SIFT 算法
原理与 OpenCV 源码解读

Dezeming Family

Dezeming

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列书，可以从我们的网站 *[https://dezeming.top/]* 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

0.1 本书前言

Sift 是我接触到的第一个计算机视觉相关的比较高级的算法，虽然时过境迁，但鉴于该算法的思想性和作用，我认为仍然是从图像处理到计算机视觉过渡的必学算法。该算法相比于一些基础算法，多了很多数学原理的应用与分析。

因为它实在是太经典和太重要了，因此我会在本书详细介绍 Sift 算法的基本原理，包括基本的数学原理和意义，然后解读 OpenCV 中的代码实现。OpenCV 源码如果涉及一些比较高级的 C++ 知识，这里也会进行讲解。当初我在学习的时候看了很多博客，但没有感觉那些博客讲得很好很清晰，也是走了一些弯路才弄明白基本原理的。网上的参考资料鱼龙混杂，很多作者的理解和记录也都互相冲突，实在不利于大家自学，而论文 [1][2] 也是晦涩难懂，讲述方法跳跃。

我不得不说，SIFT 不是一个很简单的算法，尤其是对于只有大学本科基本数学基础的人来说更是如此，很多人断断续续研究大半年都是搞不太明白里面的内容。因此，我写本书的初衷是为了写一个真正通俗易懂，详细全面的 Sift 算法教程。尽管网上资料混乱且水平参差不齐，但为了更好地进行叙述和梳理，在构建本书时参考了大量的博客和技术摘要，以此更好讲述 Sift 技术。在参考文献部分大家可以看到我参考的网络资料和论文以及书籍。

本书定价为 18 元，但用户可以免费获取并使用。如果您得到了本书，在学习中对自己有所帮助，可以往 DezemingFamily 的支付宝账户（17853140351）支持 18 元。您的支持将是我们 DezemingFamily 发布更多电子书和努力提高电子书质量的动力！本书是我定价到目前为止最高的一本书，但我认为它是值得的。比起自己当初花费了巨大的精力去搜寻资料和苦苦思索，这本书我相信是一个非常好的从入门到完全理解 Sift 算法的教材。

2021-7-2：本书发布第一版。

1. Sift 算法基本介绍

本章介绍什么是图像特征，我们需要的特征应该具备什么特点。并介绍 *Sift* 算法的作用、意义，简单介绍一下该算法的基本过程。

1.1 Sift 简介

SIFT 全名为尺度不变特征转换 (即 Scale-invariant feature transform)，是一种计算机视觉算法，用来检测与描述图像中的局部特征。对于图像特征，我们应该知道的是，图像在经过一定的仿射变换后它的特征应该是不变的，比如你的眼角有一颗痣，把你在图像上的位置进行平移和旋转，我们还是能通过这颗痣将你识别出来，这个痣一直都在你的眼角，这就是你的不变特征。

SIFT 算法就是在空间尺度中寻找特征，并提取出其位置、尺度和旋转不变量，此算法由在 1999 年所发表 [1]，2004 年又进行了完善和总结 [2]。

该算法的特点主要是：

- 平移旋转不变性、尺度缩放不变性。
- 明暗度变化保持不变性，即场景亮度变化，该特征不变。
- SIFT 特征是图像的局部区域的特征。
- 对噪声不敏感，有一定的噪声容错能力。
- 信息量丰富，信息具有独特性，匹配准确。
- 简单场景也可以产生大量 SIFT 特征。

当然，虽然说得这么神乎其神，特点那么多，但其实有很多特点都是固有特点，比如 SIFT 特征多为边缘点和角点，因此不太受光照影响，仅仅受到图像局部颜色变化梯度的影响。而尺度不变是因为在不同的尺度空间上计算特征，关于不同的尺度空间，我们可以先简单理解为将原图像进行模糊。

1.2 Sift 算法的基本步骤

SIFT 其实总结来说只分为两步即可，第一步是特征点检测，第二步是生成特征描述。检测其实与一般的边缘检测没有本质区别，但是如何描述我们的特征，比如一个人，不论她离你很远或者很近，她站着还是躺着，你还是能识别出来这个人是谁，这就说明你掌握了这个人的某种不变性特征。

在论文 [4] 中，SIFT 算法分为 4 步，因为还没有讲述原理，所以下面几个阶段暂时不需要完全弄懂，只要大体知道做了什么就可以了，我们后面会详细进行介绍。

- 第一阶段的计算搜索所有的尺度上的所有图像区域。该算法利用高斯函数的微分来识别对尺度和方向不变性的潜在兴趣点。
- 第二阶段是关键点定位，在每个候选位置，一个精细的模型用于确定位置和尺度。关键点是根据它们的稳定性来选择的。
- 第三阶段是方向分配，基于局部图像梯度方向为每个关键点位置分配一个或多个方向。所有的后续操作都是对已相对于每个特征的指定方向、比例和位置进行变换的图像执行的，从而为这些变换提供不变性。
- 最后生成关键点描述子（descriptor，又叫描述符），在每个关键点周围区域的选定比例上测量局部图像梯度。这些被转换成一种表示方法，允许较大的局部形状失真和照明变化。

分别寻找到两张图像的特征描述后，就可以使用匹配算法将两张图像的特征进行匹配了。我会先讲解 SIFT 特征的提取方法，后面再介绍如何使用这些特征进行图像匹配。

关于基本步骤，我再简单提一句：对于 SIFT 算法，最难的地方在于尺度不变性的构建，至于其他的不变性其实并不难理解。相信跟着本书走，您就能够清晰的学会 SIFT 算法。

1.3 本书结构安排

本书会首先介绍基本的图像处理方法和基本的数学原理，然后介绍 SIFT 构建尺度空间的原理和流程。

然后接下来会介绍如何生成特征的描述方法，因为 Sift 发展了很多年，出现了各种乱七八糟的解释，我会花大量篇幅去介绍这里面的内容。希望读者能够对里面的操作有一个具体的印象，但不必把全部细节都搞懂，因为后面我会在源码解析中告诉大家最经典的方法是怎么做的。

介绍完基本原理以后开始会讲解 OpenCV3 的 Sift 源码的实现方法。

然后会讲 Sift 的重要应用：图像匹配算法的基本原理。

之后会介绍 OpenCV3 实现的图像匹配实现方法。

SIFT 算法虽然已经产生了二十年了，但并不是一个简单的算法，因此大家在学习时一定要注意认真理解，我在写书时也会尽可能讲得清楚和明白。

2. Sift 尺度空间

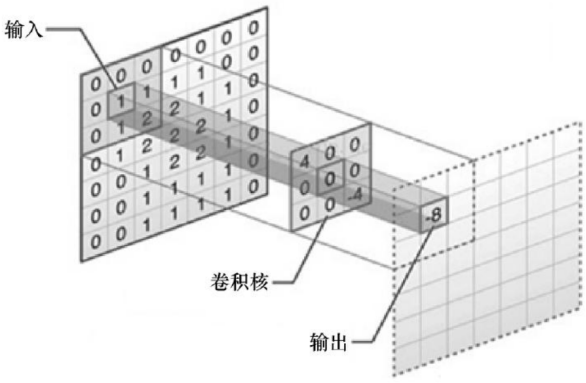
本章第一节会介绍 *SIFT* 算法用到的图像处理和数学方面的知识。然后根据算法流程介绍 *SIFT* 中尺度空间构建的基本过程。本章的内容主要是 *SIFT* 尺度空间的构建以及从尺度空间中找到特征点，主要使用 *DOG* 算子进行，其实与一般的边缘和角点检测并没有本质区别，只是在不同尺度下会得到多尺度下的特征。

本节大家可能会对其中的很多概念，例如不同的尺度，组内尺度，金字塔尺度等混乱不清，如果您读了两遍也不明白具体的细节，那么我相信源码解读章节一定会给您一个足够清晰和明确的解释。

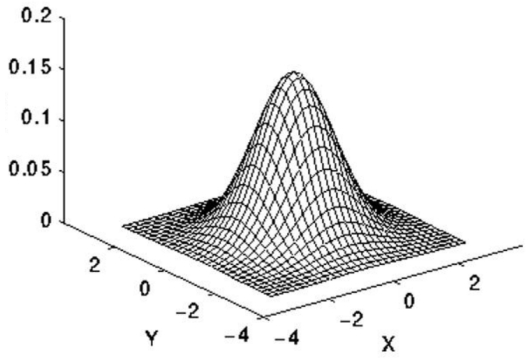
2.1 图像处理和数学基础

补充 2.1 (高斯卷积)

构建尺度空间的方法是将图像进行模糊，模糊是将局部几个像素的信息合并到一个像素里。图像卷积其实就是在模糊范围里，每个像素与卷积核（又叫滤波核）对应元素相乘再相加。卷积核不断移动，完成对整张图像的模糊。示意如下，输出的图像每个像素都包含了原图像中以该像素为中心，周边 8 个像素值的总影响：



而在构建尺度空间时，Koenderink（1984）和 Lindeberg（1994）已经证明，在各种合理的假设下，唯一可能的尺度空间核是高斯函数。高斯核函数表示如下：



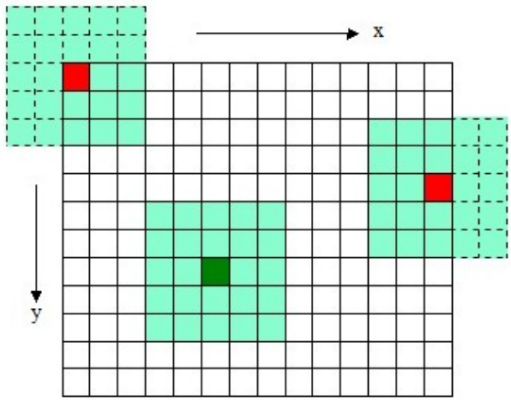
高斯函数又叫正态函数，符合高斯分布的 x 的概率密度为：

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{2.1}$$

图像卷积比一维信号的卷积好理解多了，我们也不单独介绍什么是信号卷积，毕竟这里用不到。其中 σ 是表示正态分布的标准差，注意 σ 越大，信号会变得越平滑（越模糊）。但是图像是二维的，而且图像数据不是连续的函数，对于二维高斯模糊，我们可以计算卷积核模板，对于 $m \times n$ 的卷积核计算公式为：

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-0.5m)^2+(y-0.5n)^2}{2\sigma^2}} \tag{2.2}$$

注意高斯卷积可能会丢失边缘区域，如下图的红色像素，我们需要进行单独处理：



比如可以计算卷积核区域内有效可卷积像素样本，然后加权累加，之后除以权重的和，其中 h_i 表示对于的高斯卷积权重：

$$I = \frac{\sum_{i=0}^{i=a} h_i I_i}{\sum_{i=0}^{i=a} h_i} \tag{2.3}$$

在做高斯模糊时，我们知道高斯核的实际大小是无穷的，但是远离中心的概率密度非常小，可以忽略不计，一般我们在进行高斯模糊时，只需要计算 $(6\sigma+1, 6\sigma+1)$ 大小的滤波核即可（为了程序方便性我们也不需要非得让滤波核设置为一个圆形区域，用方形区域就可以了）。

所谓尺度不变，意思就是，你隔着我两米我知道这是你，你离着我十米我还是能认出你。我们定义尺度空间的构建表示为：

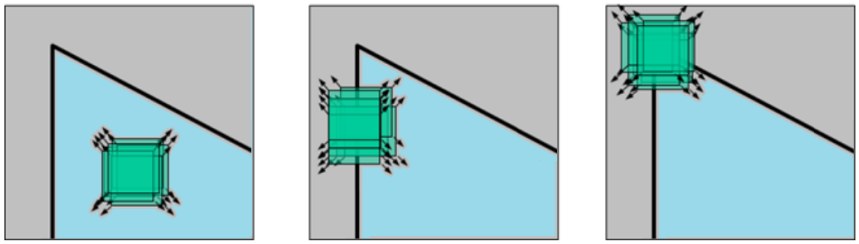
$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y) \tag{2.4}$$

其中，滤波核的范围和构建前面已经详细介绍过了。我们设置不同的 σ，计算出相应的滤波核半径，然后就可以进行图像卷积，得到不同的尺度图像了。σ 越大，卷积范围越大，图像就越模糊，反之亦然。

补充 2.2 (角点检测)

在图像匹配中，我们一般提取图像局部特征，有几个好处：特征是局部的，所以对图像上大范围遮挡和杂乱的位置表现非常鲁棒；局部特征也具有一定独特性，作为可以区分对象的大型数据库（数量多）；局部性因此计算量少，可实时实现特征检测；对特征有一定概括性，在不同情况下可以利用不同类型的功能。因此，在图像局部区域中，我们认为特征是具有颜色明显变化的位置，准确来说角点不是特征，但检测出来的角点可以用来提取和表示总结为特征。

在局部小范围里，如果在各个方向上移动窗口，窗口内区域的灰度发生较大变化，就认为在窗口内遇到了角点；如果窗口在各个方向移动时，窗口内图像的灰度没有发生变化，那么窗口内就不存在角点。



角点检测然后提取特征是比较基础的方法，详情可见 DezemingFamily 的《Harris 角点检测原理及 OpenCV 代码描述》书。

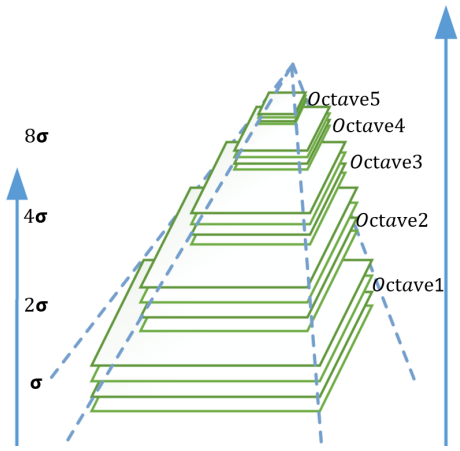
2.2 LOG 和 DOG

LOG 全称为 Laplacian of Gaussian，即高斯拉普拉斯算子。DOG 全称为 Difference of Gaussian，即高斯差分。这两种方法是用来检测极值点（突变信息、边缘信息）的，其中 DOG 是用来近似 LOG 的。

这两种算子都非常简单，详细的代码实现过程可以参考 DezemingFamily 的《边缘检测 LOG 与 DOG 原理与 OpenCV 代码实战》，该书对这两种算子的数学原理进行了讲解。

2.3 Sift 尺度空间的构建

在 SIFT 中尺度空间一般就是构建高斯金字塔，构建方法为（1）对图像做高斯模糊（2）对图像做降采样。金字塔就是不断降采样得到的：



其中 Octave 表示产生的图像组数，每组图像有 Interval 个图像（表示为图像层），因此一组图像有多个层。每层图像都是该组初始图像（该组最底层的图像）用不同高斯核做高斯卷积得到不同尺度得到的。每组图像的初始图像是上一组图像中的某层图像（比如我们先假定为上一组最底层的图像）进行隔点采样得到的。）我们先不要管上图中左边的 $2^o\sigma$ 这些系数是怎么来的，后面会讲。

一般而言，在最底图像组里（假设有 L 层），每一层的图像的平滑因子分别对应：

$$0, \sigma, k\sigma, k^2\sigma, k^3\sigma, ..., k^{L-2}\sigma \tag{2.5}$$

不过实际中其实我们一般使用平滑因子为：

$$\sigma, k\sigma, k^2\sigma, k^3\sigma, ..., k^{L-1}\sigma \tag{2.6}$$

也就是说我们舍弃了最高空域采样率，将输入图像进行模糊来作为第 0 个组的第 0 层图像。我们后面介绍坐标时会再提到。

DOG 图像金字塔的基本原理

我们简单阐述一下原理。

Lindeberg（1994）发现尺度归一化的 DOG 是 LOG $\sigma^2\nabla^2 G$ 的很好的近似。之后 Mikolajczyk（2002）发现 $\sigma^2\nabla^2 G$ 的极大极小值相比于梯度、Hessian 或 Harris 角点检测来说，可以提供更稳定的图像特性。DOG 与 LOG 的关系可以推导得到：

$$\frac{\partial G}{\partial \sigma} = \frac{-2\sigma^2 + x^2 + y^2}{2\pi\sigma^3} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.7}$$

$$\sigma\nabla^2 G = \frac{-2\sigma^2 + x^2 + y^2}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.8}$$

$$\frac{\partial G}{\partial \sigma} = \sigma\nabla^2 G \tag{2.9}$$

利用差分近似来代替微分：

$$\sigma\nabla^2 G = \frac{\partial G}{\partial \sigma} = \lim_{\Delta\sigma\to 0}\frac{G(x,y,\sigma+\Delta\sigma)-G(x,y,\sigma)}{(\sigma+\Delta\sigma)-\sigma} \tag{2.10}$$

