

计算机视觉-HW2

xTyer

2024/10/24

1 Harris Corner Detector

1.1 Calculate spatial derivative

思路：先判断图片通道数，若图片为三通道则转成灰度图，然后将图片转为 numpy 数组进行操作。先给图片加高斯模糊去噪，使梯度平滑，再对 x,y 方向分别用对应的 sobel kernel 进行卷积操作。

```
1 def gradient_x(img):
2     # TODO
3     gray_img_cv=img
4     if gray_img_cv.shape[2]==3: #三通道BGR2一通道Gray
5         gray_img_cv = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
6
7     gray_img_cv=np.float64(gray_img_cv)
8     blur_gray_img_array = ndimage.gaussian_filter(gray_img_cv,sigma=3.0,mode='reflect')
9     #高斯模糊 增加连续性
10
11    kernel_x = np.array([[1,0,-1],
12                          [2,0,-2],
13                          [1,0,-1]]) # sobel operation
14
15    grad_x_img_array = ndimage.convolve(blur_gray_img_array,kernel_x)
16
17
18
19 def gradient_y(img):
20     # TODO
21     gray_img_cv=img
22     if gray_img_cv.shape[2]==3: #三通道BGR2一通道Gray
23         gray_img_cv = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
24     gray_img_cv=np.float64(gray_img_cv)
25     blur_gray_img_array = ndimage.gaussian_filter(gray_img_cv,sigma=3.0,mode='reflect')
#高斯模糊 增加连续性
```

```

26     kernel_y = np.array([[1,2,1],
27                           [0,0,0],
28                           [-1,-2,-1]]) # sobel operation
29     grad_y_img_array = ndimage.convolve(blur_gray_img_array,kernel_y)
30
31     return grad_y_img_array

```

1.2 Calculate Harris response

思路: 先得到x,y 方向的梯度分布, 然后以 win_size 作为高斯核的大小, 以保证梯度的平滑性, 分别求出 $I_{x^2}, I_{y^2}, I_x I_y$ 这三个用于方便得到 $\det(M)$ 和 $\text{tr}(M)$ 的矩阵, 最后利用公式直接求出 harris_response 值。

```

1 def gauss_2d_calculate(k_sigma,center_point,tar_point):
2     distance_ = np.linalg.norm(center_point-tar_point)
3     return np.exp(-(distance_*distance_)/(2*k_sigma*k_sigma))
4
5     def gaussian_blur_kernel_2d(k_size,k_sigma=1):
6         gauss_kernel = np.zeros((k_size,k_size), dtype=np.float32)
7         mean_ = k_size//2
8         for i in range(0,k_size):
9             for j in range(0,k_size):
10                 gauss_kernel[i,j] =
11                     gauss_2d_calculate(k_sigma,np.array([mean_,mean_]),np.array([i,j]))
12
13     return gauss_kernel/np.sum(gauss_kernel)
14
15
16     def harris_response(img, alpha, win_size):
17         # TODO
18         img_x_grad_array = gradient_x(img)
19         img_y_grad_array = gradient_y(img)
20
21         Ix2 = gaussian_blur_2d(img_x_grad_array*img_x_grad_array,k_size=win_size,k_sigma=5)
22         Iy2 = gaussian_blur_2d(img_y_grad_array*img_y_grad_array,k_size=win_size,k_sigma=5)
23         IxIy = gaussian_blur_2d(img_x_grad_array*img_y_grad_array,k_size=win_size,k_sigma=5)
24
25         detM = Ix2*Iy2-IxIy**2
26         trM = Ix2+Iy2
27
28         output_harris_array=detM-alpha*(trM**2)
29

```

```
30     return output_harris_array
```

1.3 Select candidate corners and non-maximal suppression

思路：先以 min_dist 为 size 对传入的 Harris_Response_matrix 作最大值滤波，然后挑选出 Harris_response 比 threshold 大的点的坐标，为了简化逻辑，在遍历选出的点列时，舍去了边缘上的一些点，然后对于每个 local_maximum 我们以边长为 min_dist 的“box”去找，若这个 box 内的每个点的值都是该 Harris_response，那么我们可以判断这是一个符合条件（size 为 min_dist 的局部的大于 threshold 的最大值）的角点。

```
1 def corner_selection(R, thresh, min_dist):
2     # TODO
3     R_ = ndimage.maximum_filter(R, size=min_dist)
4     pixels_lst = []
5     select_points = np.argwhere(R_>thresh) # get pixel where local maximum
6
7     expand_=min_dist//2
8     R_height,R_width = np.shape(R_)
9     for point_index in range(len(select_points)):
10         center_y = select_points[point_index][0]
11         center_x = select_points[point_index][1]
12
13         if center_y<expand_ or center_y>R_height-expand_-1 or center_x<expand_ or
14             center_x>R_width-expand_-1:
15             continue
16
17         box_up = max(center_y-expand_,0)
18         box_down = min(center_y+expand_+1,R_height)
19         box_left = max(center_x-expand_,0)
20         box_right = min(center_x+expand_+1,R_width)
21
22         if(np.unique(R_[box_up:box_down,box_left:box_right]).size==1 and
23             np.sum(R_[box_up:box_down,box_left:box_right])!=0):
24             pixels_lst.append(tuple((center_x,center_y)))
25
26     return pixels_lst
```

