目录

[一 SIFT概述 2](#_Toc9908)

[1 SIFT 算法的特点 2](#_Toc14335)

[2 SIFT 算法可以解决的问题 2](#_Toc3766)

[3 SIFT 算法步骤 2](#_Toc25847)

[二 SIFT详细流程及编程思路 4](#_Toc9331)

[1 尺度空间极值检测 4](#_Toc27379)

[1.1 高斯模糊 4](#_Toc25994)

[1.2 尺度空间理论 7](#_Toc13485)

[1.3 空间极值点检测(关键点的初步探查） 16](#_Toc20225)

[2 关键点定位 17](#_Toc23423)

[2.1 关键点的精确定位 17](#_Toc2953)

[2.2 消除边缘响应 23](#_Toc782)

[3 关键点的方向分配 24](#_Toc11525)

[3.1 幅值和角度 24](#_Toc18236)

[3.2 拟合二次曲线过程 25](#_Toc4915)

[3.3 关键点结果展示 26](#_Toc30378)

[3.4 消除重复的关键点 28](#_Toc28834)

[4 关键点特征描述 29](#_Toc26196)

[4.1确定计算描述子所需的图像区域 29](#_Toc10334)

[4.2 将坐标轴旋转为关键点的方向（旋转不变性） 30](#_Toc20240)

[4.3 梯度直方图的生成 30](#_Toc20985)

[4.4 三线性插值 31](#_Toc27633)

[4.5 归一化特征向量 33](#_Toc24281)

[5 SIFT特点总结 33](#_Toc2275)

[6 SIFT参考资料 34](#_Toc29424)

[三 图片拼接 35](#_Toc27382)

[1 关键点匹配 35](#_Toc21584)

[1.1 Lowe's算法 35](#_Toc2097)

[1.2 RANSAC算法 35](#_Toc12338)

[1.3 匹配结果 36](#_Toc29422)

[2 图像配准 37](#_Toc10607)

[3 配准结果 37](#_Toc15617)

[四 代码 40](#_Toc888)

[tools.py代码 41](#_Toc31041)

[sift.py代码 49](#_Toc5706)

[splice.py代码 60](#_Toc30251)

# 一 SIFT概述

尺度不变特征转换(Scale-invariant feature transform 或SIFT)是一种电脑视觉的算法用来侦测与描述影像中的局部性特征，它在空间尺度中寻找极值点，并提取出其位置、尺度、旋转不变量，此算法由David Lowe 在1999 年所发表，2004 年完善总结。

局部影像特征的描述与侦测可以帮助辨识物体，SIFT 特征是基于物体上的一些局部外观的兴趣点而与影像的大小和旋转无关。对于光线、噪声、些微视角改变的容忍度也相当高。基于这些特性，它们是高度显著而且相对容易撷取，在母数庞大的特征数据库中，很容易辨识物体而且鲜有误认。使用SIFT 特征描述对于部分物体遮蔽的侦测率也相当高，甚至只需要3 个以上的SIFT 物体特征就足以计算出位置与方位。在现今的电脑硬件速度下和小型的特征数据库条件下，辨识速度可接近即时运算。SIFT 特征的信息量大，适合在海量数据库中快速准确匹配。

## 1 SIFT 算法的特点

（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）独特性（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。

## 2 SIFT 算法可以解决的问题

目标的自身状态、场景所处的环境和成像器材的成像特性等因素影响图像配准/目标识别跟踪的性能。而SIFT 算法在一定程度上可解决：

（1）目标的旋转、缩放、平移（RST）

（2）图像仿射/投影变换（视点viewpoint）

（3）光照影响（illumination）

（4）目标遮挡（occlusion）

（5）杂物场景（clutter）

（6）噪声

SIFT 算法的实质是在不同的尺度空间上查找关键点(特征点)，并计算出关键点的方向。

SIFT 所查找到的关键点是一些十分突出，不会因光照，仿射变换和噪音等因素而变化的点，如角点、边缘点、暗区的亮点及亮区的暗点等。

## 3 SIFT 算法步骤

Lowe 将SIFT 算法分解为如下四步：

（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。

# 二 SIFT详细流程及编程思路

## 1 尺度空间极值检测

### 1.1 高斯模糊

#### 1.1.1 高斯模糊理论基础

高斯模糊是一种图像低通滤波器，它使用正态分布（高斯函数）计算模糊模板，并使用该模板与原图像做卷积运算，达到模糊图像的目的。



其中，是正态分布的标准差，值越大，图像越模糊（平滑）。m和n为模板的尺寸，x和y为模板上的的坐标点。

在二维空间中，这个公式生成的曲面的等高线是从中心开始呈正态分布的同心圆，如图1所示。分布不为零的像素组成的卷积矩阵与原始图像做卷积变换。每个像素的值都是周围相邻像素值的加权平均。原始像素的值有最大的高斯分布值，所以有最大的权重，相邻像素随着距离原始像素越来越远，其权重也越来越小。这样进行模糊处理比其它的均衡模糊滤波器更高地保留了边缘效果。

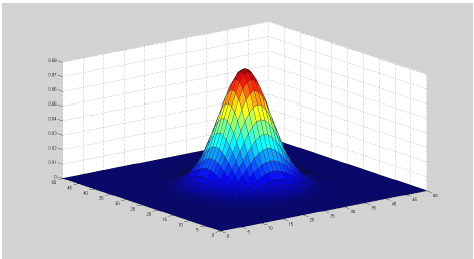


图1 二维高斯曲面

理论上来讲，图像中每点的分布都不为零，这也就是说每个像素的计算都需要包含周围图像。在实际应用中，在计算高斯函数的离散近似时，在大概3σ距离之外的像素都可以看作不起作用（类似正态分布的3σ原则），这些像素的计算也就可以忽略。通常，图像处理程序只需要计算(6σ +1)×(6σ +1)的矩阵就可以保证相关像素影响。

同时根据上图也可以看出，σ决定了一个像素要参考周围像素点的比例。也就是说σ决定了图1中各个同心圆的大小。所以如果图片边长缩小一倍σ不变，对于一个像素点，要参考周围像素点的比例增大了。相当于在原图片上增大了σ。

#### 1.1.2 二维高斯滤波器

下面是一个5X5，σ为0.6的高斯卷积核：

**6.58573478e-06 4.24780506e-04 1.70353617e-03 4.24780506e-04 6.58573478e-06**

**4.24780506e-04 2.73983822e-02 1.09878241e-01 2.73983822e-02 4.24780506e-04**

**1.70353617e-03 1.09878241e-01 4.40654774e-01 1.09878241e-01 1.70353617e-03**

**4.24780506e-04 2.73983822e-02 1.09878241e-01 2.73983822e-02 4.24780506e-04**

**6.58573478e-06 4.24780506e-04 1.70353617e-03 4.24780506e-04 6.58573478e-06**

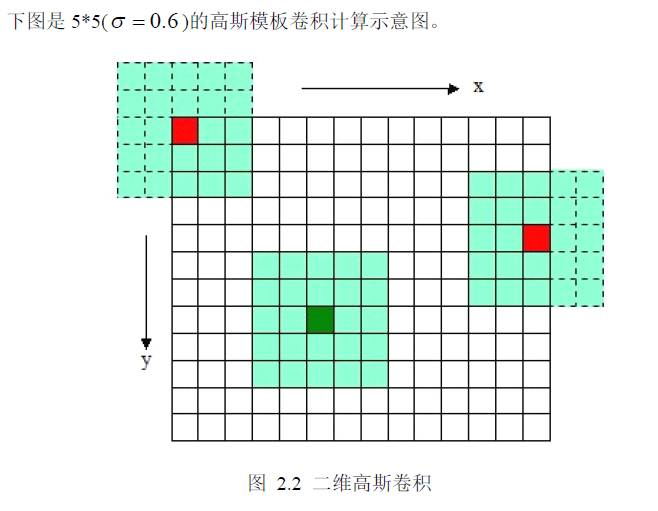


图2 二维高斯卷积图

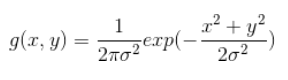
#### 1.1.3 分离高斯卷积

上面这样直接与图像卷积，速度比较慢，同时图像边缘信息也会损失严重。后来，不知哪位学者发现，可以使用分离的高斯卷积（即先用1xN的模板沿着X方向对图像卷积一次，然后用Nx1的模板沿着Y方向对图像再卷积一次，其中N=[（6σ+1）]且向上取最邻近奇数），这样既省时也减小了直接卷积对图像边缘信息的严重损失。



图3 二维分离高斯卷积图

#### 1.1.4 高斯核的实现



卷积的思路已经在之前的报告中写过，在此不介绍。首先由于之前我写的卷积函数限制，卷积核边长为奇数。

（1）输入(卷积核边长KernalSize, σ)

（2）获得中心坐标CenterIndex = KernalSize // 2（除2取整）

（3）遍历卷积核得到(x, y)，x=i-CenterIndex，y=j-CenterIndex (i,j从0到KernalSize-1)

（4）如果σ等于0（上面的公式用不了），σ=（（KernalSize-1）\*0.5-1）\*0.3+0.8（来源于opencv源码）

（5）至此x，y，σ都得到了，根据上面的高斯核公式计算，得到一个KernalSize\*KernalSize大小的二维矩阵

（6）归一化

**注：**分离高斯卷积核与二维高斯卷积核类似，只需把x或y等于0.

#### 1.1.5 分离高斯卷积结果

原图片:

[[ 1 2 3 4 5 6 7]

[ 8 9 10 11 12 13 14]

[15 16 17 18 19 20 21]

[22 23 24 25 26 27 28]

[29 30 31 32 33 34 35]

[36 37 38 39 40 41 42]

[43 44 45 46 47 48 49]]

高斯卷积：

[[ 4.56 5.11 6.11 7.11 8.11 9.11 9.67]

[ 8.44 9. 10. 11. 12. 13. 13.56]

[15.44 16. 17. 18. 19. 20. 20.56]

[22.44 23. 24. 25. 26. 27. 27.56]

[29.44 30. 31. 32. 33. 34. 34.56]

[36.44 37. 38. 39. 40. 41. 41.56]

[40.33 40.89 41.89 42.89 43.89 44.89 45.44]]

分离高斯卷积：

[[12.5 5.11 4.23 4.92 5.62 6.31 4.63]

[ 7.54 9. 10. 11. 12. 13. 9.38]

[11.49 16. 17. 18. 19. 20. 14.23]

[15.54 23. 24. 25. 26. 27. 19.07]

[20.38 30. 31. 32. 33. 34. 23.92]

[25.23 37. 38. 39. 40. 41. 28.76]

[19.32 28.3 28.99 29.69 30.38 31.07 21.77]]

cv2:

[[ 6 6 7 8 9 10 11]

[ 9 9 10 11 12 13 13]

[16 16 17 18 19 20 20]

[23 23 24 25 26 27 27]

[30 30 31 32 33 34 34]

[37 37 38 39 40 41 41]

[39 40 41 42 43 44 44]]

可以看到中间的结果完全一样，边缘由于卷积的边缘处理不同有所不同。同时速度会有较好的提升。

### 1.2 尺度空间理论

#### 1.2.1 尺度空间理论

尺度空间使用高斯金字塔表示。Tony Lindeberg指出尺度规范化的LOG(Laplacion of Gaussian)算子具有真正的尺度不变性。Lowe使用高斯金字塔近似LOG算法，在尺度空间检测稳定的关键点。

尺度空间理论的基本思想是：在图像信息处理模型中引入一个被视为尺度的参数，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，对这些序列进行尺度空间主轮廓的提取，并以该主轮廓作为一种特征向量，实现边缘、角点检测和不同分辨率上的特征提取等。

尺度空间方法将传统的单尺度图像信息处理技术纳入尺度不断变化的动态分析框架中，更容易获取图像的本质特征。尺度空间中各尺度图像的模糊程度逐渐变大，能够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。

尺度空间满足视觉不变性。该不变性的视觉解释如下：当我们用眼睛观察物体时，一方面当物体所处背景的光照条件变化时，视网膜感知图像的亮度水平和对比度是不同的，因此要求尺度空间算子对图像的分析不受图像的灰度水平和对比度变化的影响，即满足灰度不变性和对比度不变性。另一方面，相对于某一固定坐标系，当观察者和物体之间的相对位置变化时，视网膜所感知的图像的位置、大小、角度和形状是不同的，因此要求尺度空间算子对图像的分析和图像的位置、大小、角度以及仿射变换无关，即满足平移不变性、尺度不变性、欧几里德不变性以及仿射不变性。

#### 1.2.2 图像尺度空间表示

一个图像的尺度空间,定义为一个变化尺度的高斯函数与原图像的卷积。

 (3-1)

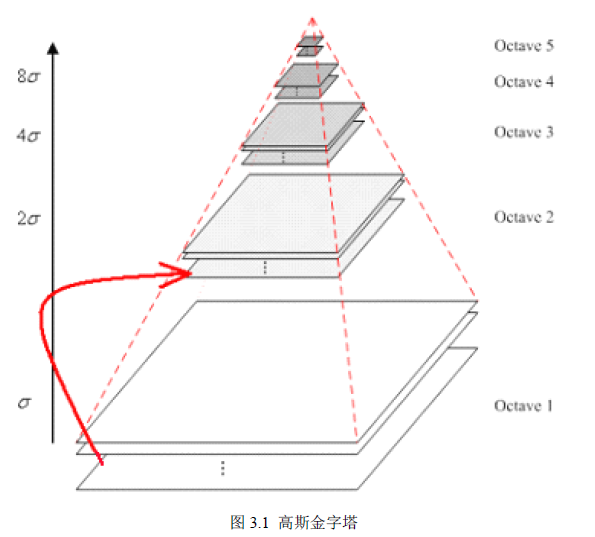
其中，\*表示卷积。m，n表示高斯模板的维度(由(6σ +1)×(6σ +1)确定)。(x, y)代表图像的像素位置。σ 是尺度空间因子，值越小表示图像被平滑的越少，相应的尺度也就越小。大尺度对应于图像的概貌特征，小尺度对应于图像的细节特征。

#### 1.2.3 高斯金字塔的构建

尺度空间在实现时使用高斯金字塔表示，高斯金字塔的构建分为两部分：

1. 对图像做不同尺度的高斯模糊

2. 对图像做降采样(隔点采样)

****

图像的金字塔模型是指，将原始图像不断降阶采样，得到一系列大小不一的图像，由大到小，从下到上构成的塔状模型。原图像为金子塔的第一层，每次降采样所得到的新图像为金字塔的一层(每层一张图像)，每个金字塔共n 层。金字塔的层数根据图像的原始大小和塔顶图像的大小共同决定，其计算公式如下：

 (1-2)

其中M，N 为原图像的大小,t 为塔顶图像的最小维数的对数值。如，对于大小为512\*512的图像，金字塔上各层图像的大小如表3.1 所示，当塔顶图像为4\*4 时，n=7，当塔顶图像为2\*2 时，n=8。



为了让尺度体现其连续性，高斯金字塔在简单降采样的基础上加上了高斯滤波。如图3.1所示，将图像金字塔每层的一张图像使用不同参数σ 做高斯模糊，使得金字塔的每层含有多张高斯模糊图像，将金字塔每层多张图像合称为一组(Octave)，金字塔每层只有一组图像，组数和金字塔层数相等，使用公式(1-2)计算，每组含有多张(也叫层Interval)图像。另外，降采样时，高斯金字塔上一组图像的初始图像(底层图像)是由前一组图像的倒数第三张图像隔点采样得到的。

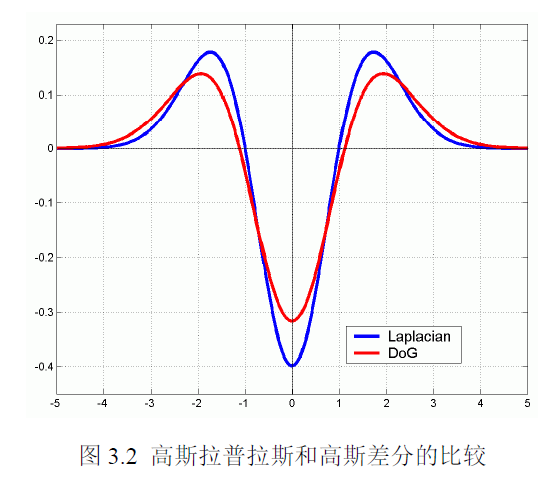
**注：**由于组内的多张图像按层次叠放，因此组内的多张图像也称做多层，为避免与金字塔层的概念混淆，本文以下内容中，若不特别说明是金字塔层数，层一般指组内各层图像。

**注：**如下节所示，为了在每组中检测S 个尺度的极值点，则DOG 金字塔每组需S+2层图像，而DOG 金字塔由高斯金字塔相邻两层相减得到，则高斯金字塔每组需S+3 层图像，实际计算时S 在3 到5 之间。

#### 1.2.4 高斯差分金字塔理论知识

2002年Mikolajczyk在详细的实验比较中发现尺度归一化的高斯拉普拉斯函数σ 的极大值和极小值同其它的特征提取函数，例如：梯度，Hessian 或Harris 角特征比较，能够产生最稳定的图像特征。

