opencv2.4.9 源码分析——SIFT

赵春江

blog.csdn.net/zhaocj

一、SIFT 算法

SIFT（尺度不变特征变换，Scale-Invariant Feature Transform）是在计算机视觉领域中检测和描述图像中局部特征的算法，该算法于 1999 年被 David Lowe 提出，并于 2004 年进行了补充和完善。该算法应用很广，如目标识别，自动导航，图像拼接，三维建模，手势识别，视频跟踪等。不幸的是，该算法已经在美国申请了专利，专利拥有者为 Lowe 所在的加拿大不列颠哥伦比亚大学，因此我们不能随意使用它。

用 SIFT 算法所检测到的特征是局部的，而且该特征对于图像的尺度和旋转能够保持不变性。同时，这些特征对于亮度变化具有很强的鲁棒性，对于噪声和视角的微小变化也能保持一定的稳定性。SIFT 特征还具有很强的可区分性，它们很容易被提取出来，并且即使在低概率的不匹配情况下也能够正确的识别出目标来。因此鲁棒性和可区分性是 SIFT 算法最主要的特点。

SIFT 算法分为 4 个阶段：

1、尺度空间极值检测：该阶段是在图像的全部尺度和全部位置上进行搜索，并通过应用高斯差分函数可以有效地识别出尺度不变性和旋转不变性的潜在特征点来；

2、特征点的定位：在每个候选特征点上，一个精细的模型被拟合出来用于确定特性点的位置和尺度。而特征点的最后选取依赖的是它们的稳定程度；

3、方向角度的确定：基于图像的局部梯度方向，为每个特性点分配一个或多个方向角度。所有后续的操作都是相对于所确定下来的特征点的角度、尺度和位置的基础上进行的，因此特征点具有这些角度、尺度和位置的不变性；

4、特征点的描述符：在所选定的尺度空间内，测量特征点邻域区域的局部图像梯度，将这些梯度转换成一种允许局部较大程度的形状变形和亮度变化的描述符形式。

下面就详细阐述 SIFT 算法的这 4 个阶段：

1、尺度空间极值检测

特征点检测的第一步是能够识别出目标的位置和尺度，对于同一个目标在不同的视角下这些位置和尺度可以被重复的分配。并且这些检测到的位置是不随图像尺度的变化而改变的，因为它们是通过搜索所有尺度上的稳定特征得到的，所应用的工具就是被称为尺度空间的连续尺度函数。

真实世界的物体只有在一定尺度上才有意义，例如我们能够看到放在桌子上的水杯，但对于整个银河系，这个水杯是不存在的。物体的这种多尺度的本质在自然界中是普遍存在的。尺度空间就是试图在数字图像领域复制这个概念。又比如，对于某幅图像，我们是想看到叶子还是想看到整棵树，如果是树，那么我们就应该有意识的去除图像的细节部分（如叶子、细枝等）。在去除细节部分的过程中，我们一定要确保不能引进新的错误的细节。因此在创建尺度空间的过程中，我们应该对原始图像逐渐的做模糊平滑处理。进行该操作的唯一方法是高斯模糊处理，因为已经被证实，高斯函数是唯一可能的尺度空间核。

图像的尺度空间用 $L(x, y, \sigma)$函数表示，它是由一个变尺度的高斯函数 $G(x, y, \sigma)$与图像 $I(x, y)$通过卷积产生，即

$$L(x,y,\sigma) = G(x,y,\sigma) \otimes I(x,y) \tag{1}$$

其中，⊗表示在 x 和 y 两个方向上进行卷积操作，而 $G(x, y, \sigma)$为

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

σ 是尺度空间因子，它决定着图像模糊平滑处理的程度。在大尺度下（σ 值大）表现的是图像的概貌信息，在小尺度下（σ 值小）表现的是图像的细节信息。因此大尺度对应着低分辨率，小尺度对应着高分辨率。(x, y)则表示在 σ 尺度下的图像像素坐标。

需要说明的是，公式 1 中的图像 $I(x, y)$是具有无限分辨率的图像，也就是说它的尺度 σ=0，即 $I(x, y) = L(x, y, 0)$。因此公式 1 得到的尺度空间图像 $L(x, y, \sigma)$是由尺度为 0 的图像 $L(x, y, 0)$生成的。很显然，尺度为 0，即无限分辨率的图像是无法获得的，Lowe 就是把初始图像的尺度设定为 0.5。那么由 $L(x, y, \sigma_1)$得到 $L(x, y, \sigma_2)$，即由尺度为 σ_1 的图像生成尺度为 σ_2 的图像的公式为：

$$L(x,y,\sigma_2) = G\left(x,y,\sqrt{\sigma_2^2-\sigma_1^2}\right) \otimes L(x,y,\sigma_1), \qquad \sigma_2 \geq \sigma_1 \tag{3}$$

其中，

$$G\left(x,y,\sqrt{\sigma_2^2-\sigma_1^2}\right) = \frac{1}{2\pi(\sigma_2^2-\sigma_1^2)} e^{-\frac{x^2+y^2}{2(\sigma_2^2-\sigma_1^2)}} \tag{4}$$

由于尺度为 0 的图像无法得到，因此在实际应用中要想得到任意尺度下的图像，一定是利用公式 3 生成的，即由一个已知尺度（该尺度不为 0）的图像生成另一个尺度的图像，并且一定是小尺度的图像生成大尺度的图像。

利用 LoG（高斯拉普拉斯方法，Laplacian of Gaussian），即图像的二阶导数，能够在不同的尺度下检测到图像的斑点特征，从而可以确定图像的特征点。但 LoG 的效率不高。因此

SIFT 算法进行了改进，通过对两个相邻高斯尺度空间的图像相减，得到一个 DoG（高斯差分，Difference of Gaussians）的响应值图像 $D(x, y, \sigma)$来近似 LoG：

$$D(x,y,\sigma) = \big(G(x,y,k\sigma) - G(x,y,\sigma)\big) \otimes I(x,y) = L(x,y,k\sigma) - L(x,y,\sigma) \tag{5}$$

其中，k 为两个相邻尺度空间倍数的常数。

可以证明 DoG 是对 LoG 的近似表示，并且用 DoG 代替 LoG 并不影响对图像斑点位置的检测。而且用 DoG 近似 LoG 可以实现下列好处：第一是 LoG 需要使用两个方向的高斯二阶微分卷积核，而 DoG 直接使用高斯卷积核，省去了卷积核生成的运算量；第二是 DoG 保留了个高斯尺度空间的图像，因此在生成某一空间尺度的特征时，可以直接使用公式 1（或公式 3）产生的尺度空间图像，而无需重新再次生成该尺度的图像；第三是 DoG 具有与 LoG 相同的性质，即稳定性好、抗干扰能力强。

为了在连续的尺度下检测图像的特征点，需要建立 DoG 金字塔，而 DoG 金字塔的建立又离不开高斯金字塔的建立，如下图所示，左侧为高斯金字塔，右侧为 DoG 金字塔：

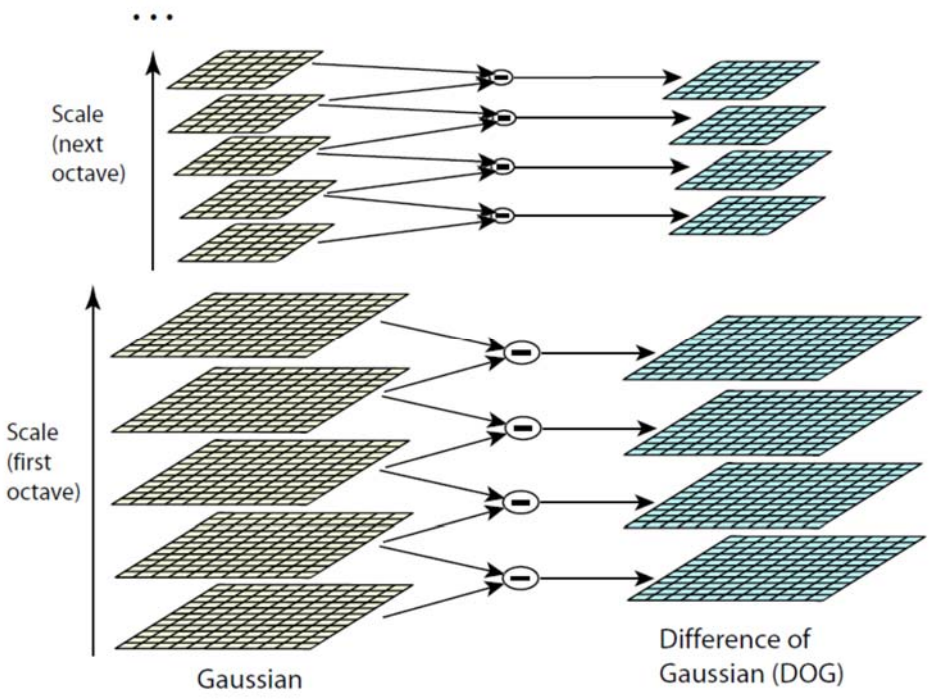

图 1 高斯金字塔和 DoG 金字塔

高斯金字塔共分 O 组（Octave），每组又分 S 层（Layer）。组内各层图像的分辨率是相同的，即长和宽相同，但尺度逐渐增加，即越往塔顶图像越模糊。而下一组的图像是由上一组图像按照隔点降采样得到的，即图像的长和宽分别减半。高斯金字塔的组数 O 是由输入图像的分辨率得到的，因为要进行隔点降采样，所以在执行降采样生成高斯金字塔时，一直到不能降采样为止，但图像太小又毫无意义，因此具体的公式为：

$$O = [\log_2 \min(X, Y) - 2] \tag{6}$$

其中，X 和 Y 分别为输入图像的长和宽，⌊ ⌋表示向下取整。

金字塔的层数 S 为：

$$S = s + 3 \tag{7}$$

Lowe 建议 s 为 3。需要注意的是，除了公式 7 中的第一个字母是大写的 S 外，后面出现的都是小写的 s。

高斯金字塔的创建是这样的：设输入图像的尺度为 0.5，由该图像得到高斯金字塔的第 0 组的第 0 层图像，它的尺度为 σ_0，我们称 σ_0 为基准层尺度，再由第 0 层得到第 1 层，它的尺度为 $k\sigma_0$，第 2 层的尺度为 $k^2\sigma_0$，以此类推。这里的 k 为：

$$k = 2^{\frac{1}{s}} \tag{8}$$

我们以 s=3 为例，第 0 组的 6（s+3=6）幅图像的尺度分别为：

$$\sigma_0，k\sigma_0，k^2\sigma_0，k^3\sigma_0，k^4\sigma_0，k^5\sigma_0 \tag{9}$$

写成更一般的公式为：

$$\sigma = k^r\sigma_0 \qquad r\in[0,\ldots,s+2] \tag{10}$$

第 0 组构建完成后，再构建第 1 组。第 1 组的第 0 层图像是由第 0 组的倒数第 3 层图像经过隔点采样得到的。由公式 10 可以得到，第 0 组的倒数第 3 层图像的尺度为 $k^s\sigma_0$，k 的值代入公式 8，得到了该层图像的尺度正好为 $2\sigma_0$，因此第 1 组的第 0 层图像的尺度仍然是 $2\sigma_0$。但由于第 1 组图像是由第 0 组图像经隔点降采样得到的，因此相对于第 1 组图像的分辨率来说，第 0 层图像的尺度为 σ_0，即尺度为 $2\sigma_0$ 是相对于输入图像的分辨率来说的，而尺度为 σ_0 是相对于该组图像的分辨率来说的。这也就是为什么我们称 σ_0 为基准层尺度的原因（它是每组图像的基准层尺度）。第 1 组其他层图像的生成与第 0 组的相同。因此可以看出，第 1 组各层图像的尺度相对于该组分辨率来说仍然满足公式 10。这样做的好处就是编程的效率会提高，并且也保证了高斯金字塔尺度空间的连续性。而之所以会出现这样的结果，是因为在参数选择上同时满足公式 7、公式 8 以及对上一组倒数第 3 层图像降采样这三个条件的原因。

