**一．实验内容：**

利用sift算法，实现全景拼接算法，将给定的两幅图片拼接为一幅.

**二．实验环境：**

主机配置: CPU :intel core i5-7300 2.50GHZ RAM :8.0GB

运行环境：win10 64位操作系统

开发环境：python3.7

**三．核心算法原理：**

**1.SIFT算法**

SIFT，即尺度不变特征变换（Scale-invariant feature transform，SIFT），是用于[图像处理](https://baike.baidu.com/item/%E5%9B%BE%E5%83%8F%E5%A4%84%E7%90%86/294902" \t "https://baike.baidu.com/item/SIFT/_blank)领域的一种描述。这种描述具有尺度不变性，可在图像中检测出关键点，是一种局部特征描述子。

**特点:**

1.SIFT特征是图像的局部特征，其对旋转、尺度缩放、亮度变化保持不变性，对视角变化、仿射变换、噪声也保持一定程度的稳定性；

2. 区分性（Distinctiveness）好，信息量丰富，适用于在海量特征数据库中进行快速、准确的匹配；

3. 多量性，即使少数的几个物体也可以产生大量的SIFT特征向量；

4.高速性，经优化的SIFT匹配算法甚至可以达到实时的要求；

5.可扩展性，可以很方便的与其他形式的特征向量进行联合。

**特征检测基本步骤：**

1.尺度空间极值检测：

搜索所有尺度上的图像位置。通过高斯微分函数来识别潜在的对于尺度和旋转不变的兴趣点。

2. 关键点定位

在每个候选的位置上，通过一个拟合精细的模型来确定位置和尺度。关键点的选择依据于它们的稳定程度。

3. 方向确定

基于图像局部的梯度方向，分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向、尺度和位置进行变换，从而提供对于这些变换的不变性。

4. 关键点描述

在每个关键点周围的邻域内，在选定的尺度上测量图像局部的梯度。这些梯度被变换成一种表示，这种表示允许比较大的局部形状的变形和光照变化。

**SIFT特征匹配阶段：**

第一阶段：SIFT特征的生成，即从多幅图像中提取对尺度缩放、旋转、亮度变化无关的特征向量。

第二阶段：SIFT特征向量的匹配。

SIFT特征的生成一般包括以下几个步骤：

1. 构建[尺度空间](https://baike.baidu.com/item/%E5%B0%BA%E5%BA%A6%E7%A9%BA%E9%97%B4" \t "https://baike.baidu.com/item/SIFT/_blank)，检测极值点，获得尺度不变性。

2. 特征点过滤并进行精确定位。

3. 为特征点分配方向值。

4. 生成特征描述子。

当两幅图像的SIFT特征向量生成以后，下一步就可以采用关键点特征向量的[欧式距离](https://baike.baidu.com/item/%E6%AC%A7%E5%BC%8F%E8%B7%9D%E7%A6%BB" \t "https://baike.baidu.com/item/SIFT/_blank)来作为两幅图像中关键点的相似性判定度量。取图1的某个关键点，通过遍历找到图像2中的距离最近的两个关键点。在这两个关键点中，如果最近距离除以次近距离小于某个阈值，则判定为一对匹配点。

1. **Lowe’s算法：**

为了进一步筛选匹配点，来获取优秀的匹配点，这就是所谓的“去粗取精”。一般会采用Lowe’s算法来进一步获取优秀匹配点。

   为了排除因为图像遮挡和背景混乱而产生的无匹配关系的关键点，SIFT的作者Lowe提出了比较最近邻距离与次近邻距离的SIFT匹配方式：取一幅图像中的一个SIFT关键点，并找出其与另一幅图像中欧式距离最近的前两个关键点，在这两个关键点中，如果最近的距离除以次近的距离得到的比率ratio少于某个阈值T，则接受这一对匹配点。因为对于错误匹配，由于特征空间的高维性，相似的距离可能有大量其他的错误匹配，从而它的ratio值比较高。显然降低这个比例阈值T，SIFT匹配点数目会减少，但更加稳定，反之亦然。