而Lindeberg 早在1994 年就发现高斯差分函数（Difference of Gaussian ，简称DOG 算子）与尺度归一化的高斯拉普拉斯函数非常近似。



如图3.2 所示，红色曲线表示的是高斯差分算子，而蓝色曲线表示的是高斯拉普拉斯算子。所以Lowe 使用更高效的高斯差分算子代替拉普拉斯算子进行极值检测，如下：

 (3-4)

其中D(x, y, σ) 和 的关系可以从如下公式推导得到：



利用差分近似代替微分，则有：

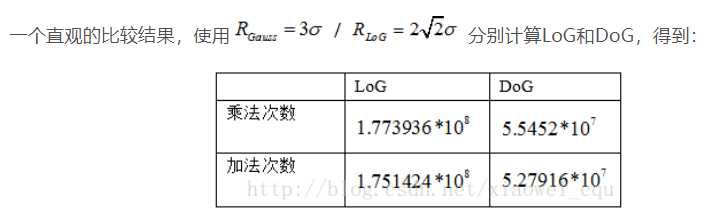


因此有



其中k-1是个常数，并不影响极值点位置的求取。此公式也表明，相邻两个尺度坐标之间的尺度值分别为：，，即高一层尺度与低一层尺度的尺度值之差为平方之后乘一个系数k。设定S为进行极值检测时候的Octave的尺度数目，S=3。设定每组尺度空间从S=1至S=3的尺度差为整数2，因此。

同时使用DoG能够显著减少运算次数：



在实际计算时，使用高斯金字塔每组中相邻上下两层图像相减，得到高斯差分图像，如图3.3 所示，进行极值检测。



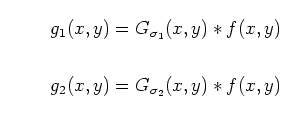
**注：**关于DoG和LoG的具体解释如下：

**DoG：**

首先，高斯函数表示定义为：

IMG_256

其次，两幅图像的高斯滤波表示为：



最后，将上面滤波得到的两幅图像g1和g2相减得到：

IMG_258

即：DOG表达式为：

IMG_259

**LoG：**

首先高斯函数的定义是：

IMG_256

然后对图像进行高斯滤波：

IMG_257

因为卷积的性质：

IMG_256

LoG可以通过先对高斯函数进行偏导操作，然后进行卷积求解。公式表示为：

IMG_258

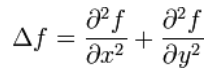
和

IMG_259

最后对滤波后的图像进行拉普拉斯运算(图中三角形表示)

IMG_258

图中三角形（拉普拉斯算子）的表达式为：



因此，我们可以LOG核函数表达式为：

IMG_260

**LoG和DoG：**

IMG_259

IMG_260

将高斯函数带入后并归一化的LoG算子（将LoG算子用G来表达）：



而DoG算子：



所以可以用DoG近似LoG。

#### 1.2.5 参数的确定

（1）每组图片的层数(Interval)S=3

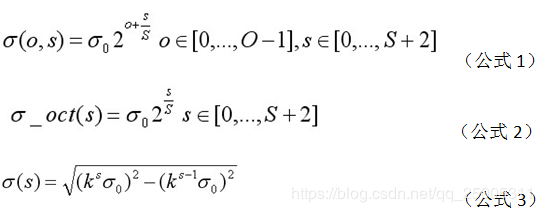
在每组高斯金字塔中需要检测的S个尺度的极值点数目。由于DOG金字塔每组需要S+2个。而DOG 金字塔由高斯金字塔相邻两层相减得到，则高斯金字塔每组需S+3 层图像，实际计算时S 在3 到5 之间。

Lowe选取S=3，即每个Octave有3层，因为：S=3时，keypoints有最高的repeatablity。

虽然当S增大时，keypoint的数量增加（某些情况下希望有较大的数量而不计较匹配成功率），但是必须考虑到，S=3时计算开销比较小。

（2）高斯金字塔组(Octave)数O，使用1.2.3高斯金字塔构建中的公式(1-2)计算

（3）尺度空间坐标



其中，。

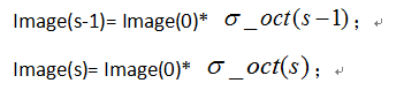
权衡keypoints的repeatability和计算开销，选取=1.6。Lowe建议=1.6（考虑到相机已经对图片进行=0.5的模糊，有时）

OpenCV中参数采用了下面3种形式计算，这让人很不理解，按照理论，应该是1或2的形式才对。根据文献解释为：提高速度（即只需要算第0组共S个）。

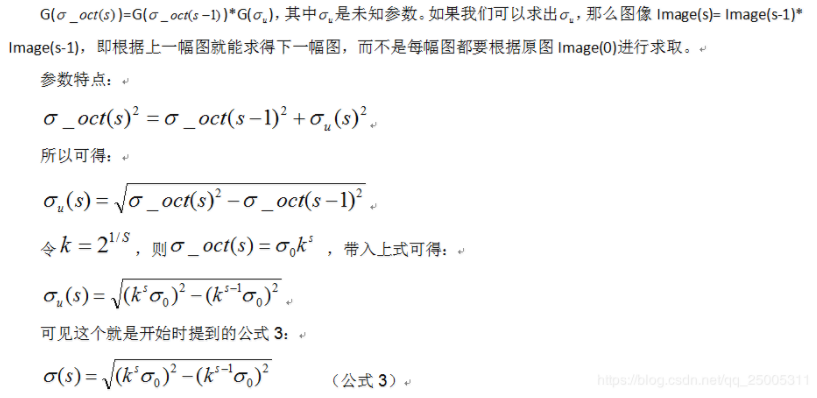
**注：**下面分析如何提速的：

高斯函数的特殊性质，即两个高斯函数的卷积等于另一个高斯函数，关系如下：  
 IMG_256

首先，看一下第一组内第s-1层和第s层图像：



根据高斯函数的性质：

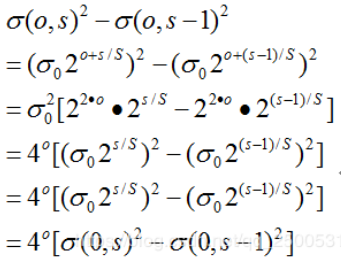


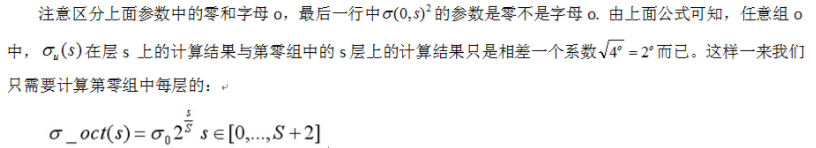
有了公式3，就可以根据上一幅图片构造下一副图片。而不是使用最原始的图片进行卷积获得新图片。但是这又有什么用呢？每个新的照片还是由一幅图像和一个卷积核卷积而得到，并没有简化计算。

这就要说下面的性质：

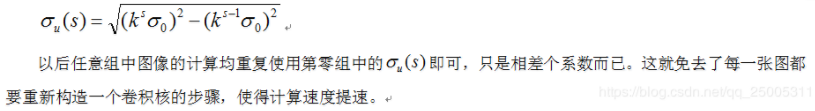
已知：

推导任意组相邻层之间的：





然后在计算相邻两层之间的：



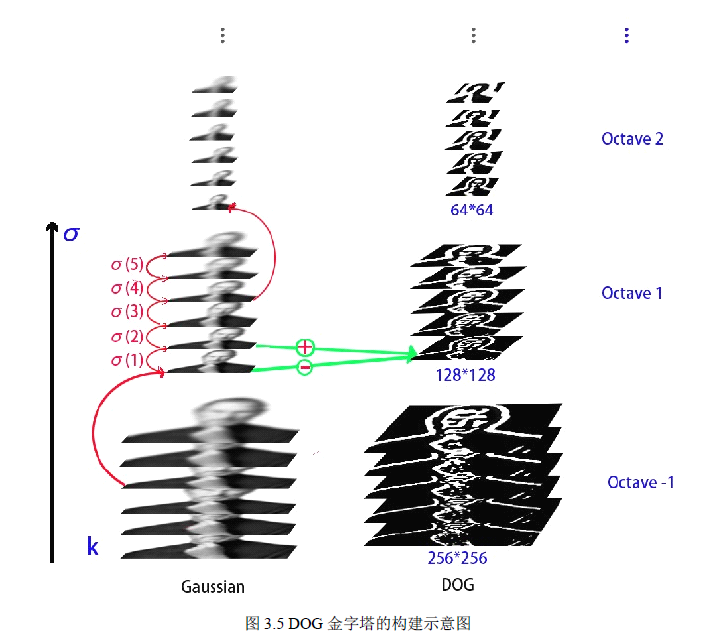
最后，由于组数的增加以2的倍数增长，但是随着组数的增加图像的尺寸确实以2的倍数减小，二者相互抵消。所以迭代过程中事实上采用的都是同一组数据。即公式3简化了组与组之间的计算。

（3）高斯模糊偏差

在最开始建立高斯金字塔时，要预先模糊输入图像来作为第0 个组的第0 层的图像，这时相当于丢弃了最高的空域的采样率。因此通常的做法是先将图像的尺度扩大一倍来生成第-1 组。

我们假定初始的输入图像为了抗击混淆现象，已经对其进行的高斯模糊，如果输入图像的尺寸用双线性插值扩大一倍，那么为了保持效果不变，相当于。即Octave-1层：

#### 1.2.6 高斯差分金字塔的构建



**高斯差分金字塔编程思路：**

1. 首先根据上节‘参数的确定’，得到S，O，第0组。
2. 将原始图片扩大2倍，并依次用进行卷积，每次卷积使用的是上一次生成的图片（原因也在上一节‘（3）尺度空间坐标’）
3. 如图3.5所示，对组内倒数第3张图片进行步长为2的降采样。
4. 重复上述过程，得到高斯金字塔
5. 如图3.5所示，高斯金字塔对应组内上下两张图片相减，得到差分高斯金字塔

### 1.3 空间极值点检测(关键点的初步探查）

关键点是由DOG 空间的局部极值点组成的，关键点的初步探查是通过同一组内各DoG相邻两层图像之间比较完成的。为了寻找DoG 函数的极值点，每一个像素点要和它所有的相邻点比较，看其是否比它的图像域和尺度域的相邻点大或者小。如图3.6 所示，中间的检测点和它同尺度的8 个相邻点和上下相邻尺度对应的9×2 个点共26 个点比较，以确保在尺度空间和二维图像空间都检测到极值点。



图3.6 DOG 空间极值检测

由于要在相邻尺度进行比较，如图3.5 右侧每组含5层的高斯差分金子塔，只能在中间三层中进行三个尺度的极值点检测，其它尺度则只能在不同组中进行。为了在每组中检测S个尺度的极值点，则DOG 金字塔每组需S+2 层图像，而DOG 金字塔由高斯金字塔相邻两层相减得到，则高斯金字塔每组需S+3 层图像，实际计算时S 在3 到5 之间。

当然这样产生的极值点并不全都是稳定的特征点，因为某些极值点响应较弱，而且DOG算子会产生较强的边缘响应。

**编程思路：**

1. 在上一节已经得到了差分高斯金字塔，可以通过Python的切片操作来获得对应矩阵，即[ FirstImg[i-1:i+2, j-1:j+2], SecondImg[i-1:i+2, j-1:j+2], ThirdImg[i-1:i+2, j-1:j+2] ]。
2. 对得到3X3X3矩阵，进行3次遍历得到，如果中心点是极值（极大或极小）。则保留坐标。
3. 在Opencv源码里，极值点的像素值还要大于一个Threshold。Threshold的计算为0.5 \* 0.04/ S\* 255（0.04的来源在下一节2.1（2）中有介绍）

**注：**OpenCV也是为了过滤噪音点，多加了一个Threshold

## 2 关键点定位

以上方法检测到的极值点是离散空间的极值点，以下通过拟合三维二次函数来精确确定关键点的位置和尺度，同时去除低对比度的关键点和不稳定的边缘响应点(因为DoG 算子会产生较强的边缘响应)，以增强匹配稳定性、提高抗噪声能力。

寻找到极值点后，保存所有Octave中每个极值点（keypoint candidate）的坐标和尺度。

### 2.1 关键点的精确定位

**（1）泰勒级数**

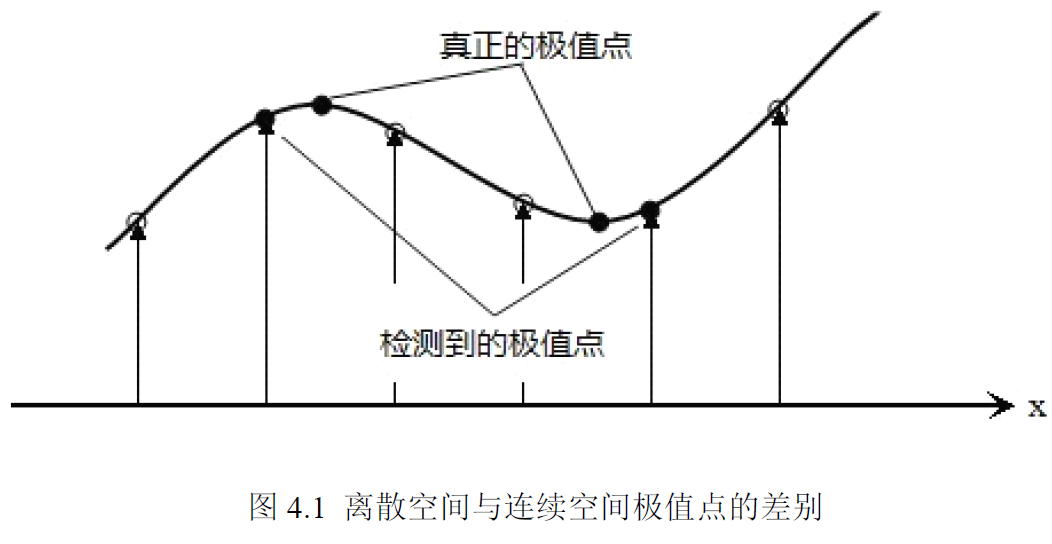
若f(x)在x0点的某个邻域内具有n+1阶导数，则在该邻域内有n阶泰勒级数：



其中表示f(x)的二阶导数在点x0处的值，这是一个具体的值，而不是一个函数。

**（2）DoG 函数在尺度空间的Taylor 展开式**

离散空间的极值点并不是真正的极值点，图4.1 显示了二维函数离散空间得到的极值点与连续空间极值点的差别。利用已知的离散空间点插值得到的连续空间极值点的方法叫做子像素插值（Sub-pixel Interpolation）。



为了提高关键点的稳定性，需要对尺度空间DoG 函数进行曲线拟合。利用DoG 函数在尺度空间的Taylor 展开式(拟合函数)为：

 (2-1)

即：



上式中D和D的各阶导数是在X0处的取值，即表示。

其中，X为偏离泰勒级数定点的偏离量，上式中各个项的具体定义如下：







令方程(2-1)对X求导，得到以下方程：



上式等于：



令上式两边等于零，可以得到以下方程：



可以得到极值点的偏移量为：

 (2-2)

将方程(2-2)代入方程(2-1)中可得对应极值点，方程的值为：

 (2-3)

其中, 代表相对插值中心的偏移量，设置离散化的步长h=1。当X在任一维度上的偏移量大于0.5时（即x 或y 或 ），意味着插值中心已经偏移到它的邻近点上，所以必须改变当前关键点的位置。同时在新的位置上反复插值直到收敛；也有可能超出所设定的迭代次数或者超出图像边界的范围，此时这样的点应该删除，在Lowe 中进行了5 次迭代。另外，过小的点易受噪声的干扰而变得不稳定，所以将小于某个经验值(Lowe 论文中使用0.03，RobHess 等人实现时使用0.04/S)的极值点删除。同时，在此过程中获取特征点的精确位置(原位置加上拟合的偏移量)以及尺度(和)。

**注：**这里的0.04就是上一节编程思路中的0.04

**（3）极值点的偏移量的求解：**

根据方程(2-2)可得：



即：



**（4）三阶矩阵求逆公式**

高阶矩阵的求逆算法主要有归一法和消元法两种，现将三阶矩阵求逆公式总结如下：

若矩阵



可逆，即



那么：



**（5）有限差分求导法**

有限差分法以变量离散取值后对应的函数值来近似微分方程中独立变量的连续取值。在有限差分方法中，我们放弃了微分方程中独立变量可以取连续值的特征，而关注独立变量离散取值后对应的函数值。但是从原则上说，这种方法仍然可以达到任意满意的计算精度。因为方程的连续数值解可以通过减小独立变量离散取值的间格，或者通过离散点上的函数值插值计算来近似得到。这种方法是随着计算机的诞生和应用而发展起来的。其计算格式和程序的设计都比较直观和简单，因而，它在计算数学中使用广泛。

