# 这里的代码需要使用opencv python 4.5.1，她的api和3.4.1.15是不一样的

实际项目中一般都直接使用封装好的sift算法。以前为了用sift，都是用的旧版本：opencv-contib-python=3.4.2.17，现在sift专利过期了，新版的opencv直接可以使用sift算法，opencv-python==4.5.1版本测试可以使用。

sift算法理论部分参考前面文章：[**sift算法理解**](https://www.cnblogs.com/silence-cho/p/15143439.html)

关于sift，opencv中主要有这个几个函数:

**1.1 sift特征点检测**

**cv2.SIFT\_create() ，注意：opencv python 3.4.1.15是：cv2.xfeatures2d\_SIFT.create()**

创建sift对象，官方文档：<https://docs.opencv.org/4.5.3/d7/d60/classcv_1_1SIFT.html>

sift = cv2.SIFT\_create(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)

参数：

nfeatures: 需要保留的特征点的个数，特征按分数排序（分数取决于局部对比度）

nOctaveLayers：每一组高斯差分金字塔的层数，sift论文中用的3。高斯金字塔的组数通过图片分辨率计算得到

contrastThreshold： 对比度阈值，用于过滤低对比度区域中的特征点。阈值越大，检测器产生的特征越少。 (sift论文用的0.03，nOctaveLayers若为3， 设置参数为0.09，实际值为：contrastThreshold/nOctaveLayers)

edgeThreshold：用于过滤掉类似图片边界处特征的阈值(边缘效应产生的特征)，注意其含义与contrastThreshold不同，即edgeThreshold越大，检测器产生的特征越多(过滤掉的特征越少)；sift论文中用的10；

sigma：第一组高斯金字塔高斯核的sigma值，sift论文中用的1.6。 （图片较模糊，或者光线较暗的图片，降低这个参数）

descriptorType：特征描述符的数据类型，支持CV\_32F和CV\_8U

返回值：sift对象(cv2.Feature2D对象)

**cv2.Feature2D.detect()**

检测特征关键点，官方文档：<https://docs.opencv.org/4.5.3/d0/d13/classcv_1_1Feature2D.html#a8be0d1c20b08eb867184b8d74c15a677>

keypoints = cv2.Feature2D.detect(image, mask)

参数：

image：需要检测关键点的图片

mask：掩膜，为0的区域表示不需要检测关键点，大于0的区域检测

返回值：

keypoints：检测到的关键点

**cv2.Feature2D.compute()**

生成特征关键点的描述符，官方文档：<https://docs.opencv.org/4.5.3/d0/d13/classcv_1_1Feature2D.html#a8be0d1c20b08eb867184b8d74c15a677>

keypoints, descriptors = cv.Feature2D.compute(image, keypoints)

参数：

image：需要生成描述子的图片

keypoints: 需要生成描述子的关键点

返回值：

keypoints：关键点（原始关键点中，不能生成描述子的关键点会被移除；）

descriptors：关键点对应的描述子

**cv2.Feature2D.detectAndCompute()**

检测关键点，并生成描述符，是上面detect()和compute()的综合

keypoints, descriptors = cv.Feature2D.detectAndCompute(image, mask)

**cv2.drawKetpoints()**

绘制检测到的关键点，官方文档：<https://docs.opencv.org/4.5.3/d4/d5d/group__features2d__draw.html#ga5d2bafe8c1c45289bc3403a40fb88920>

outImage = cv2.drawKeypoints(image, keypoints, outImage, color, flags)

