这几天研究SIFT和SURF算子，在网站上找了好多人的研究资料，以下直接转载过来做个备份

1. SIFT

SIFT（Scale-invariant feature transform）是一种检测局部特征的算法，该算法通过求一幅图中的特征点（interest points,or corner points）及其有关scale 和 orientation 的描述子得到特征并进行图像特征点匹配，获得了良好效果，详细解析如下：

算法描述

SIFT特征不只具有尺度不变性，即使改变旋转角度，图像亮度或拍摄视角，仍然能够得到好的检测效果。整个算法分为以下几个部分：

1. 构建尺度空间

这是一个初始化操作，尺度空间理论目的是模拟图像数据的多尺度特征。

高斯卷积核是实现尺度变换的唯一线性核，于是一副二维图像的尺度空间定义为：

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y)$$

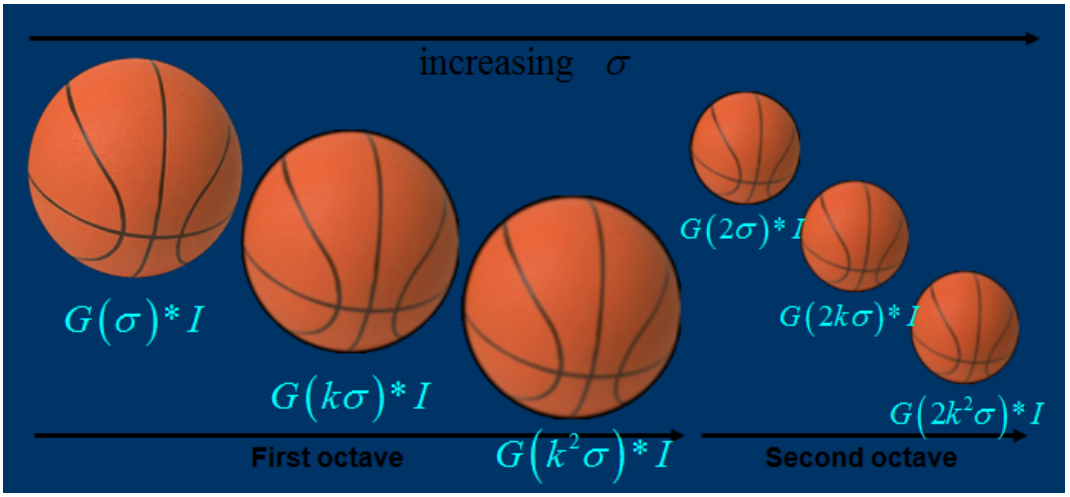
其中 G(x,y,σ) 是尺度可变高斯函数

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

（x，y）是空间坐标，是尺度坐标。σ大小决定图像的平滑程度，大尺度对应图像的概貌特征，小尺度对应图像的细节特征。大的σ值对应粗糙尺度(低分辨率)，反之，对应精细尺度(高分辨率)。为了有效的在尺度空间检测到稳定的关键点，提出了高斯差分尺度空间（DOG scale-space）。利用不同尺度的高斯差分核与图像卷积生成。

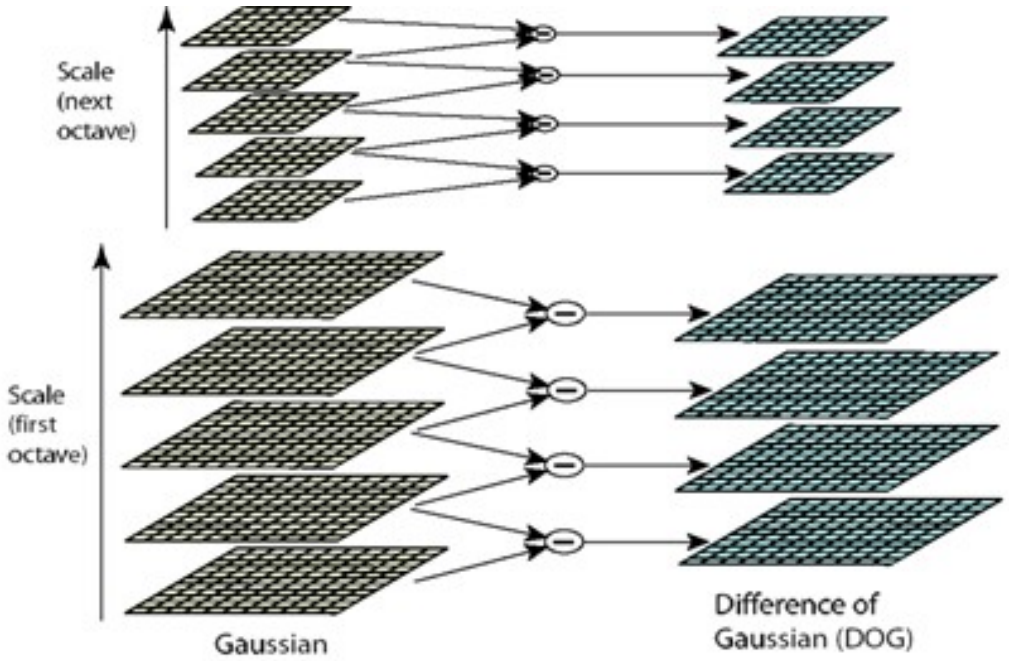
$$\begin{aligned} D(x,y,\sigma) &= (G(x,y,k\sigma) - G(x,y,\sigma)) * I(x,y) \\ &= L(x,y,k\sigma) - L(x,y,\sigma). \end{aligned}$$

下图所示不同σ下图像尺度空间：



关于尺度空间的理解说明：2kσ中的2是必须的，尺度空间是连续的。在　Lowe的论文中 ，将第0层的初始尺度定为1.6（最模糊），图片的初始尺度定为0.5（最清晰）. 在检测极值点前对原始图像的高斯平滑以致图像丢失高频信息，所以 Lowe 建议在建立尺度空间前首先对原始图像长宽扩展一倍，以保留原始图像信息，增加特征点数量。尺度越大图像越模糊。

图像金字塔的建立：对于一幅图像I,建立其在不同尺度(scale)的图像，也成为子八度（octave），这是为了scale-invariant，也就是在任何尺度都能够有对应的特征点，第一个子八度的scale为原图大小，后面每个octave为上一个octave降采样的结果，即原图的1/4（长宽分别减半），构成下一个子八度（高一层金字塔）。



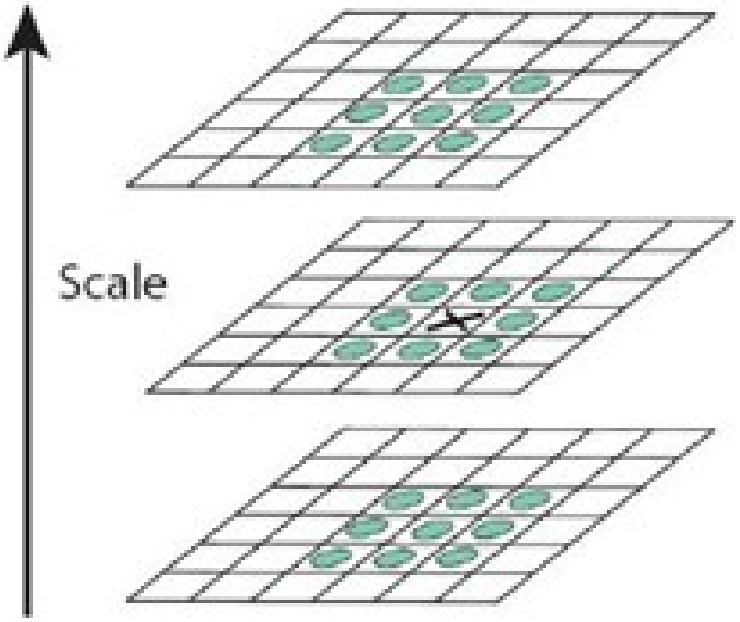
$$2^{i-1}(\sigma, k\sigma, k^2\sigma, \cdots k^{n-1}\sigma),\ k = 2^{1/s}$$

