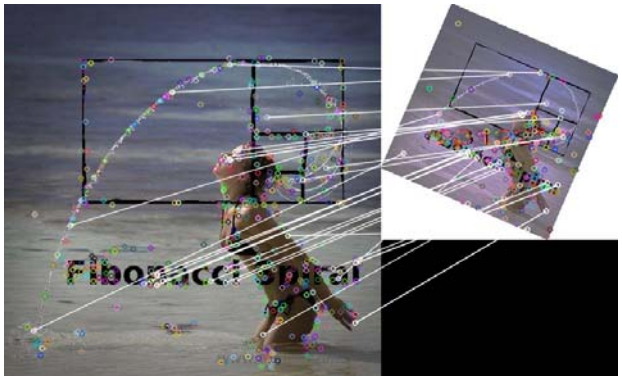


图像处理之SIFT算法与OpenCV实现

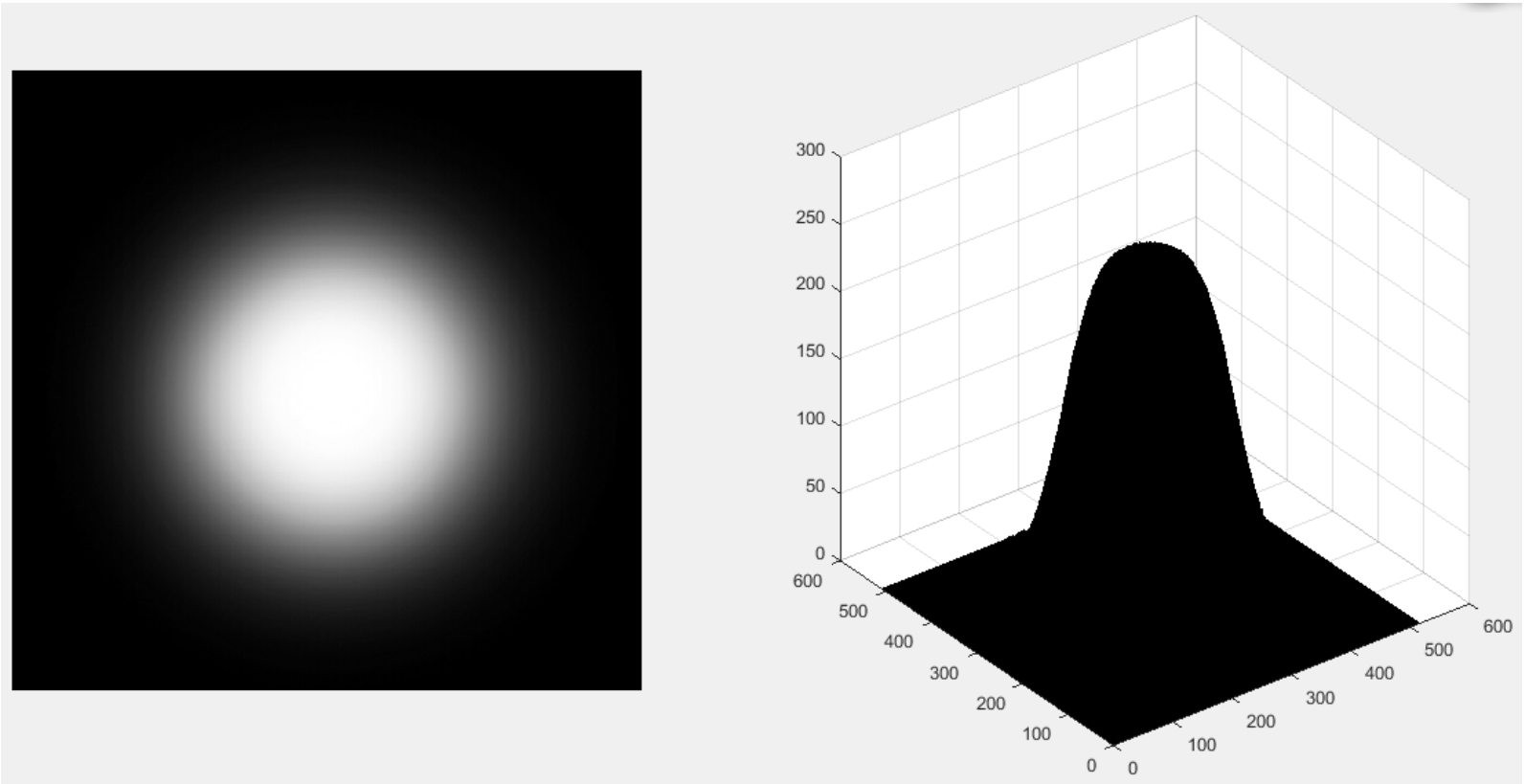
Google Developers

开放 分享 创新

developers.google.com

图像的本质

- 光晕，黑白色508X516X1矩阵，彩色508X516X3矩阵



拉普拉斯算子

场论回顾

- 哈密尔顿算子▽，定义为 $\nabla = \frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j} + \frac{\partial}{\partial z}\vec{k}$
- 或者写成向量符号。 $\nabla = \left\{\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right\}$
- 梯度grad u= $\nabla u = (\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j} + \frac{\partial}{\partial z}\vec{k})u = \frac{\partial u}{\partial x}\vec{i} + \frac{\partial u}{\partial y}\vec{j} + \frac{\partial u}{\partial z}\vec{k}$
- 散度= $\operatorname{div}\mathbf{F} = \nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}.$
- 旋度 $\operatorname{curl}\mathbf{F} = \nabla \times \mathbf{F} = \left(\frac{\partial F_3}{\partial y} - \frac{\partial F_2}{\partial z}\right)\mathbf{e}_1 - \left(\frac{\partial F_3}{\partial x} - \frac{\partial F_1}{\partial z}\right)\mathbf{e}_2 + \left(\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y}\right)\mathbf{e}_3.$
- 拉普拉斯算子△，定义为△=▽●▽ $= \nabla \bullet \nabla = \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$

直观的看，拉普拉斯算子是一个标量场的梯度的散度。

离散化处理

- 对于1维离散情况，其二阶导数变为二阶差分。
- 1）首先，其一阶差分为 $\nabla f[n] = f'[n] = D_n[f[n]] = f[n+1] - f[n]$
- 2）因此，二阶差分

$$\begin{aligned} \triangle f[n] &: \quad \nabla^2 f[n] = f''[n] = D_n^2[f[n]] = f'[n] - f'[n-1] \\ &: \quad (f[n+1] - f[n]) - (f[n] - f[n-1]) = f[n+1] - 2f[n] + f[n-1] \end{aligned}$$

- 因此，1维拉普拉斯运算可以通过1维卷积核[1 -2 1]实现。
- 对于2维离散情况（图像），拉普拉斯算子是2个维上二阶差分的和，其公式为：

$$\begin{aligned} \triangle f[m,n] &: \quad D_m^2[f[m,n]] + D_n^2[f[m,n]] = f[m+1,n] - 2f[m,n] + f[m-1,n] + f[m,n+1] - 2f[m,n] + f[m,n-1] \\ &: \quad f[m+1,n] + f[m-1,n] + f[m,n+1] + f[m,n-1] - 4f[m,n] \end{aligned}$$

- 常用的二维拉普拉斯卷积核有：

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

拉普拉斯算子性质

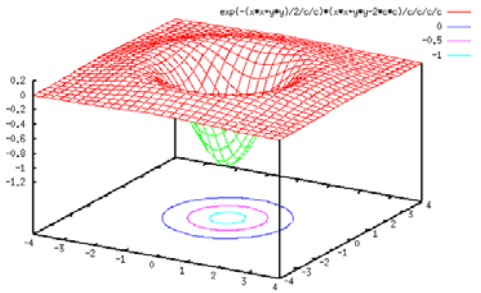
- 拉普拉斯算子具有各同向性，图片可以任意旋转

0	−1	0
−1	4	−1
0	−1	0

−1	−1	−1
−1	8	−1
−1	−1	−1

- 拉普拉斯运算和卷积运算的可交换性，降低运算量

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 1 & 2 & -16 & 2 & 1 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

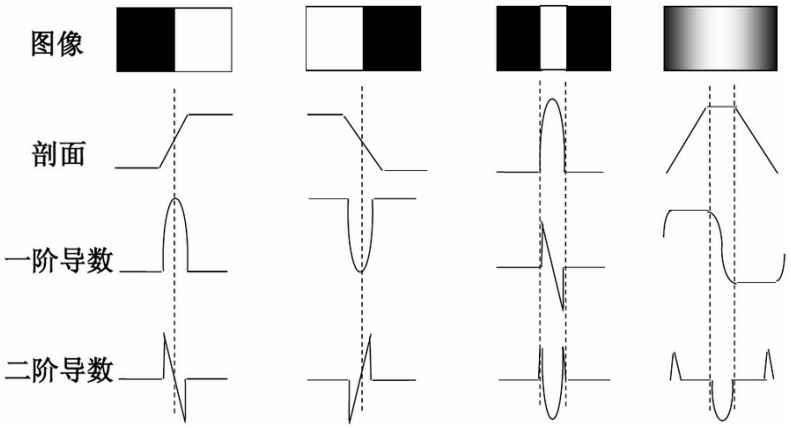


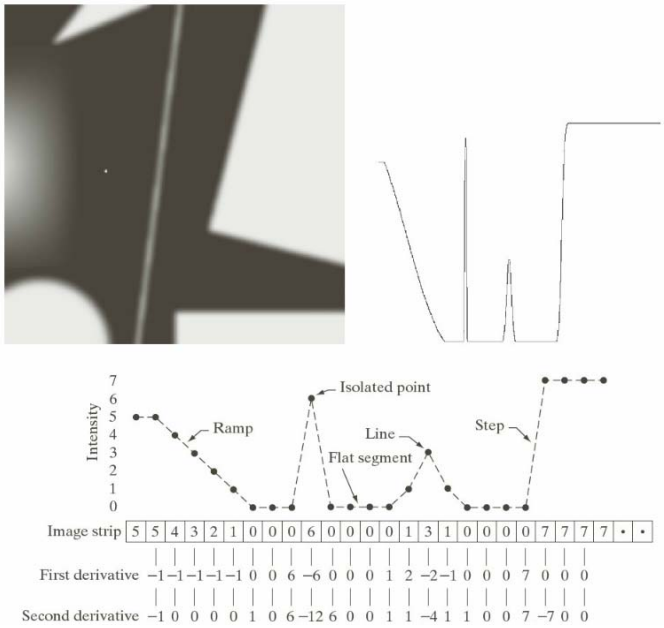
$$\triangle[G_\sigma(x,y) * f(x,y)] = [\triangle G_\sigma(x,y)] * f(x,y) = LoG * f(x,y)$$

$$\frac{d}{dt}[h(t)*f(t)] = \frac{d}{dt}\int f(\tau)h(t-\tau)d\tau = \int f(\tau)\frac{d}{dt}h(t-\tau)d\tau = f(t)*\frac{d}{dt}h(t)$$