参数：

image：检测关键点的原始图像

keypoints：检测到的关键点

outImage：绘制关键点后的图像，其内容取决于falgs的设置

color：绘制关键点采用的颜色

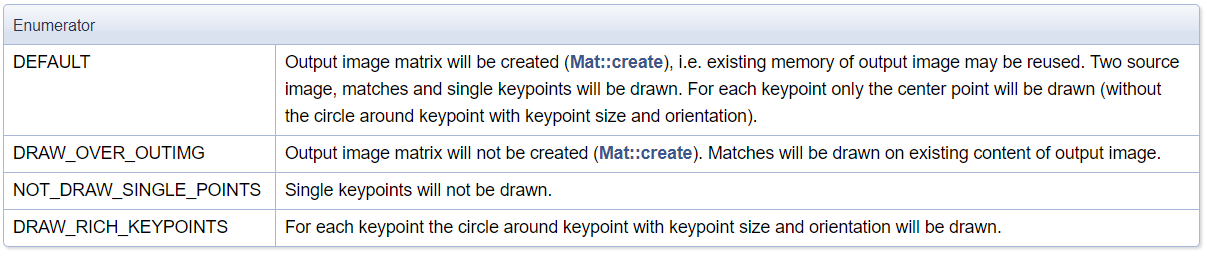
flags：

cv2.DRAW\_MATCHES\_FLAGS\_DEFAULT:默认值，匹配了的关键点和单独的关键点都会被绘制

cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS: 绘制关键点，且每个关键点都绘制圆圈和方向

cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_OVER\_OUTIMG：

cv2.DRAW\_MATCHES\_FLAGS\_NOT\_DRAW\_SINGLE\_POINTS：只绘制匹配的关键点，单独的关键点不绘制

flags的原始含义如下：  


**keypoint**

<https://docs.opencv.org/4.5.3/d2/d29/classcv_1_1KeyPoint.html#aea339bc868102430087b659cd0709c11>

上述检测到的keypoint，在opencv中是一个类对象，其具有如下几个属性：

angle: 特征点的方向，值在0-360

class\_id: 用于聚类id,没有进行聚类时为-1

octave: 特征点所在的高斯金差分字塔组

pt: 特征点坐标

response: 特征点响应强度，代表了该点时特征点的程度（特征点分数排序时，会根据特征点强度）

size:特征点领域直径

**descriptor**

检测点对应的descriptor，是一个128维的向量。

**sift简单使用**

opencv中sift特征点检测和绘制，使用代码和结果如下：(opencv版本：opencv-python==4.5.1.48)

import cv2

img = cv2.imread(r"./lenna.png")

img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

sift = cv2.SIFT\_create(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)

keypoints, descriptors = sift.detectAndCompute(img\_gray, None)

for keypoint,descriptor in zip(keypoints, descriptors):

print("keypoint:", keypoint.angle, keypoint.class\_id, keypoint.octave, keypoint.pt, keypoint.response, keypoint.size)

print("descriptor: ", descriptor.shape)

img = cv2.drawKeypoints(image=img\_gray, outImage=img, keypoints=keypoints,

flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS,

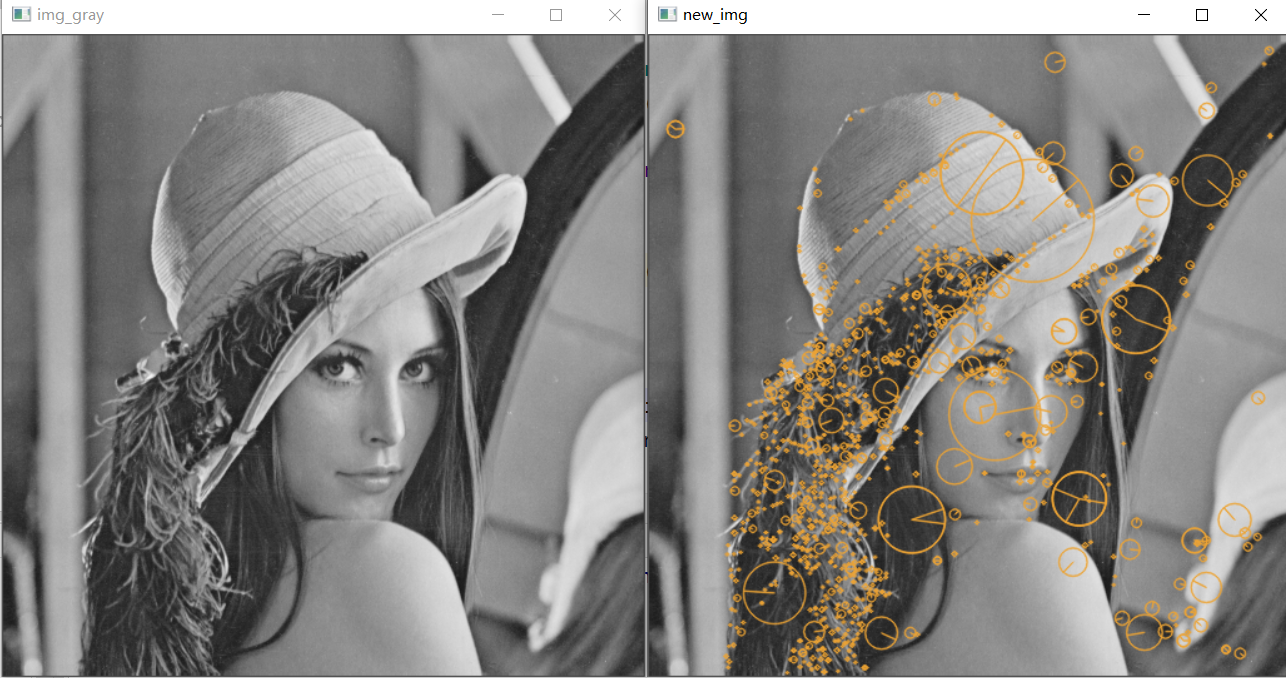
color=(51, 163, 236))