尺度空间的所有取值，i为octave的塔数（第几个塔），s为每塔层数

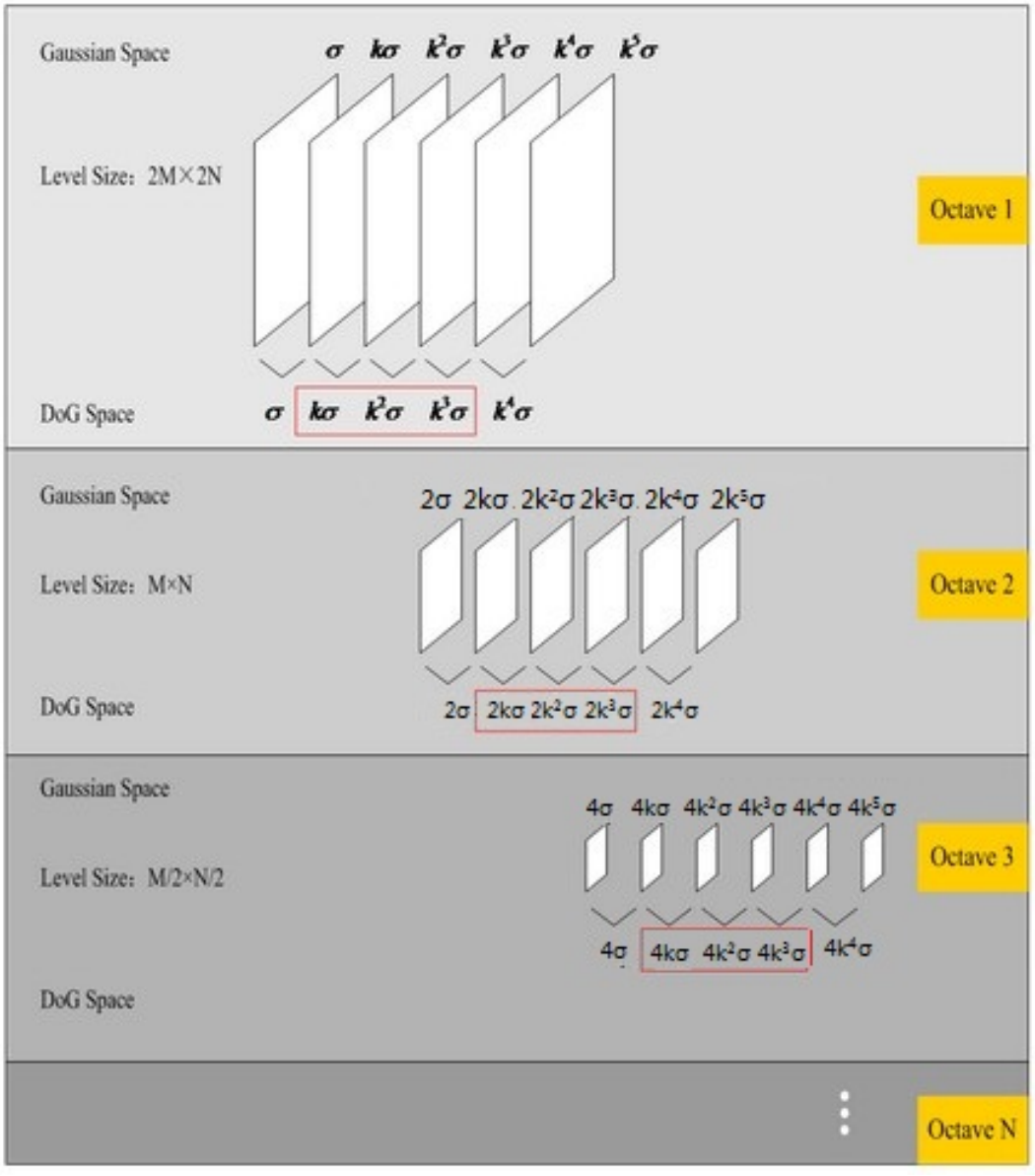
由图片*size*决定建几个塔，每塔几层图像*(S*一般为*3-5*层*)*。*0*塔的第*0*层是原始图像*(*或你*double*后的图像*)*，往上每一层是对其下一层进行*Laplacian*变换（高斯卷积，其中σ值渐大，例如可以是σ，k*σ，k*k*σ…），直观上看来越往上图片越模糊。塔间的图片是降采样关系，例如*1*塔的第*0*层可以由*0*塔的第*3*层*down sample*得到，然后进行与*0*塔类似的高斯卷积操作。

2.　LoG近似DoG找到关键点<检测DOG尺度空间极值点>

为了寻找尺度空间的极值点，每一个采样点要和它所有的相邻点比较，看其是否比它的图像域和尺度域的相邻点大或者小。如图所示，中间的检测点和它同尺度的8个相邻点和上下相邻尺度对应的9×2个点共26个点比较，以确保在尺度空间和二维图像空间都检测到极值点。 一个点如果在DOG尺度空间本层以及上下两层的26个领域中是最大或最小值时，就认为该点是图像在该尺度下的一个特征点,如图所示。



同一组中的相邻尺度（由于k的取值关系，肯定是上下层）之间进行寻找



s=3的情况

在极值比较的过程中，每一组图像的首末两层是无法进行极值比较的，**为了满足尺度变化的连续性**（下面有详解）
，我们在每一组图像的顶层继续用高斯模糊生成了 3 幅图像，高斯金字塔有每组S+3层图像。DOG金字塔每组有S+2层图像.

==========

这里有的童鞋不理解什么叫“为了满足尺度变化的连续性”，现在做仔细阐述：
假设s=3，也就是每个塔里有3层，则$k=2^{1/s}=2^{1/3}$，那么按照上图可得Gauss Space和DoG space 分别有3个（s个）和2个（s-1个）分量，在DoG space中，1st-octave两项分别是σ,kσ； 2nd-octave两项分别是2σ,2kσ;由于无法比较极值，我们必须在高斯空间继续添加高斯模糊项，使得形成*σ,kσ,k2σ,k3σ,k4σ这样就可以选择DoG space中的中间三项kσ,k2σ,k3σ（只有左右都有才能有极值），那么下一octave中（由上一层降采样获得）所得三项即为2kσ,2k2σ,2k3σ，其首项2kσ=2^{4/3}。刚好与上一octave末项k3σ=2^{3/3}尺度变化连续起来，所以每次要在Gaussian space添加3项，每组（塔）共S+3层图像，相应的DoG金字塔有S+2层图像。*

==========

使用Laplacian of Gaussian能够很好地找到找到图像中的兴趣点，但是需要大量的计算量，所以使用Difference of Gaussian图像的极大极小值近似寻找特征点.DOG算子计算简单，是尺度归一化的LoG算子的近似,有关DOG寻找特征点的介绍及方法详见
http://blog.csdn.net/abcjennifer/article/details/7639488， 极值点检测用的*Non-Maximal Suppression*。

3. 除去不好的特征点

通过拟和三维二次函数以精确确定关键点的位置和尺度（达到亚像素精度），同时去除低对比度的关键点和不稳定的边缘响应点(因为DoG算子会产生较强的边缘响应)，以增强匹配稳定性、提高抗噪声能力，在这里使用近似Harris Corner检测器。

①空间尺度函数泰勒展开式如下：

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

，对上式求导,并令其为0,得到精确的位置，得

$$\hat{\mathbf{x}} = -\frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}}.$$

②在已经检测到的特征点中,要去掉低对比度的特征点和不稳定的边缘响应点。去除低对比度的点：把公式(2)代入公式(1)，即在DoG Space的极值点处D(x)取值，只取前两项可得：

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D^T}{\partial \mathbf{x}}\hat{\mathbf{x}}.$$

若　　，该特征点就保留下来，否则丢弃。

③边缘响应的去除

一个定义不好的高斯差分算子的极值在横跨边缘的地方有较大的主曲率，而在垂直边缘的方向有较小的主曲率。主曲率通过一个2×2 的Hessian矩阵H求出：

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

导数由采样点相邻差估计得到。

D的主曲率和H的特征值成正比，令α为较大特征值，β为较小的特征值，则

$$\mathrm{Tr}(\mathbf{H}) = D_{xx} + D_{yy} = \alpha + \beta,$$

$$\mathrm{Det}(\mathbf{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta.$$

令α=γ β，则

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r}$$

(r + 1)2/r的值在两个特征值相等的时候最小，随着r的增大而增大，因此，为了检测主曲率是否在某域值r下，只需检测

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} < \frac{(r+1)^2}{r},$$

if　*(α + β)/ α β > (r+1)2/r, throw it out.*　在Lowe的文章中，取r＝10。

4.　给特征点赋值一个128维方向参数

上一步中确定了每幅图中的特征点，为每个特征点计算一个方向，依照这个方向做进一步的计算，　利用关键点邻域像素的梯度方向分布特性为每个关键点指定方向参数，使算子具备旋转不变性。

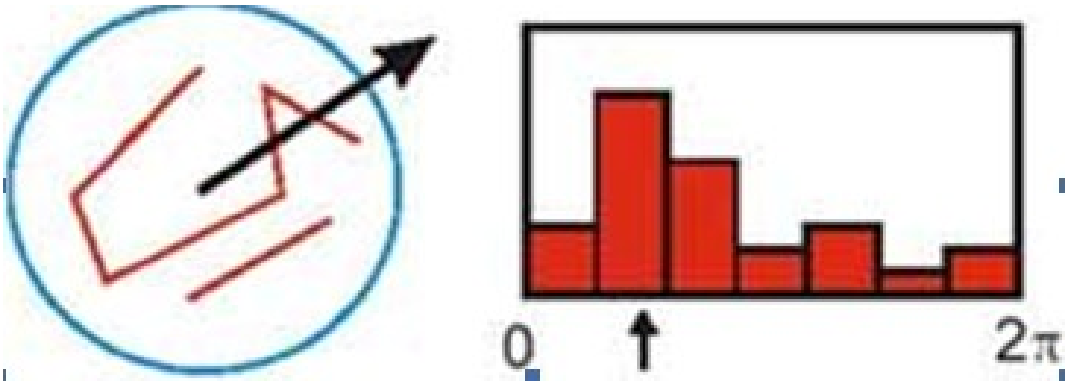
$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \alpha\tan 2((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