那么第 1 组各层图像相对于输入图像来说，它们的尺度为：

$$\sigma = 2k^r\sigma_0 \qquad r\in[0,\ldots,s+2] \tag{11}$$

该公式与公式 10 相比较可以看出，第 1 组各层图像的尺度比第 0 组相对应层图像的尺度大了一倍。高斯金字塔的其他组的构建以此类推，不再赘述。下面给出相对于输入图像的

各层图像的尺度公式：

$$\sigma(o,r) = 2^o k^r \sigma_0 \qquad o\in[0,\ldots,O\text{-}1],\quad r\in[0,\ldots,s+2] \tag{12}$$

其中，o 表示组的坐标，r 表示层的坐标，σ_0 为基准层尺度。k 用公式 8 代入，得：

$$\sigma(o,r) = \sigma_0 2^{o+\frac{r}{s}} \qquad o \in [0,\ldots,O-1],\ r \in [0,\ldots,s+2] \tag{13}$$

在高斯金字塔中，第 0 组第 0 层的图像是输入图像经高斯模糊后的结果，模糊后的图像的高频部分必然会减少，因此为了最大程度的保留原图的信息量，Lowe 建议在创建尺度空间前首先对输入图像的长宽扩展一倍，这样就形成了高斯金字塔的第-1 组。设输入图像的尺度为 0.5，那么相对于输入图像，分辨率扩大一倍后的尺度应为 1，由该图像依次进行高斯平滑处理得到第-1 组的各个层的尺度图像，方法与其他组的一样。由于增加了第-1 组，因此公式 13 重新写为：

$$\sigma(o,r) = \sigma_0 2^{o+\frac{r}{s}} \qquad o \in [-1,0,\ldots,O-1],\ r \in [0,\ldots,s+2] \tag{14}$$

DoG 金字塔是由高斯金字塔得到的，即高斯金字塔组内相邻两层图像相减得到 DoG 金字塔。如高斯金字塔的第 0 组的第 0 层和第 1 层相减得到 DoG 金字塔的第 0 组的第 0 层图像，高斯金字塔的第 0 组的第 1 层和第 2 层相减得到 DoG 金字塔的第 0 组的第 1 层图像，以此类推。需要注意的是，高斯金字塔的组内相邻两层相减，而两组间的各层是不能相减的。因此高斯金字塔每组有 s+3 层图像，而 DoG 金字塔每组则有 s+2 层图像。

极值点的搜索是在 DoG 金字塔内进行的，这些极值点就是候选的特征点。

在搜索之前，我们需要在 DoG 金字塔内剔除那些像素值过小的点，因为这些像素具有较低的对比度，它们肯定不是稳定的特征点。

极值点的搜索不仅需要在它所在尺度空间图像的邻域内进行，还需要在它的相邻尺度空间图像内进行，如图 2 所示。

每个像素在它的尺度图像中一共有 8 个相邻点，而在它的下一个相邻尺度图像和上一个相邻尺度图像还各有 9 个相邻点（图 2 中绿色标注的像素），也就是说，该点是在 3×3×3 的立方体内被包围着，因此该点在 DoG 金字塔内一共有 26 个相邻点需要比较，来判断其是否为极大值或极小值。这里所说的相邻尺度图像指的是在同一个组内，因此在 DoG 金字塔内，每一个组的第 0 层和最后一层各只有一个相邻尺度图像，所以在搜索极值点时无需在这两层尺度图像内进行，从而使极值点的搜索就只在每组的中间 s 层尺度图像内进行。

搜索的过程是这样的：从每组的第 1 层开始，以第 1 层为当前层，对第 1 层的 DoG 图

像中的每个点取一个 3×3×3 的立方体，立方体上下层分别为第 0 层和第 2 层。这样，搜索得到的极值点既有位置坐标（该点所在图像的空间坐标），又有尺度空间坐标（该点所在层的尺度）。当第 1 层搜索完成后，再以第 2 层为当前层，其过程与第 1 层的搜索类似，以此类推。

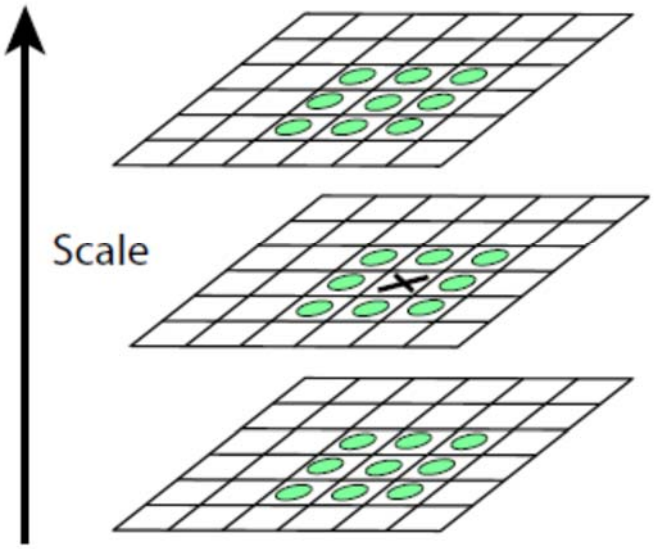

图 2 DoG 中极值点的搜索

2、特征点的定位

通过上一步，我们得到了极值点，但这些极值点还仅仅是候选的特征点，因为它们还存在一些不确定的因素。首先是极值点的搜索是在离散空间内进行的，并且这些离散空间还是经过不断的降采样得到的。如果把采样点拟合成曲面后我们会发现，原先的极值点并不是真正的极值点，也就是离散空间的极值点并不是连续空间的极值点。在这里，我们是需要精确定位特征点的位置和尺度的，也就是要达到亚像素精度，因此必须进行拟合处理。

我们使用泰勒级数展开式作为拟合函数。如上所述，极值点是一个三维矢量，即它包括极值点所在的尺度，以及它的尺度图像坐标，即 $\boldsymbol{X} = (x, y, \sigma)^T$，因此我们需要三维函数的泰勒级数展开式，设我们在 $\boldsymbol{X_0} = (x_0, y_0, \sigma_0)^T$ 处进行泰勒级数展开，则它的矩阵形式为：

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

$$\frac{1}{2}([x \quad y \quad \sigma]-[x_0 \quad y_0 \quad \sigma_0])\begin{bmatrix}\frac{\partial^2 f}{\partial x\partial x} & \frac{\partial^2 f}{\partial x\partial y} & \frac{\partial^2 f}{\partial x\partial \sigma}\\ \frac{\partial^2 f}{\partial x\partial y} & \frac{\partial^2 f}{\partial y\partial y} & \frac{\partial^2 f}{\partial y\partial \sigma}\\ \frac{\partial^2 f}{\partial x\partial \sigma} & \frac{\partial^2 f}{\partial y\partial \sigma} & \frac{\partial^2 f}{\partial \sigma\partial \sigma}\end{bmatrix}\left(\begin{bmatrix}x\\ y\\ \sigma\end{bmatrix}-\begin{bmatrix}x_0\\ y_0\\ \sigma_0\end{bmatrix}\right) \tag{15}$$

公式 15 为舍去高阶项的形式，而它的矢量表示形式为：

$$f(\boldsymbol{X}) = f(\boldsymbol{X_0}) + \frac{\partial f^T}{\partial \boldsymbol{X}}(\boldsymbol{X}-\boldsymbol{X_0}) + \frac{1}{2}(\boldsymbol{X}-\boldsymbol{X_0})^T\frac{\partial^2 f}{\partial \boldsymbol{X}^2}(\boldsymbol{X}-\boldsymbol{X_0}) \tag{16}$$

在这里 $\boldsymbol{X_0}$ 表示离散空间下的插值中心（在离散空间内也就是采样点）坐标，$\boldsymbol{X}$ 表示拟合后连续空间下的插值点坐标，设$\widehat{\boldsymbol{X}} = \boldsymbol{X}-\boldsymbol{X_0}$，则$\widehat{\boldsymbol{X}}$表示相对于插值中心，插值后的偏移量。因此公式 16 经过变量变换后，又可写成：

$$f(\widehat{\boldsymbol{X}}) = f(\boldsymbol{X_0}) + \frac{\partial f^T}{\partial \boldsymbol{X}}\widehat{\boldsymbol{X}} + \frac{1}{2}\widehat{\boldsymbol{X}}^T\frac{\partial^2 f}{\partial \boldsymbol{X}^2}\widehat{\boldsymbol{X}} \tag{17}$$

对上式求导，得

$$\frac{\partial f(\widehat{\boldsymbol{X}})}{\partial \widehat{\boldsymbol{X}}} = \frac{\partial f^T}{\partial \boldsymbol{X}} + \frac{1}{2}\left(\frac{\partial^2 f}{\partial \boldsymbol{X}^2} + \frac{\partial^2 f^T}{\partial \boldsymbol{X}^2}\right)\hat{X} = \frac{\partial f^T}{\partial \boldsymbol{X}} + \frac{\partial^2 f}{\partial \boldsymbol{X}^2}\widehat{\boldsymbol{X}} \tag{18}$$

让公式 17 的导数为 0，即公式 18 为 0，就可得到极值点下的相对于插值中心 $\boldsymbol{X_0}$ 的偏移量：

$$\widehat{\boldsymbol{X}} = -\frac{\partial^2 f^{-1}}{\partial \boldsymbol{X}^2}\frac{\partial f}{\partial \boldsymbol{X}} \tag{19}$$

把公式 19 得到的极值点带入公式 17 中，就得到了该极值点下的极值：

$$\begin{aligned} f(\widehat{\boldsymbol{X}}) &= f(\boldsymbol{X_0}) + \frac{\partial f^T}{\partial \boldsymbol{X}}\widehat{\boldsymbol{X}} + \frac{1}{2}\left(-\frac{\partial^2 f^{-1}}{\partial \boldsymbol{X}^2}\frac{\partial f}{\partial \boldsymbol{X}}\right)^T\frac{\partial^2 f}{\partial \boldsymbol{X}^2}\left(-\frac{\partial^2 f^{-1}}{\partial \boldsymbol{X}^2}\frac{\partial f}{\partial \boldsymbol{X}}\right)\\ &= f(\boldsymbol{X_0}) + \frac{\partial f^T}{\partial \boldsymbol{X}}\widehat{\boldsymbol{X}} + \frac{1}{2}\frac{\partial f^T}{\partial \boldsymbol{X}}\frac{\partial^2 f^{-T}}{\partial \boldsymbol{X}^2}\frac{\partial^2 f}{\partial \boldsymbol{X}^2}\frac{\partial^2 f^{-1}}{\partial \boldsymbol{X}^2}\frac{\partial f}{\partial \boldsymbol{X}}\\ &= f(\boldsymbol{X_0}) + \frac{\partial f^T}{\partial \boldsymbol{X}}\widehat{\boldsymbol{X}} + \frac{1}{2}\frac{\partial f^T}{\partial \boldsymbol{X}}\frac{\partial^2 f^{-1}}{\partial \boldsymbol{X}^2}\frac{\partial f}{\partial \boldsymbol{X}}\\ &= f(\boldsymbol{X_0}) + \frac{\partial f^T}{\partial \boldsymbol{X}}\widehat{\boldsymbol{X}} + \frac{1}{2}\frac{\partial f^T}{\partial \boldsymbol{X}}\left(-\widehat{\boldsymbol{X}}\right)\\ &= f(\boldsymbol{X_0}) + \frac{1}{2}\frac{\partial f^T}{\partial \boldsymbol{X}}\widehat{\boldsymbol{X}} \end{aligned} \tag{20}$$

对于公式 19 所求得的偏移量如果大于 0.5（只要 x、y 和 σ 任意一个量大于 0.5），则表明插值点已偏移到了它的临近的插值中心，所以必须改变当前的位置，使其为它所偏移到的