cv2.imshow("img\_gray", img\_gray)

cv2.imshow("new\_img", img)

cv2.waitKey(0)

cv2.destroyAllWindows()



**1.2 sift特征点匹配**

通过sift得到图片特征点后，一般会进行图片之间的特征点匹配。

**1. 匹配方法**

opencv中的特征点匹配主要有两种方法：BFMatcher，FlannBasedMatcher:

**BFMatcher**

官方文档：<https://docs.opencv.org/4.5.3/d3/da1/classcv_1_1BFMatcher.html>

Brute Froce Matcher: 简称暴力匹配，意思就是尝试所有可能匹配，实现最佳匹配。其继承于类cv2.DescriptorMatcher,

matcher = cv2.BFMatcher(normType=cv2.NORM\_L2, crossCheck=False) # 创建BFMatcher对象

**FlannBasedMatcher**

官方文档：<https://docs.opencv.org/4.5.3/dc/de2/classcv_1_1FlannBasedMatcher.html>

Flann-based descriptor matcher: 最近邻近似匹配。 是一种近似匹配方法，并不追求完美，因此速度更快。 可以调整FlannBasedMatcher参数改变匹配精度或算法速度。其继承于类cv2.DescriptorMatcher。

FLANN\_INDEX\_KDTREE = 0 # 建立FLANN匹配器的参数

indexParams = dict(algorithm=FLANN\_INDEX\_KDTREE, trees=5) # 配置索引，密度树的数量为5

searchParams = dict(checks=50) # 指定递归次数

matcher = cv2.FlannBasedMatcher(indexParams, searchParams) # 建立FlannBasedMatcher对象

indexParams:

algorithm:

FLANN\_INDEX\_LINEAR: 线性暴力(brute-force)搜索

FLANN\_INDEX\_KDTREE: 随机kd树，平行搜索。默认trees=4

FLANN\_INDEX\_KMEANS: 层次k均值树。默认branching=32，iterations=11，centers\_init = CENTERS\_RANDOM, cb\_index =0.2

FLANN\_INDEX\_COMPOSITE: 随机kd树和层次k均值树来构建索引。默认trees =4，branching =32，iterations =11，centers\_init=CENTERS\_RANDOM,cb\_index =0.2

searchParams:

SearchParams (checks=32, eps=0, sorted=true)

checks: 默认32

eps: 默认为0

sorted: 默认True

**2. 绘制匹配**

**KMatch**

官方文档：<https://docs.opencv.org/4.5.3/d4/de0/classcv_1_1DMatch.html>

上述通过匹配方法得到的匹配，在opencv中都用KMatch类表示，其具有几个属性如下：

queryIdx:查询点的索引

trainIdx:被查询点的索引

distance:查询点和被查询点之间的距离

下面代码中，每个点寻找两个最近邻匹配点，即对于kp1中的每个关键点，在kp2中寻找两个和它距离最近的特征点，所以每个关键点产生两组匹配，即两个KMatch类。kp1相当于索引关键点，对应queryIdx； kp2相当于查询关键点，对应trainIdx。

import cv2

img1 = cv2.imread("iphone1.png")

img2 = cv2.imread("iphone2.png")

sift = cv2.SIFT\_create()

kp1, des1 = sift.detectAndCompute(img1, None)

kp2, des2 = sift.detectAndCompute(img2, None)