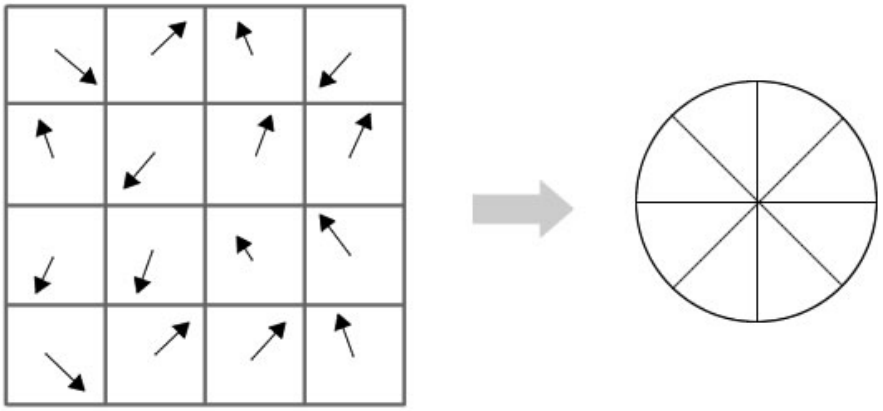
为(x,y)处梯度的模值和方向公式。其中L所用的尺度为每个关键点各自所在的尺度。至此，图像的关键点已经检测完毕，每个关键点有三个信息：位置，所处尺度、方向，由此可以确定一个SIFT特征区域。

梯度直方图的范围是0～360度，其中每10度一个柱，总共36个柱。随着距
中心点越远的领域其对直方图的贡献也响应减小.Lowe论文中还提到要使用高斯函数对直方图进行平滑，减少突变的影响。

在实际计算时，我们在以关键点为中心的邻域窗口内采样，并用直方图统计邻域像素的梯度方向。梯度直方图的范围是0～360度，其中每45度一个柱，总共8个柱，或者每10度一个柱，总共36个柱。Lowe论文中还提到要使用高斯函数对直方图进行平滑，减少突变的影响。直方图的峰值则代表了该关键点处邻域梯度的主方向，即作为该关键点的方向。

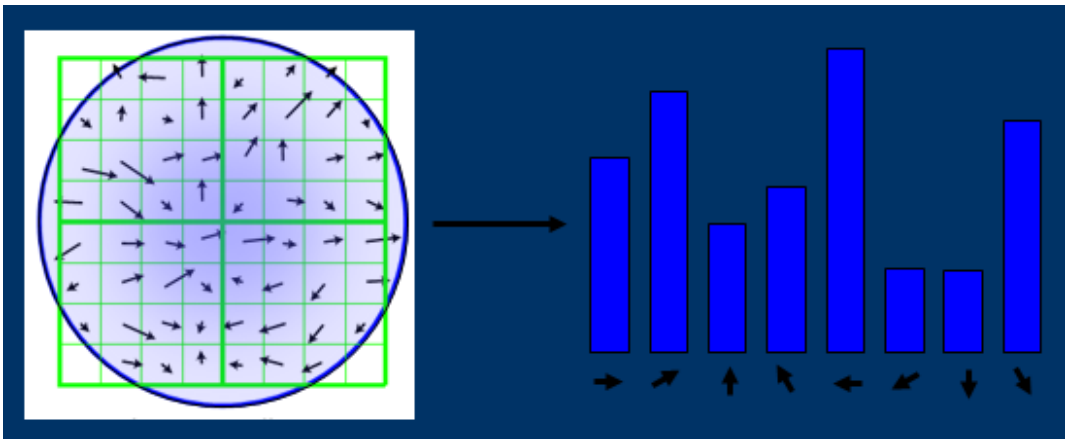


直方图中的峰值就是主方向，其他的达到最大值80%的方向可作为辅助方向



由梯度方向直方图确定主梯度方向

该步中将建立所有scale中特征点的描述子（128维）



Identify peak and assign orientation and sum of magnitude to key point.

The user may choose a threshold to exclude key points based on their assigned sum of magnitudes.

关键点描述子的生成步骤

通过对关键点周围图像区域分块，计算块内梯度直方图，生成具有独特性的向量，这个向量是该区域图像信息的一种抽象，具有唯一性。

5. 关键点描述子的生成

首先将坐标轴旋转为关键点的方向，以确保旋转不变性。以关键点为中心取8×8的窗口。

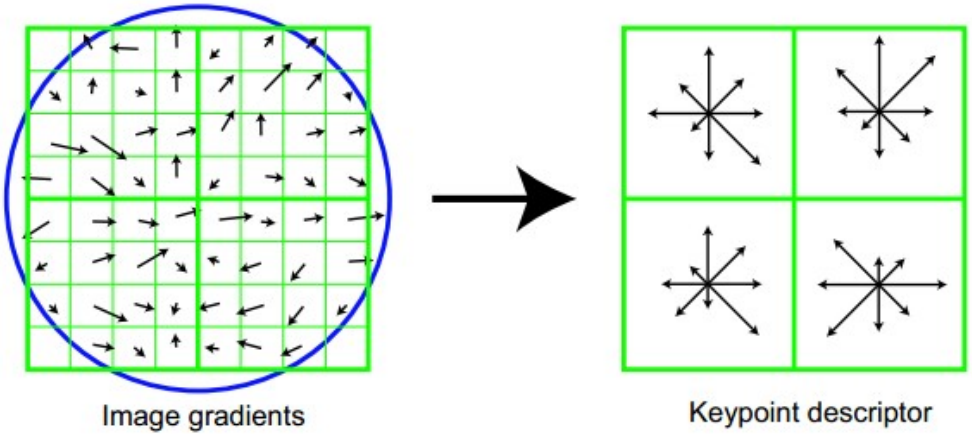
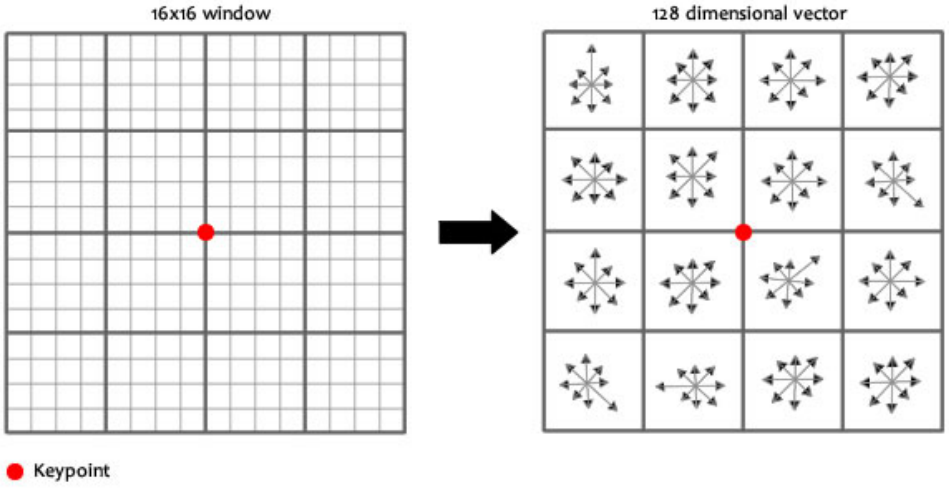


Figure.16*16的图中其中1/4的特征点梯度方向及scale，右图为其加权到8个主方向后的效果。

图左部分的中央为当前关键点的位置，每个小格代表关键点邻域所在尺度空间的一个像素，利用公式求得每个像素的梯度幅值与梯度方向，箭头方向代表该像素的梯度方向，箭头长度代表梯度模值，然后用高斯窗口对其进行加权运算。

图中蓝色的圈代表高斯加权的范围（越靠近关键点的像素梯度方向信息贡献越大）。然后在每4×4的小块上计算8个方向的梯度方向直方图，绘制每个梯度方向的累加值，即可形成一个种子点，如图右部分示。此图中一个关键点由2×2共4个种子点组成，每个种子点有8个方向向量信息。这种邻域方向性信息联合的思想增强了算法抗噪声的能力，同时对于含有定位误差的特征匹配也提供了较好的容错性。

计算keypoint周围的16*16的window中每一个像素的梯度，而且使用高斯下降函数降低远离中心的权重。



在每个4*4的1/16象限中，通过加权梯度值加到直方图8个方向区间中的一个，计算出一个梯度方向直方图。

这样就可以对每个feature形成一个4*4*8=128维的描述子，每一维都可以表示4*4个格子中一个的scale/orientation. 将这个向量归一化之后，就进一步去除了光照的影响。

5. 根据SIFT进行Match

生成了A、B两幅图的描述子，（分别是k1*128维和k2*128维），就将两图中各个scale（所有scale）的描述子进行匹配，匹配上128维即可表示两个特征点match上了。