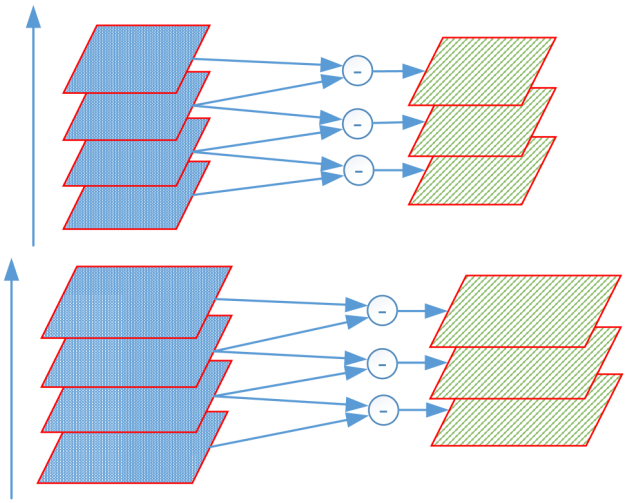
$$\approx \frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma-\sigma} \tag{2.11}$$

$$\Longrightarrow G(x,y,k\sigma)-G(x,y,\sigma)\approx(k-1)\sigma^2\nabla^2 G \tag{2.12}$$

我们可以看到 $(k-1)$ 在所有尺度下都是一个常数，因此并不影响极值点位置的求取。

当 k 趋近于 1 的时候误差趋近于 0，在实际应用中发现，即使在尺度上存在显著差异，这种近似也几乎不影响极值检测和定位的稳定性（例如我们可以选择 $k=\sqrt{2}$）。

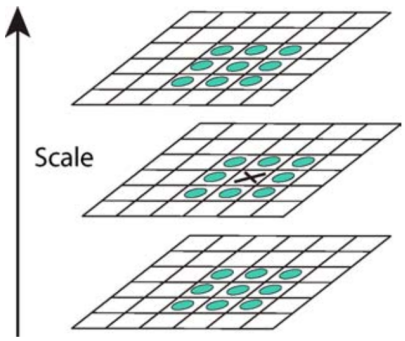
在每组之间进行差分运算（即 DOG 运算），其实这也是因为我们需要计算任意尺度空间的特征描述，因此这样操作可以将图像简单相减就行了，比 LOG 计算更简单。



我们再重复一遍，每层图像都是该组初始图像（该组最底层的图像）用不同高斯核做高斯卷积得到不同尺度得到的。每组图像的初始图像是上一组图像某层进行隔点采样（以 2 为因子下采样）得到的，然后再通过高斯模糊得到该组其他层的图像。

局部最大极值点

在 DOG 图像金字塔上，我们需要检测 DOG 的极值点，如下图所示，检测极值点不但包括与本层临近像素之间相互比较，还需要和上下两层的像素之间相互比较，一共需要比较 $8+2*9=26$ 个像素，这是为了保证在图像空间和尺度空间都能检测到极值点：



仅当该层某像素大于所有这些临近像素或小于所有临近像素时，才会选中它作为极值点。此检测的成本相当低，因为在最初的几次检测之后，大多数样本点都会被去除了。

为了在每组中检测出 S 个尺度的极值点，则 DOG 图像金字塔每组需 $S+2$ 层图像，而 DOG 图像金字塔是由高斯图像金字塔相邻两层相减得到的，因此高斯图像金字塔每组需 $S+3$ 层图像，一般 S 会取值在 $[3,5]$ 之间，Lowe 建议 S 为 3，后面会介绍为 3 的好处。

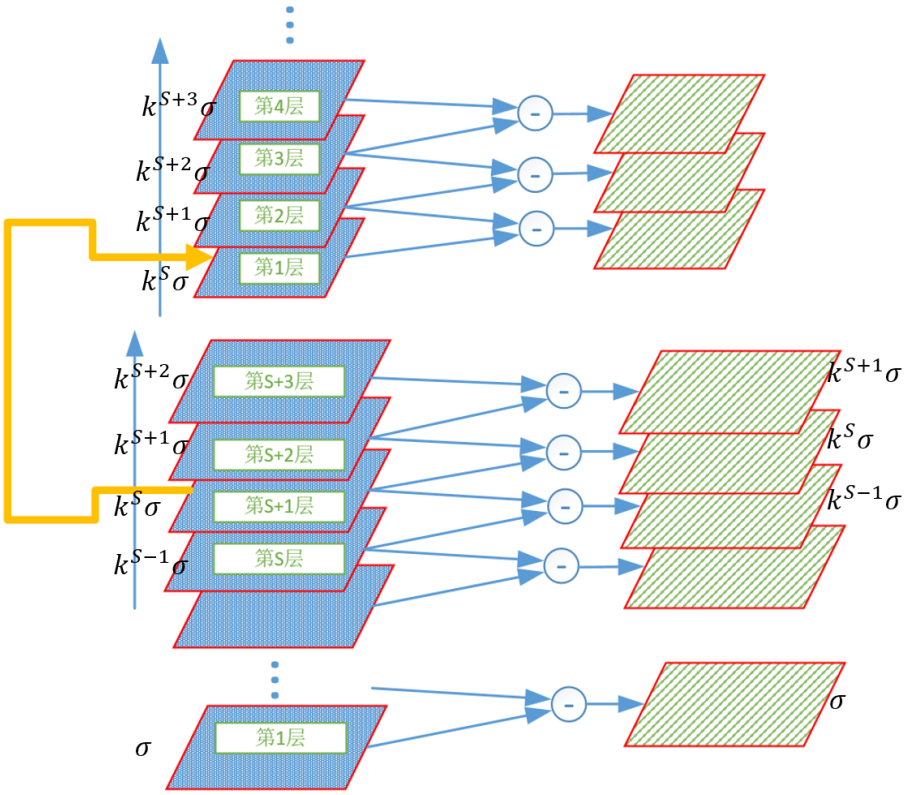
2.4 尺度空间的坐标表示

在这个尺度空间里，如何表示一个坐标呢？

我们设三个变量：σ 表示尺度空间坐标，O 表示组数，o 表示当前组的索引，$o \in [0,...,O-1]$；S 上一节已经叙述过，表示我们要检测局部最大极值点时的极值点分布层数，s 表示当前层的索引，$s \in [0,...,s+2]$。

尺度空间是由 (o,s) 构成的，o 表示尺寸，s 表示模糊程度。

我们知道每个组里有 $S+3$ 层图像，得到的 DOG 图像一共有 $S+2$ 层，我们用一个示意图来描述：



其中：

$$k = 2^{\frac{1}{S}} \tag{2.13}$$

DOG 图像的尺度表示，我们这里再定义一下：

$$D(x,y,\sigma) = \Big(Gauss(x,y,\sigma(s+1)) - Gauss(x,y,\sigma(s))\Big) * I(x,y) \tag{2.14}$$

因此我们就知道第 o 组图像的第一层图的模糊系数为 $2^o\sigma$ 。

因此由上面可知，尺度坐标定义为：

$$\sigma(o,s) = \sigma_0 2^{o+\frac{s}{S}} = \sigma_0 2^o k^s \tag{2.15}$$

其中 σ_0 表示第 1 组第 1 层对于原始图像的模糊系数，即上图中的 σ。

而对于一组内某一层图像尺度，我们定义为 $\sigma_L = \sigma_0 2^{\frac{s}{S}}$, $s \in [0,...,S+2]$。

关于第一层的模糊

定义一下，初始图像 I 是我们拥有的图像的初始版本，原始图像是自然图像。我们一般认为我们拥有的初始图像是原始图像在获取时经过高斯模糊以后得到的。

我们在上面构建的高斯金字塔里，第一组的最下面一层图像已经是被 σ 因子模糊过的了，因此我们需要上采样（或者我们直接把原图当成已经被 σ 因子模糊过的图也行）。

当相机拍摄图像时，其实镜头已经算是对图像进行了一次模糊了，我们想得到该图像的上一组图像，因此有时候我们会设定一个第 0 组，o 坐标在第 0 组表示为 -1，当然这不是绝对的，有两种主要方法。我们想获得原始图像经过 $\sigma_0 = 1.6$ （这是 lowe 实验以后得到的比较稳定的数值）模糊以后得到的第 1 组第 1 层图像。

关于 $S = 3$ 和 $\sigma_0 = 1.6$ 是怎么得到的，论文 [2] 中有解释，其实就是测试不同的值所对应的检测到的关键点数和正确关键点数的比值（因为检测到的关键点不一定就是真的关键点），从而发现取这些值时测试比较稳定，检测到的正确关键点的比例比较大。

首先先说一下方法一。

第一组第一层图像是原图像与 $\sigma_0 2^0 k^0 = \sigma_0$ 卷积得到的。但问题是，我们原图像并不是我们的初始图像 I ，我们设初始图像是已经被 σ_g 高斯模糊过的图像了，即我们并不知道原图像，但我们已经有了被 $\sigma_g = 0.5$ 模糊过的图像，我们想求被 $\sigma_0 = 1.6$ 模糊过的图像，因此根据高斯平滑的关系，应该用 $\sigma = \sqrt{\sigma_0^2 - \sigma_g^2} = \sqrt{1.6^2 - 0.5^2} \approx 1.52$ 去平滑初始图像，得到第 1 组第 1 层的图像。

再说一下方法二。

我们一般会假设输入图像是经过高斯 $\sigma_g = 0.5$ 平滑处理后的，即我们假设一个第 0 组，o 索引为 -1。这组图像首先是初始图像 I 经过双线性插值放大两倍以后得到的。因为 $I(x,y)$ 已经是被 $\sigma_g = 0.5$ 模糊过的图像了，因此 I 放大两倍以后得到的 I_s 可以看做是被 $2\sigma_g$ 高斯模糊后的原始图像。

我们先双线性插值得到 I_s，然后使用高斯核 $\sigma = \sqrt{\sigma_0^2 - (2\sigma_g)^2}$ 进行模糊，得到第 1 组第一层的图像。

获取倒数第三层图像

为什么是上一组提取倒数第三层图像然后再进行降采样呢？我们观察上面的图可知本组第一层图像的尺度因子与上一组的倒数第三层尺度因子是一致的，因此获取本组第一层图像的过程就不再需要根据原图再卷积了，而是可以直接降采样来得到本组的初始图像。

同时我们可以看到用来做关键点检测的 DOG 层也要保证连续，相邻两组全部的 DOG 层（每三层为一个比较单元，中间点要与邻域和上下层邻域之间相互比较）为（设 $S = 3$ ，第一组初始

为 $\sigma(s)$)：

$$\sigma_s,\ 2^{\frac{1}{3}}\sigma_s,\ 2^{\frac{2}{3}}\sigma_s,\ 2^{\frac{3}{3}}\sigma_s,\ 2^{\frac{4}{3}}\sigma_s \tag{2.16}$$

$$2^{\frac{3}{3}}\sigma_s,\ 2^{\frac{4}{3}}\sigma_s,\ 2^{\frac{5}{3}}\sigma_s,\ 2^{\frac{6}{3}}\sigma_s,\ 2^{\frac{7}{3}}\sigma_s \tag{2.17}$$

用来做关键点检测的 GOD 层尺度因子为：

$$2^{\frac{1}{3}}\sigma_s,\ 2^{\frac{2}{3}}\sigma_s,\ 2^{\frac{3}{3}}\sigma_s \tag{2.18}$$

$$2^{\frac{4}{3}}\sigma_s,\ 2^{\frac{5}{3}}\sigma_s,\ 2^{\frac{6}{3}}\sigma_s \tag{2.19}$$

可以看到这样每组之间和每层之间的尺度因子就都是连续的了，因此我们不会漏掉其中的某些尺度。

构建每组内的图像

在构建高斯金字塔的时候，组内每层的尺度坐标表示如下，不同组相同层的组内尺度坐标 $\sigma(s)$ 是相同的，该组的第 s 层图像是 $s-1$ 层以 $\sigma(s)$ 进行模糊得到的：

$$\sigma(s) = \sqrt{(k^s\sigma_0)^2 - (k^{s-1}\sigma_0)^2} \tag{2.20}$$

使用相对尺度来模糊下一层的图像或许是为了编程和实现的时候更能体现“层与层”之间的差值，如果直接使用本组初始图像去模糊得到本组其他不同尺度的图像，则很有可能得到的 DOG 图像响应非常弱（从数值精确度的角度去考虑），总之，实际实践时可以有多种不同的方法。

3. Sift 特征描述

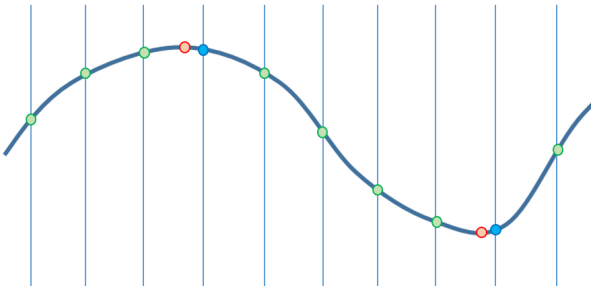
本质上来说，特征描述才是 *Sift* 最本质和核心的内容。

本章会详细介绍特征描述的原理，但为了保证原理介绍不过于冗杂，一些具体的实践方法我会有所省略，而在后面的源码中您就能找到足够详细和透彻的实际实现方法。

3.1 关键点定位

我们上一章已经讲过在 DOG 图像层中得到关键点的方法，即与上下两层邻域即本层邻域之间相互比较，找到极值点。

但问题是这些离散空间的极值点并不一定是真的极值点：



如上图，很可能出现的情况是：检测出的极值点为蓝色点，但是实际为红色点，因此就需要在离散点之间插值来得到连续空间的极值。在 1999 年的论文 [1] 时并没有进行精准定位，只是找到蓝色点作为极值位置即可，在 2002 年的论文 [4] 中提出可以拟合三维二次函数来更精确地定位关键点位置与尺度。

首先需要对 DOG 函数进行拟合，我们把尺度空间函数定义为 $D(\boldsymbol{x})$，其中 $\boldsymbol{x}=(x,y,\sigma)^T$，这是一个有三元变量的函数，当然也是可以和微积分一样进行展开的。DOG 在尺度空间的泰勒展开（参考 DezemingFamily 的《多元函数（及向量函数）的泰勒展开》一书）为：

$$D(\boldsymbol{x}) = D + \frac{\partial D^T}{\partial \boldsymbol{x}}\boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^T\frac{\partial^2 D}{\partial \boldsymbol{x}^2}\boldsymbol{x} \tag{3.1}$$

$$\boldsymbol{x} = (x,y,\sigma)^T \tag{3.2}$$

其实这个式子是写的很有欺骗性，大家看了肯定很迷茫，论文 [2] 中其实写的很不规范，本节最后我会重新说明。对函数求导，并令导数为 0，就可以得到极值点的偏移量为：

$$\hat{\boldsymbol{x}} = -\frac{\partial^2 D^{-1}}{\partial \boldsymbol{x}^2}\frac{\partial D}{\partial \boldsymbol{x}} \tag{3.3}$$

代表相对于当前关键点位置的偏移量。当在任一维度上偏移量 (x,y) 大于 0.5 时，表示插值中心已经便宜到了它邻近点上了，因此必须改变当前的关键点的位置，同时在新的位置上反复插值直到收敛（lowe 论文中迭代 5 次）。如果迭代中超出了图像范围，就删除这个点。

对应我们求出的极值点偏移，DOG 函数值为：

$$D(\hat{\boldsymbol{x}}) = D + \frac{1}{2}\frac{\partial D^T}{\partial \boldsymbol{x}}\hat{\boldsymbol{x}} \tag{3.4}$$

在第五章我源码介绍时我会再进行更详细的描述，大家也可以根据 DezemingFamily 的《多元函数泰勒展开》以及《函数对向量求导-通俗易懂的描述》书进行学习。这里的 D 只是一个函数，而矩阵是一个函数组，因此这里对多元函数求导的计算就显得非常容易了。

泰勒展开求导

首先我们需要注意的是，$\frac{\partial D^T}{\partial \boldsymbol{x}}$ 表示的是在当前极值点处展开的泰勒函数，泰勒公式是拟合极值点附近位置的 DOG 值。因此上述泰勒公式（来自论文 [2]）并不是那么严谨，没有把 $\frac{\partial D^T}{\partial \boldsymbol{x}}$ 中的当前检测到的极值点和我们想拟合的其他位置的极值点区分开。