插值中心处，然后在新的位置上重新进行泰勒级数插值拟合，直到偏移量小于 0.5 为止（x、y 和 σ 都小于 0.5），这是一个迭代的工程。当然，为了避免无限次的迭代，我们还需要设置一个最大迭代次数，在达到了迭代次数但仍然没有满足偏移量小于 0.5 的情况下，该极值点就要被剔除掉。另外，如果由公式 20 所得到的极值$f(\hat{\boldsymbol{X}})$过小，即$\left|f(\hat{\boldsymbol{X}})\right| < 0.03$时（假设图像的灰度值在 0～1.0 之间），则这样的点易受到噪声的干扰而变得不稳定，所以这些点也应该剔除。而在 opencv 中，使用的是下列公式来判断其是否为不稳定的极值：

$$\left|f(\hat{\boldsymbol{X}})\right| < \frac{T}{s} \tag{21}$$

其中 T 为经验阈值，系统默认初始化为 0.04。

极值点的求取是在 DoG 尺度图像内进行的，DoG 图像的一个特点就是对图像边缘有很强的响应。一旦特征点落在图像的边缘上，这些点就是不稳定的点。这是因为一方面图像边缘上的点是很难定位的，具有定位的歧义性；另一方面这样的点很容易受到噪声的干扰而变得不稳定。因此我们一定要把这些点找到并剔除掉。它的方法与 Harris 角点检测算法相似，即一个平坦的 DoG 响应峰值往往在横跨边缘的地方有较大的主曲率，而在垂直边缘的方向上有较小的主曲率，主曲率可以通过 2×2 的 Hessian 矩阵 $\boldsymbol{H}$ 求出：

$$\boldsymbol{H}(x,y) = \begin{bmatrix} D_{xx}(x,y) & D_{xy}(x,y) \\ D_{xy}(x,y) & D_{yy}(x,y) \end{bmatrix} \tag{22}$$

其中 $D_{xx}(x,y)$、$D_{yy}(x,y)$和 $D_{xy}(x,y)$分别表示对 DoG 图像中的像素在 x 轴方向和 y 轴方向上求二阶偏导和二阶混合偏导。在这里，我们不需要求具体的矩阵 $\boldsymbol{H}$ 的两个特征值——α 和 β，而只要知道两个特征值的比例就可以知道该像素点的主曲率。

矩阵 $\boldsymbol{H}$ 的直迹和行列式分别为：

$$\mathrm{Tr}(\boldsymbol{H}) = D_{xx} + D_{yy} = \alpha + \beta \tag{23}$$

$$\mathrm{Det}(\boldsymbol{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta \tag{24}$$

我们首先剔除掉那些行列式为负数的点，即 $\mathrm{Det}(\boldsymbol{H}) < 0$，因为如果像素的曲率有不同的符号，则该点肯定不是特征点。设 $\alpha > \beta$，并且 $\alpha = \gamma\beta$，其中 $\gamma > 1$，则

$$\frac{\mathrm{Tr}(\boldsymbol{H})^2}{\mathrm{Det}(\boldsymbol{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(\gamma\beta+\beta)^2}{\gamma\beta^2} = \frac{(\gamma+1)^2}{\gamma} \tag{25}$$

上式的结果只与两个特征值的比例有关，而与具体的特征值无关。我们知道，当某个像素的 $\boldsymbol{H}$ 矩阵的两个特征值相差越大，即 γ 很大，则该像素越有可能是边缘。对于公式 25，当两个特征值相等时，等式的值最小，随着 γ 的增加，等式的值也增加。所以，要想检查主曲率的比值是否小于某一阈值 γ，只要检查下式是否成立即可：

$$\frac{\mathrm{Tr}(\boldsymbol{H})}{\mathrm{Det}(\boldsymbol{H})}<\frac{(\gamma+1)^2}{\gamma} \tag{26}$$

对于不满足上式的极值点就不是特征点，因此应该把它们剔除掉。Lowe 给出 γ 为 10。在上面的运算中，需要用到有限差分法求偏导，在这里我们给出具体的公式。为方便起见，我们以图像为例只给出二元函数的实例。与二元函数类似，三元函数的偏导可以很容易的得到。

设 $f(i,j)$是 y 轴为 i、x 轴为 j 的图像像素值，则在(i,j)点处的一阶、二阶及二阶混合偏导为：

$$\frac{\partial f}{\partial x}=\frac{f(i,j+1)-f(i,j-1)}{2h},\quad \frac{\partial f}{\partial y}=\frac{f(i+1,j)-f(i-1,j)}{2h} \tag{27}$$

$$\frac{\partial^2 f}{\partial x^2}=\frac{f(i,j+1)+f(i,j-1)-2f(i,j)}{h^2},\quad \frac{\partial^2 f}{\partial y^2}=\frac{f(i+1,j)+f(i-1,j)-2f(i,j)}{h^2} \tag{28}$$

$$\frac{\partial^2 f}{\partial x\,\partial y}=\frac{f(i-1,j-1)+f(i+1,j+1)-f(i-1,j+1)-f(i+1,j-1)}{4h^2} \tag{29}$$

由于在图像中，相邻像素之间的间隔都是 1，所以这里的 $h=1$。

3、方向角度的确定

经过上面两个步骤，一幅图像的特征点就可以完全找到，而且这些特征点是具有尺度不变性。但为了实现旋转不变性，还需要为特征点分配一个方向角度，也就是需要根据检测到的特征点所在的高斯尺度图像的局部结构求得一个方向基准。该高斯尺度图像的尺度 σ 是已知的，并且该尺度是相对于高斯金字塔所在组的基准层的尺度，也就是公式 10 所表示的尺度。而所谓局部结构指的是在高斯尺度图像中以特征点为中心，以 r 为半径的区域内计算所有像素梯度的幅角和幅值，半径 r 为：

$$r=3\times 1.5\sigma \tag{30}$$

其中 σ 就是上面提到的相对于所在组的基准层的高斯尺度图像的尺度。

像素梯度的幅值和幅角的计算公式为：

$$m(x,y)=\sqrt{(L(x+1,y)-L(x-1,y))^2+(L(x,y+1),L(x,y-1))^2} \tag{31}$$

$$\theta(x,y)=\arctan\left(\frac{L(x,y+1)-L(x,y-1)}{L(x+1,y)-L(x-1,y)}\right) \tag{32}$$

因为在以 r 为半径的区域内的像素梯度幅值对圆心处的特征点的贡献是不同的，因此还需要对幅值进行加权处理，这里采用的是高斯加权，该高斯函数的方差 σ_m 为：

$$\sigma_m = 1.5\sigma \tag{33}$$

其中，公式中的 σ 也就是公式 30 中的 σ。

在完成特征点邻域范围内的梯度计算后，还要应用梯度方向直方图来统计邻域内像素的梯度方向所对应的幅值大小。具体的做法是，把 360°分为 36 个柱，则每 10°为一个柱，即 0°～9°为第 1 柱，10°～19°为第 2 柱，以此类推。在以 r 为半径的区域内，把那些梯度方向在 0°～9°范围内的像素找出来，把它们的加权后的梯度幅值相加在一起，作为第 1 柱的柱高；求第 2 柱以及其他柱的高度的方法相同，不再赘述。为了防止某个梯度方向角度因受到噪声的干扰而突变，我们还需要对梯度方向直方图进行平滑处理。Opencv2.4.9 所使用的平滑公式为：

$$H(i) = \frac{h(i-2) + h(i+2)}{16} + \frac{4 \times (h(i-1) + h(i+1))}{16} + \frac{6 \times h(i)}{16}, \qquad i = 0, \dots, 15 \tag{34}$$

其中 h 和 H 分别表示平滑前和平滑后的直方图。由于角度是循环的，即 0°=360°，如果出现 $h(j)$，j 超出了(0,…,15)的范围，那么可以通过圆周循环的方法找到它所对应的、在 0°～360°之间的值，如 $h(-1) = h(15)$。

这样，直方图的主峰值，即最高的那个柱体所代表的方向就是该特征点处邻域范围内图像梯度的主方向，也就是该特征点的主方向。由于柱体所代表的角度只是一个范围，如第 1 柱的角度为 0°～9°，因此还需要对离散的梯度方向直方图进行插值拟合处理，以得到更精确的方向角度值。例如我们已经得到了第 i 柱所代表的方向为特征点的主方向，则拟合公式为：

$$B = i + \frac{H(i-1) - H(i+1)}{2 \times (H(i-1) + H(i+1) - 2 \times H(i))}, \qquad i = 0, \dots, 15 \tag{35}$$

$$\theta = 360 - 10 \times B \tag{36}$$

其中，H 为由公式 34 得到的直方图，角度 θ 的单位是度。同样的，公式 35 和公式 36 也存在着公式 34 所遇到的角度问题，处理的方法同样还是利用角度的圆周循环。

每个特征点除了必须分配一个主方向外，还可能有一个或更多个辅方向，增加辅方向的目的是为了增强图像匹配的鲁棒性。辅方向的定义是，当存在另一个柱体高度大于主方向柱体高度的 80%时，则该柱体所代表的方向角度就是该特征点的辅方向。

在第 2 步中，我们实现了用两个信息量来表示一个特征点，即位置和尺度。那么经过上面的计算，我们对特征点的表示形式又增加了一个信息量——方向，即 $\boldsymbol{K}(x, y, \sigma, \theta)$。如果某个特征点还有一个辅方向，则这个特征点就要用两个值来表示——$\boldsymbol{K}(x, y, \sigma, \theta_1)$和 $\boldsymbol{K}(x, y, \sigma, \theta_2)$，其中 θ_1 表示主方向，θ_2 表示辅方向，而其他的变量——x, y, σ 不变。

4、特征点描述符生成

通过上面三个步骤的操作，每个特征点被分配了坐标位置、尺度和方向。在图像局部区域内，这些参数可以重复的用来描述局部二维坐标系统，因为这些参数具有不变性。下面就来计算局部图像区域的描述符，描述符既具有可区分性，又具有对某些变量的不变性，如光亮或三维视角。

描述符是与特征点所在的尺度有关的，所以描述特征点是需要在该特征点所在的高斯尺度图像上进行的。在高斯尺度图像上，以特征点为中心，将其附近邻域划分为 $d \times d$ 个子区域（Lowe 取 $d = 4$）。每个子区域都是一个正方形，正方形的边长为 3σ，也就是说正方形的边长有 3σ 个像素点（这里当然要对 3σ 取整）。σ 为相对于特征点所在的高斯金字塔的组的基准层图像的尺度，即公式 10 所表示的尺度。考虑到实际编程的需要，特征点邻域范围的边长应为 $3\sigma(d+1)$，因此特征点邻域区域一共应有 $3\sigma(d+1) \times 3\sigma(d+1)$个像素点。

为了保证特征点具有旋转不变性，还需要以特征点为中心，将上面确定下来的特征点邻域区域旋转 θ（θ 就是该特征点的方向）。由于是对正方形进行旋转，为了使旋转后的区域包括整个正方形，应该以从正方形的中心到它的边的最长距离为半径，也就是正方形对角线长度的一半，即：

$$r = \frac{3\sigma(d+1)\sqrt{2}}{2} \tag{37}$$

所以上述的特征点邻域区域实际应该有$(2r+1) \times (2r+1)$个像素点。由于进行了旋转，则这些采样点的新坐标为：

$$\begin{bmatrix} x` \\ y` \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad x, y \in [-r, r] \tag{38}$$

其中$[x`, y`]^T$为旋转后的像素新坐标，$[x, y]^T$为旋转前的坐标。

这时，我们需要计算旋转以后特征点邻域范围内像素的梯度幅值和梯度幅角。这里的梯度幅值还需要根据其对中心特征点贡献的大小进行加权处理，加权函数仍然采用高斯函数，它的方差的平方为 $d^2/2$。在实际应用中，我们是先以特征点为圆心，以公式 37 中的 r 为半径，计算该圆内所有像素的梯度幅角和高斯加权后的梯度幅值，然后再根据公式 38 得到这些幅值和幅角所对应的像素在旋转以后新的坐标位置。

在计算特征点描述符的时候，我们不需要精确知道邻域内所有像素的梯度幅值和幅角，我们只需要根据直方图知道其统计值即可。这里的直方图是三维直方图，如图 3 所示。

图 3 中的三维直方图为一个立方体，立方体的底就是特征点邻域区域。如前面所述，该区域被分为 4×4 个子区域，即 16 个子区域，邻域内的像素根据其坐标位置，把它们归属于这 16 个子区域中的一个。立方体的三维直方图的高为邻域像素幅角的大小。我们把 360 度的幅角范围进行 8 等分，每一个等份为 45 度。则再根据邻域像素梯度幅角的大小，把它们