拉普拉斯算子用于边缘检测

- 边缘有四类：阶跃边缘、脉冲边缘（左右正负阶跃）、阶梯边缘、屋顶边缘（左右正负阶梯）





- 查找二阶导数的过零点，来确定边缘点。然后当然通过hough变换来确定边缘直线。参考 http://wenku.baidu.com/view/be0ba7d628ea81c758f57897.html

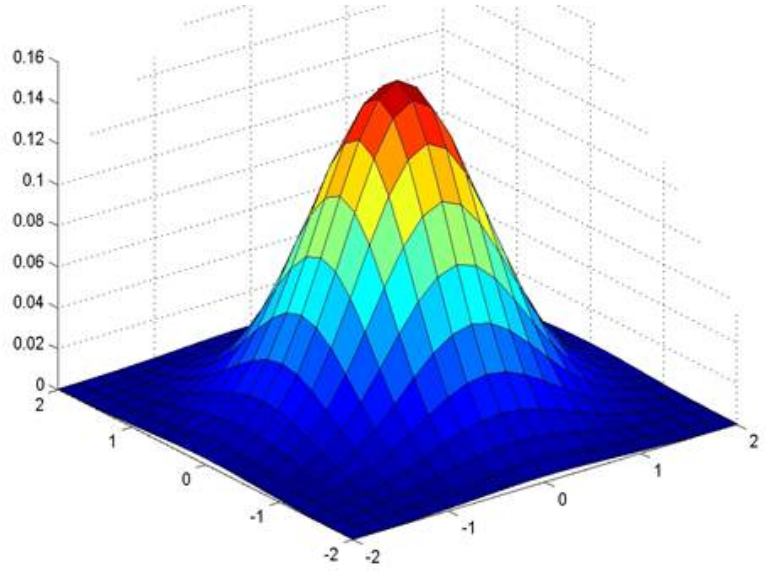
拉普拉斯算子用于特征点检测

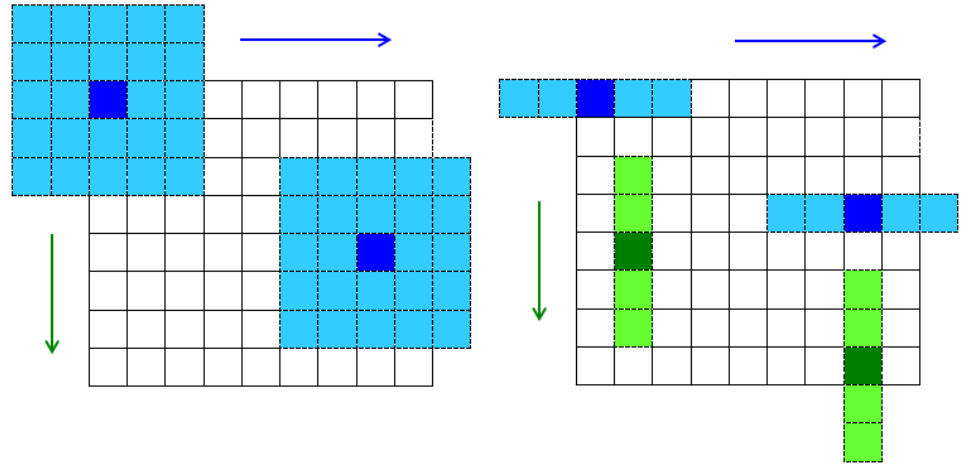
- 拉普拉斯算子 $\triangle \triangleq \nabla^2 \triangleq \nabla \cdot \nabla = (\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j}) \cdot (\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j}) = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$
- 把原图像看做一个标量场u(x,y)，把拉普拉斯算子是一个标量场的梯度的散度。那么对原图像运用拉普拉斯算子，也就是求取一个标量场输出的散度标量场。
- 直观的看，拉普拉斯算子求取的值正值，应该是一个正源点，负值应该是一个负源点。
- 那么很显然，一片白色的墙壁上的一个黑点，一定是一个负的拉普拉斯值，一片黑色墙壁上的白色的点，一定是一个正的拉普拉斯值。
- 坑：拉普拉斯函数用于寻找边缘和寻找孤立点的算法基本一样，但判断极值和判断过零点的标准是不一样，很多文章混淆写错了。这将直接影响对代码的理解。

高斯卷积核

高斯卷积核函数

- 一维高斯卷积核 $$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$
- 二维高斯卷积核 $$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x - x_0)^2 + (y - y_0)^2}{2\sigma^2}\right)$$





- 在OpenCV也只需要一句代码：
- GaussianBlur(dbl, dbl, Size(), sig_diff, sig_diff);

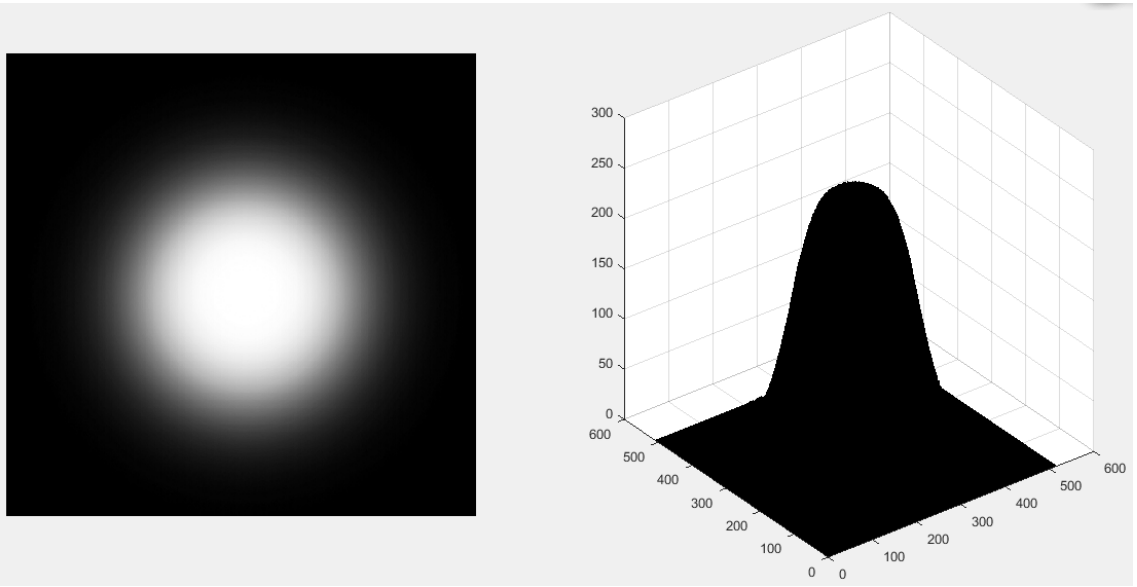
高斯卷积核的性质

- 高斯模版是圆对称的，且卷积的结果使原始像素值有最大的权重，距离中心越远的相邻像素值权重也越小。在实际应用中，在计算高斯函数的离散近似时，在大概3σ距离之外的像素都可以看作不起作用，这些像素的计算也就可以忽略。所以，通常程序只计算(6σ+1)*(6σ+1)就可以保证相关像素影响。
- 高斯模糊另一个很厉害的性质就是线性可分：使用二维矩阵变换的高斯模糊可以通过在水平和竖直方向各进行一维高斯矩阵变换相加得到。O(N^2*m*n)次乘法就缩减成了O(N*m*n)+O(N*m*n)次乘法。

特征点=拉普拉斯高斯LoG的极值点

特征点的直观定义

- 一幅图像，在不同的模糊尺度下（与不同的高斯卷积核卷积），在不同的尺度产生不同的子图像
- 对于每个子图像，梯度的散度（拉普拉斯算子），取得局部极值，这个极值也是上下模糊尺度的极值，那么这个位置就是特征点。
- 以下图为例，梯度，是一圈对外的辐射线；梯度的散度，理论上是一个单峰。这个单峰的位置就是特征点。



拉普拉斯高斯函数

- LoG，即拉普拉斯高斯的简称，laplace of Gaussian
  - 先做高斯平滑，将图像与高斯函数卷积
  - 然后将拉普拉斯算子作用在图像上
  - 输出拉普拉斯高斯图像。
- 第一，不同尺度的高斯函数，对应不同尺度的LoG图像。
- 第二，根据以下性质，将拉普拉斯算子与卷及运算交换顺序，用内存换速度

$$\triangle[G_\sigma(x,y) * f(x,y)] = [\triangle G_\sigma(x,y)] * f(x,y) = LoG * f(x,y)$$

- 缺点：卷积核要储备很多尺度，且卷积运算的运算量还是很大。

拉普拉斯高斯LoG
差分拉普拉斯DoG

推导过程

- 对后面delta的指数线性颗粒度而言，
- G对ln （delta ）的导数，刚刚好就是LoG！

$$DoG = \frac{\partial}{\partial\sigma}G(x,y,\sigma) = \frac{\partial}{\partial\sigma}\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})$$

$$= \frac{1}{\sqrt{2\pi}}\left(-2\sigma^{-3}\right)\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}) +$$

$$\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})\left[-\frac{(x-x_0)^2+(y-y_0)^2}{2}\right]\left(-2\sigma^{-3}\right)$$

$$= \frac{1}{\sqrt{2\pi}}\left(-2\sigma^{-3}\right)\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}) +$$

$$\frac{1}{\sqrt{2\pi}}\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})\frac{(x-x_0)^2+(y-y_0)^2}{\sigma^5}$$

$$= \frac{1}{\sqrt{2\pi}}\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})\left[\frac{(x-x_0)^2+(y-y_0)^2-2\sigma^2}{\sigma^5}\right]$$

$$LoG = \Delta G(x,y,\sigma) = \nabla^2 G(x,y,\sigma) =$$

$$\frac{1}{\sqrt{2\pi\sigma^2}}\left[\frac{(x-x_0)^2+(y-y_0)^2-2\sigma^2}{\sigma^4}\right]$$

$$\exp(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2})$$

$$\frac{\partial G}{\partial\sigma}\cdot\sigma = \nabla^2 G(x,y,\sigma)$$

$$\frac{\partial G}{\partial\ln\sigma}\cdot\frac{\partial\ln\sigma}{\partial\sigma}\cdot\sigma = \nabla^2 G(x,y,\sigma)$$

$$\frac{\partial G}{\partial\ln\sigma} = \nabla^2 G(x,y,\sigma)$$

DoG的好处

- 将常规的两次求导和一次卷积运算，转化为一次卷积和一次差分运算，大幅降低运算量。
- 结合下采样，高斯卷积核可以复用；结合下采样，运算量进一步下降到四分之一。
- 这个公式很早就已经发现，不属于SIFT的专利
- 坑：高斯卷积核的公式，这个尺度值是否进根号，是否有平方，很多错误。这将直接影响DoG代替LoG的合理性，这是SIFT算法的最核心思想，没考虑清楚这个，将对后面无数公式代码理解产生无穷隐患。
- 坑：尺度空间为什么在指数上等间隔，而不是尺度本身等间隔，没有一篇文章介绍，但实际上只有推导过，才能体会后期调整的时候0.5的阈值的意义，难道是Lowe的心机，难道是故意不说。