因此，$\frac{\partial D^T}{\partial \boldsymbol{x}}$ 其实相当于一个常数矩阵，我们在对该函数进行求导的时候，就要把 $\frac{\partial D^T}{\partial \boldsymbol{x}}$ 当成一个常量矩阵，而不是带有自变量 $\boldsymbol{x}$ 的函数，我建议将上述公式描述如下，其中 $\boldsymbol{x_0}$ 表示当前检测出的极值点，$\boldsymbol{x}$ 表示空间中的任意一点，$\hat{\boldsymbol{x}}$ 表示 $\boldsymbol{x}$ 到 $\boldsymbol{x_0}$ 的偏移：

$$\boldsymbol{x_0} = (x_0, y_0, \sigma_0)^T \tag{3.5}$$

$$\boldsymbol{x} = (x, y, \sigma)^T \tag{3.6}$$

$$\hat{\boldsymbol{x}} = \boldsymbol{x} - \boldsymbol{x_0} \tag{3.7}$$

$$D(\boldsymbol{x}) = D(\boldsymbol{x_0}) + \frac{\partial D^T(\boldsymbol{x_0})}{\partial \boldsymbol{x}}\hat{\boldsymbol{x}} + \frac{1}{2}\hat{\boldsymbol{x}}^T\frac{\partial^2 D(\boldsymbol{x_0})}{\partial \boldsymbol{x}^2}\hat{\boldsymbol{x}} \tag{3.8}$$

$$= D(\boldsymbol{x_0}) + \frac{1}{2}\frac{\partial D^T(\boldsymbol{x_0})}{\partial \boldsymbol{x}}\hat{\boldsymbol{x}} \tag{3.9}$$

其中，最后一个等式是把 $\hat{\boldsymbol{x}}$ 代入最终得到的，并不复杂，就不再讲了。

3.2 关键点排查

我们得到的关键点可能响应不够大，过小的点容易受噪声的干扰从而不稳定，因此我们舍弃 $|D(\boldsymbol{x})|$ 过小的点。当值小于 0.03 时，论文 [2] 认为应该舍弃。但是 Rob Hess 在实现代码的是用的不是 0.03，而是 T/S ，其中 T=0.04，我们看一下 OpenCV 源码：

```
1 //OpenCV源码：adjustLocalExtrema函数
2 //nOctaveLayers即检测层数，也就是S；contrastThreshold是0.04。
3 if( std::abs( contr ) * nOctaveLayers < contrastThreshold )
4 return false;
```

DOG 在图像边缘处会有比较大的值（对边缘的响应也很强），但并非所有的边缘点都可以作为稳定的好的特征关键点，我们更喜欢角点作为关键点，而不仅仅只是物体的边界区（因为边界区并不是一个简单的“点”，而是一条线，我们并不能用一个点的信息去代表一条线的特征）。这一步称为“消除边缘响应”，我们消除的是 DOG 图像的边缘响应，我们引入海森（Hessian）矩阵来解决这个问题：

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix} \tag{3.10}$$

由海森矩阵与海森矩阵特征值的性质（见 Dezemingfamily 的《海森（Hessian）矩阵》）以及《矩阵二次型》，我们可以理解为特征值表示 D 在相互垂直方向上的变化速度，设 λ_{max} 为较大特征值，λ_{min} 为较小特征值，$r = \frac{\lambda_{max}}{\lambda_{min}}$。我们不能要 r 特别大的点，因为这说明 D 只在一个方向变化很大，所以肯定不是角点。

计算矩阵的迹：

$$Tr(H) = D_{xx} + D_{yy} = \lambda_{max} + \lambda_{min} \tag{3.11}$$

$$Det(H) = D_{xx}D_{yy} - (D_{xy})^2 = \lambda_{max}\lambda_{min} \tag{3.12}$$

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\lambda_{max} + \lambda_{min})^2}{\lambda_{max}\lambda_{min}} = \frac{(r\lambda_{min} + \lambda_{min})^2}{r\lambda_{min}^2} = \frac{(r+1)^2}{r} \tag{3.13}$$

我们知道 r 一定是大于等于 1 的，而且 $\frac{(r+1)^2}{r}$ 在 $r \geq 1$ 时是单调递增的，要想让 r 不太大，只需要让 $\frac{(r+1)^2}{r}$ 不太大即可，也就是说当我们取某个 $r = r_0$ 时，需要保证：

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r_0+1)^2}{r_0} \tag{3.14}$$

Lowe 建议这里的 r 取 10。

海森矩阵与图像之间的关系

为什么要用二阶导构成的海森矩阵来判断极值是否是角点，因为我们知道我们求出的极值点是一阶导为 0 的点，而二阶导表示的是变化率。

一般图像的泰勒展开写为：

$$f(x,y) \approx f(x_0,y_0) + \begin{bmatrix} f_x' & f_y' \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} \cdot \begin{bmatrix} f_{xx}' & f_{xy}' \\ f_{yx}' & f_{yy}' \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{3.15}$$

泰勒展开的最后一项其实是一个二次型矩阵 $\boldsymbol{x}^T A \boldsymbol{x}$，设中间矩阵为 A：

$$\hat{\boldsymbol{x}}^T \frac{\partial^2 D(\boldsymbol{x_0})}{\partial \boldsymbol{x}^2} \hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}^T A \hat{\boldsymbol{x}} \tag{3.16}$$

如果您了解什么是二次型矩阵，或者看过 DezemingFamily 的《矩阵二次型》，您就能明白，二次型矩阵的特征值与特征向量其实表示的是这个矩阵达到某一个值的快慢。矩阵二次型相乘以后得到的其实是一个数（如上式，大家可以自己乘一下），当中间的矩阵 A 被固定以后，我们设二次型 $\boldsymbol{x}^T A \boldsymbol{x} = a$，其中 $\boldsymbol{x}$ 是自变量。

向量沿着最大的特征值所在方向变化可以最快到达 a 值，沿最小的特征值会最慢到达 a 值，因为该矩阵向量运算的结果表示变化率，所以可以认为特征值大的方向（特征向量的方向）变化率大，特征值小的方向变化率小。

3.3 关键点方向分配

我们已经获得了一些关键点，但是我们需要来描述它们，以获得旋转不变性（即你的脸旋转 30 度还是你的脸）。

基于局部图像属性，为每个关键点指定一致的方向（基准方向），关键点描述子可以相对于该方向来表示，从而实现对图像旋转的不变性。

有许多分配局部方向的方法，我们选择其中最稳定的结果：关键点的尺度信息用来选择离关键点尺度最近的层的尺度图像 L，该层的尺度为 $\sigma_L = \sigma_0 2^{\frac{s}{S}}$，因此使得该操作是在尺度不变的。对于每个图像采样点 $L(x,y)$，我们计算出在这个尺度下的梯度 $m(x,y)$ 和朝向 $\theta(x,y)$：

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{3.17}$$

$$\theta(x,y) = tan^{-1}(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}) \tag{3.18}$$

由围绕关键点的区域内的样本点的梯度方向形成方向直方图。方向直方图有 36 个 bins，涵盖 360 度方向范围（每 10 度划分在一个 bin 里）。添加到直方图中的每个样本都通过其梯度幅度和一个以尺度 σ 高斯加权圆形窗口进行加权，σ 是关键点所在尺度的 1.5 倍（即 $1.5\sigma_L$）。前面说过，尺度采样的范围不用太大，因为离中心点越远贡献越小，可以忽略，因此比较合理的采样范围是 $(6\sigma+1)\times(6\sigma+1)$，也就是说半径为 $3\times\sigma$（即 $3\times1.5\sigma_L$）。

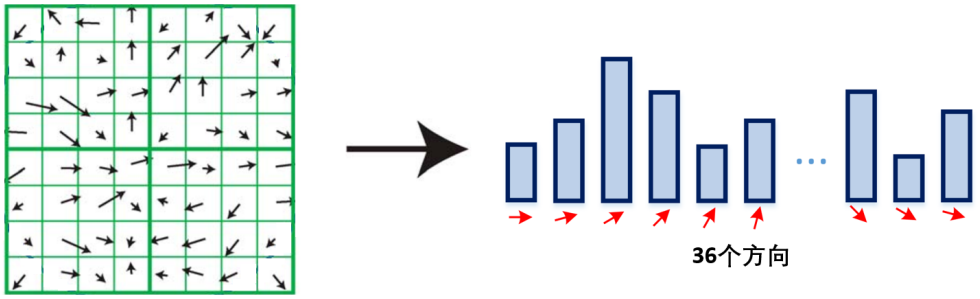
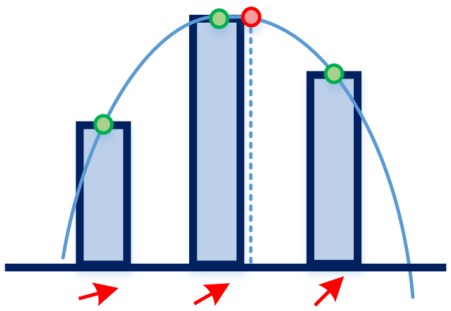


Image gradients

之后我们还需要对直方图进行一下平滑，防止某个方向因为噪声干扰等原因突变，可以使用简单的高斯卷积来平滑，也可以使用其他方法，等到程序解读中我们再介绍 OpenCV 程序使用的平滑方法。

方向直方图中的峰值对应于局部梯度的主导方向。检测直方图中的最高峰值，然后，在最高峰值 80% 以内的任何其他局部峰值也创建具有该方向的关键点（也就是说只有峰值大于主方向峰值 80% 的方向才可以被保留，同时作为该关键点方向的辅助方向）。因此，对于具有相似大小的多个峰值的位置，将在相同的位置和比例创建方向不同的多个关键点（实际其实就是复制成多份关键点，并给这些关键点分配不同的方向）。只有大约 15% 的关键点被分配了多个方向，但是这些方向对匹配的稳定性有很大的贡献。

最后，可以将最接近每一峰值的直方图上的 3 个值拟合成抛物线，对峰值位置进行插值，以提高方向精度：



3.4 关键点特征描述子

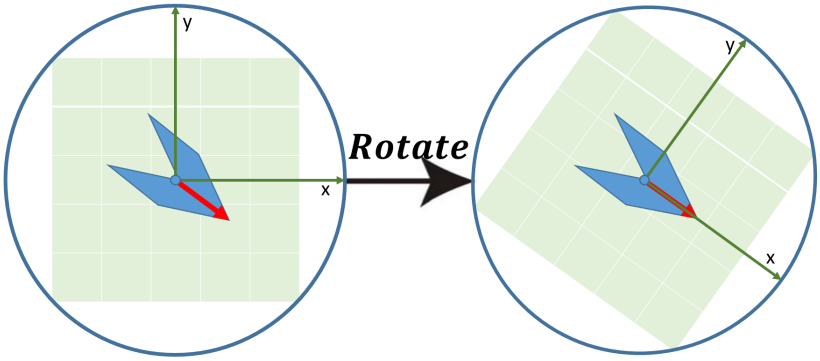
我们已经拥有了大量关键点，我们知道这些关键点的响应值，知道它们的位置和尺度，并且也知道了它们的方向。我们现在需要想办法来描述这些特征，使得当视角变化、光照变化时，这些特征并不会变化。

虽然想出一种特征描述方法并不是多么困难的事情，但是要求其鲁棒性和尽可能好用则也是具有一定的挑战。我们应该让关键点描述子也包含一定周边的像素点，并且对于不同的关键点要保证它们具备一定的独特性，以便匹配成功（比如眼睛的描述和鼻子的描述应该尽可能区分开，不能都是“很小”或“很大”，应该分别描述为“明亮而透彻”和“小巧而挺翘”）。

Sift 描述子其实就是关键点周围区域的高斯图像的信息描述，是一种具有唯一性的抽象。一个明显的方法是在适当的尺度下对关键点周围的局部图像强度进行采样，并使用归一化相关度量来匹配这些强度。然而，图像块的简单相关度量对于一些图像变化（例如仿射或三维视点变化或非刚性变形）容易造成样本匹配错误。

坐标轴旋转

为了保证旋转不变性，必须要先将该局部区域先旋转到关键点的方向上，然后再进行这些操作。需要注意的是，图像梯度早就在预计算中计算出来了，所以我们不必每次都会每个局部区域的范围采样时再计算图像梯度，因此我们只需要划定范围，然后旋转即可。



假如要旋转的角度是 θ，原坐标 $[x,y]$ 旋转以后的新的坐标 $[x',y']$ 就是：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.19}$$

其中 $[x,y]$ 的坐标都是相对于局部区域而言的，要在局部区域的半径内：

$$x,y \in [-radius, radius] \tag{3.20}$$

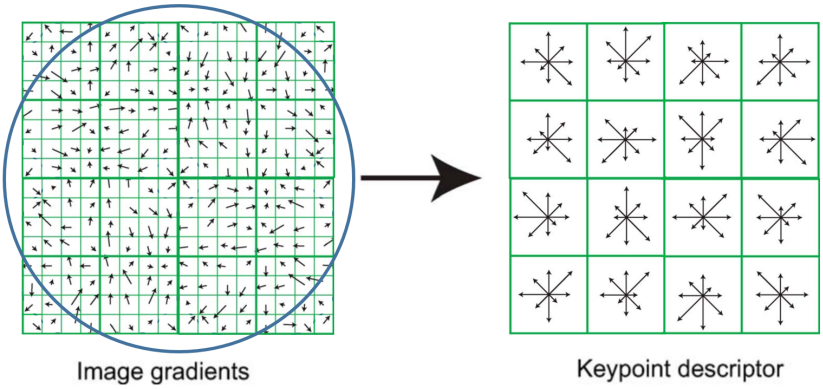
其中，radius 的单位是像素范围。

关键点局部区域图像梯度

Lowe 在论文 [2] 中，对描述子使用关键点所在尺度空间内 4×4 个子区域中计算 8 个方向的梯度信息，也就是总共 $4\times4\times8=128$ 维向量。

首先对关键点位置周围的图像梯度大小和方向进行采样，采样是在关键点所在尺度的图像上进行的。为了获得方向不变性，描述子的坐标和梯度方向相对于关键点的梯度方向进行旋转。

图像梯度大小由高斯窗口加权，在图上表示为圆形窗口。高斯加权函数 σ 等于描述子窗口宽度的一半，用于为每个采样点的幅值指定权重。该高斯窗的目的是避免在窗口位置发生微小变化的情况下描述子发生突然变化，并减少远离描述子中心的梯度的权重，因为这些梯度受匹配错误的影响最大。



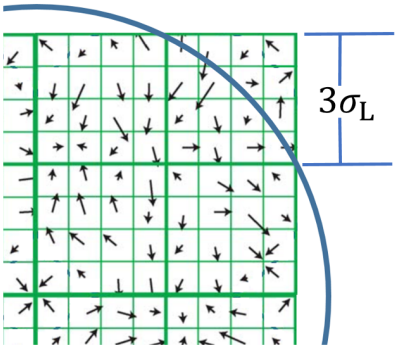
上图表示的是生成的描述子周边 4×4 的区域的示意图，这是论文 [2] 给出的建议。每个区域中的梯度方向的统计中，高斯加权是以整个大区域的中心为高斯核中心的，而不是每个区域自己的中心为高斯核中心。把每个子区域所有样本点的梯度加权求方向，就得到了上图右边的表示。

局部区域的大小

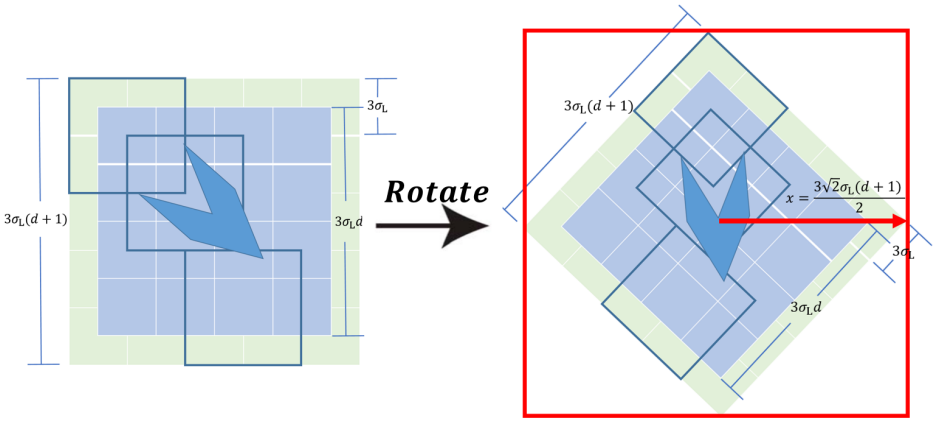
上面我们已经介绍了采样局部区域图像梯度，但是这个区域应该有多大呢？

在这里大家主要知道为什么半径的计算里有 $\sqrt{2}$，以及子区域大小是 $3\sigma_L$ 就好了，别的内容只需要简单了解，在实际代码实现过程其实与讲述的内容还是有不少区别的，而且不同的人理解也不一样，我们会在源码讲解中讲解 OpenCV 实现过程的操作流程。但为了完整性，这里还是会详细地进行一下描述，大家有所疑问也不用过于在意。

前面已经说过，Lowe 建议 4×4 个子区域。关键点的位置并不一定正好是在像素位置，因为我们上面已经讲过，对关键点需要精准定位，因此其他像素位置采样也不一定在像素上，所以需要插值。每个子区域的尺寸为 $3\sigma_L$ 个像素（注意这里的像素指的是像素宽度）：



