图像拼接实验报告

目录

1. 实验任务

1.1 实验要求

输入两幅相同机位、不同视角拍摄的图片，输出两者拼接结果。
考察重点为拼接后的场景连续性、拼接后的光照一致性、关键点映射可视化。

1.2 实验分析

本实验的整体思路为先寻找两幅图像中的关键点，然后将关键点进行匹配，根据匹配结果对图像进行转换，然后将转换后的图像进行拼接。转换后的图像进行拼接时需要处理光照，使图像的拼接痕迹不过于明显。

1.3 实验流程

先将 images 文件夹放入当前目录下，运行 main.py。
输入：待拼接图像的序号 ***n***，合法输入为：1，2，3，4。
输出：关键点匹配结果（./middle_res/before***n***.jpg）与图像拼接结果（./result/output***n***.jpg）。
步骤 1. 使用 SIFT 算法寻找图像的特征点；
步骤 2. 使用 knnMatch 对特征点进行匹配；
步骤 3. 对 left 与 right 图像的光照进行处理；
步骤 4. 根据特征点匹配结果求解变换矩阵 H，并将 right 图像进行变换；
步骤 5. 对结果图像的融合边界进行处理；
步骤 6. 将两侧图像分别与掩码相乘，并融合两侧图像，得到最终结果。

2. 算法介绍

2.1 寻找关键点

我使用了 SIFT 算法寻找关键点。SIFT 算法，即 Scale Invariant Feature Transform，尺度不变特征转换。David Lowe 在 1998 年的论文 *Object Recognition from Local Scale-Invariant Features* 中首次提出了 SIFT 算法，并在 2004 年的论文 *Distinctive Image Features from Scale-Invariant Keypoints* 中提出了对 SIFT 算法的改进，此后，该算法受到的关注越来越高。
SIFT 算法找到的特征点具有尺度不变性，旋转不变性，光照不变性。首先构建高斯金字塔，然后检测尺度空间的极值点，再对极值点进行精确定位，选取特征点方向，最后生成关键点描述子。该算法通过连续变化尺度参数获得多尺度下的尺度空间表示序列，并对这些序列进行尺度空间主轮廓的提取，使用高斯函数实现平滑，尺度参数为标准差。通过尺度空间中各尺度图像的模糊程度逐渐变大来模拟人在距离目标由近到远时目标在视网膜上的形成过程。该算法的具体实现细节见**代码设计**。

SIFT 算法的缺点为实时性不高，对边缘光滑的目标难以准确提取特征点。本次实验我自己实现了 SIFT 算法，但该算法计算出一副输入图像（约 9M 左右）的特征点需要**若干小时**，速度非常慢。直接使用 opencv 中的 SIFT 算法可以在一分钟内算出所有特征点。

2.2 匹配关键点

我使用了 knnMatch 算法进行关键点匹配。该算法遍历描述符，将每个对应的描述符的特征进行比较，每次比较都会给出一个距离值并进行排序，最好的结果被认为是一个匹配。本次实验使用的距离测量为

$$||distance|| = \sqrt[2]{\sum_i x_i^2}$$

2.3 光照处理

本次实验我一共尝试了两种光照处理方法。
第一种方法为将两幅图片从 RGB 转为 HSV，然后分别计算两幅图像 V 分量的和，并求出 V 分量的比例 k，对右侧图像的 V 进行补偿，从而使两幅图的 V 分量接近，达到光照处理的效果。
第二种方法为直方图匹配。即将左侧图像作为参考图像，并求出其累计直方图。将右侧图像作为待变换图像，计算出其累计直方图。然后找出两个直方图之间的映射关系，将右侧的直方图匹配到左侧的直方图，使两幅图像的整体色调看起来一致，从而实现光照补偿。
这两种方法的效果见**实验结果**部分。

2.4 图像融合

第一种方法，直接将右侧图像拼接到左侧图像上，使用掩码使右侧图像只保留需要拼接的部分，去掉所有融合部分。
第二种方法，使用一个固定的偏移值 offset 并进行 feathering 处理，使用加权平均颜色值融合重叠的像素。将中心像素处的 mask 设为 1，与边界像素之间线性递减，边界处的值为 0。使用这种方法，当 offset 设置较大时，可能会出现重影。当 offset 设置过小时，过渡区域过小，接缝处仍然比较明显。
第三种方法，根据图像重合区域大小确定 offset 并进行 feathering 处理。计算出图像重合区域大小，并取其五分之一作为 offset 值，进行 feathering 处理。
图像融合部分需要考虑两方面的问题，对于左侧和右侧图像整体光照看起来差别不大时，offset 应尽可能小，这样可以保留更多左侧图片的信息，并且可以使图像不出现鬼影。当左侧和右侧图像整体光照差别较大，应当在合理范围内使 offset 尽可能大，增加过渡区域可以使图像看起来更加自然。
因此，我最终使用的方法为计算两幅图像的 V 分量的和，当两幅图像 V 分量和差别不大时，采用第二种方法进行融合，并将 offset 设置为 40；当两幅图像 V 分量和差别较大时，使用第三种方法进行融合，offset 为图像重合部分大小的 1/5。

3. 代码设计

3.1 实验整体设计

首先读入图像，然后创建 SIFT 对象并获取关键点和描述子，使用 detectAndCompute 函数获取关键点和特征描述符。然后进行关键点匹配，创建一个暴力匹配器，返回 DMatch 对象列表，每个 DMatch 对象表示关键点的一个匹配结果，尝试所有匹配从而找到最佳匹配。其中，distance 属性表示距离，距离越小匹配值越高。

```
    print("Reading images ...")
    left = cv2.imread(img1_path)
    right = cv2.imread(img2_path)
    print("Computing keypoints and descriptors ...")
    sift = mysift.MySift()
    left_kp, left_feature = sift.detectAndCompute(left, None)
    right_kp, right_feature = sift.detectAndCompute(right, None)
    print("Matching keypoints ...")
    matcher = cv2.BFMatcher()
    feature_match = matcher.knnMatch(left_feature, right_feature, k=2)
```

应用比例测试选择要使用的匹配结果，并将其记录到 good_points 和 good_matches 中，绘制关键点匹配图像，存储到./middle_res/before.jpg 中。根据筛选出的点重新确定关键点坐标，H 为转换矩阵，status 为 mask 掩码，标注出内点与外点。

```
    for m1, m2 in feature_match:
        if m1.distance < 0.75*m2.distance:
            good_points.append((m1.queryIdx, m1.trainIdx))
            good_matches.append([m1])
    match_img = cv2.drawMatchesKnn(left, left_kp, right, right_kp,
good_matches, None, flags=2)
    middle_res_path = "./middle_res/before" + img_seq + ".jpg"
    cv2.imwrite(middle_res_path, match_img)
    print(f"    matching image is {middle_res_path}")
    left_good_kp = np.float32([left_kp[i].pt for (i, _) in good_points])
    right_good_kp = np.float32([right_kp[i].pt for (_, i) in good_points])
    H, status = cv2.findHomography(right_good_kp, left_good_kp, cv2.RANSAC,
5.0)
```

计算图片的尺寸。计算左侧图像与右侧图像 V 分量的比例，计算左侧与右侧掩码。最后合并左图与右图，裁剪掉右侧的黑边，并将输出图像写入./result/ouputn.jpg。

```
left_height = left.shape[0]
    left_width = left.shape[1]
    right_width = right.shape[1]
```

```
    mix_height = left_height  # mix_height = max(l_h, r_h)
    mix_width = left_width + right_width
    left_mask_img = np.zeros((mix_height, mix_width, 3))
    bright_k = func.compute_ratio_V(left, right)
    '''
    计算 mask，具体代码见 3.3
    '''
    right_mask_img = right_mask_img * right_mask
    mix_img = left_mask_img + right_mask_img
    rows, cols = np.where(mix_img[:, :, 0] != 0)
    output_img = mix_img[min(rows):max(rows), min(cols):max(cols), :]
    result_path = "./result/output" + img_seq + ".jpg"
    cv2.imwrite(result_path, output_img)
```