用DoG和LoG的关系快速求取LoG

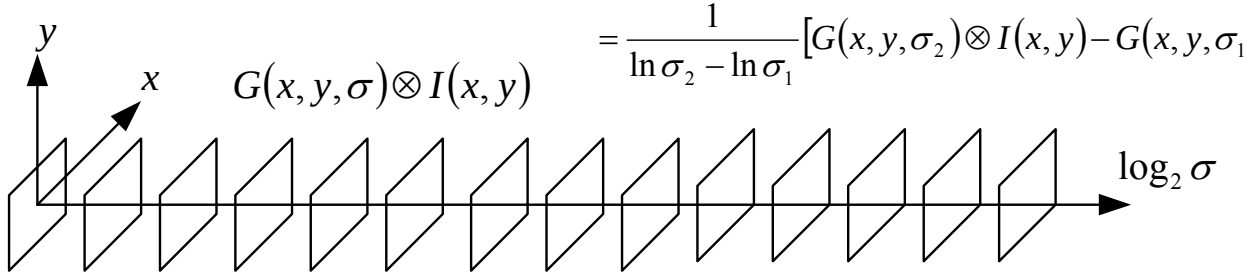
传统的LoG计算

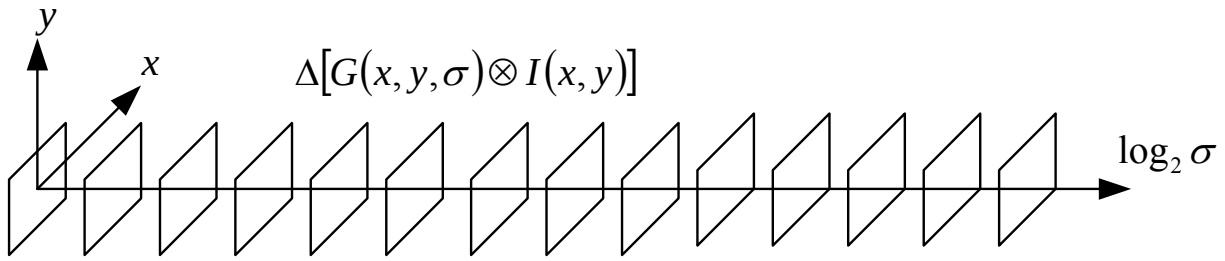
- 传统方法，对图像做不同尺度的高斯函数模糊，然后求取每个不同尺度下的拉普拉斯函数值，查找极值，作为特征点。如下图所示。但是计算量非常大。

$$\Delta[G(x,y,\sigma)\otimes I(x,y)]=[\Delta G(x,y,\sigma)]\otimes I(x,y)=\frac{\partial G(x,y,\sigma)}{\partial \ln\sigma}\otimes I(x,y)$$

$$\approx\left[\frac{G(x,y,\sigma_2)-G(x,y,\sigma_1)}{\ln\sigma_2-\ln\sigma_1}\right]\otimes I(x,y)$$

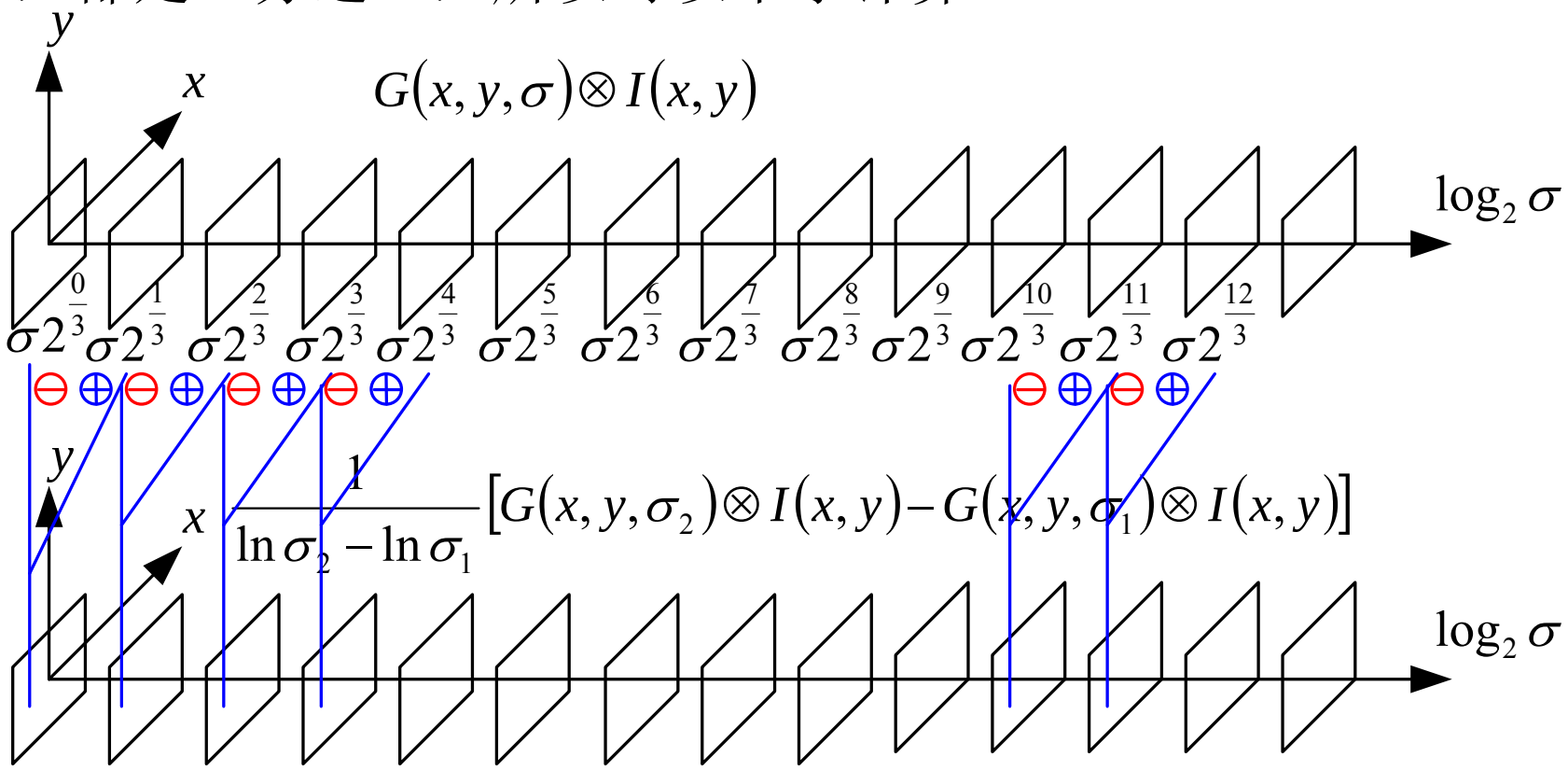
$$=\frac{1}{\ln\sigma_2-\ln\sigma_1}[G(x,y,\sigma_2)\otimes I(x,y)-G(x,y,\sigma_1)\otimes I(x,y)]$$





DoG代替LoG计算

- 以假设delta=1.6，模糊尺度的步进2^1/3为例。通过相临的相减，来代替某个delta点上的导数值。如下图所示。而且由于delta是指数关系，所以他们的共同因子相等，都是三分之一，所以可以不予计算。



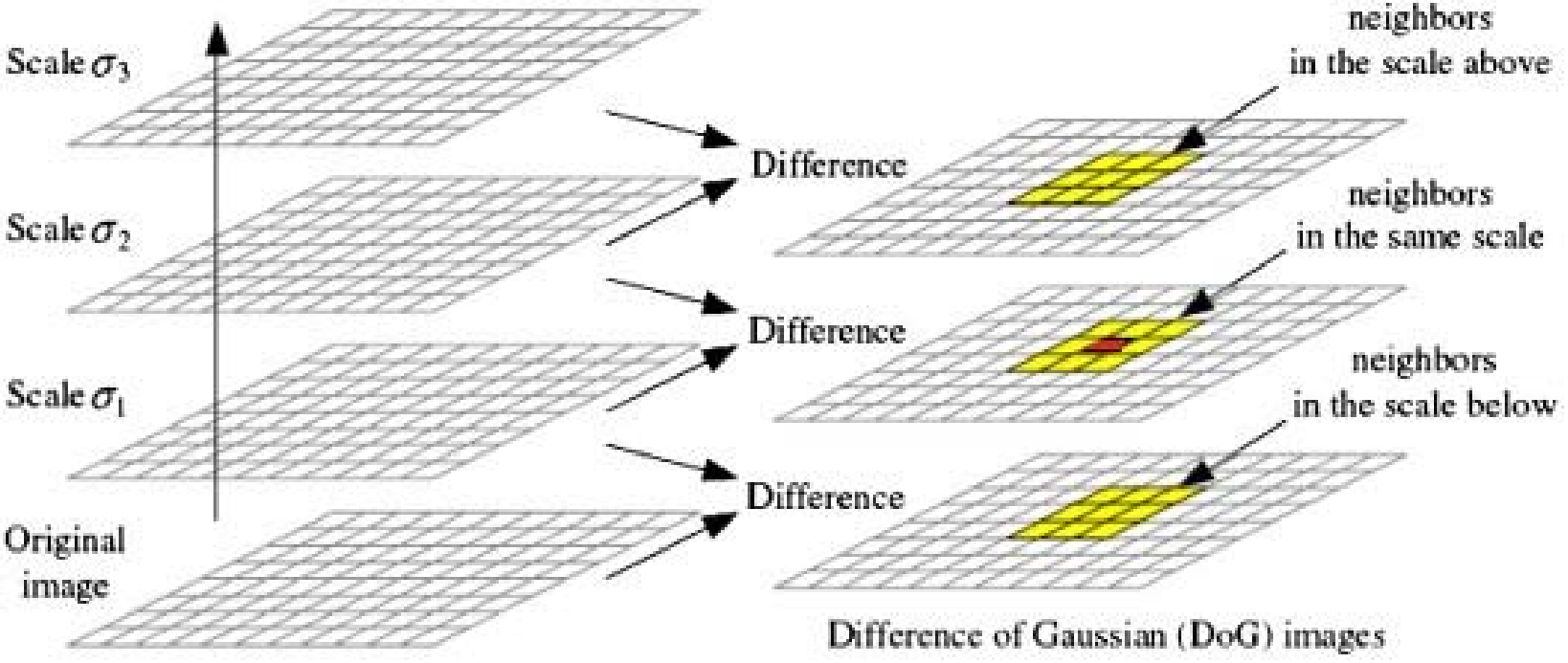
工程实现

通过下采样降低高斯卷积的运算量

- 高斯卷积可以看做一个低通滤波器。delta越大，纳入掩膜加权的像素越多，高频细节信息损耗越大，保留的都是低频信息，图像越模糊。
- 所以，对图像加以2*delta的高斯卷积，等效于对图像先做1/2的下采样，然后做delta的高斯卷积。计算量减小到四分之一。
- 降低运算量的方法就是，确定初始的delta，开始卷积，当模糊系数delta以2^1/3位系数指数变大，变大到2^3/3=2倍的时候，换行进行下采样，然后在小一点的图片上进行继续的卷积

