Opencv2.4.9 源码分析——SURF

赵春江

blog.csdn.net/zhaocj

一、SURF 算法

SURF (Speeded Up Robust Features)是一种具有鲁棒性的局部特征检测算法，它首先由 Herbert Bay 等人于 2006 年提出，并在 2008 年进行了完善。其实该算法是 Herbert Bay 在博士期间的研究内容，并作为博士毕业论文的一部分发表。

SURF 算法的部分灵感来自于 SIFT 算法，但正如它的名字一样，该算法除了具有重复性高的检测器和可区分性好的描述符特点外，还具有很强的鲁棒性以及更高的运算速度，如 Bay 所述，SURF 至少比 SIFT 快 3 倍以上，综合性能要优于 SIFT 算法。与 SIFT 算法一样，SURF 算法也在美国申请了专利。

之所以 SURF 算法有如此优异的表现，尤其是在效率上，是因为该算法一方面在保证正确性的前提下进行了适当的简化和近似，另一方面它多次运用积分图像（integral image）的概念。

在讲解 SURF 算法之前，我们先来介绍一下积分图像。

积分图像很早就被应用在计算机图形学中，但直到 2001 年才由 Viola 和 Jones 应用到计算机视觉领域中。积分图像 $I_\Sigma(x,y)$的大小尺寸与原图像 $I(x,y)$的大小尺寸相等，而积分图像在(x, y)处的值等于原图像中横坐标小于等于 x 并且纵坐标也小于等于 y 的所有像素灰度值之和，也就是在原图像中，从其左上角到(x,y)处所构成的矩形区域内所有像素灰度值之和，即

$$I_\Sigma(x,y)=\sum_{i=0}^{i\le x}\sum_{j=0}^{j\le 0} I(x,y) \tag{1}$$

事实上，积分图像的计算十分简单，只需要对原图像进行一次扫描，就可以得到一幅完整的积分图像，它的计算公式有几种，下列公式是其中的一种：

$$I_\Sigma(x,y)=I(x,y)+I_\Sigma(x-1,y)+I_\Sigma(x,y-1)-I_\Sigma(x-1,y-1) \tag{2}$$

其中，$I_\Sigma(x\text{-}1,y)$、$I_\Sigma(x,y\text{-}1)$和 $I_\Sigma(x\text{-}1,y\text{-}1)$都是在计算 $I_\Sigma(x,y)$之前得到的值。

利用积分图像可以计算原图像中任意矩形内像素灰度值之和。如图 1 所示，某图像 $I(x,y)$中有四个点，它们的坐标分别为 A=(x_0,y_0)，B=(x_1,y_0)，C=(x_0,y_1)和 D=(x_1,y_1)。由这 4 个点组成了矩阵 W，该 W 内的像素灰度之和为：

$$\sum_{\substack{x_0<x\le x_1\\ y_0<y\le y_1}} I(x,y) = I_\Sigma(\mathrm{D}) + I_\Sigma(\mathrm{A}) - I_\Sigma(\mathrm{B}) - I_\Sigma(\mathrm{C}) \tag{3}$$

其中，I_Σ 为 I 的积分图像。

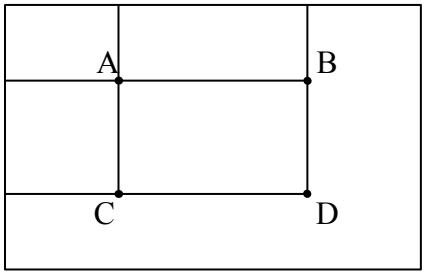

图 1 积分图像求矩阵内灰度值之和

由公式 3 可知，一旦图像的积分图像确定下来，图像中任意矩阵区域内的灰度值之和就可以很容易的得到。更重要的是，无论矩阵面积多大，所需要的运算量都是相同的。因此当算法中需要大量重复的计算不同矩阵区域内的灰度值之和时，应用积分图像就可以大大地提高效率。SURF 算法正是很好的利用了这个性质，以近乎恒定的时间完成了不同尺寸大小的盒状滤波器（box filter）的快速卷积运算。

积分图像就介绍到这里，下面进入主题，重点讲解 SURF 算法。

SURF 算法包括下面几个阶段：

第一部分：特征点检测

1、基于 Hessian 矩阵的特征点检测

2、尺度空间表示

3、特征点定位

第二部分：特征点描述

1、方向角度的分配

2、基于 Haar 小波的特征点描述符

基于 Hessian 矩阵的特征点检测

关于特征点，还没有一个统一的定义，但 Bay 认为，特征点是那些在两个不同方向上局部梯度有着剧烈变化的小的图像区域。因此角点、斑点和 T 型连接都可以被认为是特征点。目前检测特征点最好的方法是基于 Harris 矩阵的方法和基于 Hessian 矩阵的方法，Harris 矩阵能够检测出角点类特征点，Hessian 矩阵能够检测出斑点类特征点。SURF 算法应用的是 Hessian 矩阵，这是因为该矩阵在运算速度和特征点检测的准确率上都具有一定的优势。Hessian 矩阵检测特征点的方法是计算图像所有像素的 Hessian 矩阵的行列式，极值点处就

是图像特征点所在的位置。由于在特征点检测的过程中采取了一系列加速运算量的方法，因此 SURF 算法中的特征点检测方法也称为 Fast-Hessian 方法。

无论是 Harris 方法还是 Hessian 方法，都无法实现尺度的不变性。LoG（高斯拉普拉斯方法，Laplacian of Gaussian）是最好的能够保证尺度不变性的方法。Mikolajczyk 和 Schmid 把 Harris 和 LoG 相结合，提出了 Harris-Laplace 方法，从而实现了 Harris 方法的尺度不变性。仿照 Harris-Laplace 方法，SURF 算法把 Hessian 和 LoG 结合，提出了 Hessian-Laplace 方法，也同样保证了用 Hessian 方法所检测到的特征点的尺度不变性。

给定图像 I 中的某点 $\mathbf{x}=(x, y)$，在该点 $\mathbf{x}$ 处，尺度为 σ 的 Hessian 矩阵 $\boldsymbol{H}(\mathbf{x}, \sigma)$定义为：

$$\boldsymbol{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix} \tag{4}$$

其中，$L_{xx}(\mathbf{x}, \sigma)$是高斯二阶微分$\frac{\partial^2 g(\sigma)}{\partial \mathbf{x}^2}$在点 $\mathbf{x}=(x, y)$处与图像 I 的卷积，$L_{xy}(\mathbf{x}, \sigma)$和 $L_{yy}(\mathbf{x}, \sigma)$具有相似的含义。这些微分就是 LoG。

Bay 指出，高斯函数虽然是最佳的尺度空间分析工具，但由于在实际应用时总是要对高斯函数进行离散化和剪裁处理，从而损失了一些特性（如重复性）。这一因素为我们用其他工具代替高斯函数对尺度空间的分析提供了可能，因为既然高斯函数会带来误差，那么其他工具所带来的误差也可以被忽略，只要误差不大就可以。况且，SIFT 利用 DoG 近似 LoG 的成功经验也进一步验证了高斯函数的可替代性。

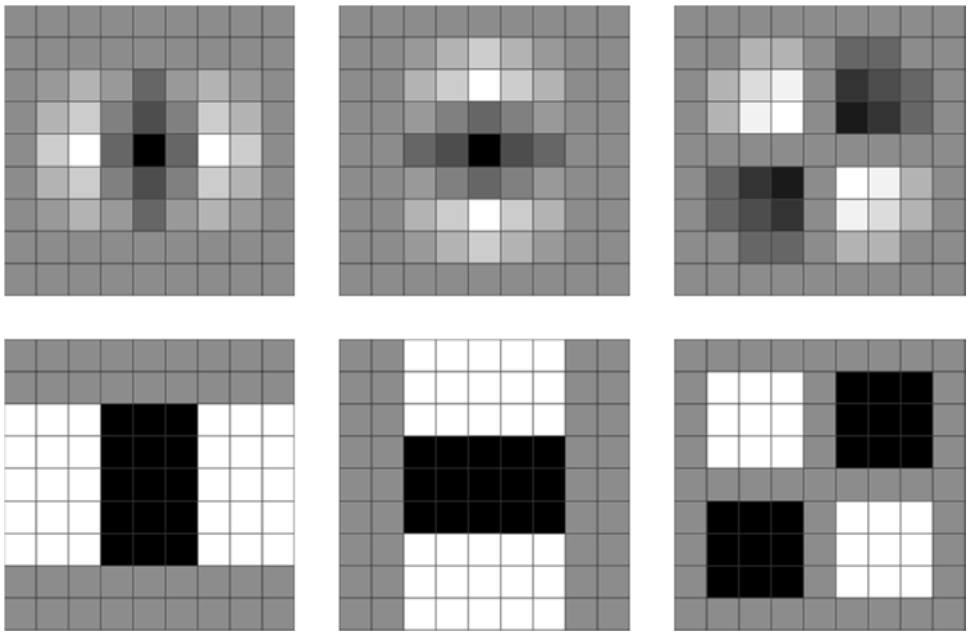

图2 LoG的近似

如图 2 所示，第一行图像就是经过离散化，并被裁减为 9×9 方格，σ = 1.2 的沿 x 方向、y 方向和 xy 方向的高斯二阶微分算子，即 L_{xx} 模板、L_{yy} 模板、L_{xy} 模板。这些微分算子可以

用加权后的 9×9 盒状滤波器——D_{xx} 模板、D_{yy} 模板、D_{xy} 模板替代，即图 2 中的第二行图像。因此尺寸大小为 9×9 的盒状滤波器对应于尺度 σ 为 1.2 的高斯二阶微分。在 SURF 算法中，σ 为 1.2 代表着最小的尺度，即最大的空间分辨率。在盒状滤波器中，白色部分的权值为 1，灰色部分的权值为 0，D_{xx} 模板和 D_{yy} 模板的黑色部分的权值为-2，D_{xy} 模板的黑色部分的权值为-1，我们把黑色部分和白色部分统称为突起部分（**lobe**），则灰色部分对应于平坦部分。经过实验证明，盒状滤波器的性能近似或更好于经过离散化和剪裁过的高斯函数。更重要的是，如果利用积分图像，盒状滤波器的运算完全不取决于它的尺寸大小。

下面我们就给出利用积分图像求 D_{xx}、D_{yy}、和 D_{xy} 方法。首先利用公式 1 把输入图像转换为积分图像，然后应用盒状滤波器逐一对积分图像中的像素进行处理。从图 2 可以看出，盒状滤波器的灰色部分的权值为 0，因此该部分不参与计算，仅仅起到填充模板大小的作用。而 D_{xx} 模板和 D_{yy} 模板都各有两个白色部分和一个黑色部分，因此它们的盒状滤波器共有三个突起部分，而 D_{xy} 模板有两个白色部分和两个黑色部分，因此它的盒状滤波器共有四个突起部分。那么利用盒状滤波器对图像进行滤波处理所得到的响应值的一般公式为：

$$D = \sum_{n=1}^{N} \frac{w_n}{S_n} (I_\Sigma(A_n) + I_\Sigma(D_n) - I_\Sigma(B_n) - I_\Sigma(C_n)) \tag{5}$$

其中，N 表示突起部分的总和，对于 D_{xx} 模板和 D_{yy} 模板来说，N=3，对于 D_{xy} 模板来说，N=4；S_n 表示第 n 个突起部分的面积，如对于 9×9 的 D_{xx} 模板和 D_{yy} 模板来说，突起部分的面积都是 15（即像素的数量），而对于 9×9 的 D_{xy} 模板来说，突起部分的面积都是 9，除以 S_n 的作用是对模板进行归一化处理；w_n 表示第 n 个突起部分的权值；而后面的括号部分就是公式 3，求模板的每个突起部分对应于图像中四个点 A、B、C、D 所组成的矩阵区域的灰度之和。