3.2 SIFT 部分

该部分代码量较大，因此不在实验报告中列出全部代码。

3.2.1 detectAndCompute 函数

为了测试简便， 我将 detectAndCompute 函数的参数设为与 opencv 类似的形式。该函数相当于 sift 算法的 main 函数，直接调用该函数，即可得到关键点与描述符。
为加快处理速度，首先将彩色图转化为灰度图，修改数据类型，将图像扩大 2 倍。然后进行高斯模糊，认为原图的模糊程度为 0.5，需要将其模糊到 1.6。由于已经将图像扩大了一倍，所以模糊程度为 2*0.5，故需要进行高斯模糊的 σ 为

```
sigma_diff = np.sqrt((self.sigma ** 2) - ((2 * self.assumed_blur) ** 2))
```

然后计算高斯金字塔，根据高斯金字塔计算 DOG 金字塔。在 DOG 尺度空间内找到极值点，并将其相关信息存储到 kp_info 中。遍历 kp_info，对于其中的每一个极值点，再次进行精确定位，并计算特征点的方向角度，增加辅方向，返回一组关键点。将该组关键点从基础图像坐标转换为输入图像坐标（通过将相关属性减半），并全部加入关键点集 keypoints 中。由于增加了许多辅方向，关键点中可能会存在一些重复项，需要对重复项进行排序和删除。最后，生成描述符，并返回关键点与描述符。

```
    image = cv2.GaussianBlur(init_img, (0, 0), sigmaX=sigma_diff,
sigmaY=sigma_diff)
        print(f"    image shape is {image.shape}")
        gaussian_imgs = self.buildGaussianPyramid(image)
        dog_imgs = self.buildDoGPyramid(gaussian_imgs)
        kp_info = self.findScaleSpaceExtrema(dog_imgs)
        keypoints = []
        for kp in kp_info:
            local_result = self.adjustLocalExtrema(kp, dog_imgs)
            if local_result is not None:
                keypoint, local_index = local_result
```

```
                kps_or = self.calcOrientationHist(keypoint, local_index,
kp[0], gaussian_imgs)
                for kp_or in kps_or:
                    if self.double_img_size:
                        kp_or.pt = 0.5 * np.array(kp_or.pt)
                        kp_or.size *= 0.5
                        kp_or.octave = (kp_or.octave & ~255) |
((kp_or.octave - 1) & 255)
                    keypoints.append(kp_or)
        keypoints = removeDuplicateSorted(keypoints)
        descriptors = self.calcSIFTDescriptors(keypoints, gaussian_imgs)
```

3.2.2 createInitialImage 函数

该函数将图像转换为 float32 存储方式，并将图像转换为灰度图，然后将图像扩大 2 倍。此处需要将图像扩大 2 倍的原因为，最开始建立高斯金字塔时，要预先模糊输入图像作为第 0 个组的第 0 层图像。这相当于丢弃了最高的空域的采样率，因此需要先将图像尺寸扩大一倍生成-1 组。

```
        image = img_uint8_to_float32(img)
        grey_img = color_to_grey_img(image)
        if self.double_img_size:
            bigger_img = cv2.resize(grey_img, (0, 0), fx=2, fy=2,
interpolation=cv2.INTER_LINEAR)
            return bigger_img
        else:
            return grey_img
```

3.2.3 buildGaussianPyramid 函数

该函数的作用为构建高斯金字塔。

首先计算高斯金字塔的组数，此处需要减 2，以防止降采样过程中得到过小的图像。

```
octaves_num = int(np.log2(min(image.shape[0], image.shape[1])) - 2)
```

然后调用 computeGaussianKernel 函数得到高斯核，根据高斯核生成高斯图像，创建尺度空间金字塔，每一层都需要根据计算出的 kernel 进行计算。

```
        gaussian_kernels = self.computeGaussianKernel()
        gaussian_imgs = []
        next_octave_base = image
        for i_octave in range(octaves_num):
            octave_image = [next_octave_base]
            for gaussian_sigma in gaussian_kernels[1:]:
                next_octave_base = cv2.GaussianBlur(next_octave_base, (0,
0), sigmaX=gaussian_sigma,
                                                    sigmaY=gaussian_sigma)
                octave_image.append(next_octave_base)
```

```
            gaussian_imgs.append(octave_image)
            next_octave_base = octave_image[-3]
            next_octave_base = cv2.pyrDown(next_octave_base)
        return np.array(gaussian_imgs)
```

上面代码中红框框出的部分为-3，因为降采样时，高斯金字塔一组的初始图像来自于前一组的倒数第 3 张图像。不妨设当前组数为o，当前组内层数为s，为了使每组具有S层 DOG 图像，高斯图像每组应有$S+2$个图像，因此s的取值范围为$1\sim S+2$。因此每组倒数第 3 张图像即为第S张图像，一组的初始图像等于前一组的倒数第 3 张图像，公式如下：

$$\sigma(o,s)=\sigma_0*2^{o+\frac{s}{S}},\ s=0,1,\dots,S+2$$

$$\sigma(o,0)=\sigma_0*2^{o+\frac{0}{S}}=\sigma_0*2^{o}=\sigma_0*2^{(o-1)+1}=\sigma_0*2^{(o-1)+\frac{S}{S}}=\sigma(o-1,S)$$

3.2.4 computeGaussianKernel 函数

不同组相同层的组内尺度坐标$\sigma(s)$相同。若 DOG 图像组内层数为 S，则高斯核数应为 S+3，因为 DOG 图像是高斯图像两层相减得到的。从 3.2.3 的公式中可以看出，高斯核中的相邻两个数相差$k=2\text{^}\{1/S\}$倍。因此循环计算需要在前一个的基础上模糊的 σ 即可。

```
        gaussian_intervals_num = self.num_intervals + 3
        gaussian_kernels = np.zeros(gaussian_intervals_num)
        gaussian_kernels[0] = self.sigma
        # σ(o, s) = σ_0 * 2^{s/S} , s = 0, 1, ... ,S+2
        k = 2 ** (1 / (gaussian_intervals_num - 3))  # k = 2^{1/S}
        sigma_previous = self.sigma
        for s in range(1, gaussian_intervals_num):
            sigma_new = k * sigma_previous
            gaussian_kernels[s] = np.sqrt(sigma_new ** 2 - sigma_previous
** 2)
            sigma_previous = sigma_new
        return gaussian_kernels
```

尺度空间里的每一层的图像（除了第 1 层）都是由其前面一层的图像和一个相对 σ 的高斯滤波器卷积生成，而不是由原图和对应尺度的高斯滤波器生成的，这一方面是因为没有“原图”，输入图像 I(x,y)已经是尺度为 σ=0.5 的图像了；另一方面是由于如果用原图计算，那么相邻两层之间相差的尺度实际上非常小，这样会造成在做高斯差分图像的时候，大部分值都趋近于 0，以致于后面很难检测到特征点。

3.2.5 buildDoGPyramid 函数

该函数作用为计算 DOG 图像金字塔。DOG 图像使用相邻的两个高斯图像做差即可。

```
for gaussian_octave_image in gaussian_imgs:
            dog_octave_image = []
            for i_internal in range(len(gaussian_octave_image) - 1):
```

```
                img_diff = cv2.subtract(gaussian_octave_image[i_internal],
gaussian_octave_image[i_internal + 1])
                dog_octave_image.append(img_diff)
            dog_imgs.append(dog_octave_image)
```