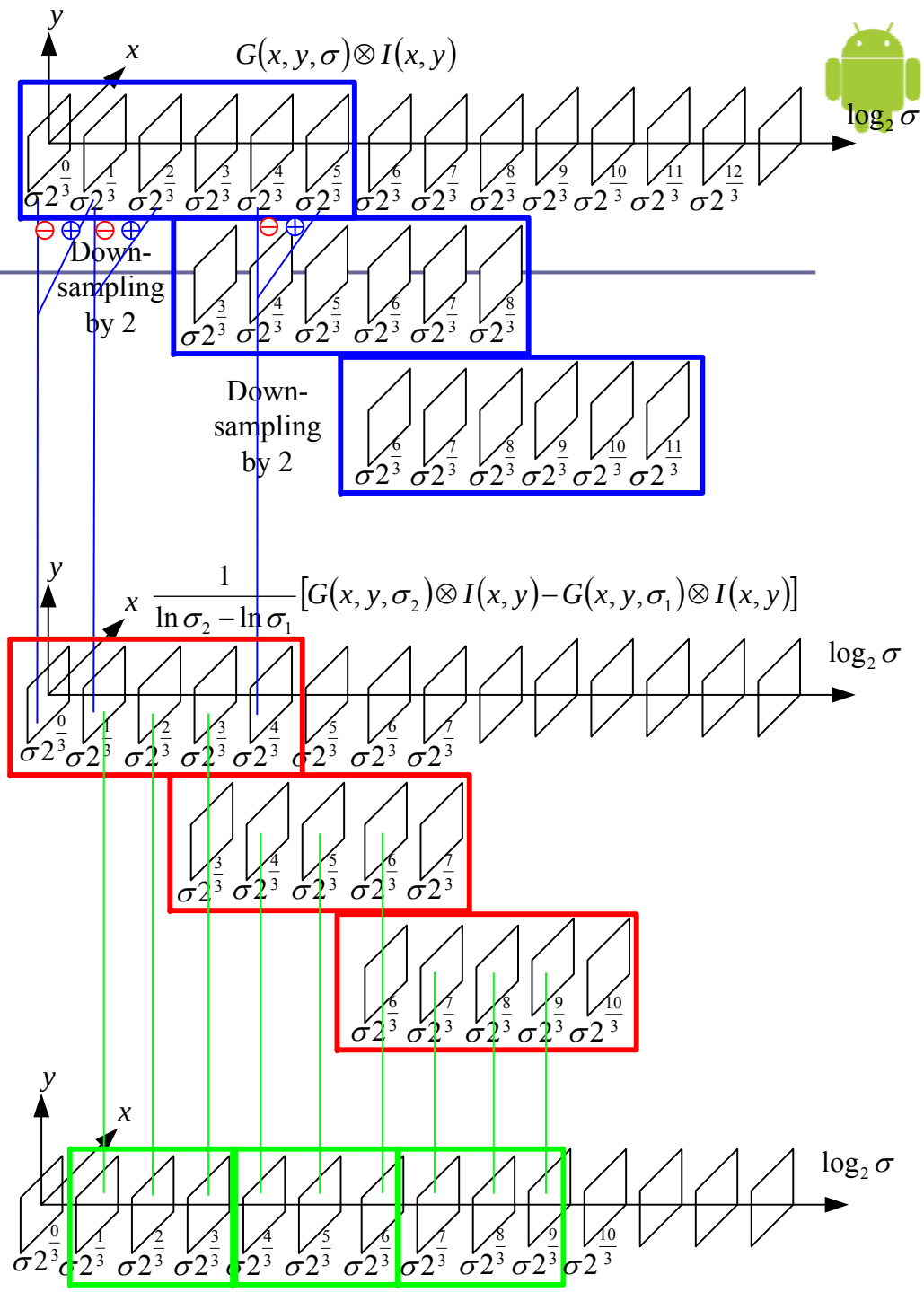
下采样时重叠以求极值

- SIFT认为，特征点是局部极值点，这个极值点考虑上下左右一圈8个点，同时考虑上下两幅临近图片的18点，合计26个点（即立方体27个点的极值是否在正中央）。
- 由于最后的极值选取，需要考虑上下两幅临近图片，因此，需要重叠的图片是三张。



一张图

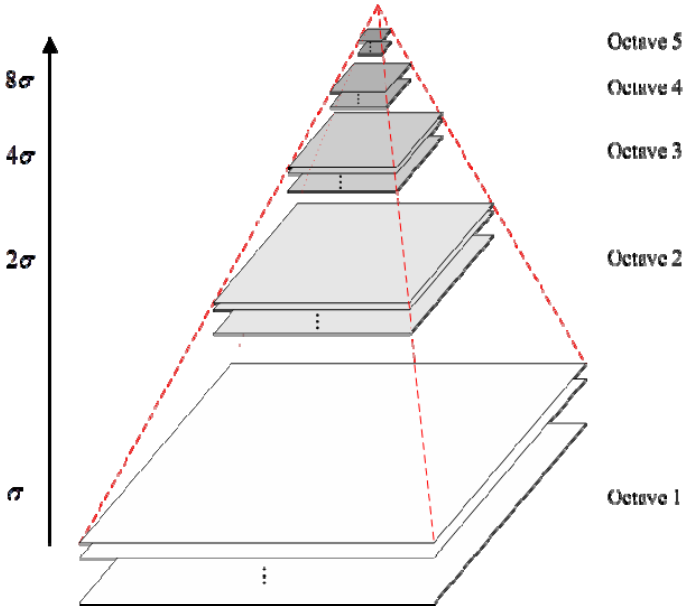
- 尺度空间得以连续
- 设S=3S=3，也就是每组有3层，则k=2^1/S=2^1/3，也就是有高斯金字塔每组有(S−1)3层图像，DoG金字塔每组有(S−1)3层图像，DoG金字塔每组有(S-2)2层图像。在DoG金字塔的第一组有两层尺度分别是σ,kσ，第二组有两层的尺度分别是2σ,2kσ，由于只有两项是无法比较取得极值的（只有左右两边都有值才能有极值）。由于无法比较取得极值，那么我们就需要继续对每组的图像进行高斯模糊，使得尺度形成σ,kσ,k^2σ,k^3σ,k^4σ，这样就可以选择中间的三项kσ,k^2σ,k^3σ。对应的下一组由上一组降采样得到的三项是2kσ,2k^2σ,2k^3σ，其首项2kσ=2·2^1/3σ=2^4/3σ，刚好与上一组的最后一项k^3σ=2^3/3σ的尺度连续起来。



DoG金字塔代码

高斯金字塔

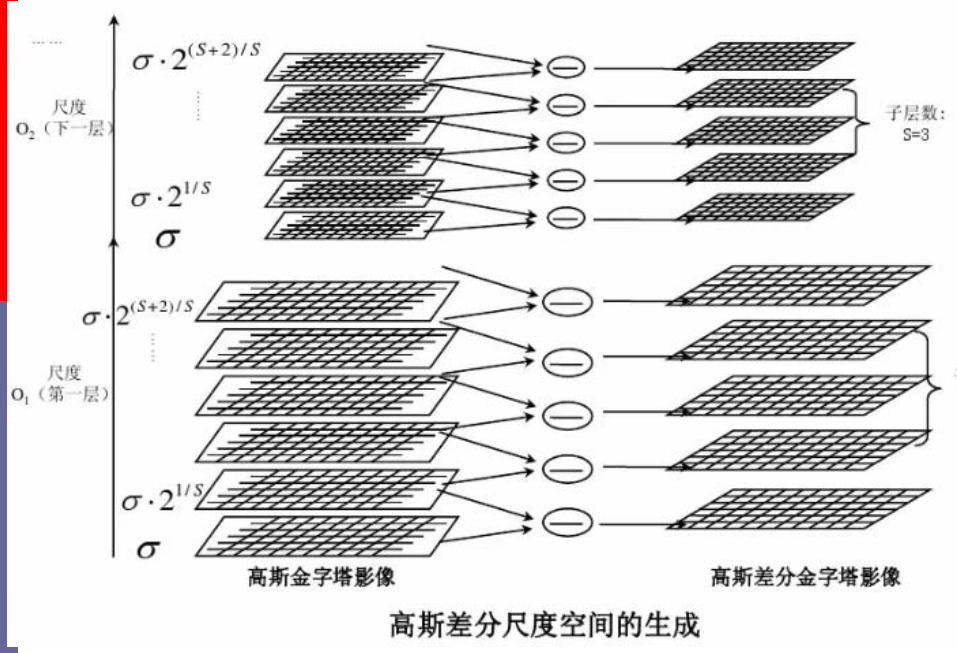
- 为了得到DoG图像，先要构造高斯金字塔。
- 高斯金字塔在多分辨率金字塔简单降采样基础上加了高斯滤波，也就是对金字塔每层图像用不同参数的σ做高斯模糊，使得每层金字塔有多张高斯模糊图像。金字塔每层多张图像合称为一组（Octave），每组有多张（也叫层Interval）图像。另外，降采样时，金字塔上边一组图像的第一张图像（最底层的一张）是由前一组（金字塔下面一组）图像的倒数第三张隔点采样得到。




```
// 构建nOctaves组（每组nOctaves+3层）高斯金字塔
void SIFT::buildGaussianPyramid( const Mat& base, vector<Mat>& pyr, int nOctaves ) const
{
    vector<double> sig(nOctaveLayers + 3);
    pyr.resize(nOctaves*(nOctaveLayers + 3));
    // precompute Gaussian sigmas using the following formula:
    //  \sigma_{total}^2 = \sigma_{i}^2 + \sigma_{i-1}^2、
        // 计算对图像做不同尺度高斯模糊的尺度因子
    sig[0] = sigma;
    double k = pow( 2., 1. / nOctaveLayers );
    for( int i = 1; i < nOctaveLayers + 3; i++ )
    {
        double sig_prev = pow(k, (double)(i-1))*sigma;
        double sig_total = sig_prev*k;
        sig[i] = std::sqrt(sig_total*sig_total - sig_prev*sig_prev);
    }
    for( int o = 0; o < nOctaves; o++ )
    {
                    // DoG金子塔需要nOctaveLayers+2层图像来检测nOctaves层尺度
                    // 所以高斯金字塔需要nOctaveLayers+3层图像得到nOctaveLayers+2层DoG金字
        塔
        for( int i = 0; i < nOctaveLayers + 3; i++ )
        {
                                    // dst为第o组（Octave）金字塔
            Mat& dst = pyr[o*(nOctaveLayers + 3) + i];
                                    // 第0组第0层为原始图像
            if( o == 0  &&  i == 0 )
                dst = base;
                                    // base of new octave is halved image from end of previous octave
                                    // 每一组第0副图像时上一组倒数第三幅图像隔点采样得到
            else if( i == 0 )
            {
                const Mat& src = pyr[(o-1)*(nOctaveLayers + 3) + nOctaveLayers];
                resize(src, dst, Size(src.cols/2, src.rows/2),
                       0, 0, INTER_NEAREST);
            }
                                    // 每一组第i副图像是由第i-1副图像进行sig[i]的高斯模糊得到
                                    // 也就是本组图像在sig[i]的尺度空间下的图像
            else
            {
                const Mat& src = pyr[o*(nOctaveLayers + 3) + i-1];
                GaussianBlur(src, dst, Size(), sig[i], sig[i]);
            }
        }
    }
```