我们在前面提到，Hessian 矩阵的行列式的极值处即为特征点，而用盒状滤波器（D_{xx}、D_{yy}、D_{xy}）近似替代高斯二阶微分算子（L_{xx}、L_{yy}、L_{xy}）得到 Hessian 矩阵的行列式不是简单 $D_{xx}D_{yy}$-D_{xy}^2，而是需要加上一定的权值，即：

$$\det(\boldsymbol{H}_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2 \tag{6}$$

其中，w 为权值，它的作用是用以平衡因近似所带来的偏差。w 的值为：

$$w = \frac{|L_{xy}(1.2)|_F |D_{yy}(9)|_F}{|L_{yy}(1.2)|_F |D_{xy}(9)|_F} = 0.912 \ldots \cong 0.9 \tag{7}$$

其中，$|x|_F$ 表示 Frobenius 范数。尽管该权值 w 是在 σ 为 1.2 和盒状滤波器的大小为 9×9 的情况下得到，但它也适用于其他的情况。这是因为我们还要对盒状滤波器进行归一化处理（公式 5 中的除以 S_n），而且把 w 保持为常数所引入的误差对最终的结果影响不大。

用盒状滤波器（D_{xx}、D_{yy}、D_{xy}）近似替代高斯二阶微分算子（L_{xx}、L_{yy}、L_{xy}）得到 Hessian 矩阵的迹的公式为：

$$\mathrm{tr}\left(\boldsymbol{H}_{approx}\right) = D_{xx} + D_{yy} \tag{8}$$

尺度空间表示

要想实现特征点的尺度不变性，就需要在尽可能多的尺度图像上检测特征点。如何建立连续多幅的尺度图像，就成了不变性特征检测的关键。目前比较好的方法是建立尺度图像金字塔，如 SIFT 那样。方法是用高斯核对输入图像进行迭代的卷积，并重复进行降采样处理，以减小它的尺寸，也就是金字塔中的图像由底向顶尺度逐渐增加，而图像尺寸大小则逐渐缩小。但该方法效率不高，因为金字塔中各层图像的创建完全依赖于它的前一层图像。而在 SURF 中，尺度图像的建立是依靠盒状滤波器模板，而不是高斯核，因此它采用了不改变输入图像的尺寸大小，而仅仅改变盒状滤波器模板大小的方式，即多幅尺度逐渐增加的尺度图像的尺寸大小完全一致，我把它称为图像堆。我们在前面介绍过，盒状滤波器的滤波处理可以使用积分图像，因此无论盒状滤波器模板是大是小，运算速度都是一样的。所以采用图像堆可以大大提高运算效率。

图像堆也是被分为若干组（octave），每组又由若干层组成，每一组的各层图像都是由输入图像经过不同尺寸大小的盒状滤波器的滤波处理得到，图像的尺度 s 等于该盒状滤波器的尺度 σ。每一组内图像的尺度的变化范围大约是 2 倍的关系。在 SURF 中，最小的尺度图像（即第 1 组第 1 层图像）是由 9×9 的盒状滤波器（即图 2）得到，它的尺度 s = 1.2，对应于高斯函数 σ=1.2，我们把它称为基准滤波器。图像堆中的其他层图像所需的盒状滤波器都是该基准滤波器通过扩展得到，而它们的模板尺寸大小与其所对应的尺度和基准滤波器模板的比率相同，即

$$s_{approx} = L \times \frac{s_0}{L_0} = L \times \frac{1.2}{9} \tag{9}$$

其中，L_0 和 s_0 分别表示基准滤波器模板的尺寸和其所对应图像的尺度，L 表示当前层滤波器模板的尺寸。

为了扩展大尺寸的盒状滤波器模板，还需要考虑下面两个因素：1、首先模板尺寸的增加受限于模板突起部分的长度，即突起部分短边的长度必须是模板大小的三分之一；2、为了使扩展后的中心不变，模板的四周必须同时扩展，并且突起部分至少扩展 2 个像素长。

由于 D_{xx} 和 D_{yy} 模板有一边包括全部 3 个突起部分，因此 D_{xy} 模板的扩展以 D_{xx} 和 D_{yy} 模板为准。突起部分的短边的长度决定着长边扩展的长度。在 D_{xx} 和 D_{yy} 模板扩展时，如果三个突起部分的短边都扩展 2 个像素长，则模板扩展 6 个像素长，所以 6 个像素长是最小的扩展步长，也就是说两个连续模板的长度最小差 6 个像素。

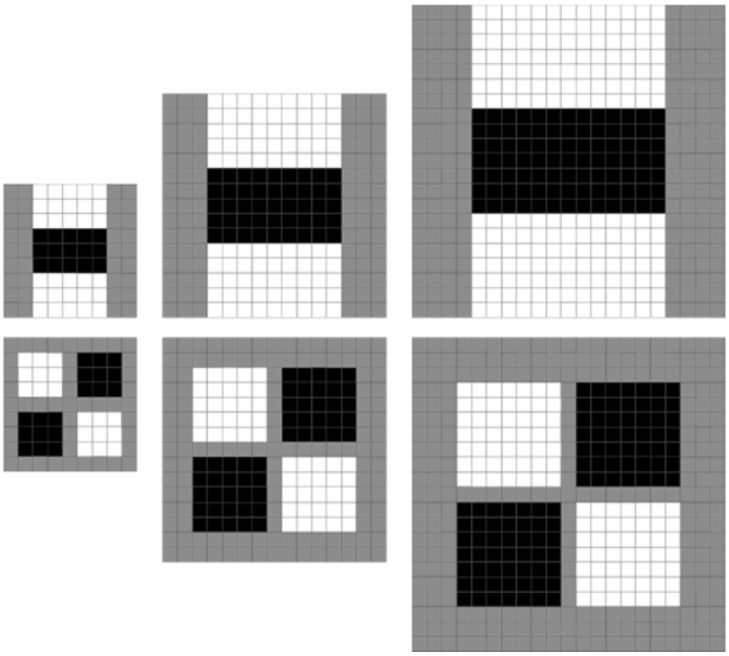

图 3 盒状滤波器模板的扩展

图 3 为 D_{yy} 和 D_{xy} 盒状滤波器模板扩展的实例，模板的尺寸从 9×9 扩展为 15×15 和 21×21。由前面分析可知，这些模板是按照最小步长的方式扩展的。由公式 9 可知，15×15 模板对应的尺度为 2.0，21×21 模板对应的尺度为 2.8。

下面介绍图像堆中各组中各层图像所使用的盒状滤波器模板的变化规律。在第一组中，各层的模板大小分别为 9×9、15×15、21×21、27×27，也就是按照最小步长 6 的方式扩展；第二组中模板扩展的步长翻倍，为 12，模板的大小为 15×15、27×27、39×39、51×51；第三组的步长再翻倍，为 24，则模板大小为 27×27、51×51、75×75、99×99；第四组的步长为 48，模板大小为 51×51、99×99、147×147、195×195；其他组以此类推。一般来说图像堆分为 3 至 4 组，每组有 4 层。下面给出每层模块的尺寸大小及其所对应的尺度：

	第一层		第二层		第三层		第四层	
	尺寸	尺度 *s*	尺寸	尺度 *s*	尺寸	尺度 *s*	尺寸	尺度 *s*
第一组	9	1.2	15	2.0	21	2.8	27	3.6
第二组	15	2.0	27	3.6	39	5.2	51	6.8
第三组	27	3.6	51	6.8	75	10.0	99	13.2
第四组	51	6.8	99	13.2	147	19.6	195	26.0

我们在前面提到过，每一组图像的尺度的变化范围大约是 2 倍的关系，我们来验证一下是否满足上述条件。以第一组为例，因为在后面我们还要在 3×3×3 的范围内进行非最大值抑制，所以不是在第一层图像而是在第二层图像找特征点，并且还要进行插值运算，因此最精细的尺度应该在第一层和第二层图像之间，即(1.2 + 2.0) / 2 = 1.6，同理最粗糙的尺度为 (3.6 + 2.8) / 2 = 3.2。它们之间的变化范围正好为 2 倍的关系。其他组的计算方法相同，尺度的变化范围要略大于 2 倍的关系，但不会超过 2.3 倍。所以满足上述条件。

从上面的表可以看出，各个层之间尺度有重叠的现象，其目的是为了覆盖所有可能的尺度。另外随着尺度的增大，被检测到的特征点的数量会锐减，因此为了降低运算量，我们可以在大尺度的图像内减少检测的次数。通常的做法是采用隔点采样的方法，即在第一组图像内，我们对所有像素都进行采样从而计算它们的滤波响应值，也就是采样间隔为 $1=2^0$；在第二组图像内，我们每隔一个像素采样一次，采样间隔为 $2=2^1$，并且只对采样像素进行滤波处理；同理第三组的采样间隔为 $4=2^2$，第四组的采样间隔为 $8=2^3$。

下面我们用公式来表述一下图像所在组和层与其所对应的滤波器模板尺寸的关系：

$$L = 3 \times [2^{o+1} \times (l+1) + 1] \qquad o = 0,1,2,3 \qquad l = 0,1,2,3 \tag{10}$$

其中，L 为模板尺寸，o 表示组索引，l 表示组内的层索引，索引值都是从 0 开始。从该公式可以看出，方括号内的部分其实是该模板突起部分短边的长度。

这里需要说明的是，公式 10 是通过 Bay 的原著总结出来的公式，但 opencv2.4.9 中没有应用该公式，而是使用的下列公式：

$$L = (9 + 6 \times l) \times 2^o \qquad o = 0,1,2,3 \qquad l = 0,1,2,3 \tag{11}$$

特征点定位

在上一步，我们通过建立图像堆，并在每一层图像上应用不同尺寸大小的盒状滤波器模板得到了滤波响应值，然后通过公式 6 得到了所有像素的 Hessian 矩阵的行列式，下面我们就基于这些行列式值找出特征点，并精确确定它们的位置。它包括三个步骤——阈值、非最大值抑制和插值。

我们首先取阈值，去掉那些行列式值低的像素，仅保留那些最强的响应值。很明显，阈值选择得大，所检测到的特征点就多，反之阈值小，特征点就少。

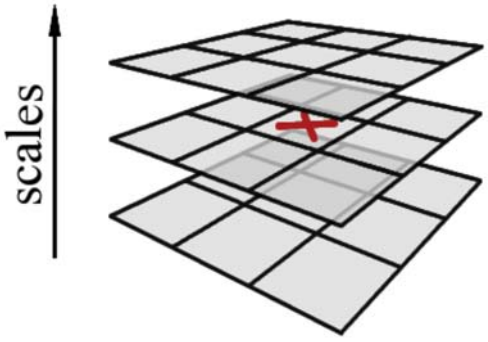

图 4 非最大值抑制所在的 3×3×3 区域

然后就是进行非最大值抑制，即在邻域范围内去掉那些不是最大值的像素。为了满足尺度不变性，在进行非最大值抑制时不仅需要与被检测像素所在的尺度图像的邻域像素相比较，还需要与相邻尺度的图像像素比较，也就是要像如图 4 所示的那样在 3×3×3 的区域内

进行比较，同一层内有 8 个邻域像素，下层和上层各有 9 个邻域像素。

非最大值抑制是在图像堆的组内进行，也就是同组的相邻层之间进行比较。由于每组的第一层图像和最后一层图像各只有一个相邻层，因此这两层不能进行非最大值抑制比较。以每组有 4 层图像为例，这样就只有中间两层可以进行比较，我们把可以进行非最大值抑制比较的层称为中间层。如果图像堆有 4 组，那么图像堆一共有 16（4×4）层图像，而中间层则只有 8（2×4）层。

最后是应用插值法确定特征点位置。由于离散化，前面所检测到的特征点的位置往往并不是真正的位置，因此我们还要应用插值法找到亚像素级精度的特征点位置。特征点的位置不仅应该包括所在层图像的坐标位置，还应该包括尺度（滤波器模板的尺寸也可以，因为它们之间可以通过公式 9 换算），因此称为尺度坐标——$\boldsymbol{X} = (x, y, s)^T$。