实际计算过程中，为了增强匹配的稳健性，Lowe建议对每个关键点使用4×4共16个种子点来描述，这样对于一个关键点就可以产生128个数据，即最终形成128维的SIFT特征向量。此时SIFT特征向量已经去除了尺度变化、旋转等几何变形因素的影响，再继续将特征向量的长度归一化，则可以进一步去除光照变化的影响。 当两幅图像的SIFT特征向量生成后，下一步我们采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。取图像1中的某个关键点，并找出其与图像2中欧式距离最近的前两个关键点，在这两个关键点中，如果最近的距离除以次近的距离少于某个比例阈值，则接受这一对匹配点。降低这个比例阈值，SIFT匹配点数目会减少，但更加稳定。为了排除因为图像遮挡和背景混乱而产生的无匹配关系的关键点,Lowe提出了比较最近邻距离与次近邻距离的方法,距离比率ratio小于某个阈值的认为是正确匹配。因为对于错误匹

配,由于特征空间的高维性,相似的距离可能有大量其他的错误匹配,从而它的ratio值比较高。Lowe推荐ratio的阈值为0.8。但作者对大量任意存在尺度、旋转和亮度变化的两幅图片进行匹配，结果表明ratio取值在0. 4~0. 6之间最佳，小于0. 4的很少有匹配点，大于0. 6的则存在大量错误匹配点。(如果这个地方你要改进，最好给出一个匹配率和ration之间的关系图，这样才有说服力)作者建议ratio的取值原则如下:

ratio=0. 4 对于准确度要求高的匹配;

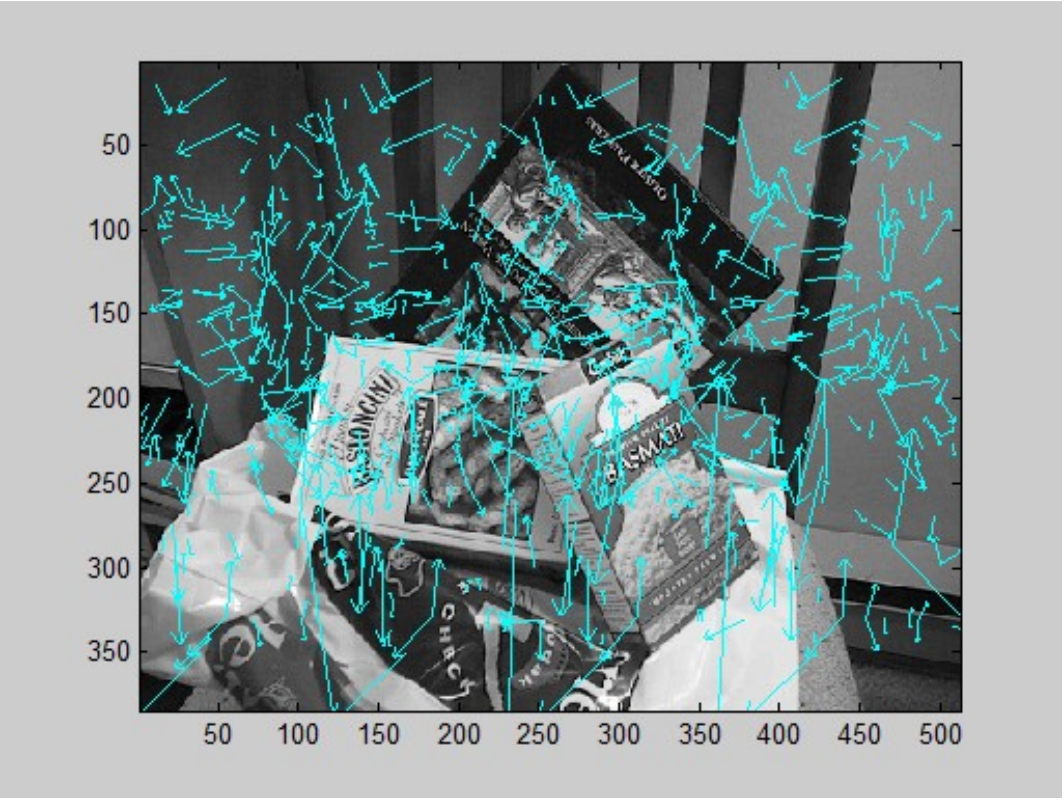
ratio=0. 6 对于匹配点数目要求比较多的匹配;

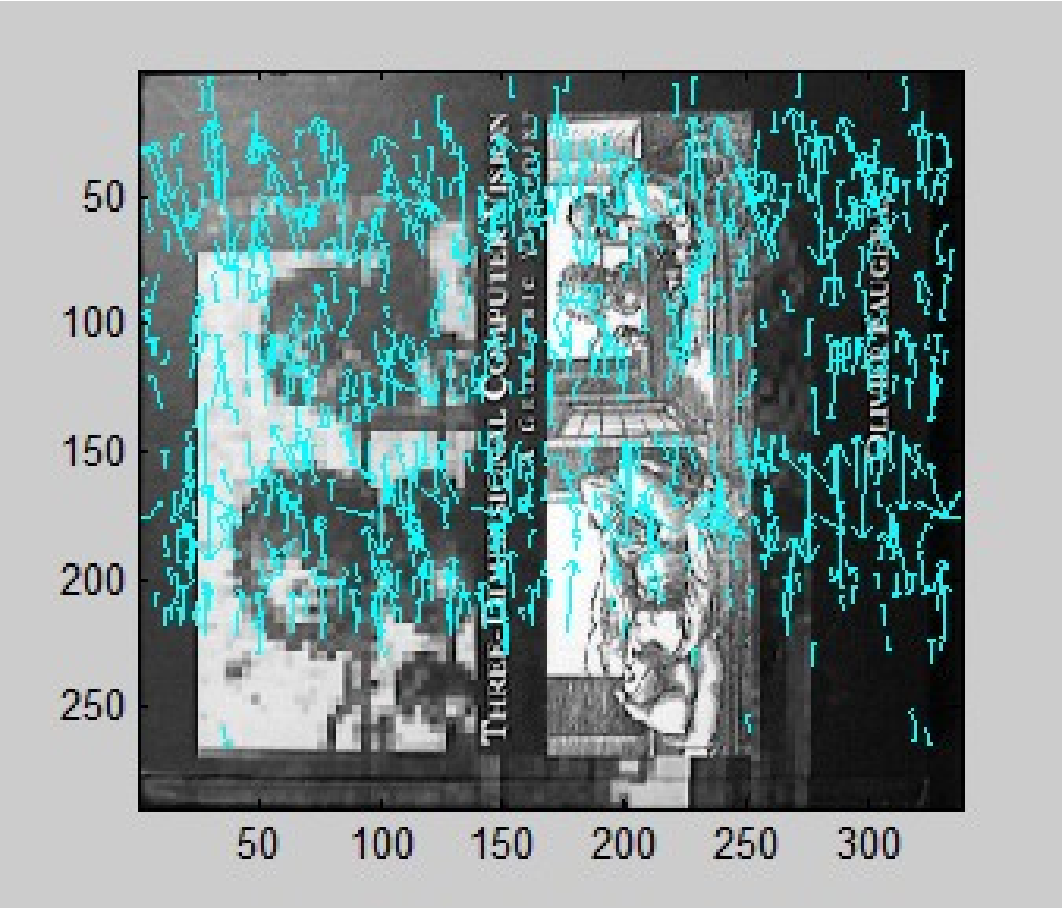
ratio=0. 5 一般情况下。

也可按如下原则:当最近邻距离<200时ratio=0. 6，反之ratio=0. 4。ratio的取值策略能排分错误匹配点。

当两幅图像的SIFT特征向量生成后，下一步我们采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。取图像1中的某个关键点，并找出其与图像2中欧式距离最近的前两个关键点，在这两个关键点中，如果最近的距离除以次近的距离少于某个比例阈值，则接受这一对匹配点。降低这个比例阈值，SIFT匹配点数目会减少，但更加稳定。

实验结果：





2.SURF

1.构建Hessian矩阵构造高斯金字塔尺度空间

其实surf构造的金字塔图像与sift有很大不同，就是因为这些不同才加快了其检测的速度。Sift采用的是DOG图像，而surf采用的是Hessian矩阵行列式近似值图像。Hessian矩阵是Surf算法的核心，为了方便运算，假设函数f(z，y)，Hessian矩阵H是由函数，偏导数组成。首先来看看图像中某个像素点的Hessian矩阵，如下：

$$H(f(x,y)) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

即每一个像素点都可以求出一个Hessian矩阵。

H矩阵判别式为：

$$det(H) = \frac{\partial^2 f}{\partial x^2}\frac{\partial^2 f}{\partial y^2} - (\frac{\partial^2 f}{\partial x \partial y})^2$$

判别式的值是H矩阵的特征值，可以利用判定结果的符号将所有点分类，根据判别式取值正负，来判别该点是或不是极值点。
在SURF算法中，用图像像素l(x，y)即为函数值f(x，y)，选用二阶标准高斯函数作为滤波器，通过特定核间的卷积计算二阶偏导数，这样便能计算出H矩阵的三个矩阵元素L_xx,L_xy,L_yy 从而计算出H矩阵：