归属于这 8 等份中的一份。这样三维直方图就建立了起来，即以特征点为中心的邻域像素根据其坐标位置，以及它的幅角的大小被划归为某个小正方体（如图 4 中的 C 点，在这里，可以通过归一化处理，得到边长都为单位长度的正方体的）内，该直方图一共有 4×4×8=128 个这样的正方体。而这个三维直方图的值则是正方体内所有邻域像素的高斯加权后的梯度幅值之和，所以一共有 128 个值。我们把这 128 个数写成一个 128 维的矢量，该矢量就是该特征点的特征矢量，所有特征点的矢量构成了最终的输入图像的 SIFT 描述符。

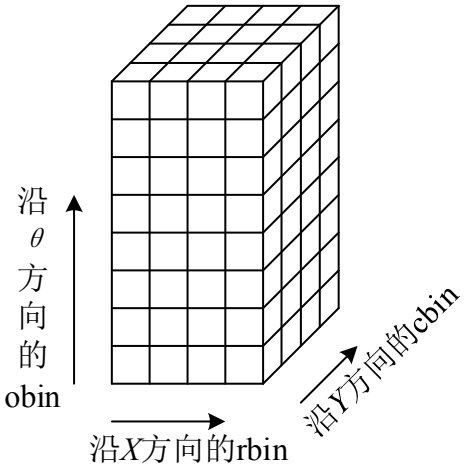

图 3 描述符的三维直方图

显然，正方体的中心应该代表着该正方体。但落入正方体内的邻域像素不可能都在中心，因此我们应该对上面提到的梯度幅值做进一步处理，根据它对中心点位置的贡献大小进行加权处理，即在正方体内，根据像素点相对于正方体中心的距离，对梯度幅值做加权处理。所以三维直方图的值，即正方体的值共需要下面 4 个步骤完成：1、计算落入该正方体内的邻域像素的梯度幅值 A；2、根据该像素相对于特征点的距离，对 A 进行高斯加权处理，得到 B；3、根据该像素相对于它所在的正方体的中心的贡献大小，再对 B 进行加权处理，得到 C；4、对所有落入该正方体内的像素做上述处理，再进行求和运算 ΣC，得到 D。

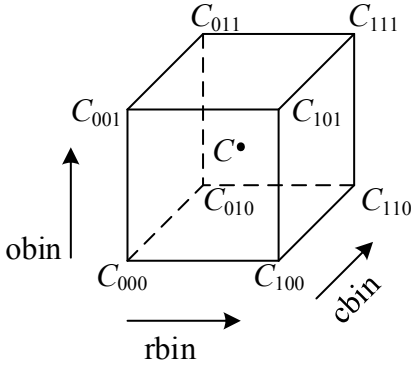

图 4 三维直方图中的正方体

由于计算相对于正方体中心点的贡献大小略显繁琐，因此在实际应用中，我们需要经过坐标平移，把中心点平移到正方体的顶点上，这样只要计算正方体内的点对正方体的 8 个顶

点的贡献大小即可。根据三线性插值法，对某个顶点的贡献值是以该顶点和正方体内的点为对角线的两个顶点，所构成的立方体的体积。如图 4 中，C_{000} 的坐标为(0, 0, 0)，C_{100} 的坐标为(1, 0, 0)，以此类推，C 的坐标为(r, c, θ)，（这里的 r, c, θ 值的大小肯定是在 0 和 1 之间），则 C 点与 8 个顶点所构成的立方体的体积，也就是对 8 个顶点的贡献分别为：

C_{000}：$r\times c\times\theta$，

C_{100}：$(1-r)\times c\times\theta$

C_{010}：$r\times(1-c)\times\theta$

C_{001}：$r\times c\times(1-\theta)$

C_{110}：$(1-r)\times(1-c)\times\theta$

C_{101}：$(1-r)\times c\times(1-\theta)$

C_{011}：$r\times(1-c)\times(1-\theta)$

C_{111}：$(1-r)\times(1-c)\times(1-\theta)$ （39）

经过上面的三维直方图的计算，最终我们得到了该特征点的特征矢量 $P=\{p_1,p_2,\ldots,p_{128}\}$。为了去除光照变化的影响，需要对特征矢量进行归一化处理，即：

$$q_i=\frac{p_i}{\sqrt{\sum_{j=1}^{128}p_j}},\qquad i=1,2,\ldots,128 \tag{40}$$

则 $Q=\{q_1,q_2,\ldots,q_{128}\}$为归一化后的特征矢量。尽管通过归一化处理可以消除对光照变化的影响，但由于照相机饱和以及三维物体表面的不同数量不同角度的光照变化所引起的非线性光照变化仍然存在，它能够影响到一些梯度的相对幅值，但不太会影响梯度幅角。为了消除这部分的影响，我们还需要设一个 $t=0.2$ 的阈值，保留 Q 中小于 0.2 的元素，而把 Q 中大于 0.2 的元素用 0.2 替代。最后再对 Q 进行一次归一化处理，以提高特征点的可区分性。

二、SIFT 源码分析

在 opencv2.4.9 中，SIFT 算法的实现是用类的方法给出的，它在 modules/nonfree/src/sift.c 中实现。

SIFT 的构造函数：

SIFT::SIFT(int nfeatures=0, int nOctaveLayers=3, double contrastThreshold=0.04, double edgeThreshold=10, double sigma=1.6)

nfeatures 表示需要输出的特征点的数量，程序能够通过排序输出最好的前 nfeatures 个特征点，如果 nfeatures=0，则表示输出所有的特征点；

nOctaveLayers 为公式 7 中的参数 s；

contrastThreshold 为公式 21 中的参数 T；

edgeThreshold 为公式 26 中的 γ；

sigma 表示基准层尺度 σ_0。

SIFT 类中的重载运算符( )：

void SIFT::operator()(InputArray img, InputArray mask, vector<KeyPoint>& keypoints, OutputArray descriptors, bool useProvidedKeypoints=false)

img 为输入的 8 位灰度图像；

mask 表示可选的输入掩码矩阵，用来标注需要检测的特征点的区域；

keypoints 为特征点矢量；

descriptors 为输出的特征点描述符矩阵，如果不想得到该值，则需要赋予该值为 cv::noArray()；

useProvidedKeypoints 为二值标识符，默认为 false 时，表示需要计算输入图像的特征点；为 true 时，表示无需特征点检测，而是利用输入的特征点 keypoints 计算它们的描述符。

下面就详细讲解 SIFT 算法的实现。

```
void SIFT::operator()(InputArray _image, InputArray _mask,
                      vector<KeyPoint>& keypoints,
                      OutputArray _descriptors,
                      bool useProvidedKeypoints) const
{
    //定义并初始化一些变量
    //firstOctave 表示金字塔的组索引是从 0 开始还是从-1 开始，-1 表示需要对输入图像的长宽扩大一倍，actualNOctaves 和 actualNLayers 分别表示实际的金字塔的组数和层数
    int firstOctave = -1, actualNOctaves = 0, actualNLayers = 0;
    //得到输入图像和掩码的矩阵
    Mat image = _image.getMat(), mask = _mask.getMat();

    //对输入图像和掩码进行必要的参数确认
    if( image.empty() || image.depth() != CV_8U )
        CV_Error( CV_StsBadArg, "image is empty or has incorrect depth (!=CV_8U)" );

    if( !mask.empty() && mask.type() != CV_8UC1 )
        CV_Error( CV_StsBadArg, "mask has incorrect type (!=CV_8UC1)" );

    //下面 if 语句表示不需要计算图像的特征点，只需要根据输入的特征点 keypoints 参数计算它们的描述符
    if( useProvidedKeypoints )
    {
        //因为不需要扩大输入图像的长宽，所以重新赋值 firstOctave 为 0
        firstOctave = 0;
        //给 maxOctave 赋予一个极小的值
        int maxOctave = INT_MIN;
        //遍历全部的输入特征点，得到最小和最大组索引，以及实际的层数
        for( size_t i = 0; i < keypoints.size(); i++ )
        {
            int octave, layer;      //组索引，层索引
            float scale;      //尺度
            //从输入的特征点变量中提取出该特征点所在的组、层、以及它的尺度
```

```
            unpackOctave(keypoints[i], octave, layer, scale);
            firstOctave = std::min(firstOctave, octave);       //最小组索引号
            maxOctave = std::max(maxOctave, octave);       //最大组索引号
            actualNLayers = std::max(actualNLayers, layer-2);       //实际层数
        }
        //确保最小组索引号不大于 0
        firstOctave = std::min(firstOctave, 0);
        //确保最小组索引号大于等于-1，实际层数小于等于输入参数 nOctaveLayers
        CV_Assert( firstOctave >= -1 && actualNLayers <= nOctaveLayers );
        //计算实际的组数
        actualNOctaves = maxOctave - firstOctave + 1;
    }
    //创建基层图像矩阵 base，详见下面对 createInitialImage 函数的分析
    //createInitialImage 函数的第二个参数表示是否进行扩大输入图像长宽尺寸操作，true 表示进行该操作，第三个参数为基准层尺度 σ0
    Mat base = createInitialImage(image, firstOctave < 0, (float)sigma);
    //gpyr 为高斯金字塔矩阵向量，dogpyr 为 DoG 金字塔矩阵向量
    vector<Mat> gpyr, dogpyr;
    //计算金字塔的组的数量，当 actualNOctaves > 0 时，表示进入了上面的 if( useProvidedKeypoints )语句，所以组数直接等于 if( useProvidedKeypoints )内计算得到的值
    //如果 actualNOctaves 不大于 0，则利用公式 6 计算组数
    //这里面还考虑了组的初始索引等于-1 的情况，所以最后加上了 – firstOctave 这项
    int nOctaves = actualNOctaves > 0 ? actualNOctaves : cvRound(log( (double)std::min( base.cols, base.rows ) ) / log(2.) - 2) - firstOctave;

    //double t, tf = getTickFrequency();
    //t = (double)getTickCount();
    //buildGaussianPyramid 和 buildDoGPyramid 分别为构建高斯金字塔和 DoG 金字塔函数，详见下面的分析
    buildGaussianPyramid(base, gpyr, nOctaves);
    buildDoGPyramid(gpyr, dogpyr);

    //t = (double)getTickCount() - t;
    //printf("pyramid construction time: %g\n", t*1000./tf);
    // useProvidedKeypoints 为 false，表示需要计算图像的特征点
    if( !useProvidedKeypoints )
    {
        //t = (double)getTickCount();
        //在尺度空间内找到极值点，后面给出了 findScaleSpaceExtrema 函数的详细分析
        findScaleSpaceExtrema(gpyr, dogpyr, keypoints);
        //在特征点检测的过程中（尤其是在泰勒级数拟合的过程中）会出现特征点被重复检测到的现象，因此要剔除掉那些重复的特征点
        //KeyPointsFilter 类是在 modules/features2d/src/keypoint.cpp 定义的
        KeyPointsFilter::removeDuplicated( keypoints );
```

```
        //保留那些最好的前 nfeatures 个特征点
        if( nfeatures > 0 )
            KeyPointsFilter::retainBest(keypoints, nfeatures);
        //t = (double)getTickCount() - t;
        //printf("keypoint detection time: %g\n", t*1000./tf);
        //如果 firstOctave < 0，则表示对输入图像进行了扩大处理，所以要对特征点的一些变量进行适当调整。这是因为 firstOctave < 0，金字塔增加了一个第-1 组，而在检测特征点的时候，所有变量都是基于这个第-1 组的基准层尺度图像的。
        if( firstOctave < 0 )
            //遍历所有特征点
            for( size_t i = 0; i < keypoints.size(); i++ )
            {
                KeyPoint& kpt = keypoints[i];       //提取出特征点
                //其实这里的 firstOctave = -1，所以 scale = 0.5
                float scale = 1.f/(float)(1 << -firstOctave);
                //重新调整特征点所在的组
                kpt.octave = (kpt.octave & ~255) | ((kpt.octave + firstOctave) & 255);
                //特征点的位置坐标调整到真正的输入图像内，即得到的坐标除以 2
                kpt.pt *= scale;
                //特征点的尺度调整到相对于输入图像的尺度，即得到的尺度除以 2
                kpt.size *= scale;
            }
        //根据掩码矩阵，只保留掩码矩阵所涵盖的特征点
        if( !mask.empty() )
            KeyPointsFilter::runByPixelsMask( keypoints, mask );
    }
    else
    {
        // filter keypoints by mask
        //KeyPointsFilter::runByPixelsMask( keypoints, mask );
    }
    //如果需要得到特征点描述符，则进入下面的 if 内，生成特征点描述符
    if( _descriptors.needed() )
    {
        //t = (double)getTickCount();
        //dsize 为特征点描述符的大小
        //即 SIFT_DESCR_WIDTH*SIFT_DESCR_WIDTH*SIFT_DESCR_HIST_BINS=4×4×8=128
        int dsize = descriptorSize();
        //创建特征点描述符，为其开辟一段内存空间
        _descriptors.create((int)keypoints.size(), dsize, CV_32F);
        //描述符的矩阵形式
        Mat descriptors = _descriptors.getMat();
        //计算特征点描述符，calcDescriptors 函数在后面将给出详细的分析
        calcDescriptors(gpyr, keypoints, descriptors, nOctaveLayers, firstOctave);
```

```
        //t = (double)getTickCount() - t;
        //printf("descriptor extraction time: %g\n", t*1000./tf);
    }
}
```