为了描述方便，设我们划分的是 $d \times d$ 个子区域，因此整个邻域空间大小为 $3\sigma_L d$。因为需要进行双线性插值，所以总共是 $3\sigma_L(d+1)$。下图中的左图蓝色框表示的是实际在运算过程中的子区域：



也就是说该区域的半径是 $\frac{3\sigma_L(d+1)}{2}$。又由于图像需要旋转，我们以旋转中可能出现的最大边长为需要进行旋转的区域范围，即上图右边中的红线长度，也就是半径的 $\sqrt{2}$ 倍。

坐标与加权

我们将旋转后的坐标系在半径内分成 $d \times d$ 个区域，旋转后的采样点落在哪个区域，就为哪个区域的区域方向（上面叙述的 8 个方向向量）贡献加权幅值。

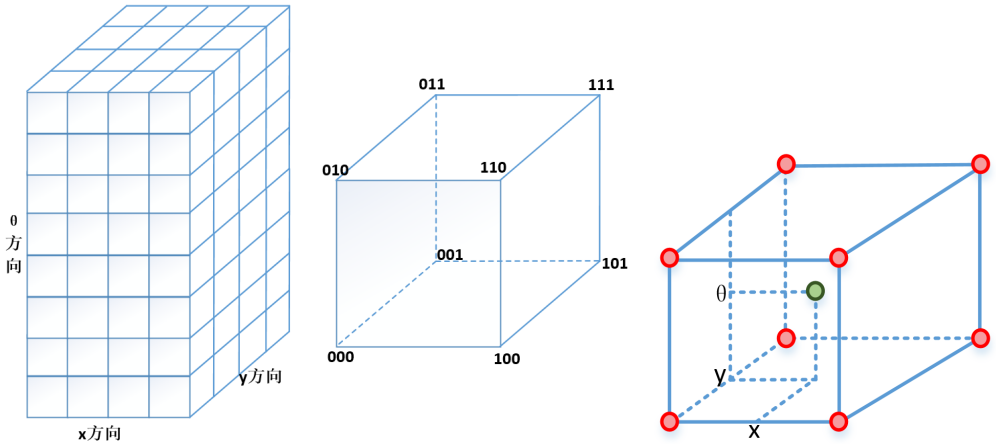
我们先介绍高斯加权系数再介绍子区域坐标。

前面说过图像梯度需要用高斯加权，这里我们说一下加权系数。Lowe 建议子区域中的像素梯度需要按照 $\sigma = 0.5d$ 的高斯系数进行加权，也就是说：

$$w = m(a+x, b+y) \cdot e^{-\frac{(x')^2+(y')^2}{2\times(0.5d)^2}} \tag{3.21}$$

其中，(a,b) 表示的是关键点的位置，(x,y) 就是局部区域点的坐标，(x',y') 是局部区域点旋转后的位置坐标，$m(a,b)$ 表示 (a,b) 点处的图像梯度。

因为旋转后的每个坐标点几乎不可能正好在像素点上，所以需要进行插值（采样点处的角度不是像方向分配时那样直接近似到最近的方向，而是按比例分配到距它最近的两个方向上（该方向两侧的两个方向，一般来说不可能正好处在上述的 8 个方向上）），我们可以将生成的 128 维向量想象成一个三维矩阵：



首先是双线性插值，每一个点对离它最近的 4 个子区域的直方图生成都会有影响；同时角度也要进行线性插值，因此可以理解为整体是三线性插值。要插值到周边的 8 个矩阵元素上并不容易，我们根据上图右边来理解一下，我们设 3 维矩阵中的八个元素分别表示右图中的红点，当前采样到的梯度像素位置为右图中的绿点（注意这里的 (x,y,θ) 只是相对位置，与前面所述的位置没有什么关系），要把绿点的数值贡献分配到周边的 8 个点上。我们会在本小节后面介绍像素区域与像素离散点之间的关系。

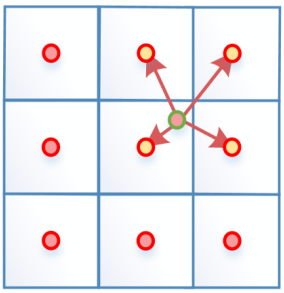
因此，现在的步骤就很明确了：计算绿点的高斯加权值，然后将加权后的值分配给周边的 8 个顶点代表的 8 个子区域直方图上。

旋转后的采样点落在 $d\times d$ 的子区域上，采样点在子区域中需要计算坐标，才能进行插值。这里的坐标计算公式为：

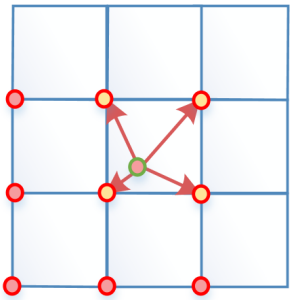
$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \frac{1}{3\sigma_L}\begin{bmatrix} x' \\ y' \end{bmatrix} + \frac{d}{2} \tag{3.22}$$

该运算将坐标计算在 $[0,0]-[d,d]$ 的范围内，之后根据其与整数坐标的差值来计算三线性插值，然后分配到周边各个点上。（这里我们忽略了插值而扩展到的 $(d+1)$ 宽度，具体情况我们会在源码中进行更详细的说明。同时我们一定要注意的是，虽然我们根据旋转的要求，需要将更大范围的像素进行旋转，但是我们生成描述向量的采样范围其实就是 $(d\times d)$ 范围，旋转更大范围只是为了让采样范围都被包含在其中。但无论怎么样，其实就是怎么方便，怎么效果好就怎么实现的，我们在实际实现过程中也可以根据我们自己的想法进行一些改进。）

在图像空间中，我们会认为一个像素代表一个区域，而实际显示的时候这个区域都是同一个值，因此我们经常会把像素的中心来代表这个像素离散点，因此二维平面上，偏离中心的位置点就要对周边 4 个像素来插值，代表其贡献，但像素中心计算比较麻烦，如下图左。因此，经常会将当前点偏移 0.5 个像素，使得该点相对于图像顶点的位置等同于该点相对于像素中心的位置，这样计算插值就会容易很多，如下图右（其实看起来就是像素中心点偏移了 0.5 的距离）。图中黄色填充点表示被贡献到的像素点。



像素区域



像素区域

因此公式中需要减去 0.5：

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \frac{1}{3\sigma_L} \begin{bmatrix} x' \\ y' \end{bmatrix} + \frac{d}{2} - 0.5 \tag{3.23}$$

而因为角度是连续值（贡献给一个范围，例如 0-45 度、45-90 度，因此不需要考虑格子的中间点位置），所以角度方向不需要偏移，我们在程序中会再进行解释。

光照不变性

现在我们已经生成了 128 维的描述向量，为了去除光照变化的影响，我们需要进行归一化处理（即，你的脸无论用什么光照射它都是你的脸）。

我们设得到的描述子向量为 $D' = [d'_1, d'_2, \ldots, d'_{128}]$，得到的归一化向量 D 为：

$$d_i = \frac{d'_i}{\sqrt{\sum_{j=1}^{128} d'_j}} \tag{3.24}$$

这样就可以消除图像因为光照变化引起的图像灰度值整体漂移。

因为光照一般是非线性的，相机饱和度以及不同位置拍摄物体的光照变化也是不一样的，因此非线性光照变化也会影响梯度值，但是它并不会影响梯度幅角（因为只是变量了而已）。所以，我们设置一个阈值 0.2，凡是 D 中小于 0.2 的元素保留其值，大于 0.2 的元素用 0.2 去取代，然后最后再对图像进行一次归一化处理，这样特征点的独特性和可区分性就再次得到进一步的提高。

现在我们就讲完 Sift 特征检测的全部内容了。但只是这样，还不能说你就完全掌握了 Sift 算法，下一章我们会进入源码，来研究实际代码中的实现过程。

4. OpenCV3 中 Sift 算法的基本流程

本章介绍 *OpenCV3* 中 *Sift* 算法的基本实现流程。我们使用的版本是 *OpenCV3.4.2*。

4.1 完整的 Sift 使用代码

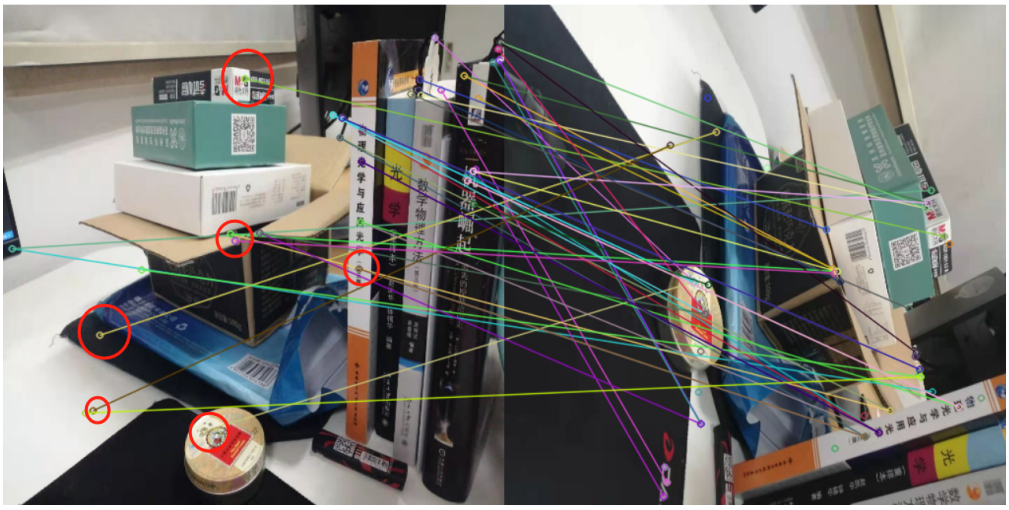
首先，我先将完整的 Sift 测试代码放在这里（OpenCV 最好编译一下 contrib，因为很多高级算法都在 contrib 里）：

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/xfeatures2d.hpp>

using std::cout;
using std::endl;
using std::vector;

using cv::Mat;
using cv::xfeatures2d::SiftFeatureDetector;
using cv::xfeatures2d::SiftDescriptorExtractor;

int main()
{
  //读入图像
  Mat srcImg1 = cv::imread("p01.jpg");
  Mat img1;
  cv::resize(srcImg1, img1, cv::Size(512, 512));
  Mat srcImg2 = cv::imread("p02.jpg");
```

```
20  Mat img2;
21  cv::resize(srcImg2, img2, cv::Size(512, 512));
22  if (img1.empty()||img2.empty())
23    return -1;
24  //检测特征点并在原图中绘制
25  int kp_number{ 50 };
26  vector<cv::KeyPoint> keypoint1, keypoint2;
27  cv::Ptr<SiftFeatureDetector> siftdtc = SiftFeatureDetector::create(
        kp_number);
28  siftdtc->detect(img1, keypoint1);
29  Mat keypointsImg;
30  cv::drawKeypoints(img1, keypoint1, keypointsImg);
31  cv::imshow("image1 keypoints", keypointsImg);
32  siftdtc->detect(img2, keypoint2);
33  Mat keypointsImg2;
34  cv::drawKeypoints(img2, keypoint2, keypointsImg2);
35  cv::imshow("image2 keypoints", keypointsImg2);
36  //打印特征点信息
37  vector<cv::KeyPoint>::iterator itvc;
38  for (itvc = keypoint1.begin(); itvc != keypoint1.end(); itvc++)
39    cout << "angle:" << itvc->angle << "\t" << itvc->class_id << "\t"
          << itvc->octave
40    << "\t" << "pt ->" << itvc->pt << "\t" << itvc->response << "\t"
          << itvc->size << endl;
41  for (itvc = keypoint2.begin(); itvc != keypoint2.end(); itvc++)
42    cout << "angle:" << itvc->angle << "\t" << itvc->class_id << "\t"
          << itvc->octave
43    << "\t" << "pt ->" << itvc->pt << "\t" << itvc->response << "\t"
          << itvc->size << endl;
44  //生成特征向量描述子
45  cv::Ptr<SiftDescriptorExtractor> extractor = SiftDescriptorExtractor
      ::create();
46  Mat descriptor1, descriptor2;
47  extractor->compute(img1, keypoint1, descriptor1);
48  extractor->compute(img2, keypoint2, descriptor2);
49  cv::imshow("descriptor1", descriptor1);
50  cout << "The size of feature matrix is: " << descriptor1.rows << "×"
       << descriptor1.cols << endl;
51  //图像特征匹配
```

```
52    cv::Ptr<cv::DescriptorMatcher> matcher = cv::DescriptorMatcher::
          create("BruteForce");
53    vector<cv::DMatch> matches;
54    Mat imgMatches;
55    matcher->match(descriptor1, descriptor2, matches);
56    cv::drawMatches(img1, keypoint1, img2, keypoint2, matches,
          imgMatches);
57    imshow("matches", imgMatches);
58    cv::waitKey(0);
59    return 0;
60 }
```

我们找两个包含相同场景的图像测试一下，得到如下匹配结果，可以看到里面有很多被正确匹配的特征点，我挑了几个感觉匹配的非常完美的点：



接下来我会介绍图像匹配的程序过程。

4.2 OpenCV3 中 Sift 算法图像匹配过程

特征检测的主要类是 SiftFeatureDetector，该类其实就是 SIFT 类，SIFT 类的基类是 Feature2D，表示的是二维图像特征类。我们创建该类时会调用 create 函数来返回智能指针。该函数的参数我们前面都介绍过：

```
1  int nfeatures = 0 //特征点数，如果为0表示输出全部特征点
2  int nOctaveLayers = 3 //每组要检测出的S个尺度的极值点
3  double contrastThreshold = 0.04 //前面介绍过的关键点排查阈值
4  double edgeThreshold = 10 //消除边缘响应r值
5  double sigma = 1.6 //初始模糊尺度
```

之后就是利用 detect 函数来检测特征点，输入参数是图像和关键点 vector，之后程序会将最大响应的关键点存储在该数组里。

生成描述子使用的是 SiftDescriptorExtractor 类。该类其实和 SiftFeatureDetector 类是完全一样的：

```
typedef SIFT SiftFeatureDetector;
typedef SIFT SiftDescriptorExtractor;
```

因此，其实我们如果直接使用 siftdtc 来检测也是完全可以的：

```
//下面的代码
siftdtc->compute(img1, keypoint1, descriptor1);
//等同于
extractor->compute(img1, keypoint1, descriptor1);
```

但为了描述得更具有步骤性（或者是为了兼容老版本 OpenCV 的使用方法），这里的做法就是依据不同的别名来定义使用不同功能的对象。

再之后就是将描述子输入到 DescriptorMatcher 对象中，然后使用 match 函数进行匹配，将匹配结果保存在数组 imgMatches 中。DescriptorMatcher 的构造也是使用 create 函数，该函数接受一个字符串参数，这里输入的是 BruteForce，表示匹配的方法是“暴力匹配”，也就是一个一个进行对比，找到其中最适合的匹配点。

因为我们还没有介绍特征匹配，我们现在只是研究 Sift 特征检测，因此我们的主要精力是放在 SIFT 类上。

4.3 SIFT 类

SIFT 类定义在 nonfree.hpp 头文件中，定义如下：

```
class CV_EXPORTS_W SIFT : public Feature2D
```

该类只新定义了一个函数，即 create，其他功能都是继承和覆盖父类 Feature2D 的函数。

Feature2D 定义在 features2d.hpp 头文件里，该类分别有两个重载的 detect 函数和两个重载的 compute 函数，其中第一组检测和计算函数如下，它输入一个图像以及一个特征点 vector，用于检测一张图像的特征点：

```
CV_WRAP virtual void detect( InputArray image, CV_OUT std::vector<
    KeyPoint>& keypoints, InputArray mask=noArray() );
CV_WRAP virtual void compute( InputArray image, CV_OUT CV_IN_OUT std::
    vector<KeyPoint>& keypoints, OutputArray descriptors );
```

第二组检测和计算函数如下，它输入一个图像 vector 以及一个特征点 vector 的 vector，用于检测多张图像的特征点：