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

但是由于我们的特征点需要具备尺度无关性，所以在进行Hessian矩阵构造前，需要对其进行高斯滤波。
这样，经过滤波后在进行Hessian的计算，其公式如下：

$$L(x,t)=G(t)\cdot I(x,t)$$

L(x,t)是一幅图像在不同解析度下的表示，可以利用高斯核G(t)与图像函数 I(x) 在点x的卷积来实现，其中高斯核G(t)：

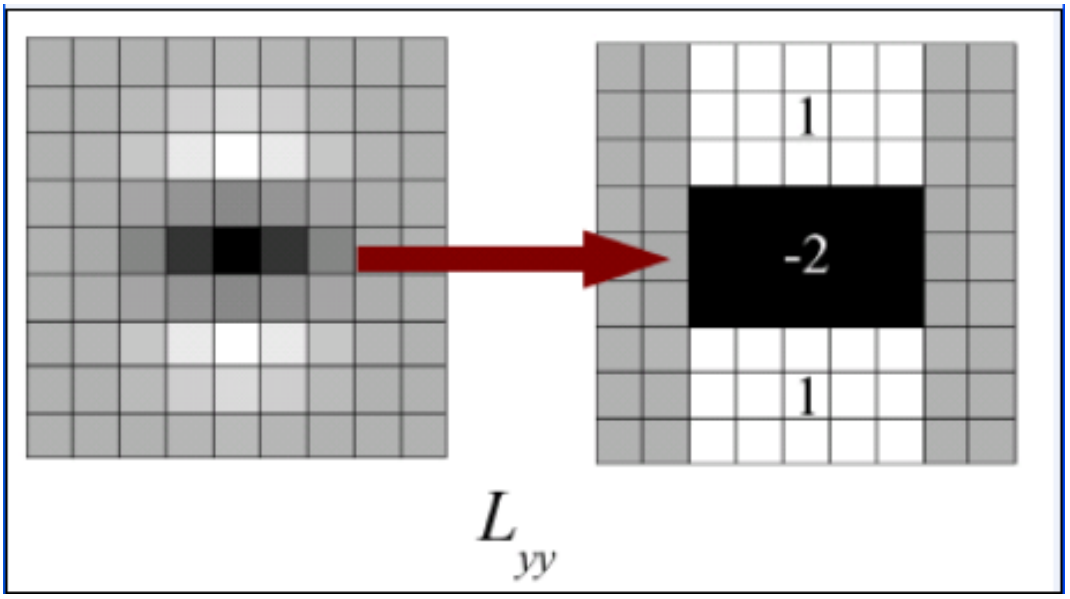
$$G(t)=\frac{\partial^2 g(t)}{\partial x^2}$$

g(x)为高斯函数，t为高斯方差。通过这种方法可以为图像中每个像素计算出其H行列式的决定值，并用这个值来判别特征点。为方便应用，Herbert Bay提出用近似值现代替L(x,t)。为平衡准确值与近似值间的误差引入权值叫，权值随尺度变化，则H矩阵判别式可表示为：

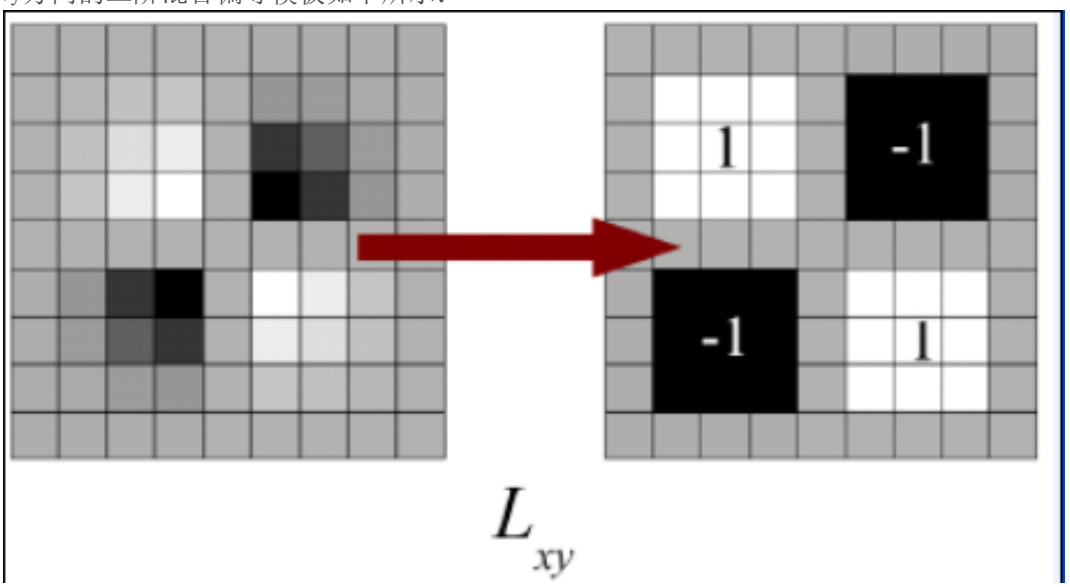
$$det(H_{approx})=D_{xx}D_{yy}-(0.9D_{xy})^2$$

其中0.9是作者给出的一个经验值，其实它是有一套理论计算的，具体去看surf的英文论文。

由于求Hessian时要先高斯平滑，然后求二阶导数，这在离散的像素点是用模板卷积形成的，这2中操作合在一起用一个模板代替就可以了，比如说y方向上的模板如下：

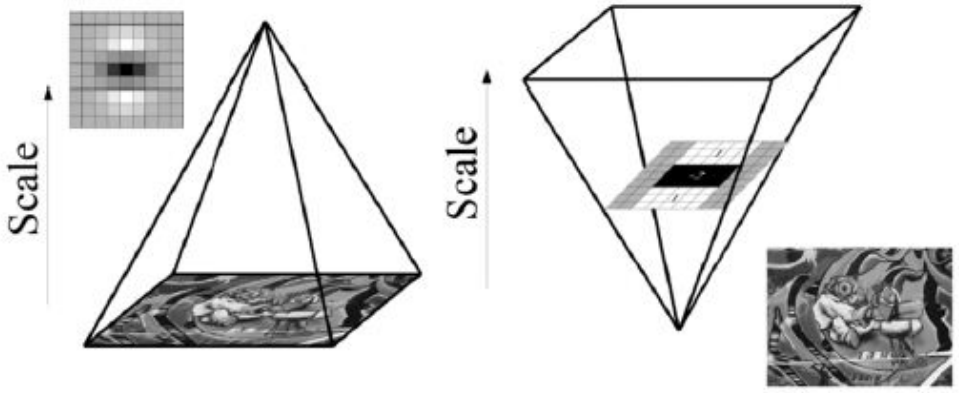


该图的左边即用高斯平滑然后在y方向上求二阶导数的模板，为了加快运算用了近似处理，其处理结果如右图所示，这样就简化了很多。并且右图可以采用积分图来运算，大大的加快了速度，关于积分图的介绍，可以去查阅相关的资料。
同理，x和y方向的二阶混合偏导模板如下所示：



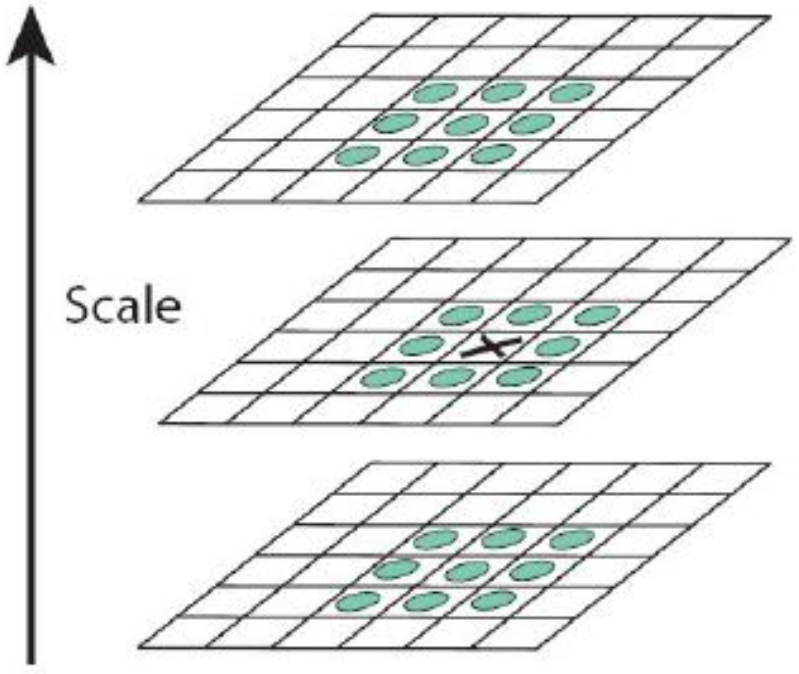
上面讲的这么多只是得到了一张近似hessian行列式图，这类似sift中的DOG图，但是在金字塔图像中分为很多层，每一层叫做一个octave，每一个octave中又有几张尺度不同的图片。在sift算法中，同一个octave层中的图片尺寸(即大小)相同，但是尺度(即模糊程度)不同，而不同的octave层中的图片尺寸大小也不相同，因为它是由上一层图片降采样得到的。在进行高斯模糊时，sift的高斯模板大小是始终不变的，只是在不同的octave之间改变图片的大小。而在surf中，图片的大小是一直不变的，不同的octave层得到的待检测图片是改变高斯模糊尺寸大小得到的，当然了，同一个octave中个的图片用到的高斯模板尺度也不同。算法允许尺度空间多层图像同时被处理，不需对图像进行二次抽样，从而提高算法性能。左图是传统方式建立一个如图所示的金字塔结构，图像的寸是变化的，并且运 算会反复使用高斯函数对子层进行平滑处理，右图说明