2 Implement Histogram of Gradients

思路：我们首先求出 image 的 x 与 y 方向上的梯度图，计算图上每个点对应的梯度方向和梯度强度。

然后对于每一个传入的“pix”，我们求梯度直方图，求梯度直方图的方法如下：

首先，使用 get_surrounding_box_array 取一个周围的区域，区域的大小可以自由设置，由传入的参数 box_size 决定，最后返回的是一个 box_size*box_size 的 np.array

在我们取完一个 box 后，我们使用 divide_box_array 对这个 box 进行分割，分割成 divide_n*divide_n 个 block，返回的是一个 np.array，其中包含了 divide_n*divide_n 个 block

我们分别将梯度角度和梯度强度进行 box 以及 block 的划分，然后对每个 block 求梯度直方图。

对于每个 block，我们利用 angle_grad_histogram 进行梯度直方图的求解，具体操作是将传入的 block 中的元素的角度分布分成 8 个方向，对每个梯度角度叠加上梯度强度，然后以线性插值的方式把权值分配给对应的方向，最后返回 1*8 的 np.array 作为对应 block 的 feature.

在求解完所有 block 的 feature 后，把这些 feature 组合成为对应 pix 的 box 的 feature_vector 我们遍历组合成的 feature_vector 找到一个最大的权值，其对应的角度作为主方向，在把所有的角度旋转到该主方向上后再进行一次 box 的 feature_vector 的求解，得出对应 pix 的梯度直方图。

```
1 def histogram_of_gradients(img, pix):
2     # TODO
3     features=[]
4     img_grad_x = gradient_x(img)
5     img_grad_y = gradient_y(img)
6     img_angle_array = np.arctan2(img_grad_y,img_grad_x)*180/np.pi+180 # 0,360
7     img_grad_array = np.sqrt(img_grad_x**2+img_grad_y**2)
8
9     for pixel_position in pix:
10         pix_outer_angle_box = get_surrounding_box_array(img_angle_array,pixel_position,9)
11
12         pix_divided_angle_boxes = divide_box_array(pix_outer_angle_box,divide_n=2) #divided
13             into n*n
14
15         pix_outer_grad_box = get_surrounding_box_array(img_grad_array,pixel_position,9)
16         pix_divided_grad_boxes = divide_box_array(pix_outer_grad_box,divide_n=2)
17
18         block_features=[]
19         dir_num = 8
20         for cell_index in range(len(pix_divided_angle_boxes)):
21             cell_histogram =
22                 angle_grad_histogram(pix_divided_angle_boxes[cell_index],pix_divided_grad_boxes[cell_index])
23                     # 分成8个direction
24
25             block_features.append(item_ for item_ in cell_histogram)
26
27
principle_agngle_index = block_features.index(max(block_features))
principle_agngle = (principle_agngle_index%dir_num)*(360//dir_num) #找到主方向
```

```

28
29     pix_new_angle_box = np.array((pix_outer_angle_box-principle_agngle+360)%360)
30     pix_new_divided_angle_boxes = divide_box_array(pix_new_angle_box,divide_n=2)
31
32     new_block_features=[]
33     for new_cell_index in range(len(pix_new_divided_angle_boxes)):
34         new_cell_histogram =
35             angle_grad_histogram(pix_new_divided_angle_boxes[new_cell_index],pix_divided_grad_boxes)
36             # 分成8个direction
37         new_block_features.extend(item_ for item_ in new_cell_histogram)
38
39     features.append(new_block_features)
40
41     return np.array(features)

```

用到的函数:

```

1 def get_surrounding_box_array(feature_array,center_position,box_size=9):
2     center_x = center_position[0]
3     center_y = center_position[1]
4
5     array_width = feature_array.shape[1] # 水平方向
6     array_height = feature_array.shape[0] # 垂直方向
7     # box_size 建议为奇数
8     box_expand = box_size//2
9     box_up = max(center_y-box_expand,0)
10    box_down = min(center_y+box_expand,array_height)
11    box_left = max(center_x-box_expand,0)
12    box_right = min(center_x+box_expand,array_width)
13
14    ret_box = feature_array[box_up:box_down,box_left:box_right]
15    return ret_box
16
17 def divide_box_array(box_array,divide_n=2):
18     box_height = box_array.shape[0]
19     box_width = box_array.shape[1]
20     divide_height_step = box_height//divide_n
21     divide_width_step = box_width//divide_n
22     ret_arr =[]
23     for i in range(divide_n):
24         for j in range(divide_n):
25             divide_box_up = i*divide_height_step
26             divide_box_down = (i+1)*divide_height_step #if i!=divide_n else box_height
27             divide_box_left = j*divide_width_step