高斯差分金字塔

- 构建高斯金字塔之后，就是用金字塔相邻图像相减构造DoG金字塔

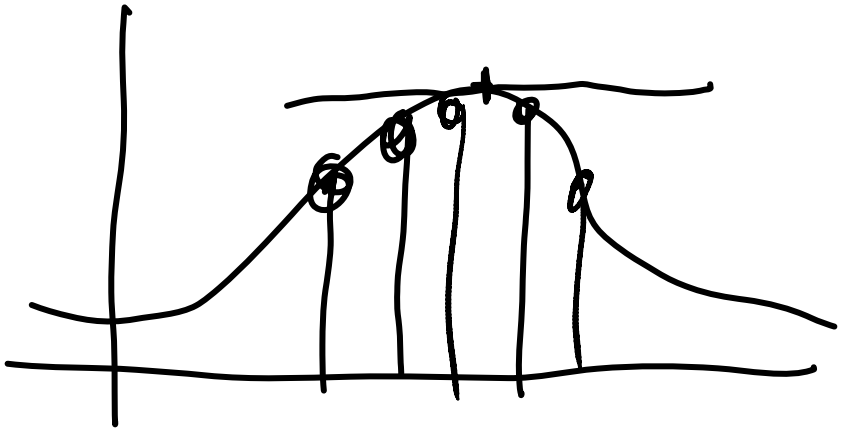


高斯差分尺度空间的生成

```
// 构建nOctaves组（每组nOctaves+2层）高斯差分金字塔
void SIFT::buildDoGPyramid( const vector<Mat>& gpyr, vector<Mat>& dogpyr ) const
{
    int nOctaves = (int)gpyr.size()/(nOctaveLayers + 3);
    dogpyr.resize( nOctaves*(nOctaveLayers + 2) );
    for( int o = 0; o < nOctaves; o++ )
    {
        for( int i = 0; i < nOctaveLayers + 2; i++ )
        {
                                    // 第o组第i副图像为高斯金字塔中第o组第i+1和i组图像相减得到
            const Mat& src1 = gpyr[o*(nOctaveLayers + 3) + i];
            const Mat& src2 = gpyr[o*(nOctaveLayers + 3) + i + 1];
            Mat& dst = dogpyr[o*(nOctaveLayers + 2) + i];
            subtract(src2, src1, dst, noArray(), CV_16S);
        }
    }
}
```

特征点的精确定位

离散的坏处



- 离散以后，要回答两个问题就比较困难了。
- 第一，极值点在哪里？
- 第二，极值点等于多少

向量的泰勒展开

- 泰勒公式的精髓在于，只要知道一个函数在一个点的0-无穷阶导数，就可以知道这个函数在任意点的值和任意阶导数。

$$f(x)=f(x_0)+f^1(x_0)(x-x_0)+f^2(x_0)\frac{(x-x_0)^2}{2!}+\cdots$$

$$f^1(x)=0+f^1(x_0)+f^2(x_0)\frac{2(x-x_0)}{2!}+f^3(x_0)\frac{3(x-x_0)^2}{3!}\cdots$$

$$f^1(x)=0+f^1(x_0)+f^2(x_0)(x-x_0)$$

$$(x-x_0)=-\left[f^2(x_0)\right]^{-1}\cdot f^1(x_0)\quad f(x)=\cdots=f(x_0)+\frac{1}{2}f^1(x_0)(x-x_0)$$

$$X=(x,y,\sigma)^T \qquad D(X)=D+\frac{\partial D^T}{\partial X}X+\frac{1}{2}X^T\frac{\partial^2 D}{\partial X^2}X$$

$$\hat{X}=-\left(\frac{\partial^2 D}{\partial X^2}\right)^{-1}\frac{\partial D}{\partial X} \qquad D(\hat{X})=D+\frac{1}{2}\frac{\partial D^T}{\partial X}\hat{X}$$

预估离散图像矩阵的极值和位置

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

这里特别说明的是，xy 的维度，是某像素某像素，当然是+1+1 步进的，偏移量小于 0.5 大于 0.5 可以作为是否偏移到临近点的依据。

但 delta 的维度，之前证明的，函数值对 ln_delta 进行求导的情况下，DoG 和 LoG 才相等。但别忘记了这是数值计算，数值计算和公式计算相比的优势就是，只要 delta 的决定参数是+1+1 步进变化的，那么不管公式再复杂，函数值对这个+1+1 步进的参数的导数，就是，函数值的差分除以步进值。因此，维度上的偏移量小于 0.5 大于 0.5 可以作为是否偏移到临近点的依据。

某点的任意阶导数

$$\left(\frac{\partial f}{\partial x}\right)_0 = \frac{f_1 - f_3}{2h}$$

$$\left(\frac{\partial f}{\partial y}\right)_0 = \frac{f_2 - f_4}{2h}$$

$$\left(\frac{\partial^2 f}{\partial x^2}\right)_0 = \frac{f_1 + f_3 - 2f_0}{h^2}$$

$$\left(\frac{\partial^2 f}{\partial y^2}\right)_0 = \frac{f_2 + f_4 - 2f_0}{h^2}$$

$$\left(\frac{\partial^2 f}{\partial x \partial y}\right)_0 = \frac{(f_8 + f_6) - (f_5 + f_7)}{4h^2}$$

$$\left(\frac{\partial^4 f}{\partial x^4}\right)_0 = \frac{1}{h^4}[6f_0 - 4(f_1 + f_3) + (f_9 + f_{11})]$$

$$\left(\frac{\partial^4 f}{\partial y^4}\right)_0 = \frac{1}{h^4}[6f_0 - 4(f_2 + f_4) + (f_{10} + f_{12})]$$

				12				
			8	4	5			
		11	3	0	1	9		
			7	2	6			
				10				

图 4.2 图像中像素 0 与其领域

$$\left(\frac{\partial^4 f}{\partial x^2 \partial y^2}\right)_0 = \frac{1}{h^4}[4f_0 - 2(f_1 + f_2 + f_3 + f_4) + (f_5 + f_6 + f_7 + f_8)]$$

识别边缘特征点和孤立特征点

边缘点的梯度的散度也可能很高

- 拉普拉斯算子的过零点检测，是为了检测边缘。而特征点检测，是查看拉普拉斯算子的极值点，也就是散度正负向最大的点。而对比度高的边缘点，会同样引发散度局部最大和局部最小。这些特征点在图像的某个物体的边缘，他们连片出现，位置飘忽，需要去掉。

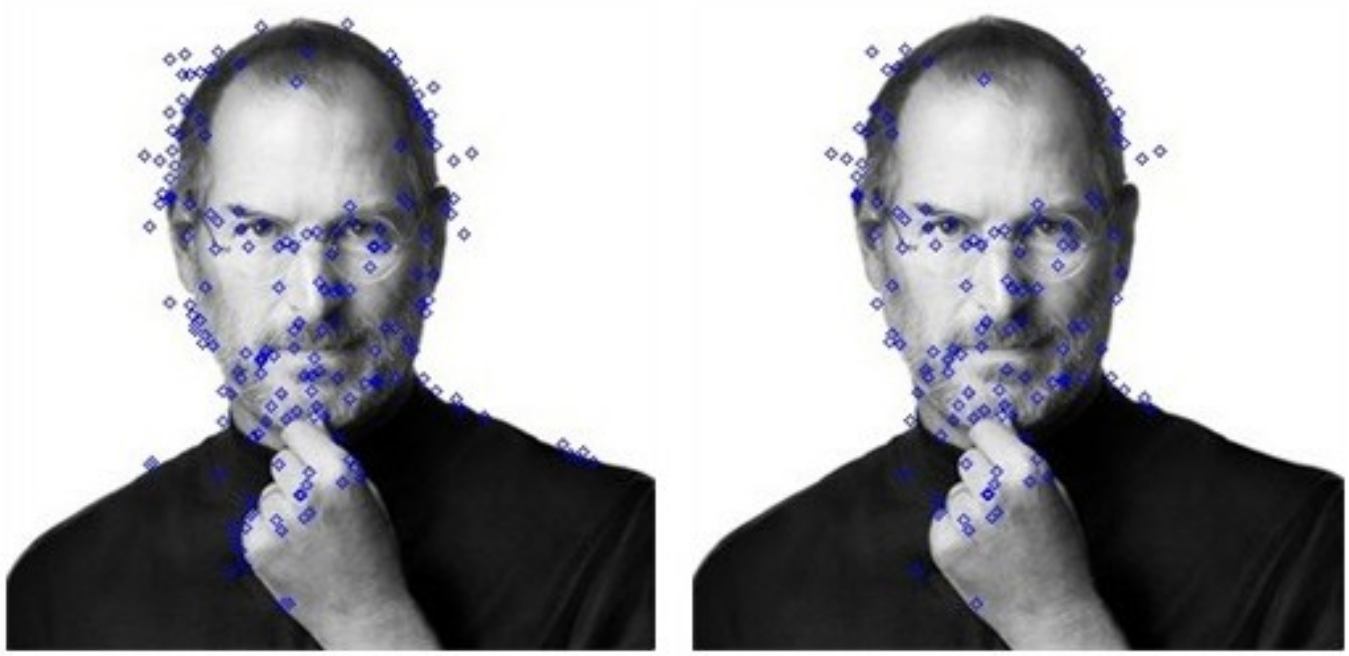


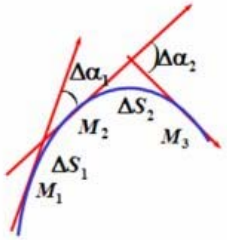
图 4.2 边缘消除响应前后的关键点

曲面的曲率

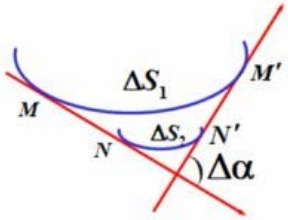
- 曲线的曲率是ds/d_alpha，就是走过s距离造成的切线转过alpha角度，这个比值的极限。

(1)定量分析

曲率是描述曲线局部性质（弯曲程度）的量.