Surf算法使原始图像保持不变而只改变滤波器大小。Surf采用这种方法节省了降采样过程，其处理速度自然也就提上去了。其金字塔图像如下所示：



2. 利用非极大值抑制初步确定特征点

此步骤和sift类似，将经过hessian矩阵处理过的每个像素点与其3维领域的26个点进行大小比较，如果它是这26个点中的最大值或者最小值，则保留下来，当做初步的特征点。检测过程中使用与该尺度层图像解析度相对应大小的滤波器进行检测，以3×3的滤波器为例，该尺度层图像中9个像素点之一图2检测特征点与自身尺度层中其余8个点和在其之上及之下的两个尺度层9个点进行比较，共26个点，图2中标记‘x’的像素点的特征值若大于周围像素则可确定该点为该区域的特征点。



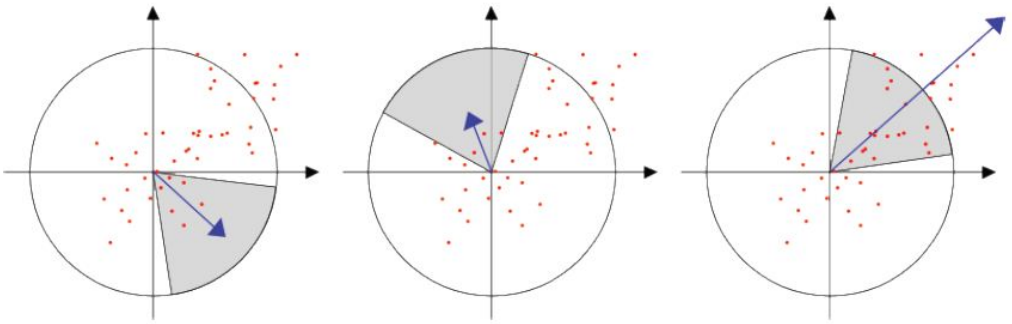
3. 精确定位极值点

这里也和sift算法中的类似，采用3维线性插值法得到亚像素级的特征点，同时也去掉那些值小于一定阈值的点，增加极值使检测到的特征点数量减少，最终只有几个特征最强点会被检测出来。

4. 选取特征点的主方向。

这一步与sift也大有不同。Sift选取特征点主方向是采用在特征点领域内统计其梯度直方图，取直方图bin值最大的以及超过最大bin值80%的那些方向做为特征点的主方向。

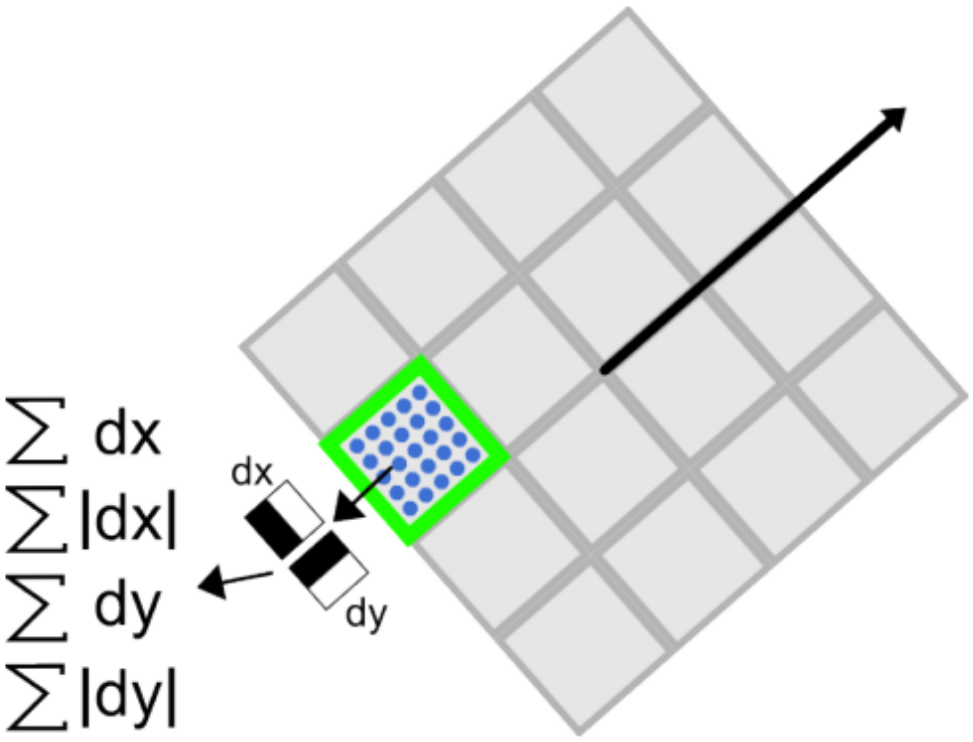
而在surf中，不统计其梯度直方图，而是统计特征点领域内的harr小波特征。即在特征点的领域(比如说，半径为6s的圆内，s为该点所在的尺度)内，统计60度扇形内所有点的水平haar小波特征和垂直haar小波特征总和，haar小波的尺寸变长为4s，这样一个扇形得到了一个值。然后60度扇形以一定间隔进行旋转，最后将最大值那个扇形的方向作为该特征点的主方向。该过程的示意图如下：



5. 构造surf特征点描述算子

在sift中，是在特征点周围取16*16的邻域，并把该领域化为4*4个的小区域，每个小区域统计8个方向梯度，最后得到4*4*8=128维的向量，该向量作为该点的sift描述子。

在surf中，也是在特征点周围取一个正方形框，框的边长为20s(s是所检测到该特征点所在的尺度)。该框带方向，方向当然就是第4步检测出来的主方向了。然后把该框分为16个子区域，每个子区域统计25个像素的水平方向和垂直方向的haar小波特征，这里的水平和垂直方向都是相对主方向而言的。该haar小波特征为水平方向值之和，水平方向绝对值之和，垂直方向之和，垂直方向绝对值之和。该过程的示意图如下所示：



这样每个小区域就有4个值，所以每个特征点就是16*4=64维的向量，相比sift而言，少了一半，这在特征匹配过程中会大大加快匹配速度。

6.结束语

Surf采用Henssian矩阵获取图像局部最值还是十分稳定的，但是在求主方向阶段太过于依赖局部区域像素的梯度方向，有可能使得找到的主方向不准确，后面的特征向量提取以及匹配都严重依赖于主方向，即使不大偏差角度也可以造成后面特征匹配的放大误差，从而匹配不成功；另外图像金字塔的层取 得不足够紧密也会使得尺度有误差，后面的特征向量提取同样依赖相应的尺度，发明者在这个问题上的折中解决方法是取适量的层然后进行插值。

Sift是一种只 利用到灰度性质的算法，忽略了色彩信息，后面又出现了几种据说比Surf更稳定的描述器其中一些利用到了色彩信息，让我们拭目以待吧。

代码：

来源：OpenCV/sample/c中的find_obj.cpp代码

需仔细注意：

1.定位部分：通过透视变换，画出了目标在图像中的位置，但是这么做会浪费很多时间，可以改进：

2.flann寻找最近的临近Keypoints：

首先，利用图像，构建多维查找树，然后，利用Knn算法找到最近的Keypoints（KNN算法：http://blog.csdn.net/sangni007/article/details/7482890）

[cpp]view plaincopy

```
1.
//Constructs a nearest neighbor search index for a given dataset
2. //利用m_image构造 a set of randomized kd-trees 一系列随机多维检索树；
3.
cv::flann::Index flann_index(m_image, cv::flann::KDTreeIndexParams(4));  // using 4 randomized kdtrees
4. //利用Knn近邻算法检索m_object；结果存入 m_indices, m_dists;
5.
flann_index.knnSearch(m_object, m_indices, m_dists, 2, cv::flann::SearchParams(64) ); // maximum number of leafs checked
```

flann算法有很多功能，

文档：

http://opencv.itseez.com/modules/flann/doc/flann_fast_approximate_nearest_neighbor_search.html?highlight=flann#fast-approximate-nearest-neighbor-search