```

```

28     divide_box_right = (j+1)*divide_width_step #if j !=divide_n else box_width
29
30     divide_box
31         =box_array[divide_box_up:divide_box_down,divide_box_left:divide_box_right].flatten()
32     ret_arr.append(divide_box)
33
34     # print(ret_arr)
35
36 def angle_grad_histogram(angle_array,grad_array,n):
37     # print(angle_array)
38
39     lower_bound = 0
40     upper_bound = 360
41     divide_step = (upper_bound-lower_bound)//n
42
43     histogram_lst = np.zeros(n)
44
45     # print(grad_array)
46     # print(angle_array)
47     for pix_index in range(len(angle_array)):
48         select_index = int(angle_array[pix_index]//divide_step)
49         # print(select_index)
50         down_ = select_index*divide_step
51         l_ = (angle_array[pix_index]-down_)/divide_step
52
53         histogram_lst[select_index]+=grad_array[pix_index]*(1-l_)
54         histogram_lst[(select_index+1)%n]+=grad_array[pix_index]*(l_) #线性加权
55
56     if np.sum(grad_array)==0:
57         return np.zeros(n)
58
59     return histogram_lst/np.linalg.norm(histogram_lst)

```

3 Local feature matching

取 threshold=0.35, 可以得到如下的输出:



图 1: local_feature_mathing_1



图 2: local_feature_mathing_2

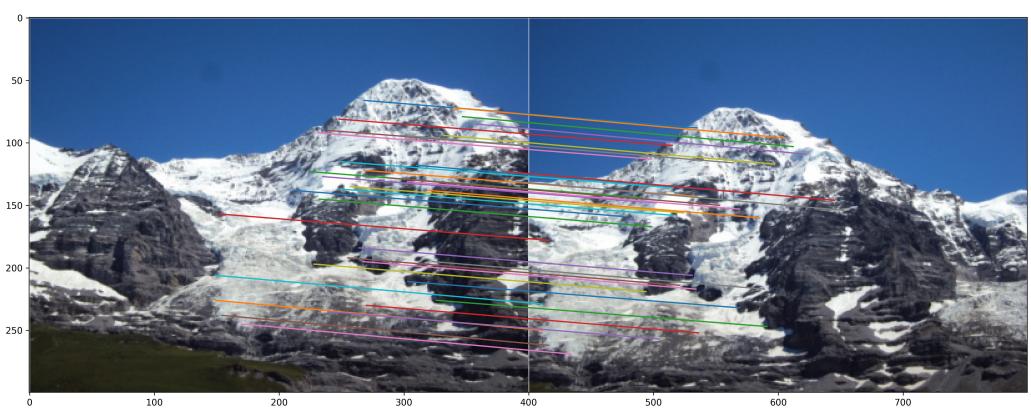


图 3: local_feature_mathing_3

4 Image stitching and Blending

4.1 Compute the alignment of image pairs

思路:为了方便计算单应性矩阵(homography),我们先把传入的点列通过 convert_to_homogeneous_coordinates 函数都转为齐次坐标的形式。

然后我们利用 python 中 array 运算的技巧来对系数矩阵进行构造。

之后利用最小二乘法通过 SVD 分解来进行求解,求解得到 h 的值为 S 最小的奇异值对应 V 中的特征向量,最后把 h 还原成 3*3 的 matrix 形式。

$$\begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix}$$

图 4: homography_compute

```
1 def convert_to_homogeneous_coordinates(pixels):
2     pinxels_array = np.array(pixels)
3     homo_coori = np.ones((np.shape(pinxels_array)[0],3))
4     homo_coori[:,0:2]=pixels
5     return homo_coori
6
7 def compute_homography(pixels_1, pixels_2):
8     # TODO
9
10    pixels_pair_num = len(pixels_1)
11    assert pixels_pair_num>=4, "To compute homography need more pixels-pair\n"
12
13    pixel_matrix1=convert_to_homogeneous_coordinates(pixels_1)
14    pixel_matrix2=convert_to_homogeneous_coordinates(pixels_2)
15    pixel_matrix_ = np.zeros((pixels_pair_num*2,9))
16    pixel_matrix_[:,0:3]=-pixel_matrix1
17    pixel_matrix_[1::2,3:6]=-pixel_matrix1
18    pixel_matrix_[:,6:9]=pixel_matrix1*pixel_matrix2[:,0:1]
19    pixel_matrix_[1::2,6:9]=pixel_matrix1*pixel_matrix2[:,1:2]
20
21    U_,S_,V_ = np.linalg.svd(np.transpose(pixel_matrix_).dot(pixel_matrix_))
22    smallest_index = np.argmin(S_)
23    get_col = V_[smallest_index] # find smallest eigenvalue
24    ret_matrix = np.reshape(get_col, (3, 3))
25
26    return ret_matrix
```