SURF 算法与 SIFT 算法一样，应用的是泰勒级数展开式来进行插值计算。设 $B(\boldsymbol{X})$为在 $\boldsymbol{X}$ 处的特征点响应值，即 Hessian 矩阵的行列式，则它的泰勒级数（只展开到二项式）为：

$$B(\boldsymbol{X}) = B + \left(\frac{\partial B}{\partial \boldsymbol{X}}\right)^T \boldsymbol{X} + \frac{1}{2}\boldsymbol{X}^T \frac{\partial^2 B}{\partial \boldsymbol{X}^2}\boldsymbol{X} \tag{12}$$

相对于点 $\boldsymbol{X}$ 的偏移量$\hat{\boldsymbol{x}}$为：

$$\hat{\boldsymbol{x}} = -\left(\frac{\partial^2 B}{\partial \boldsymbol{X}^2}\right)^{-1} \frac{\partial B}{\partial \boldsymbol{X}} \tag{13}$$

其中，

$$\frac{\partial^2 B}{\partial \boldsymbol{X}^2} = \begin{bmatrix} d_{xx} & d_{xy} & d_{xs} \\ d_{xy} & d_{yy} & d_{ys} \\ d_{xs} & d_{ys} & d_{ss} \end{bmatrix} \tag{14}$$

$$\frac{\partial B}{\partial \boldsymbol{X}} = \begin{bmatrix} d_x \\ d_y \\ d_s \end{bmatrix} \tag{15}$$

公式 14 和公式 15 中等号右侧分别表示对 Hessian 矩阵行列式值图像的二阶偏导和一阶导数。

公式 13 的推导详见我的另一篇文章——《Opencv2.4.9 源码分析——SIFT》。则通过泰勒级数插值拟合出的新尺度坐标$\acute{\boldsymbol{X}}$为：

$$\acute{\boldsymbol{X}} = \boldsymbol{X} + \hat{\boldsymbol{x}} \tag{16}$$

方向角度的分配

为了实现旋转不变性，就必须为每一个特征点分配一个复现性好的方向角度。为此，我们首先建立一个以特征点为中心，半径为 6s 的圆形邻域，我们称为 6s 圆邻域，并对该圆以 *s* 为采样间隔进行采样，其中 *s* 为特征点所在的尺度图像的尺度，则结果如图 5 所示。图中中心的位置为特征点，其他的点为采样像素。

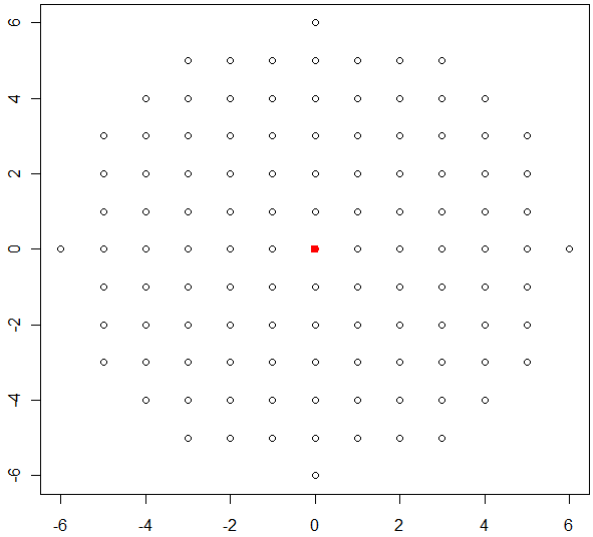

图 5 6*s* 圆邻域

然后对该 6*s* 圆邻域内的所有采样像素计算它们的 *x* 方向和 *y* 方向上的 Haar 小波响应值， Haar 小波响应的边长尺寸是 4*s*。Haar 小波是一种最简单的滤波器，用它可以检测出 *x* 方向和 *y* 方向的梯度。如图 6 所示，左侧为 *x* 方向的 Haar 小波响应，右侧为 *y* 方向。黑色部分的权值为 1，白色部分的权值为-1。用 Haar 小波的另一个好处是结合积分图像，不管 Haar 小波响应的尺寸多大，都只需 6 个运算即可得到 *x* 方向和 *y* 方向的梯度。

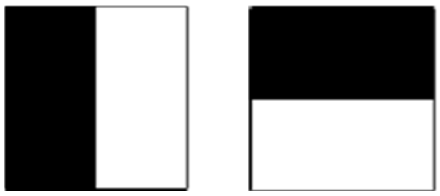

图 6 Haar 小波响应

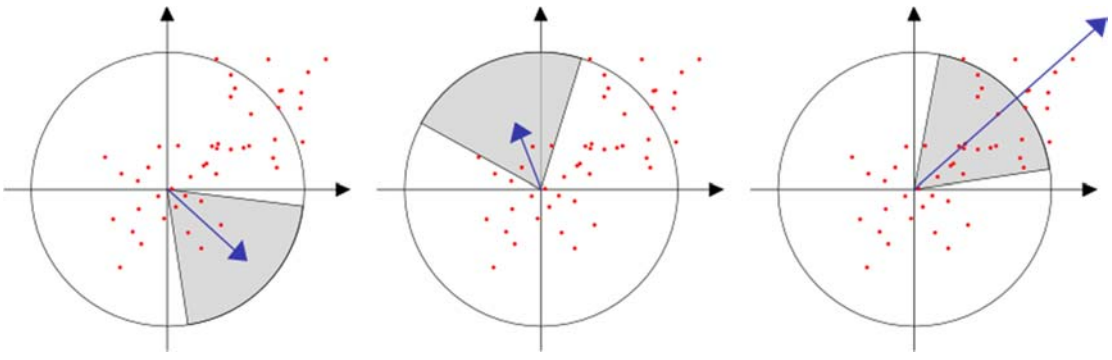

图 7 梯度坐标系内求方向

很明显，距离特征点越近，采样像素对特征点的影响越大，因此我们还需要对 *x* 方向和

y 方向的 Haar 小波响应值进行加权处理。常用的加权函数为高斯函数，它的方差设为 $2s$。

最后我们分别以加权后的 x 方向和 y 方向的 Haar 小波响应值为 x 轴和 y 轴建立一个梯度坐标系，所有的 $6s$ 圆邻域内的采样点都分布在该坐标系内。如图 7 所示，坐标系内的点为采样点，它的 x 轴坐标和 y 轴坐标分别对应于加权后的 x 方向和 y 方向的 Haar 小波响应值。为了求取特征点的方向，我们需要在该梯度坐标系内设计一个以原点为中心，张角为 60 度的扇形滑动窗口，如图 7 所示，要以一定的步长旋转这个滑动窗口，并对该滑动窗口内的所有点累计其 x 轴坐标和 y 轴坐标值，计算累计和的模和幅角：

$$m_w = \sum_k x + \sum_k y \tag{17}$$

$$\theta_w = arctan\left(\frac{\sum_k x}{\sum_k y}\right) \tag{18}$$

其中，m_w 和 θ_w 分别为 w 扇形滑动窗口内采样点的模和幅角，假设 w 窗口内共有 k 个点，则 $\sum_k x$ 和 $\sum_k y$ 分别表示这 k 个点的横坐标和纵坐标之和，前面已经介绍过，坐标之和也就是 Haar 小波响应值之和。

这样旋转扇形滑动窗口，直至旋转一周为止，比较所有窗口下的模值，模值最大的那个窗口所对应的幅角就是该特征点的方向角度：

$$\theta = \theta_{w|max\{m_w\}} \tag{19}$$

要说明一点的是，在一些应用中，旋转不变性并不是必须的。这就像一个人如果保持正常的姿态，让他去观察一个固定的建筑物，可能会由于观察位置的变化建筑物的尺度和视角会变化，还可能由于天气的原因，亮度会变化，但建筑物的旋转角度变化不大，甚至不会变化。Bay 把这种不需要确定特征点方向的方法称为 U-SURF。实验表明 U-SURF 计算速度更快，并且对在正负 15 度范围内的旋转变化下，U-SURF 具备较强的鲁棒性。

特征点描述符的生成

对特征点检测的要求是重复性要好，而对特征点描述符的要求是可区分性要好。

提取 SURF 描述符的第一步是构造一个以特征点为中心的正方形邻域，正方形的边长为 $20s$，s 仍然指的是尺度，我们称该邻域为 $20s$ 方邻域。当然为了实现旋转不变性，该 $20s$ 方邻域需要校正为与特征点的方向一致，如图 8 所示。我们对 $20s$ 方邻域进行采样间隔为 s 的等间隔采样，并把该邻域划分为 4×4=16 个子区域，则 $20s$ 方邻域内共有 400 个采样像素，而每个子区域则有 25 个采样像素。

对每个子区域内的所有 25 个采样像素，仍然采用图 6 所示的 Haar 小波计算它们的 x 方向和 y 方向的梯度，在这里 Haar 小波响应的尺寸为 2s。我们把 x 方向和 y 方向的 Haar 小波

响应值分别定义为 *dx* 和 *dy*。对于某个采样像素的 *dx* 和 *dy* 仍然需要根据该像素与中心特征点的距离进行高斯加权处理，高斯函数的方差为 3.3*s*。

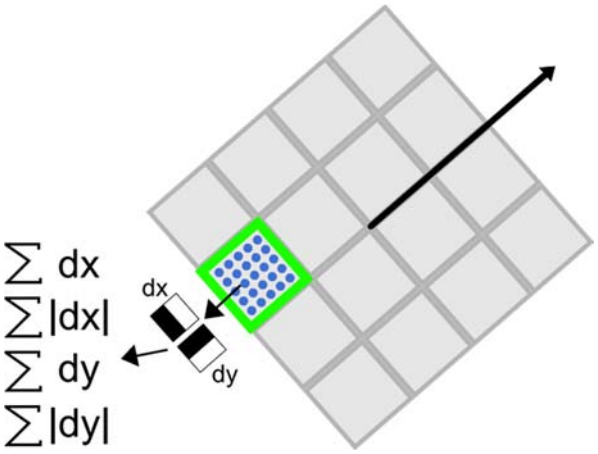

图 8 描述符表示

在每个子区域，我们需要累加所有 25 个采样像素的 *dx* 和 *dy*，这样形成了描述符的一部分。而为了把强度变化的极性信息也包括进描述符中，我们还需要对 *dx* 和 *dy* 的绝对值进行累加。这样每个子区域就可以用一个 4 维特征矢量 **v** 表示：

$$\mathbf{v} = [\ \Sigma dx，\ \Sigma dy，\ \Sigma|dx|，\ \Sigma|dy|\ ] \qquad （20）$$

把所有 4×4 个子区域的 4 维特征矢量 **v** 组合在一起，就形成了一个 64 维特征矢量，即 SURF 描述符。

为了去除光照变化的影响，我们还需要对该描述符进行归一化处理。设 64 维特征矢量的 SURF 描述符为 $P=\{p_1,p_2,\ldots,p_{64}\}$，归一化公式为：

$$q_i = \frac{p_i}{\sqrt{\sum_{j=1}^{64} p_j^2}}, \qquad i = 1,2,\ldots,64 \qquad （21）$$

其中，$Q=\{q_1,q_2,\ldots,q_{64}\}$为归一化后的特征矢量。

Bay 在论文中指出，他们对各类小波特性进行了大量的实验，使用了 dx^2 和 dy^2，高阶小波，PCA，中值，均值等，并通过评估，得出了使用公式 20 那样的形式能够达到最佳的效果。而且他们还通过改变 20*s* 方邻域内的采样像素和子区域的数量，得出了 4×4 个子区域是最佳的划分方式。多于 4×4 个子区域的划分鲁棒性较差，并且增加了匹配时间；反之，匹配效果较差，但缩短了匹配时间。但 3×3 个子区域划分方式仍然是可以接受的，因为它要好过其他的描述符。