有限差分法的具体操作分为两个部分：

1. 用差分代替微分方程中的微分，将连续变化的变量离散化，从而得到差分方程组的数学形式；

2. 求解差分方程组。

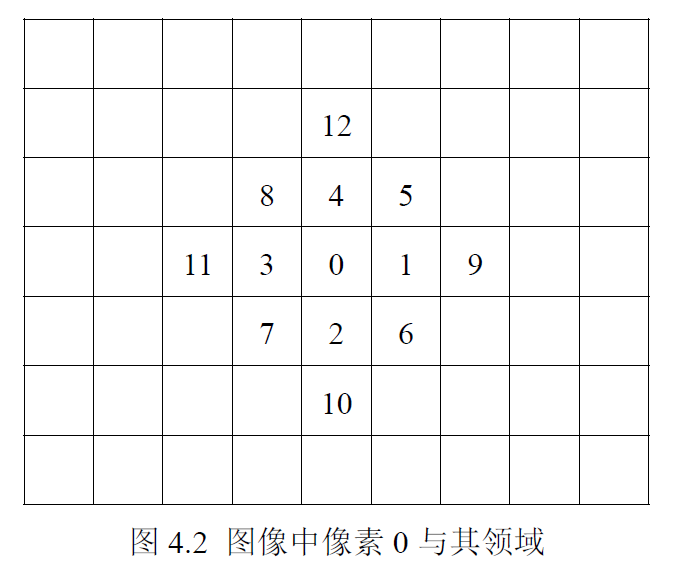
一个函数在x 点上的一阶和二阶微商，可以近似地用它所临近的两点上的函数值的差分来表示。如对一个单变量函数f (x )，x 为定义在区间[a,b]上的连续变量，以步长h = Δx 将[a,b]区间离散化，我们会得到一系列节点

：节点xi的一阶向前差分

：节点xi 的一阶向后差分

： 节点xi的一阶中心差分

本文使用中心差分法利用泰勒展开式求解第四节所使用的导数。图像中中心像素位置0与其邻域坐标如下图所示：



取h=1，则关于图像中心像素0的各种导数差分计算公式如下：











上式可有以下推导得出：

函数f(x)在x0处的泰勒展开式为;



则：



忽略h平方之后的项，联立方程(4-9)和(4-10)可得：

由减去可得：



由加上可得：



当绝对值 大于阈值Thre时，当前采样点（极值点）被接受，否则被认为对比度低而剔除。Lowe假设像素范围为[0,1]，取阈值Thre为0.03。

**注：**3X3X3的矩阵要除255，原因为Lowe假设像素范围为[0,1]

**（6）关键点的精确定位编程思路：**

1. 首先确定相关参数Thre、迭代次数NumOfAttempts=5
2. 对于求得的每一个极值点，计算hessian阵和梯度（计算方法同上面的介绍）
3. 根据公式（2-2）计算偏移量，如果有任何一维的绝对值大于0.5，则需要更新后再次计算。
4. 如果任何一维超出了范围，则放弃极值点（x的范围为[0-Width],y的范围为[0-Height],ImageIndex的范围为[1-3]，即不要上下两层）
5. 重复上述过程，如果在迭代次数内算出来了，则找到了关键点精确定位
6. 对于新的关键点，根据公式（2-3）求新的值

**注：**这里要注意，不一定退出循环就是关键点。只有偏移量的任何一维小于0.5，这时退出循环，才是想要的关键点坐标，这时的坐标也有可能是五。而迭代了五次没找到关键点也会退出循环并返回一个值，这时得到的并不是想要的关键点。

**注：**从这里开始的计算，都应该包括金字塔-1组（与OpenCV一致），因为多数关键点都是存在于-1组的

Octave0，Sum:424.0

Octave1，Sum:138.0

Octave2，Sum:63.0

Octave3，Sum:6.0

Octave4，Sum:0.0

Octave5，Sum:0.0

Octave6，Sum:0.0

Octave7，Sum:0.0

可以看到-1组的关键点最多。并且在后续删除重复关键点（3.3节介绍）后，共有622个关键点，而OpenCV求得的关键点数为603。

### 2.2 消除边缘响应

一个定义不好的高斯差分算子的极值在横跨边缘的地方有较大的主曲率，而在垂直边缘的方向有较小的主曲率。

DOG 算子会产生较强的边缘响应，需要剔除不稳定的边缘响应点。获取特征点处的Hessian 矩阵，主曲率通过一个2x2 的Hessian 矩阵H 求出：



H 的特征值和代表x和y方向的梯度，



Tr (H )表示矩阵H 对角线元素之和，Det (H )表示矩阵H 的行列式。假设是较大的特征值，而是较小的特征值，令，则



导数由采样点相邻差估计得到,在下上节中已经说明。

D 的主曲率和H 的特征值成正比，令为最大特征值，为最小的特征值，则公式的值在两个特征值相等时最小，随着的增大而增大。值越大，说明两个特征值的比值越大，即在某一个方向的梯度值越大，而在另一个方向的梯度值越小，而边缘恰恰就是这种情况。所以为了剔除边缘响应点，需要让该比值小于一定的阈值，因此，为了检测主曲率是否在某域值下，只需检测

 公式（2-4）

在Lowe 的文章中，取＝10。

**消除边缘响应编程思路：**

1. 上一步已经得到了3X3的Hessian阵，切片截取左上角的dxx，dxy，dyy
2. 按照上面公式（2-4） 进行筛选，得到的就是真正的关键点坐标了。

**注：**每个关键点要保存坐标(x,y),金字塔层数o,尺寸坐标,并且保存的是原始图片的，也就是x,y要乘（与OpenCV一致）

**注：**这里关键点的储存使用cv2.KeyPoint。但是KeyPoint类里不包含层数s，而后面需要通过尺度反过来求s时（第4节），如果直接对计算得到的s取整，有时会跟真正的s对不上（通过输出发现是计算上的细微偏差导致的）。我本来是打算继承KeyPoint类，并添加一个属性s和==方法（def \_\_eq\_\_()，用时删除重复关键点（第3.4节）时的判断），尝试后发现cv2不允许。然后我用另外一个列表单独存储。后来对照OpenCV的代码得到一个完全意想不到的方法：在KeyPoint的Octave里，分成24位，0-7位存OctaveIndex，8-15位存ImgIndex，16-24位存偏移量。

## 3 关键点的方向分配

对上面提取的每个关键点，围绕该点**在对应高斯金字塔的图片里**选择一个窗口（圆形区域），窗口内各采样点的梯度方向构成一个方向直方图，根据直方图的峰值确定关键点的方向。关键点的尺度用来选择哪个高斯滤波图像参与计算，还用来决定窗口的大小——为了保证不同尺度下的同一关键点的方向都包含相同的信息量，那么窗口的大小必然不一样：同一个原始图像，尺度越大，窗口应该越大；反之，如果窗口大小不变，尺度越大的图像，该窗口内的信息越少。

L 为关键点所在的尺度空间值，按Lowe 的建议，梯度的模值m(x, y)按的高斯分布加成，按尺度采样的原则(高斯函数的原则：高斯函数的距离之外的像素都可以看作不起作用)，领域窗口半径为。

每个累加到梯度方向直方图的采样点的梯度值都要进行权重处理，加权采用圆形高斯加权函数，其标准偏差为。**SIFT算法只考虑了尺度和旋转不变性，并没有考虑仿射不变性。通过高斯加权，使特征点附近的梯度幅值有较大的权重，这样可以部分弥补仿射不变性而产生的特征点不稳定的问题。**

因此，选取的窗口大小和关键点尺度成正比。这样提取的信息，就具有尺度不变性。

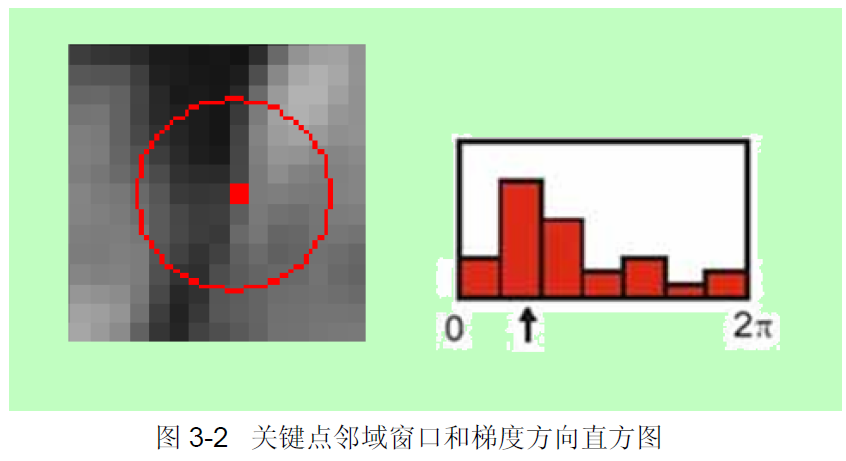
### 3.1 幅值和角度

对于窗口内的每个采样点L(x,y)，其梯度向量的幅度和方向m(x,y ),θ(x,y)公式为：



注意到上面公式定义的的并不是幅度的标准定义，各维度上都少了一个系数0.5，但这对后面的计算没有影响（所有幅度同等比例变化）。

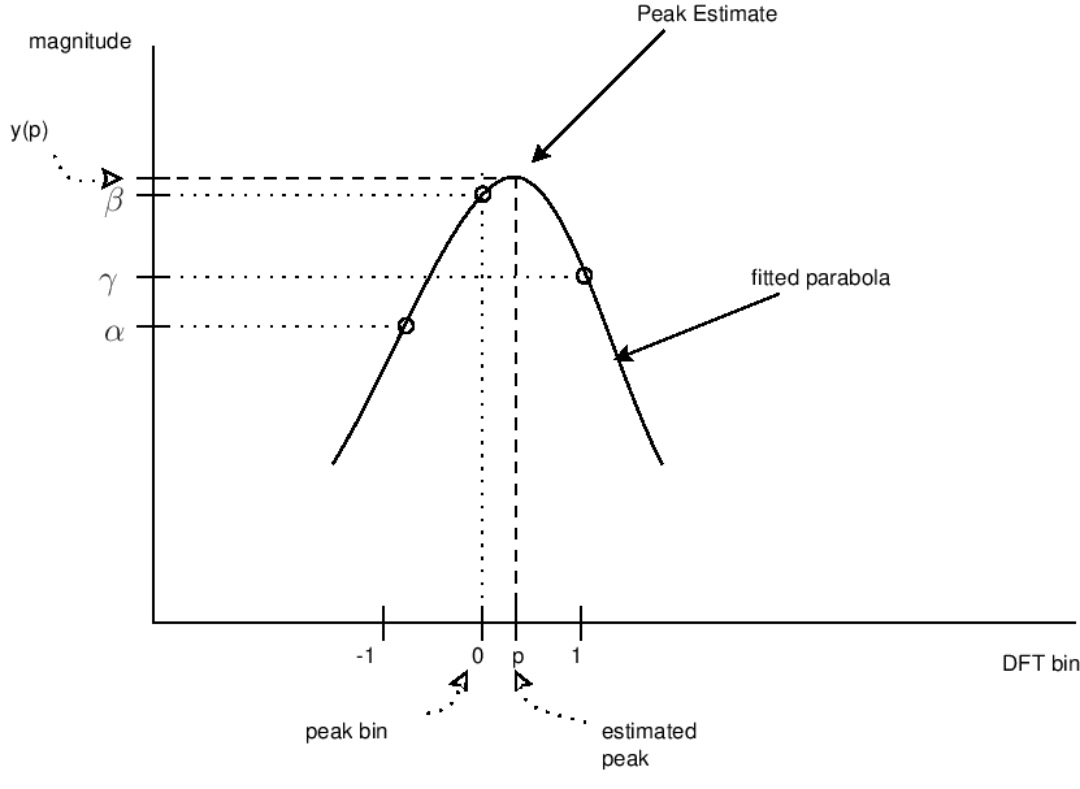
做一个梯度方向的直方图，范围是0～360度，其中每10度一个柱，总共36个柱。每个采样点按照其梯度方向θ(x,y)加权统计到直方图，权值为幅度m(x,y )和贡献因子的乘积。贡献因子是采样点到关键点（窗口中心）距离的量度，距离越大，贡献因子越小



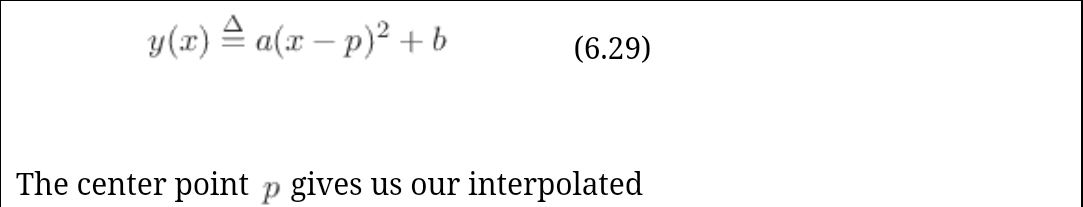
直方图的峰值代表了该关键点处邻域梯度的主方向，如图3-2，左图是关键点邻域窗口，右图是7个柱的梯度直方图（实际是36个柱）。在直方图统计时，每相邻三个像素点采用高斯加权，根据Lowe的建议，模板采用[0.25,0.5,0.25],并且连续加权两次.

Lowe指出，直方图的峰值确定以后，任何大于峰值80%的方向（柱）创建一个具有该方向的关键点，因此，对于多峰值（幅值大小接近）的情形，在同一位置和尺度就会产生多个具有不同方向的关键点。虽然这样的点只占15%，但是它们却能显著地提高匹配的稳定性。用每个峰值和左右两个幅值拟合二次曲线，以定位峰值的实际位置（抛物线的最高点）。峰值方向的精度高于10度。