4.2 Align the image with RANSAC

思路: 从传入的点列中随机抽出四对点, 根据这四对点计算 homography_matrix, 再根据得出的 homography_matrix 来对 pixels_1 点列作变换, 将变换后得到的点列与 pixels_2 作差求出 distance_matrix, 每一列代表对应点在 x 方向上和 y 方向上的误差, 即“误差向量”, 求出这个误差向量的模长, 根据 threshold 判断得到的是“inlier”or“outlier”, 根据迭代次数选出得到 inlier 个数最多的那个 homo_matrix 作为最终的变换矩阵。

```
1 def align_pair(pixels_1, pixels_2):
2     pixels_pair_num = len(pixels_1)
3     assert pixels_pair_num>=4, "To compute need more pixels-pair\n"
4
5     final_homo_matrix=np.zeros((3,3)) # initial matrix
6     final_inliers_num = -1
7     pixels_num =len(pixels_1)
8     itration_times = 800
9     threshold_ = 3
10    homo_coor_pixels_1=convert_to_homogeneous_coordinates(pixels_1)
11    homo_coor_pixels_2=convert_to_homogeneous_coordinates(pixels_2)
12
13    range_num = [i for i in range(pixels_num)]
14    for it_ in range(itration_times):
15        random_samples_indices = np.random.choice(range_num,4,replace=False)
16        sample_pixels_1=[]
17        sample_pixels_2=[]
18        for index_ in random_samples_indices:
19            sample_pixels_1.append(pixels_1[index_])
20            sample_pixels_2.append(pixels_2[index_])
21
22        sample_homo_matrix = compute_homography(sample_pixels_1,sample_pixels_2)
23        sample_matrix1_dot_homo = sample_homo_matrix.dot(homo_coor_pixels_1.transpose())
24        sample_distance_matrix_norm_transpose =
25            (sample_matrix1_dot_homo/sample_matrix1_dot_homo[2]).transpose()
26
27        sample_distance_matrix =
28            (sample_distance_matrix_norm_transpose-homo_coor_pixels_2)[:,0:2]
29        sample_distance_norm = np.array([np.linalg.norm(tmp_) for tmp_ in
30            sample_distance_matrix])
31
32        sample_inlier_cnt = np.sum([sample_distance_norm<threshold_])
33
34        if sample_inlier_cnt>final_inliers_num:
35            final_homo_matrix=sample_homo_matrix
36            final_inliers_num=sample_inlier_cnt
```

```
34  
35     return final_homo_matrix
```

4.3 Stitch and blend the image

尝试但最终失败，只是简化了一些计算:(

```
1  def my_stitch_blend(img_1,img_2,est_homo):  
2      img_1 = img_1.astype(np.float32)  
3      img_2 = img_2.astype(np.float32)  
4  
5      h1, w1 = img_1.shape[:2]  
6      h2, w2 = img_2.shape[:2]  
7  
8      corners_1 = np.array([[0, 0,1], [0, h1,1], [w1, 0,1], [w1, h1,1]], dtype=np.float32)  
9      corners_1_homogeneous = est_homo.dot(corners_1.transpose())  
10  
11     corners_1_change = []  
12     for i in range(2):  
13         tmp_ = corners_1_homogeneous[:,i]  
14         corners_1_change.append(tmp_[0:2]/tmp_[2])  
15  
16     corners_1_change=np.array(corners_1_change)  
17  
18     corners_2 = np.array([[0, 0], [0, h2], [w2, 0], [w2, h2]], dtype=np.float32)  
19     all_corners = np.concatenate((corners_1_change,corners_2),axis=0)  
20     x_min, y_min = np.int32(all_corners.min(axis=0))  
21     x_max, y_max = np.int32(all_corners.max(axis=0))  
22  
23     x_range = np.arange(x_min, x_max+1, 1)  
24     y_range = np.arange(y_min, y_max+1, 1)  
25     x, y = np.meshgrid(x_range, y_range)  
26     x=x.astype(np.float32)  
27     y=y.astype(np.float32)  
28  
29  
30     homo_inv = np.linalg.pinv(est_homo).astype(np.float32)  
31     trans_z = homo_inv[2, 0]*x+homo_inv[2, 1]*y+homo_inv[2, 2]  
32     trans_x = (homo_inv[0, 0]*x+homo_inv[0, 1]*y+homo_inv[0, 2])/trans_z  
33     trans_y = (homo_inv[1, 0]*x+homo_inv[1, 1]*y+homo_inv[1, 2])/trans_z  
34  
35     est_img_1 = cv2.remap(img_1, trans_x, trans_y, cv2.INTER_LINEAR,  
                           borderMode=cv2.BORDER_CONSTANT)
```

```

36 est_img_2 = cv2.remap(img_2, x,y, cv2.INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT)
37
38 alpha1 = cv2.remap(np.ones(np.shape(img_1)), trans_x,
39                     trans_y, cv2.INTER_LINEAR)
40 alpha2 = cv2.remap(np.ones(np.shape(img_2)), x, y, cv2.INTER_LINEAR)
41
42 alpha = alpha1+alpha2
43 alpha[alpha == 0] = 2
44 alpha1 = alpha1/alpha
45 alpha2 = alpha2/alpha
46 est_img = est_img_1*alpha1 + est_img_2*alpha2
47
48 est_img = np.clip(est_img, 0, 255).astype(np.uint8)
49
50 return est_img