# 采用暴力匹配

matcher = cv2.BFMatcher()

matches = matcher.knnMatch(des1, des2, k=2) # k=2,表示寻找两个最近邻

# 上面每个点寻找两个最近邻匹配点，即对于kp1中的每个关键点，在kp2中寻找两个和它距离最近的特征点，所以每个关键点产生两组匹配，即两个KMatch类

# kp1相当于索引关键点，对应queryIdx； kp2相当于查询关键点，对应trainIdx

for m in matches: # 若寻找三个最近邻点，则m包括三个KMacth

print(m[0].queryIdx, m[0].queryIdx, m[0].distance) # m[0]表示距离最近的那个匹配

print(m[1].queryIdx, m[1].queryIdx, m[1].distance) # m[1]表示距离第二近的那个匹配

参考：<https://blog.csdn.net/wphkadn/article/details/85805105>

**drawMatches()**

opencv中drawMatches()能绘制匹配特征点

官方文档：<https://docs.opencv.org/4.5.3/d4/d5d/group__features2d__draw.html#ga5d2bafe8c1c45289bc3403a40fb88920>

outImg = cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches1to2, outImg, matchColor, singlePointColor, matchesMask, flags)

参数：

img1：图像1

keypoints1：图像1的特征点

img2：图像2

keypoints1：图像2的特征点

matches1to2：图像1特征点到图像2特征点的匹配，keypoints1[i]和keypoints2[matches[i]]为匹配点

outImg: 绘制完的输出图像

matchColor:匹配特征点和其连线的颜色，-1时表示颜色随机

singlePointColor:未匹配点的颜色，-1时表示颜色随机

matchesMask: mask决定那些匹配点被画出，若为空，则画出所有匹配点

flags: 和上述cv2.drawKeypoints()中flags取值一样

下面代码中对sift提取的特征点进行匹配，采用暴力法匹配，并根据knn近邻法中两个邻居的距离，筛选出了部分匹配点，代码和结果如下：

**import cv2**

**import numpy as np**

**#自己绘制匹配连线**

**def drawMatchesKnn\_cv2(img1, kp1, img2, kp2, goodMatch):**

**h1, w1 = img1.shape[:2]**

**h2, w2 = img2.shape[:2]**

**vis = np.zeros((max(h1, h2), w1 + w2, 3), np.uint8)**

**vis[:h1, :w1] = img1**

**vis[:h2, w1:w1 + w2] = img2**

**p1 = [kpp.queryIdx for kpp in goodMatch]**

**p2 = [kpp.trainIdx for kpp in goodMatch]**

**post1 = np.int32([kp1[pp].pt for pp in p1])**

**post2 = np.int32([kp2[pp].pt for pp in p2]) + (w1, 0)**

**for (x1, y1), (x2, y2) in zip(post1, post2):**

**cv2.line(vis, (x1, y1), (x2, y2), (0, 0, 255))**

**cv2.imshow("match", vis)**

**img1 = cv2.imread("iphone1.png")**

**img2 = cv2.imread("iphone2.png")**

**sift = cv2.SIFT\_create()**

**kp1, des1 = sift.detectAndCompute(img1, None)**

**kp2, des2 = sift.detectAndCompute(img2, None)**

**# 采用暴力匹配**

**matcher = cv2.BFMatcher()**

**matches = matcher.knnMatch(des1, des2, k=2) # k=2,表示寻找两个最近邻**

**# 采用最近邻近似匹配**

**# FLANN\_INDEX\_KDTREE = 0 # 建立FLANN匹配器的参数**

**# indexParams = dict(algorithm=FLANN\_INDEX\_KDTREE, trees=5) # 配置索引，密度树的数量为5**

**# searchParams = dict(checks=50) # 指定递归次数**

**# matcher = cv2.FlannBasedMatcher(indexParams, searchParams) # 建立FlannBasedMatcher对象**

**# matches = matcher.knnMatch(des1, des2, k=2) # k=2,表示寻找两个最近邻**

**h1, w1 = img1.shape[:2]**

**h2, w2 = img2.shape[:2]**

**out\_img1 = np.zeros((max(h1, h2), w1 + w2, 3), np.uint8)**

**out\_img1[:h1, :w1] = img1**

**out\_img1[:h2, w1:w1 + w2] = img2**

**out\_img1 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, matches, out\_img1)**

**good\_match = []**

**for m, n in matches:**

**if m.distance < 0.5\*n.distance: # 如果第一个邻近距离比第二个邻近距离的0.5倍小，则保留**

**good\_match.append(m)**

**out\_img2 = np.zeros((max(h1, h2), w1 + w2, 3), np.uint8)**

**out\_img2[:h1, :w1] = img1**

**out\_img2[:h2, w1:w1 + w2] = img2**

**# p1 = [kp1[kpp.queryIdx] for kpp in good\_match] # kp1中挑选处的关键点**

**# p2 = [kp2[kpp.trainIdx] for kpp in good\_match] # kp2中挑选处的关键点**

**out\_img2 = cv2.drawMatches(img1, kp1, img2, kp2, good\_match, out\_img2)**

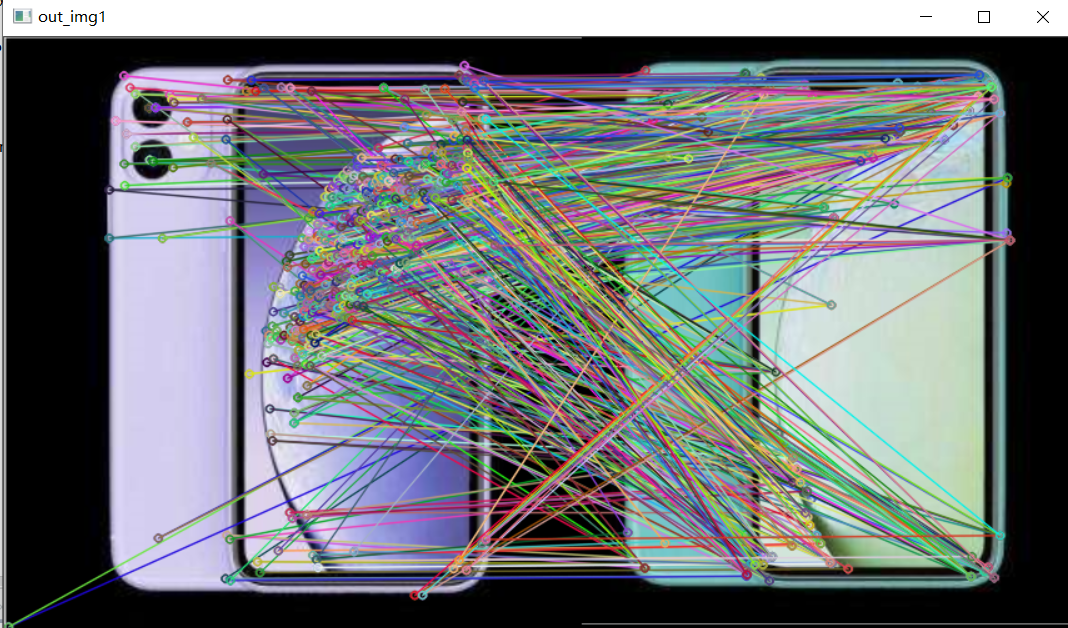
**# drawMatchesKnn\_cv2(img1, kp1, img2, kp2, good\_match)**

**cv2.imshow("out\_img1", out\_img1)**

**cv2.imshow("out\_img2", out\_img2)**

**cv2.waitKey(0)**

**cv2.destroyAllWindows()**





参考：<https://www.cnblogs.com/wangguchangqing/p/4333873.html>

<https://blog.csdn.net/claroja/article/details/83411108>

分类: [opencv和计算机视觉](https://www.cnblogs.com/silence-cho/category/1480451.html)