3.2.6 findScaleSpaceExtrema 函数

该函数的作用为初步探测极值点，邻域为同层的 8 个点、上层的 9 个点、下层的 9 个点，共 26 个点。使用两层循环，处理每个组以及每个组中的层之间的 DOG 图像，获得关键点。

```
        for i_octave in range(len(dog_imgs)):
            dog_octave_img = dog_imgs[i_octave]
            for i_interval in range(len(dog_octave_img) - 2):
                img1 = dog_octave_img[i_interval]
                img2 = dog_octave_img[i_interval + 1]
                img3 = dog_octave_img[i_interval + 2]
                for i in range(self.image_border_width, img1.shape[0] -
self.image_border_width):
                    for j in range(self.image_border_width, img1.shape[1]
- self.image_border_width):
                        if self.isScaleSpaceExtrema(img1, img2, img3, i,
j):
                            kp_info.append([i_octave, i_interval, i, j])
```

如代码中框出的部分，边缘区域的 5 个像素范围不用来被检测关键点。
该函数非常耗时，需要若干小时才能得到全部极值点。

3.2.7 isScaleSpaceExtrema 函数

该函数在 findScaleSpaceExtrema 函数中被调用，参数为 img1, img2, img3, i, j；作用为判断 img2 的(i, j)位置的点是否是 3.2.6 中提到的极值点。
该函数唯一值得一提的是阈值计算。阈值 threshold 用于判断 DOG 图像中像素值的绝对值是否足够大。此处阈值的选取参考了 opencv 的实现方式。将像素可能的最大值乘以 $\frac{0.5*const_{threshold}}{num_intervals}$作为阈值，小于该阈值的点不能被当作关键点。此处的 num_intervals 为 3，const_threshold 为 0.04，像素可能的最大值为 255。

```
threshold = np.floor(0.5 * self.const_threshold / self.num_intervals * 255)
```

3.2.8 adjustLocalExtrema 函数

该函数作用为精确定位特征点，实现方法基于 Lowe 论文的第四部分，参数均按照 Lowe 论文中建议的参数设置，也部分参考了 opencv 的实现方式。
进行 5 次调整尝试，每次都计算出该点的梯度、二阶导数，得到 hessian 矩阵。然后使用线性回归得到调整的值 xr,xc,xi。如果调整的像素小于 0.5，表示已经收敛，可以直接退出尝试。否则，将调整像素大小加到原本的值中，并判断是否超出 cube 范围，然后进行下次尝试。

如果超出范围，直接返回 None，表明该处不存在极值点。如果进行完 5 次尝试仍未收敛，也表明此处不存在极值点，直接返回 None。
如果存在极值点，则更新极值点处的函数值，然后消除边缘响应。首先保证行列式的值大于 0，因为行列式等于 0 说明有为 0 的特征值，小于 0 说明两个特征值为一正一负，不符合对特征值描述函数变化快慢的性质的要求。然后判断变换响应，Lowe 的论文中建议将 r 设为 10。最后将得到的关键点信息整理，按照 opencv 中的格式进行存储。

```
        if det <= 0 or r * (tr * tr) >= ((r + 1) ** 2) * det:
            return None
        keypoint = cv2.KeyPoint()
        keypoint.pt = ((j + xc) * (1 << i_octave),
                       (i + xr) * (1 << i_octave))
        keypoint.octave = i_octave + i_img * (1 << 8) + int(round((xi + 0.5)
* 255)) * (1 << 16)
        keypoint.size = self.sigma * (2 ** ((i_img + xi) /
np.float32(self.num_intervals))) * (1 << (i_octave + 1))
        keypoint.response = abs(contr)
```

如红框部分所示，此处应为 i_octave+1，因为输入图像被扩大了，是原来的 2 倍。

3.2.9 calcOrientationHist 函数

该函数的作用为计算梯度方向直方图。这里也部分参考了 opencv 的实现方式，因为高斯分布的概率函数 99.7%位于 3 个标准差内，因此这里只统计 3 σ 之内的，即 radius 为 3 σ 。在计算 kpt.size 时，使用了相对于第 0 层扩大一倍后的初始图象，因此需要乘 0.5，再除以2^{i_o}。

```
        scale = 1.5 * keypoint.size * 0.5 / (1 << i_octave)
        x_center = round(keypoint.pt[0] / (1 << i_octave))
        y_center = round(keypoint.pt[1] / (1 << i_octave))
        radius = round(3 * scale)
```

使用 36 个方向的直方图，并使用调整后的关键点所在的 dog 图像层，此处不可以使用调整前的。
使用双重循环对位于关键点周围 3 σ 范围内的点进行统计，计算其一阶导数，高斯加权，并存储。使用高斯加权可以使离关键像素较远的 pixel 影响较小。

```
        for i in range(-radius, radius + 1):
            y = y_center + i
            if y < 0 or y >= gaussian_img.shape[0] - 1:
                continue
            for j in range(-radius, radius + 1):
                x = x_center + j
                if x <= 0 or x >= gaussian_img.shape[1] - 1:
                    continue
                dx = gaussian_img[y, x + 1] - gaussian_img[y, x - 1]
                dy = gaussian_img[y - 1, x] - gaussian_img[y + 1, x]
                weight = np.exp((i * i + j * j) * -1.0 / (2 * scale * scale))
                grad_collect.append([dx, dy, weight])
```

对于刚才存储的 grad_collect，遍历统计邻域中的所有像素，计算直方图，将其梯度从 360 映射到 36 个 bin 中，通过取余实现循环处理。然后计算平滑后的直方图，使用了 36 个 bin，每个 bin 代表 10 度，需要对峰值位置进行插值。最接近每一个峰值的直方图上的 3 个值拟合成抛物线。

```
    for dx, dy, weight in grad_collect:
        # 邻域中的所有像素
        grad_orientation = np.rad2deg(np.arctan2(dy, dx))
        histogram_index = int(round(grad_orientation * 36 / 360))
        # 通过取余实现循环处理
        raw_hist[histogram_index % 36] += np.sqrt(dx * dx + dy * dy) *
weight
    for n in range(36):
        smooth_hist[n] = (raw_hist[n - 2] + raw_hist[(n + 2) % 36]) /
16 + (
                raw_hist[n - 1] + raw_hist[(n + 1) % 36]) / 4 + 3 *
raw_hist[n] / 8
```

最后为每个辅方向创建一个关键点。辅方向为大于 0.8 倍主方向的所有 bin，Lowe 在论文中提到，这些额外的关键点在实际应用中显著有助于检测稳定性，因此该步骤是不可或缺的。辅方向除角度外，所有参数均与原关键点相同。

3.2.10 calcSIFTDescriptors 函数

该函数的作用为计算生成描述子。描述符对关键点邻域的信息进行编码，Lowe 提出当梯度方向直方图是 4*4 维时，SIFT，描述符具有很高的区分度。与 3.2.9 中不同，此处的每个梯度直方图有 8 个方向，一个方向代表 45 度。因此，每个描述符的大小为 4*4*8=128，直方图宽度为 4。将特征点附近邻域划分成 4*4 个子区域，每个子区域的尺寸为 3 σ ， σ 为当前特征点的尺度值。考虑到实际计算时，需要采用双线性插值，因此计算的图像区域应大于 4；考虑到旋转，应再乘根号 2。半径为图像区域除以 2，即为

$$radius = 3\sigma * (4 + 1) * \sqrt{2}/2$$

对于每个关键点，先提取当前特征点的所在层/组/尺度，并算出角度，计算出半径 radius。同时，也应确保半径小于图像斜对角线长度。

```
hist_width = 1.5 * scale * keypoint.size
radius = round(hist_width * np.sqrt(2) * (window_width + 1) * 0.5)
radius = min(radius, np.sqrt(gaussian_image.shape[0] ** 2 +
                                        gaussian_image.shape[1] ** 2))
```