```

4.4 Generate a panorama

取 threshold=0.5，对提供的图片进行拼接，基本上能够完成三四张图片的拼接，有一定概率能完成五六张图片的拼接。

由于镜头转动过程中导致的图片的形变，图片的梯度信息受到“干扰”，随着照片个数增加至至五六张时，生成出成功缝合图片的概率较小，甚至会出现匹配点个数不足导致无法计算 est_homo 的情况，但偶尔能够生成质量较好的图片，说明算法本身逻辑没有问题，但是匹配精度和匹配策略仍有较大优化空间。

效果图如下：基本上都是对五张图片进行拼接产生的质量较高的图片。

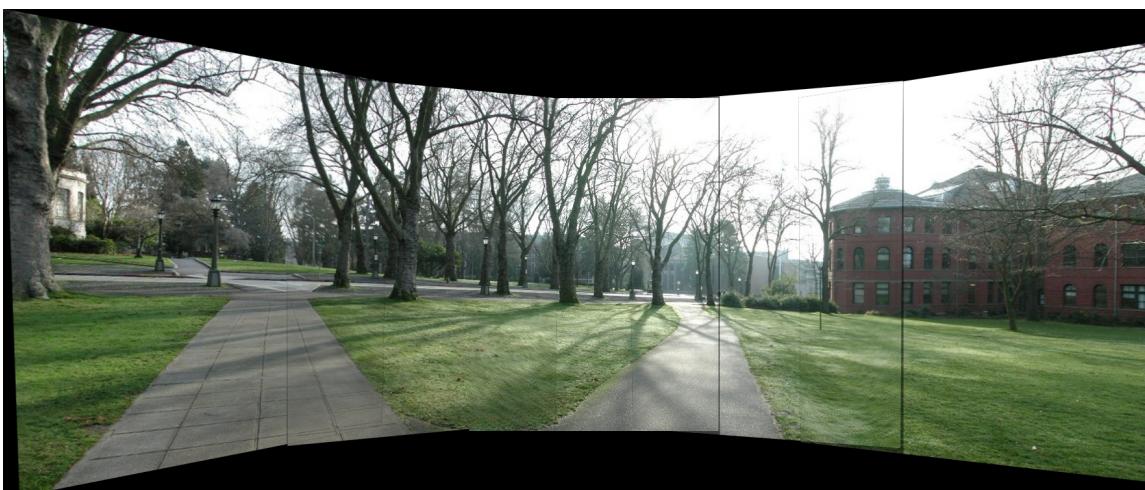


图 5: panorama-parrington

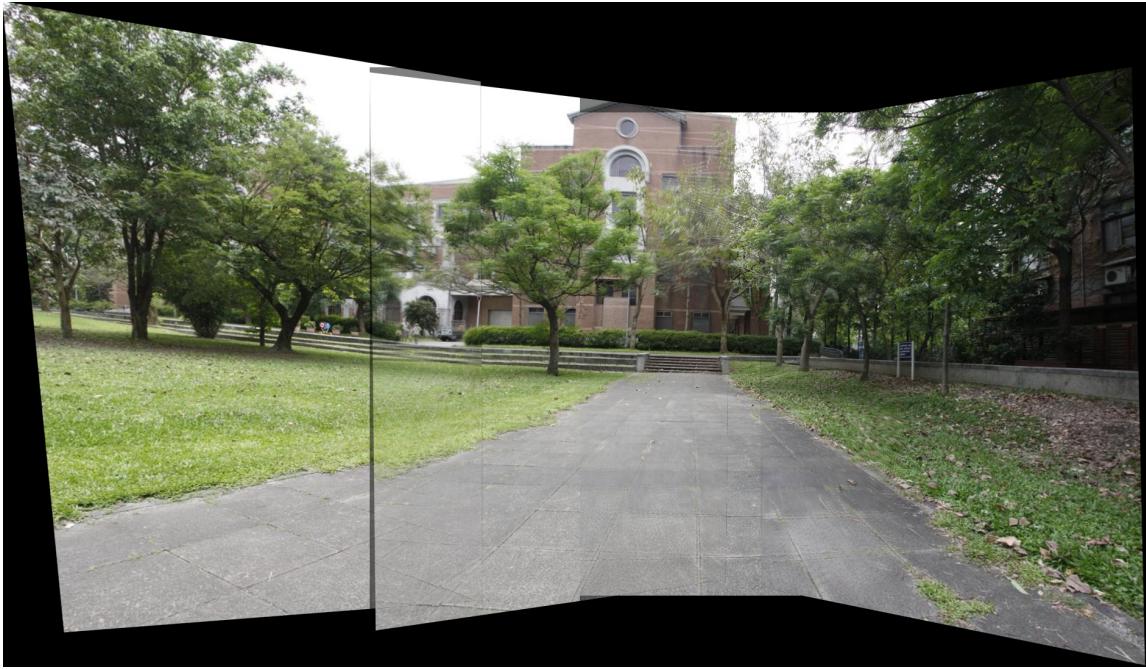


图 6: panorama-library



图 7: panorama-grail



图 8: panorama-Xue_Mount

5 Analyze

5.1 hyperparameters

5.1.1 size of block in 'histogram_of_gradients'

考虑提取 HOG 算子时 block 的大小，发现调大 block 会使得匹配的 align pairs 的数目上升，原因是调大了 block 增加了信息捕获的范围，提高了“匹配”的概率，但与此同时也削减了匹配的精准度，不够精细，容易出现错误的匹配点对。

当我们把 box 大小取成 12×12 与 9×9 , block 为四分之一 box, 得到的每次匹配的点的个数如下：

```
final_choose_num: 42  
final_choose_num: 27
```

图 9: when box size= 12×12

```
final_choose_num: 26  
final_choose_num: 12
```

图 10: when box size= 9×9

5.1.2 threshold in 'feature_matching'