创建基图像矩阵 createInitialImage 函数：

```
static Mat createInitialImage( const Mat& img, bool doubleImageSize, float sigma )
{
    Mat gray, gray_fpt;
    //如果输入图像是彩色图像，则需要转换成灰度图像
    if( img.channels() == 3 || img.channels() == 4 )
        cvtColor(img, gray, COLOR_BGR2GRAY);
    else
        img.copyTo(gray);
    //调整图像的像素数据类型
    gray.convertTo(gray_fpt, DataType<sift_wt>::type, SIFT_FIXPT_SCALE, 0);

    float sig_diff;

    if( doubleImageSize )      //需要扩大图像的长宽尺寸
    {
        // SIFT_INIT_SIGMA 为 0.5，即输入图像的尺度，SIFT_INIT_SIGMA×2=1.0，即图像扩大 2 倍以后的尺度，sig_diff 为公式 4 中的高斯函数所需要的方差
        sig_diff = sqrtf( std::max(sigma * sigma - SIFT_INIT_SIGMA * SIFT_INIT_SIGMA * 4, 0.01f) );
        Mat dbl;
        //利用双线性插值法把图像的长宽都扩大 2 倍
        resize(gray_fpt, dbl, Size(gray.cols*2, gray.rows*2), 0, 0, INTER_LINEAR);
        //利用公式 3 对图像进行高斯平滑处理
        GaussianBlur(dbl, dbl, Size(), sig_diff, sig_diff);
        return dbl;      //输出图像矩阵
    }
    else      //不需要扩大图像的尺寸
    {
        // sig_diff 为公式 4 中的高斯函数所需要的方差
        sig_diff = sqrtf( std::max(sigma * sigma - SIFT_INIT_SIGMA * SIFT_INIT_SIGMA, 0.01f) );
        //利用公式 3 对图像进行高斯平滑处理
        GaussianBlur(gray_fpt, gray_fpt, Size(), sig_diff, sig_diff);
        return gray_fpt;              //输出图像矩阵
    }
}
```

构建高斯金字塔 buildGaussianPyramid 函数：

```
void SIFT::buildGaussianPyramid( const Mat& base, vector<Mat>& pyr, int nOctaves ) const
{
    //向量数组 sig 表示每组中计算各层图像所需的方差，nOctaveLayers + 3 即为公式 7
    vector<double> sig(nOctaveLayers + 3);
    //定义高斯金字塔的总层数，nOctaves*(nOctaveLayers + 3)即组数×层数
    pyr.resize(nOctaves*(nOctaveLayers + 3));

    // precompute Gaussian sigmas using the following formula:
    //    \sigma_{total}^2 = \sigma_{i}^2 + \sigma_{i-1}^2
    //提前计算好各层图像所需的方差
    sig[0] = sigma;       //第一层图像的尺度为基准层尺度 σ0
    //由公式 8 计算 k 值
    double k = pow( 2., 1. / nOctaveLayers );
    //遍历所有层，计算方差
    for( int i = 1; i < nOctaveLayers + 3; i++ )
    {
        //由公式 10 计算前一层图像的尺度
        double sig_prev = pow(k, (double)(i-1))*sigma;
        //由公式 10 计算当前层图像的尺度
        double sig_total = sig_prev*k;
        //计算公式 4 中高斯函数所需的方差，并存入 sig 数组内
        sig[i] = std::sqrt(sig_total*sig_total - sig_prev*sig_prev);
    }
    ////遍历高斯金字塔的所有层，构建高斯金字塔
    for( int o = 0; o < nOctaves; o++ )
    {
        for( int i = 0; i < nOctaveLayers + 3; i++ )
        {
            //dst 为当前层图像矩阵
            Mat& dst = pyr[o*(nOctaveLayers + 3) + i];
            //如果当前层为高斯金字塔的第 0 组第 0 层，则直接赋值
            if( o == 0    &&    i == 0 )
                //把由 createInitialImage 函数得到的基层图像矩阵赋予该层
                dst = base;
            // base of new octave is halved image from end of previous octave
            //如果当前层是除了第 0 组以外的其他组中的第 0 层，则要进行降采样处理
            else if( i == 0 )
            {
                //提取出当前层所在组的前一组中的倒数第 3 层图像
                const Mat& src = pyr[(o-1)*(nOctaveLayers + 3) + nOctaveLayers];
                //隔点降采样处理
                resize(src, dst, Size(src.cols/2, src.rows/2),
                       0, 0, INTER_NEAREST);
            }
```

```
            //除了以上两种情况以外的其他情况的处理
            else
            {
                //提取出当前层的前一层图像
                const Mat& src = pyr[o*(nOctaveLayers + 3) + i-1];
                //根据公式 3，由前一层尺度图像得到当前层的尺度图像
                GaussianBlur(src, dst, Size(), sig[i], sig[i]);
            }
        }
    }
}
```

构建 DoG 金字塔 buildDoGPyramid 函数：

```
void SIFT::buildDoGPyramid( const vector<Mat>& gpyr, vector<Mat>& dogpyr ) const
{
    //计算金字塔的组的数量
    int nOctaves = (int)gpyr.size()/(nOctaveLayers + 3);
    //定义 DoG 金字塔的总层数，DoG 金字塔比高斯金字塔每组少一层
    dogpyr.resize( nOctaves*(nOctaveLayers + 2) );
    //遍历 DoG 的所有层，构建 DoG 金字塔
    for( int o = 0; o < nOctaves; o++ )
    {
        for( int i = 0; i < nOctaveLayers + 2; i++ )
        {
            //提取出高斯金字塔的当前层图像
            const Mat& src1 = gpyr[o*(nOctaveLayers + 3) + i];
            //提取出高斯金字塔的上层图像
            const Mat& src2 = gpyr[o*(nOctaveLayers + 3) + i + 1];
            //提取出 DoG 金字塔的当前层图像
            Mat& dst = dogpyr[o*(nOctaveLayers + 2) + i];
            //DoG 金字塔的当前层图像等于高斯金字塔的当前层图像减去高斯金字塔的
上层图像，即公式 5
            subtract(src2, src1, dst, noArray(), DataType<sift_wt>::type);
        }
    }
}
```

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

```
//
// Detects features at extrema in DoG scale space.    Bad features are discarded
// based on contrast and ratio of principal curvatures.
void SIFT::findScaleSpaceExtrema( const vector<Mat>& gauss_pyr, const vector<Mat>& dog_pyr,
                                  vector<KeyPoint>& keypoints ) const
```

```
{
    //得到金字塔的组数
    int nOctaves = (int)gauss_pyr.size()/(nOctaveLayers + 3);
    //SIFT_FIXPT_SCALE = 1，设定一个阈值用于判断在 DoG 尺度图像中像素的大小
    int threshold = cvFloor(0.5 * contrastThreshold / nOctaveLayers * 255 * SIFT_FIXPT_SCALE);
    //SIFT_ORI_HIST_BINS = 36，定义梯度方向的数量
    const int n = SIFT_ORI_HIST_BINS;
    float hist[n];    //定义梯度方向直方图变量
    KeyPoint kpt;

    keypoints.clear();      //清空

    for( int o = 0; o < nOctaves; o++ )
        for( int i = 1; i <= nOctaveLayers; i++ )
        {
            int idx = o*(nOctaveLayers+2)+i;      //DoG 金字塔的当前层索引
            const Mat& img = dog_pyr[idx];      //DoG 金字塔当前层的尺度图像
            const Mat& prev = dog_pyr[idx-1];      //DoG 金字塔下层的尺度图像
            const Mat& next = dog_pyr[idx+1];      //DoG 金字塔上层的尺度图像
            int step = (int)img.step1();
            int rows = img.rows, cols = img.cols;      //图像的长和宽
            //SIFT_IMG_BORDER = 5，该变量的作用是保留一部分图像的四周边界
            for( int r = SIFT_IMG_BORDER; r < rows-SIFT_IMG_BORDER; r++)
            {
                //DoG 金字塔当前层图像的当前行指针
                const sift_wt* currptr = img.ptr<sift_wt>(r);
                //DoG 金字塔下层图像的当前行指针
                const sift_wt* prevptr = prev.ptr<sift_wt>(r);
                //DoG 金字塔上层图像的当前行指针
                const sift_wt* nextptr = next.ptr<sift_wt>(r);

                for( int c = SIFT_IMG_BORDER; c < cols-SIFT_IMG_BORDER; c++)
                {
                    sift_wt val = currptr[c];      //DoG 金字塔当前层尺度图像的像素值

                    // find local extrema with pixel accuracy
                    //精确定位局部极值点
                    //如果满足 if 条件，则找到了极值点，即候选特征点
                    if( std::abs(val) > threshold &&    //像素值要大于一定的阈值才稳定，
即要具有较强的对比度
                        //下面的逻辑判断被“与”分为两个部分，前一个部分要满足像素
值大于 0，在 3×3×3 的立方体内与周围 26 个邻近像素比较找极大值，后一个部分要满足像
素值小于 0，找极小值
                        ((val > 0 && val >= currptr[c-1] && val >= currptr[c+1] &&
```

```
                        val >= currptr[c-step-1] && val >= currptr[c-step] && val >=
currptr[c-step+1] &&
                        val >= currptr[c+step-1] && val >= currptr[c+step] && val >=
currptr[c+step+1] &&
                        val >= nextptr[c] && val >= nextptr[c-1] && val >= nextptr[c+1] &&
                        val >= nextptr[c-step-1] && val >= nextptr[c-step] && val >=
nextptr[c-step+1] &&
                        val >= nextptr[c+step-1] && val >= nextptr[c+step] && val >=
nextptr[c+step+1] &&
                        val >= prevptr[c] && val >= prevptr[c-1] && val >= prevptr[c+1] &&
                        val >= prevptr[c-step-1] && val >= prevptr[c-step] && val >=
prevptr[c-step+1] &&
                        val >= prevptr[c+step-1] && val >= prevptr[c+step] && val >=
prevptr[c+step+1]) ||
                       (val < 0 && val <= currptr[c-1] && val <= currptr[c+1] &&
                        val <= currptr[c-step-1] && val <= currptr[c-step] && val <=
currptr[c-step+1] &&
                        val <= currptr[c+step-1] && val <= currptr[c+step] && val <=
currptr[c+step+1] &&
                        val <= nextptr[c] && val <= nextptr[c-1] && val <= nextptr[c+1] &&
                        val <= nextptr[c-step-1] && val <= nextptr[c-step] && val <=
nextptr[c-step+1] &&
                        val <= nextptr[c+step-1] && val <= nextptr[c+step] && val <=
nextptr[c+step+1] &&
                        val <= prevptr[c] && val <= prevptr[c-1] && val <= prevptr[c+1] &&
                        val <= prevptr[c-step-1] && val <= prevptr[c-step] && val <=
prevptr[c-step+1] &&
                        val <= prevptr[c+step-1] && val <= prevptr[c+step] && val <=
prevptr[c+step+1])))
                    {
                        //三维坐标，长、宽、层（层与尺度相对应）
                        int r1 = r, c1 = c, layer = i;
                        // adjustLocalExtrema 函数的作用是调整局部极值的位置，即找
到亚像素级精度的特征点，该函数在后面给出详细的分析
                        //如果满足 if 条件，说明该极值点不是特征点，继续上面的 for
循环
                        if( !adjustLocalExtrema(dog_pyr, kpt, o, layer, r1, c1,
                                                nOctaveLayers,
(float)contrastThreshold,
                                                (float)edgeThreshold, (float)sigma) )
                            continue;
                        //计算特征点相对于它所在组的基准层的尺度，即公式 10 所得
尺度
                        float scl_octv = kpt.size*0.5f/(1 << o);
```

```
                    // calcOrientationHist 函数为计算特征点的方向角度，后面给出该函数的详细分析
                    //SIFT_ORI_SIG_FCTR = 1.5f ， SIFT_ORI_RADIUS = 3 * SIFT_ORI_SIG_FCTR
                    float omax = calcOrientationHist(gauss_pyr[o*(nOctaveLayers+3) + layer],
                                                     Point(c1, r1),
                                                     cvRound(SIFT_ORI_RADIUS * scl_octv),
                                                     SIFT_ORI_SIG_FCTR * scl_octv,
                                                     hist, n);
                    //SIFT_ORI_PEAK_RATIO = 0.8f
                    //计算直方图辅方向的阈值，即主方向的 80%，
                    float mag_thr = (float)(omax * SIFT_ORI_PEAK_RATIO);
                    //计算特征点的方向
                    for( int j = 0; j < n; j++ )
                    {
                        //j 为直方图当前柱体索引，l 为前一个柱体索引，r2 为后一个柱体索引，如果 l 和 r2 超出了柱体范围，则要进行圆周循环处理
                        int l = j > 0 ? j - 1 : n - 1;
                        int r2 = j < n-1 ? j + 1 : 0;
                        //方向角度拟合处理
                        //判断柱体高度是否大于直方图辅方向的阈值，因为拟合处理的需要，还要满足柱体的高度大于其前后相邻两个柱体的高度
                        if( hist[j] > hist[l] && hist[j] > hist[r2] && hist[j] >= mag_thr )
                        {
                            //公式 35
                            float bin = j + 0.5f * (hist[l]-hist[r2]) / (hist[l] - 2*hist[j] + hist[r2]);
                            //圆周循环处理
                            bin = bin < 0 ? n + bin : bin >= n ? bin - n : bin;
                            //公式 36，得到特征点的方向
                            kpt.angle = 360.f - (float)((360.f/n) * bin);
                            //如果方向角度十分接近于 360 度，则就让它等于 0 度
                            if(std::abs(kpt.angle - 360.f) < FLT_EPSILON)
                                kpt.angle = 0.f;
                            //保存特征点
                            keypoints.push_back(kpt);
                        }
                    }
                }
            }
```

```
            }
        }
}
```