使用两层循环，对上述 radius 范围内的所有点进行计算。为保持旋转不变性，需要先将坐标轴旋转为关键点方向，然后把邻域区域的原点从中心位置移到该区域的左下角（即加 0.5 倍的 window_width）,减 0.5 是为了进行坐标平移。

```
row_rot = col * sin_angle + row * cos_angle
col_rot = col * cos_angle - row * sin_angle
row_bin = (row_rot / hist_width) + 0.5 * window_width - 0.5
col_bin = (col_rot / hist_width) + 0.5 * window_width - 0.5
```

然后进行差分求梯度，计算 x 和 y 的一阶导数。这里省略了分母 2，是因为没有分母不影响后面的归一化。分别计算梯度幅值、梯度辐角与高斯加权函数，并存储。

```
dx = gaussian_image[window_row, window_col + 1] - gaussian_image[window_row,
window_col - 1]
dy = gaussian_image[window_row - 1, window_col] - gaussian_image[window_row
+ 1, window_col]
gradient_magnitude = np.sqrt(dx * dx + dy * dy)
gradient_orientation = np.rad2deg(np.arctan2(dy, dx)) % 360
weight = np.exp(
    exp_scale * ((row_rot / hist_width) ** 2 + (col_rot / hist_width) ** 2))
row_list.append(row_bin)
col_list.append(col_bin)
magnitude_list.append(weight * gradient_magnitude)
orientation_list.append((gradient_orientation - angle) * bins_per_rad)
```

根据上面计算出的梯度幅值、梯度辐角、高斯加权函数的值，再次遍历，此处的实现也参考了 opencv 的实现思路。取三维坐标的整数部分，判断在 4*4*8 区域中属于哪个正方体。然后再取小数部分。将 0-360 度以外的角度按照圆周循环调整至 0-360。

```
    row_bin_floor = np.floor(row_bin).astype(int)
    col_bin_floor = np.floor(col_bin).astype(int)
    orientation_bin_floor = np.floor(orientation_bin).astype(int)
    row_fraction = row_bin - row_bin_floor
    col_fraction = col_bin - col_bin_floor
    orientation_fraction = orientation_bin - orientation_bin_floor
    if orientation_bin_floor < 0:
        orientation_bin_floor += num_bins
    if orientation_bin_floor >= num_bins:
        orientation_bin_floor -= num_bins
```

按照三线性插值法，计算像素对正方体的 8 个顶点的贡献大小。

```
r1 = magnitude * row_fraction
r0 = magnitude - r1
rc11 = r1 * col_fraction
rc10 = r1 - rc11
rc01 = r0 * col_fraction
rc00 = r0 - rc01
rco111 = rc11 * orientation_fraction
rco110 = rc11 - rco111
rco101 = rc10 * orientation_fraction
rco100 = rc10 - rco101
rco011 = rc01 * orientation_fraction
rco010 = rc01 - rco011
rco001 = rc00 * orientation_fraction
rco000 = rc00 - rco001
```

得到像素点在三维直方图中的索引。

```
ori_plus = (orientation_bin_floor + 1) % num_bins
histograms[row_bin_floor + 1, col_bin_floor + 1, orientation_bin_floor] +=
rco000
histograms[row_bin_floor + 1, col_bin_floor + 1, ori_plus] += rco001
histograms[row_bin_floor + 1, col_bin_floor + 2, orientation_bin_floor] +=
rco010
histograms[row_bin_floor + 1, col_bin_floor + 2, ori_plus] += rco011
histograms[row_bin_floor + 2, col_bin_floor + 1, orientation_bin_floor] +=
rco100
histograms[row_bin_floor + 2, col_bin_floor + 1, ori_plus] += rco101
histograms[row_bin_floor + 2, col_bin_floor + 2, orientation_bin_floor] +=
rco110
histograms[row_bin_floor + 2, col_bin_floor + 2, ori_plus] += rco111
```

最后进行归一化，将大于 0.2 的元素设置为 0.2。为避免累计误差，使用 0.2 乘平方和开方值，得到反归一化阈值。再次遍历数组，将大于阈值的元素替换为阈值，进行归一化，并将浮点数转化为整型。由于要将数据保存到 uint8 中，因此需要乘 512。

```
thr = np.linalg.norm(descriptor_vector) * descriptor_max_value
descriptor_vector[descriptor_vector > thr] = thr
# 从float32 转化到 unsigned char
descriptor_vector /= max(np.linalg.norm(descriptor_vector),
self.float_epsilon)
descriptor_vector = np.round(512 * descriptor_vector)
descriptor_vector[descriptor_vector < 0] = 0
descriptor_vector[descriptor_vector > 255] = 255
descriptors.append(descriptor_vector)
```

3.3 图像融合部分

3.3.1 compute_ratio_V 函数

该函数的作用为计算左图与右图的 V 分量的比例。将两幅图像从 RGB 转为 HSV，然后拆分出 V 分量，计算 V 分量的和，返回二者比例。

```
def compute_ratio_V(left, right):
    right_hsv = cv2.cvtColor(right, cv2.COLOR_BGR2HSV)
    right_v = computeV(right_hsv)
    left_hsv = cv2.cvtColor(left, cv2.COLOR_BGR2HSV)
    left_v = computeV(left_hsv)
    bright_k = left_v / right_v
    return bright_k
```

3.3.2 偏移量 offset

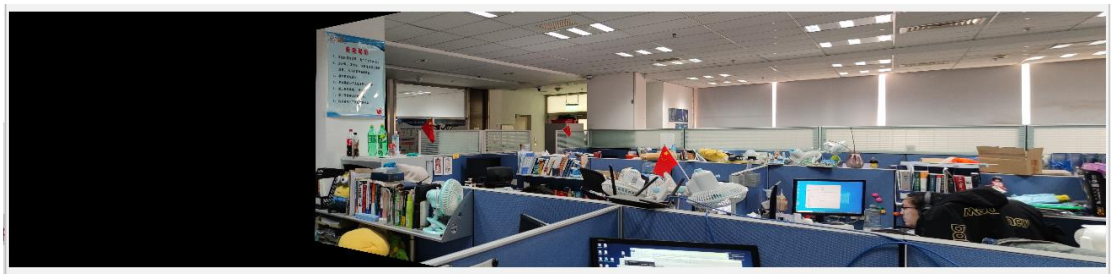
以左图为基准，将右图进行仿射变换。检测变换后的右图的非零值，并取出最小的非零值所在的列，记为 min_index。再算出该图中最大非零值所在的列，记为 max(col)。

```
right_mask_img = cv2.warpPerspective(right, H, (mix_width, mix_height))
zero_test = np.nonzero(right_mask_img)
min_index = zero_test[0][0]
row, col = np.where(right_mask_img[:, :, 0] == 0)
print(max(col))
```

将 offset 初始化为 600，判断计算出的 V 分量比例。若两幅图 V 分量差别不大，则将 offset 设置为 40；否则，计算如果 max(col)离左图边界太近或者超出了左图边界，如下图情况所示：

此时需要采用左侧第一个非零值 min_index 来计算 offset，将左图的 left_width 与 min_index 的差乘 0.15 作为 0ffset。

如果 max(col)距离左图边界较远，说明右侧没有黑边，如下图情况所示：



此时的掩码 left_width 与 max(col)的差的 0.2 倍。

```
offset = 600
if 0.95 < bright_k < 1.05:
    offset = 40
elif max(col) > (left_width - 400):
    zero_test = np.nonzero(right_mask_img)
    min_index = zero_test[0][0]
    offset = (left_width - min_index) * 0.15
else:
    offset = (left_width - max(col)) * 0.2 + 1
offset = int(offset)
print(offset)
```