另外除了 64 维特征矢量的 SURF 描述符外，Bay 还提出了 128 维特征矢量的描述符方法。两者方法基本相同，区别在于根据 *dy* 是否小于 0，Σ*dx* 和 Σ|*dx*|分别被划分成两个部分，同理 Σ*dy* 和 Σ|*dy*|也是根据 *dx* 的正负号分别被划分成两个部分。这样每个子区域就用一个 8

维特征矢量表示，则描述符就增加到了 128 维。这么做虽然增加了匹配时间，但描述符的可区分性更强。

为了实现快速匹配，我们还可以利用特征点的拉普拉斯响应的正负号。SURF 算法检测的是斑点类的特征点，而特征点的拉普拉斯正负号分别代表着黑背景下的亮斑和白背景下的黑斑，如图 9 所示。在匹配的时候，只有拉普拉斯符合相同的特征点才能进行相互匹配，显然，这样可以节省特征点匹配搜索的时间。拉普拉斯符号就是 Hessian 矩阵的迹的符号，因此只要求出 Hessian 矩阵的迹的符号即可。这一步骤并不会增加运算量，因为在特征点检测时已对 Hessian 矩阵的迹进行了计算，我们只要把该值作为特征点的一个变量保存下来即可。

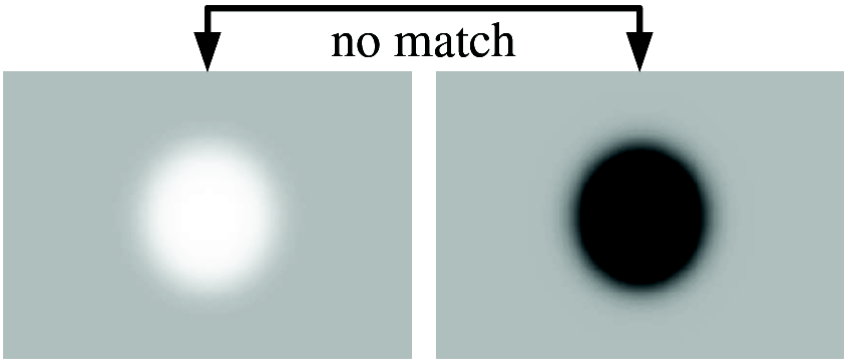

图 9 特征点的两种拉普拉斯响应

SURF 算法在许多地方都借鉴了 SIFT 算法的成功经验，下面我们就比较一下两者：

1、SURF 算法的抗干扰能力要强于 SIFT 算法；

2、SURF 算法不像 SIFT 算法，它可以实现并行计算，这样更加快了运算速度；

3、SURF 算法的精度略逊于 SIFT 算法；

4、在亮度不变性和视角不变性方面，SURF 算法不如 SIFT 算法。

二、SURF 源码分析

在 opencv2.4.9 中，实现 SURF 算法的文件为 sources/modules/nonfree/src/surf.cpp

SURF 类的构造函数为：

```
SURF::SURF( )
SURF::SURF(double hessianThreshold, int nOctaves=4, int nOctaveLayers=2, bool extended=true,
        bool upright=false )
```

参数的含义为：

hessianThreshold 为 Hessian 矩阵行列式响应值的阈值

nOctaves 为图像堆的组数

nOctaveLayers 为图像堆中每组中的中间层数，该值加 2 等于每组图像中所包含的层数

extended 表示是 128 维描述符，还是 64 维描述符，为 true 时，表示 128 维描述符

upright 表示是否采用 U-SURF 算法，为 true 时，采用 U-SURF 算法

SURF 类的重载运算符( )为：

```
void SURF::operator()(InputArray _img, InputArray _mask,
                      CV_OUT vector<KeyPoint>& keypoints,
                      OutputArray _descriptors,
                      bool useProvidedKeypoints) const
//_img 表示输入 8 位灰度图像
//_mask 为掩码矩阵
//keypoints 为特征点矢量数组
//_descriptors 为特征点描述符
//useProvidedKeypoints 表示是否运行特征点的检测，该值为 true 表示不进行特征点的检测，而只是利用输入的特征点进行特征点描述符的运算。
{
    //定义、初始化各种矩阵，sum 为输入图像的积分图像矩阵，mask1 和 msum 为掩码所需矩阵
    Mat img = _img.getMat(), mask = _mask.getMat(), mask1, sum, msum;
    //变量 doDescriptors 表示是否进行特征点描述符的运算
    bool doDescriptors = _descriptors.needed();
    //确保输入图像的正确
    CV_Assert(!img.empty() && img.depth() == CV_8U);
    if( img.channels() > 1 )
        cvtColor(img, img, COLOR_BGR2GRAY);
    //确保各类输入参数的正确
    CV_Assert(mask.empty() || (mask.type() == CV_8U && mask.size() == img.size()));
    CV_Assert(hessianThreshold >= 0);
    CV_Assert(nOctaves > 0);
    CV_Assert(nOctaveLayers > 0);
    //计算输入图像的积分图像
    integral(img, sum, CV_32S);

    // Compute keypoints only if we are not asked for evaluating the descriptors are some given locations:
    //是否进行特征点检测运算，useProvidedKeypoints=false 表示进行该运算
    if( !useProvidedKeypoints )
    {
        if( !mask.empty() )      //掩码
        {
            cv::min(mask, 1, mask1);
            integral(mask1, msum, CV_32S);      //msum 为掩码图像的积分图像
        }
        //进行 Fast-Hessian 特征点检测，fastHessianDetector 函数在后面给出详细解释
        fastHessianDetector( sum, msum, keypoints, nOctaves, nOctaveLayers, (float)hessianThreshold );
    }
```

```
    //N 为所检测到的特征点数量
    int i, j, N = (int)keypoints.size();
    if( N > 0 )       //如果检测到了特征点
    {
        Mat descriptors;       //描述符变量矩阵
        bool _1d = false;
        //由 extended 变量得到描述符是 128 维的还是 64 维的
        int dcols = extended ? 128 : 64;
        //一个描述符的存储空间大小
        size_t dsize = dcols*sizeof(float);

        if( doDescriptors )       //需要得到描述符
        {
            //定义所有描述符的排列形式
            _1d = _descriptors.kind() == _InputArray::STD_VECTOR && _descriptors.type() ==
CV_32F;
            if( _1d )
            {
                //所有特征点的描述符连在一起，组成一个矢量
                _descriptors.create(N*dcols, 1, CV_32F);
                descriptors = _descriptors.getMat().reshape(1, N);
            }
            else
            {
                //一个特征点就是一个描述符，一共 N 个描述符
                _descriptors.create(N, dcols, CV_32F);
                descriptors = _descriptors.getMat();
            }
        }

        // we call SURFInvoker in any case, even if we do not need descriptors,
        // since it computes orientation of each feature.
        //方向角度的分配和描述符的生成，SURFInvoker 在后面给出了详细的解释
        parallel_for_(Range(0, N), SURFInvoker(img, sum, keypoints, descriptors, extended,
upright) );

        // remove keypoints that were marked for deletion
        //删除那些被标注为无效的特征点
        // i 表示特征点删除之前的特征点索引，j 表示删除之后的特征点索引
        for( i = j = 0; i < N; i++ )
        {
            if( keypoints[i].size > 0 )       //size 大于 0，表示该特征点为有效的特征点
            {
                if( i > j )       // i > j 表示在 i 之前有被删除的特征点
```

```
                {
                    //用后面的特征点依次填补被删除的那些特征点的位置
                    keypoints[j] = keypoints[i];
                    //相应的描述符的空间位置也要调整
                    if( doDescriptors )
                        memcpy( descriptors.ptr(j), descriptors.ptr(i), dsize);
                }
                j++;      //索引值计数
            }
        }
        //N 表示特征点删除之前的总数，j 表示删除之后的总数，N > j 表明有一些特征点被删除
        if( N > j )
        {
            //重新赋值特征点的总数
            N = j;
            keypoints.resize(N);
            if( doDescriptors )
            {
                //根据描述符的排列形式，重新赋值
                Mat d = descriptors.rowRange(0, N);
                if( _1d )
                    d = d.reshape(1, N*dcols);
                d.copyTo(_descriptors);
            }
        }
    }
}
```

SURF 算法中特征点检测函数 fastHessianDetector：

```
static void fastHessianDetector( const Mat& sum, const Mat& mask_sum, vector<KeyPoint>& keypoints, int nOctaves, int nOctaveLayers, float hessianThreshold )
{
    /* Sampling step along image x and y axes at first octave. This is doubled
       for each additional octave. WARNING: Increasing this improves speed,
       however keypoint extraction becomes unreliable. */
    //为了减少运算量，提高计算速度，程序设置了采样间隔。即在第一组的各层图像的每个像素都通过盒状滤波器模块进行滤波处理，而在第二组的各层图像中采用隔点（采样间隔为 2^1）滤波器处理的方式，第三组的采样间隔为 2^2，其他组以此类推。
    //定义初始采样间隔，即第一组各层图像的采样间隔
    const int SAMPLE_STEP0 = 1;
    //总的层数，它等于组数乘以每组的层数
    int nTotalLayers = (nOctaveLayers+2)*nOctaves;
    //中间层数，即进行非最大值抑制的层数
    int nMiddleLayers = nOctaveLayers*nOctaves;
```

//各类矢量变量，dets 为 Hessian 矩阵行列式矢量矩阵，traces 为 Hessian 矩阵迹矢量矩阵，sizes 为盒状滤波器的尺寸大小矢量，sampleSteps 为每一层的采样间隔矢量，middleIndices 为中间层（即进行非最大值抑制的层）的索引矢量

```
vector<Mat> dets(nTotalLayers);
vector<Mat> traces(nTotalLayers);
vector<int> sizes(nTotalLayers);
vector<int> sampleSteps(nTotalLayers);
vector<int> middleIndices(nMiddleLayers);
//清空特征点变量
keypoints.clear();

// Allocate space and calculate properties of each layer
//index 为变量 nTotalLayers 的索引，middleIndex 为变量 nMiddleLayers 的索引，step 为采样间隔
int index = 0, middleIndex = 0, step = SAMPLE_STEP0;
//分配内存空间，计算各层的数据
//由于采用了采样间隔，因此各层图像所检测到的行列式、迹等数量会不同，这样就需要为每一层分配不同的空间大小，保存不同数量的数据
for( int octave = 0; octave < nOctaves; octave++ )
{
    for( int layer = 0; layer < nOctaveLayers+2; layer++ )
    {
        /* The integral image sum is one pixel bigger than the source image*/
        dets[index].create( (sum.rows-1)/step, (sum.cols-1)/step, CV_32F );
        traces[index].create( (sum.rows-1)/step, (sum.cols-1)/step, CV_32F );
        //SURF_HAAR_SIZE0 = 9 表示第一组第一层所使用的盒状滤波器的尺寸大小
        //SURF_HAAR_SIZE_INC = 6 表示盒状滤波器尺寸大小增加的像素数量
        //公式 11
        sizes[index] = (SURF_HAAR_SIZE0 + SURF_HAAR_SIZE_INC*layer) << octave;
        sampleSteps[index] = step;

        if( 0 < layer && layer <= nOctaveLayers )
            middleIndices[middleIndex++] = index;      //标记中间层对应于图像堆中的所在层
        index++;      //层的索引值加 1
    }
    step *= 2;      //采样间隔翻倍
}

// Calculate hessian determinant and trace samples in each layer
//计算每层的 Hessian 矩阵的行列式和迹，具体实现的函数为 SURFBuildInvoker，该函数在后面给出详细的讲解
//parallel_for_表示这些计算可以进行平行处理（当然在编译的时候要选择 TBB 才能实现平行计算）。
```

```
    parallel_for_( Range(0, nTotalLayers),
                   SURFBuildInvoker(sum, sizes, sampleSteps, dets, traces) );

    // Find maxima in the determinant of the hessian
    //检测特征点，具体实现的函数为 SURFFindInvoker，该函数在后面给出详细的讲解
    parallel_for_( Range(0, nMiddleLayers),
                   SURFFindInvoker(sum, mask_sum, dets, traces, sizes,
                                   sampleSteps, middleIndices, keypoints,
                                   nOctaveLayers, hessianThreshold) );
    //特征点排序
    std::sort(keypoints.begin(), keypoints.end(), KeypointGreater());
}
```