精确找到图像的特征点 adjustLocalExtrema 函数：

```
//
// Interpolates a scale-space extremum's location and scale to subpixel
// accuracy to form an image feature. Rejects features with low contrast.
// Based on Section 4 of Lowe's paper.
//dog_pyr 为 DoG 金字塔，kpt 为特征点，octv 和 layer 为极值点所在的组和组内的层，r 和 c 为极值点的位置坐标
static bool adjustLocalExtrema( const vector<Mat>& dog_pyr, KeyPoint& kpt, int octv,
                                int& layer, int& r, int& c, int nOctaveLayers,
                                float contrastThreshold, float edgeThreshold, float sigma )
{
    // SIFT_FIXPT_SCALE = 1，img_scale 为对图像进行归一化处理的系数
    const float img_scale = 1.f/(255*SIFT_FIXPT_SCALE);
    //公式 27 中分母的倒数
    const float deriv_scale = img_scale*0.5f;
    //公式 28 中分母的倒数
    const float second_deriv_scale = img_scale;
    //公式 29 中分母的倒数
    const float cross_deriv_scale = img_scale*0.25f;

    float xi=0, xr=0, xc=0, contr=0;
    int i = 0;
    // SIFT_MAX_INTERP_STEPS = 5，表示循环迭代 5 次
    for( ; i < SIFT_MAX_INTERP_STEPS; i++ )
    {
        //找到极值点所在的 DoG 金字塔的层索引
        int idx = octv*(nOctaveLayers+2) + layer;
        const Mat& img = dog_pyr[idx];      //当前层尺度图像
        const Mat& prev = dog_pyr[idx-1];      //下层尺度图像
        const Mat& next = dog_pyr[idx+1];      //上层尺度图像
        //变量 dD 就是公式 19 中的∂f / ∂X，一阶偏导的公式如公式 27
        Vec3f dD((img.at<sift_wt>(r, c+1) - img.at<sift_wt>(r, c-1))*deriv_scale,     //f 对 x 的一阶偏导
                 (img.at<sift_wt>(r+1, c) - img.at<sift_wt>(r-1, c))*deriv_scale,     //f 对 y 的一阶偏导
                 (next.at<sift_wt>(r, c) - prev.at<sift_wt>(r, c))*deriv_scale);     //f 对尺度 σ 的一阶偏导

        //当前像素值的 2 倍
        float v2 = (float)img.at<sift_wt>(r, c)*2;
```

```
        //下面是求二阶纯偏导，如公式 28
        //这里的 x，y，s 分别代表 X = (x, y, σ)^T 中的 x，y，σ
        float dxx = (img.at<sift_wt>(r, c+1) + img.at<sift_wt>(r, c-1) - v2)*second_deriv_scale;
        float dyy = (img.at<sift_wt>(r+1, c) + img.at<sift_wt>(r-1, c) - v2)*second_deriv_scale;
        float dss = (next.at<sift_wt>(r, c) + prev.at<sift_wt>(r, c) - v2)*second_deriv_scale;
        //下面是求二阶混合偏导，如公式 29
        float dxy = (img.at<sift_wt>(r+1, c+1) - img.at<sift_wt>(r+1, c-1) -
                        img.at<sift_wt>(r-1, c+1) + img.at<sift_wt>(r-1, c-1))*cross_deriv_scale;
        float dxs = (next.at<sift_wt>(r, c+1) - next.at<sift_wt>(r, c-1) -
                        prev.at<sift_wt>(r, c+1) + prev.at<sift_wt>(r, c-1))*cross_deriv_scale;
        float dys = (next.at<sift_wt>(r+1, c) - next.at<sift_wt>(r-1, c) -
                        prev.at<sift_wt>(r+1, c) + prev.at<sift_wt>(r-1, c))*cross_deriv_scale;
        //变量 H 就是公式 16 中的∂²f / ∂X²，它的具体展开形式可以在公式 15 中看到
        Matx33f H(dxx, dxy, dxs,
                  dxy, dyy, dys,
                  dxs, dys, dss);
        //求方程 A×X=B，即 X=A^-1×B，这里 A 就是 H，B 就是 dD
        Vec3f X = H.solve(dD, DECOMP_LU);
        //可以看出上面求得的 X 与公式 19 求得的变量差了一个符号，因此下面的变量都加上了负号
        xi = -X[2];       //层坐标的偏移量，这里的层与图像尺度相对应
        xr = -X[1];       //纵坐标的偏移量
        xc = -X[0];       //横坐标的偏移量
        //如果由泰勒级数插值得到的三个坐标的偏移量都小于 0.5，说明已经找到特征点，则退出迭代
        if( std::abs(xi) < 0.5f && std::abs(xr) < 0.5f && std::abs(xc) < 0.5f )
            break;
        //如果三个坐标偏移量中任意一个大于一个很大的数，则说明该极值点不是特征点，函数返回
        if( std::abs(xi) > (float)(INT_MAX/3) ||
            std::abs(xr) > (float)(INT_MAX/3) ||
            std::abs(xc) > (float)(INT_MAX/3) )
            return false;       //没有找到特征点，返回
        //由上面得到的偏移量重新定义插值中心的坐标位置
        c += cvRound(xc);
        r += cvRound(xr);
        layer += cvRound(xi);
        //如果新的坐标超出了金字塔的坐标范围，则说明该极值点不是特征点，函数返回
        if( layer < 1 || layer > nOctaveLayers ||
            c < SIFT_IMG_BORDER || c >= img.cols - SIFT_IMG_BORDER    ||
            r < SIFT_IMG_BORDER || r >= img.rows - SIFT_IMG_BORDER )
            return false;       //没有找到特征点，返回
    }
```

```
// ensure convergence of interpolation
//进一步确认是否大于迭代次数，
if( i >= SIFT_MAX_INTERP_STEPS )
    return false;      //没有找到特征点，返回

{
    //由上面得到的层坐标计算它在 DoG 金字塔中的层索引
    int idx = octv*(nOctaveLayers+2) + layer;
    const Mat& img = dog_pyr[idx];      //该层索引所对应的 DoG 金字塔的当前层尺度
图像
    const Mat& prev = dog_pyr[idx-1];      //DoG 金字塔的下层尺度图像
    const Mat& next = dog_pyr[idx+1];      //DoG 金字塔的上层尺度图像
    //再次计算公式 19 中的∂f / ∂X
    Matx31f dD((img.at<sift_wt>(r, c+1) - img.at<sift_wt>(r, c-1))*deriv_scale,
               (img.at<sift_wt>(r+1, c) - img.at<sift_wt>(r-1, c))*deriv_scale,
               (next.at<sift_wt>(r, c) - prev.at<sift_wt>(r, c))*deriv_scale);
    //dD 点乘(xc, xr, xi)，点乘类似于 MATLAB 中的点乘
    //变量 t 就是公式 20 中等号右边的第 2 项内容
    float t = dD.dot(Matx31f(xc, xr, xi));
    //计算公式 20，求极值点处图像的灰度值，即响应值
    contr = img.at<sift_wt>(r, c)*img_scale + t * 0.5f;
    //由公式 21 判断响应值是否稳定
    if( std::abs( contr ) * nOctaveLayers < contrastThreshold )
        return false;      //不稳定的极值，说明没有找到特征点，返回

    // principal curvatures are computed using the trace and det of Hessian
    //边缘极值点的判断
    float v2 = img.at<sift_wt>(r, c)*2.f;      //当前像素灰度值的 2 倍
    //计算矩阵 H 的 4 个元素
    //二阶纯偏导，如公式 28
    float dxx = (img.at<sift_wt>(r, c+1) + img.at<sift_wt>(r, c-1) - v2)*second_deriv_scale;
    float dyy = (img.at<sift_wt>(r+1, c) + img.at<sift_wt>(r-1, c) - v2)*second_deriv_scale;
    //二阶混合偏导，如公式 29
    float dxy = (img.at<sift_wt>(r+1, c+1) - img.at<sift_wt>(r+1, c-1) -
                 img.at<sift_wt>(r-1, c+1) + img.at<sift_wt>(r-1, c-1)) * cross_deriv_scale;
    float tr = dxx + dyy;      //求矩阵的直迹，公式 23
    float det = dxx * dyy - dxy * dxy;      //求矩阵的行列式，公式 24
    //逻辑“或”的前一项表示矩阵的行列式值不能小于 0，后一项如公式 26
    if( det <= 0 || tr*tr*edgeThreshold >= (edgeThreshold + 1)*(edgeThreshold + 1)*det )
        return false;      //不是特征点，返回
}
//保存特征点信息
//特征点对应于输入图像的横坐标位置
kpt.pt.x = (c + xc) * (1 << octv);
```

```
    //特征点对应于输入图像的纵坐标位置
    //需要注意的是，这里的输入图像特指实际的输入图像扩大一倍以后的图像，因为这里的 octv 是包括了前面理论分析部分提到的金字塔的第-1 组
    kpt.pt.y = (r + xr) * (1 << octv);
    //按一定格式保存特征点所在的组、层以及插值后的层的偏移量
    kpt.octave = octv + (layer << 8) + (cvRound((xi + 0.5)*255) << 16);
    //特征点相对于输入图像的尺度，即公式 13
    //同样的，这里的输入图像也是特指实际的输入图像扩大一倍以后的图像，所以下面的语句其实是比公式 13 多了一项——乘以 2
    kpt.size = sigma*powf(2.f, (layer + xi) / nOctaveLayers)*(1 << octv)*2;
    //特征点的响应值
    kpt.response = std::abs(contr);

    return true;      //是特征点，返回
}
```