3.3.3 掩码 mask

将 left_width-2*offset 到 left_width 区域的范围作为过渡区域。过渡区域左侧，采用左图的值，将左图掩码设置为 1，右图掩码设置为 0。过渡区域右侧，采用右图的值，将左侧掩码设置为 0，右侧掩码设置为 1。在过渡区域处，左侧掩码的值从 1 到 0 逐渐变化，右侧掩码的值从 0 到 1 逐渐变化，从而使两幅图像的融合比较平滑不突兀。计算出掩码后，将其变为 3 通道的，并将左图与右图分别与其掩码相乘。

```
    barrier = left_width - offset
    mask = np.zeros((mix_height, mix_width))
    mask[:, barrier - offset:barrier + offset] = np.tile(np.linspace(1, 0,
2 * offset), (mix_height, 1))
    mask[:, :barrier - offset] = 1

    left_mask = np.stack((mask, mask, mask), axis=2)
    left_mask_img[0:left_height, 0:left_width, :] = left
    left_mask_img = left_mask_img * left_mask
    mask2[:, barrier - offset:barrier + offset] = np.tile(np.linspace(0, 1,
2 * offset), (mix_height, 1))
    mask2[:, barrier + offset:] = 1

    right_mask = np.stack((mask2, mask2, mask2), axis=2)
    right_mask_img = right_mask_img * right_mask
```

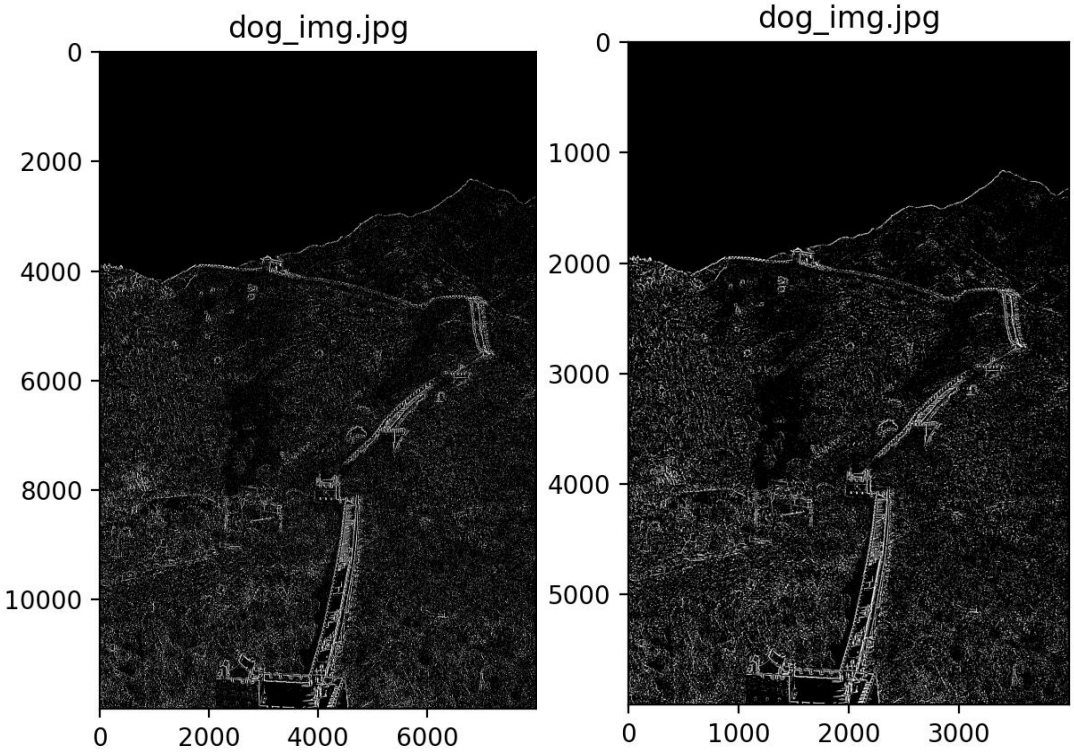
最后将得到的两幅图像相加，并裁剪掉右侧的黑边即可。

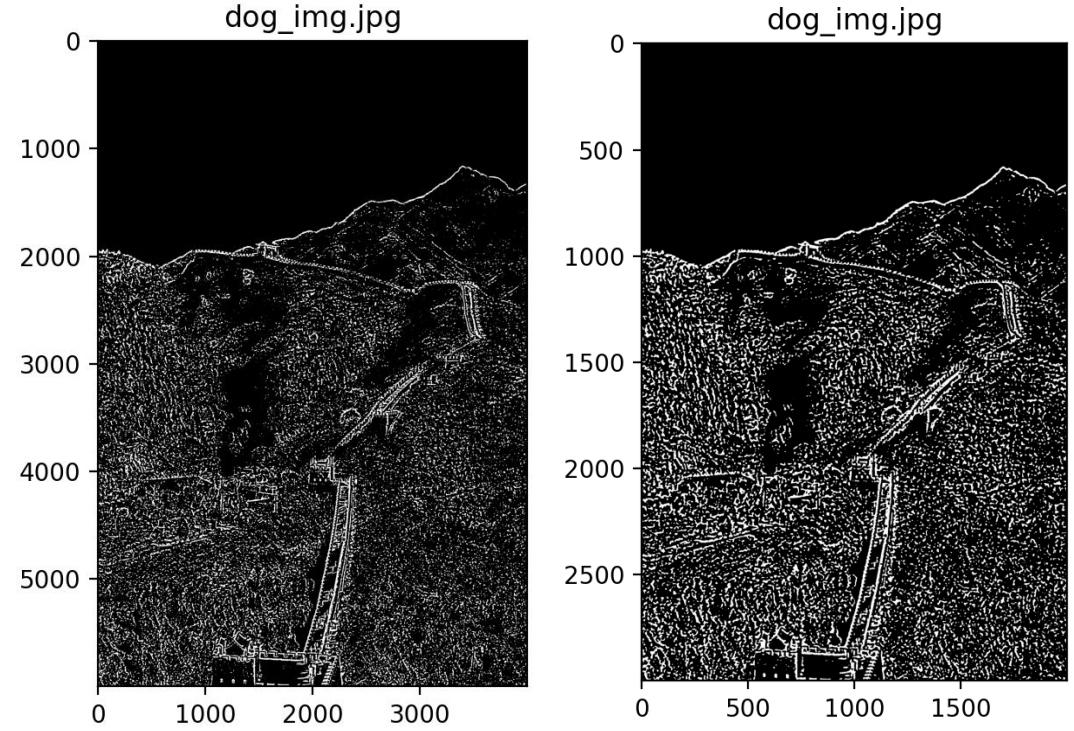
```
    mix_img = left_mask_img + right_mask_img
    rows, cols = np.where(mix_img[:, :, 0] != 0)
    output_img = mix_img[min(rows):max(rows), min(cols):max(cols), :]
    result_path = "./result/output" + img_seq + ".jpg"
    cv2.imwrite(result_path, output_img)  # 写入文件
```