计算行列式和迹的类：

```
// Multi-threaded construction of the scale-space pyramid
struct SURFBuildInvoker : ParallelLoopBody
{
    //构造函数
    SURFBuildInvoker( const Mat& _sum, const vector<int>& _sizes,
                      const vector<int>& _sampleSteps,
                      vector<Mat>& _dets, vector<Mat>& _traces )
    {
        sum = &_sum;
        sizes = &_sizes;
        sampleSteps = &_sampleSteps;
        dets = &_dets;
        traces = &_traces;
    }
    //重载( )运算符
    void operator()(const Range& range) const
    {
        //遍历所有层，计算每层的行列式和迹，calcLayerDetAndTrace 函数在后面给出详细
介绍
        for( int i=range.start; i<range.end; i++ )
            calcLayerDetAndTrace( *sum, (*sizes)[i], (*sampleSteps)[i], (*dets)[i], (*traces)[i] );
    }
    //定义类的成员变量
    const Mat *sum;
    const vector<int> *sizes;
    const vector<int> *sampleSteps;
    vector<Mat>* dets;
    vector<Mat>* traces;
};
```

计算某一层的行列式和迹 calcLayerDetAndTrace 函数：

```
/*
 * Calculate the determinant and trace of the Hessian for a layer of the
 * scale-space pyramid
 */
static void calcLayerDetAndTrace( const Mat& sum, int size, int sampleStep,
                                  Mat& det, Mat& trace )
{
    //NX、NY、NXY 分别表示 Dxx 模板、Dyy 模板和 Dxy 模板中突起部分的数量，即公式 5 中的变量 N
    const int NX=3, NY=3, NXY=4;
    //dx_s 变量、dy_s 变量、dxy_s 变量的意思是分别用二维数组表示 9×9 的 Dxx 模板、Dyy 模板和 Dxy 模板，二维数组的第二维代表模板的突起部分，第一维代表突起部分的坐标位置和权值。以模板的左上角为坐标原点，数组第一维中 5 个元素中的前两个表示突起部分左上角坐标，紧接着的后 2 个元素表示突起部分的右下角坐标，最后一个元素表示该突起部分的权值
    const int dx_s[NX][5] = { {0, 2, 3, 7, 1}, {3, 2, 6, 7, -2}, {6, 2, 9, 7, 1} };
    const int dy_s[NY][5] = { {2, 0, 7, 3, 1}, {2, 3, 7, 6, -2}, {2, 6, 7, 9, 1} };
    const int dxy_s[NXY][5] = { {1, 1, 4, 4, 1}, {5, 1, 8, 4, -1}, {1, 5, 4, 8, -1}, {5, 5, 8, 8, 1} };
    //Dx、Dy 和 Dxy 为尺寸重新调整以后的模板的信息数据
    SurfHF Dx[NX], Dy[NY], Dxy[NXY];
    //判断盒状滤波器的尺寸大小是否超出了输入图像的边界
    if( size > sum.rows-1 || size > sum.cols-1 )
        return;
    //由 9×9 的初始模板生成其他尺寸大小的盒状滤波器模板，resizeHaarPattern 函数在后面给出详细的解释
    resizeHaarPattern( dx_s , Dx , NX , 9, size, sum.cols );
    resizeHaarPattern( dy_s , Dy , NY , 9, size, sum.cols );
    resizeHaarPattern( dxy_s, Dxy, NXY, 9, size, sum.cols );

    /* The integral image 'sum' is one pixel bigger than the source image */
    //由该层的采样间隔计算图像的行和列的采样像素的数量
    int samples_i = 1+(sum.rows-1-size)/sampleStep;
    int samples_j = 1+(sum.cols-1-size)/sampleStep;

    /* Ignore pixels where some of the kernel is outside the image */
    //由该层的模板尺寸和采样间隔计算因为边界效应而需要预留的图像边界的大小
    int margin = (size/2)/sampleStep;
    //遍历该层所有的采样像素
    for( int i = 0; i < samples_i; i++ )
    {
        //在积分图像中，行首地址指针
        const int* sum_ptr = sum.ptr<int>(i*sampleStep);
        //去掉边界预留部分的该行的行列式首地址指针
        float* det_ptr = &det.at<float>(i+margin, margin);
```

```
            //去掉边界预留部分的该行的迹首地址指针
            float* trace_ptr = &trace.at<float>(i+margin, margin);
            for( int j = 0; j < samples_j; j++ )
            {
                //利用公式 5 计算三个盒状滤波器的响应值，calcHaarPattern 函数很简单，就不再给出解释
                float dx  = calcHaarPattern( sum_ptr, Dx , 3 );
                float dy  = calcHaarPattern( sum_ptr, Dy , 3 );
                float dxy = calcHaarPattern( sum_ptr, Dxy, 4 );
                //指向积分图像中该行的下一个像素
                sum_ptr += sampleStep;
                //利用公式 6 计算行列式
                det_ptr[j] = dx*dy - 0.81f*dxy*dxy;
                //利用公式 8 计算迹
                trace_ptr[j] = dx + dy;
            }
        }
    }
```

由 9×9 的初始模板生成其他尺寸大小的盒状滤波器模板 resizeHaarPattern 函数：

```
// src 为源模板，即 9×9 的模板
// dst 为得到的新模板的信息数据
// n 为该模板的突起部分的数量
// oldSize 为源模板 src 的尺寸，即为 9
// newSize 为新模板 dst 的尺寸
// widthStep 为积分图像的宽步长，也就是输入图像的宽步长
static void
resizeHaarPattern( const int src[][5], SurfHF* dst, int n, int oldSize, int newSize, int widthStep )
{
    //由新、旧两个模板的尺寸计算它们的比值
    float ratio = (float)newSize/oldSize;
    //遍历模板的所有突起部分
    for( int k = 0; k < n; k++ )
    {
        //由比值计算新模板突起部分的左上角和右下角坐标
        int dx1 = cvRound( ratio*src[k][0] );
        int dy1 = cvRound( ratio*src[k][1] );
        int dx2 = cvRound( ratio*src[k][2] );
        int dy2 = cvRound( ratio*src[k][3] );
        //由突起部分的坐标计算它在积分图像的坐标位置，即图 1 中的四个点 A、B、C、D
        dst[k].p0 = dy1*widthStep + dx1;
        dst[k].p1 = dy2*widthStep + dx1;
        dst[k].p2 = dy1*widthStep + dx2;
        dst[k].p3 = dy2*widthStep + dx2;
```

```
            //公式 5 中的 w_n/s_n
            dst[k].w = src[k][4]/((float)(dx2-dx1)*(dy2-dy1));
        }
    }
```

检测特征点的类：

```
// Multi-threaded search of the scale-space pyramid for keypoints
struct SURFFindInvoker : ParallelLoopBody
{
    //构造函数
    SURFFindInvoker( const Mat& _sum, const Mat& _mask_sum,
                     const vector<Mat>& _dets, const vector<Mat>& _traces,
                     const vector<int>& _sizes, const vector<int>& _sampleSteps,
                     const vector<int>& _middleIndices, vector<KeyPoint>& _keypoints,
                     int _nOctaveLayers, float _hessianThreshold )
    {
        sum = &_sum;       //积分图像
        mask_sum = &_mask_sum;       //掩码所需的积分图像
        dets = &_dets;       //行列式
        traces = &_traces;       //迹
        sizes = &_sizes;       //尺寸
        sampleSteps = &_sampleSteps;       //采样间隔
        middleIndices = &_middleIndices;       //中间层
        keypoints = &_keypoints;       //特征点
        nOctaveLayers = _nOctaveLayers;       //层
        hessianThreshold = _hessianThreshold;       //行列式的阈值
    }
    //成员函数，该函数在后面给出详细解释
    static void findMaximaInLayer( const Mat& sum, const Mat& mask_sum,
                    const vector<Mat>& dets, const vector<Mat>& traces,
                    const vector<int>& sizes, vector<KeyPoint>& keypoints,
                    int octave, int layer, float hessianThreshold, int sampleStep );
    //重载( )运算符
    void operator()(const Range& range) const
    {
        //遍历中间层
        for( int i=range.start; i<range.end; i++ )
        {
            int layer = (*middleIndices)[i];       //提取出中间层所对应的图像堆中的所在层
            int octave = i / nOctaveLayers;       //计算该层所对应的组索引
            //检测特征点
            findMaximaInLayer( *sum, *mask_sum, *dets, *traces, *sizes,
                               *keypoints, octave, layer, hessianThreshold,
                               (*sampleSteps)[layer] );
        }
```

```
    }
    //成员变量
    const Mat *sum;
    const Mat *mask_sum;
    const vector<Mat>* dets;
    const vector<Mat>* traces;
    const vector<int>* sizes;
    const vector<int>* sampleSteps;
    const vector<int>* middleIndices;
    vector<KeyPoint>* keypoints;
    int nOctaveLayers;
    float hessianThreshold;

    static Mutex findMaximaInLayer_m;
};
```