计算特征点的方向角度 calcOrientationHist 函数：

```
// Computes a gradient orientation histogram at a specified pixel
//img 为特征点所在的高斯尺度图像；
//pt 为特征点在该尺度图像的坐标点；
//radius 为邻域半径，即公式 30；
//sigma 为高斯函数的方差，即公式 33；
//hist 为梯度方向直方图；
//n 为梯度方向直方图柱体的数量，n=36
//该函数返回直方图的主峰值
static float calcOrientationHist( const Mat& img, Point pt, int radius,
                                  float sigma, float* hist, int n )
{
    //len 为计算特征点方向时的特征点领域像素的数量
    int i, j, k, len = (radius*2+1)*(radius*2+1);
    // expf_scale 为高斯加权函数中 e 指数中的常数部分
    float expf_scale = -1.f/(2.f * sigma * sigma);
    //分配一段内存空间
    AutoBuffer<float> buf(len*4 + n+4);
    //X 表示 x 轴方向的差分，Y 表示 y 轴方向的差分，Mag 为梯度幅值，Ori 为梯度幅角，W 为高斯加权值
    //上述变量在 buf 空间的分配是：X 和 Mag 共享一段长度为 len 的空间，Y 和 Ori 分别占用一段长度为 len 的空间，W 占用一段长度为 len+2 的空间，它们的空间顺序为 X（Mag）在 buf 的最下面，然后是 Y，Ori，最后是 W
    float *X = buf, *Y = X + len, *Mag = X, *Ori = Y + len, *W = Ori + len;
    //temphist 表示暂存的梯度方向直方图，空间长度为 n+2，空间位置是在 W 的上面
    //之所以 temphist 的长度是 n+2，W 的长度是 len+2，而不是 n 和 len，是因为要进行圆周循环操作，必须给 temphist 的前后各留出两个空间位置
    float* temphist = W + len + 2;
```

```
//直方图变量清空
for( i = 0; i < n; i++ )
    temphist[i] = 0.f;
//计算 x 轴、y 轴方向的导数，以及高斯加权值
for( i = -radius, k = 0; i <= radius; i++ )
{
    int y = pt.y + i;       //邻域像素的 y 轴坐标
    //判断 y 轴坐标是否超出图像的范围
    if( y <= 0 || y >= img.rows - 1 )
        continue;
    for( j = -radius; j <= radius; j++ )
    {
        int x = pt.x + j;       //邻域像素的 x 轴坐标
        //判断 x 轴坐标是否超出图像的范围
        if( x <= 0 || x >= img.cols - 1 )
            continue;
        //分别计算 x 轴和 y 轴方向的差分，即公式 27 的分子部分，因为只需要相对值，所有分母部分可以不用计算
        float dx = (float)(img.at<sift_wt>(y, x+1) - img.at<sift_wt>(y, x-1));
        float dy = (float)(img.at<sift_wt>(y-1, x) - img.at<sift_wt>(y+1, x));
        //保存变量，这里的 W 为高斯函数的 e 指数
        X[k] = dx; Y[k] = dy; W[k] = (i*i + j*j)*expf_scale;
        k++;       //邻域像素的计数值加 1
    }
}
//这里的 len 为特征点实际的邻域像素的数量
len = k;

// compute gradient values, orientations and the weights over the pixel neighborhood
//计算邻域中所有元素的高斯加权值 W，梯度幅角 Ori 和梯度幅值 Mag
exp(W, W, len);
fastAtan2(Y, X, Ori, len, true);
magnitude(X, Y, Mag, len);
//计算梯度方向直方图
for( k = 0; k < len; k++ )
{
    //判断邻域像素的梯度幅角属于 36 个柱体的哪一个
    int bin = cvRound((n/360.f)*Ori[k]);
    //如果超出范围，则利用圆周循环确定其真正属于的那个柱体
    if( bin >= n )
        bin -= n;
    if( bin < 0 )
        bin += n;
    //累积经高斯加权处理后的梯度幅值
```

```
            temphist[bin] += W[k]*Mag[k];
        }

        // smooth the histogram
        //平滑直方图
        //为了圆周循环，提前填充好直方图前后各两个变量
        temphist[-1] = temphist[n-1];
        temphist[-2] = temphist[n-2];
        temphist[n] = temphist[0];
        temphist[n+1] = temphist[1];
        for( i = 0; i < n; i++ )
        {
            //利用公式 34，进行平滑直方图操作
            hist[i] = (temphist[i-2] + temphist[i+2])*(1.f/16.f) +
                (temphist[i-1] + temphist[i+1])*(4.f/16.f) +
                temphist[i]*(6.f/16.f);
        }
        //计算直方图的主峰值
        float maxval = hist[0];
        for( i = 1; i < n; i++ )
            maxval = std::max(maxval, hist[i]);

        return maxval;      //返回直方图的主峰值
}
```

计算特征点描述符 calcDescriptors 函数：

```
static void calcDescriptors(const vector<Mat>& gpyr, const vector<KeyPoint>& keypoints,
                            Mat& descriptors, int nOctaveLayers, int firstOctave )
{
    //SIFT_DESCR_WIDTH = 4，SIFT_DESCR_HIST_BINS = 8
    int d = SIFT_DESCR_WIDTH, n = SIFT_DESCR_HIST_BINS;
    //遍历所有特征点
    for( size_t i = 0; i < keypoints.size(); i++ )
    {
        KeyPoint kpt = keypoints[i];      //当前特征点
        int octave, layer;      // octave 为组索引，layer 为层索引
        float scale;      //尺度
        //从特征点结构变量中分离出该特征点所在的组、层以及它的尺度
        //一般情况下，这里的尺度 scale = 2^-o，o 表示特征点所在的组，即 octave 变量
        unpackOctave(kpt, octave, layer, scale);
        //确保组和层在合理的范围内
        CV_Assert(octave >= firstOctave && layer <= nOctaveLayers+2);
        //得到当前特征点相对于它所在高斯金字塔的组的基准层尺度图像的尺度，即公式
10
```

```
            //特征点变量保存的尺度是相对于输入图像的尺度，即公式 12
            //公式 12 的表达式乘以 2^-o 就得到了公式 10 的表达式
            float size=kpt.size*scale;
            //得到当前特征点所在的高斯尺度图像的位置坐标，具体原理与上面得到的尺度相
类似
            Point2f ptf(kpt.pt.x*scale, kpt.pt.y*scale);
            //得到当前特征点所在的高斯尺度图像矩阵
            const Mat& img = gpyr[(octave - firstOctave)*(nOctaveLayers + 3) + layer];
            //得到当前特征点的方向角度
            float angle = 360.f - kpt.angle;
            //如果方向角度十分接近于 360 度，则就让它等于 0 度
            if(std::abs(angle - 360.f) < FLT_EPSILON)
                angle = 0.f;
            //计算特征点的特征矢量，calcSIFTDescriptor 函数详见下面的分析
            // size*0.5f，这里尺度又除以 2，可能是为了减小运算量（不是很确定）
            calcSIFTDescriptor(img, ptf, angle, size*0.5f, d, n, descriptors.ptr<float>((int)i));
        }
    }
```

计算 SIFT 算法中特征点的特征矢量 calcSIFTDescriptor 函数：

```
static void calcSIFTDescriptor( const Mat& img, Point2f ptf, float ori, float scl,
                                int d, int n, float* dst )
{
    Point pt(cvRound(ptf.x), cvRound(ptf.y));      //特征点的位置坐标
    //特征点方向的余弦和正弦，即 cosθ 和 sinθ
    float cos_t = cosf(ori*(float)(CV_PI/180));
    float sin_t = sinf(ori*(float)(CV_PI/180));
    //n = 8，45 度的倒数
    float bins_per_rad = n / 360.f;
    //高斯加权函数中的 e 指数的常数部分
    float exp_scale = -1.f/(d * d * 0.5f);
    //SIFT_DESCR_SCL_FCTR = 3.f，即 3σ
    float hist_width = SIFT_DESCR_SCL_FCTR * scl;
    //特征点邻域区域的半径，即公式 37
    int radius = cvRound(hist_width * 1.4142135623730951f * (d + 1) * 0.5f);
    // Clip the radius to the diagonal of the image to avoid autobuffer too large exception
    //避免邻域过大
    radius = std::min(radius, (int) sqrt((double) img.cols*img.cols + img.rows*img.rows));
    //归一化处理
    cos_t /= hist_width;
    sin_t /= hist_width;
    //len 为特征点邻域区域内像素的数量，histlen 为直方图的数量，即特征矢量的长度，
实际应为 d×d×n，之所以每个变量又加上了 2，是因为要为圆周循环留出一定的内存空间
    int i, j, k, len = (radius*2+1)*(radius*2+1), histlen = (d+2)*(d+2)*(n+2);
```

```
int rows = img.rows, cols = img.cols;       //特征点所在的尺度图像的长和宽
//开辟一段内存空间
AutoBuffer<float> buf(len*6 + histlen);
```

//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，它们在 buf 的顺序为 X 在最下面，然后是 Y（Mag），Ori，最后是 W

```
float *X = buf, *Y = X + len, *Mag = Y, *Ori = Mag + len, *W = Ori + len;
```

//下面是三维直方图的变量，RBin 和 CBin 分别表示 d×d 邻域范围的横、纵坐标，hist 表示直方图的值。RBin 和 CBin 的长度都是 len，hist 的长度为 histlen，顺序为 RBin 在 W 的上面，然后是 CBin，最后是 hist。

```
float *RBin = W + len, *CBin = RBin + len, *hist = CBin + len;
//直方图数组 hist 清零
for( i = 0; i < d+2; i++ )
{
    for( j = 0; j < d+2; j++ )
        for( k = 0; k < n+2; k++ )
            hist[(i*(d+2) + j)*(n+2) + k] = 0.;
}
//遍历当前特征点的邻域范围
for( i = -radius, k = 0; i <= radius; i++ )
    for( j = -radius; j <= radius; j++ )
    {
        // Calculate sample's histogram array coords rotated relative to ori.
        // Subtract 0.5 so samples that fall e.g. in the center of row 1 (i.e.
        // r_rot = 1.5) have full weight placed in row 1 after interpolation.
        //根据公式 38 计算旋转后的位置坐标
        float c_rot = j * cos_t - i * sin_t;
        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，以便后面的使用。因为变量 cos_t 和 sin_t 都已进行了归一化处理，所以原点位移时只需要加 d/2 即可。而再减 0.5f 的目的是进行坐标平移，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。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，无需平移

```
        float rbin = r_rot + d/2 - 0.5f;
        float cbin = c_rot + d/2 - 0.5f;
        //得到邻域像素点的位置坐标
        int r = pt.y + i, c = pt.x + j;
        //确定邻域像素是否在 d×d 的正方形内，以及是否超过了图像边界
        if( rbin > -1 && rbin < d && cbin > -1 && cbin < d &&
            r > 0 && r < rows - 1 && c > 0 && c < cols - 1 )
        {
```

//根据公式 27 计算 x 和 y 方向的一阶导数，这里省略了公式中的分母部分，因为没有分母部分不影响后面所进行的归一化处理

```
            float dx = (float)(img.at<sift_wt>(r, c+1) - img.at<sift_wt>(r, c-1));
```

```
                    float dy = (float)(img.at<sift_wt>(r-1, c) - img.at<sift_wt>(r+1, c));
                    //保存到各自的数组中
                    X[k] = dx; Y[k] = dy; RBin[k] = rbin; CBin[k] = cbin;
                    //高斯加权函数中的 e 指数部分
                    W[k] = (c_rot * c_rot + r_rot * r_rot)*exp_scale;
                    k++;      //统计实际的邻域像素的数量
                }
            }

        len = k;      //赋值
        fastAtan2(Y, X, Ori, len, true);      //计算梯度幅角
        magnitude(X, Y, Mag, len);      //计算梯度幅值
        exp(W, W, len);      //计算高斯加权函数
        //遍历所有邻域像素
        for( k = 0; k < len; k++ )
        {
            //得到 d×d 邻域区域的坐标，即三维直方图的底内的位置
            float rbin = RBin[k], cbin = CBin[k];
            //得到幅角所属的 8 等份中的某一个等份，即三维直方图的高的位置
            float obin = (Ori[k] - ori)*bins_per_rad;
            float mag = Mag[k]*W[k];      //得到高斯加权以后的梯度幅值
            //向下取整
            //r0，c0 和 o0 为三维坐标的整数部分，它表示在图 3 中属于的哪个正方体
            int r0 = cvFloor( rbin );
            int c0 = cvFloor( cbin );
            int o0 = cvFloor( obin );
            //小数部分
            //rbin，cbin 和 obin 为三维坐标的小数部分，即在图 4 中 C 点在正方体的坐标
            rbin -= r0;
            cbin -= c0;
            obin -= o0;
            //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
            if( o0 < 0 )
                o0 += n;
            if( o0 >= n )
                o0 -= n;