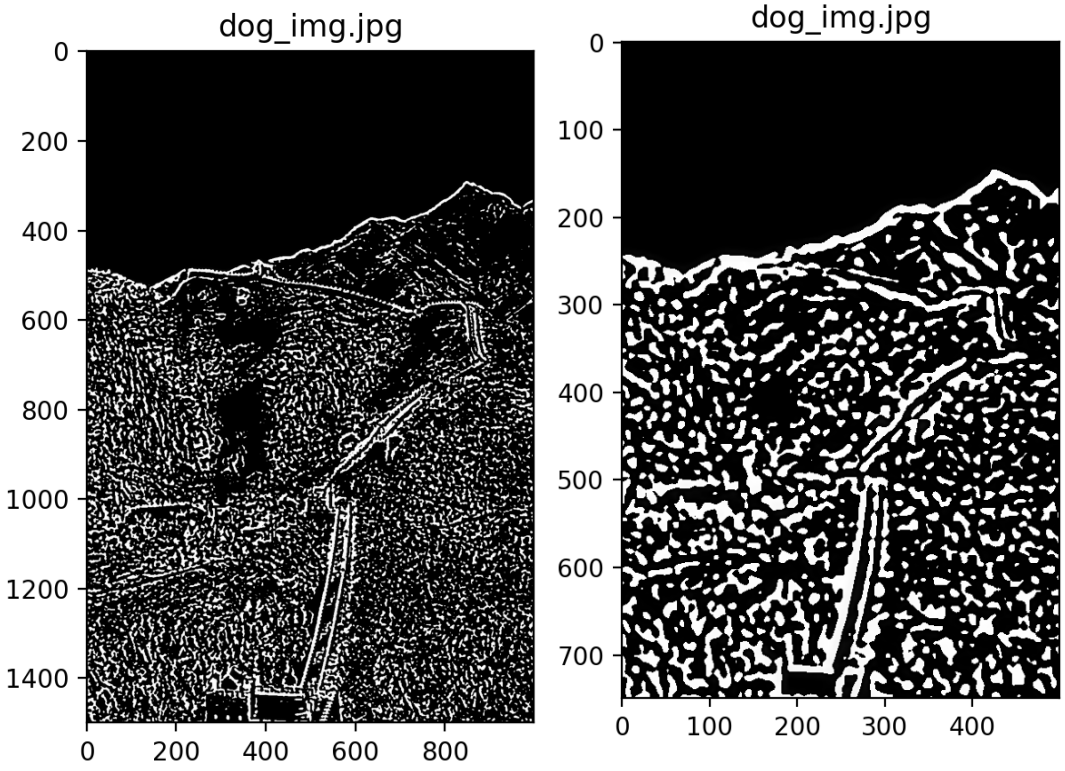
4. 实验结果

4.1 SIFT 算法中间结果

查看 SIFT 中间过程中得到的 DOG 图像，可以看出 DOG 图像中能显示出原图的轮廓特征。在同组的 DOG 图像中，得到的图像中的轮廓线会随层数增大而变得更加明显；在不同组的 DOG 图像中，随组数增大，DOG 图像的轮廓线变得越来越粗，看到的细节越来越少，可以体现出人在距离目标由近到远时目标在视网膜上的形成过程，从而使 SIFT 特征具有尺度不变性。这里使用的原始图像为 left1.jpg 图像，下面列出的 6 个图像为不同组不同层的 DOG 图像：


dog_img.jpg
0
2000
4000
6000
8000
10000
0
2000
4000
6000
dog_img.jpg
0
1000
2000
3000
4000
5000
0
1000
2000
3000


dog_img.jpg
0
1000
2000
3000
4000
5000
0
1000
2000
3000
dog_img.jpg
0
500
1000
1500
2000
2500
0
500
1000
1500



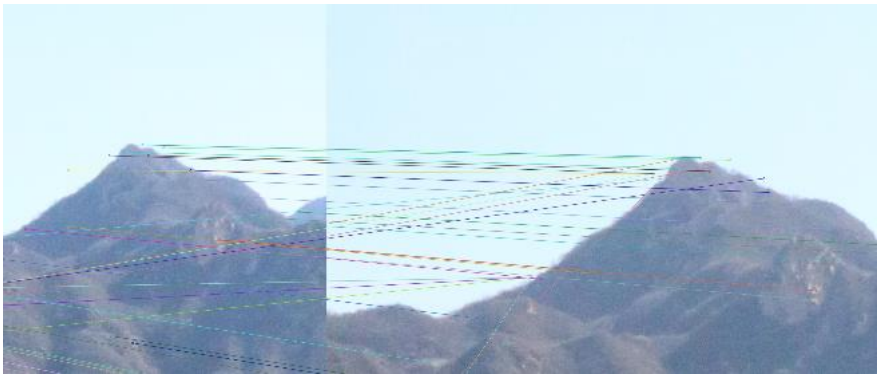
将图像中探测到的关键点绘制到图像上，用彩色圆圈圈出的即为关键点。部分区域如下图所示：

4.2 关键点匹配结果

将得到的左图与右图的 SIFT 特征点进行匹配，匹配结果存储在 middle_res 文件夹中。其中，

left1 与 right1 图像的匹配结果如下图所示：

查看其比较明显的山头位置，可以看出这里虽然有错配的部分点，但整体结果还是非常好的。



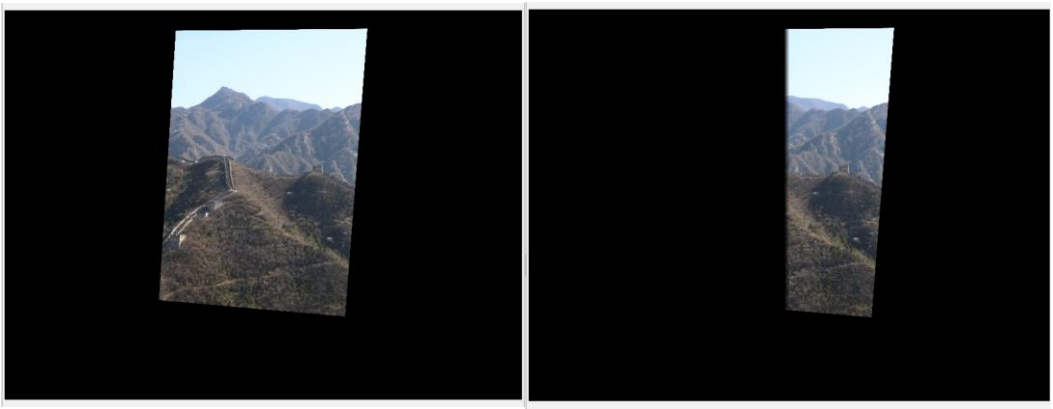
其他图像的匹配结果也都存储在 middle_res 文件夹下。

4.3 图像融合结果

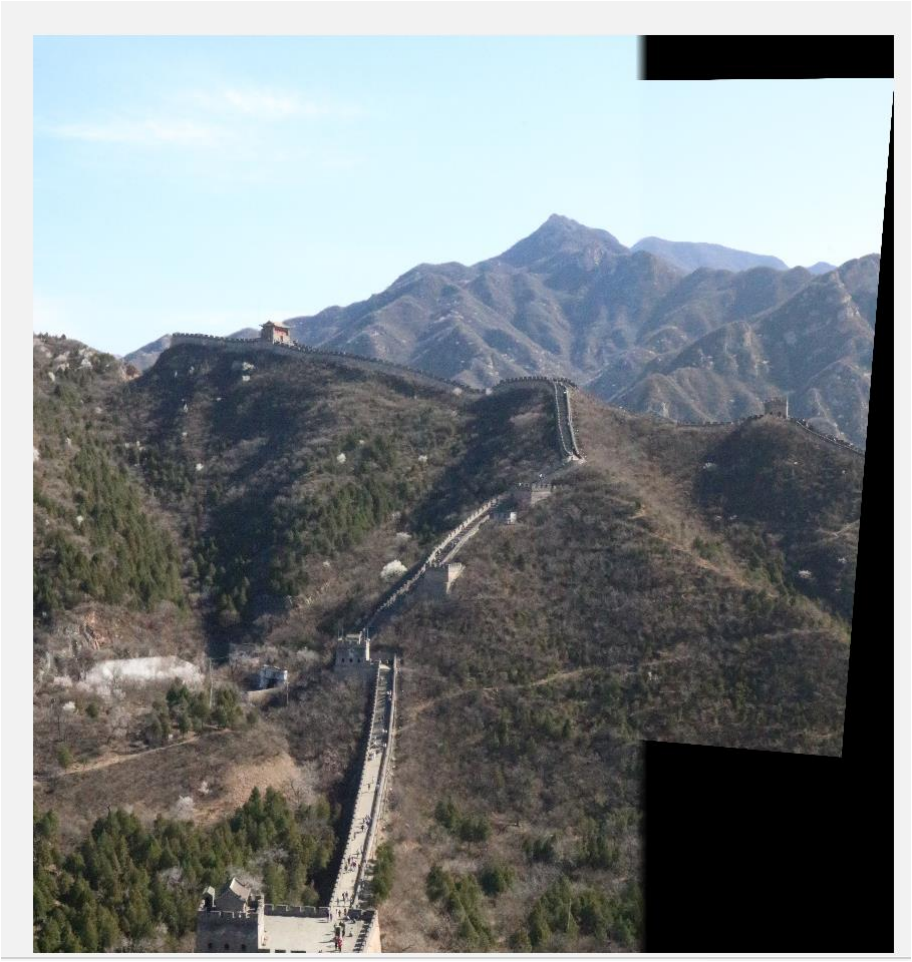
图像融合前需要先根据匹配结果做仿射变换，然后判断其融合方式，选择合适的 offset 进行融合。