检测特征点的 findMaximaInLayer 函数：

```
/*
 * Find the maxima in the determinant of the Hessian in a layer of the
 * scale-space pyramid
 */
void SURFFindInvoker::findMaximaInLayer( const Mat& sum, const Mat& mask_sum,
                           const vector<Mat>& dets, const vector<Mat>& traces,
                           const vector<int>& sizes, vector<KeyPoint>& keypoints,
                           int octave, int layer, float hessianThreshold, int sampleStep )
{
    // Wavelet Data
    //下面变量在使用掩码时会用到
    const int NM=1;
    const int dm[NM][5] = { {0, 0, 9, 9, 1} };
    SurfHF Dm;
    //该层所用的滤波器模板的尺寸
    int size = sizes[layer];

    // The integral image 'sum' is one pixel bigger than the source image
    //根据采样间隔得到该层图像每行和每列的采样像素数量
    int layer_rows = (sum.rows-1)/sampleStep;
    int layer_cols = (sum.cols-1)/sampleStep;

    // Ignore pixels without a 3x3x3 neighbourhood in the layer above
    //在每组内，上一层所用的滤波器模板要比下一层的模板尺寸大，因此由于边界效应而预留的边界空白区域就要多，所以在进行 3×3×3 邻域比较时，就要把多出来的这部分区域去掉
    int margin = (sizes[layer+1]/2)/sampleStep+1;
    //如果要用掩码，则需要调整掩码矩阵的大小
```

```
if( !mask_sum.empty() )
    resizeHaarPattern( dm, &Dm, NM, 9, size, mask_sum.cols );
//计算该层图像的宽步长
int step = (int)(dets[layer].step/dets[layer].elemSize());
//遍历该层图像的所有像素
for( int i = margin; i < layer_rows - margin; i++ )
{
    //行列式矩阵当前行的首地址指针
    const float* det_ptr = dets[layer].ptr<float>(i);
    //迹矩阵当前行的首地址指针
    const float* trace_ptr = traces[layer].ptr<float>(i);
    for( int j = margin; j < layer_cols-margin; j++ )
    {
        float val0 = det_ptr[j];       //行列式矩阵当前像素
        if( val0 > hessianThreshold )       //行列式值要大于所设置的阈值
        {
            /* Coordinates for the start of the wavelet in the sum image. There
                is some integer division involved, so don't try to simplify this
                (cancel out sampleStep) without checking the result is the same */
            //该点对应于输入图像的坐标位置
            int sum_i = sampleStep*(i-(size/2)/sampleStep);
            int sum_j = sampleStep*(j-(size/2)/sampleStep);

            /* The 3x3x3 neighbouring samples around the maxima.
                The maxima is included at N9[1][4] */

            const float *det1 = &dets[layer-1].at<float>(i, j);       //下一层像素
            const float *det2 = &dets[layer].at<float>(i, j);       //当前层像素
            const float *det3 = &dets[layer+1].at<float>(i, j);       //上一层像素
            //3×3×3 区域内的所有 27 个像素的行列式值
            float N9[3][9] = { { det1[-step-1], det1[-step], det1[-step+1],
                                 det1[-1]    , det1[0] , det1[1],
                                 det1[step-1] , det1[step] , det1[step+1]    },
                               { det2[-step-1], det2[-step], det2[-step+1],
                                 det2[-1]    , det2[0] , det2[1],
                                 det2[step-1] , det2[step] , det2[step+1]    },
                               { det3[-step-1], det3[-step], det3[-step+1],
                                 det3[-1]    , det3[0] , det3[1],
                                 det3[step-1] , det3[step] , det3[step+1]    } };

            /* Check the mask - why not just check the mask at the center of the wavelet?
*/
            //去掉掩码没有包含的像素
            if( !mask_sum.empty() )
```

```
            {
                const int* mask_ptr = &mask_sum.at<int>(sum_i, sum_j);
                float mval = calcHaarPattern( mask_ptr, &Dm, 1 );
                if( mval < 0.5 )
                    continue;
            }

            /* Non-maxima suppression. val0 is at N9[1][4]*/
            //非最大值抑制，当前点与其周围的 26 个点比较，val0 就是 N9[1][4]
            if( val0 > N9[0][0] && val0 > N9[0][1] && val0 > N9[0][2] &&
                val0 > N9[0][3] && val0 > N9[0][4] && val0 > N9[0][5] &&
                val0 > N9[0][6] && val0 > N9[0][7] && val0 > N9[0][8] &&
                val0 > N9[1][0] && val0 > N9[1][1] && val0 > N9[1][2] &&
                val0 > N9[1][3]                    && val0 > N9[1][5] &&
                val0 > N9[1][6] && val0 > N9[1][7] && val0 > N9[1][8] &&
                val0 > N9[2][0] && val0 > N9[2][1] && val0 > N9[2][2] &&
                val0 > N9[2][3] && val0 > N9[2][4] && val0 > N9[2][5] &&
                val0 > N9[2][6] && val0 > N9[2][7] && val0 > N9[2][8] )
            {
                /* Calculate the wavelet center coordinates for the maxima */
                //插值的需要，坐标位置要减 0.5
                float center_i = sum_i + (size-1)*0.5f;
                float center_j = sum_j + (size-1)*0.5f;
                //定义该候选特征点
                KeyPoint kpt( center_j, center_i, (float)sizes[layer],
                              -1, val0, octave, CV_SIGN(trace_ptr[j]) );

                /* Interpolate maxima location within the 3x3x3 neighbourhood   */
                //ds 为滤波器模板尺寸的变化率，即当前层尺寸与下一层尺寸的差值
                int ds = size - sizes[layer-1];
                //插值计算，interpolateKeypoint 函数在后面给出详细解释
                int interp_ok = interpolateKeypoint( N9, sampleStep, sampleStep, ds, kpt );

                /* Sometimes the interpolation step gives a negative size etc. */
                //如果得到合理的插值结果，保存该特征点
                if( interp_ok )
                {
                    /*printf( "KeyPoint %f %f %d\n", point.pt.x, point.pt.y, point.size );*/
                    cv::AutoLock lock(findMaximaInLayer_m);
                    keypoints.push_back(kpt);       //当前像素为特征点，保存该数据
                }
            }
        }
    }
```

```
    }
}
```

插值计算 interpolateKeypoint 函数：

```
/*
 * Maxima location interpolation as described in "Invariant Features from
 * Interest Point Groups" by Matthew Brown and David Lowe. This is performed by
 * fitting a 3D quadratic to a set of neighbouring samples.
 *
 * The gradient vector and Hessian matrix at the initial keypoint location are
 * approximated using central differences. The linear system Ax = b is then
 * solved, where A is the Hessian, b is the negative gradient, and x is the
 * offset of the interpolated maxima coordinates from the initial estimate.
 * This is equivalent to an iteration of Netwon's optimisation algorithm.
 *
 * N9 contains the samples in the 3x3x3 neighbourhood of the maxima
 * dx is the sampling step in x
 * dy is the sampling step in y
 * ds is the sampling step in size
 * point contains the keypoint coordinates and scale to be modified
 *
 * Return value is 1 if interpolation was successful, 0 on failure.
 */
//N9 为极值所在的 3×3×3 邻域
//dx，dy，ds 为横、纵坐标和尺度坐标的变化率（单位步长），dx 和 dy 就是采样间隔
static int
interpolateKeypoint( float N9[3][9], int dx, int dy, int ds, KeyPoint& kpt )
{
    //b 为公式 15 中的负值
    Vec3f b(-(N9[1][5]-N9[1][3])/2,  // Negative 1st deriv with respect to x
            -(N9[1][7]-N9[1][1])/2,  // Negative 1st deriv with respect to y
            -(N9[2][4]-N9[0][4])/2); // Negative 1st deriv with respect to s
    //A 为公式 14
    Matx33f A(
        N9[1][3]-2*N9[1][4]+N9[1][5],                  // 2nd deriv x, x
        (N9[1][8]-N9[1][6]-N9[1][2]+N9[1][0])/4, // 2nd deriv x, y
        (N9[2][5]-N9[2][3]-N9[0][5]+N9[0][3])/4, // 2nd deriv x, s
        (N9[1][8]-N9[1][6]-N9[1][2]+N9[1][0])/4, // 2nd deriv x, y
        N9[1][1]-2*N9[1][4]+N9[1][7],                  // 2nd deriv y, y
        (N9[2][7]-N9[2][1]-N9[0][7]+N9[0][1])/4, // 2nd deriv y, s
        (N9[2][5]-N9[2][3]-N9[0][5]+N9[0][3])/4, // 2nd deriv x, s
        (N9[2][7]-N9[2][1]-N9[0][7]+N9[0][1])/4, // 2nd deriv y, s
        N9[0][4]-2*N9[1][4]+N9[2][4]);                 // 2nd deriv s, s
    //x=A^-1b，公式 13 的结果，即偏移量
    Vec3f x = A.solve(b, DECOMP_LU);
```

```
    //判断偏移量是否合理，不能大于 1，否则会偏移到别的特征点位置上
    bool ok = (x[0] != 0 || x[1] != 0 || x[2] != 0) &&
        std::abs(x[0]) <= 1 && std::abs(x[1]) <= 1 && std::abs(x[2]) <= 1;

    if( ok )      //如果偏移量合理
    {
        //由公式 16，计算偏移后的位置
        kpt.pt.x += x[0]*dx;
        kpt.pt.y += x[1]*dy;
        //其实这里的 size 表示的是盒状滤波器模板的尺寸，但由于模板尺寸和图像尺度可以通过公式 9 转换，所有保存尺寸信息也无妨
        kpt.size = (float)cvRound( kpt.size + x[2]*ds );
    }
    return ok;
}
```