            // histogram update using tri-linear interpolation
            //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，即公式 39 中
得到的 8 个立方体的体积，当然这里还需要乘以高斯加权后的梯度值 mag
            float v_r1 = mag*rbin, v_r0 = mag - v_r1;
            float v_rc11 = v_r1*cbin, v_rc10 = v_r1 - v_rc11;
            float v_rc01 = v_r0*cbin, v_rc00 = v_r0 - v_rc01;
```

```
        float v_rco111 = v_rc11*obin, v_rco110 = v_rc11 - v_rco111;
        float v_rco101 = v_rc10*obin, v_rco100 = v_rc10 - v_rco101;
        float v_rco011 = v_rc01*obin, v_rco010 = v_rc01 - v_rco011;
        float v_rco001 = v_rc00*obin, v_rco000 = v_rc00 - v_rco001;
        //得到该像素点在三维直方图中的索引
        int idx = ((r0+1)*(d+2) + c0+1)*(n+2) + o0;
        //8 个顶点对应于坐标平移前的 8 个直方图的正方体，对其进行累加求和
        hist[idx] += v_rco000;
        hist[idx+1] += v_rco001;
        hist[idx+(n+2)] += v_rco010;
        hist[idx+(n+3)] += v_rco011;
        hist[idx+(d+2)*(n+2)] += v_rco100;
        hist[idx+(d+2)*(n+2)+1] += v_rco101;
        hist[idx+(d+3)*(n+2)] += v_rco110;
        hist[idx+(d+3)*(n+2)+1] += v_rco111;
    }

    // finalize histogram, since the orientation histograms are circular
    //由于圆周循环的特性，对计算以后幅角小于 0 度或大于 360 度的值重新进行调整，使其在 0～360 度之间
    for( i = 0; i < d; i++ )
        for( j = 0; j < d; j++ )
        {
            int idx = ((i+1)*(d+2) + (j+1))*(n+2);
            hist[idx] += hist[idx+n];
            hist[idx+1] += hist[idx+n+1];
            for( k = 0; k < n; k++ )
                dst[(i*d + j)*n + k] = hist[idx+k];
        }
    // copy histogram to the descriptor,
    // apply hysteresis thresholding
    // and scale the result, so that it can be easily converted
    // to byte array
    float nrm2 = 0;
    len = d*d*n;       //特征矢量的维数——128
    for( k = 0; k < len; k++ )
        nrm2 += dst[k]*dst[k];       //平方和
    // SIFT_DESCR_MAG_THR = 0.2f
    //为了避免计算中的累加误差，对光照阈值进行反归一化处理，即 0.2 乘以公式 40 中的分母部分，得到反归一化阈值 thr
    float thr = std::sqrt(nrm2)*SIFT_DESCR_MAG_THR;
    for( i = 0, nrm2 = 0; i < k; i++ )
    {
        //把特征矢量中大于反归一化阈值 thr 的元素用 thr 替代
```

```
            float val = std::min(dst[i], thr);
            dst[i] = val;
            nrm2 += val*val;        //平方和
        }
        //SIFT_INT_DESCR_FCTR = 512.f，浮点型转换为整型时所用到的系数
        //归一化处理，计算公式 40 中的分母部分
        nrm2 = SIFT_INT_DESCR_FCTR/std::max(std::sqrt(nrm2), FLT_EPSILON);

#if 1
        for( k = 0; k < len; k++ )
        {
            dst[k] = saturate_cast<uchar>(dst[k]*nrm2);        //最终归一化后的特征矢量
        }
#else
        float nrm1 = 0;
        for( k = 0; k < len; k++ )
        {
            dst[k] *= nrm2;
            nrm1 += dst[k];
        }
        nrm1 = 1.f/std::max(nrm1, FLT_EPSILON);
        for( k = 0; k < len; k++ )
        {
            dst[k] = std::sqrt(dst[k] * nrm1);//saturate_cast<uchar>(std::sqrt(dst[k] *
nrm1)*SIFT_INT_DESCR_FCTR);
        }
#endif
}
```

三、SIFT 应用实例

下面我们就给出具体的应用实例。

首先给出的是特征点的检测：

```
#include "opencv2/core/core.hpp"
#include "highgui.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/nonfree/nonfree.hpp"

using namespace cv;
//using namespace std;

int main(int argc, char** argv)
```

```
{
    Mat img = imread("box_in_scene.png");

    SIFT sift;      //实例化 SIFT 类

    vector<KeyPoint> key_points;      //特征点
    // descriptors 为描述符，mascara 为掩码矩阵
    Mat descriptors, mascara;
    Mat output_img;      //输出图像矩阵

    sift(img,mascara,key_points,descriptors);      //执行 SIFT 运算
    //在输出图像中绘制特征点
    drawKeypoints(img,      //输入图像
        key_points,      //特征点矢量
        output_img,      //输出图像
        Scalar::all(-1),      //绘制特征点的颜色，为随机
        //以特征点为中心画圆，圆的半径表示特征点的大小，直线表示特征点的方向
        DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

    namedWindow("SIFT");
    imshow("SIFT", output_img);
    waitKey(0);

    return 0;
}
```

结果如下图所示：

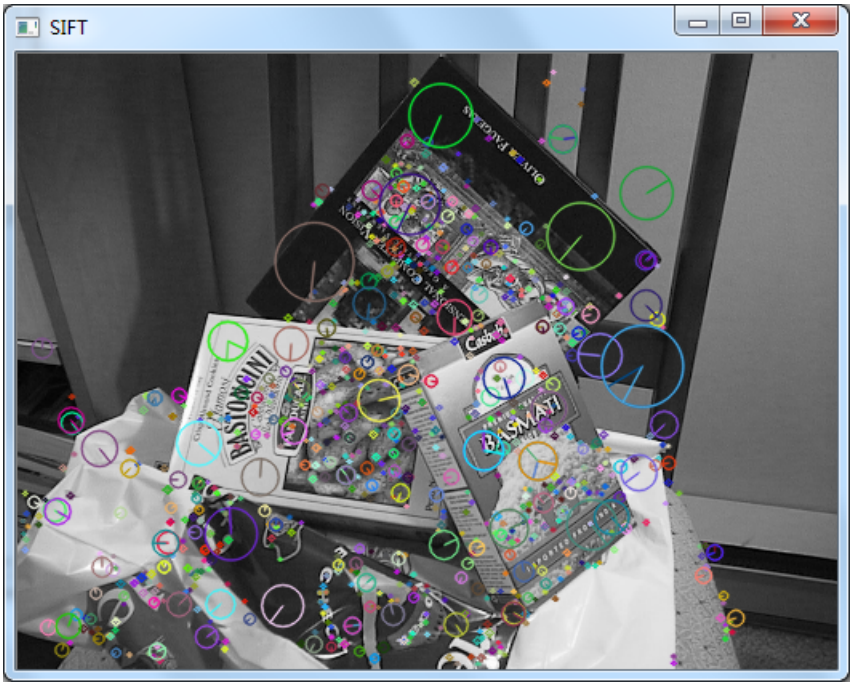

图 5 SIFT 特征点检测结果

上面的程序需要说明一点的是，如果需要改变 SIFT 算法的默认参数，可以通过实例化 SIFT 类的时候更改，例如我们只想检测 20 个最佳的特征点，则实例化 SIFT 的语句为：

```
SIFT sift(20);
```

下面给出利用描述符进行图像匹配的实例：

```
#include "opencv2/core/core.hpp"
#include "highgui.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/legacy/legacy.hpp"

using namespace cv;
using namespace std;

int main(int argc, char** argv)
{
    //待匹配的两幅图像，其中 img1 包括 img2，也就是要从 img1 中识别出 img2
    Mat img1 = imread("box_in_scene.png");
    Mat img2 = imread("box.png");

    SIFT sift1, sift2;

    vector<KeyPoint> key_points1, key_points2;

    Mat descriptors1, descriptors2, mascara;

    sift1(img1,mascara,key_points1,descriptors1);
    sift2(img2,mascara,key_points2,descriptors2);

    //实例化暴力匹配器——BruteForceMatcher
    BruteForceMatcher<L2<float>> matcher;
    //定义匹配器算子
    vector<DMatch>matches;
    //实现描述符之间的匹配，得到算子 matches
    matcher.match(descriptors1,descriptors2,matches);

    //提取出前 30 个最佳匹配结果
    std::nth_element(matches.begin(),       //匹配器算子的初始位置
        matches.begin()+29,       // 排序的数量
        matches.end());           // 结束位置
    //剔除掉其余的匹配结果
    matches.erase(matches.begin()+30, matches.end());
```

```
    namedWindow("SIFT_matches");
    Mat img_matches;
    //在输出图像中绘制匹配结果
    drawMatches(img1,key_points1,              //第一幅图像和它的特征点
        img2,key_points2,          //第二幅图像和它的特征点
        matches,           //匹配器算子
        img_matches,          //匹配输出图像
        Scalar(255,255,255));        //用白色直线连接两幅图像中的特征点
    imshow("SIFT_matches",img_matches);
    waitKey(0);

    return 0;
}
```

结果如下图所示：

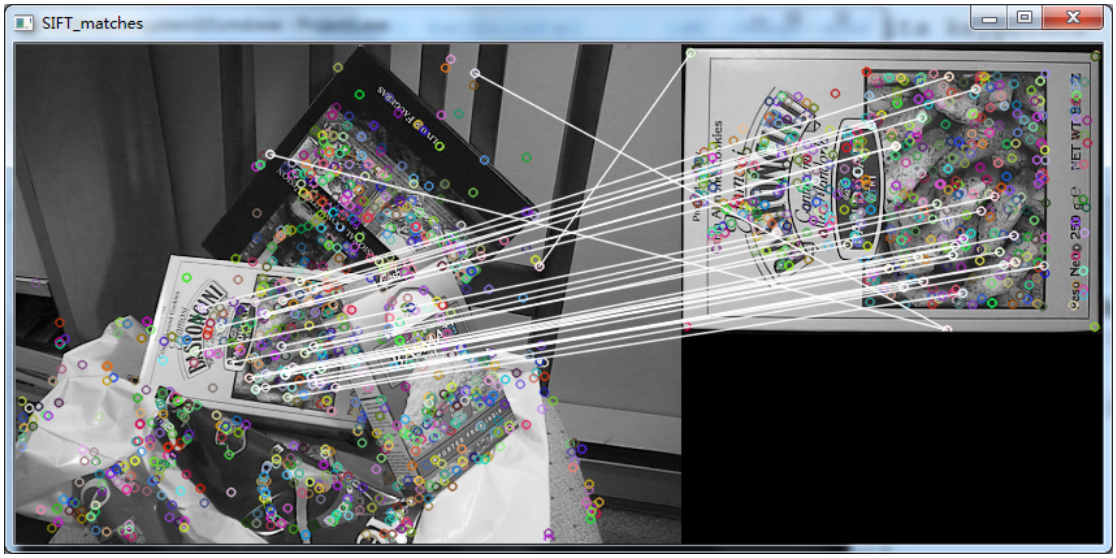

图 6 SIFT 匹配结果

程序是通过距离测度实现两幅图像描述符之间的比较的，距离越小，匹配性越好，越说明这两个描述符表示的是同一事物。描述符的匹配结果保存在匹配器算子 matches 中。如果直接使用 matches，匹配效果并不好，因为它是尽可能的匹配所有的描述符。因此我们要进行筛选，只保留那些好的结果。在这里，我们利用排序，选择距离最小的前 30 个匹配结果，并进行输出。另外，matcher.match 函数中，两个描述符的顺序一定不能写反，否则运行会出错。