考虑 feature matching 中的 threshold, 发现调大 threshold 会使得匹配的 align pairs 数目上升, 原因是提高 threshold 会使得一些一定程度上相似的点对被考虑, 一定程度上缓解了有时会出现的点对数目不足导致 est_homography 无法计算的问题, 但 threshold 的提升可能会导致坏点的匹配, 使得最后拼接出的图像产生剧烈形变, 这也导致了后续图片无法正常拼接。

当我们把 threshold 大小取为 0.5 和 0.45, 得到的每次匹配的点个数如下:

```
final_choose_num: 30  
final_choose_num: 11
```

图 11: when threshold=0.5

```
final_choose_num: 26  
final_choose_num: 6
```

图 12: when threshold=0.45

5.2 Camera Moving

对图像进行缝合本质上是把一张张不同视角照片铺列在一个平面上, 找到一个合适的变换方式使得照片的某些关键位置重合。

而我们现在做的实际上是一个线性的映射, 因此在摄像机在一个方向上平移或保持摄像机朝向不变, 垂直朝向进行旋转时时会取得比较好的效果, 因为这些 camera moving 都是局限在一个平面上的, 线性映射能起到较好的变换效果。

但是如果摄像头的移动不局限在一个平面上, 如果拍摄者以自身为轴转动摄像机, 改变摄像机的朝向, 那么这时我们需要做的相当于是一个化曲面为平面的过程, 这样的变换线性映射的效果可能就要大打折扣, 导致图片产生较大的形变, 进而导致拼接上的图的梯度受到较大的“破坏”, 这也就是为什么在拼接图片的张数增多时, 拼接的效果会陡然变差。

下图是我仅以镜头的平移拍摄的鼠标垫照片, 可以发现缝合的效果非常好, 即使这张图片上有许多重复的细微元素(如下方的小字母‘E’):

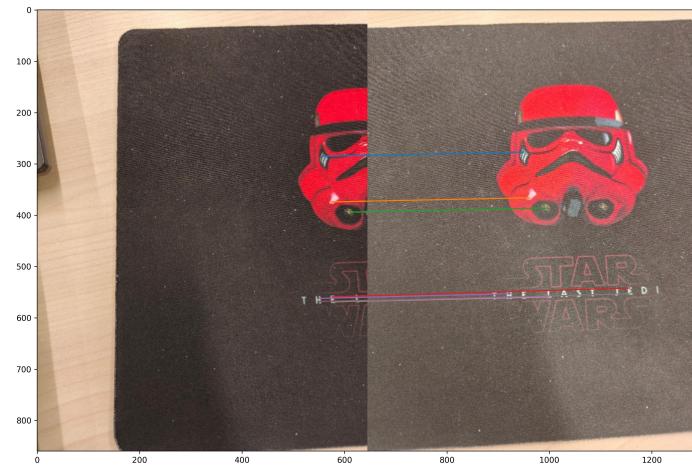


图 13: feature match, 可以发现下方的白色小字母存在错误匹配 (红线)