   Lowe推荐ratio的阈值为0.8，但作者对大量任意存在尺度、旋转和亮度变化的两幅图片进行匹配，结果表明ratio取值在0. 4~0. 6 之间最佳，小于0. 4的很少有匹配点，大于0. 6的则存在大量错误匹配点，所以建议ratio的取值原则如下:

ratio=0. 4：对于准确度要求高的匹配；

ratio=0. 6：对于匹配点数目要求比较多的匹配；

ratio=0. 5：一般情况下。

**3.RANSAC算法**

   随机抽样一致算法（RANdom SAmple Consensus,RANSAC）,采用迭代的方式从一组包含离群的被观测数据中估算出数学模型的参数。RANSAC算法假设数据中包含正确数据和异常数据（或称为噪声）。正确数据记为内点（inliers），异常数据记为外点（outliers）。同时RANSAC也假设，给定一组正确的数据，存在可以计算出符合这些数据的模型参数的方法。该算法核心思想就是随机性和假设性，随机性是根据正确数据出现概率去随机选取抽样数据，根据大数定律，随机性模拟可以近似得到正确结果。假设性是假设选取出的抽样数据都是正确数据，然后用这些正确数据通过问题满足的模型，去计算其他点，然后对这次结果进行一个评分。

**算法基本思想**

（1）要得到一个直线模型，需要两个点唯一确定一个直线方程。所以第一步随机选择两个点。

（2）通过这两个点，可以计算出这两个点所表示的模型方程y=ax+b。

（3）将所有的数据点套到这个模型中计算误差。

（4）找到所有满足误差阈值的点。

（5）然后我们再重复（1）~（4）这个过程，直到达到一定迭代次数后，选出那个被支持的最多的模型，作为问题的解。

1. **处理步骤：**

Step1:从输入的两张图片里检测关键点、提取局部不变特征。(sift)

Step2:匹配的两幅图像之间的特征(Lowe’s算法)

Step3:使用RANSAC算法利用匹配特征向量估计单映矩阵（homography matrix）。

Step4:利用Step3得到的单映矩阵应用扭曲变换。

**具体步骤分析见 六. 步骤分析**

**五．核心代码：**

import numpy as np

import imutils

import cv2

class Stitcher:

def \_\_init\_\_(self):

# determine if we are using OpenCV v3.X

self.isv3 = imutils.is\_cv3()

def stitch(self, images, ratio=0.75, reprojThresh=4.0,

showMatches=False):

# unpack the images, then detect keypoints and extract

# local invariant descriptors from them

(imageB, imageA) = images

(kpsA, featuresA) = self.detectAndDescribe(imageA)

(kpsB, featuresB) = self.detectAndDescribe(imageB)

# match features between the two images

M = self.matchKeypoints(kpsA, kpsB,

featuresA, featuresB, ratio, reprojThresh)

# if the match is None, then there aren't enough matched

# keypoints to create a panorama

if M is None:

return None

# otherwise, apply a perspective warp to stitch the images

# together

(matches, H, status) = M

result = cv2.warpPerspective(imageA, H,

(imageA.shape[1] + imageB.shape[1], imageA.shape[0]))

result[0:imageB.shape[0], 0:imageB.shape[1]] = imageB

# check to see if the keypoint matches should be visualized

if showMatches:

vis = self.drawMatches(imageA, imageB, kpsA, kpsB, matches,

status)

# return a tuple of the stitched image and the

# visualization

return (result, vis)

# return the stitched image

return result

#接收照片，检测关键点和提取局部不变特征

#用到了高斯差分（Difference of Gaussian (DoG)）关键点检测，和SIFT特征提取

#detectAndCompute方法用来处理提取关键点和特征

#返回一系列的关键点

def detectAndDescribe(self, image):

# convert the image to grayscale

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# check to see if we are using OpenCV 3.X

if self.isv3:

# detect and extract features from the image

descriptor = cv2.xfeatures2d.SIFT\_create()

(kps, features) = descriptor.detectAndCompute(image, None)

# otherwise, we are using OpenCV 2.4.X

else:

# detect keypoints in the image

detector = cv2.FeatureDetector\_create("SIFT")

kps = detector.detect(gray)

# extract features from the image

extractor = cv2.DescriptorExtractor\_create("SIFT")

(kps, features) = extractor.compute(gray, kps)

# convert the keypoints from KeyPoint objects to NumPy

# arrays

kps = np.float32([kp.pt for kp in kps])

# return a tuple of keypoints and features

return (kps, features)

#matchKeypoints方法需要四个参数，第一张图片的关键点和特征向量，第二张图片的关键点特征向量。

#David Lowe’s ratio测试变量和RANSAC重投影门限也应该被提供。

def matchKeypoints(self, kpsA, kpsB, featuresA, featuresB,

ratio, reprojThresh):

# compute the raw matches and initialize the list of actual

# matches

matcher = cv2.DescriptorMatcher\_create("BruteForce")

rawMatches = matcher.knnMatch(featuresA, featuresB, 2)

matches = []

# loop over the raw matches

for m in rawMatches:

# ensure the distance is within a certain ratio of each

# other (i.e. Lowe's ratio test)

if len(m) == 2 and m[0].distance < m[1].distance \* ratio:

matches.append((m[0].trainIdx, m[0].queryIdx))

# computing a homography requires at least 4 matches

if len(matches) > 4:

# construct the two sets of points

ptsA = np.float32([kpsA[i] for (\_, i) in matches])

ptsB = np.float32([kpsB[i] for (i, \_) in matches])

# compute the homography between the two sets of points

(H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC,

reprojThresh)

# return the matches along with the homograpy matrix

# and status of each matched point

return (matches, H, status)

# otherwise, no homograpy could be computed

return None

#连线画出两幅图的匹配

def drawMatches(self, imageA, imageB, kpsA, kpsB, matches, status):

# initialize the output visualization image

(hA, wA) = imageA.shape[:2]

(hB, wB) = imageB.shape[:2]

vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")

vis[0:hA, 0:wA] = imageA

vis[0:hB, wA:] = imageB

# loop over the matches

for ((trainIdx, queryIdx), s) in zip(matches, status):

# only process the match if the keypoint was successfully

# matched

if s == 1:

# draw the match

ptA = (int(kpsA[queryIdx][0]), int(kpsA[queryIdx][1]))

ptB = (int(kpsB[trainIdx][0]) + wA, int(kpsB[trainIdx][1]))

cv2.line(vis, ptA, ptB, (0, 255, 0), 1)

# return the visualization

return vis

if \_\_name\_\_ == '\_\_main\_\_':

# load the two images and resize them to have a width of 400 pixels

# (for faster processing)

imageA = cv2.imread('D:/test1.jpg')

imageB = cv2.imread('D:/test2.jpg')

#imageA = imutils.resize(imageA, width=400)

#imageB = imutils.resize(imageB, width=400)

# stitch the images together to create a panorama

# showMatches=True 展示两幅图像特征的匹配,返回vis

stitcher = Stitcher()

(result, vis) = stitcher.stitch([imageA, imageB], showMatches=True)

# show the images

cv2.imwrite('D:/vis1.jpg', vis)

cv2.imwrite('D:/result.jpg', result)

**六．步骤分析：**

**概述：**

Sitich调用detectAndDescribe，检测两张图片里的关键点、提取局部不变特征。

有了关键点和特征，sitich调用matchKeypoints方法来匹配两张图片里的特征。如果返回匹配的M为None，就是因为现有的关键点不足以匹配生成全景图。假设M不返回None，拆包返回元组，包含关键点匹配matches、从RANSAC算法中得到的最优单映射变换矩阵H以及最后的单映计算状态列表status，用来表明那些已经成功匹配的关键点。

有了最优单映射变换矩阵H后，就可将两张图片“缝合起来”。stitch调用cv2.warpPerspective进行缝合。这样，就返回一个拼接的图片。

最后sitich调用drawMatches函数用来将两张图片关键点的匹配可视化。

**函数分析：**

**detectAndDescribe**

detectAndDescribe方法用来接收照片，检测关键点和提取局部不变特征。实现中用到了高斯差分（Difference of Gaussian (DoG)）关键点检测，和SIFT特征提取。

