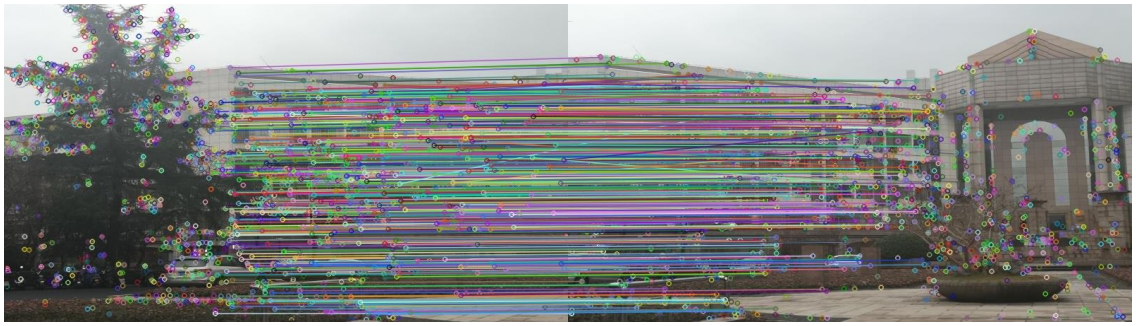
②匹配连线图（单张图片之间）

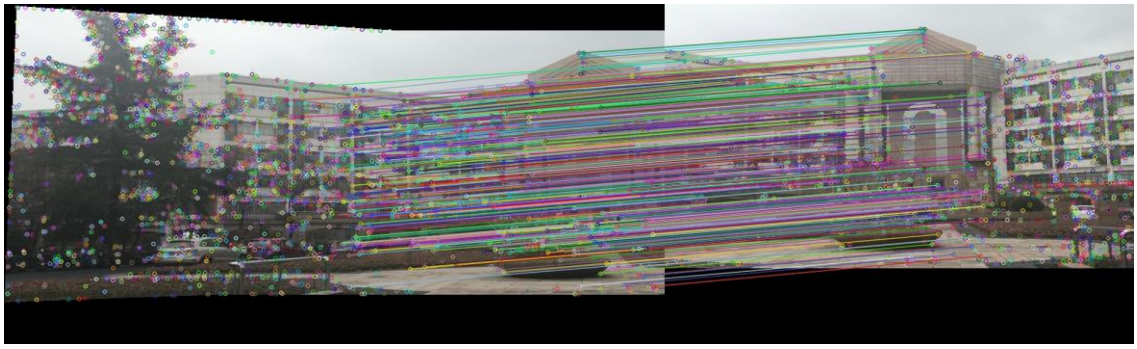


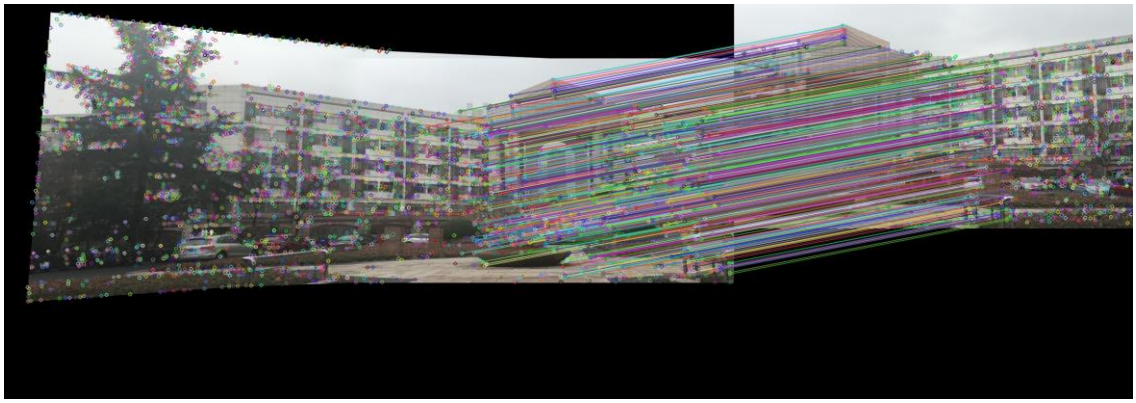




③匹配连线图（实际操作过程中）







从最终结果和中间过程来看，特征点的匹配和图片的拼接效果还是比较好的，取得了一定的成果。

## 六、 结论与心得体会

在完成本次实验的过程中主要遇到了两个问题，接下来将主要介绍这两个问题及其解决方案。

第一个问题出现在特征点的选取上，刚开始对所有特征点都进行匹配，绘制中间过程图像时出现了很多错误，后来通过比率检测得到比较好的特征点，绘制连接 good 列表中所含有的对应点，解决了这一问题。

第二个问题出现在 H 矩阵的对应问题上，刚开始并没有仔细考虑到 H 矩阵具体对应的哪张图片到哪张图片的映射，导致后面计算矩阵和估算拼接图片大小的时候在一直报错，最后对应好匹配关系和 H 的对应问题后，这个问题得到了解决。

通过本次实验，我了解了图像拼接算法的流程，在动手做、查资料、调用函数的过程发现并解决问题，在不断的思考和改正中完成了本次作业，同时巩固了 Python 的语法及运用，感觉收获很大，受益匪浅。

## 七、 参考文献


knn 匹配： https://blog.csdn.net/weixin_44072651/article/details/89262277

SIFT 特征值： https://blog.csdn.net/weixin_43772533/article/details/103253642

单应性变换： https://blog.csdn.net/moonlightpeng/article/details/80426227