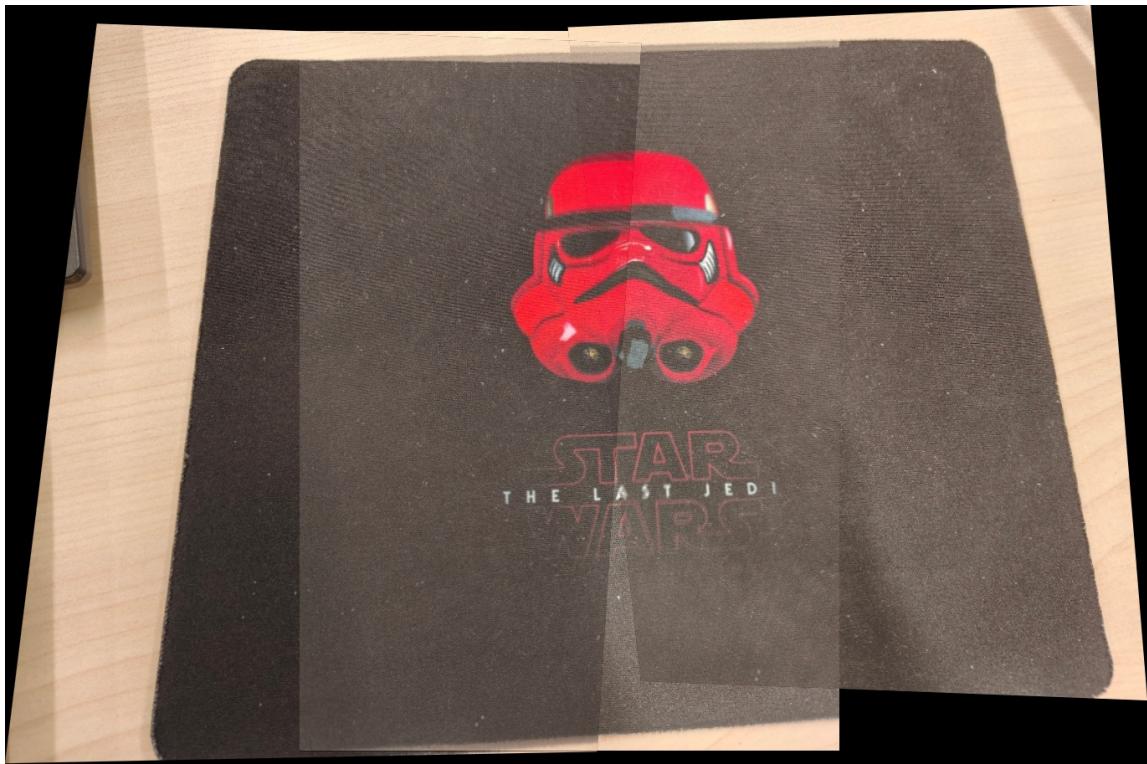


图 14: pano-pad

5.3 How to blend

我尝试过改变 blend 图片的顺序, 框架中给出的方法是从图片序列 mid 开始, 先往右匹配, 然后往左匹配。

我试着改成从第零张图开始, 直接一次性往右匹配, 发现这样的效果比较差, 我分析了一下原因, 这样的拼接逻辑会导致随着拼接图数的增加, 原图的形变程度会比从中间开始进行拼接造成的形变更大, 且这样的拼接方式只要一张图拼错就会导致后续的图完全没法拼接上, 而从中间开始至少保证了左右两张图的拼接。