方向分配和确定描述符类：

```
struct SURFInvoker : ParallelLoopBody
{
    // ORI_RADIUS 表示 6s 圆邻域的采样半径长，ORI_WIN 表示扇形滑动窗口的张角，PATCH_SZ 表示 20s 方邻域的采样边长
    enum { ORI_RADIUS = 6, ORI_WIN = 60, PATCH_SZ = 20 };
    //构造函数
    SURFInvoker( const Mat& _img, const Mat& _sum,
                 vector<KeyPoint>& _keypoints, Mat& _descriptors,
                 bool _extended, bool _upright )
    {
        keypoints = &_keypoints;      //特征点
        descriptors = &_descriptors;      //描述符
        img = &_img;      //输入图像
        sum = &_sum;      //积分图像
        extended = _extended;      //描述符是 128 维还是 64 维
        upright = _upright;      //是否采用 U-SURF 算法

        // Simple bound for number of grid points in circle of radius ORI_RADIUS
        //以特征点为中心、半径为 6s 的圆邻域内，以 s 为采样间隔时所采样的像素数量
        //nOriSampleBound 为正方形内的像素数量，在这里是用该变量来限定实际的采样数量
        const int nOriSampleBound = (2*ORI_RADIUS+1)*(2*ORI_RADIUS+1);

        // Allocate arrays
        //分配数组空间
        apt.resize(nOriSampleBound);
        aptw.resize(nOriSampleBound);      //aptw 为 6s 圆邻域内的高斯加权系数
        DW.resize(PATCH_SZ*PATCH_SZ);      //DW 为 20s 方邻域内的高斯加权系数
```

```
        /* Coordinates and weights of samples used to calculate orientation */
        //得到 6s 圆邻域内的高斯加权函数（方差为 2.5s，与 Bay 原著取值不同）模板的系数，SURF_ORI_SIGMA = 2.5f;
        Mat G_ori = getGaussianKernel( 2*ORI_RADIUS+1, SURF_ORI_SIGMA, CV_32F );
        //实际 6s 圆邻域内的采样数
        nOriSamples = 0;
        //事先计算好 6s 圆邻域内所有采样像素的高斯加权系数
        for( int i = -ORI_RADIUS; i <= ORI_RADIUS; i++ )
        {
            for( int j = -ORI_RADIUS; j <= ORI_RADIUS; j++ )
            {
                if( i*i + j*j <= ORI_RADIUS*ORI_RADIUS )
                {
                    apt[nOriSamples] = cvPoint(i,j);      //相对于圆心的坐标位置
                    //高斯加权系数
                    aptw[nOriSamples++]      =      G_ori.at<float>(i+ORI_RADIUS,0)      *
G_ori.at<float>(j+ORI_RADIUS,0);
                }
            }
        }
        //判断实际的采样数量是否超过了最大限定数量
        CV_Assert( nOriSamples <= nOriSampleBound );

        /* Gaussian used to weight descriptor samples */
        //得到 20s 方邻域内的高斯加权函数（方差为 3.3s）模板的系数，SURF_DESC_SIGMA
= 3.3f
        Mat G_desc = getGaussianKernel( PATCH_SZ, SURF_DESC_SIGMA, CV_32F );
        //事先计算好 20s 方邻域内所有采样像素的高斯加权系数
        for( int i = 0; i < PATCH_SZ; i++ )
        {
            for( int j = 0; j < PATCH_SZ; j++ )
                DW[i*PATCH_SZ+j] = G_desc.at<float>(i,0) * G_desc.at<float>(j,0);
        }
    }
    //重载( )运算符
    void operator()(const Range& range) const
    {
        /* X and Y gradient wavelet data */
        //x 方向和 y 方向的 Haar 小波响应的突起部分的数量
        const int NX=2, NY=2;
        //dx_s 和 dy_s 的含义与盒状滤波器模板的含义相同，数组中五个元素的前四个元素表示突起部分左上角和右下角的坐标位置，第五个元素为权值
        const int dx_s[NX][5] = {{0, 0, 2, 4, -1}, {2, 0, 4, 4, 1}};
```

```
const int dy_s[NY][5] = {{0, 0, 4, 2, 1}, {0, 2, 4, 4, -1}};

// Optimisation is better using nOriSampleBound than nOriSamples for
// array lengths.  Maybe because it is a constant known at compile time
//以特征点为中心、半径为 6s 的圆邻域内，以 s 为采样间隔时所采样的像素数量
//nOriSampleBound 实为正方形内的像素数量，正方形的面积要大于圆形，所以为了保险起见，后面在定义变量的时候，用该值表示变量的数量
const int nOriSampleBound =(2*ORI_RADIUS+1)*(2*ORI_RADIUS+1);
//数组 X 保存 6s 圆邻域内的 Haar 小波响应的 x 方向梯度值，数组 Y 保存 y 方向梯度，数组 angle 保存角度
float X[nOriSampleBound], Y[nOriSampleBound], angle[nOriSampleBound];
//定义 20s 方邻域的二维数组
uchar PATCH[PATCH_SZ+1][PATCH_SZ+1];
//DX 和 DY 分别为 20s 方邻域内采样像素的 dx 和 dy
float DX[PATCH_SZ][PATCH_SZ], DY[PATCH_SZ][PATCH_SZ];
//由于矩阵 matX，matY 和_angle 的 rows 都为 1，所以这三个矩阵变量其实是矢量变量。它们的值分别为 X，Y 和 angle 数组的值
CvMat matX = cvMat(1, nOriSampleBound, CV_32F, X);
CvMat matY = cvMat(1, nOriSampleBound, CV_32F, Y);
CvMat _angle = cvMat(1, nOriSampleBound, CV_32F, angle);
//定义 20s 方邻域的矩阵变量
Mat _patch(PATCH_SZ+1, PATCH_SZ+1, CV_8U, PATCH);
//确定描述符是 128 维还是 64 维
int dsize = extended ? 128 : 64;
//k1 为 0，k2 为 N，即特征点的总数
int k, k1 = range.start, k2 = range.end;
//该变量代表特征点的最大尺度
float maxSize = 0;
//遍历所有特征点，找到特征点的最大尺度，其实找到的是盒状滤波器模板的最大尺寸
for( k = k1; k < k2; k++ )
{
    maxSize = std::max(maxSize, (*keypoints)[k].size);
}
//得到 20s 方邻域内的最大值，即带入尺度 s 的最大值
int imaxSize = std::max(cvCeil((PATCH_SZ+1)*maxSize*1.2f/9.0f), 1);
//在系统中为 20s 方邻域开辟一段包括了所有可能的 s 值的内存空间
Ptr<CvMat> winbuf = cvCreateMat( 1, imaxSize*imaxSize, CV_8U );
//遍历所有特征点
for( k = k1; k < k2; k++ )
{
    int i, j, kk, nangle;
    float* vec;
    SurfHF dx_t[NX], dy_t[NY];
```

```
KeyPoint& kp = (*keypoints)[k];       //当前特征点
float size = kp.size;       //特征点尺度，其实是盒状滤波器模板的尺寸
Point2f center = kp.pt;       //特征点的位置坐标
/* The sampling intervals and wavelet sized for selecting an orientation
  and building the keypoint descriptor are defined relative to 's' */
//由公式 9，把模板尺寸转换为图像尺度，得到真正的尺度信息
float s = size*1.2f/9.0f;
/* To find the dominant orientation, the gradients in x and y are
  sampled in a circle of radius 6s using wavelets of size 4s.
  We ensure the gradient wavelet size is even to ensure the
  wavelet pattern is balanced and symmetric around its center */
//得到确定方向时所用的 haar 小波响应的尺寸，即 4s
int grad_wav_size = 2*cvRound( 2*s );
//如果 haar 小波响应的尺寸大于输入图像的尺寸，则进入 if 语句
if( sum->rows < grad_wav_size || sum->cols < grad_wav_size )
{
    /* when grad_wav_size is too big,
     * the sampling of gradient will be meaningless
     * mark keypoint for deletion. */
    //进入该 if 语句，说明 haar 小波响应的尺寸太大，在这么大的尺寸下进行任何操作都是没有意义的，所以标注该特征点为无效的特征点，待后面的代码删除该特征点
    kp.size = -1;
    continue;       //继续下一个特征点的操作
}

float descriptor_dir = 360.f - 90.f;
if (upright == 0)       //不采用 U-SURF 算法
{
    //调整两个 Haar 小波响应的尺寸大小
    resizeHaarPattern( dx_s, dx_t, NX, 4, grad_wav_size, sum->cols );
    resizeHaarPattern( dy_s, dy_t, NY, 4, grad_wav_size, sum->cols );
    //遍历 6s 圆邻域内的所有采样像素
    for( kk = 0, nangle = 0; kk < nOriSamples; kk++ )
    {
        //x 和 y 为该采样像素在输入图像的坐标位置
        int x = cvRound( center.x + apt[kk].x*s - (float)(grad_wav_size-1)/2 );
        int y = cvRound( center.y + apt[kk].y*s - (float)(grad_wav_size-1)/2 );
        //判断该坐标是否超出了图像的边界
        if( y < 0 || y >= sum->rows - grad_wav_size ||
            x < 0 || x >= sum->cols - grad_wav_size )
            continue;       //超出了，则继续下一个采样像素的操作
        //积分图像的像素
        const int* ptr = &sum->at<int>(y, x);
```

```
                    //分别得到 x 方向和 y 方向的 Haar 小波响应值
                    float vx = calcHaarPattern( ptr, dx_t, 2 );
                    float vy = calcHaarPattern( ptr, dy_t, 2 );
                    //对 Haar 小波响应值进行高斯加权处理
                    X[nangle] = vx*aptw[kk];
                    Y[nangle] = vy*aptw[kk];
                    nangle++;       //为实际采样像素的总数计数
                }
                if( nangle == 0 )       // nangle == 0 表明 6s 圆邻域内没有采样像素
                {
                    // No gradient could be sampled because the keypoint is too
                    // near too one or more of the sides of the image. As we
                    // therefore cannot find a dominant direction, we skip this
                    // keypoint and mark it for later deletion from the sequence.
                    //这种情况说明该特征点与图像的边界过于接近，因此也要把它删除
                    kp.size = -1;
                    continue;       //继续下一个特征点的操作
                }
                //把 matX，matY 和_angle 的长度重新定义为实际的 6s 圆邻域内的采样点
数
                matX.cols = matY.cols = _angle.cols = nangle;
                //用 cvCartToPolar 函数（直角坐标转换为极坐标）求梯度坐标系下采样点
的角度，结果保存在矩阵_angle 对应的 angle 数组内
                cvCartToPolar( &matX, &matY, 0, &_angle, 1 );

                float bestx = 0, besty = 0, descriptor_mod = 0;
                //SURF_ORI_SEARCH_INC = 5
                //以 5 度的步长，遍历 360 度的圆周
                for( i = 0; i < 360; i += SURF_ORI_SEARCH_INC )
                {
                    //sumx 和 sumy 分别表示梯度坐标系下扇形滑动窗口内所有采样像素
的横、纵坐标之和，temp_mod 表示模
                    float sumx = 0, sumy = 0, temp_mod;
                    //遍历所有的采样像素
                    for( j = 0; j < nangle; j++ )
                    {
                        //得到当前采样点相对于旋转扇区的角度
                        int d = std::abs(cvRound(angle[j]) - i);
                        //判断 d 是否在正负 30 度之间，即判断当前点是否落在该扇形
滑动窗口内
                        if( d < ORI_WIN/2 || d > 360-ORI_WIN/2 )
                        {
                            //累计在该滑动窗口内的像素的横、纵坐标之和
                            sumx += X[j];
```

```
                    sumy += Y[j];
                }
            }
            //公式 17，求该滑动窗口内的模
            temp_mod = sumx*sumx + sumy*sumy;
            //判断该滑动窗口内的模是否为已计算过的最大值
            if( temp_mod > descriptor_mod )
            {
                //重新给模最大值，横、纵坐标之和赋值
                descriptor_mod = temp_mod;
                bestx = sumx;
                besty = sumy;
            }
        }
        //公式 18 和公式 19，求特征点的方向
        descriptor_dir = fastAtan2( -besty, bestx );
}
//为特征点的方向赋值，如果采用 U-SURF 算法，则它的方向恒为 270 度；
kp.angle = descriptor_dir;
//如果不需要生成描述符，则不进行 if 后面的操作
if( !descriptors || !descriptors->data )
    continue;
//以下部分为描述符的生成
/* Extract a window of pixels around the keypoint of size 20s */
// win_size 为 20s，即 20s 方邻域的真正尺寸
int win_size = (int)((PATCH_SZ+1)*s);
//确保 20s 方邻域的尺寸不超过事先设置好的内存空间
CV_Assert( winbuf->cols >= win_size*win_size );
//定义 20s 方邻域的矩阵
Mat win(win_size, win_size, CV_8U, winbuf->data.ptr);

if( !upright )      //不采用 U-SURF 算法
{
    //把特征点的方向角度转换为弧度形式
    descriptor_dir *= (float)(CV_PI/180);
    //方向角度的正弦值和余弦值
    float sin_dir = -std::sin(descriptor_dir);
    float cos_dir =    std::cos(descriptor_dir);

    /* Subpixel interpolation version (slower). Subpixel not required since
    the pixels will all get averaged when we scale down to 20 pixels */
    /*
    float w[] = { cos_dir, sin_dir, center.x,
    -sin_dir, cos_dir , center.y };
```

```
                CvMat W = cvMat(2, 3, CV_32F, w);
                cvGetQuadrangleSubPix( img, &win, &W );
                */

                float win_offset = -(float)(win_size-1)/2;
                //得到旋转校正以后的 20s 方邻域的左上角坐标
                float start_x = center.x + win_offset*cos_dir + win_offset*sin_dir;
                float start_y = center.y - win_offset*sin_dir + win_offset*cos_dir;
                //20s 方邻域内像素的首地址指针
                uchar* WIN = win.data;
#if 0
                // Nearest neighbour version (faster)
                for( i = 0; i < win_size; i++, start_x += sin_dir, start_y += cos_dir )
                {
                    float pixel_x = start_x;
                    float pixel_y = start_y;
                    for( j = 0; j < win_size; j++, pixel_x += cos_dir, pixel_y -= sin_dir )
                    {
                        int x = std::min(std::max(cvRound(pixel_x), 0), img->cols-1);
                        int y = std::min(std::max(cvRound(pixel_y), 0), img->rows-1);
                        WIN[i*win_size + j] = img->at<uchar>(y, x);
                    }
                }
#else
                int ncols1 = img->cols-1, nrows1 = img->rows-1;
                size_t imgstep = img->step;
                //找到 20s 方邻域内的所有像素
                for( i = 0; i < win_size; i++, start_x += sin_dir, start_y += cos_dir )
                {
                    //得到旋转校正后的 20s 方邻域的行的首像素坐标
                    double pixel_x = start_x;
                    double pixel_y = start_y;
                    for( j = 0; j < win_size; j++, pixel_x += cos_dir, pixel_y -= sin_dir )
                    {
                        //得到旋转校正后的 20s 方邻域的像素坐标
                        int ix = cvFloor(pixel_x), iy = cvFloor(pixel_y);
                        //判断该像素坐标是否超过了图像的边界
                        if( (unsigned)ix < (unsigned)ncols1 &&
                            (unsigned)iy < (unsigned)nrows1 )       //没有超过边界
                        {
                            //计算偏移量
                            float a = (float)(pixel_x - ix), b = (float)(pixel_y - iy);
                            //提取出图像的灰度值
                            const uchar* imgptr = &img->at<uchar>(iy, ix);
```

```
                        //由线性插值法计算 20s 方邻域内所对应的像素灰度值
                        WIN[i*win_size + j] = (uchar)
                            cvRound(imgptr[0]*(1.f - a)*(1.f - b) +
                                    imgptr[1]*a*(1.f - b) +
                                    imgptr[imgstep]*(1.f - a)*b +
                                    imgptr[imgstep+1]*a*b);
                    }
                    else      //超过了图像边界
                    {
                        //用边界处的灰度值代替超出部分的 20s 方邻域内的灰度值
                        int x = std::min(std::max(cvRound(pixel_x), 0), ncols1);
                        int y = std::min(std::max(cvRound(pixel_y), 0), nrows1);
                        WIN[i*win_size + j] = img->at<uchar>(y, x);
                    }
                }
            }
#endif
        }
        else      //采用 U-SURF 算法
        {
            // extract rect - slightly optimized version of the code above
            // TODO: find faster code, as this is simply an extract rect operation,
            //            e.g. by using cvGetSubRect, problem is the border processing
            // descriptor_dir == 90 grad
            // sin_dir == 1
            // cos_dir == 0

            float win_offset = -(float)(win_size-1)/2;
            //得到 20s 方邻域的左上角坐标
            int start_x = cvRound(center.x + win_offset);
            int start_y = cvRound(center.y - win_offset);
            uchar* WIN = win.data;
            //找到 20s 方邻域内的所有像素
            for( i = 0; i < win_size; i++, start_x++ )
            {
                //行的首像素坐标
                int pixel_x = start_x;
                int pixel_y = start_y;
                for( j = 0; j < win_size; j++, pixel_y-- )
                {
                    //x 和 y 为像素坐标
                    int x = MAX( pixel_x, 0 );
                    int y = MAX( pixel_y, 0 );
                    //确保坐标不超过图像边界
```

```
                    x = MIN( x, img->cols-1 );
                    y = MIN( y, img->rows-1 );
                    WIN[i*win_size + j] = img->at<uchar>(y, x);       //赋值
                }
            }
        }
        // Scale the window to size PATCH_SZ so each pixel's size is s. This
        // makes calculating the gradients with wavelets of size 2s easy
        //利用面积相关法，把边长为 20s 的正方形缩小为边长为 20 的正方形（尺度 s 一定是大于 1，所以是缩小，而不是扩大），也就是相当于对边长为 20s 的正方形进行采样间隔为 s 的等间隔采样
        resize(win, _patch, _patch.size(), 0, 0, INTER_AREA);

        // Calculate gradients in x and y with wavelets of size 2s
        //遍历 20s 方邻域内的所有采样像素
        for( i = 0; i < PATCH_SZ; i++ )
            for( j = 0; j < PATCH_SZ; j++ )
            {
                float dw = DW[i*PATCH_SZ + j];       //得到高斯加权系数
                //计算加权后的 dx 和 dy
                float vx = (PATCH[i][j+1] - PATCH[i][j] + PATCH[i+1][j+1] - PATCH[i+1][j])*dw;
                float vy = (PATCH[i+1][j] - PATCH[i][j] + PATCH[i+1][j+1] - PATCH[i][j+1])*dw;
                DX[i][j] = vx;
                DY[i][j] = vy;
            }

        // Construct the descriptor
        //描述符矢量变量
        vec = descriptors->ptr<float>(k);
        //描述符矢量变量清零
        for( kk = 0; kk < dsize; kk++ )
            vec[kk] = 0;
        double square_mag = 0;       //模的平方
        if( extended )       //128 维描述符
        {
            // 128-bin descriptor
            //遍历 20s 方邻域内的 4×4 个子区域
            for( i = 0; i < 4; i++ )
                for( j = 0; j < 4; j++ )
                {
                    //遍历子区域内的 25 个采样像素
                    for(int y = i*5; y < i*5+5; y++ )
                    {
                        for(int x = j*5; x < j*5+5; x++ )
```

```
                    {
                        float tx = DX[y][x], ty = DY[y][x];       //得到 dx 和 dy
                        if( ty >= 0 )       //dy≥0
                        {
                            vec[0] += tx;       //累加 dx
                            vec[1] += (float)fabs(tx);       //累加|dx|
                        } else {       //dy < 0
                            vec[2] += tx;       //累加 dx
                            vec[3] += (float)fabs(tx);       //累加|dx|
                        }
                        if ( tx >= 0 )       //dx≥0
                        {
                            vec[4] += ty;       //累加 dy
                            vec[5] += (float)fabs(ty);       //累加|dy|
                        } else {       //dx < 0
                            vec[6] += ty;       //累加 dy
                            vec[7] += (float)fabs(ty);       //累加|dy|
                        }
                    }
                }
                for( kk = 0; kk < 8; kk++ )
                    square_mag += vec[kk]*vec[kk];
                vec += 8;
            }
        }
        else       //64 维描述符
        {
            // 64-bin descriptor
            //遍历 20s 方邻域内的 4×4 个子区域
            for( i = 0; i < 4; i++ )
                for( j = 0; j < 4; j++ )
                {
                    //遍历子区域内的 25 个采样像素
                    for(int y = i*5; y < i*5+5; y++ )
                    {
                        for(int x = j*5; x < j*5+5; x++ )
                        {
                            float tx = DX[y][x], ty = DY[y][x];       //得到 dx 和 dy
                            //累加子区域内的 dx、dy、|dx|和|dy|
                            vec[0] += tx; vec[1] += ty;
                            vec[2] += (float)fabs(tx); vec[3] += (float)fabs(ty);
                        }
                    }
                    for( kk = 0; kk < 4; kk++ )
```

```
                        square_mag += vec[kk]*vec[kk];
                    vec+=4;
                }
            }

            // unit vector is essential for contrast invariance
            //为满足对比度不变性，即去除光照变化的影响，对描述符进行归一化处理
            vec = descriptors->ptr<float>(k);
            float scale = (float)(1./(sqrt(square_mag) + DBL_EPSILON));
            for( kk = 0; kk < dsize; kk++ )
                vec[kk] *= scale;           //公式 21
        }
    }

    // Parameters
    //成员变量
    const Mat* img;
    const Mat* sum;
    vector<KeyPoint>* keypoints;
    Mat* descriptors;
    bool extended;
    bool upright;

    // Pre-calculated values
    int nOriSamples;
    vector<Point> apt;
    vector<float> aptw;
    vector<float> DW;
};
```