弧段弯曲程度
越大转角越大



转角相同弧段越
短弯曲程度越大

- 曲面的曲率比较复杂，先看一下曲面的函数表达式。

$$z(x,\ y) = Ax^2 + By^2 + Cxy + Dx + Ey + F$$

式(5)中的系数可用一阶和二阶导数表示：

$$A = \frac{1}{2}\frac{d^2z}{dx^2}$$

$$B = \frac{1}{2}\frac{d^2z}{dy^2}$$

$$C = \frac{d^2z}{dxdy}$$

$$D = \frac{dz}{dx}$$

$$E = \frac{dz}{dy}$$

平均曲率：	$K_m = \frac{A(1+E^2)+B(1+D^2)-CDE}{(1+D^2+E^2)^{\frac{3}{2}}}$ 平均曲率：是空间上曲面上某一点任意两个相互垂直的正交曲率的平均值。如果一组相互垂直的正交曲率可表示为K1,K2,那么平均曲率则为：K = (K1 +K2) / 2。
主曲率：	过曲面上某个点上具有无穷个正交曲率，其中存在一条曲线使得该曲线的曲率为极大，这个曲率为极大值 Kmax，垂直于极大曲率面的曲率为极小值 Kmin。这两个曲率属性为主曲率。他们代表着法曲率的极值。
高斯曲率：	$K_g = \frac{4AB - C^2}{(1+D^2+E^2)^2}$ 高斯曲率：两个主曲率的乘积即为高斯曲率，又称总曲率，反映某点上总的完全程度。
极大与极小曲率：	$K_{max} = K_m + \sqrt{K_m^2 - K_g}$ $K_{min} = K_m - \sqrt{K_m^2 - K_g}$
最大正曲率、最小负曲率：	$K_+ = (A+B) + \sqrt{(A-B)^2 + C^2}$ $K_- = (A+B) - \sqrt{(A-B)^2 + C^2}$

海森矩阵和主曲率

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

$$\begin{bmatrix} dxx & dxy \\ dyx & dyy \end{bmatrix} \sim \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$$

- 通过对一个曲面在某个点的2X2海森矩阵的特征值分解，可以得到这个曲面的曲率。特征值就是主曲率！（不知道为什么，没学过微分几何，但是这个结论很漂亮，用上即可。）
- （当曲率取最大与最小值的两个法平面方向总是垂直的，这是欧拉在1760年的一个结论，称之为主方向。从现代的观点来看，这个定理来自谱定理因为它们可以作为对应于高斯映射微分的一个对称矩阵的本征向量。）
- 那么根据在某点的x二阶导数、y二阶导数、xy混合导数，可以得到alpha和beta，就是这个曲面的一对主曲率值。
- 根据主曲率的性质，第一这两个曲率分量得方向一定是相互垂直，第二这两个曲率分量一定是在这个曲面的这个点上取到的最大和最小的曲率值。
- 接下来就是看，这个最大和最小值差别到底多大，差别太大，那么他就是边缘了
- 坑，剔除边缘点，计算海森矩阵，来考察主曲率的时候，四个矩阵的元素的计算方法，很多文章都是错的，非常荒谬。

主曲率的工程计算

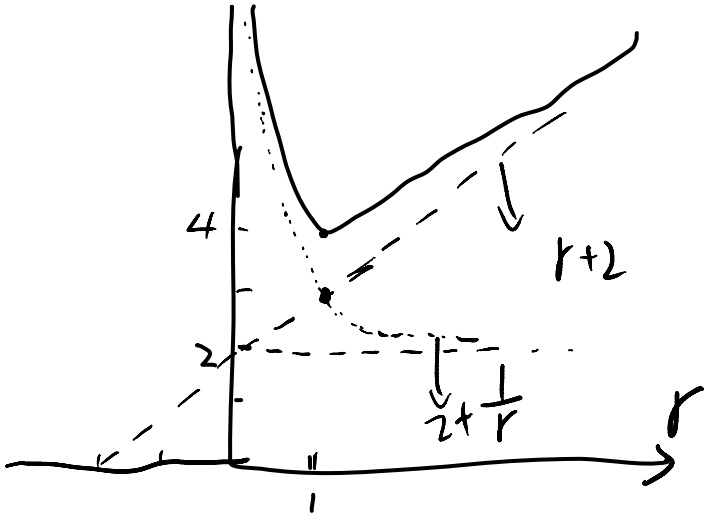
$$\begin{bmatrix} dxx & dxy \\ dyx & dyy \end{bmatrix} \sim \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$$

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

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

$$\alpha = r\beta.$$

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



- 当r>10的时候或者r<0.1的时候，函数值都很大，只有在某个范围区间内，才能说明两个分量的比例差别不大。因此，只有符合如下判断原则，才认为不是边缘点。

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

寻找特征点坐标的OpenCV代码

- 总函数是：**findScaleSpaceExtrema**，其中，调用了**adjustLocalExtrema**

```
void SIFT::findScaleSpaceExtrema( const vector<Mat>& gauss_pyr, const vector<Mat>& dog_pyr,
                                  vector<KeyPoint>& keypoints ) const
```

- 特征点微调和筛选边缘的**OpenCV**代码

```
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 )
```

特征点的描述

特征点的主方向

- 计算以关键点为中心，以3×1.5σ为半径的区域内图像梯度的幅角和幅值，公式如下：

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

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

- 代码是

```
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));
X[k] = dx; Y[k] = dy;
fastAtan2(Y, X, Ori, len, true);
magnitude(X, Y, Mag, len);
```

- 每个点的幅度要乘以一个权重，加入累计统计量

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

- W这个权重，是一个高斯钟，如下

```
expf_scale = -1.f/(2.f * sigma * sigma);
W[k] = (i*i + j*j)*expf_scale;
exp(W, W, len);
```

- 最后统计哪个方向的累积统计量最多，那个就是主方向

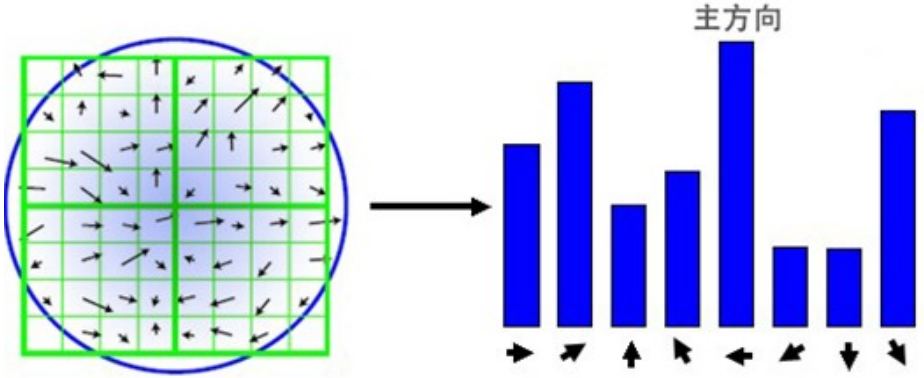
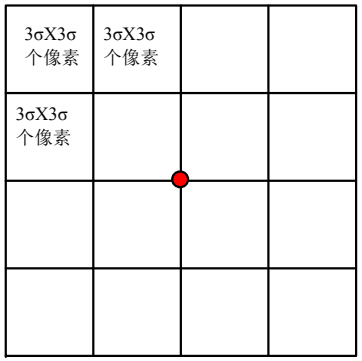


图 5.1 关键点方向直方图

描述之前需要旋转到主方向

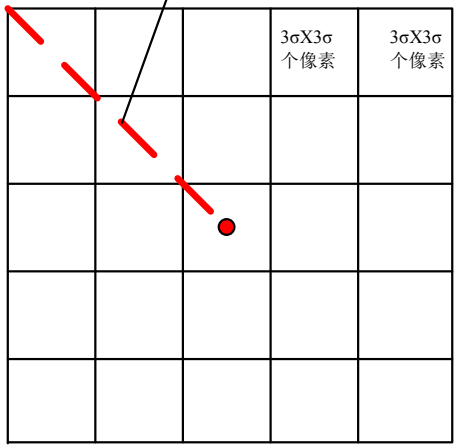
- Lowe建议d=4，m=3，那么旋转后要有d×d个子区域（每个区域间隔为m σ 像元）的话，反向推导，就要求最开始的兴趣区域的像素是下图最右边的那个样子，才能保证，旋转后一定有d×d个子区域，而且每个区域间隔为m σ 像元。

hist_width = SIFT_DESCR_SCL_FCTR * scl;
//SIFT_DESCR_SCL_FCTR=3,就是m，原点3倍的σ之外，认为高斯模糊已经忽略
//scl是σ,是当前的高斯模糊尺度
hist_width就是mσ

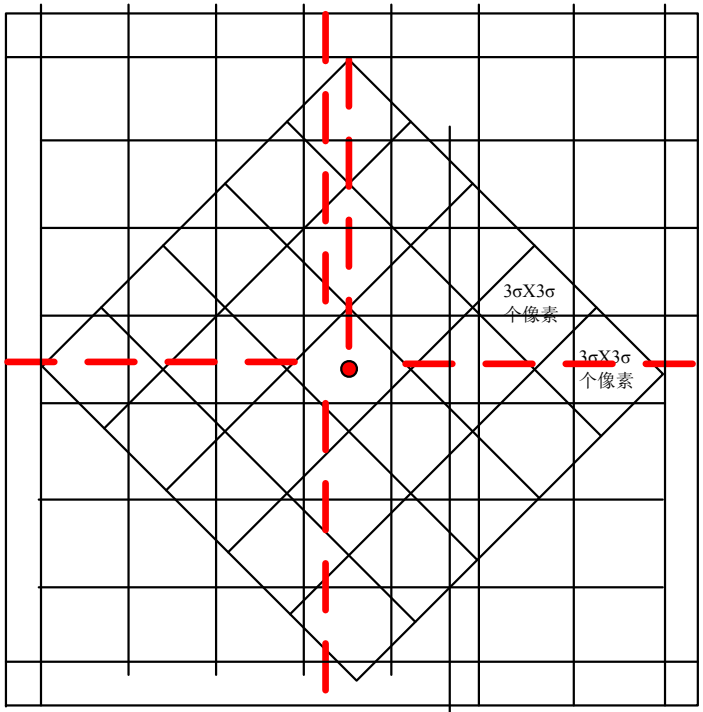


4X4 grid of locations
16个子区域
那么d=4

radius = cvRound(hist_width * 1.4142135623730951f * (d + 1) * 0.5f);



hist_width * (d + 1)
考虑到要三线性插值，因此多加一个grid区域



这样的方框一共有len个像素
len = (radius*2+1)*(radius*2+1),
主循环体的遍历为
for(i = -radius, k = 0; i <= radius; i++)
for(j = -radius; j <= radius; j++)