### 3.2 拟合二次曲线过程

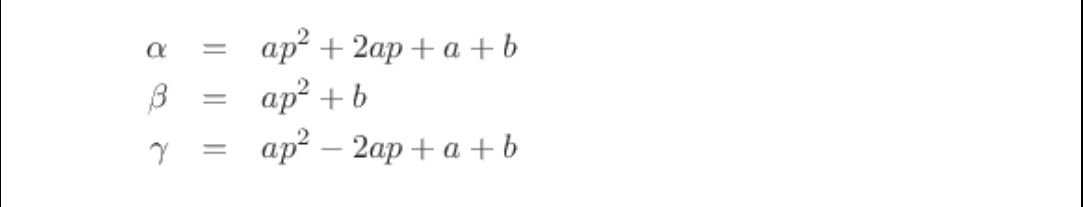


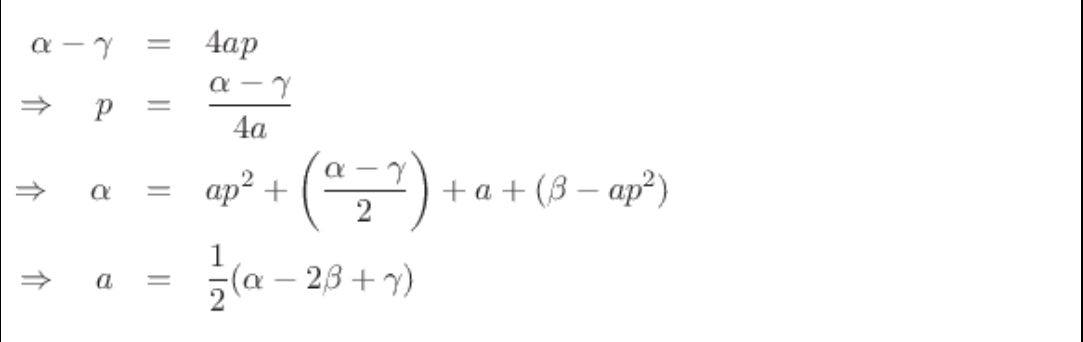
假设二次曲线：



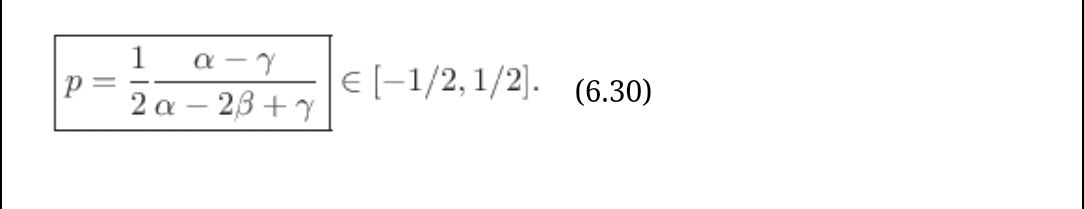
p是想得到的点坐标，则：

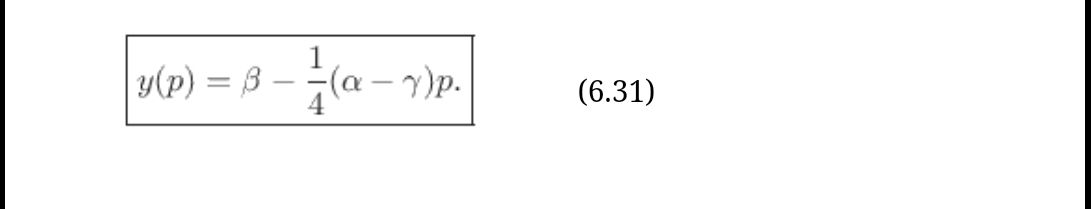






所以





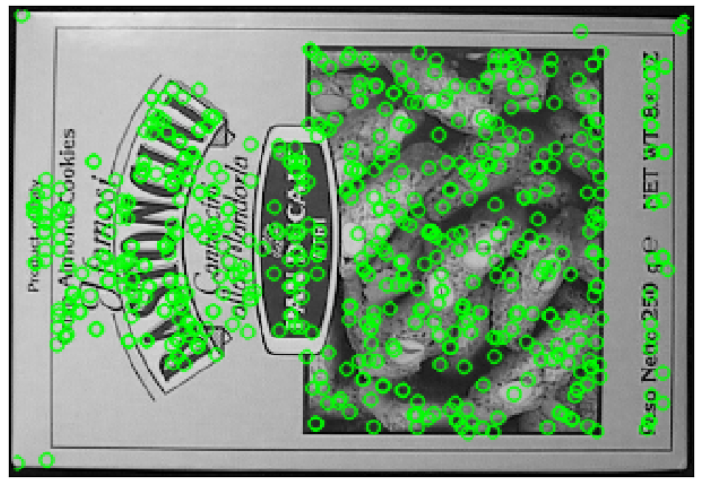
**关键点的方向分配编程思路：**

在上一步，得到了各个关键点的坐标和尺度。

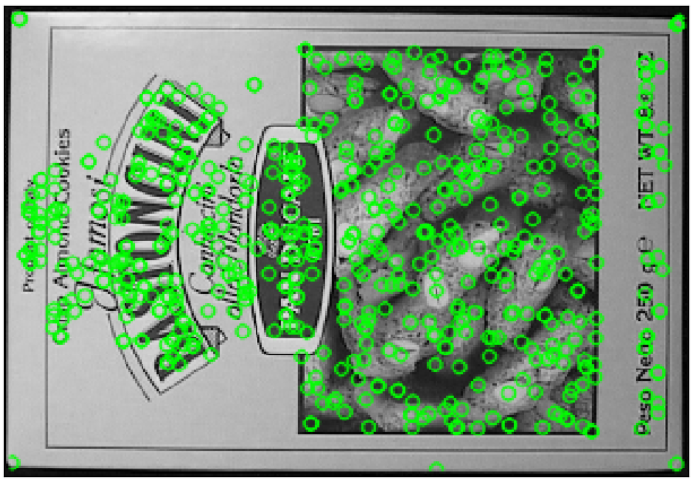
1. 计算出3原则，求取窗口半径Radius
2. 对于方形窗口内的每一点（i，j）如果到中心的距离小于Radius则计算梯度方向（i，j从-Radius到Radius）
3. 对得到的直方图进行两次平滑滤波
4. 找到平滑直方图峰值（可以使用np.where和np.roll，np.roll分别左滑和右滑，np.where进行判断）
5. 拟合二次曲线，得到拟合点坐标，在乘以10就是这个关键点的度数
6. 重复上述过程

### 3.3 关键点结果展示

至此就得到了图片的所有关键点：



实验结果图



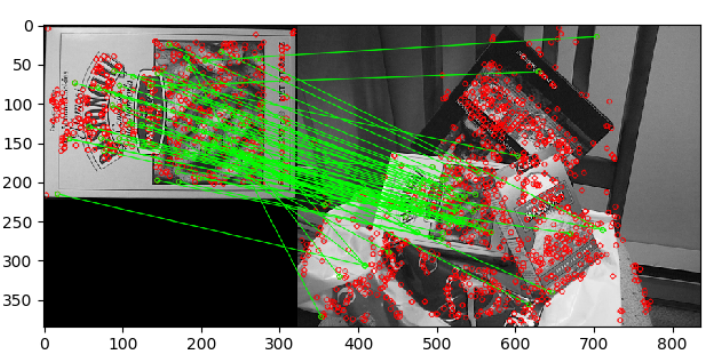
OpenCV结果图

可以看到两张图效果差不多。

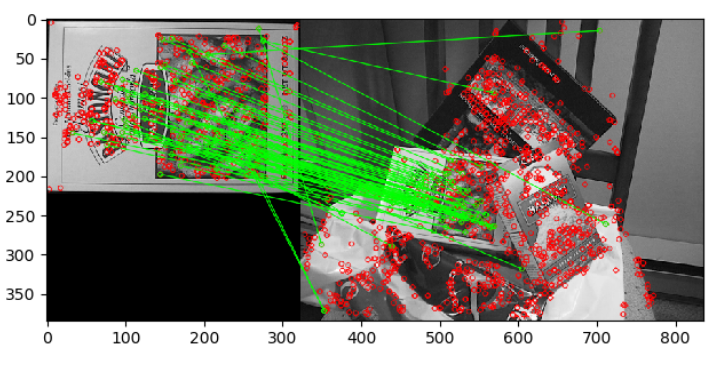
### 3.4 消除重复的关键点

此时，已经得到全部关键点了，但是有的关键点是重复的，所以我们要删掉重复的关键点，否则影响后续的匹配效果。我的实验证明，删掉关键点匹配出的效果更好。

如果不去掉重复的关键点：



去掉重复的关键点：



可以看到虽然依旧不是太好，但是比上一张图有明显的改善（第一张大概15对不准确，第二张减少了9对）。

**编程思路：**

上面得到了全部想要的极值点信息并以KeyPoint存储

1. 从前到后遍历全部KeyPoints，每个KeyPoint与他后的作比较，如果参数全都一样，则删除前面的KeyPoint。

## 4 关键点特征描述

通过以上步骤，对于每一个关键点，拥有三个信息：位置、尺度以及方向。接下来就是为每个关键点建立一个描述符，用一组向量将这个关键点描述出来，使其不随各种变化而改变，比如光照变化、视角变化等等。这个描述子不但包括关键点，也包含关键点周围对其有贡献的像素点，并且描述符应该有较高的独特性，以便于提高特征点正确匹配的概率。

SIFI 描述子h(x, y, θ)是对特征点附近邻域内高斯图像梯度统计结果的一种表示，它是一个三维的阵列，但通常将它表示成一个矢量。矢量是通过对三维阵列按一定规律进行排列得到的。特征描述子与特征点所在的尺度有关，因此，**对梯度的求取应在特征点对应的高斯图像上进行**。通过对关键点周围图像区域分块，计算块内梯度直方图，生成具有独特性的向量，这个向量是该区域图像信息的一种抽象，具有唯一性。

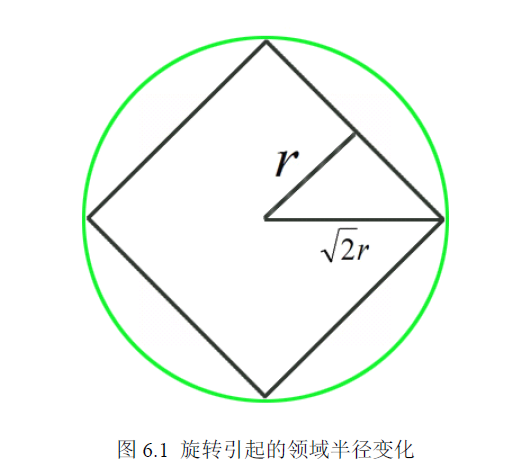
Lowe 建议描述子使用在关键点尺度空间内4×4的窗口中计算的8 个方向的梯度信息，共4×4×8 =128维向量表征。表示步骤如下：

### 4.1确定计算描述子所需的图像区域

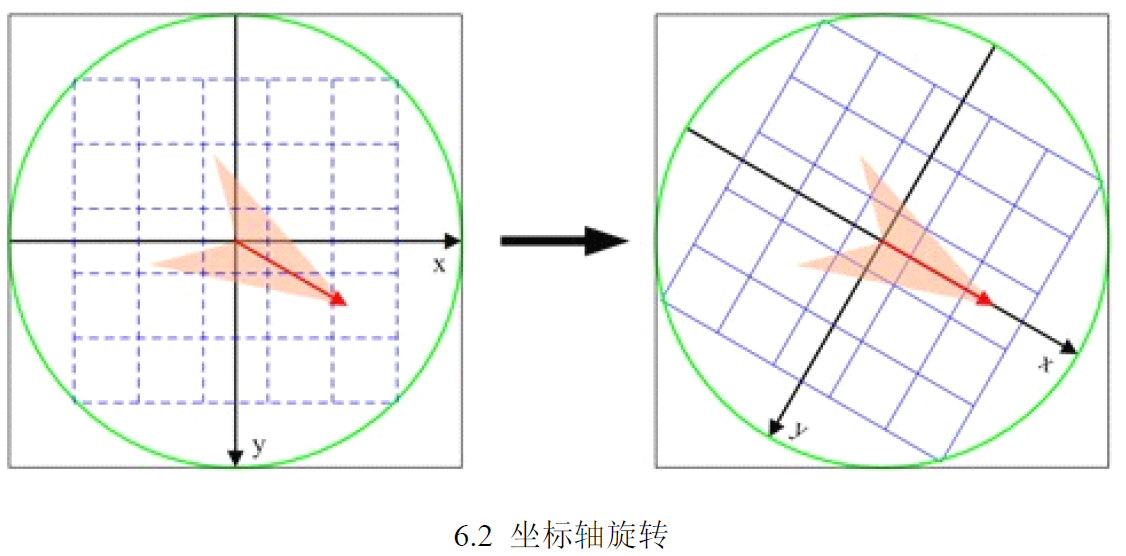
特征描述子与特征点所在的尺度有关，因此，对梯度的求取应在特征点对应的高斯图像上进行。将关键点附近的领域划分为d ×d (Lowe 建议d=4)个子区域，每个子区域做为一个种子点，每个种子点有8 个方向。每个子区域的大小与关键点方向分配时相同，即每个区域有个子像素，为每个子区域分配边长为的矩形区域进行采样(个子像素实际用边长为的矩形区域即可包含， 不大，为了简化计算取其边长为，并且采样点宜多不宜少)。考虑到实际计算时，需要采用双线性插值，所需图像窗口边长为。在考虑到旋转因素(方便下一步将坐标轴旋转到关键点的方向)，如下图6.1 所示，实际计算所需的图像区域半径为：



计算结果四舍五入取整。



### 4.2 将坐标轴旋转为关键点的方向（旋转不变性）



旋转后领域内采样点的新坐标为：



**注：**网上的博客很多都是这张图，但是要额外注意的是，如4.4节左图所示，蓝色框是为了解释为了插值而进行的扩增，蓝色框内的中心，是上一节当中dXd（Lowe建议4）子区域的中心，整好16个点。同时也说明每个子区域受4个蓝色区域影响。

**上两小节的编程思路：**遍历关键点的（x，y，angle），带入上面的公式

### 4.3 梯度直方图的生成

将领域内的采样点分配到对应的子区域内，将子区域内的梯度值分配到8 个方向上，计算其权值。旋转后的采样点坐标在半径为radius的圆内被分配到d×d 的子区域，计算影响子区域的采样点的梯度和方向，分配到8个方向上。坐标归一化后，此后在特征点局部坐标系中把作为一个单元长度。正常的结果范围为加是把坐标系由特征点平移到左上角，与图片坐标系一致，结果的范围就变成了，方便后面三线性插值计算。在减0.5则是回移坐标系至描述子区间中的第一个子区域的中心处，平衡之前计算radius时多加的0.5.。

旋转后的采样点(x’,y’)落在子区域的下标为



Lowe 建议子区域的像素的梯度大小按的高斯加权计算，即

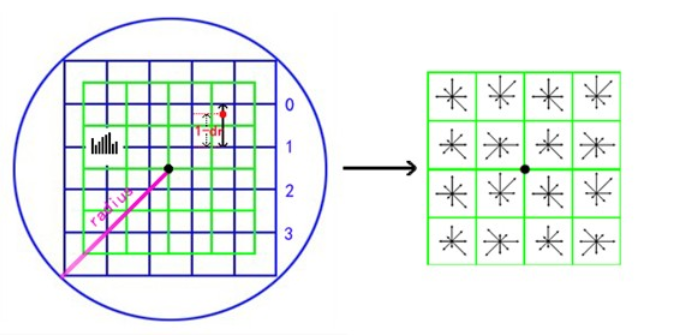


其中a，b 为关键点在高斯金字塔图像中的位置坐标。

**前3小节编程思路：**遍历关键点，带入上面的各个公式。此刻计算得到了描述子计算所需的图像区域的每一个关键点描述子区域的信息（4X4方格内的位置(x’’,y’’)，旋转后的相对坐标(x’,y’)，方向的直方图坐标，加权后的幅值）

### 4.4 三线性插值

考虑二维情况，如下图左，若(x'',y")距离X轴和Y轴的距离分别为（dr，dc），则另外的几个点分别为 （dr，1-dc）,（1-dr，dc）,（1-dr，1-dc）；根据权重与距离成反比的原则计算邻近的四个种子点分配的到权重：



则分配到b1的值为：W \* (1-dr)\*(1-dc)

            b2的值为： W \* (1-dr)\*dc

            b3的值为： W \* dr\*dc

            b4的值为： W \* dr\*(1-dc)

在看上图右，添加了角度分量，所以由二维变成了三维，双线性插值就变成了三维线性插值。如下方图所示。角度转化为比例分量，为do，那么距离其邻近的另外一个方向的距离为(1-do)。

则点(x”,y") 的W分配的权重如下：

分配给b1在 方向1上的权重为： W \* (1-dr)\*(1-dc) \*(1-do)

方向2上的权重为：W \* (1-dr)\*(1-dc) \*do

分配给b2在 方向1上的权重为： W \* (1-dr)\*dc \*(1-do)

方向2上的权重为：W \* (1-dr)\*dc \*do

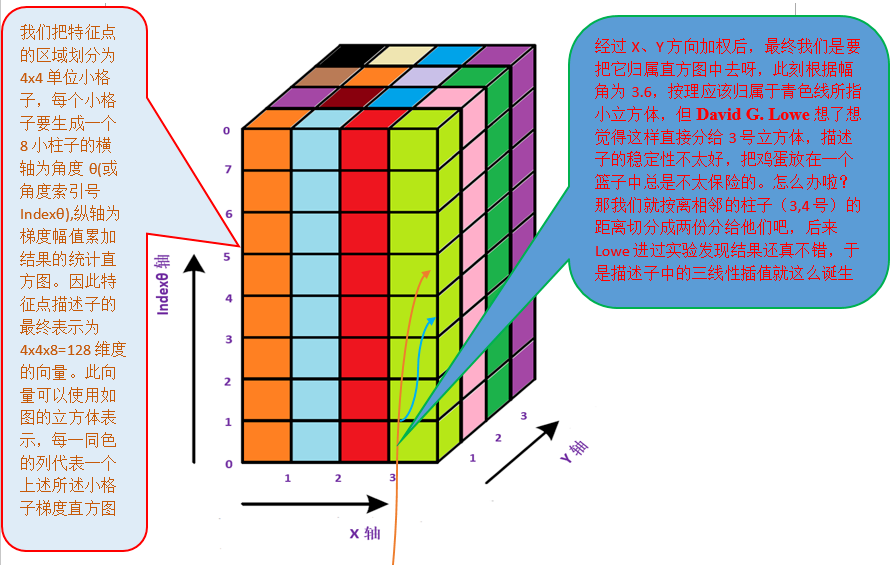
分配给b3在 方向1上的权重为： W \* dr\*dc \*(1-do)

方向2上的权重为：W \* dr\*dc \*do

分配给b4在 方向1上的权重为： W \* dr\*(1-dc) \*(1-do)

方向2上的权重为：W \* dr\*(1-dc) \*do1

最终形成的特征形式应该如下图所示。



最后得到的是4X4X8特征，一般表示为一个矢量，这个矢量是通过对三维特征按一定规律进行排列得到的。

**编程思路：**

1. 由于每个点都会得到一个2X2X2的数组用于分配到各个梯度直方图的Bin中，所以一些边缘的点得到的8个数实际只有4个在特征向量内，所以创建的特征数组可以为6X6X8，在计算完成后，在截取其中的4X4X8。
2. 每一个关键点描述子区域的信息，带入上面的公式得到8个值，放入上图中对应的梯度直方图的Bin中做累加。
3. 最后进行扁平化（numpy.array..flatten()）

**注：**第一步我本来创建的特征数组为6X6X10，但是通过对比OpenCV发现，第三维度是角度，大于360应该用角度-360，小于0则360+角度。

### 4.5 归一化特征向量

为了去除光照变化的影响，需对上述生成的特征向量进行归一化处理，对于图像灰度值整体漂移，图像各点的梯度是邻域像素相减得到，所以能去除光照的影响。在归一化处理后，对特征矢量大于0.2的要进行截断处理，即大于0.2的值只取0.2，然后重新进行一次归一化处理，其目的是为了提高鉴别性。