```
1 CV_WRAP virtual void detect( InputArrayOfArrays images, CV_OUT std::
    vector<std::vector<KeyPoint> >& keypoints, InputArrayOfArrays
    masks=noArray() );
2 CV_WRAP virtual void compute( InputArrayOfArrays images, CV_OUT
    CV_IN_OUT std::vector<std::vector<KeyPoint> >& keypoints,
    OutputArrayOfArrays descriptors );
```

但是 SIFT 中并没有继承和实现 detect 函数和 compute 函数，这些函数都会调用 detectAndCompute 函数，SIFT_Impl 继承自 SIFT 类，实现了该函数。该函数的最后一个参数是 bool 类型的参数，表示是否使用提供的关键点，如果是 false，就表示不用提供的关键点，需要重新生成关键点，因此 Feature2D::detect 调用时该参数就是 false；而 Feature2D::compute 调用时该参数是就是 true。detectAndCompute 的倒数第二个参数是生成的描述子数组，因此对于 detect 函数而言，该参数传入的是 noArray()，即我们不需要生成描述子；而对于 compute 函数而言，则是需要计算生成描述子的。

其他函数主要就是一些读写操作了，不是主要的算法。

我们最后看一下 KeyPoint 类的定义，该类定义在 types.hpp 文件中。我们根据调试信息就可以看出该类里面的变量所代表的意义。

class_id 表示当要对图片进行分类时，可以用 class_id 进行区分，未设定时为默认值-1，需要该变量时则根据图像类别去自己设定。

octave 表示是从金字塔中提取到该特征点的组和层，我们如果输出就会发现这个值一般会特别大，这是一种表示格式，需要拆分到不同的项中（也就是说它虽然是 int 类型的数据，但其实是按位表示的），我们会在下一章进行解释。

学完本章，您应该已经知道如何去使用 OpenCV3 的 SIFT 模块了，下一章我们就会逐步讲解算法中的每一步原理，其实就是 detectAndCompute 函数，而该函数其实也调用了很多其他的成员函数，所以其实是一个代码行很多的函数，讲解起来也没有那么容易，因此希望大家能够认真去学习，尤其是要对照着本书和源码一遍一遍看，这样相信您很快就能攻克 SIFT 代码的！

5. SIFT 源码详解

本章讲解 *SIFT* 算法的实现细节，主要就是 *detectAndCompute* 函数的实现细节。对于前三章中您如果对其中某些部分还留有疑问，那么实际代码一定会给您一个答案。相信本章过后，您也能自己去实现一个详细的 *SIFT* 程序了。

5.1 detectAndCompute 函数的大致流程

该函数的输入参数中，mask 表示寻找特征点位置的掩模，即它是一个和图像同样大小的矩阵，其中非 0 的像素处表示可以用来检测特征点的位置。该矩阵的作用就是，比如我们只想检测某个区域的特征点，我们就可以设置这个区域的掩模值不为 0 而其他区域值为 0。

firstOctave 表示金字塔的组索引是从 -1 还是 0 开始，从 -1 开始则输入图像长宽需要扩大一倍，actualNOctaves 和 actualNLayers 分别表示实际的高斯图像金字塔的组数和每组层数。

然后判断图像和掩模的格式是否正确。注意 depth() 函数表示深度，这里读出来的深度都是 0（注意和通道数 channels 不一样）。

之后，我们写一下程序流程：

```
1  if(需要使用提供的关键点){
2      （1）提取关键点信息，对firstOctave、actualNOctaves进行新的赋值
3  }
4   （2）创建高斯金字塔和DOG图像金字塔。
5  if(不需要使用提供的关键点){
6      （3）进行关键点检测
7  }
8  if(需要提取关键点描述子){
9      （4）计算生成描述子
10 }
```

我们为了方便描述，把上面的流程分为 4 个步骤。对于 detect 函数，其实执行流程是：

```
（2）创建高斯金字塔和DOG图像金字塔。
（3）进行关键点检测。
```

而对于已经有输入关键点数组的 compute 函数来说，其实执行流程是：

```
（1）提取关键点信息，对firstOctave、actualNOctaves进行新的赋值。
（2）创建高斯金字塔和DOG图像金字塔。
（4）计算生成描述子。
```

我们本节将里面内容最少的步骤 (1) 讲解完，然后本章其他章节会讲解完整个 SIFT 实现流程。

(1) 其实就是计算组和层数，我们其实完全可以把这些内容复制到主程序里，然后打印，为了不冲突，这里给函数名后加了“_1”：

```
static inline void unpackOctave_1(const cv::KeyPoint& kpt, int& octave
    , int& layer, float& scale){
  octave = kpt.octave & 255;
  layer = (kpt.octave >> 8) & 255;
  octave = octave < 128 ? octave : (-128 | octave);
  scale = octave >= 0 ? 1.f / (1 << octave) : (float)(1 << -octave);
}
//main函数中，在检测完关键点之后，可以使用下面的程序来输出一下关键点的
    信息：
int firstOctave = -1, actualNOctaves = 0, actualNLayers = 0;
firstOctave = 0;
int maxOctave = INT_MIN;
for (size_t i = 0; i < keypoint1.size(); i++)
{
  int octave, layer;
  float scale;
  unpackOctave_1(keypoint1[i], octave, layer, scale);
  firstOctave = std::min(firstOctave, octave);
  maxOctave = std::max(maxOctave, octave);
  actualNLayers = std::max(actualNLayers, layer - 2);
}
firstOctave = std::min(firstOctave, 0);
actualNOctaves = maxOctave - firstOctave + 1;
```

我们可以打印一下 firstOctave 和 maxOctave，包括每个关键点的 scale 和所在 octave 以及 layer，比如我们检测的特征点信息中，firstOctave 是-1，maxOctave 是 4，因此一共有 6 组。unpackOctave 就是把 octave 按位拆解开。octave 的最低的 8 位表示组，后面 8 位表示层。组数需

要判断是否是正数还是负数，如果是正数则小于 128，否则就大于等于 128，则通过 (-128|octave) 就能得到负数值，比如 (-128|255) 就是-1（注意-128 的十六进制表示为 ffffff80）。

scale 的计算可以看出，当组数为-1，得到 2；组数为 0，得到 1；组数为 1，得到 $\frac{1}{2}$；以此类推。还记得我们之前讲过，第 o 组的第一层图像模糊系数为 $2^o\sigma_0$，这里用 2^o 来表示尺度。scale 的计算式为 2^{-o}，该值在后面会用于将图像在整个金字塔的尺度 $\sigma_0 2^{o+\frac{s}{S}}$ 转换为图像在当前组的尺度 $\sigma_0 2^{\frac{s}{S}}$ （除以 2^o 耗费的时间远大于乘以 2^{-o}）。

接下来，本章的内容就是分别介绍 (2) 创建高斯金字塔和 DOG 图像金字塔；(3) 进行关键点检测；以及 (4) 计算生成描述子，其中这些功能是由下面的几个函数构成的：

- createInitialImage：创建基图像。
- buildGaussianPyramid：构建高斯金字塔。
- buildDoGPyramid：构建 DoG 图像金字塔。
- findScaleSpaceExtrema：在 DoG 尺度空间内找到极值点。
- adjustLocalExtrema：定位精准特征点。
- calcOrientationHist：计算特征点的方向角度。
- calcDescriptors：计算特征点描述子。
- calcSIFTDescriptor：计算特征点的特征矢量。

5.2 创建图像金字塔

创建高斯金字塔的过程用到了下面三个函数：

- createInitialImage：创建基图像。
- buildGaussianPyramid：构建高斯金字塔。
- buildDoGPyramid：构建 DoG 图像金字塔。

createInitialImage

该函数在 sift.cpp 文件里。

首先判断如果是彩图，就先转为灰度图。然后调用 convertTo 函数来根据 scale 去调整图像的像素数据类型。Mat 类型数据的深度和通道数不一定满足运算的要求，因此使用 convertTo() 函数，负责转换成目标数据类型的 Mat，例如这里转换为 float 类型：

```
1 typedef float sift_wt;
2 static const int SIFT_FIXPT_SCALE = 1;
3 //convertTo函数原型:
4 void convertTo( OutputArray m, int rtype, double alpha=1, double
    beta=0 ) const;
```

这里的 alpha 就是 scale，beta 表示偏移。即我们调用该函数后，第 i 个像素值 $I(i)$ 就会变为 $scale \times I(i) + beta$。

然后就是尺度计算了，SIFT_INIT_SIGMA 表示初始图像尺度，参考“关于第一层的模糊”小节。该函数根据 doubleImageSize 参数来判断是否要扩增图像，而该参数传入时其实是“firstOctave < 0”，也就是说，当用来检测特征点时，firstOctave 被赋初值-1 以后就没有再被修改过了，因此为 true；而当用来计算描述子时，前面描述的 useProvidedKeypoints 为 true，可能会对 firstOctave 进行修改（当然，对于我们的程序生成的特征点，第一层的索引就是-1，因此结果也是 doubleImageSize）。大家可以返回到 detectAndCompute 函数来看一下程序流程。

```
1  if( doubleImageSize ) {
2      sig_diff = sqrtf( std::max(sigma * sigma - SIFT_INIT_SIGMA *
           SIFT_INIT_SIGMA * 4, 0.01f) );
3      Mat dbl;
4      GaussianBlur(dbl, dbl, Size(), sig_diff, sig_diff);
5      return dbl;
6  }
7  else {
8      sig_diff = sqrtf( std::max(sigma * sigma - SIFT_INIT_SIGMA *
           SIFT_INIT_SIGMA, 0.01f) );
9      GaussianBlur(gray_fpt, gray_fpt, Size(), sig_diff, sig_diff);
10     return gray_fpt;
11 }
```

当 doubleImageSize 为 true 时，我们使用的是方法二（在前面“关于第一层的模糊”中介绍的两种方法），即使用双线性插值将图像长宽扩大两倍，sig_diff 值的计算也可以参考方法二，SIFT_INIT_SIGMA×2=1.0，即图像扩大 2 倍以后的尺度。之后再用尺度值来进行高斯模糊。

当 doubleImageSize 为 false 时，我们使用的是方法一，即不扩大图像，而是用 1.52 的高斯方差去平滑我们的原图像来得到第一组第一层的图像（不过程序里默认都是扩大两倍再进行计算）。

调用完 createInitialImage 函数以后，detectAndCompute 函数里会计算 nOctaves 值，如果该值等于 0，则说明没有被赋值过，因此需要根据要生成的高斯金字塔计算组数，计算方法如下：

```
1 int nOctaves = actualNOctaves > 0 ? actualNOctaves : cvRound(std::log(
    (double)std::min( base.cols, base.rows ) ) / std::log(2.) - 2) -
  firstOctave;
```

即选择图像长宽中的短边，然后根据如下公式来计算生成的高斯金字塔组数。之所以要减 2 是因为如果图像只有几个像素的话就没有什么意义了，所以防止降采样中得到过小的图像：

$$\left(\log_2 \min(Width, Height)\right) - 2 \tag{5.1}$$

$$\log_2 A = \frac{\ln(A)}{ln(2)}, \quad A = \min(Width, Height) \tag{5.2}$$

之后就进入下一个环节：buildGaussianPyramid。

buildGaussianPyramid

该函数的输入参数是基图像 base，高斯金字塔图像数组 gpyr 以及刚刚计算的图像组数 nOctaves。

sig 数组表示的是每组中各层图像的方差，我想大家应该已经对 +3 很熟悉了。之后对 pyr 数组进行初始化，大小为图像组数乘以每组层数。

```
1  std::vector<double> sig(nOctaveLayers + 3);
2  pyr.resize(nOctaves*(nOctaveLayers + 3));
3  // precompute Gaussian sigmas using the following formula:
4  //  \sigma_{total}^2 = \sigma_{i}^2 + \sigma_{i-1}^2
5  sig[0] = sigma;
```

之后程序的注释说明已经很明确，使用公式计算每组的层与层之间的模糊系数（因为我们是用一层图像模糊得到下一层）。sig[0] 赋值为 sigma，该值默认为 1.6。然后计算 k 值，k 值为 $2^{\frac{1}{s}}$，之后就是把第一组图像的所有层的高斯模糊系数都计算出来。

```
1  double k = std::pow( 2., 1. / nOctaveLayers );
2  for( int i = 1; i < nOctaveLayers + 3; i++ ) {
3      double sig_prev = std::pow(k, (double)(i-1))*sigma;
4      double sig_total = sig_prev*k;
5      sig[i] = std::sqrt(sig_total*sig_total - sig_prev*sig_prev);
6  }
```

在不同的组中，例如第一组的第一层到第二层的模糊系数以及第二组的第一层到第二层的模糊系数是一样的（参见“构建每组内的图像”小节）。

然后就是根据模糊系数把每层图像进行模糊了，注意上一组的第 nOctaveLayers 层图像用来缩小两倍后作为下一组的第一层图像。过程很简单，这里就不再放上源码了。

buildDoGPyramid

该过程非常简单，就是在每组中用第 i+1 层减去第 i 层的图像，就得到了 DOG 图像。该过程是一个并行处理的函数，buildDoGPyramidComputer 类继承自 ParallelLoopBody，然后会将 operator() 实现的内容并行处理。这个并行结构比较简单，我们会在后面详细介绍一下并行机制。

相信大家在掌握了这部分源码以后，已经对 SIFT 中构建 DOG 金字塔非常熟悉了。构建图像金字塔部分相比于后面其实是非常简单的，后面的关键点检测部分则会较难，尽管程序逻辑并不复杂，但是需要一定的数学知识。

5.3 进行关键点检测

进行关键点检测的过程用到了下面三个函数：

- findScaleSpaceExtrema：在 DoG 尺度空间内找到极值点。

- adjustLocalExtrema：定位精准特征点。
- calcOrientationHist：计算特征点的方向角度。

findScaleSpaceExtrema

这个函数比前面讲过的函数都要长，大家要做好心理准备。

首先计算得到金字塔组数 nOctaves。

然后设定一个阈值 threshold，用来判断在 DOG 尺度图像中像素值的绝对值是否足够大，我们后面遇到时再详细讲解它的意义：

```
1  //SIFT_FIXPT_SCALE=1, contrastThreshold默认为0.04, nOctaveLayers默认
       为3
2  const int threshold = cvFloor(0.5 * contrastThreshold /
       nOctaveLayers * 255 * SIFT_FIXPT_SCALE);
```

之后清空关键点数组（需要一个干净的容器），以及定义一个关键点存储 TLSData 结构，该结构我们待会再介绍，其实也是一个容器。

```
1  keypoints.clear();
2  TLSData<std::vector<KeyPoint> > tls_kpts_struct;
```

之后就是两层 for 循环，用来处理每个组以及每组中的层之间 DOG 图像，来获得关键点

```
1  for( int o = 0; o < nOctaves; o++ )
2    for( int i = 1; i <= nOctaveLayers; i++ )
3    {
4        //每组有nOctaveLayers+2层 DOG 图像。
5        const int idx = o*(nOctaveLayers+2)+i;
6        const Mat& img = dog_pyr[idx];
7        //函数原型：size_t step1(int i=0) const; 这里可以理解为每行图像
             的字节数，比如512X512的三通道RGB图像，这里就是512X3=1536。
8        const int step = (int)img.step1();
9        const int rows = img.rows, cols = img.cols;
10       //并行程序计算关键点信息
11       ......
12   }
```

获取到关键点后，再将关键点保存的结构放入 kpt_vecs 数组中，然后再加入到 keypoints 数组中：

```
1  std::vector<std::vector<KeyPoint>*> kpt_vecs;
2  tls_kpts_struct.gather(kpt_vecs);
3  for (size_t i = 0; i < kpt_vecs.size(); ++i) {
```

```
4     keypoints.insert(keypoints.end(), kpt_vecs[i]->begin(), kpt_vecs[i
          ]->end());
5 }
```

在讲解并行程序计算关键点之前，我们还有两个遗留问题，即 TLSData 结构和上面的 gather 函数。我单独列在了一个小的版块里，因为下面的内容就算是不掌握也完全没有问题，但为了完整性我还是打算简单介绍一下。

补充 5.1 (TLSData 结构和 gather 函数) TLSData（Thread Local Storage Data）借助 TlsStorage 来存储内存信息，将某些数据和某个特定线程关联起来。
TLS 技术可以保证各个线程间数据安全，使得每个线程的数据不会被另外线程访问和破坏。由于 OpenCV3 是使用并行方法来加速关键点搜索，因此每个关键点存储放在一个局部线程里是有必要的。在 OpenCV 中使用也不难，OpenCV 封装了自己的 TLS 操作方法，我们只需要在多线程程序里使用：TLSData<T>::get() 函数就能申请到一块线程局部内存，之后每个线程的内存都会被该 TLSData 对象进行管理。并行程序结束以后，通过使用 TLSData<T>::gather() 函数就能把它们整合到一个内存区。

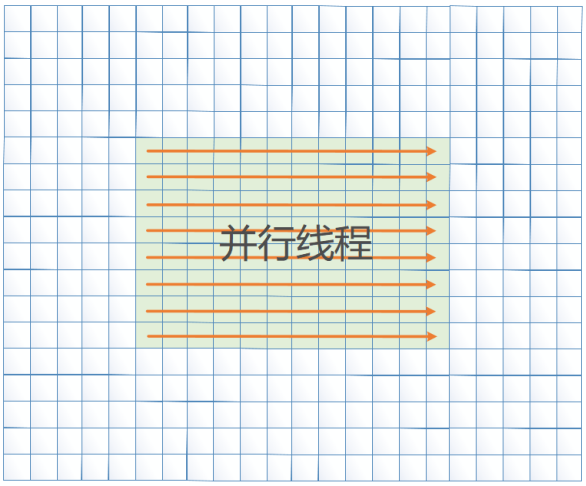
最后，我们就介绍算法的重点，即并行程序：