参数为单个的图片。

返回值kps为keypoints,features为局部不变特征。

检测是否用了OpenCV 3.X，如果是，就用cv2.xfeatures2d.SIFT\_create方法来实现DoG关键点检测和SIFT特征提取。detectAndCompute方法用来处理提取关键点和特征。

如果用OpenCV2.4，则cv2.FeatureDetector\_create方法来实现关键点的检测（DoG）。detect方法返回一系列的关键点。用SIFT关键字来初始化cv2.DescriptorExtractor\_create，设置SIFT特征提取。调用extractor的compute方法返回一组关键点周围量化检测的特征向量。

最后，关键点从KeyPoint对象转换为NumPy数组后返回给调用函数。

**matchKeypoints**

用于匹配两张图片的特征。

matchKeypoints方法需要六个参数，第一张图片的关键点和特征向量，第二张图片的关键点特征向量，David Lowe’s ratio测试变量ratio和RANSAC重投影门限reprojThresh。

返回值为matches, H, status。分别为匹配的关键点matches，最优单映射变换矩阵 H（3x3)，单映计算的状态列表status用于表示已经成功匹配的关键点。

匹配特征过程: 循环每张图片的描述子，计算距离，最后找到每对描述子的最小距离。

因为这是计算机视觉中的一个非常普遍的做法，OpenCV已经内置了cv2.DescriptorMatcher\_create方法，用来匹配特征。BruteForce参数表示我们能够更详尽计算两张图片直接的欧式距离，以此来寻找每对描述子的最短距离。

knnMatch方法是K=2的两个特征向量的k-NN匹配（k-nearest neighbors algorithm，K近邻算法），表明每个匹配的前两名作为特征向量返回。之所以我们要的是匹配的前两个而不是只有第一个，是因为我们需要用David Lowe’s ratio来测试假匹配然后做修剪。

之后，用第79行的rawMatches来计算每对描述子，但是这些描述子可能是错误的，也就是这是图片不是真正的匹配。去修剪这些有误的匹配，我们可以运用 Lowe’s ratio测试特别的来循环rawMatches，这是用来确定高质量的特征匹配。正常的Lowe’s ratio 值在[0.7,0.8].

我们用Lowe’s ratio 测试得到matches的值后，我们就可以计算这两串关键点之间的单映性。 计算两串关键点的单映性需要至少四个匹配。为了获得更为可信的单映性，我们至少需要超过四个匹配点。

调用cv2.findHomography计算H和status。

**drawMatches**

drawMatches用来将两张图片关键点的匹配可视化。

参数为原始图片A,B,关键点kpsA，kpsB，匹配的关键点matches以及单映的状态列表status。

返回值为将两张图片中的匹配点用直线连接起来的图片。

运用参数，我们可以通过将两张图片匹配的关键点N和关键点M画直线连接，并返回包含这些直线的图片来实现可视化。

**Stitch**

参数

images。传入图片的列表，缝合在一起形成全景图。注意传入的图像是从左到右的顺序。如果提供的不是这样的顺序，程序仍然可以跑，但是输出全景是不正确的。ratio ，用于特征匹配时David Lowe比率测试，reprojthresh 是RANSAC算法中最大像素“回旋的余地”，最后的showMatches，是一个布尔类型的值，用于表明是否调用drawMatches进行关键点匹配可视化。

返回值，返回拼接好的图片result已经用于可视化匹配关键点的图片vis.