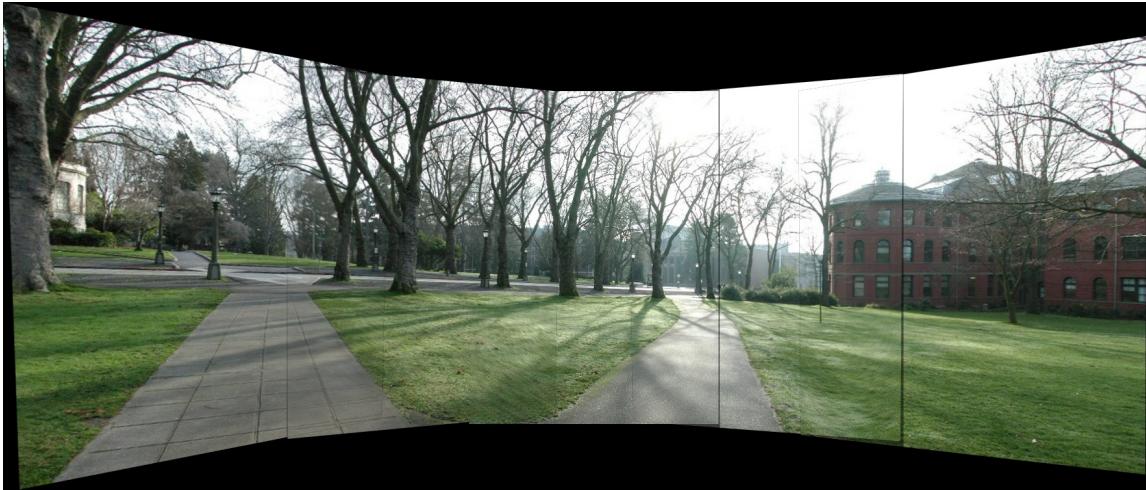


图 15: blend from mid - 5pictures

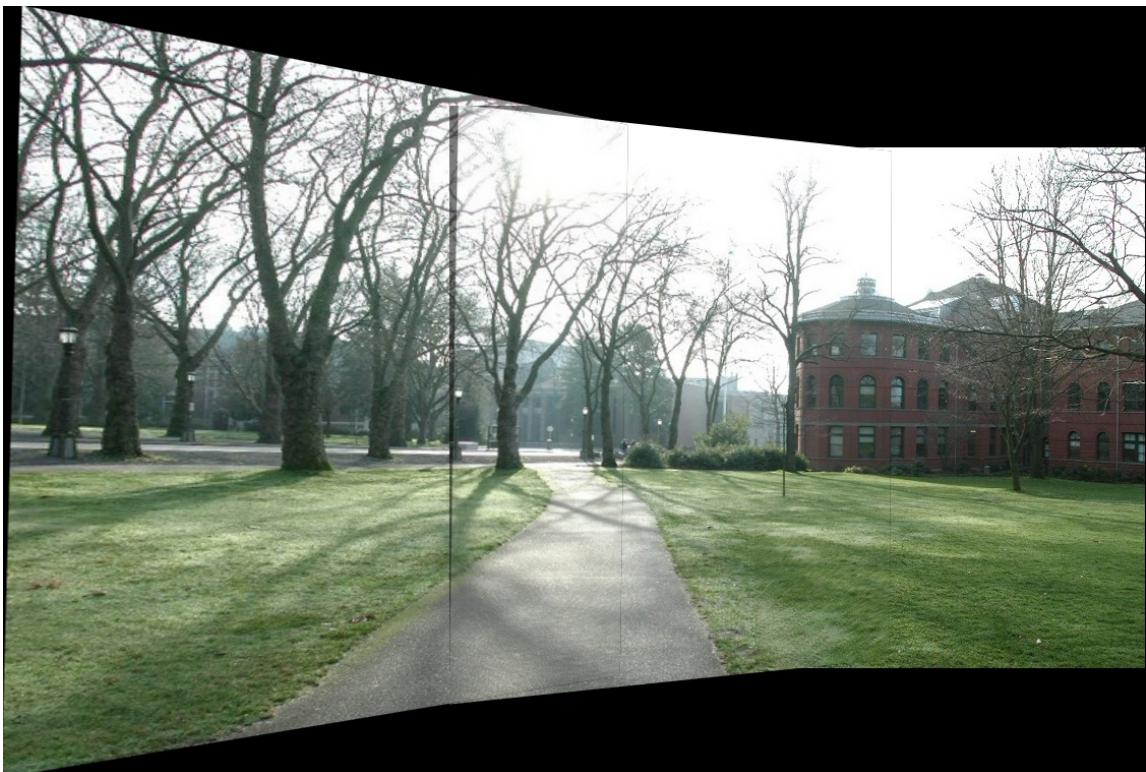


图 16: blend from left - 3pictures

我想到的可以改进的拼接方法有以下两个：

- 每拼接上一张图片 P_1 , 记录下这张图片的变换矩阵 M_1 , 然后在拼下一张图时计算逆矩阵还原出原先的这张图片 P_1 , 将还原出的图片 P_1 与下张图 P_2 进行配对, 得出下张图的变换矩阵 M_2 , 最终得到 P_1 变换矩阵仍为 M_1 , P_2 变换矩阵为 $M_1 \cdot M_2$, 这样就一定程度上削弱了由于 blend 造成的梯度形变对点对匹配造成的影响
- 每往 principle 图上拼一张图片 P_1 , 记录下 P_1 的位置, 在进行下一张图片匹配时, 将匹配的点可能出现的位置限制在 P_1 的这个位置范围内, 这样就降低了坏点匹配的概率。