[cpp]view plaincopy

```
1. /*
2.  * A Demo to OpenCV Implementation of SURF
```

```
3.  * Further Information Refer to "SURF: Speed-
Up Robust Feature"
4.  * Author: Liu Liu
5.  * liuliu.1987+opencv@gmail.com
6.  */
7. #include "opencv2/objdetect/objdetect.hpp"
8. #include "opencv2/features2d/features2d.hpp"
9. #include "opencv2/highgui/highgui.hpp"
10. #include "opencv2/calib3d/calib3d.hpp"
11. #include "opencv2/imgproc/imgproc_c.h"#include
12. #include
13. #include using namespace std;
14. void help()
15. {
16.     printf(
17.
"This program demonstrated the use of the SURF Detector and Desc
riptor using\n"
18.
"either FLANN (fast approx nearst neighbor classification) or br
ute force matching\n"
19. "on planar objects.\n"
20. "Usage:\n"
21.
"./find_obj  , default is box.png  and box_in_scene.png\n\n");
22. return;
23. }// define whether to use approximate nearest-
neighbor search
24. #define USE_FLANN
25.
IplImage* image = 0;double compareSURFDescriptors( constfloat* d
1, constfloat* d2, double best, int length )
26. {
27. double total_cost = 0;
28.     assert( length % 4 == 0 );
29. for( int i = 0; i < length; i += 4 )
30.     {
31. double t0 = d1[i  ] - d2[i  ];
32. double t1 = d1[i+1] - d2[i+1];
```

```
33. double t2 = d1[i+2] - d2[i+2];
34. double t3 = d1[i+3] - d2[i+3];
35.         total_cost += t0*t0 + t1*t1 + t2*t2 + t3*t3;
36. if( total_cost > best )
37. break;
38.     }
39. return total_cost;
40. }
41. int naiveNearestNeighbor( constfloat* vec, int laplacian,
42. const CvSeq* model_keypoints,
43. const CvSeq* model_descriptors )
44. {
45. int length = (int)(model_descriptors-
>elem_size/sizeof(float));
46. int i, neighbor = -1;
47. double d, dist1 = 1e6, dist2 = 1e6;
48.     CvSeqReader reader, kreader;
49.     cvStartReadSeq( model_keypoints, &kreader, 0 );
50.
    cvStartReadSeq( model_descriptors, &reader, 0 );    for( i =
 0; i < model_descriptors->total; i++ )
51.     {
52. const CvSURFPoint* kp = (const CvSURFPoint*)kreader.ptr;
53. constfloat* mvec = (constfloat*)reader.ptr;
54.      CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
55.         CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
56. if( laplacian != kp->laplacian )
57. continue;
58.
        d = compareSURFDescriptors( vec, mvec, dist2, length );

59. if( d < dist1 )
60.         {
61.             dist2 = dist1;
62.             dist1 = d;
63.             neighbor = i;
64.         }
65. elseif ( d < dist2 )
66.             dist2 = d;
```

```
67.     }
68. if ( dist1 < 0.6*dist2 )
69. return neighbor;
70. return -1;
71. }//用于找到两幅图像之间匹配的点对，并把匹配的点对存储在 ptpairs 向量中，其中物体(object)图像的特征点
72. //及其相应的描述器（局部特征）分别存储在 objectKeypoints 和 objectDescriptors，场景(image)图像的特
73. //征点及其相应的描述器（局部特征）分别存储在 imageKeypoints 和 imageDescriptors
74.
void findPairs( const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
75.
const CvSeq* imageKeypoints, const CvSeq* imageDescriptors, vector<int>& ptpairs )
76. {
77. int i;
78.     CvSeqReader reader, kreader;
79.     cvStartReadSeq( objectKeypoints, &kreader );
80.     cvStartReadSeq( objectDescriptors, &reader );
81.     ptpairs.clear();    for( i = 0; i < objectDescriptors->total; i++ )
82.     {
83. const CvSURFPoint* kp = (const CvSURFPoint*)kreader.ptr;
84. constfloat* descriptor = (constfloat*)reader.ptr;
85.         CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
86.         CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
87. int nearest_neighbor = naiveNearestNeighbor( descriptor, kp->laplacian, imageKeypoints, imageDescriptors );
88. if( nearest_neighbor >= 0 )
89.         {
90.             ptpairs.push_back(i);
91.             ptpairs.push_back(nearest_neighbor);
92.         }
93.     }
94. }//Fast Library for Approximate Nearest Neighbors (FLANN)
```

```
95.
void flannFindPairs( const CvSeq*, const CvSeq* objectDescriptor
s,
96.
const CvSeq*, const CvSeq* imageDescriptors, vector<int>& ptpair
s )
97. {
98. int length = (int)(objectDescriptors-
>elem_size/sizeof(float));    cv::Mat m_object(objectDescriptors
->total, length, CV_32F);
99.  cv::Mat m_image(imageDescriptors->total, length, CV_32F);
100. // copy descriptors
101.     CvSeqReader obj_reader;
102. float* obj_ptr = m_object.ptr<float>(0);
103.     cvStartReadSeq( objectDescriptors, &obj_reader );
104. //objectDescriptors to m_object
105. for(int i = 0; i < objectDescriptors->total; i++ )
106.     {
107. constfloat* descriptor = (constfloat*)obj_reader.ptr;
108.         CV_NEXT_SEQ_ELEM( obj_reader.seq-
>elem_size, obj_reader );
109.
        memcpy(obj_ptr, descriptor, length*sizeof(float));
110.         obj_ptr += length;
111.     }
112. //imageDescriptors to m_image
113.     CvSeqReader img_reader;
114. float* img_ptr = m_image.ptr<float>(0);
115.     cvStartReadSeq( imageDescriptors, &img_reader );
116. for(int i = 0; i < imageDescriptors->total; i++ )
117.     {
118. constfloat* descriptor = (constfloat*)img_reader.ptr;
119.         CV_NEXT_SEQ_ELEM( img_reader.seq-
>elem_size, img_reader );
120.
        memcpy(img_ptr, descriptor, length*sizeof(float));
121.         img_ptr += length;
122.     }    // find nearest neighbors using FLANN
```

```
123.      cv::Mat m_indices(objectDescriptors-
>total, 2, CV_32S);
124.      cv::Mat m_dists(objectDescriptors->total, 2, CV_32F);
125.
//Constructs a nearest neighbor search index for a given dataset
126. //利用m_image构造 a set of randomized kd-trees 一系列随机多维检
索树;
127.
    cv::flann::Index flann_index(m_image, cv::flann::KDTreeIndex
Params(4));  // using 4 randomized kdtrees
128. //利用Knn近邻算法检索m_object; 结果存入 m_indices, m_dists;
129.
    flann_index.knnSearch(m_object, m_indices, m_dists, 2, cv::f
lann::SearchParams(64) ); // maximum number of leafs checked
int* indices_ptr = m_indices.ptr(0);
130. float* dists_ptr = m_dists.ptr<float>(0);
131. for (int i=0;i
132.  {
133. if (dists_ptr[2*i]<0.6*dists_ptr[2*i+1])
134.   {
135.       ptpairs.push_back(i);
136.       ptpairs.push_back(indices_ptr[2*i]);
137.      }
138.     }
139. }//用于寻找物体(object)在场景(image)中的位置,位置信息保存在参数
dst_corners中, 参数src_corners由物
140. //体(object的width几height等决定, 其他部分参数如上findPairs
141. /* a rough implementation for object location */
142.
int locatePlanarObject( const CvSeq* objectKeypoints, const CvSe
q* objectDescriptors,
143.
const CvSeq* imageKeypoints, const CvSeq* imageDescriptors,
144. const CvPoint src_corners[4], CvPoint dst_corners[4] )
145. {
146. double h[9];
147.      CvMat _h = cvMat(3, 3, CV_64F, h);
148.      vector<int> ptpairs;
149.      vector pt1, pt2;
```

```
150.     CvMat _pt1, _pt2;
151. int i, n;#ifdef USE_FLANN
152.
    flannFindPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
153. #else
154.
    findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
155. #endif    n = (int)(ptpairs.size()/2);
156. if( n < 4 )
157. return 0;    pt1.resize(n);
158.     pt2.resize(n);
159. for( i = 0; i < n; i++ )
160.     {
161.
        pt1[i] = ((CvSURFPoint*)cvGetSeqElem(objectKeypoints,ptpairs[i*2]))->pt;
162.
        pt2[i] = ((CvSURFPoint*)cvGetSeqElem(imageKeypoints,ptpairs[i*2+1]))->pt;
163.     }    _pt1 = cvMat(1, n, CV_32FC2, &pt1[0] );
164.     _pt2 = cvMat(1, n, CV_32FC2, &pt2[0] );
165. if( !cvFindHomography( &_pt1, &_pt2, &_h, CV_RANSAC, 5 ))//计算两个平面之间的透视变换
166. return 0;    for( i = 0; i < 4; i++ )
167.     {
168. double x = src_corners[i].x, y = src_corners[i].y;
169. double Z = 1./(h[6]*x + h[7]*y + h[8]);
170. double X = (h[0]*x + h[1]*y + h[2])*Z;
171. double Y = (h[3]*x + h[4]*y + h[5])*Z;
172.         dst_corners[i] = cvPoint(cvRound(X), cvRound(Y));
173.     }    return 1;
174. }
175. int main(int argc, char** argv)
176. {
177. //物体(object)和场景(scene)的图像向来源
178.
constchar* object_filename = argc == 3 ? argv[1] : "D:/src.jpg";
```