Sitich调用detectAndDescribe，检测两张图片里的关键点、提取局部不变特征。

调用matchKeypoints方法来匹配两张图片里的特征

调用cv2.warpPerspective进行缝合。需要三个参数：想要“缝合”上来的照片（本程序里的右边的图片）；还有3\*3的最优单映射转换矩阵H；最后就是塑造出要输出的照片。我们得到输出图像的宽是两图片之和，高即为第二张图像的高度。

调用drawMatches函数用来将两张图片关键点的匹配可视化。

1. **结果展示：**

给定图片为老师提供的IMG\_201901.jpg和IMG\_201901.jpg，重命名为test1.jpg,test2.jpg。

输出拼接好的图片为rusult.jpg。

展示关键点匹配的图片vis.jpg。

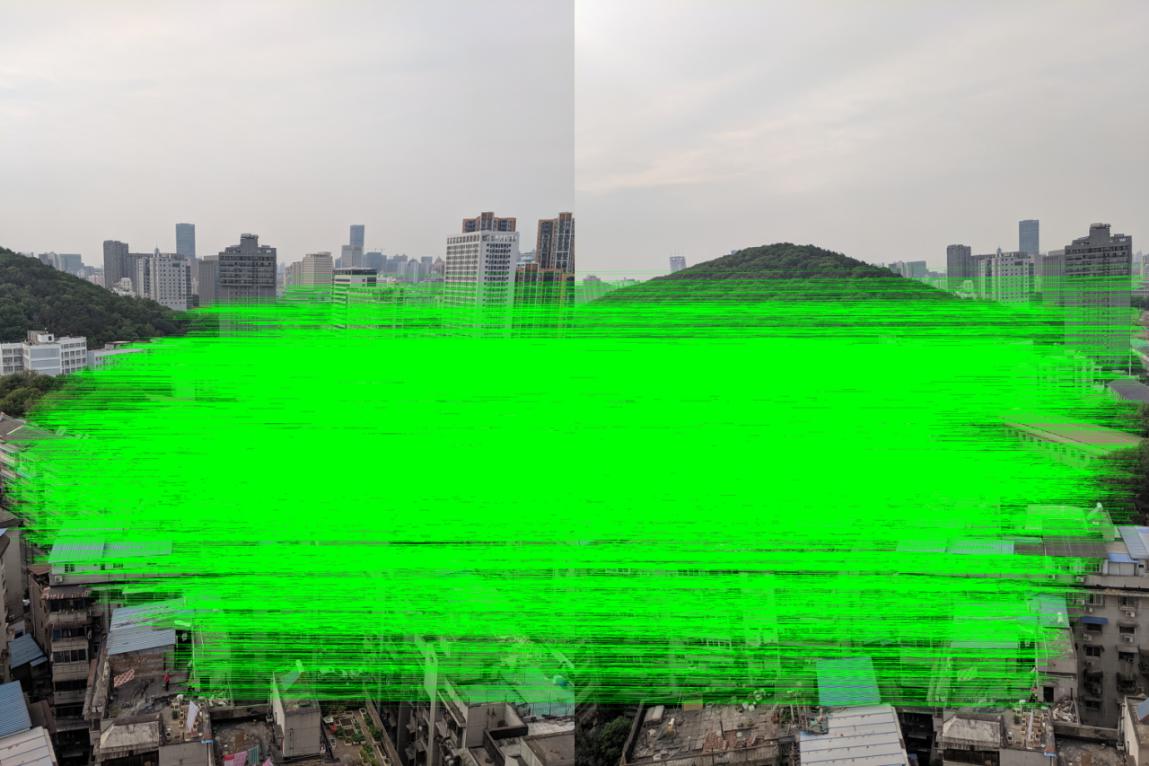
**传入图片**

|  |  |
| --- | --- |
| **test1** | **test2** |
| **test1.jpg** | **test2.jpg** |

**拼接好的图片：result.jpg**

****

**可视化关键点匹配:Vis.jpg**

****

**8.性能分析：**

**时间：**

**项目运行总时间304.79945 s**

单张图片detectAndDescribe检测关键点、提取局部不变特征用时8.31044 s

matchKeypoints匹配两张图片里的特征用时292.67749 s

cv2.warpPerspective缝合图像用时0.27427 s

drawMatches建立直线关键点的匹配可视化用时0.33913 s

可见matchKeypoints计算量最大

Lowe’s ratio循环消耗时间较长，同时cv2.findHomography计算最优单映矩阵H与status占用一定时间。

**时间优化：**

使用比SIFT快的SURF方法，

调节它的参数，减少一些关键点，只获取64维而不是128维的向量等，加快速度。

**图像拼接质量：**

拼接比较流畅，肉眼判断为一张图片。

缺陷在于**看到有一条像折痕一样的线条，这个就是两个图片的拼接线，主要原因是光线的变化。**

****拼接质量优化：****

对于衔接处存在的缝隙问题，有一个解决办法是按一定权重叠加图1和图2的重叠部分，在重叠处图2的比重是1，向着图1的方向，越远离衔接处，图1的权重越来越大，图2的权重越来越低，实现平稳过渡****。****