```
1  //SIFT_IMG_BORDER默认为5，表示边缘区域5个像素范围不被用来检测关键
       点。
2  parallel_for_(Range(SIFT_IMG_BORDER, rows-SIFT_IMG_BORDER),
3  findScaleSpaceExtremaComputer(
4      o, i, threshold, idx, step, cols,
5      nOctaveLayers,
6      contrastThreshold,
7      edgeThreshold,
8      sigma,
9      gauss_pyr, dog_pyr, tls_kpts_struct));
```

长宽上下 5 个像素内不用来检测关键点，这里的范围表示行像素的检测范围（并行程序为需要检测的每一行都开一个线程），列像素的限定范围定义在了要执行的并行程序内部，我们后面会遇到：

```
1  for( int c = SIFT_IMG_BORDER; c < cols-SIFT_IMG_BORDER; c++)
```

如下示意图，橘色箭头表示开启的每一个线程：



补充 5.2 (TLS 的好处) 上面的程序里，不同的线程都是为了计算一张 DOG 图像上的关键点位置，如果保存到同一个内存区，就得用互斥锁来防止数据存储冲突，这样会降低效率。因此使用 TLS 管理每个线程的局部内存区则会加快速度（相应地，也会增加内存消耗，因为需要提前安排可用内存）。

补充 5.3 (OpenCV 中的行和列) OpenCV 中，rows 代表图像有多少行，cols 代表图像有多少列。
但是有时候访问顺序会不一样，比如 cv::size(cols, rows) 是先描述列再描述行。
而访问某个像素的 at 函数，例如 Image.at<Vec3b>(i, j)[0]，i 表示第几行，j 表示第几列。

parallel_for_ 函数会调用 findScaleSpaceExtremaComputer 的 operator() 运算符重载，该运算符里面就是检测一行像素中的关键点的程序。findScaleSpaceExtremaComputer 继承自 ParallelLoopBody，覆盖了其中的 operator() 函数，覆盖后，只要实现下面的并行加速内容，就可以对其进行加速：

```
1  virtual void operator ()(const Range& range) const {
2      for (int r = range.start; r < range.end; r++){
3        //并行加速内容
4        ......
5      }
6  }
```

findScaleSpaceExtremaComputer 构造函数的传入参数中，sigma 是初始图像模糊系数，默认为 1.6，用来调整极值点位置，我们用到再讲解。其继承的 operator() 函数首先定义了一些数据：

```
1  //进行并行加速的范围
2  const int begin = range.start;
3  const int end = range.end;
4  //存储方向的桶，一共36个方向
```

```
5   static const int n = SIFT_ORI_HIST_BINS;
6   float hist[n];
7   //本层和上下两层的DOG图像
8   const Mat& img = dog_pyr[idx];
9   const Mat& prev = dog_pyr[idx-1];
10  const Mat& next = dog_pyr[idx+1];
11  //获得局部线程内存存储，防止内存访问冲突，将保存到改存储中
12  std::vector<KeyPoint> *tls_kpts = tls_kpts_struct.get();
13  //定义一个关键点对象
14  KeyPoint kpt;
```

之后开始执行并行程序大循环，代码如下：

```
1  for( int r = begin; r < end; r++) {
2      //分别索引到上中下三层DOG图像的当前行图像（注意这里的内容都会被并
           行执行，每一行r都是一个线程）
3      const sift_wt* currptr = img.ptr<sift_wt>(r);
4      const sift_wt* prevptr = prev.ptr<sift_wt>(r);
5      const sift_wt* nextptr = next.ptr<sift_wt>(r);
6      //对每行图像的每一列进行检测
7      for( int c = SIFT_IMG_BORDER; c < cols-SIFT_IMG_BORDER; c++){
8        //当前像素值
9          sift_wt val = currptr[c];
10         //检测极值
11         ......
12     }
13 }
```

其中 sift_wt 是 SIFT 的数据类型，一般就是 float 型的数据（float 类型的灰度值）。

然后就是检测极值点了，这是一个相当庞大的 if 判断，如果检测为真，说明是极值点，然后就执行 if 里面的内容，对极值点进行精准定位和计算朝向，否则就说明它不是极值点，则继续检测下一个像素。

这里的 if 结构其实很简单，如果是局部 $3\times3\times3$ 像素内最亮的点或者局部最暗的点，说明就是极值点（这里注意 DOG 图像因为是相减得到的，其中像素值可以为负数）：

```
1  if( val绝对值大于threshold && ((val是局部最亮的点) || (val是局部最暗的
       点)) ){
2    //是关键点
3    ......
4  }
```

我们本节还剩最后一个内容没有详细解释，就是这里的阈值 threshold。讲解完阈值以后，关键点检测的内容就讲解完了，后面的章节再介绍精准定位和方向计算。我们已经知道，当前像素值的绝对值必须要大于这个阈值，才能有可能被当做关键点，这个阈值的计算为：

$$threshold = Floor\left(\frac{0.5 * contrastThreshold * 255 * SIFT\_FIXPT\_SCALE}{nOctaveLayers}\right) \tag{5.3}$$

$$= Floor\left(\frac{0.5 * 0.04 * 255}{3}\right) \tag{5.4}$$

因为我们之前在将图像转换为 float 类型灰度图时的设置的数据缩放比例为 SIFT_FIXPT_SCALE，也就是 1，换句话说，灰度值为 255 时直接就转换为了 255.0，因此上述操作就相当于将灰度值小于“像素可能的最大值再乘以 $\frac{0.5*0.04}{S}$”的点都去掉。因为灰度图的像素范围是 $[0 \sim 255]$，DOG 图像的图像范围就是 $[-255 \sim 255]$ 之间。因此上述操作将像素值的绝对值小于 $\frac{5.1}{3}$ 的像素去除，不作为关键点。

当我们检测到关键点时，我们要通过插值调整其位置，然后再计算朝向，这是接下来的两节的内容。本节内容虽然有点多，但并不需要多少数学知识，程序结构也比较清晰，下一节相对较难，涉及到一定量的数学原理。

adjustLocalExtrema

该函数的输入参数中，octv 和 layer 为极值点所在的组和所在组内的层。r 和 c 表示为 row 和 column，为极值点的位置坐标。

关于多元函数的泰勒展开，为了更清晰的叙述，我们再介绍一下。在 $\boldsymbol{x_0}$ 附近的函数泰勒展开可以写为：

$$f(\boldsymbol{x}) = f(\boldsymbol{x_0}) + \frac{\partial D^T}{\partial \boldsymbol{x}}(\boldsymbol{x}-\boldsymbol{x_0}) + \frac{1}{2}(\boldsymbol{x}-\boldsymbol{x_0})^T \frac{\partial^2 D}{\partial \boldsymbol{x}^2}(\boldsymbol{x}-\boldsymbol{x_0}) \tag{5.5}$$

$$= f\left(\begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right) + \begin{bmatrix} \frac{\partial D}{\partial x} & \frac{\partial D}{\partial y} & \frac{\partial D}{\partial \sigma} \end{bmatrix}\left(\begin{bmatrix} x \\ y \\ \sigma \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right) \tag{5.6}$$

$$+ \frac{1}{2}\left(\begin{bmatrix} x & y & \sigma \end{bmatrix} - \begin{bmatrix} x_0 & y_0 & \sigma_0 \end{bmatrix}\right)\begin{bmatrix} \frac{\partial^2 D}{\partial x \partial x} & \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial x \partial \sigma} \\ \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial y \partial y} & \frac{\partial^2 D}{\partial y \partial \sigma} \\ \frac{\partial^2 D}{\partial x \partial \sigma} & \frac{\partial^2 D}{\partial y \partial \sigma} & \frac{\partial^2 D}{\partial \sigma \partial \sigma} \end{bmatrix}\left(\begin{bmatrix} x \\ y \\ \sigma \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right) \tag{5.7}$$

补充 5.4 (正确的偏导表示方法) 补充一下，在论文 [2] 以及网上的表示方法都是这么表示的：$\frac{\partial D^T}{\partial \boldsymbol{x}}$，但我们一定要注意，其实应该表示为 $\left(\frac{\partial D}{\partial \boldsymbol{x}}\right)^T$，因为这其实是求导然后再取转置（本质上是一个函数组，取转置是为了对应矩阵运算），因此 $\left(\frac{\partial D}{\partial \boldsymbol{x}}\right)^T(\boldsymbol{x_0})$ 就是把 $\boldsymbol{x_0}$ 代入到该函数组里得到的向量。在后面的表示中，为了兼顾不同的表示方法，不对 $\frac{\partial D^T}{\partial \boldsymbol{x}}$ 和 $\left(\frac{\partial D}{\partial \boldsymbol{x}}\right)^T$ 再加以区分。

我们重点是要研究偏移量，即 $\boldsymbol{x}-\boldsymbol{x_0}$，因此：

$$f(\boldsymbol{x}) = f(\boldsymbol{x_0}) + \left(\frac{\partial D}{\partial \boldsymbol{x}}\right)^T(\hat{\boldsymbol{x}}) + \frac{1}{2}(\hat{\boldsymbol{x}})^T \frac{\partial^2 D}{\partial \boldsymbol{x}^2}(\hat{\boldsymbol{x}}) \tag{5.8}$$

这里的 D 函数对向量求导比矩阵对向量（见《函数对向量求导-通俗易懂的描述》）求导要容易，因为矩阵相当于一个函数组。求导得到：

$$\frac{\partial f(\hat{\boldsymbol{x}})}{\partial \hat{\boldsymbol{x}}} = \left(\frac{\partial D}{\partial \boldsymbol{x}}\right)^T + \frac{1}{2}\left(\frac{\partial^2 D}{\partial \boldsymbol{x}^2} + \left(\frac{\partial^2 D}{\partial \boldsymbol{x}^2}\right)^T\right)\hat{\boldsymbol{x}} \tag{5.9}$$

由于上面描述的二阶海森矩阵，转置等于其本身，因此上式可以化为：

$$\frac{\partial D(\hat{\boldsymbol{x}})}{\partial \hat{\boldsymbol{x}}} = \left(\frac{\partial D}{\partial \boldsymbol{x}}\right)^T + \frac{\partial^2 D}{\partial \boldsymbol{x}^2}\hat{\boldsymbol{x}} \tag{5.10}$$

补充 5.5 (为什么对向量的二阶偏导是矩阵) 对于二阶海森矩阵，我们可以理解为，$\frac{\partial D}{\partial \boldsymbol{x}}$ 是一个向量，而对于变量 $\boldsymbol{x}$ 它是一个函数组，因此函数组再对向量 $\boldsymbol{x}$ 求导就变成了一个矩阵。

求偏导需要用到有限差分方法，大家可以参考 DezemingFamily 的《有限差分方法近似偏导》。对于二元函数，一阶偏导数表示为：

$$\frac{\partial D}{\partial x} = \frac{D(i,j+1) - D(i,j-1)}{2h} \tag{5.11}$$

$$\frac{\partial D}{\partial y} = \frac{D(i+1,j) - D(i-1,j)}{2h} \tag{5.12}$$

二阶偏导和二阶混合偏导可以表示为：

$$\frac{\partial^2 D}{\partial x^2} = \frac{\big(D(i,j+1) - D(i,j)\big) - \big(D(i,j) - D(i,j-1)\big)}{h^2} \tag{5.13}$$

$$\frac{\partial^2 D}{\partial y^2} = \frac{\big(D(i+1,j) - D(i,j)\big) - \big(D(i,j) - D(i-1,j)\big)}{h^2} \tag{5.14}$$

$$\frac{\partial^2 D}{\partial x \partial y} = \frac{D(i+1,j+1) + D(i-1,j-1) - D(i-1,j+1) - D(i+1,j-1)}{4h^2} \tag{5.15}$$

由于像素间距间隔为 1，因此上面的 $h = 1$ 。

在实际插值过程中，向量 $\boldsymbol{x}$ 其实是一个三维向量，分量为 $[x, y, layer]$，即在不同层之间拟合。但类似上面的二维向量，只是多了一维而已，可以类推过去（下面讲到程序就非常清楚了）。

现在开始讲解程序。函数主体为：

```
1  //img_scale相当于对图像进行归一化，将DOG数据压缩到[-1,1]之间
2  const float img_scale = 1.f/(255*SIFT_FIXPT_SCALE);
3  //一阶偏导系数（2h）
4  const float deriv_scale = img_scale*0.5f;
5  //二阶偏导系数（h^2)
6  const float second_deriv_scale = img_scale;
7  //二阶混合偏导系数（4h^2）
8  const float cross_deriv_scale = img_scale*0.25f;
9  //插值的层偏移量、行和列偏移量以及关键点响应值
10 float xi=0, xr=0, xc=0, contr=0;
11 //反复迭代5次计算插值
```

```
12  int i = 0;
13  for( ; i < SIFT_MAX_INTERP_STEPS; i++ ){
14     ......
15  }
16  //迭代次数超过5次，则判定为未找到准确特征点
17  if( i >= SIFT_MAX_INTERP_STEPS )
18  return false;
19  //对特征点的响应信息和稳定性进行判定
20  {......}
21  //保存特征点信息，因为不同的组会将图像降采样，所以关键点位置需要再乘
       以当前组数，得到在原图尺寸中的位置。
22  kpt.pt.x = (c + xc) * (1 << octv);
23  kpt.pt.y = (r + xr) * (1 << octv);
24  //按格式保存特征点所在的组、层以及插值后的层的偏移量
25  kpt.octave = octv + (layer << 8) + (cvRound((xi + 0.5)*255) << 16);
26  //特征点相对于输入图像的尺度，因为我们设置的输入图像是实际输入图像扩
       大一倍以后的图像，因此需要乘以2
27  kpt.size = sigma*powf(2.f, (layer + xi) / nOctaveLayers)*(1 << octv)
       *2;
28  kpt.response = std::abs(contr);
29  return true;
```

其中，kpt.octave 的前 8 位保存组序号，中间 8 位保存层序号；cvRound 是四舍五入整数值，这里为了将 [-0.5,0.5] 之间的数据保存为非浮点数（因为不需要那么精确），因此这里的偏移间隔就是 $1.0f/255.0f$。

kpt.size 的计算其实就是：

$$\sigma_0 2^{o+\frac{r}{s}} \qquad o \in [0,1,...,O-1], \quad r \in [0,1,...,s+2] \tag{5.16}$$

由于加入了第 0 组，因此 o 的范围就变为了 $o \in [-1,0,...,O-1]$。但是因为我们假定初始图像是实际输入图像扩大了一倍以后的结果，因此相比于初始图像的尺度要再乘以 2。该尺度用于计算生成关键点的局部方向的局部区域半径（注意这不是描述图像在当前组该层的尺度 σ_L，而是图像在整个图像金字塔的尺度，见前文“关键点方向分配”一节）。

现在我们还剩最后两个代码块，也就是上面代码中的省略号部分。

关键点精准定位

在第一个代码块中，首先计算当前层索引，然后索引上下两个层：

```
1  int idx = octv*(nOctaveLayers+2) + layer;
2  const Mat& img = dog_pyr[idx];
3  const Mat& prev = dog_pyr[idx-1];
```

```
4   const Mat& next = dog_pyr[idx+1];
```

然后计算 dD，也就是 $\frac{\partial D}{\partial x}$。

然后计算二阶偏导 dxx，dyy，dss，dxy，dxs，dys。其中 dss 表示层之间的偏导，之后生成海森矩阵 H。令导数为 0：

$$\frac{\partial D(\hat{x})}{\partial \hat{x}} = \frac{\partial D^T}{\partial x} + \frac{\partial^2 D}{\partial x^2}\hat{x} = 0 \tag{5.17}$$

其中，$\frac{\partial D^T}{\partial x}$ 就是 dD，$\frac{\partial^2 D}{\partial x^2}$ 就是 H。而 $\hat{x}$ 就是我们要解的矩阵求逆过程。

$$dD + HX = 0 \tag{5.18}$$

关于如何求 X，程序使用了矩阵的 LU 分解来快速求逆，这个过程大家可以参考 Dezeming-Family 的《矩阵的 LU 分解与应用》。

如果解出的三个方向的偏移量都小于 0.5，说明已经定位到精确点了，就可以跳出迭代循环，去计算关键点响应尺度了。

如果三个偏移量任意一个大于一定阈值（程序里阈值是最大整数的 $\frac{1}{3}$），则说明精准定位错误，此关键点将被抛弃。

如果上面条件都不满足，则将关键点进行偏移。如果偏移后的关键点位置超过了图像边界范围，就返回 false。关键点只能在该组内的区域移动，不能移动到其他组里。如果没有超过移动范围，且偏移量的绝对值都大于 0.5，则说明关键点需要继续迭代拟合。

计算响应尺度

由上面更新的层坐标 layer 计算关键点所在层在 DoG 金字塔中的层索引：