4.3.1 left1 与 right1

如下图所示，第一对图像的左图与右图的 V 分量差别不大，因此融合时可以使 offset 较小，以更多地保留图像信息，并且可以尽可能避免出现重影现象。原图与乘上掩码 mask 得到的图像如下图所示：

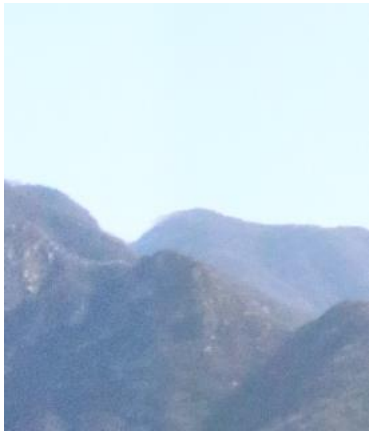


将左侧图像与右侧图像相加，并裁剪掉右侧黑边，得到的最终结果为



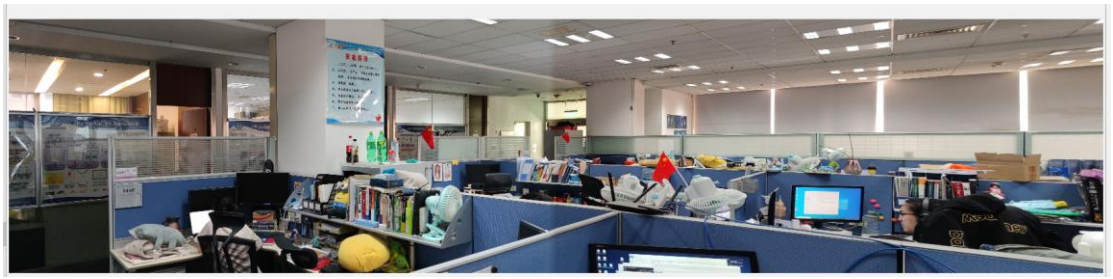
由于 left_width 左侧基本全部采用了左图，并未采用右图与左图融合，因此最终结果的山头处、长城处均未出现重影，过渡区域十分自然。