得到的描述子向量为，归一化后的特征向量为则：



**注：**最后得到的特征向量要乘以512，取整，并在0到255之间饱和，以从float32转换为无符号字符（OpenCV约定）

## 5 SIFT特点总结

**优点：**

1. 能提取物体上的一些局部外观的兴趣点，而且不会考虑到边缘响应。

原因：D 的主曲率和H 的特征值成正比，令为最大特征值，为最小的特征值，则公式的值在两个特征值相等时最小，随着的增大而增大。值越大，说明两个特征值的比值越大，即在某一个方向的梯度值越大，而在另一个方向的梯度值越小，而边缘恰恰就是这种情况。通过让该比值小于一定的阈值，剔除边缘响应点。

1. 与影像的大小无关

原因：考虑到了各个尺度的信息，并且使用的是尺度变换中惟一的线性变换核，能够较好地（恰当的）保留信息。

1. 旋转不变性

原因：计算关键点描述时，统一将坐标轴旋转到同一方向（关键点方向）。

1. 光照影响小

原因：特征向量归一化后只保留大于阈值的关键点，得到了相对来讲对亮度不明显的关键点，同时整个过程对于图像灰度值整体漂移，图像各点的梯度是邻域像素相减得到，梯度只是看对比，所以能去除光照的影响。

1. 适用于在海量特征数据库中进行快速、准确的匹配

原因：SIFT特征找的是位置、尺度、旋转不变量，独特性强，便于匹配。

1. 对平移有很好的稳定性

原因：因为SIFT提取的是关键点，关键点只跟局部信息有关，所以平移对特性影响较少。

1. 对仿射变换有很好的稳定性

原因：通过高斯加权，使特征点附近的梯度幅值有较大的权重，这样可以部分弥补仿射不变性而产生的特征点不稳定的问题。

1. 对噪音有一定的稳定性。

原因：过小的点易受噪声的干扰而变得不稳定，所以SIFT中将小于某个经验值(Lowe 论文中使用0.03，RobHess 等人实现时使用0.04/S)的极值点删除。

1. 可扩展，可与不同的特征相互补充，合并使用。

**缺点：**

光照的变化很大程度上影响着彩色物体识别的效果。在物体描述与匹配中,颜色可以提供更加有用的信息，物体的颜色信息被忽略，致使一些物体会被错误地分类。在图像处理中彩色图像能够表达更多的信息，彩色信息可以获得更高的辨别率。SIFT描述子对图像的高斯梯度进行编码，该描述子在空间模式下描述了灰度图像16个种子点及每个种子点8个梯度方向。由于SIFT算法只是利用图像的灰度信息，不能很好地区分形状相似但颜色不同的物体。所有后续产生了许多SIFT彩色描述子。

## 6 SIFT参考资料

**参考博客（部分博客，有的忘记收藏了。。）：**

<https://github.com/rmislam/PythonSIFT/tree/8cc46eba1ed3530089d6de459252f043cec6e694>

<https://medium.com/@russmislam/implementing-sift-in-python-a-complete-guide-part-1-306a99b50aa5>

<https://medium.com/@russmislam/implementing-sift-in-python-a-complete-guide-part-2-c4350274be2b>

（因为David G. Lowe申请了算法的版权，所以网上很少有OpenCV的sift.cpp文件源码解读，有也是很少的几行。上面这个github作者自己说自己代码各个步骤输出跟OpenCV一致。下方两个链接是作者写的sift教程。当我将自己的代码与仓库代码部分细节一致后，效果得到了显著提升。具体对照细节已在上文的各小节中说明。）

<https://blog.csdn.net/lingyunxianhe/article/details/79063547>

<https://www.cnblogs.com/JiePro/p/sift_4.html>

<https://blog.csdn.net/fzthao/article/details/62424271>

<https://blog.csdn.net/civiliziation/article/details/38364651>

<https://blog.csdn.net/u010440456/article/details/81483145>

<https://blog.csdn.net/civiliziation/article/details/38364651>

**参考论文：**

[Distinctive Image Features from Scale-Invariant Keypoints, David G. Lowe](https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf)

SIFT原理与编程总结，北京理工大学，郝志会

# 三 图片拼接

## 1 关键点匹配

### 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，但通过后人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。

### 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。

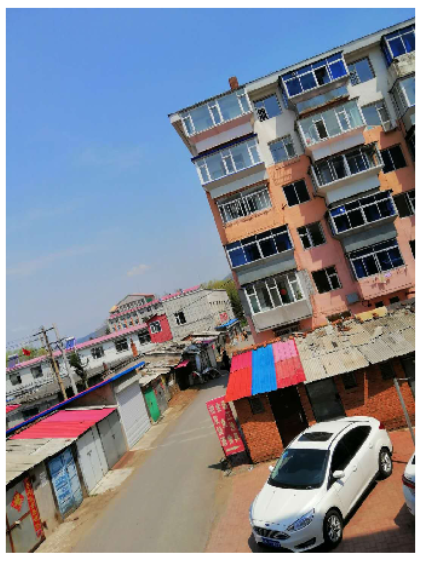
（3）将所有的数据点套到这个模型中计算误差。

（4）找到所有满足误差阈值的点。

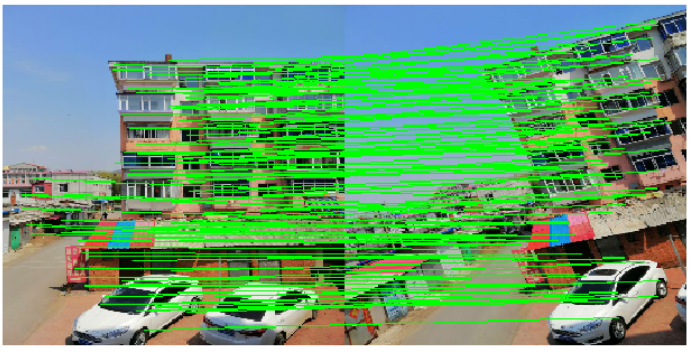
（5）然后我们再重复（1）~（4）这个过程，直到达到一定迭代次数后，选出那个被支持的最多的模型，作为问题的解。

### 1.3 匹配结果

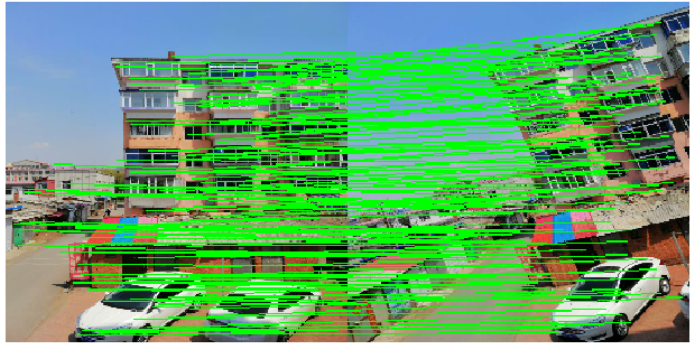
手机拍摄的原始图片：



匹配结果（考虑计算时间，统一把图片缩放成（256,256））：



实验结果



OpenCV结果

## 2 图像配准

图像配准是一种确定待拼接图像间的重叠区域以及重叠位置的技术，它是整个图像拼接的核心。本节采用的是基于特征点的图像配准方法，即通过匹配点对构建图像序列之间的变换矩阵，从而完成全景图像的拼接。

变换矩阵H求解是图像配准的核心，其求解的**算法流程如下：**

（1）检测每幅图像中特征点。

（2）计算特征点之间的匹配。

（3）计算图像间变换矩阵的初始值。

（4）迭代精炼H变换矩阵。

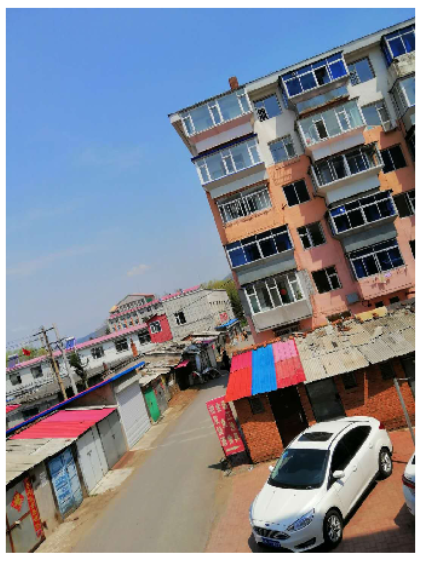
（5）引导匹配。用估计的H去定义对极线附近的搜索区域，进一步确定特征点的对应。

（6）重复迭代4）和5）直到对应点的数目稳定为止。

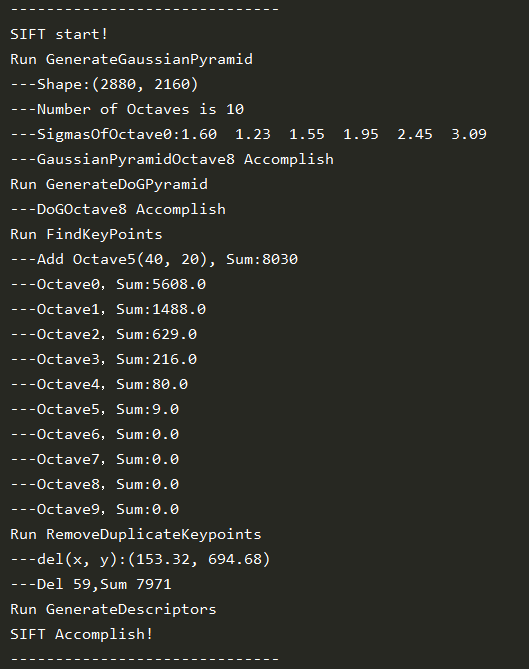
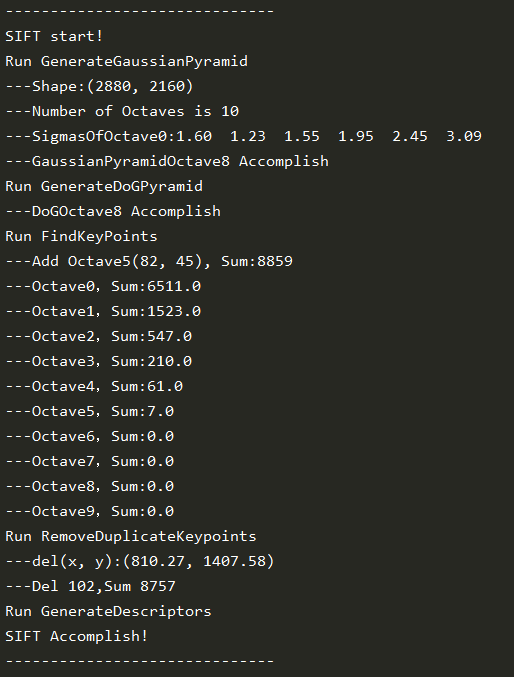
设图像序列之间的变换为投影变换，可用4组最佳匹配计算出H矩阵的8 个自由度参数hi=( i=0,1,...,7)，并以此作为初始值。

## 3 配准结果

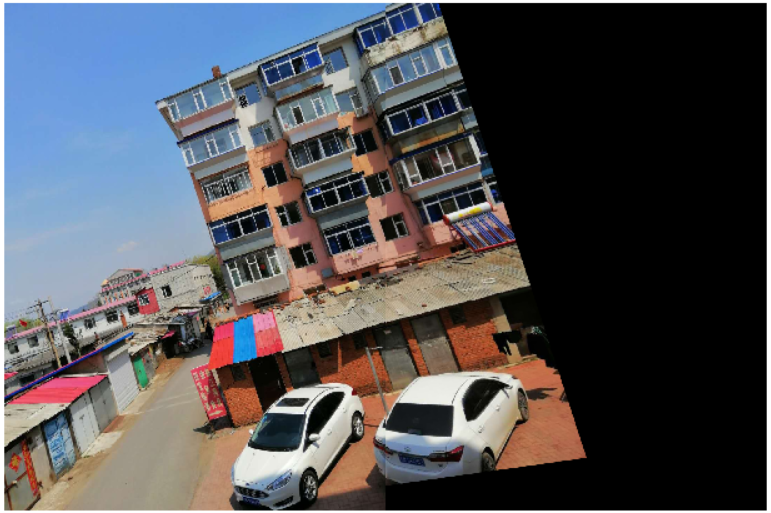
手机拍摄的原始图片：



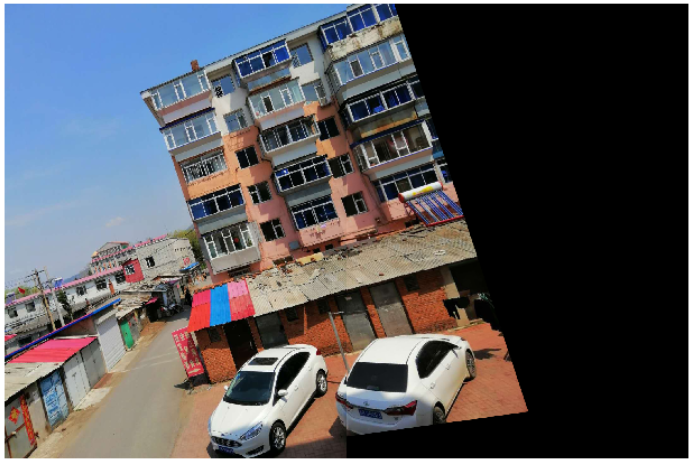
运行结果：

配准结果：



实验结果



OpenCV SIFT结果

可以看到结果大致一样，说明SIFT特征大致一样。至于右上角的明显差异，是配准方式的问题。

# 四 代码

文件结构

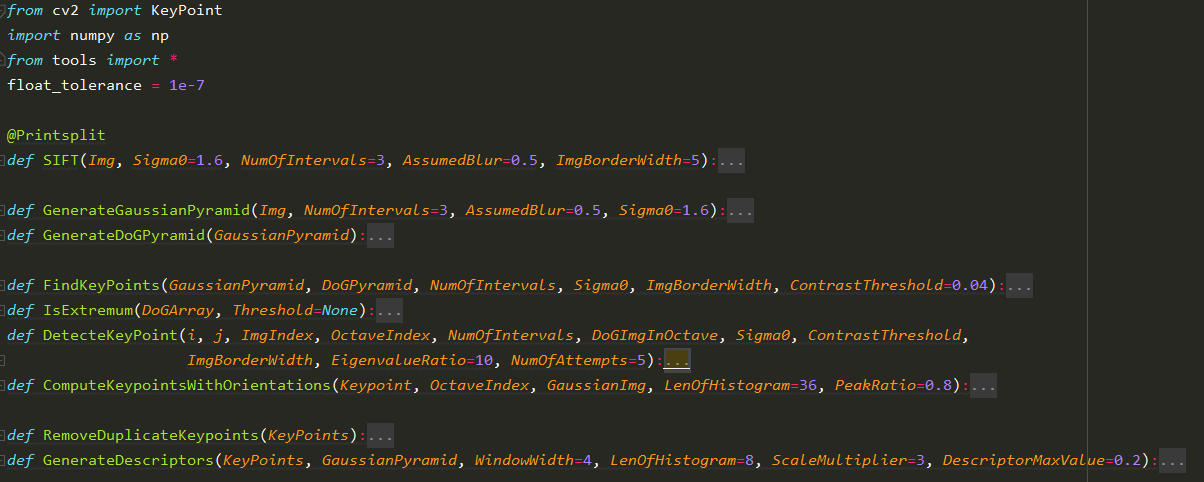


代码结构：

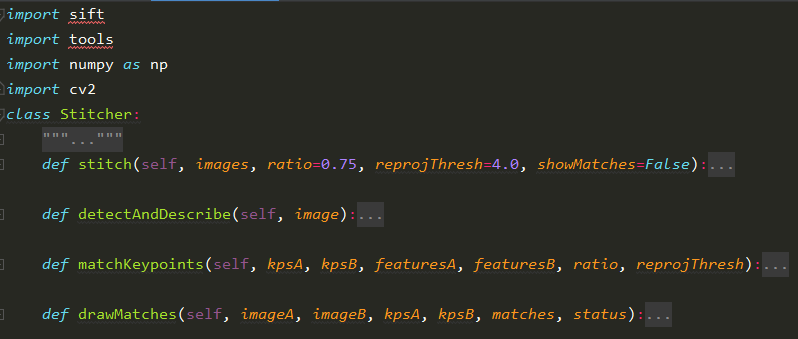
tools.py



sift.py



Splice.py结构



tools.py代码：

*import* numpy *as* np  
*def* Timing(*f*):  
 """  
 函数装饰器，计时器，输出函数运行时间  
 @param f: 被计时函数  
 @return: 被计时函数运行结果  
 """  
 *def* rf(*\*args*, *\*\*kwargs*):  
 *import* time  
 start\_time = time.time()  
 result = f(\**args*, \*\**kwargs*)  
 end\_time = time.time()  
 print('用时：{}ms'.format((end\_time - start\_time) \* 1000))  
 *return* result  
 *return* rf  
*def* ShowPict(*pict*, *Gray*=*False*):  
 """  
 函数装饰器，显示图片，可选择是否按照灰度图显示  
 @param pict: 要显示的图片  
 @param Gray: 是否按照灰度图显示  
 """  
 *import* matplotlib.pyplot *as* plt  
 *if Gray*:  
 plt.imshow(*pict*.astype(np.float), cmap='gray')  
 *else*:  
 plt.imshow(*pict*.astype(np.int))  
 plt.axis('off')  
 plt.show()  
*def* Printsplit(*f*):  
 """  
 函数装饰器，对函数输出进行分隔  
 ---------------------------------  
 （中间是函数自身的输出结果）  
 ---------------------------------  
 @param f: 被装饰的函数  
 @return: 被装饰的函数的运行结果  
 """  
 *def* rf(*\*args*, *\*\*kwargs*):  
 print('-' \* 30)  
 result = f(\**args*, \*\**kwargs*)  
 print('-'\*30)  
 *return* result  
 *return* rf  
  