执行旋转的代码

- i和j从-radius到0到+radius遍历，计算float dx和float dy，放入X[k] = dx; Y[k] = dy，同时需要判断ij这个点是否在旋转后的4X4的区域内
- 判断方法如下：
- 第一步，执行旋转float c_rot = j * cos_t - i * sin_t;，float r_rot = j * sin_t + i * cos_t;对应公式为

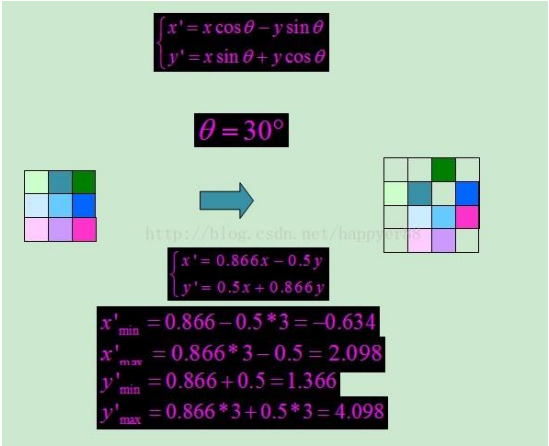
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

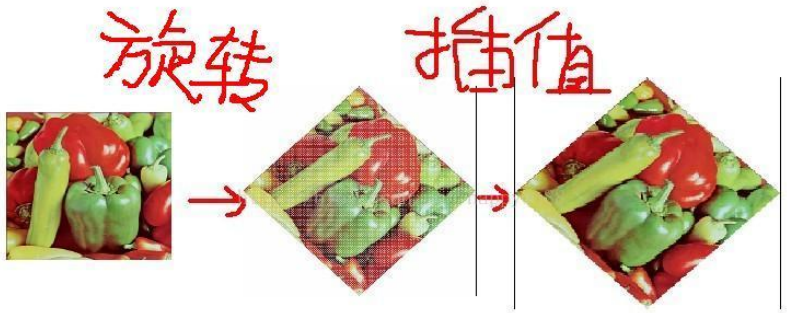
- 第二步，从中心坐标取得起点坐标，代码为float rbin = r_rot + d/2 - 0.5f;float cbin = c_rot + d/2 - 0.5f;
- 第三步，判断是否超出d的范围，代码为if(rbin > -1 && rbin < d && cbin > -1 && cbin < d && r > 0 && r < rows - 1 && c > 0 && c < cols - 1)
- 只有在这个区域内，那么除了记录dx和dy外，还要记录起点坐标并存储RBin[k] = rbin; CBin[k] = cbin;同时，计算高斯权重W[k] = (c_rot * c_rot + r_rot * r_rot)*exp_scale;（个人觉得高斯权重完全可以从ij平方和获得，节约计算资源）

```
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.
      float c_rot = j * cos_t - i * sin_t;
      float r_rot = j * sin_t + i * cos_t;
      float rbin = r_rot + d/2 - 0.5f;
      float cbin = c_rot + d/2 - 0.5f;
      int r = pt.y + i, c = pt.x + j;
      if( rbin > -1 && rbin < d && cbin > -1 && cbin <
    d &&
          r > 0 && r < rows - 1 && c > 0 && c < cols - 1
    )
      {
          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));
          X[k] = dx; Y[k] = dy; RBin[k] = rbin; CBin[k] =
    cbin;
          W[k] = (c_rot * c_rot + r_rot * r_rot)*exp_scale;
          k++;
      }
```

旋转后插值

- 在opencv中旋转图片时，旋转后的图片中会出现许多白点，对这些白点进行填充，就是插值处理。在二维图像中，就是做双线性插值





- 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。

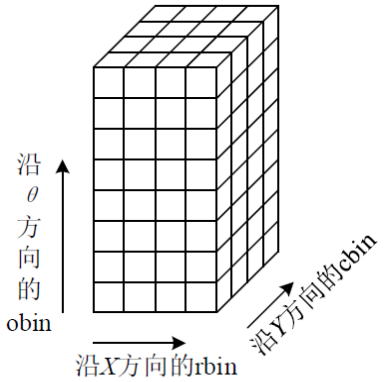


图 3 描述符的三维直方图

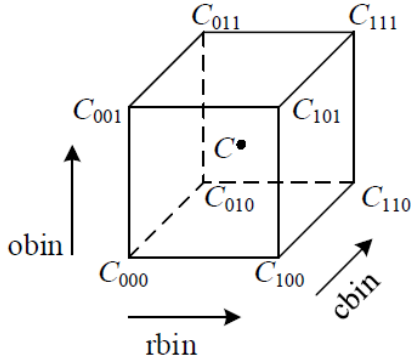


图 4 三维直方图中的正方体

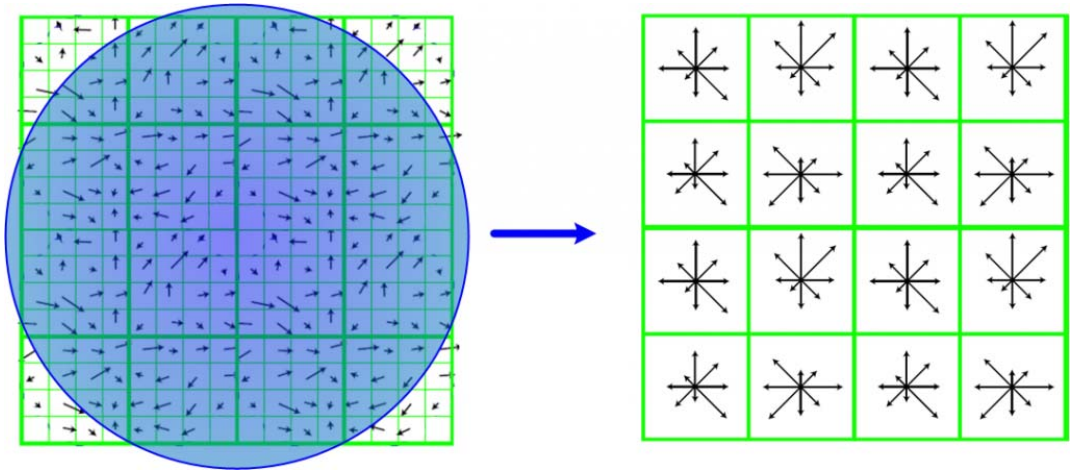
插值的代码

- 这里r0c0o0都是向下取整，并且rbin -= r0; cbin -= c0; obin -= o0;，意味着rbin、cbin、obin这三个变量，分别是像素C点，距离这个正方体右上角C111的距离。
- 以C111点为例，v_rco111=mag*rbin*cbin*obin，在程序里头，就对应着float v_rco111 = v_rc11*obin，其中，float v_rc11 = v_r1*cbin，其中，float v_r1 = mag*rbin。
- 其他点以此类推。

```
for( k = 0; k < len; k++ )
    {
        float rbin = RBin[k], cbin = CBin[k];
        float obin = (Ori[k] - ori)*bins_per_rad;
        float mag = Mag[k]*W[k];
        int r0 = cvFloor( rbin );
        int c0 = cvFloor( cbin );
        int o0 = cvFloor( obin );
        rbin -= r0;
        cbin -= c0;
        obin -= o0;
                    //判断邻域像素的梯度幅角属于36 个柱体的哪一个
        //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( 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 个立方的积，当然mag是高斯加权后的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
//r0从0开始，一行有d+2个，c0也是从0开始，因此index都要+1，r和c
        的平面不断重复遍历的话，提高一层则再次计算，
        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;
    }
```

定位特征点的描述子的区域

- 将旋转后区域划分为d×d个子区域（每个区域间隔为mσ像元），在子区域内计算8个方向的梯度直方图，绘制每个方向梯度方向的累加值，形成一个种子点。
- 与求主方向不同的是，此时，每个子区域梯度方向直方图将0°~360°划分为8个方向区间，每个区间为45°。即每个种子点有8个方向区间的梯度强度信息。由于存在d×d，即4×4个子区域，所以最终共有4×4×8=128个数据，形成128维SIFT特征矢量。
- 这128维向量就是这个特征点的描述。



描述子向量门限

- 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。
- 非线性光照，相机饱和度变化对造成某些方向的梯度值过大，而对方向的影响微弱。因此设置门限值(向量归一化后，一般取0.2)截断较大的梯度值。然后，再进行一次归一化处理，提高特征的鉴别性。
- 先计算总的平方和，让那些大于平方和的平方根的20%的值，设置成平方和的平方根的20%。
- 然后归一化，归一化的原则是，让这些向量元素的平方和等于SIFT_INT_DESCR_FCTR，默认的SIFT_INT_DESCR_FCTR是等于512.

最后，dst就是特征向量了！

```
        // 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];
        float thr = std::sqrt(nrm2)*SIFT_DESCR_MAG_THR;
//SIFT_DESCR_MAG_THR=0.2
        for( i = 0, nrm2 = 0; i < k; i++ )
        {
            float val = std::min(dst[i], thr);// 描述子向量门限。非线性光照，相机饱和度变化对造成某些方向的梯度值过大，而对方向的影响微弱。因此设置门限值(向量归一化后，一般取0.2)截断较大的梯度值。然后，再进行一次归一化处理，提高特征的鉴别性。
            dst[i] = val;
            nrm2 += val*val;
        }
        nrm2 = SIFT_INT_DESCR_FCTR/std::max(std::sqrt(nrm2),
FLT_EPSILON);
    // FLT_EPSILON是最小精确度，避免除以0
    #if 1
        for( k = 0; k < len; k++ )
        {
            dst[k] = saturate_cast<uchar>(dst[k]*nrm2);
        }
```

SIFT的缺点

- SIFT在图像的不变特征提取方面拥有无与伦比的优势，但并不完美，仍然存在：
- 1. 实时性不高。
- 2. 有时特征点较少。
- 3. 对边缘光滑的目标无法准确提取特征点。
- 等缺点，如下图7.1所示，对模糊的图像和边缘平滑的图像，检测出的特征点过少，对圆更是无能为力。近来不断有人改进，其中最著名的有SURF和CSIFT。