```
1   int idx = octv*(nOctaveLayers+2) + layer;
```

之后再次计算新关键点位置 $\boldsymbol{x}$ 的 $\frac{\partial D}{\partial x}$。

极值响应值的计算为：

$$D(\boldsymbol{x}) = D(\boldsymbol{x_0}) + \frac{1}{2}\frac{\partial D^T(\boldsymbol{x_0})}{\partial \boldsymbol{x}}\hat{x} \tag{5.19}$$

$\left(\frac{\partial D(\boldsymbol{x_0})}{\partial x}\right)^T \hat{x}$ 的计算表示如下，其实就是点乘：

```
1   float t = dD.dot(Matx31f(xc, xr, xi));
```

然后计算关键点响应是否大于 T/S 这个阈值（其实前面介绍原理时已经介绍过这两行代码了），如果没到阈值就说明不是关键点。

然后再消除边缘响应，计算方法之前讲得非常清楚了，这里就不再赘述了。在判断时，首先保证行列式值要大于 0，然后再判断变换响应。edgeThreshold 默认设置为 10。

```
1   if( det <= 0 || tr*tr*edgeThreshold >= (edgeThreshold + 1)*(
        edgeThreshold + 1)*det )
2   return false;
```

为什么要让 Det 大于 0 呢，因为如果等于 0，说明有为 0 的特征值，则要舍去。如果小于 0，说明两个特征值一正一负，也不符合我们对“特征值描述函数变化快慢的性质”的要求。

到目前为止，关于精准定位关键点的内容我们已经讲完了。之后，我们就要去计算关键点的方向属性。

calcOrientationHist

该函数的参数列表如下：

```
static float calcOrientationHist( const Mat& img, Point pt, int radius
    , float sigma, float* hist, int n );
```

img 表示调整以后的关键点所在 DOG 图像层，pt 表示在该尺度图像上的坐标点：

```
//具体输入的参数
img: gauss_pyr[o*(nOctaveLayers+3) + layer]
pt: Point(c1, r1)
```

radius 即邻域半径，我们计算时设置的是 $3\times1.5\sigma_L$。因为我们在计算 kpt.size 时使用的公式是 $\sigma_0 2^{o+\frac{s}{S}}$，而且因为是相对于第 0 层扩大一倍以后的初始图像，因此乘以了 2，所以我们在计算 σ_L 时要先除以 2（乘 0.5），然后再除以 2^o。

样本方向也要被高斯加权滤波，前面也介绍过，滤波系数 $\sigma = 1.5\times\sigma_L$。

```
float scl_octv = kpt.size*0.5f/(1 << o);
// determines gaussian sigma for orientation assignment
static const float SIFT_ORI_SIG_FCTR = 1.5f;
// determines the radius of the region used in orientation
    assignment
static const float SIFT_ORI_RADIUS = 3 * SIFT_ORI_SIG_FCTR;
radius: cvRound(SIFT_ORI_RADIUS * scl_octv
sigma: SIFT_ORI_SIG_FCTR * scl_octv
```

hist 和 n 表示存储 n 个方向的直方图，这里的 n 为 36：

```
static const int SIFT_ORI_HIST_BINS = 36;
static const int n = SIFT_ORI_HIST_BINS;
hist: float hist[n];
```

我们先跳过 calcOrientationHist 函数主体部分，接着往下讲。该函数返回值是表示主方向值 omax，之后计算辅方向阈值，即主方向的 80%：

```
// orientation magnitude relative to max that results in new feature
static const float SIFT_ORI_PEAK_RATIO = 0.8f;
float mag_thr = (float)(omax * SIFT_ORI_PEAK_RATIO);
```

之后的操作也与我们前面在“关键点方向分配”一节的介绍相同，注意需要进行循环圆周处理，即 hist[-1] 要改为 hist[n-1]，即 hist[-2] 要改为 hist[n-2]：

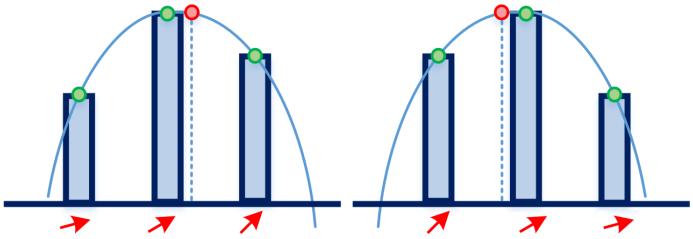
```
1  //直方图当前直方图柱索引，l为前一个柱索引，r2为后一个柱索引。如果小
       于0或者大于等于n则需要循环圆周处理
2  for( int j = 0; j < n; j++ ){
3      int l = j > 0 ? j - 1 : n - 1;
4      int r2 = j < n-1 ? j + 1 : 0;
5      //判断当前柱是否大于直方图辅方向阈值，以及是否大于前后相邻的两个
           柱。
6      if( hist[j] > hist[l]  &&  hist[j] > hist[r2]  &&  hist[j] >=
           mag_thr ) {
7          ......
8      }
9  }
```

因为柱表示的是一个范围，比如直方图第一柱的角度范围为 0 度到 9 度，第二柱的范围是 10 度到 19 度，因此对于峰值位置需要进行插值拟合来得到辅方向值（将最接近每一峰值的直方图上的三个值拟合成抛物线），假如我们得到第 i 个柱为主峰值，则抛物线拟合公式如下，其中 $H(i)$ 代表的是直方图第 i 个柱。这里的 $i-1$、i 和 $i+1$ 都要做循环圆周处理：

$$H = i + \frac{H(i-1) - H(i+1)}{2 \times \big(H(i-1) + H(i+1) - 2 \times H(i)\big)}, \quad i = 0, \ldots, 35 \tag{5.20}$$

$$\theta = 360 - 10 \times H \tag{5.21}$$

可以看到，当主峰值的右柱更高时，拟合点是整数值往右偏，左柱更高则相反：



至于这里的 θ 为什么要这么计算我也不太清楚，因为在后面使用角度时又重新使用 360 减去 kpt.angle 得到角度值。也就是说其实 θ 的值就是 $10 \times H$。

程序描述如下：

```
1  //即上述拟合公式
2  float bin = j + 0.5f * (hist[l]-hist[r2]) / (hist[l] - 2*hist[j] +
       hist[r2]);
3  //这也是在做循环圆周处理：
4  bin = bin < 0 ? n + bin : bin >= n ? bin - n : bin;
5  kpt.angle = 360.f - (float)((360.f/n) * bin);
```

然后判断，如果当前角度与 360 度非常接近，就算为 0 度。

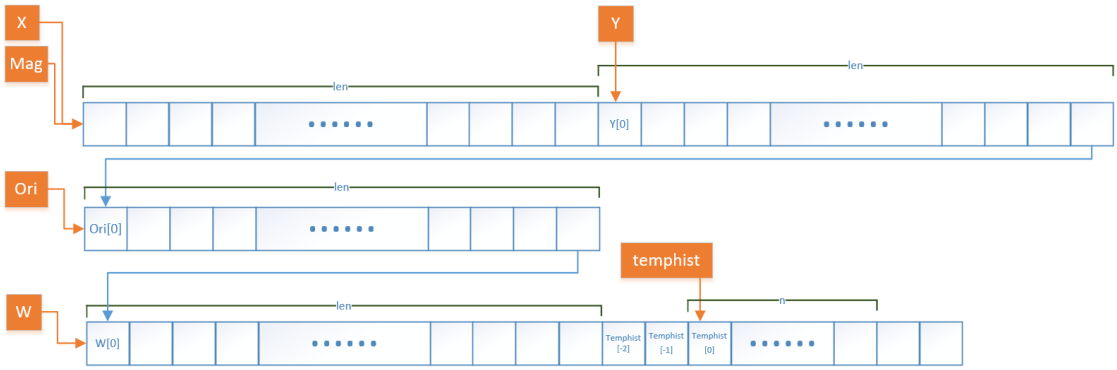
```
if(std::abs(kpt.angle - 360.f) < FLT_EPSILON)
  kpt.angle = 0.f;
```

处理完以后，就将该关键点放入关键点数组中。因此可知，calcOrientationHist 函数只是求一个最主要的方向值，该值的百分之八十作为判断是否为关键点辅方向的阈值，每当有辅方向大于阈值，就作为关键点的一个辅方向（其实就是创建为一个新的关键点）。

我们最后开始介绍 calcOrientationHist 函数。该函数的传入参数我们本节前面已经介绍过，我们现在介绍函数内部的执行过程。

首先定义了 i,j,k,len 这几个变量，len 表示计算特征点方向时的特征点邻域像素数。expf_scale 是高斯加权系数中的 e 的指数的常数部分。

之后使用 AutoBuffer 来分配一段内存 buf，这段内存的分配比较复杂，我详细解释一下。程序定义了几个 float 类型的指针，其中 X 和 Mag 共同使用一段长度为 len 的空间，X 表示 x 轴方向的差分，Mag 表示梯度幅值，这样分配是为了复用内存空间来节省程序运行中使用的内存空间；Y 使用一段长度为 len 的空间，表示 Y 方向的差分；Ori 占用一段长度为 len 的空间，表示梯度幅角；W 为高斯加权值，长度是 len（下一个指针 temphist 偏离了 len+2 的长度，但这多余的 2 的长度是为 temphist 提供的）；temphist 表示暂存梯度方向直方图，考虑到要进行循环圆周处理，长度为 n+4（根据申请的总长度 len*4 + n+4 就可以计算出来，它的长度是 n+2，再加上用于循环圆周处理的 temphist[-2] 和 temphist[-1]，就是 n+4 的长度了）。我们画一个示意图来表示一下（为了避免太长，蓝色连接线表示是连续的内存区）：



程序表示如下：

```
AutoBuffer<float> buf(len*4 + n+4);
float *X = buf, *Y = X + len, *Mag = X, *Ori = Y + len, *W = Ori +
    len;
float* temphist = W + len + 2;
//初始化为0
for( i = 0; i < n; i++ )
  temphist[i] = 0.f;
```

然后就是一个循环函数，该函数循环关键点所在邻域，跳过超出图像边界的点，计算每个点的 x 方向和 y 方向的梯度 $L(x+1,y)-L(x-1,y)$ 以及 $L(x,y+1)-L(x,y-1)$，保存在 X 和 Y 中。

然后调用函数来求高斯加权权重，幅角方向和幅值：

```
cv::hal::exp32f(W, W, len);
cv::hal::fastAtan2(Y, X, Ori, len, true);
cv::hal::magnitude32f(X, Y, Mag, len);
```

然后计算将方向计算添加到每个直方图中，并为了后面的平滑，进行循环圆周处理的赋值工作：

```
k = 0;
for( ; k < len; k++ )
{
    int bin = cvRound((n/360.f)*Ori[k]);
    if( bin >= n )
        bin -= n;
    if( bin < 0 )
        bin += n;
    temphist[bin] += W[k]*Mag[k];
}
temphist[-1] = temphist[n-1];
temphist[-2] = temphist[n-2];
temphist[n] = temphist[0];
temphist[n+1] = temphist[1];
```

之后使用公式进行平滑：

$$H(i) = \frac{h(i-2)+h(i+2)}{16} + \frac{4\times(h(i-1)+h(i+1))}{16} + \frac{6\times h(i)}{16}, \quad i = 0,\ldots,35 \tag{5.22}$$

平滑完以后找到最大值作为主方向峰值即可。

到目前为止，我们已经获取了关键点信息，接下来我们的任务就是计算生成关键点描述子了。

5.4 计算生成描述子

计算生成描述子的过程用到了下面两个函数：

- calcDescriptors：计算特征点描述子。
- calcSIFTDescriptor：计算特征点的特征矢量。

calcDescriptors

在调用该函数前，detectAndCompute 函数里的内容如下：

```
if( _descriptors.needed() ) {
  // SIFT_DESCR_WIDTH*SIFT_DESCR_WIDTH*SIFT_DESCR_HIST_BINS=4×4×8=128
```

```
3   int dsize = descriptorSize();
4   // 为特征点描述符分配足够的空间，可以理解为是一个Mat，列数等于dsize
        ，行数等于关键点个数。
5   _descriptors.create((int)keypoints.size(), dsize, CV_32F);
6   Mat descriptors = _descriptors.getMat();
7   //计算特征点描述子，gpyr是高斯金字塔，nOctaveLayers默认是3，
        firstOctave是第一组的索引
8   calcDescriptors(gpyr, keypoints, descriptors, nOctaveLayers,
        firstOctave);
9 }
```

calcDescriptors 也是调用了并行函数，每个线程为一个关键点计算描述符：

```
1 parallel_for_(Range(0, static_cast<int>(keypoints.size())),
    calcDescriptorsComputer(gpyr, keypoints, descriptors,
    nOctaveLayers, firstOctave));
```

calcDescriptorsComputer 的 operator() 操作符函数里，先把当前特征点的所在层和组以及尺度使用 unpackOctave 来提取，然后判断一下层和组是否在范围内：

```
1 KeyPoint kpt = keypoints[i];
2 int octave, layer;
3 float scale;
4 unpackOctave(kpt, octave, layer, scale);
5 CV_Assert(octave >= firstOctave && layer <= nOctaveLayers+2);
```

然后计算出当前特征点相对于它所在组的基准层尺度图像的尺度，即 $\sigma_0 2^{\frac{s}{S}}$，我们前面计算的 kpt.size 的计算式是 $\sigma_0 2^{o+\frac{s}{S}}$，上面程序中的 scale 则是 2^{-o}，因此用 scale 乘以 kpt.size 就可以得到特征点所在层在当前组的尺度了：

```
1   float size=kpt.size*scale;
```

计算当前特征点所在的高斯尺度图像的位置坐标。因为我们之前在 adjustLocalExtrema 函数中得到的关键点位置都是相对于初始图像的位置（注意在计算朝向直方图时使用的点坐标是 Point(c1, r1)，即在当前组图像的像素位置，而不是 kpt.pt）：

```
1   //代码回顾：adjustLocalExtrema
2   kpt.pt.x = (c + xc) * (1 << octv);
3   kpt.pt.y = (r + xr) * (1 << octv);
```

因此我们索引到该组内的位置索引就应该除以 2^o，即乘以 scale。

```
1   Point2f ptf(kpt.pt.x*scale, kpt.pt.y*scale);
```

最后定位到关键点所在的高斯图像：

```
const Mat& img = gpyr[(octave - firstOctave)*(nOctaveLayers + 3) +
    layer];
```

前面讲过，angle 在这里被 360 又减了一次，因此其实值就是 $10\times H$。之后再做一次判断，如果方向角度非常接近 360 度，则让它等于 0 度：

```
float angle = 360.f - kpt.angle;
if(std::abs(angle - 360.f) < FLT_EPSILON)
    angle = 0.f;
```

之后，再为当前的关键点计算描述子，即调用 calcSIFTDescriptor 函数。

calcSIFTDescriptor

终于讲到了最后一个函数。讲完该函数以后，我们就把整个 SIFT 算法讲解完了。但我不得不说，这个函数是当前所有函数里比较难理解的，而且非常长。不过幸运的是不涉及多少数学知识。

calcSIFTDescriptor 函数在调用时的参数中：

```
scl: size * 0.5f
d: SIFT_DESCR_WIDTH = 4 //描述子宽度
n: SIFT_DESCR_HIST_BINS = 8 //即8个方向
dst: descriptors.ptr<float>((int)i) //即保存描述子的数组
```

上面输入参数里，可能是为了减少计算范围，令 size 乘以了 0.5（猜测，实际不太清楚）；$d\times d\times n$ 等于 128，我们在前面已经讲过。接下来我们进入程序内部。

首先根据四舍五入值求得特征点位置坐标，并计算特征点方向的正弦余弦：

```
Point pt(cvRound(ptf.x), cvRound(ptf.y)); //特征点的位置坐标
float cos_t = cosf(ori*(float)(CV_PI/180));
float sin_t = sinf(ori*(float)(CV_PI/180));
```

然后计算 45 度的倒数和高斯加权函数中的 e 的指数的常数部分：

```
float bins_per_rad = n / 360.f;
float exp_scale = -1.f/(d * d * 0.5f);
```

然后计算 $3\sigma_L$ 以及特征点邻域半径。下面代码的浮点数值为 $\sqrt{2}$，radius 的值即 $\frac{3\sigma_L(d+1)\sqrt{2}}{2}$：

```
float hist_width = SIFT_DESCR_SCL_FCTR * scl;
int radius = cvRound(hist_width * 1.4142135623730951f * (d + 1) *
    0.5f);
//为避免半径过大，使其小于图像斜对角线长度
radius = std::min(radius, (int) sqrt(((double) img.cols)*img.cols +
    ((double) img.rows)*img.rows));
```

在“坐标与加权”一节介绍过，因为要把坐标压缩到 $d \times d$ 的区域内，因此坐标计算为：

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \frac{1}{3\sigma_L} \begin{bmatrix} x' \\ y' \end{bmatrix} + \frac{d}{2} \tag{5.23}$$

程序里也就是：

```
1  cos_t /= hist_width;
2  sin_t /= hist_width;
```

之后开辟内存空间。len 为特征点邻域内像素的数量，histlen 中之所以每个维度变量都加了 2，是为了循环圆周处理留出一定的空间：