*def* Resize(*OriginalImage*, *TargetSize*):  
 """  
 单通道图片进行缩放  
 @param OriginalImage: 原始图片  
 @param TargetSize: 目标图片的大小，格式为（宽，高）  
 @return: 缩放后的单通道图片  
 """  
 OriginalImageWidth, OriginalImageHeight = len(*OriginalImage*[0]), len(*OriginalImage*)  
 TargetWidth, TargetHeight = *TargetSize*[0], *TargetSize*[1]  
 WidthScale = OriginalImageWidth / TargetWidth  
 HeightScale = OriginalImageHeight / TargetHeight  
 TargetImage = np.zeros((TargetHeight, TargetWidth), dtype=np.uint8)  
 *for* i *in* range(TargetHeight):  
 *for* j *in* range(TargetWidth):  
 x, y = (i + 0.5) \* HeightScale - 0.5, (j + 0.5) \* WidthScale - 0.5  
 x1, y1 = int(x), int(y)  
 x2, y2 = min(x1 + 1, OriginalImageHeight - 1), min(y1 + 1, OriginalImageWidth - 1)  
 E1 = (x - x1) \* (int(*OriginalImage*[x2][y1]) - int(*OriginalImage*[x1][y1])) + int(*OriginalImage*[x1][y1])  
 E2 = (x - x1) \* (int(*OriginalImage*[x2][y2]) - int(*OriginalImage*[x1][y2])) + int(*OriginalImage*[x1][y2])  
 f = (y - y1) \* (E2 - E1) + E1  
 TargetImage[i][j] = f  
 *return* TargetImage  
*def* GenerateGaussianKernel2D(*KernelSize*=3, *Sigma*=0):  
 """  
 生成2维高斯卷积核  
 @param KernelSize:高斯核边长  
 @param Sigma:Sigma  
 @return: 2维高斯卷积核  
 """  
 Kernel = np.zeros([*KernelSize*, *KernelSize*])  
 Center = *KernelSize* // 2  
 *if Sigma* == 0:  
 Sigma = ((*KernelSize* - 1) \* 0.5 - 1) \* 0.3 + 0.8  
 s = 2 \* (*Sigma* \*\* 2)  
 SumValue = 0  
 *for* i *in* range(0, *KernelSize*):  
 *for* j *in* range(0, *KernelSize*):  
 x = i - Center  
 y = j - Center  
 Kernel[i, j] = np.exp(-(x \*\* 2 + y \*\* 2) / s)  
 SumValue += Kernel[i, j]  
 SumValue = 1 / SumValue  
 *return* Kernel \* SumValue  
*def* GenerateGaussianKernel1D(*KernelSize*=3, *Sigma*=0):  
 """  
 生成两个分离高斯卷积核  
 @param KernelSize:高斯核边长  
 @param Sigma:Sigma  
 @return: 两个分离高斯卷积核，大小为1XN和NX1  
 """  
 Kernel = np.zeros([1,*KernelSize*])  
 Center = *KernelSize* // 2  
 *if Sigma* == 0:  
 Sigma = ((*KernelSize* - 1) \* 0.5 - 1) \* 0.3 + 0.8  
 s = 2 \* (*Sigma* \*\* 2)  
 SumValue = 0  
 *for* i *in* range(0, *KernelSize*):  
 x = i - Center  
 Kernel[0, i] = np.exp(-(x \*\* 2) / s)  
 SumValue += Kernel[0, i]  
 SumValue = 1 / SumValue  
 Kernal1XN = Kernel \* SumValue  
 KernalNX1 = Kernal1XN.T  
 *return* Kernal1XN, KernalNX1  
  
*def* ComputeGradientAtCenterPixel(*DoGArray*):  
 """  
 求DoG矩阵（3X3X3）中心点的梯度  
 @param DoGArray: DoG矩阵（3X3X3）  
 @return: 中心点的梯度[dx, dy, ds]  
 """  
 dx = 0.5 \* (*DoGArray*[1, 1, 2] - *DoGArray*[1, 1, 0])  
 dy = 0.5 \* (*DoGArray*[1, 2, 1] - *DoGArray*[1, 0, 1])  
 ds = 0.5 \* (*DoGArray*[2, 1, 1] - *DoGArray*[0, 1, 1])  
 *return* np.array([dx, dy, ds])  
*def* ComputeHessianAtCenterPixel(*DoGArray*):  
 """  
 求DoG矩阵（3X3X3）中心点的hessian阵  
 @param DoGArray: DoG矩阵（3X3X3）  
 @return: 中心点的hessian阵 [[dxx, dxy, dxs],  
 [dxy, dyy, dys],  
 [dxs, dys, dss]]  
 """  
 CenterPixelValue = *DoGArray*[1, 1, 1]  
 dxx = *DoGArray*[1, 1, 2] - 2 \* CenterPixelValue + *DoGArray*[1, 1, 0]  
 dyy = *DoGArray*[1, 2, 1] - 2 \* CenterPixelValue + *DoGArray*[1, 0, 1]  
 dss = *DoGArray*[2, 1, 1] - 2 \* CenterPixelValue + *DoGArray*[0, 1, 1]  
 dxy = 0.25 \* (*DoGArray*[1, 2, 2] - *DoGArray*[1, 2, 0] - *DoGArray*[1, 0, 2] + *DoGArray*[1, 0, 0])  
 dxs = 0.25 \* (*DoGArray*[2, 1, 2] - *DoGArray*[2, 1, 0] - *DoGArray*[0, 1, 2] + *DoGArray*[0, 1, 0])  
 dys = 0.25 \* (*DoGArray*[2, 2, 1] - *DoGArray*[2, 0, 1] - *DoGArray*[0, 2, 1] + *DoGArray*[0, 0, 1])  
 *return* np.array([[dxx, dxy, dxs],  
 [dxy, dyy, dys],  
 [dxs, dys, dss]])  
  
*def* ComputeDet3(*Arr*):  
 """  
 求3X3矩阵的行列式  
 @param Arr: 3X3矩阵  
 @return: 3X3矩阵的行列式  
 """  
 *return Arr*[0,0]\**Arr*[1,1]\**Arr*[2,2] + *Arr*[1,0]\**Arr*[2,1]\**Arr*[0,2] + *Arr*[2,0]\**Arr*[0,1]\**Arr*[1,2] - *Arr*[0,0]\**Arr*[2,1]\**Arr*[1,2] - *Arr*[2,0]\**Arr*[1,1]\**Arr*[0,2] - *Arr*[1,0]\**Arr*[0,1]\**Arr*[2,2]  
*def* ComputeInv3(*Arr*):  
 """  
 求3X3矩阵的逆矩阵  
 @param Arr: 3X3矩阵  
 @return: 3X3矩阵的逆矩阵  
 """  
 det = ComputeDet3(*Arr*)  
 *return* np.array([[*Arr*[1,1]\**Arr*[2,2]-*Arr*[1,2]\**Arr*[2,1], *Arr*[0,2]\**Arr*[2,1]-*Arr*[0,1]\**Arr*[2,2], *Arr*[0,1]\**Arr*[1,2]-*Arr*[0,2]\**Arr*[1,1]],  
 [*Arr*[1,2]\**Arr*[2,0]-*Arr*[1,0]\**Arr*[2,2], *Arr*[0,0]\**Arr*[2,2]-*Arr*[0,2]\**Arr*[2,0], *Arr*[0,2]\**Arr*[1,0]-*Arr*[0,0]\**Arr*[1,2]],  
 [*Arr*[1,0]\**Arr*[2,1]-*Arr*[1,1]\**Arr*[2,0], *Arr*[0,1]\**Arr*[2,0]-*Arr*[0,0]\**Arr*[2,1], *Arr*[0,0]\**Arr*[1,1]-*Arr*[0,1]\**Arr*[1,0]]])/det  
  
*def* GenerateGrayImg(*Img*):  
 """  
 将3通道图片转换成灰度图  
 @param Img: 3通道或1通道图片  
 @return: 输入图片的灰度图  
 """  
 ImgShape = np.shape(*Img*)  
 *if* len(ImgShape) == 3:  
 r, g, b = [*Img*[:, :, i] *for* i *in* range(3)]  
 GrayImg = r \* 0.299 + g \* 0.587 + b \* 0.114  
 *if* len(ImgShape) == 2:  
 GrayImg = *Img  
 return* GrayImg  
*def* Convolve(*Img*, *Filter*, *Mode*='AdjustWeight'):  
 """  
 对1通道或3通道图片进行卷积  
 @param Img: 图片数组  
 @param Filter: 卷积核，边长必须为大于0的单数，必须是二维  
 @param Mode: 边缘处理方式（Symmetry,AdjustWeight）  
 @return: 输出卷积后的图片，大小不变  
 """  
 ImgShape = np.shape(*Img*)  
 Filter = np.array(*Filter*, dtype=np.float)  
 ConvImg = np.zeros\_like(*Img*, dtype=np.float)  
 *if* len(ImgShape) == 3:  
 *for* i *in* range(3):  
 ConvImg[:, :, i] = \_Convolve(*Img*[:, :, i], *Filter*, *Mode*)  
 *if* len(ImgShape) == 2:  
 ConvImg = \_Convolve(*Img*, *Filter*, *Mode*)  
 *return* ConvImg  
*def* \_Convolve(*Img*, *Filter*, *Mode*):  
 """  
 单通道图片卷积  
 @param Img: 图片数组  
 @param Filter: 卷积核  
 @param Mode: 边缘处理方式（Symmetry,AdjustWeight）  
 @return: 卷积后的图片  
 """  
 FilterHeight = *Filter*.shape[0]  
 FilterWidth = *Filter*.shape[1]  
 ConvImgHeight = *Img*.shape[0]  
 ConvImgWidth = *Img*.shape[1]  
 ConvImg = np.zeros((len(*Img*), len(*Img*[0])), dtype=np.float)  
 *if Mode* == 'Symmetry':  
 Height = *Filter*.shape[0] // 2  
 Width = *Filter*.shape[1] // 2  
 PadImg = np.zeros((len(*Img*) + 2 \* Height, len(*Img*[0]) + 2 \* Width), dtype=np.float)  
 PadImg[Height:Height+len(*Img*), Width:Width+len(*Img*[0])] = *Img* PadImg[0:Height, :] = PadImg[Height:2\*Height, :]  
 PadImg[len(*Img*) + Height:len(*Img*) + 2 \* Height, :] = PadImg[len(*Img*):len(*Img*)+Height, :]  
 PadImg[:, 0:Width] = PadImg[:, Width:2\*Width]  
 PadImg[:, len(*Img*[0]) + Width:len(*Img*[0]) + 2 \* Width] = PadImg[:, len(*Img*[0]):len(*Img*[0]) + Width]  
 Img = PadImg  
 *for* i *in* range(ConvImgHeight):  
 *for* j *in* range(ConvImgWidth):  
 ConvImg[i][j] = (*Img*[i:i + FilterHeight, j:j + FilterWidth] \* *Filter*).sum()  
  
 *if Mode* == 'AdjustWeight':  
 Height = *Filter*.shape[0] // 2  
 Width = *Filter*.shape[1] // 2  
 PadImg = np.zeros((len(*Img*) + 2 \* Height, len(*Img*[0]) + 2 \* Width), dtype=np.float)  
 PadImg[Height:Height + len(*Img*), Width:Width + len(*Img*[0])] = *Img* Img = PadImg  
 IsPointInTheSet0Img = *lambda x*, *y*:(x>=FilterHeight//2 *and* x<len(*Img*)-3\*(FilterHeight//2)) *and* (y>=FilterWidth//2 *and* y<len(*Img*[0])-3\*(FilterWidth//2))  
 IsPointOutsideTheSet0Img = *lambda x*, *y*:(x<FilterHeight//2 *or* x>=len(*Img*)-FilterHeight//2) *or* (y<FilterWidth//2 *or* y>=len(*Img*[0])-FilterWidth//2)  
 *for* i *in* range(ConvImgHeight):  
 *for* j *in* range(ConvImgWidth):  
 # print('\*'\*20)  
 # print(Img)  
 # print('卷积核坐上角坐标 i,j：',i,j,IsPointInTheSet0Img(i, j))  
 *if* IsPointInTheSet0Img(i, j):  
 ConvImg[i][j] = (*Img*[i:i + FilterHeight, j:j + FilterWidth] \* *Filter*).sum()  
 *else*:  
 y, x = i+FilterHeight//2, j+FilterWidth//2  
 # print('卷积中心坐标 x,y：',x,y)  
 AdjancetX = [x+ix *for* ix *in* range(-FilterHeight//2+1,FilterHeight//2+1)]\*FilterWidth  
 AdjancetY\_ = [y+iy *for* iy *in* range(-FilterWidth//2+1,FilterWidth//2+1)]  
 AdjancetY = []  
 *for* k *in* range(FilterWidth):  
 *for* z *in* range(FilterHeight):  
 AdjancetY.append(AdjancetY\_[k])  
 OutsidesPointLogical = [*not*(IsPointOutsideTheSet0Img(a, b)) *for* a,b *in* zip(AdjancetX, AdjancetY)]  
 FilterSum = 0  
 # print(AdjancetX)  
 # print(AdjancetY)  
 # print(OutsidesPointLogical)  
 *for* Logical,AX,AY *in* zip(OutsidesPointLogical,AdjancetX,AdjancetY):  
 # print('Logical,AX,AY:',Logical,AX-x+FilterHeight//2,AY-y+FilterWidth//2)  
 *if* Logical:  
 FilterSum += *Filter*[AX-x+FilterHeight//2][AY-y+FilterWidth//2]  
 # print('卷积核：', Filter)  
 # print('卷积区域:', Img[i:i + FilterHeight, j:j + FilterWidth])  
 ConvSum = (*Img*[i:i + FilterHeight, j:j + FilterWidth] \* *Filter*).sum()  
 ConvImg[i][j] = ConvSum/FilterSum *if* FilterSum!=0 *else* ConvSum  
 # print('结果：',ConvSum/FilterSum if FilterSum!=0 else ConvSum)  
 *return* ConvImg  
*def* DownSampling(*Img*):  
 """  
 对单通道图片进行步长为2的下采样（隔点采样）  
 @param Img: 图片  
 @return: 采样后的图片  
 """  
 Height, Width = len(*Img*), len(*Img*[0])  
 NewImg = []  
 *for* x *in* range(0,Height,2):  
 Row = []  
 *for* y *in* range(0,Width,2):  
 Row.append(*Img*[x][y])  
 NewImg.append(Row)  
 *return* np.array(NewImg)  
*def* RGB2HSI(*Img*):  
 """  
 rgb转hsi  
 @param Img: rgb图片  
 @return: hsi图片  
 """  
 rows = int(*Img*.shape[0])  
 cols = int(*Img*.shape[1])  
 r, g, b = [*Img*[:, :, i] *for* i *in* range(3)]  
 # 归一化到[0,1]  
 b = b / 255.0  
 g = g / 255.0  
 r = r / 255.0  
 RawHSI = *Img*.copy()  
 H, S, I = [*Img*[:, :, i] *for* i *in* range(3)]  
 *for* i *in* range(rows):  
 *for* j *in* range(cols):  
 num = 0.5 \* ((r[i, j]-g[i, j])+(r[i, j]-b[i, j]))  
 den = np.sqrt((r[i, j]-g[i, j])\*\*2+(r[i, j]-b[i, j])\*(g[i, j]-b[i, j]))  
 theta = np.float(np.arccos(num/den))  
  
 *if* den == 0:  
 H = 0  
 *elif* b[i, j] <= g[i, j]:  
 H = theta  
 *else*:  
 H = 2\*3.14169265 - theta  
  
 min\_RGB = min(min(b[i, j], g[i, j]), r[i, j])  
 sum = b[i, j]+g[i, j]+r[i, j]  
 *if* sum == 0:  
 S = 0  
 *else*:  
 S = 1 - 3\*min\_RGB/sum  
  
 H = H/(2\*3.14159265)  
 I = sum/3.0  
 # 输出HSI图像，扩充到255以方便显示，一般H分量在[0,2pi]之间，S和I在[0,1]之间  
 RawHSI[i, j, 0] = H\*255  
 RawHSI[i, j, 1] = S\*255  
 RawHSI[i, j, 2] = I\*255  
 *return* RawHSI  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *import* cv2  
 *import* imageio  
 Img = imageio.imread('11.jpg')  
 RawHSI = RGB2HSI(Img)  
  
 cv2.imshow('Img', Img)  
 cv2.imshow('RawHSI', RawHSI)  
  
 key = cv2.waitKey(0) & 0xFF  
 *if* key == ord('q'):  
 cv2.destroyAllWindows()