放大查看图像的细节，两幅图片过渡区域的山脊、下面的一条小山路也都没有出现错位的现象，整体拼接效果较好。下图展示了融合区域的两处细节：



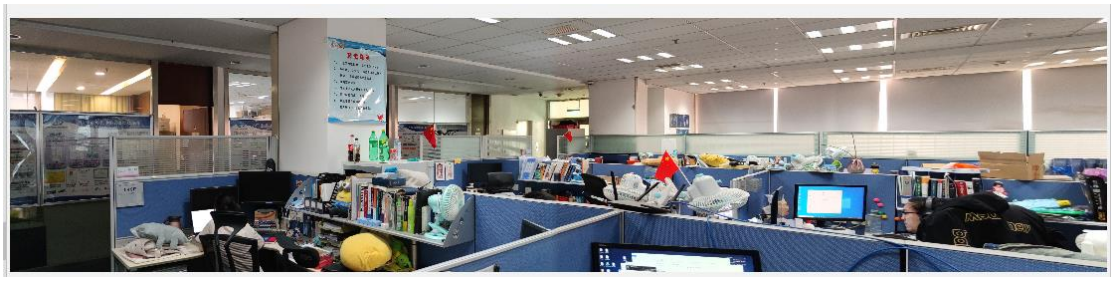

4.3.2 left2 与 right2

左图与右图的 V 分量比值为 0.9，差别较大，因此过渡区域不可过小，否则边界线会十分明显。将 offset 设为 40，拼接结果如下图所示：



从图中可以明显看出，左侧灯光较亮，右侧整体灯光较暗。

按照 2.3 节中提到的第一种光照处理方法进行处理，结果如下图所示：

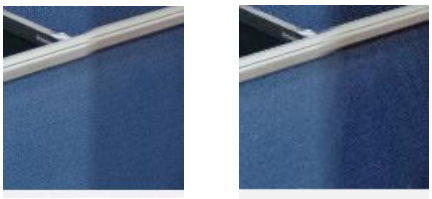


该方法与直接拼接相比，效果有所改善，但接缝处的色差依然明显，放大即可看出颜色差距。下面的左图为直接拼接接缝处色差，右图为采用第一种光照处理方法处理后效果：

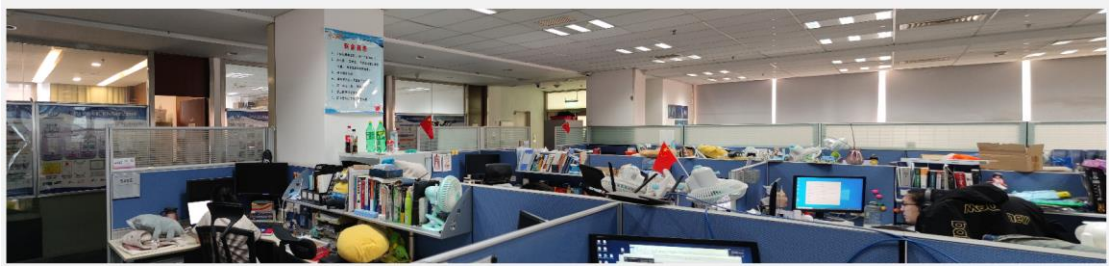
按照 2.3 节中提到的第二种光照处理方法进行处理，结果如下图所示：

如图所示，直方图匹配后，左侧与右侧差异更加明显，接缝处的色差也变大了。

下面的左图为直接拼接接缝处色差，右图为采用第二种光照处理方法处理后效果：

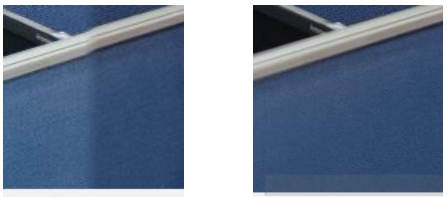


不进行光照处理，增大融合过渡区域 offset，按照 2.4 中提到的融合方法进行拼接，得到的拼接结果为



整体来看，拼接痕迹不太明显，左侧与右侧光照虽有差异，但过渡十分自然，远看难以识别出这是拼接图像。

下面的左图为原本的拼接接缝处色差，右图为最终得到的拼接接缝处效果：



接缝处已经看不出拼接痕迹，效果非常好。

4.3.3 left3 与 right3

左图与右图的 V 分量比值为 1.11，差别较大，因此过渡区域不可过小，否则边界线会十分明显。将 offset 设为 40，直接进行拼接，拼接结果如下图所示：

从图中可以明显看出，右侧图像整体发白，左侧图像对比度更高。

采用第一种光照处理方法进行处理，拼接得到的结果为：

从图中可以看出，原本右侧图像较亮，调整过后右侧图像变为较暗，整体差距仍然十分明显，效果不好。

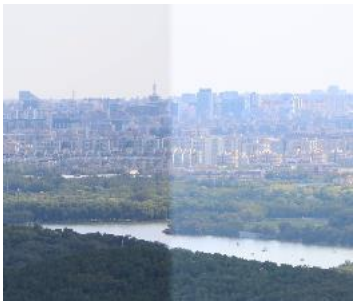
采用第二种光照处理方法进行处理，拼接得到的结果为：

右侧图像出现了失真，因此这种光照处理方法不可行。

不进行光照处理，增大融合过渡区域 offset，按照 2.4 中提到的融合方法进行拼接，得到的拼接结果为

虽然最右侧看起来依然有些发白，不如左侧图像清晰，但整体效果看起来还算自然，拼接痕迹不是非常明显，远看也难以辨别出该图为拼接图像。

下面的左图为原本的拼接接缝处色差，右图为最终得到的拼接接缝处效果：



此时的接缝处也很难分辨出拼接痕迹。

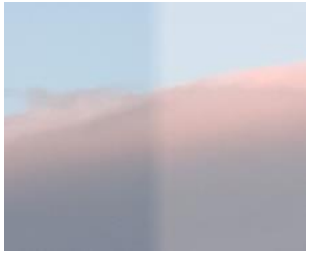
4.3.4 left4 与 right4

左图与右图的 V 分量比值为 0.9，差别较大，因此过渡区域不可过小，否则边界线会十分明显。将 offset 设为 40，拼接结果如下图所示：

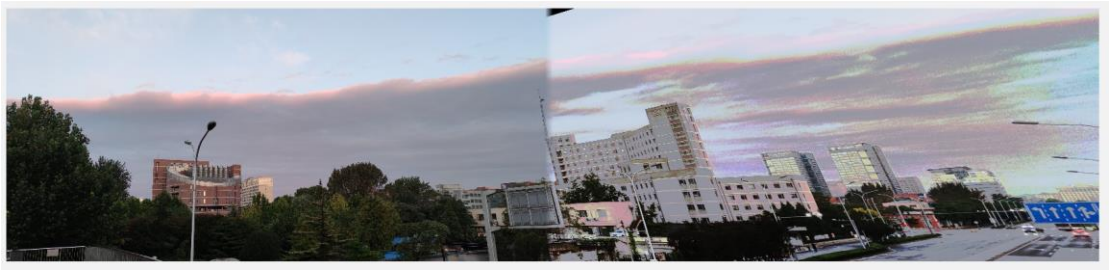
可以明显看出，左侧图片偏蓝，右侧图像有些发黄。

采用第一种光照处理方法进行处理，拼接得到的结果为：

可以看出，处理后右侧图像只是稍微变暗，但两侧仍一个偏蓝一个偏黄。处理前确实右侧图像更亮，因此调整了右侧图像的亮度，让其变暗一些，也确实做到了。但这幅图像的问题在于右侧图像发黄，因此这样处理不能解决问题。
下面的左图为直接拼接接缝处细节，右图为采用第一种光照处理方法处理后效果：

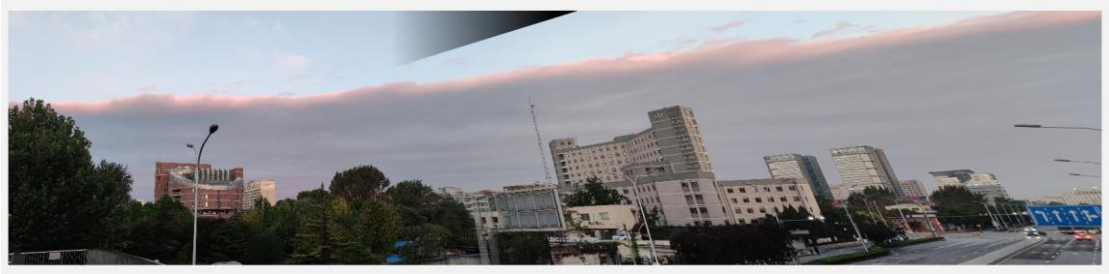



采用第二种光照处理方法进行处理，拼接得到的结果为：



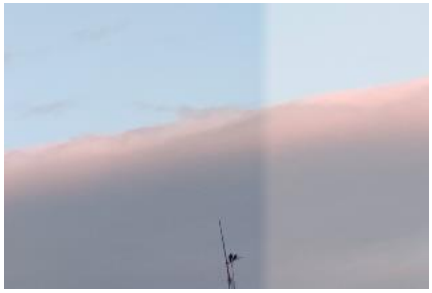
右侧图像也出现了失真。

不进行光照处理，增大融合过渡区域 offset，按照 2.4 中提到的融合方法进行拼接，得到的拼接结果为



整体来看，依然存在右侧发黄左侧偏蓝的问题，但拼接痕迹不太明显，左侧与右侧虽有差异，但过渡十分自然。

下面的左图为原本的拼接接缝处色差，右图为最终得到的拼接接缝处效果：




接缝处过渡非常自然，完全看不出拼接痕迹。

5. 实验总结

5.1 光照处理失败分析

在本次实验中，使用的 2 种光照处理方法效果都不太好。第一种光照处理方法对第二组图片取得了一定效果，但对第三组、第四组图片均造成了负面效果，拼接痕迹更加明显。第二种光照处理方法使第二组图片左右差别更大，第三组、第四组图片的右图出现了失真。
第一种方法为调整 V 分量，并对右侧图像进行补偿。但 RGB 图像转 HSV 图像时，V 分量其实为 Cmax，用 V 分量之和衡量整张图片亮度的做法未必是合理的。比如在第三组图像中，左图的视觉效果比右图暗，但算出二者的 V 分量比例却大于 1，从 V 分量来看应当是左图比右图亮。且补偿方法也比较粗糙，如第三组图像中，明显过渡补偿，将右侧比左侧看起来亮变为右侧比左侧看起来更暗。其次，一些图像左侧右侧的差别不单单是光照，而是整体的色调都有差别，如第四组图像，单纯调节 V 分量作用有限。这三种原因造成了第一种光照处理方法的失败。
第二种方法为将左图与右图进行直方图匹配，导致出现了失真，图片颜色明显不正常。理论上，使用直方图匹配可以使两幅图像的色调更一致，使两张图的视觉感觉接近，但在本次实际使用中其效果并不好。

5.2 算法优缺点与适用范围

5.2.1 特征点寻找

本次实验使用的寻找特征点的 SIFT 算法的优缺点已经在 2.1 节中介绍过了。我实现的 SIFT 的算法的主要缺点为计算速度过慢，计算一张图片需要若干小时，要等待很长时间。如果将其中的一些步骤改为并行处理，运算速度可能会大大提升。

5.2.2 图像融合方法

由于没有找到合适的光照处理方法，最终我没有处理光照，只是调整了图像的融合方法，根据带拼接图像的特点判断合适的过渡区域大小。这样做的优点是不处理光照也可以得到比较自然的图像，过渡区域几乎看不出拼接痕迹，整体效果还不错。当两侧图像亮度接近时，过渡区域尽可能小，以避免重合部分过多出现重影。但缺点在于，当两侧图像 V 分量之和接近，整体光照不同时，该方法并不能很好的判断并给出最优的 offset。

5.3 实验感想

整个图像拼接实验断断续续写了将近两个月才终于完成。这里面花费时间最多的部分是实现 SIFT 算法。SIFT 算法实现的原理似乎不困难，但真正动手实现时才发现里面有许多细节问题都需要仔细思考。

6. 参考资料

https://ieeexplore.ieee.org/abstract/document/790410
https://link.springer.com/article/10.1023/B:VISI.0000029664.99615.94
https://github.com/opencv/opencv
https://dezeming.top/wp-content/uploads/2021/07/Sift%E7%AE%97%E6%B3%95%E5%8E%9F%E7%90%86%E4%B8%8EOpenCV%E6%BA%90%E7%A0%81%E8%A7%A3%E8%AF%BB.pdf