```
1  int i, j, k, len = (radius*2+1)*(radius*2+1), histlen = (d+2)*(d+2)
       *(n+2);
2  int rows = img.rows, cols = img.cols;
```

然后开始分配内存，这里的内存结构仍然是比较复杂，我详细介绍一下。

X 表示 x 方向梯度；Y 表示 y 方向梯度；Mag 表示梯度幅值；Ori 表示梯度幅角；W 为高斯加权值，其中 Y 和 Mag 共享一段内存空间，长度都为 len。

RBin 和 CBin 分别表示 $d \times d$ 的邻域范围的点旋转后的横、纵坐标，长度都是 len；hist 表示直方图的值，hist 的长度为 histlen。

```
1  AutoBuffer<float> buf(len*6 + histlen);
2  float *X = buf, *Y = X + len, *Mag = Y, *Ori = Mag + len, *W = Ori +
       len;
3  float *RBin = W + len, *CBin = RBin + len, *hist = CBin + len;
4  //直方图值清零
5  for( i = 0; i < d+2; i++ ) {
6      for( j = 0; j < d+2; j++ )
7          for( k = 0; k < n+2; k++ )
8              hist[(i*(d+2) + j)*(n+2) + k] = 0.;
9  }
```

然后遍历整个邻域像素：

```
1  for( i = -radius, k = 0; i <= radius; i++ )
2    for( j = -radius; j <= radius; j++ ) {
3      ......
4    }
```

针对邻域中的每个像素，首先计算旋转以后的位置坐标：

```
1  float c_rot = j * cos_t - i * sin_t;
2  float r_rot = j * sin_t + i * cos_t;
```

把邻域区域的原点从中心位置移到该区域的左下角，即加 $\frac{d}{2}$ 即可。再减 0.5f 的目的是进行坐标平移（参考“坐标与加权”一节），从而在三线性插值计算中，计算的是正方体内的点对正方体 8 个顶点的贡献大小，而不是对正方体的中心点的贡献大小:

```
1   float rbin = r_rot + d/2 - 0.5f;
2   float cbin = c_rot + d/2 - 0.5f;
```

之所以没有对角度 obin 进行坐标平移是因为角度是连续的量，可以直接分配从而无需平移，但像素坐标是有中心点的，所以插值需要计算相对于中心点的偏移，因此平移可以方便计算；角度既然是个连续值，也就是说对应于角度方向的 bins 中，第一个 bin 代表的范围可以是 [0——45] 度，也可以是 [-22.5——22.5] 度之间，我们的目标只是为了划分为 8 个方向，而不是一定要局限于每个方向的划分范围。

得到该像素点的位置坐标，对该位置计算梯度：

```
1  int r = pt.y + i, c = pt.x + j;
2  //确定邻域像素是否在 dXd 的正方形内，以及是否超过了图像边界。注意rbin
       和cbin都是浮点数，也就是说它们会给坐标在一定范围加权
3  if( rbin > -1 && rbin < d && cbin > -1 && cbin < d && r > 0 && r <
       rows - 1 && c > 0 && c < cols - 1 ) {
4    //使用差分来计算 x 和 y 方向的一阶导数，这里省略了的分母，是因为没有
         分母部分不影响后面的归一化处理
5    float dx = (float)(img.at<sift_wt>(r, c+1) - img.at<sift_wt>(r, c-1)
         );
6    float dy = (float)(img.at<sift_wt>(r-1, c) - img.at<sift_wt>(r+1, c)
         );
7    //保存数据
8    X[k] = dx; Y[k] = dy; RBin[k] = rbin; CBin[k] = cbin;
9    //高斯加权系数
10   W[k] = (c_rot * c_rot + r_rot * r_rot) * exp_scale;
11   //k记录的是实际的邻域像素数量（旋转以后不在邻域的像素要被去除）
12   k++;
13 }
```

然后将已经存储的量进行一下计算：

```
1   len = k; //赋值
2   fastAtan2(Y, X, Ori, len, true); //计算梯度幅角
3   magnitude(X, Y, Mag, len); //计算梯度幅值
4   exp(W, W, len); //计算高斯加权函数
```

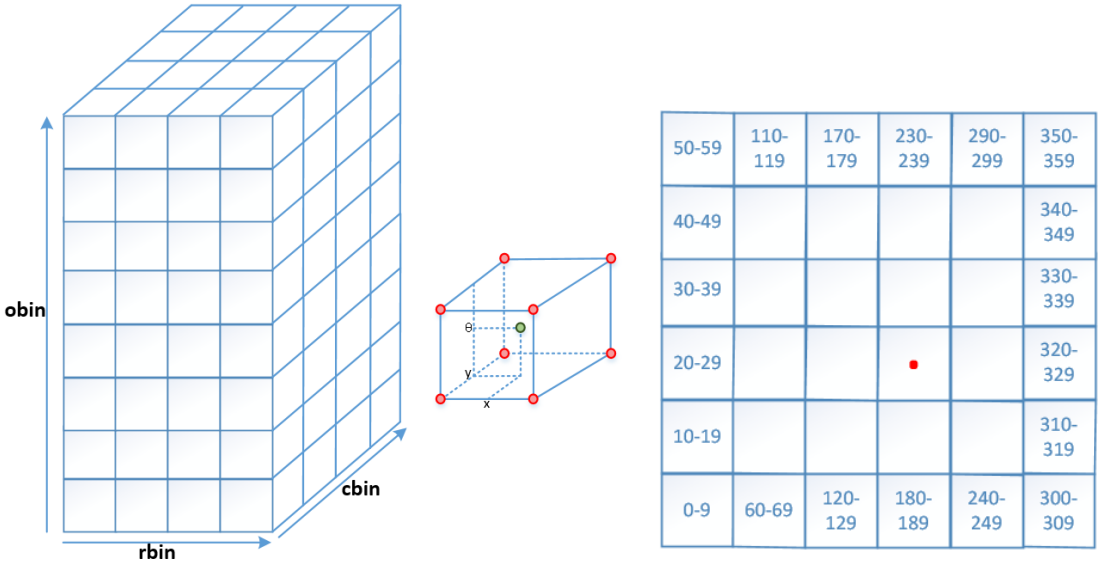
然后遍历所有的邻域像素（我们先看程序，然后再解释里面的坐标）：

```
1  k = 0;
2  for( ; k < len; k++ ) {
```

```
3   //得到它在 dXd 邻域区域的坐标
4   float rbin = RBin[k], cbin = CBin[k];
5   //得到幅角在 8 等份中的某一等份，即三维直方图中的高的位置；注意幅角
        值是相对于ori值的大小
6   float obin = (Ori[k] - ori)*bins_per_rad;
7   //高斯加权以后的梯度幅值
8   float mag = Mag[k]*W[k];
9   //三维坐标的整数部分，表示在 d X d X n区域中属于哪个正方体
10  int r0 = cvFloor( rbin );
11  int c0 = cvFloor( cbin );
12  int o0 = cvFloor( obin );
13  //小数部分，rbin， cbin 和 obin 为三维坐标的小数部分
14  rbin -= r0;
15  cbin -= c0;
16  obin -= o0;
17  //如果角度 o0 小于 0 度或大于 360 度，则根据圆周循环，把该角度调整到
        0～360 度之间
18  if( o0 < 0 )
19      o0 += n;
20  if( o0 >= n )
21      o0 -= n;
22  // 根据三线性插值法，计算该像素对正方体的 8 个顶点的贡献大小
23  float v_r1 = mag*rbin, v_r0 = mag - v_r1;
24  float v_rc11 = v_r1*cbin, v_rc10 = v_r1 - v_rc11;
25  float v_rc01 = v_r0*cbin, v_rc00 = v_r0 - v_rc01;
26  float v_rco111 = v_rc11*obin, v_rco110 = v_rc11 - v_rco111;
27  float v_rco101 = v_rc10*obin, v_rco100 = v_rc10 - v_rco101;
28  float v_rco011 = v_rc01*obin, v_rco010 = v_rc01 - v_rco011;
29  float v_rco001 = v_rc00*obin, v_rco000 = v_rc00 - v_rco001;
30  // 得到该像素点在三维直方图中的索引
31  int idx = ((r0+1)*(d+2) + c0+1)*(n+2) + o0;
32  hist[idx] += v_rco000;
33  hist[idx+1] += v_rco001;
34  hist[idx+(n+2)] += v_rco010;
35  hist[idx+(n+3)] += v_rco011;
36  hist[idx+(d+2)*(n+2)] += v_rco100;
37  hist[idx+(d+2)*(n+2)+1] += v_rco101;
38  hist[idx+(d+3)*(n+2)] += v_rco110;
39  hist[idx+(d+3)*(n+2)+1] += v_rco111;
```

```
40 }
```

bin 的结构表示如下，注意为了处理插值，因此 $d \times d \times n$ 的区域需要左右前后上下各扩大一个格子来存储。根据上面的 idx 的计算方式就能得到空间排列方法，即对于 $d \times d$ 的邻域，先排列底层每个格子的 $8+2$ 个角度格子，然后再排列底层的列方向 cbin 的 d+2 个格子，最后再排列行方向的 d+2 个格子：



在左图中，比如我们要计算 $d \times d \times n$ 的区域中，第 3 行 2 列（索引为 (2,1)）第 5 个方向 bin 中的值，即左图红点格子里 obin 方向第 6 个格子 (索引为 5) 里的值：((2+1)*(4+2)+1+1)*10 + 5 = 205。

之后，对幅角小于 0 或者大于 360 的值进行调整：

```
1  for( i = 0; i < d; i++ )
2    for( j = 0; j < d; j++ )
3    {
4      //d X d区域格子索引
5      int idx = ((i+1)*(d+2) + (j+1))*(n+2);
6      //循环圆周处理
7      hist[idx] += hist[idx+n];
8      hist[idx+1] += hist[idx+n+1];
9      //幅角直方图数组不需要用来插值的扩展格子
10     for( k = 0; k < n; k++ )
11         dst[(i*d + j)*n + k] = hist[idx+k];
12   }
```

上面的程序我实际分析并测试了一下，好像 hist[idx+n+1] 这个格子里并不会被插值到任何数据，因此是 0，不知道为什么还要多给一个格子。其实在前面的代码中已经当前格子的整数索引限定在了 [0-n-1]，因此能够插值到的格子就限定到了 [0-n]：

```
1    if( o0 < 0 )
```

```
    o0 += n;
    if( o0 >= n )
    o0 -= n;
```

补充 5.6 (调试方法) 我在源码中设定只要 hist[idx + n + 1] 大于 0 就打印输出问号，否则打印输出句号，最后输出的全是句号，说明 hist[idx + n + 1] 的值都是 0，所以很明显源码多给了角度方向一个格子。

然后再求特征矢量值的平方和：

```
    float nrm2 = 0;
    len = d*d*n;
    k = 0;
    for( ; k < len; k++ )
      nrm2 += dst[k]*dst[k];
```

因为归一化以后，特征矢量中大于 0.2 的元素要被设定为 0.2。这里采取的方法是反归一化处理，为了避免累加误差，使用 0.2 乘以平方和开方值，得到反归一化阈值 thr：

```
    float thr = std::sqrt(nrm2)*SIFT_DESCR_MAG_THR;
```

然后再次遍历数组，将大于阈值 thr 的元素替换为 thr，然后进行归一化，并最终输出到目标数组中：

```
    i = 0, nrm2 = 0;
    for( ; i < len; i++ )
    {
        float val = std::min(dst[i], thr);
        dst[i] = val;
        nrm2 += val*val;
    }
    nrm2 = SIFT_INT_DESCR_FCTR/std::max(std::sqrt(nrm2), FLT_EPSILON);
    //赋值给最终要输出的描述子向量:
    k = 0;
    for( ; k < len; k++ )
    {
        dst[k] = saturate_cast<uchar>(dst[k]*nrm2);
    }
```

SIFT_INT_DESCR_FCTR 的值是 512.f，作为将浮点型转换为整型时所用到的系数。我们最后是要将数据保存到 uchar 类型的存储结构中，因此乘以这个系数。

至此，全部的 OpenCV 的 Sift 源码分析就已经讲完了。回顾整个过程，真的是非常漫长和艰

辛，但相信大家的收获也是非常大的。希望大家能够对照着本书将源码过程至少看两到三遍，加深一下印象。

6. 图像特征匹配

本章我们讲解 *Sift* 特征的重要应用——图像的特征匹配。因为并不是本书重点内容，所以我们会在这一章讲解一般的特征匹配流程，介绍一下源码中的暴力匹配方法。

本章内容相当少，但我还是将其作为单独的一章来描述，顺便在本章写点结语和感悟吧。

6.1 使用 OpenCV 的源码匹配

回顾一下前面列出的匹配代码：

```
1 cv::Ptr<cv::DescriptorMatcher> matcher = cv::DescriptorMatcher::
      create("BruteForce");
2 vector<cv::DMatch> matches;
3 Mat imgMatches;
4 matcher->match(descriptor1, descriptor2, matches);
5 cv::drawMatches(img1, keypoint1, img2, keypoint2, matches,
      imgMatches);
6 imshow("matches", imgMatches);
```

其中，DMatch 类一共有下面几个变量：

```
1 CV_PROP_RW int queryIdx; //!< query descriptor index
2 CV_PROP_RW int trainIdx; //!< train descriptor index
3 CV_PROP_RW int imgIdx;   //!< train image index
4
5 CV_PROP_RW float distance;
```

在匹配的结束后，matches 数组里的 queryIdx 会从 0 依次递增，表示第一个描述子数组里的描述子向量索引，trainIdx 对应于第二个描述子数组里匹配到的描述子索引。imgIdx 这里用不到。

DescriptorMatcher 定义在 features2d.hpp 文件里。

DescriptorMatcher::create 函数实体在 matchers.cpp 文件中，对于 BruteForce 暴力匹配，会生成 BFMatcher，该类继承自 DescriptorMatcher，normType=NORM_L2，该值为 4。

这里使用的匹配方法就是 knn，即两个向量相减以后的二范数越小，说明越接近，对于数组 1 里的描述子 A，遍历数组 2，计算与里面所有向量的距离二范数，然后找到二范数最小的作为匹配到的特征点，保存为到 matches 中。

关于匹配的过程我也不想再介绍过多的内容了，就让本书在这里完结吧！

6.2 本书结语

从 5 月 10 日开始，加班熬夜，一直到 7 月 2 日，总算完成了这本书的第一版。

在写源码解读中，我也遇到了很多新的以前没有注意的问题，为此，我也对源码进行了各种修改和调试，意图将源码的全部思想研究明白。除了个别地方有歧义之外，整个代码的实现和 SIFT 原理也是紧密结合的。

SIFT 还有很多变体，甚至还有一些稳定的边缘提取和重构的方法，都非常有意思，也很值得大家去学习。SIFT 的一些加强版算法，例如 SURF 和 CSIFT，用来优化提取模糊图像的关键点信息等其他方面，使得图像特征提取越来越稳定和强大，但是，我们人脑如何理解特征，如何分析特征，这个过程我想恐怕还要等很久以后才能得到解释，但我相信，图像的特征提取方法和特征的解释性会不断在大家的努力和奋斗中越来越强大。

最后，感谢大家能够看完本书，如果对于本书的描述有建议，以及发现本书中的错误，也欢迎在网站留言。您的帮助将是我们不断完善电子书的有力支持！

Bibliography

[1] Lowe D G . Object recognition from local scale-invariant features[C]// Proc of IEEE International Conference on Computer Vision. 1999.

[2] Lowe D G . Distinctive Image Features from Scale-Invariant Keypoints[J]. International Journal of Computer Vision, 2004, 60(2):91-110.

[3] Name F, Training O, Training P, et al. Computer Vision Algorithms and Applications. 2014.

[4] Brown, M. and Lowe, D.G. 2002. Invariant features from interest point groups. In British Machine Vision Conference, Cardiff, Wales, pp. 656–665.

[5] 重要参考：https://www.cnblogs.com/ronny/p/4028776.html

[6] https://blog.csdn.net/zddblog/article/details/7521424

[7] https://blog.csdn.net/qq_37374643/article/details/88606351

[8] https://zhuanlan.zhihu.com/p/49447503

[9] 重要参考：https://zhuanlan.zhihu.com/p/261697473

[10] https://blog.csdn.net/onlyzkg/article/details/11570965?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-9.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-9.control

[11] https://blog.csdn.net/lwzkiller/article/details/55050275 （文中关于关键点检测的内容构建参考了这里的叙述）

[12] https://blog.csdn.net/xiaowei_cqu/article/details/8113565

[13] opencv2.4.9 源码分析——SIFT 赵春江 blog.csdn.net/zhaocj

[14] https://blog.csdn.net/haima1998/article/details/82079042

[15] https://blog.csdn.net/masibuaa/article/details/9191309

[16] https://blog.csdn.net/qq_31806429/article/details/79242399 (OpenCV 并行与循环)