## sift.py代码

*from* cv2 *import* KeyPoint  
*import* numpy *as* np  
*from* tools *import* \*  
float\_tolerance = 1e-7  
  
@Printsplit  
*def* SIFT(*Img*, *Sigma0*=1.6, *NumOfIntervals*=3, *AssumedBlur*=0.5, *ImgBorderWidth*=5):  
 """  
 SIFT，流程和OpenCV一样  
 @param Img: 输入灰度图  
 @param Sigma0: SIFT里的Sigma0  
 @param NumOfIntervals: SIFT里的大S  
 @param AssumedBlur: 高斯模糊偏差，用于预先模糊2倍输入灰度图  
 @param ImgBorderWidth: 抛弃的图片边缘的长度，便于计算，网上说一般为5  
 @return: 关键点，特征  
 关键点用OpenCV里的KeyPoint存储  
 关键点的特征为128维  
 """  
 print('SIFT start!')  
 GaussianPyramid = GenerateGaussianPyramid(*Img*, *NumOfIntervals*, *AssumedBlur*, *Sigma0*)  
 DoGPyramid = GenerateDoGPyramid(GaussianPyramid)  
  
 KeyPoints = FindKeyPoints(GaussianPyramid, DoGPyramid, *NumOfIntervals*, *Sigma0*, *ImgBorderWidth*)  
 KeyPoints = RemoveDuplicateKeypoints(KeyPoints)  
  
 Descriptors = GenerateDescriptors(KeyPoints, GaussianPyramid)  
 print('SIFT Accomplish!')  
 *return* KeyPoints, Descriptors  
  
*def* GenerateGaussianPyramid(*Img*, *NumOfIntervals*=3, *AssumedBlur*=0.5, *Sigma0*=1.6):  
 """  
 生成灰度图的高斯金字塔  
 @param Img: 输入灰度图  
 @param NumOfIntervals: SIFT里的大S  
 @param AssumedBlur: 高斯模糊偏差，用于SIFT里预先模糊2倍输入灰度图  
 @param Sigma0: SIFT里的Sigma0（Octave0的SIgma0）  
 @return: 高斯金字塔，用列表存储  
 """  
 # 图片准备  
 print('Run GenerateGaussianPyramid')  
  
 *Img*.astype(np.float64)  
 ImgHeight, ImgWidth = len(*Img*), len(*Img*[0])  
  
 DoubleShapeImg = Resize(*Img*, (2 \* ImgWidth, 2 \* ImgHeight))  
 SigmaDiff = np.sqrt(max((*Sigma0* \*\* 2) - ((2 \* *AssumedBlur*) \*\* 2), 0.01))  
 GaussianKernelSize = int((6 \* SigmaDiff) // 2 \* 2 + 1)  
 Kernal\_1XN, Kernal\_NX1 = GenerateGaussianKernel1D(GaussianKernelSize, SigmaDiff)  
 # # 分离高斯卷积  
 BaseImg = Convolve(Convolve(DoubleShapeImg, Kernal\_1XN), Kernal\_NX1)  
 print('---Shape:({}, {})'.format(2 \* ImgHeight, 2 \* ImgWidth))  
 # 计算高斯金字塔组数  
 ImgShape = np.shape(BaseImg)  
 NumOfOctaves = np.int(np.round(np.log2(min(ImgShape)) - 1))  
 print('---Number of Octaves is', NumOfOctaves)  
 # 第0组Sigma准备  
 print('---SigmasOfOctave0:{:.2f} '.format(*Sigma0*), end='')  
 NumPerOctave = *NumOfIntervals* + 3  
 k = 2 \*\* (1. / *NumOfIntervals*)  
  
 GaussianKernelsSigmas = [*Sigma0*]  
  
 *for* ImgIndex *in* range(1, NumPerOctave):  
 SigmaPrevious = (k \*\* (ImgIndex - 1)) \* *Sigma0* SigmaTotal = k \* SigmaPrevious  
 Sigma = np.sqrt(SigmaTotal \*\* 2 - SigmaPrevious \*\* 2)  
 print('{:.2f} '.format(Sigma), end='')  
 GaussianKernelsSigmas.append(Sigma)  
 print(' ')  
 # 计算高斯金字塔  
 GaussianPyramid = []  
 Img = BaseImg  
  
 *for* OctaveIndex *in* range(NumOfOctaves):  
 GaussianImgInOctave = []  
 GaussianImgInOctave.append(*Img*)  
 *for* GaussianKernelSigma *in* GaussianKernelsSigmas[1:]:  
  
 GaussianKernelSize = int((6 \* GaussianKernelSigma) // 2 \* 2 + 1)  
 Kernal\_1XN, Kernal\_NX1 = GenerateGaussianKernel1D(GaussianKernelSize, GaussianKernelSigma)  
  
 Img = Convolve(Convolve(*Img*, Kernal\_1XN), Kernal\_NX1)  
 GaussianImgInOctave.append(*Img*)  
  
 GaussianPyramid.append(GaussianImgInOctave)  
 NextOctaveBaseImg = GaussianImgInOctave[-3]  
 Img = DownSampling(NextOctaveBaseImg)  
 print('\r---GaussianPyramidOctave{} Accomplish'.format(OctaveIndex - 1), end='', flush=*True*)  
 print('')  
 *return* np.array(GaussianPyramid)  
*def* GenerateDoGPyramid(*GaussianPyramid*):  
 """  
 差分高斯金字塔  
 @param GaussianPyramid: 高斯金字塔  
 @return: 差分高斯金字塔  
 """  
 print('Run GenerateDoGPyramid')  
 DoGPyramid = []  
 DoGOctaveIndex = 0  
  
 *for* GaussianImgInOctave *in GaussianPyramid*:  
 DoGImgInOctave = []  
 *for* FirstImg, SecondImg *in* zip(GaussianImgInOctave, GaussianImgInOctave[1:]):  
 DoGImgInOctave.append(np.array(SecondImg - FirstImg))  
  
 DoGPyramid.append(DoGImgInOctave)  
 print('\r---DoGOctave{} Accomplish'.format(DoGOctaveIndex - 1), end='', flush=*True*)  
 DoGOctaveIndex += 1  
 print('')  
 *return* np.array(DoGPyramid)  
  
*def* FindKeyPoints(*GaussianPyramid*, *DoGPyramid*, *NumOfIntervals*, *Sigma0*, *ImgBorderWidth*, *ContrastThreshold*=0.04):  
 """  
 找到关键点  
 @param GaussianPyramid: 灰度图的高斯金字塔，找关键点方向时要用  
 @param DoGPyramid: 灰度图的差分高斯金字塔，找关键点要用  
 @param NumOfIntervals: SIFT里的s  
 @param Sigma0: SIFT里的Sigma0  
 @param ImgBorderWidth: 抛弃的图片边缘的长度，便于计算，网上说一般为5  
 @param ContrastThreshold: SIFT里用于消除噪音点  
 @return: 关键点列表，里面的关键点用OpenCV的KeyPoint存储  
 """  
 print('Run FindKeyPoints')  
 Threshold = np.floor(0.5 \* *ContrastThreshold* / *NumOfIntervals* \* 255) # 与OpenCV一致  
 KeyPoints = []  
 SumOfKeyPoints = 0 # 关键点总数  
 OctaveStatistic = np.zeros(len(*DoGPyramid*)) # 各个Octave内的关键点数统计  
  
 *for* OctaveIndex, DoGImgInOctave *in* enumerate(*DoGPyramid*):  
 *for* ImgIndex *in* range(1, len(DoGImgInOctave)-1):  
 DoGImgs = [DoGImgInOctave[ImgIndex - 1], DoGImgInOctave[ImgIndex], DoGImgInOctave[ImgIndex + 1]] # 差分金字塔内的三张图片  
 ImgHeight, ImgWidth = np.shape(DoGImgInOctave[ImgIndex])  
 *for* i *in* range(*ImgBorderWidth*, ImgHeight - *ImgBorderWidth*):  
 *for* j *in* range(*ImgBorderWidth*, ImgWidth - *ImgBorderWidth*):  
 DoGArray = np.array([DoGImg[i - 1:i + 2, j - 1:j + 2] *for* DoGImg *in* DoGImgs]) # DoG数组（3X3X3）  
 *if* IsExtremum(DoGArray, Threshold):  
 Keypoint = DetecteKeyPoint(i, j, ImgIndex, OctaveIndex, *NumOfIntervals*, DoGImgInOctave, *Sigma0*, *ContrastThreshold*, *ImgBorderWidth*)  
 *if* Keypoint *is not None*:  
 Keypoint, LocalizedImgIndex = Keypoint # 关键点  
 KeyPointWithOrientations = ComputeKeypointsWithOrientations(Keypoint, OctaveIndex,*GaussianPyramid*[OctaveIndex][LocalizedImgIndex])  
 *for* KeyPointWithOrientation *in* KeyPointWithOrientations: # 带方向的关键点  
 SumOfKeyPoints += 1  
 OctaveStatistic[OctaveIndex] += 1  
 print('\r---Add Octave{}({}, {}), Sum:{}'.format(OctaveIndex, i, j, SumOfKeyPoints),  
 end='', flush=*True*)  
 KeyPoints.append(KeyPointWithOrientation)  
 print(' ')  
 *for* i *in* range(len(OctaveStatistic)):  
 print('---Octave{}，Sum:{}'.format(i, OctaveStatistic[i]))  
 *return* KeyPoints  
*def* IsExtremum(*DoGArray*, *Threshold*=*None*):  
 """  
 判断是否是极值点  
 @param DoGArray: DoG数组（3X3X3）  
 @param Threshold: 阈值，消除噪音点  
 @return: 返回True或False  
 """  
 *if Threshold is not None*:  
 *if* abs(*DoGArray*[1, 1, 1]) <= *Threshold*:  
 *return False* Max = *DoGArray*[1, 1, 1]  
 MaxIndex = [1, 1, 1]  
 Min = *DoGArray*[1, 1, 1]  
 MinIndex = [1, 1, 1]  
 *for* i *in* range(3):  
 *for* j *in* range(3):  
 *for* k *in* range(3):  
 *if DoGArray*[i, j, k] > Max:  
 Max = *DoGArray*[i, j, k]  
 MaxIndex = [i, j, k]  
 *elif DoGArray*[i, j, k] < Min:  
 Min = *DoGArray*[i, j, k]  
 MinIndex = [i, j, k]  
 *if* MaxIndex == [1, 1, 1] *or* MinIndex == [1, 1, 1]:  
 *return True  
 else*:  
 *return False  
def* DetecteKeyPoint(*i*, *j*, *ImgIndex*, *OctaveIndex*, *NumOfIntervals*, *DoGImgInOctave*, *Sigma0*, *ContrastThreshold*,  
 *ImgBorderWidth*, *EigenvalueRatio*=10, *NumOfAttempts*=5):  
 """  
 探测极值点是否是关键点  
 @param i: 极值点坐标i  
 @param j: 极值点坐标j  
 @param ImgIndex: 图片的层数s  
 @param OctaveIndex: 图片所在组数O  
 @param NumOfIntervals: SIFT里的大S  
 @param DoGImgInOctave: 差分高斯金字塔的一组图片，探测时可能移动层数  
 @param Sigma0: SIFT里的Sigma0  
 @param ContrastThreshold: SIFT里用于消除噪音点  
 @param ImgBorderWidth: 抛弃的图片边缘的长度，便于计算，网上说一般为5  
 @param EigenvalueRatio: 消除边缘响应的阈值里的r  
 @param NumOfAttempts: 关键点的探测次数  
 @return: 如果探测到关键点就返回KeyPoint，探测不到就返回None  
 """  
 ImgShape = (len(*DoGImgInOctave*[0]), len(*DoGImgInOctave*[0][0]))  
 *for* AttemptIndex *in* range(*NumOfAttempts*):  
 FirstImg, SecondImg, ThirdImg = *DoGImgInOctave*[*ImgIndex* - 1:*ImgIndex* + 2]  
 DoGArray = np.array([FirstImg[*i* - 1:*i* + 2, *j* - 1:*j* + 2],  
 SecondImg[*i* - 1:*i* + 2, *j* - 1:*j* + 2],  
 ThirdImg[*i* - 1:*i* + 2, *j* - 1:*j* + 2]]).astype(np.float) / 255 # Lowe要求归一化，为了跟阈值比较，消除噪音点  
  
 Gradient = ComputeGradientAtCenterPixel(DoGArray)  
 Hessian = ComputeHessianAtCenterPixel(DoGArray)  
 CoordinatesUpdateValueOfKeyPoint = -1 \* np.dot(ComputeInv3(Hessian), np.array([Gradient]).reshape((3, 1)))  
 CoordinatesUpdateValueOfKeyPoint.reshape(-1) # 偏移量  
  
 *if* abs(CoordinatesUpdateValueOfKeyPoint[0]) < 0.5 *and* abs(CoordinatesUpdateValueOfKeyPoint[1]) < 0.5 *and* abs(  
 CoordinatesUpdateValueOfKeyPoint[2]) < 0.5:  
 *break  
 j* += int(np.round(CoordinatesUpdateValueOfKeyPoint[0]))  
 *i* += int(np.round(CoordinatesUpdateValueOfKeyPoint[1]))  
 *ImgIndex* += int(np.round(CoordinatesUpdateValueOfKeyPoint[2]))  
 *if i* < *ImgBorderWidth or i* >= ImgShape[0] - *ImgBorderWidth or j* < *ImgBorderWidth or j* >= ImgShape[1] - *ImgBorderWidth or ImgIndex* < 1 *or ImgIndex* > *NumOfIntervals*:  
 *return None  
 if* AttemptIndex >= *NumOfAttempts* - 1:  
 *return None* NewKeyPointValue = DoGArray[1, 1, 1] + 0.5 \* np.dot(Gradient, CoordinatesUpdateValueOfKeyPoint)  
  
 *if* abs(NewKeyPointValue) \* *NumOfIntervals* >= *ContrastThreshold*:  
 xy\_Hessian = Hessian[:2, :2]  
 Tr = xy\_Hessian[0, 0] + xy\_Hessian[1, 1]  
 Det = xy\_Hessian[0, 0] \* xy\_Hessian[1, 1] - (xy\_Hessian[0, 1] \*\* 2)  
 *if* Det > 0 *and EigenvalueRatio* \* (Tr \*\* 2) < ((*EigenvalueRatio* + 1) \*\* 2) \* Det:  
 Keypoint = KeyPoint()  
 Keypoint.pt = ((*j* + CoordinatesUpdateValueOfKeyPoint[0]) \* (2 \*\* *OctaveIndex*),  
 (*i* + CoordinatesUpdateValueOfKeyPoint[1]) \* (2 \*\* *OctaveIndex*))  
 # Keypoint.pt 要以初始图片大小为准，这里的初始图片是扩大二倍后的图片  
 Keypoint.octave = *OctaveIndex* + *ImgIndex* \* (2 \*\* 8) + np.int(np.round((CoordinatesUpdateValueOfKeyPoint[2]) \* 255)) \* (2 \*\* 16)  
 # Keypoint.octave 0-7位保存Octave，8-15位保存ImgIndex， 16-23位保存sigma，防止在后面用Sigma返过来求s时，s直接取整会出错  
 Keypoint.size = *Sigma0* \* (2 \*\* ((*ImgIndex* + CoordinatesUpdateValueOfKeyPoint[2]) / np.float32(*NumOfIntervals*))) \* (2 \*\* (*OctaveIndex* + 1))  
 # Keypoint.size 这里的初始图片是扩大二倍后的图片，各个Sigma要多乘一个2, 所以OctaveIndex + 1  
 Keypoint.response = abs(NewKeyPointValue)  
 # 为什么不直接从初始图片开始算？SIFT关键点主要在-1组，具体在报告中介绍  
  
 *return* Keypoint, int(*ImgIndex* + CoordinatesUpdateValueOfKeyPoint[2])  
 *return None  
def* ComputeKeypointsWithOrientations(*Keypoint*, *OctaveIndex*, *GaussianImg*, *LenOfHistogram*=36, *PeakRatio*=0.8):  
 """  
 计算关键点方向  
 @param Keypoint: 关键点  
 @param OctaveIndex: 关键点所在组数，用于求Sigma\_oct  
 @param GaussianImg: 关键点所在高斯金字塔的图片  
 @param LenOfHistogram: 梯度直方图的长度这里是36，一个柱10度  
 @param PeakRatio: 大于主方向的PeakRatio倍，也要存储，Lowe推荐0.8  
 @return: 带方向的极值点KeyPoint  
 """  
 KeyPointWithOrientations = []  
 ImgShape = np.shape(*GaussianImg*)  
  
 Sigma\_oct = 1.5 \* *Keypoint*.size / np.float32(2 \*\* (*OctaveIndex* + 1))  
 Radius = int(np.round(3 \* Sigma\_oct))  
 WeightFactor = -0.5 / (Sigma\_oct \*\* 2)  
 Histogram = np.zeros(*LenOfHistogram*)  
 SmoothHistogram = np.zeros(*LenOfHistogram*)  
  