三、SURF 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");

    SURF surf(3500.);       //设置行列式阈值为 3000

    vector<KeyPoint> key_points;

    Mat descriptors, mascara;
    Mat output_img;

    surf(img,mascara,key_points,descriptors);
    drawKeypoints(img, key_points, output_img, Scalar::all(-1),
DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

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

    return 0;
}
```

结果如图 10 所示：

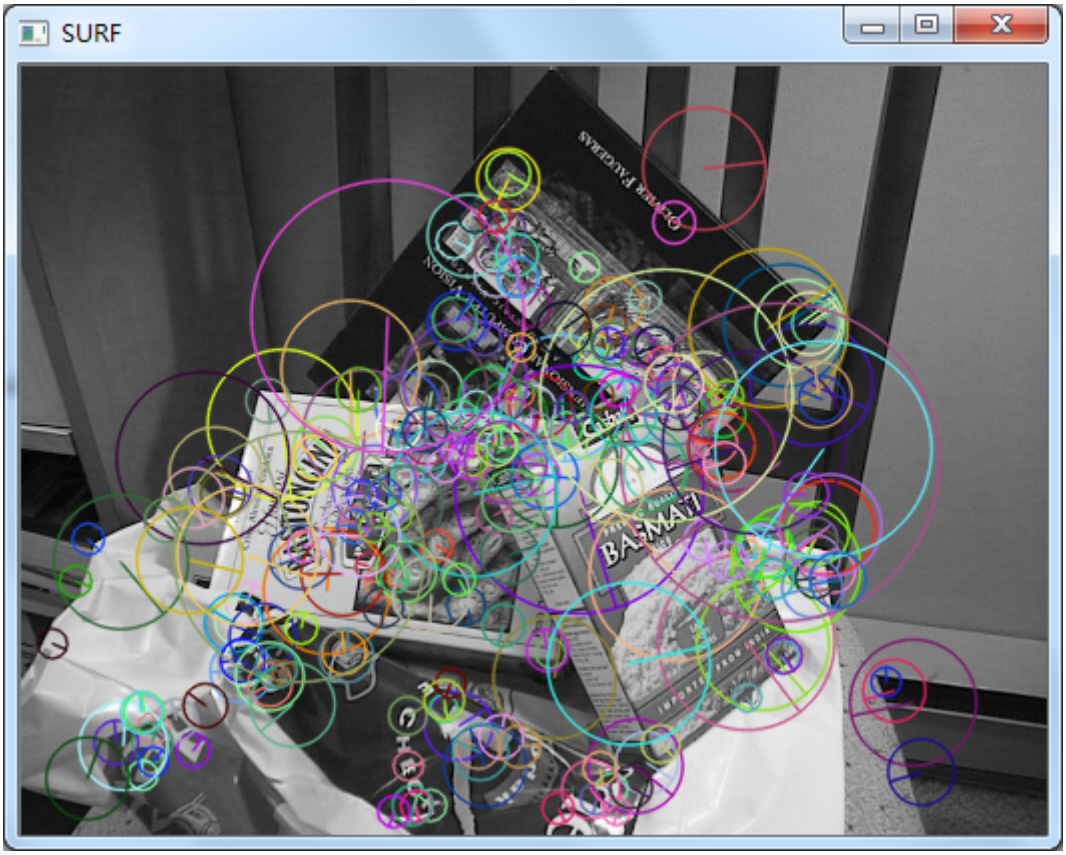

图 10 SURF 算法特征点检测

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)
{
    Mat img1 = imread("box_in_scene.png");
    Mat img2 = imread("box.png");

    SURF surf1(3000.), surf2(3000.);

    vector<KeyPoint> key_points1, key_points2;

    Mat descriptors1, descriptors2, mascara;

    surf1(img1,mascara,key_points1,descriptors1);
    surf2(img2,mascara,key_points2,descriptors2);

    BruteForceMatcher<L2<float>> matcher;
    vector<DMatch>matches;
    matcher.match(descriptors1,descriptors2,matches);

    std::nth_element(matches.begin(), // initial position
        matches.begin()+29, // position of the sorted element
        matches.end()); // end position

    matches.erase(matches.begin()+30, matches.end());

    namedWindow("SURF_matches");
    Mat img_matches;
    drawMatches(img1,key_points1,
        img2,key_points2,
        matches,
        img_matches,
        Scalar(255,255,255));
    imshow("SURF_matches",img_matches);
    waitKey(0);
```

```
    return 0;
}
```

结果如图 11 所示：

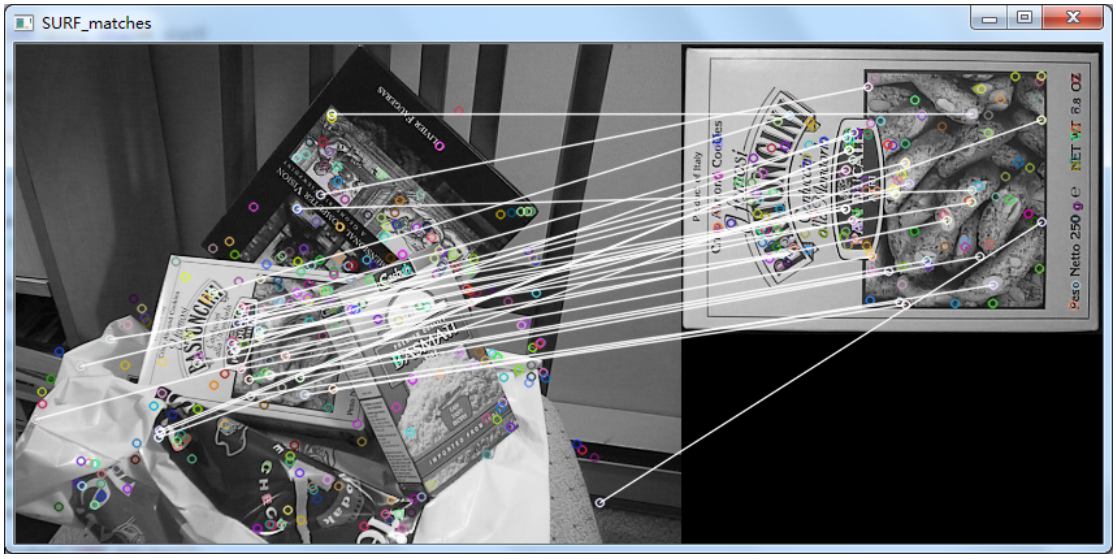

图 11 SURF 匹配结果