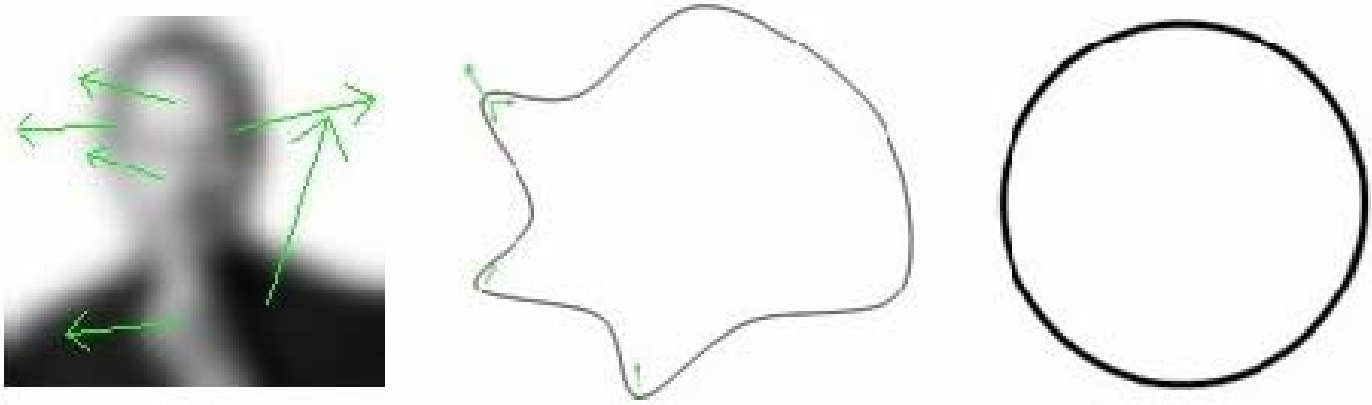


图 7.1 sift 算法检测效果

实践案例

安装环境

- ubuntu安装opencv以及SIFT非免费开发包

```
sudo apt-get update
sudo apt-get install libopencv-nonfree-dev
sudo apt-get install libopencv-dev
```

找出关键点

- 找出关键点
- 图片为f0.jpg
- 编译语句为g++，而不是gcc，否则出错

```
zcr@zcr-virtual-machine:~/SIFT-h4$ g++ SIFT_f1.cpp  -o SIFT_f1 `pkg-config --cflags --libs opencv`
```

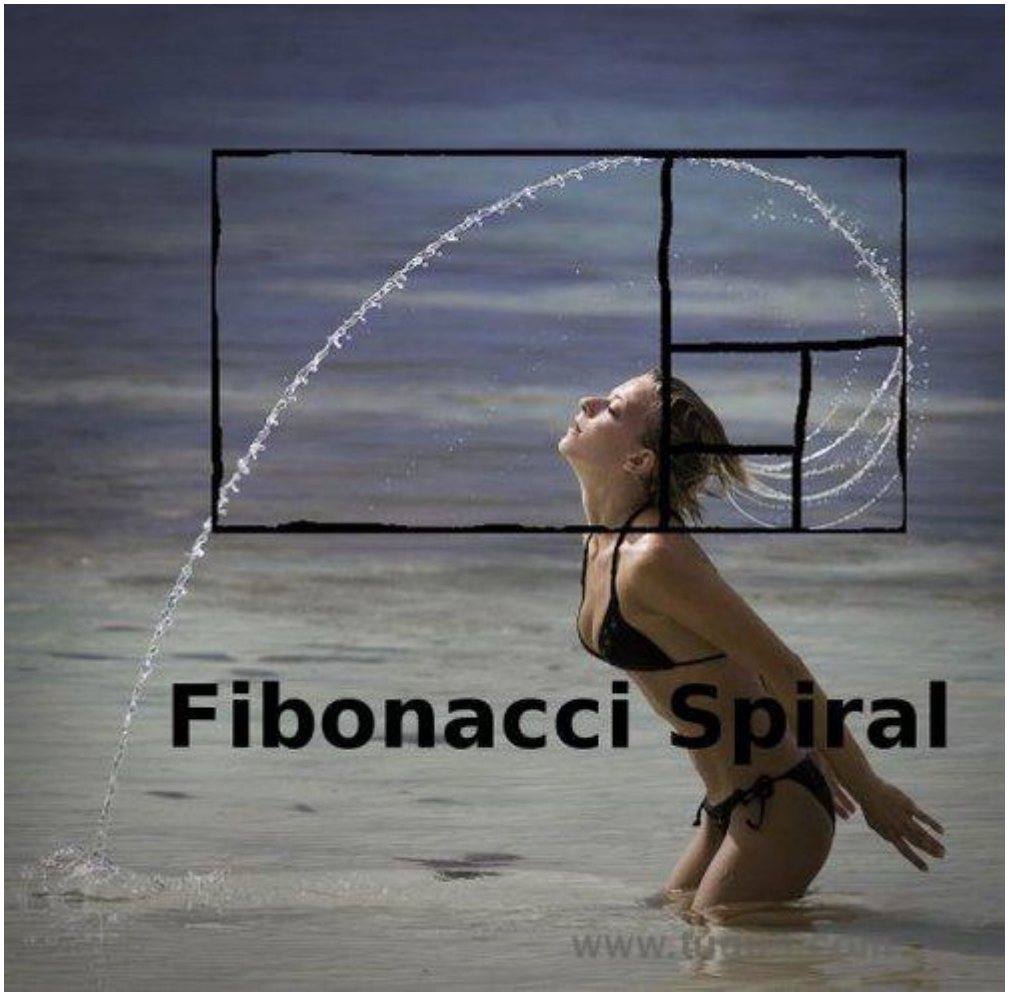
```
//#include "opencv2/sift-ZCR.cpp"
#include "opencv2/opencv.hpp"
#include "opencv2/core/core.hpp"
#include "highgui.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/opencv_modules.hpp"
#include "opencv2/nonfree/nonfree.hpp"
using namespace cv;
//using namespace std;
int main(int argc, char** argv)
{
Mat img = imread("f0.jpg");
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;
}
```

图像匹配

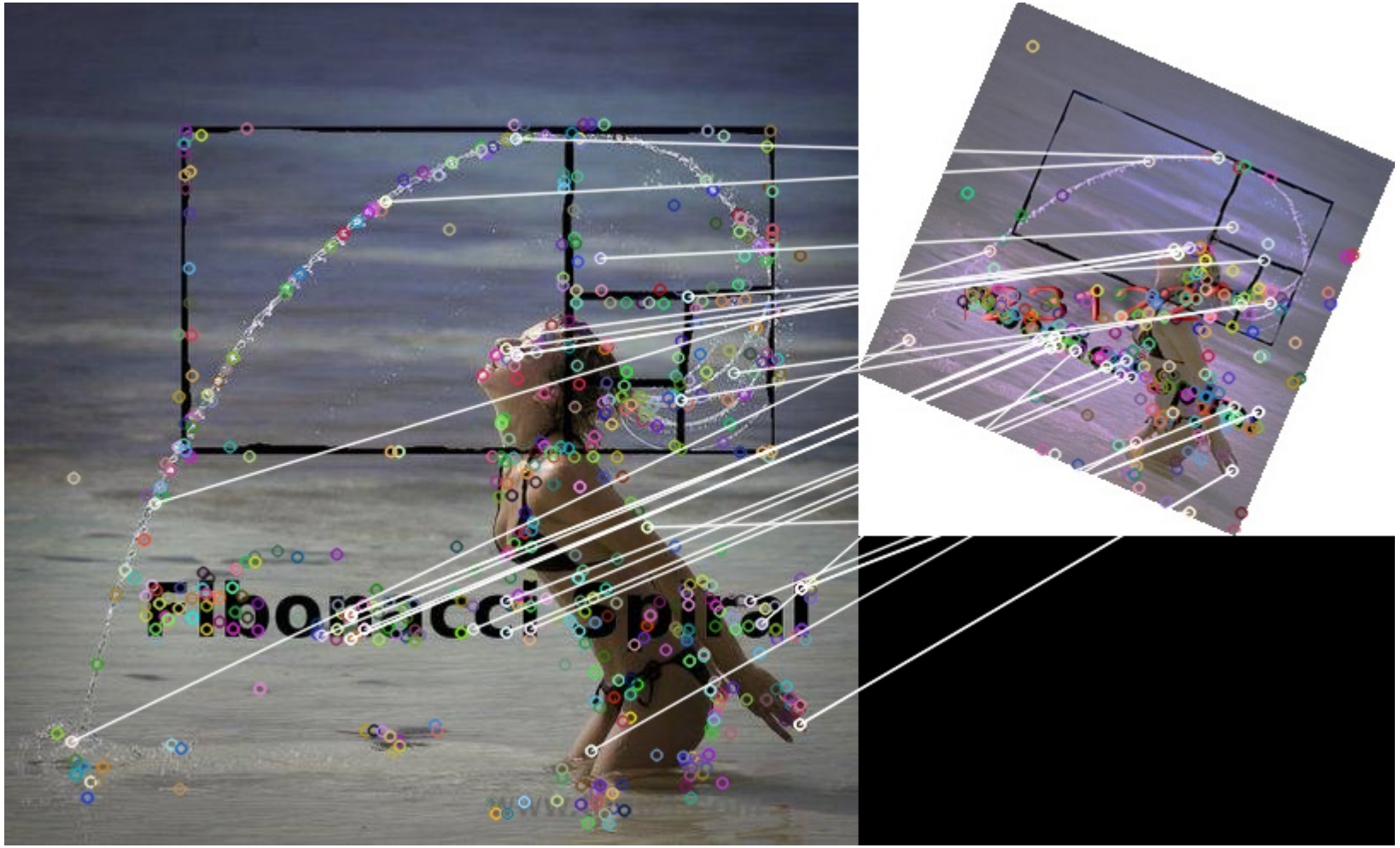
- 图片为Mat img1 = imread("f0.jpg");
- Mat img2 = imread("f1.jpg");
- 输出图片为imwrite("SIFT_match.jpg", img_matches);
- 编译用g++

```
zcr@zcr-virtual-machine:~/SIFT-h4$ g++ SIFT_match.cpp  -o SIFT_match `pkg-config --cflags --libs opencv`
```

```
#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("f0.jpg");
Mat img2 = imread("f1.jpg");
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);
imwrite( "SIFT_match.jpg", img_matches );
waitKey(0);
return 0;
}
```


Fibonacci Spiral


Fibonacci Spiral


Fibonacci Spiral

福州GDG社区的资源

Google+

https://plus.google.com/+FzgdgOrg/

如果网页访问有困难，发送订阅邮件到**gdg-fuzhou+subscribe@googlegroups.com**

Google

网上论坛

https://groups.google.com/forum/#!forum/gdg-fuzhou

Google+

社群

https://plus.google.com/communities/113834866481619712463

Google Developers

搜索

https://developers.google.com/groups/chapter/101458543060997187108/

日常沟通交流

- 微信公众号
- ID:福州GDG

- 交流QQ群
- 群号209723070

福州GDG

谢谢

Eric ZHANG

indeedzcr@gmail.com