 *for* i *in* range(-Radius, Radius + 1):  
 Region\_y = int(np.round(*Keypoint*.pt[1] / np.float32(2 \*\* *OctaveIndex*))) + i  
 *if* Region\_y > 0 *and* Region\_y < ImgShape[0] - 1:  
 *for* j *in* range(-Radius, Radius + 1):  
 Region\_x = int(np.round(*Keypoint*.pt[0] / np.float32(2 \*\* *OctaveIndex*))) + j  
 *if* Region\_x > 0 *and* Region\_x < ImgShape[1] - 1:  
 *if* np.sqrt(i\*\*2 + j\*\*2) < Radius: # 圆形区域内  
 dx = *GaussianImg*[Region\_y, Region\_x + 1] - *GaussianImg*[Region\_y, Region\_x - 1]  
 dy = *GaussianImg*[Region\_y - 1, Region\_x] - *GaussianImg*[Region\_y + 1, Region\_x]  
 Magnitude = np.sqrt(dx \* dx + dy \* dy)  
 Orientation = np.rad2deg(np.arctan2(dy, dx)) # 1-360  
  
 Weight = np.exp(WeightFactor \* (i \*\* 2 + j \*\* 2))  
 HistogramIndex = int(np.round(Orientation \* *LenOfHistogram* / 360.)) % *LenOfHistogram* Histogram[HistogramIndex] = Histogram[HistogramIndex] + Weight \* Magnitude  
 # 在直方图统计时，每相邻三个像素点采用高斯加权，根据Lowe的建议，模板采用[0.25,0.5,0.25],并且连续加权两次.  
 *for* i *in* range(2):  
 *for* n *in* range(*LenOfHistogram*):  
 *if* n == 0:  
 SmoothHistogram[n] = 0.5 \* Histogram[n] + 0.25 \* (Histogram[*LenOfHistogram* - 1] + Histogram[n + 1])  
 *if* n == *LenOfHistogram* - 1:  
 SmoothHistogram[n] = 0.5 \* Histogram[n] + 0.25 \* (Histogram[n - 1] + Histogram[0])  
 *else*:  
 SmoothHistogram[n] = 0.5 \* Histogram[n] + 0.25 \* (Histogram[n - 1] + Histogram[n + 1])  
 Histogram = SmoothHistogram  
  
 OrientationMax = max(SmoothHistogram)  
 SmoothHistogramPeaks = np.where(  
 np.logical\_and(SmoothHistogram > np.roll(SmoothHistogram, 1), SmoothHistogram > np.roll(SmoothHistogram, -1)))[  
 0] # 找平滑后的直方图的极值点  
  
 *for* PeakIndex *in* SmoothHistogramPeaks:  
 PeakValue = SmoothHistogram[PeakIndex]  
 *if* PeakValue >= *PeakRatio* \* OrientationMax: # 统计主方向和大于0.8倍主方向的方向  
 LeftValue = SmoothHistogram[(PeakIndex - 1) % *LenOfHistogram*]  
 RightValue = SmoothHistogram[(PeakIndex + 1) % *LenOfHistogram*]  
  
 InterpolatedPeakIndex = 0.5 \* (LeftValue - RightValue) / (LeftValue - 2 \* PeakValue + RightValue)  
 Orientation = (PeakIndex + InterpolatedPeakIndex) \* (360 / *LenOfHistogram*)  
 Orientation = 360 - Orientation  
 NewkeyPoint = KeyPoint(\*tuple(0.5 \* np.array(*Keypoint*.pt)), 0.5 \* *Keypoint*.size, Orientation, *Keypoint*.response, *Keypoint*.octave)  
 # 这里的KeyPoint的极值点就全部检测完成了，为回到初始图片的大小所以乘以0.5  
 KeyPointWithOrientations.append(NewkeyPoint)  
 *return* KeyPointWithOrientations  
  
*def* RemoveDuplicateKeypoints(*KeyPoints*):  
 """  
 消除重复的极值点  
 @param KeyPoints: 关键点列表  
 @return: 去除重复关键点的列表  
 """  
 print('Run RemoveDuplicateKeypoints')  
 *if* len(*KeyPoints*) < 2:  
 *return KeyPoints* NeedDel = []  
 Unique = []  
 LenOfKeyPoints = len(*KeyPoints*)  
 *for* i *in* range(LenOfKeyPoints):  
 *for* j *in* range(i+1, LenOfKeyPoints):  
 *if KeyPoints*[i].pt[0] == *KeyPoints*[j].pt[0] *and KeyPoints*[i].pt[1] == *KeyPoints*[j].pt[1] *and KeyPoints*[i].angle == *KeyPoints*[j].angle:  
 NeedDel.append(i)  
 print('\r---del(x, y):({:.2f}, {:.2f})'.format(\**KeyPoints*[i].pt), end='', flush=*True*)  
 print(' ')  
 *for* i *in* range(LenOfKeyPoints):  
 *if* i *not in* NeedDel:  
 Unique.append(*KeyPoints*[i])  
 *if* len(Unique) == LenOfKeyPoints:  
 print('---No points to delete')  
 *else*:  
 print('---Del {},Sum {}'.format(LenOfKeyPoints - len(Unique), len(Unique)))  
 *return* Unique  
*def* GenerateDescriptors(*KeyPoints*, *GaussianPyramid*, *WindowWidth*=4, *LenOfHistogram*=8, *ScaleMultiplier*=3, *DescriptorMaxValue*=0.2):  
 """  
 计算关键点特征  
 @param KeyPoints: 关键点列表  
 @param GaussianPyramid: 高斯金字塔，计算特征在高斯金字塔上算  
 @param Sigma0: SIFT里的Sigma0  
 @param NumOfIntervals: SIFT里的大S  
 @param WindowWidth: 描述子所需区域的划分边长，Lowe推荐为4  
 @param LenOfHistogram: 梯度直方图的长度，这里为8，一个柱45度  
 @param ScaleMultiplier: 描述子所需区域的半径的系数  
 @param DescriptorMaxValue: 消除光照变化的影响的阈值  
 @return: 128维特征  
 """  
 print('Run GenerateDescriptors')  
 Descriptors = []  
  
 *for* Keypoint *in KeyPoints*:  
 Octave = Keypoint.octave & 255  
 *if* Octave >= 128:  
 Octave = Octave | -128  
 Layer = (Keypoint.octave >> 8) & 255  
  
 GaussianImg = *GaussianPyramid*[Octave + 1][Layer]  
 ImgHeight, ImgWidth = np.shape(GaussianImg)  
  
 Point = (np.round(np.array(Keypoint.pt) / (np.float32(2 \*\* Octave) *if* Octave >= 0 *else* np.float32(2 \*\* (-Octave))))).astype(np.int) # 从原始图片尺寸回到取样高斯图片尺寸上  
 Angle = 360. - Keypoint.angle  
 cos = np.cos(np.deg2rad(Angle))  
 sin = np.sin(np.deg2rad(Angle))  
 KeyPointInfomations = [] # 关键点信息，包括（4X4方格内的位置(x’’,y’’)，旋转后的相对坐标(x’,y’)，加权后的幅值，方向的直方图坐标）  
 HistogramTensor = np.zeros((*WindowWidth* + 2, *WindowWidth* + 2, *LenOfHistogram*))  
  
 ThreeSigma\_oct = *ScaleMultiplier* \* 0.5 \* Keypoint.size / (np.float32(2 \*\* Octave) *if* Octave >= 0 *else* np.float32(2 \*\* (-Octave)))  
 Radius = int(np.round(ThreeSigma\_oct \* np.sqrt(2) \* (*WindowWidth* + 1) \* 0.5))  
 *for* i *in* range(-Radius, Radius + 1):  
 *for* j *in* range(-Radius, Radius + 1):  
 yi = j \* sin + i \* cos  
 xi = j \* cos - i \* sin  
 yii = (yi / ThreeSigma\_oct) + 0.5 \* *WindowWidth* - 0.5  
 xii = (xi / ThreeSigma\_oct) + 0.5 \* *WindowWidth* - 0.5  
 *if* yii > -1 *and* yii < *WindowWidth and* xii > -1 *and* xii < *WindowWidth*:  
 WindowRow = int(np.round(Point[1] + i))  
 WindowCol = int(np.round(Point[0] + j))  
 *if* WindowRow > 0 *and* WindowRow < ImgHeight - 1 *and* WindowCol > 0 *and* WindowCol < ImgWidth - 1:  
 dx = GaussianImg[WindowRow, WindowCol + 1] - GaussianImg[WindowRow, WindowCol - 1]  
 dy = GaussianImg[WindowRow - 1, WindowCol] - GaussianImg[WindowRow + 1, WindowCol]  
 Magnitude = np.sqrt(dx \* dx + dy \* dy)  
 Orientation = np.rad2deg(np.arctan2(dy, dx)) % 360  
 Weight = np.exp(-0.5 / ((0.5 \* *WindowWidth*) \*\* 2) \* ((yi / ThreeSigma\_oct) \*\* 2 + (xi / ThreeSigma\_oct) \*\* 2))  
 KeyPointInfomations.append((yii, xii, Weight \* Magnitude, (Orientation - Angle) \* (*LenOfHistogram* / 360.)))  
  
 *for* yii, xii, Magnitude, OrientationBin *in* KeyPointInfomations:  
 yii\_floor, xii\_floor, o\_floor = np.floor([yii, xii, OrientationBin]).astype(np.int)  
 dr, dc, do = yii - yii\_floor, xii - xii\_floor, OrientationBin - o\_floor  
 *if* o\_floor < 0:  
 o\_floor += *LenOfHistogram  
 if* o\_floor >= *LenOfHistogram*:  
 o\_floor -= *LenOfHistogram* HistogramTensor[yii\_floor + 1, xii\_floor + 1, o\_floor % *LenOfHistogram*] += Magnitude \* (1 - dr) \* (1 - dc) \* (1 - do)  
 HistogramTensor[yii\_floor + 1, xii\_floor + 1, (o\_floor + 1) % *LenOfHistogram*] += Magnitude \* (1 - dr) \* (1 - dc) \* do  
 HistogramTensor[yii\_floor + 1, xii\_floor + 2, o\_floor % *LenOfHistogram*] += Magnitude \* (1 - dr) \* dc \* (1 - do)  
 HistogramTensor[yii\_floor + 1, xii\_floor + 2, (o\_floor + 1) % *LenOfHistogram*] += Magnitude \* (1 - dr) \* dc \* do  
 HistogramTensor[yii\_floor + 2, xii\_floor + 1, o\_floor % *LenOfHistogram*] += Magnitude \* dr \* (1 - dc) \* (1 - do)  
 HistogramTensor[yii\_floor + 2, xii\_floor + 1, (o\_floor + 1) % *LenOfHistogram*] += Magnitude \* dr \* (1 - dc) \* do  
 HistogramTensor[yii\_floor + 2, xii\_floor + 2, o\_floor % *LenOfHistogram*] += Magnitude \* dr \* dc \* (1 - do)  
 HistogramTensor[yii\_floor + 2, xii\_floor + 2, (o\_floor + 1) % *LenOfHistogram*] += Magnitude \* dr \* dc \* do  
  
 DescriptorVector = HistogramTensor[1:-1, 1:-1, :].flatten()  
 # 消除光照影响  
 Threshold = np.linalg.norm(DescriptorVector) \* *DescriptorMaxValue* DescriptorVector[DescriptorVector > Threshold] = Threshold  
 DescriptorVector /= max(np.linalg.norm(DescriptorVector), float\_tolerance) # 与OpenCV一致  
 # 仿照OpenCV  
 DescriptorVector = np.round(512 \* DescriptorVector)  
 DescriptorVector[DescriptorVector < 0] = 0  
 DescriptorVector[DescriptorVector > 255] = 255  
 Descriptors.append(DescriptorVector)  
 *return* np.array(Descriptors, dtype='float32')

## splice.py代码

*import* sift  
*import* tools  
*import* numpy *as* np  
*import* cv2  
*class* Stitcher:  
 """  
 图片拼接  
 """  
 *def* stitch(self, *images*, *ratio*=0.75, *reprojThresh*=4.0, *showMatches*=*False*):  
 # 获取输入图片，输入两张彩色图片  
 (imageB, imageA) = *images* # # OpenCV默认读取顺序是BGR，如果是cv2读取图片，要反过来  
 # imageB = imageB[:,:,::-1]  
 # imageA = imageA[:,:,::-1]  
  
 # 检测A、B图片的SIFT关键特征点，并计算特征描述子  
 kpsA, featuresA = self.detectAndDescribe(imageA)  
 kpsB, featuresB = self.detectAndDescribe(imageB)  
  
 # 匹配两张图片的所有特征点，返回匹配结果  
 M = self.matchKeypoints(kpsA, kpsB, featuresA, featuresB, *ratio*, *reprojThresh*)  
  
 # 如果返回结果为空，没有匹配成功的特征点，退出算法  
 *if* M *is None*:  
 *return None* # 否则，提取匹配结果  
 # H是3x3视角变换矩阵  
 (matches, H, status) = M  
 # 将图片A进行视角变换，result是变换后图片  
 result = cv2.warpPerspective(imageA, H, (imageA.shape[1] + imageB.shape[1], imageA.shape[0]))  
  
 # 将图片B传入result图片最左端  
 result[0:imageB.shape[0], 0:imageB.shape[1]] = imageB  
  
 # 检测是否需要显示图片匹配  
 *if showMatches*:  
 # 生成匹配图片  
 vis = self.drawMatches(imageA, imageB, kpsA, kpsB, matches, status)  
 # 返回结果  
 *return* (result, vis)  
  
 # 返回匹配结果  
 *return* result  
  
 *def* detectAndDescribe(self, *image*):  
 # # 将彩色图片转换成HSI # 老师在群里说可以用HSI，处理I。这里不是很懂，SIFT应该就是计算灰度图的，可能老师说的是SIFT的彩色描述子，大家可以去试一下  
 # hsi = tools.RGB2HSI(image)  
 # h, s, i = [hsi[:, :, i] for i in range(3)]  
 # gray = i  
  
 # # 将彩色图片转换成灰度图  
 # gray = tools.GenerateGrayImg(image)  
  
 # 建立SIFT生成器  
 descriptor = cv2.xfeatures2d.SIFT\_create()  
 # 检测SIFT特征点，并计算描述子  
 (kps, features) = descriptor.detectAndCompute(*image*, *None*)  
  
 # # 建立ORB生成器  
 # orb = cv2.ORB\_create()  
 # # 检测关键点和特征描述  
 # kps, features = orb.detectAndCompute(image, None)  
  
 # # 自己写的sift  
 # kps, features = sift.SIFT(gray)  
  
 # 将结果转换成NumPy数组  
 kps = np.float32([kp.pt *for* kp *in* kps])  
  
 # 返回特征点集，及对应的描述特征  
 *return* kps, features  
  
 *def* matchKeypoints(self, *kpsA*, *kpsB*, *featuresA*, *featuresB*, *ratio*, *reprojThresh*):  
 # 建立暴力匹配器  
 matcher = cv2.DescriptorMatcher\_create("BruteForce")  
  
  
 # 使用KNN检测来自A、B图的SIFT特征匹配对，K=2  
 rawMatches = matcher.knnMatch(*featuresA*, *featuresB*, 2)  
  
 matches = []  
 *for* m *in* rawMatches:  
 # 当最近距离跟次近距离的比值小于ratio值时，保留此匹配对  
 *if* len(m) == 2 *and* m[0].distance < m[1].distance \* *ratio*:  
 # 存储两个点在featuresA, featuresB中的索引值  
 matches.append((m[0].trainIdx, m[0].queryIdx))  
  
 # 当筛选后的匹配对大于4时，计算视角变换矩阵  
 *if* len(matches) > 4:  
 # 获取匹配对的点坐标  
 ptsA = np.float32([*kpsA*[i] *for* (\_, i) *in* matches])  
 ptsB = np.float32([*kpsB*[i] *for* (i, \_) *in* matches])  
  
 # 计算视角变换矩阵  
 (H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC, *reprojThresh*)  
  
 # 返回结果  
 *return* (matches, H, status)  
  
 # 如果匹配对小于4时，返回None  
 *return None  
  
 def* drawMatches(self, *imageA*, *imageB*, *kpsA*, *kpsB*, *matches*, *status*):  
 # 初始化可视化图片，将A、B图左右连接到一起  
 (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* # 联合遍历，画出匹配对  
 *for* ((trainIdx, queryIdx), s) *in* zip(*matches*, *status*):  
 # 当点对匹配成功时，画到可视化图上  
 *if* s == 1:  
 # 画出匹配对  
 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)  
  
 # 返回可视化结果  
 *return* vis  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *import* imageio  
 # 读取拼接图片  
 imageA = imageio.imread("11.jpg")  
 imageB = imageio.imread("22.jpg")  
  
 imageA = cv2.resize(imageA, (256, 256))  
 imageB = cv2.resize(imageB, (256, 256))  
  
 tools.ShowPict(imageA)  
 tools.ShowPict(imageB)  
  
 # 把图片拼接成全景图  
 stitcher = Stitcher()  
 (result, vis) = stitcher.stitch([imageA, imageB], showMatches=*True*)  
  
 tools.ShowPict(vis)  
 tools.ShowPict(result)