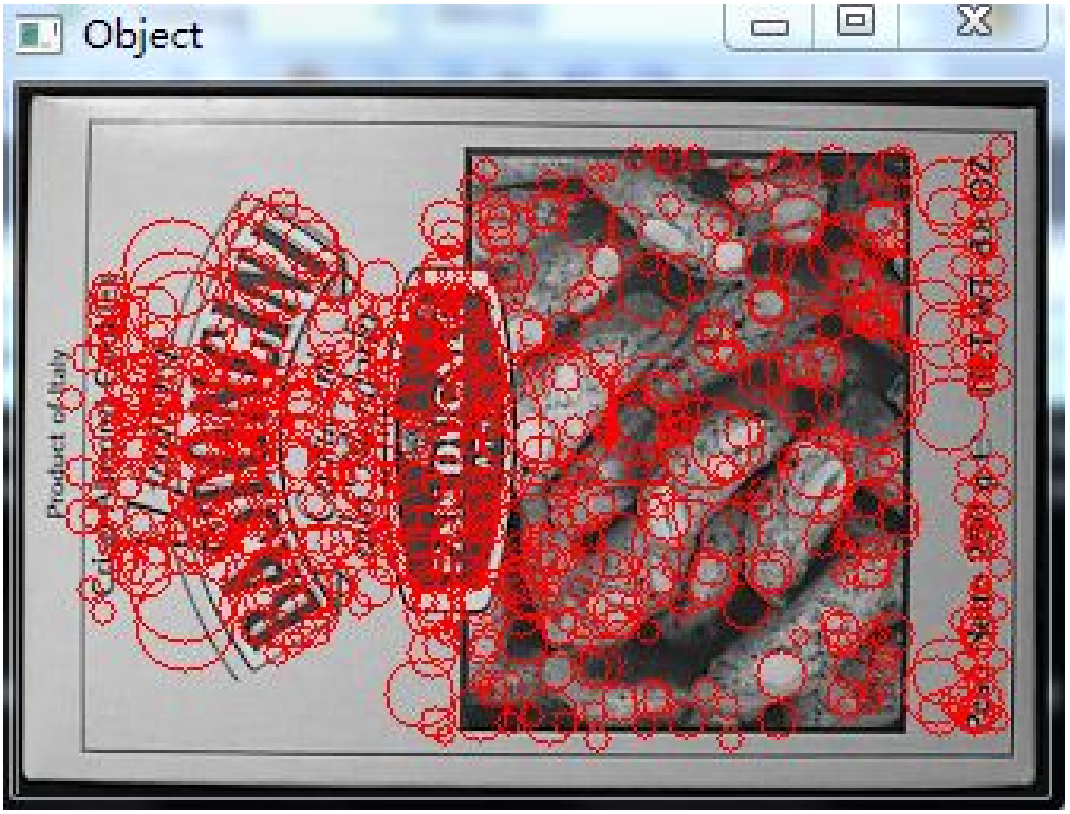
```
179.
constchar* scene_filename = argc == 3 ? argv[2] : "D:/Demo.jpg";
    help();    IplImage* object = cvLoadImage( object_filename,
CV_LOAD_IMAGE_GRAYSCALE );
180.
    IplImage* image = cvLoadImage( scene_filename, CV_LOAD_IMAGE
_GRAYSCALE );
181. if( !object || !image )
182.     {
183.         fprintf( stderr, "Can not load %s and/or %s\n",
184.             object_filename, scene_filename );
185.         exit(-1);
186.     }
187. //内存存储器
188.
    CvMemStorage* storage = cvCreateMemStorage(0);    cvNamedWin
dow("Object", 1);
189.
    cvNamedWindow("Object Correspond", 1);    static CvScalar co
lors[] =
190.     {
191.         {{0,0,255}},
192.         {{0,128,255}},
193.         {{0,255,255}},
194.         {{0,255,0}},
195.         {{255,128,0}},
196.         {{255,255,0}},
197.         {{255,0,0}},
198.         {{255,0,255}},
199.         {{255,255,255}}
200.     };
201.
    IplImage* object_color = cvCreateImage(cvGetSize(object), 8,
 3);
202.
    cvCvtColor( object, object_color, CV_GRAY2BGR );    CvSeq* o
bjectKeypoints = 0, *objectDescriptors = 0;
203.     CvSeq* imageKeypoints = 0, *imageDescriptors = 0;
```

```
204. int i;
205. /*
206.  CvSURFParams params = cvSURFParams(500, 1);//SURF参数设置:
阈值500，生成128维描述符
207.  cvSURFParams 函数原型如下:
208.  CvSURFParams cvSURFParams(double threshold, int extended)
209.  {
210.   CvSURFParams params;
211.   params.hessianThreshold = threshold; // 特征点选取
的 hessian 阈值
212.   params.extended = extended; // 是否扩展，1 - 生成128维描述
符，0 - 64维描述符
213.   params.nOctaves = 4;
214.   params.nOctaveLayers = 2;
215.   return params;
216.  }
217.  */
218.
 CvSURFParams params = cvSURFParams(500, 1);    double tt = (dou
ble)cvGetTickCount();//计时
219. /*
220.  提取图像中的特征点，函数原型:
221.
 CVAPI(void) cvExtractSURF( const CvArr* img, const CvArr* mask,
222.  CvSeq** keypoints, CvSeq** descriptors,
223.
 CvMemStorage* storage, CvSURFParams params, int useProvidedKeyP
ts CV_DEFAULT(0) );
224.  第3、4个参数返回结果：特征点和特征点描述符，数据类型是指针的指针，
225.  */
226.
    cvExtractSURF( object, 0, &objectKeypoints, &objectDescripto
rs, storage, params );
227.     printf("Object Descriptors: %d\n", objectDescriptors-
>total);    cvExtractSURF( image, 0, &imageKeypoints, &imageDesc
riptors, storage, params );
228.     printf("Image Descriptors: %d\n", imageDescriptors-
>total);
```

```
229.
    tt = (double)cvGetTickCount() - tt;    printf( "Extraction t
ime = %gms\n", tt/(cvGetTickFrequency()*1000.));
230.     CvPoint src_corners[4] = {{0,0}, {object-
>width,0}, {object->width, object->height}, {0, object-
>height}};
231. //定义感兴趣的区域
232.  CvPoint dst_corners[4];
233.     IplImage* correspond = cvCreateImage( cvSize(image-
>width, object->height+image->height), 8, 1 );
234. //设置感兴趣区域
235. //形成一大一小两幅图显示在同一窗口
236.     cvSetImageROI( correspond, cvRect( 0, 0, object-
>width, object->height ) );
237.     cvCopy( object, correspond );
238.     cvSetImageROI( correspond, cvRect( 0, object-
>height, correspond->width, correspond->height ) );
239.     cvCopy( image, correspond );
240.     cvResetImageROI( correspond );#ifdef USE_FLANN
241.
    printf("Using approximate nearest neighbor search\n");
242. #endif
243. //寻找物体(object)在场景(image)中的位置，并将信息保存（矩形框）
244.
if( locatePlanarObject( objectKeypoints, objectDescriptors, imag
eKeypoints,
245.         imageDescriptors, src_corners, dst_corners ))
246.     {
247. for( i = 0; i < 4; i++ )
248.         {
249.             CvPoint r1 = dst_corners[i%4];
250.             CvPoint r2 = dst_corners[(i+1)%4];
251.    cvLine( correspond, cvPoint(r1.x, r1.y+object-
>height ),
252.     cvPoint(r2.x, r2.y+object->height ), colors[8] );
253.         }
254.     }
255. //定义并保存物体(object)在场景(image)图形之间的匹配点对，并将其存储
在向量 ptpairs 中，之后可以对
```

```
256. //ptpairs 进行操作
257.     vector<int> ptpairs;
258. #ifdef USE_FLANN
259.
    flannFindPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
260. #else
261.
    findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
262. #endif
263. //显示匹配结果（直线连接）
264. for( i = 0; i < (int)ptpairs.size(); i += 2 )
265.     {
266.
        CvSURFPoint* r1 = (CvSURFPoint*)cvGetSeqElem( objectKeypoints, ptpairs[i] );
267.
        CvSURFPoint* r2 = (CvSURFPoint*)cvGetSeqElem( imageKeypoints, ptpairs[i+1] );
268.         cvLine( correspond, cvPointFrom32f(r1->pt),
269.             cvPoint(cvRound(r2->pt.x), cvRound(r2->pt.y+object->height)), colors[8] );
270.     }    cvShowImage( "Object Correspond", correspond );
271. //显示物体(object)的所有特征点
272. for( i = 0; i < objectKeypoints->total; i++ )
273.     {
274.
        CvSURFPoint* r = (CvSURFPoint*)cvGetSeqElem( objectKeypoints, i );
275.         CvPoint center;
276. int radius;
277.         center.x = cvRound(r->pt.x);
278.         center.y = cvRound(r->pt.y);
279.         radius = cvRound(r->size*1.2/9.*2);
280.
        cvCircle( object_color, center, radius, colors[0], 1, 8, 0 );
281.     }
```

```
282.
    cvShowImage( "Object", object_color );    cvWaitKey(0); //释
放窗口所占用的内存
283.     cvDestroyWindow("Object");
284.     cvDestroyWindow("Object Correspond");    return 0;
285. }
```



SIFT参考地址：

http://blog.csdn.net/abcjennifer/article/details/7639681

SURF参考地址：

http://wenku.baidu.com/view/cf0c164f2e3f5727a5e96238.html

http://blog.csdn.net/yangtrees/article/details/7482960

http://blog.csdn.net/leaglave_jyan/article/details/6676549

http://www.cnblogs.com/blue-lg/archive/2012/07/17/2385755.html