****九.优化：****

**时间优化：**使用比SIFT快的SURF方法，使用Hessian算法检测关键点。在使用SURF时，还可以调节它的参数，减少一些关键点，只获取64维而不是128维的向量等，加快速度。

**拼接质量优化：**对第一张图和它的重叠区做一些加权处理，重叠部分，离左边图近的，左边图的权重就高一些，离右边近的，右边旋转图的权重就高一些，然后两者相加，实现平滑过渡。

****优化代码：****

**import cv2**

**import numpy as np**

**from matplotlib import pyplot as plt**

**import time**

**MIN = 10**

**starttime=time.time()**

**img1 = cv2.imread('D:/test1.jpg')**

**img2 = cv2.imread('D:/test1.jpg')**

**surf=cv2.xfeatures2d.SURF\_create(10000,nOctaves=4,extended=False,upright=True)**

**kp1,descrip1=surf.detectAndCompute(img1,None)**

**kp2,descrip2=surf.detectAndCompute(img2,None)**

**FLANN\_INDEX\_KDTREE = 0**

**indexParams = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)**

**searchParams = dict(checks=50)**

**flann=cv2.FlannBasedMatcher(indexParams,searchParams)**

**match=flann.knnMatch(descrip1,descrip2,k=2)**

**good=[]**

**for i,(m,n) in enumerate(match):**

**if(m.distance<0.75\*n.distance):**

**good.append(m)**

**if len(good)>MIN:**

**src\_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)**

**ano\_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)**

**M,mask=cv2.findHomography(src\_pts,ano\_pts,cv2.RANSAC,5.0)**

**warpImg = cv2.warpPerspective(img2, np.linalg.inv(M), (img1.shape[1]+img2.shape[1], img2.shape[0]))**

**direct=warpImg.copy()**

**direct[0:img1.shape[0], 0:img1.shape[1]] =img1**

**rows,cols=img1.shape[:2]**

**for col in range(0,cols):**

**if img1[:, col].any() and warpImg[:, col].any():#开始重叠的最左端**

**left = col**

**break**

**for col in range(cols-1, 0, -1):**

**if img1[:, col].any() and warpImg[:, col].any():#重叠的最右一列**

**right = col**

**break**

**res = np.zeros([rows, cols, 3], np.uint8)**

**for row in range(0, rows):**

**for col in range(0, cols):**

**if not img1[row, col].any():#如果没有原图，用旋转的填充**

**res[row, col] = warpImg[row, col]**

**elif not warpImg[row, col].any():**

**res[row, col] = img1[row, col]**

**else:**

**srcImgLen = float(abs(col - left))**

**testImgLen = float(abs(col - right))**

**alpha = srcImgLen / (srcImgLen + testImgLen)**

**res[row, col] = np.clip(img1[row, col] \* (1-alpha) + warpImg[row, col] \* alpha, 0, 255)**

**warpImg[0:img1.shape[0], 0:img1.shape[1]]=res**

**final=time.time()**

**img4=cv2.cvtColor(warpImg,cv2.COLOR\_BGR2RGB)**

**plt.imshow(img4,),plt.show()**

**print("\ntotal cost %f"%(final-starttime))**

**cv2.imwrite("D:/better.png",warpImg)**

**else:**

**print("not enough matches!")**

****优化效果：****

****用时****

****116.264745s****

****平滑的拼接：****

****Better.png****

********

****可见采用优化的算法降低了运行时间，并实现了平滑的拼接****