copencv源码分析

源码分析目录：

1. 直方图均衡化
2. 高斯滤波
3. 坎尼算子
4. 形态学腐蚀和膨胀

主要知识点目录：

1. 灰度变换
2. 直方图均衡化
3. 卷积
4. 图像平滑
5. 图像锐化
6. 边缘检测
7. 门限处理
8. 形态学腐蚀
9. 形态学膨胀

文档特色：

本opencv源码分析报告所分析的算法主要基于但不限于课堂所教授的知识，课堂所教授的知识在源码分析的过程中得到了很多体现，但也有适度的延伸，到达了在实践中加深对知识的理解的目的。

一、opencv源码：直方图均衡化

1. 算法原理

直方图均衡化是一种通过拉伸像素强度分布来增强图像对比度的方法，直方图均衡化的算法原理如名字一样，先求直方图，而后做均衡化，分为两部分：

（1）直方图

关于直方图函数的推来过程在数字图像处理的教材上有着详细的推到，这里不做赘述。计算公式：

$$P_r(r_k)=\frac{n_k}{MN},\quad k=0,1,2...L-1$$

说明：MN是图像的像素总数量，nk表示灰度为rk的像素个数，L表示灰度级的数量，例如8bit图像，灰度 级为256；与rk相对的pr(rk)图形通常称为直方图。也就是说，图像的直方图就是图像中每个灰度级的像素在图 像中出现的频率。获得直方图以后，就需要进行均衡化处理。

（2）均衡化

图像均衡化处理其实是积分的过程，对于数字图像的离散值便是求和。计算公式：

$$S_k=\frac{L-1}{MN}\sum_{j=0}^{k}n_k,\quad k=0,1,2...L-1$$

说明：均衡化公式结果其实就是(L-1)与像素出现频率累加和的乘积。

2. API介绍

equalizeHist 函数：

```
void cv::equalizeHist(InputArray src, OutputArray dst)
```

必要参数：

- src，输入图像，即源图像，填 Mat 类的对象即可，但需要为 8 位单通道的图像；
- dst，输出结果，需要和源图像有一样的尺寸和类型。

3. 源码分析

虽然均衡化算法的原理很简单，但是在opencv工程源代码代码中也并不简单，其中涉及到各个平台的优化的工作还是很复杂的，接下来我不深入分析各个平台优化的代码，只针对最关键的一部分源码给出分析，目的也在于能够更好的阐述算法的原理。

opencv源码：equalizeHist 函数

源码说明：

该函数的均衡化计算过程主要分3步：

- 计算直方图(calcBody)
- 计算映射表：映射表是一个数组，索引就是灰度级，即0-255，对应的值就是该灰度级均衡化以后的灰度级。例如Lut[6]=36；表示图片中像素值为6的像素，均衡化以后，像素值为36；
- 利用映射表进行均衡化：运用映射表，将原图像中的像素替换为均衡化以后的像素，结果存入参数dst。

源码注释：

```
void cv::equalizeHist( InputArray _src, OutputArray _dst )
{
    CV_INSTRUMENT_REGION();

    CV_Assert( _src.type() == CV_8UC1 );

    if (_src.empty())
        return;
```

```
9
10      CV_OCL_RUN(_src.dims() <= 2 && _dst.isUMat(),
11                 ocl_equalizeHist(_src, _dst))
12
13      Mat src = _src.getMat();
14      _dst.create( src.size(), src.type() );
15      Mat dst = _dst.getMat();
16
17      CV_OVX_RUN(!ovx::skipSmallImages<VX_KERNEL_EQUALIZE_HISTOGRAM>(src.cols,
src.rows),
18                 openvx_equalize_hist(src, dst))
19
20      Mutex histogramLockInstance;
21
22      // HIST_SZ = 256;
23      const int hist_sz = EqualizeHistCalcHist_Invoker::HIST_SZ;
24      int hist[hist_sz] = {0,};
25      int lut[hist_sz];
26
27      //初始化计算程序
28      EqualizeHistCalcHist_Invoker calcBody(src, hist,
&histogramLockInstance);
29      EqualizeHistLut_Invoker      lutBody(src, dst, lut);
30      cv::Range heightRange(0, src.rows);
31
32      //判断书否使用多线程
33      if(EqualizeHistCalcHist_Invoker::isWorthParallel(src))
34          //计算直方图
35          parallel_for_(heightRange, calcBody);
36      else
37          //计算直方图
38          calcBody(heightRange);
39
40      int i = 0;
41      //查找图片中第一个频率不为0的灰度级i
42      while (!hist[i]) ++i;
43
44      int total = (int)src.total();
45      //如果第一个出现频率不为0的灰度级的出现的频次与输入图像的像素总数相等；
46      //也就是说整幅图片就一个灰度级，那么均衡化的结果为dst所有像素为i
47      if (hist[i] == total)
48       {
49           dst.setTo(i);
50           return;
51       }
52
53       //这里就是上文均衡化公式的(L-1)/MN
54       //需要注意这里分母减去hist[i];这里是为了剔除前i个连续的频次为0的灰度级
55       //前n个连续的频次为0的点在累加过程中是没有作用的，所以剔除
56       float scale = (hist_sz - 1.f)/(total - hist[i]);
57       int sum = 0;
58
59      //计算Lut
60       for (lut[i++] = 0; i < hist_sz; ++i)
61       {
62           sum += hist[i];
63           lut[i] = saturate_cast<uchar>(sum * scale);
64       }
```

```
    //图片均衡化
    //判断是否使用多线程并行
     if(EqualizeHistLut_Invoker::isWorthParallel(src))
         parallel_for_(heightRange, lutBody);
     else
         lutBody(heightRange);
 }
```

opencv源码：函数`EqualizeHistCalcHist_Invoker`

源码说明：

函数`EqualizeHistCalcHist_Invoker`在`cv::equalizeHist`中被调用，利用`cv::equalizeHist`中已经计算得到的映射表（即：变量lut[]）完成后续的均衡化步骤。

源码注释：

```
class EqualizeHistCalcHist_Invoker : public cv::ParallelLoopBody
{
public:
    enum {HIST_SZ = 256};

    EqualizeHistCalcHist_Invoker(cv::Mat& src, int* histogram, cv::Mutex*
histogramLock)
        : src_(src), globalHistogram_(histogram),
histogramLock_(histogramLock)
    { }
    //计算主体
    void operator()( const cv::Range& rowRange ) const CV_OVERRIDE
    {
        int localHistogram[HIST_SZ] = {0, };

        const size_t sstep = src_.step;

        int width = src_.cols;
        int height = rowRange.end - rowRange.start;
        //如果内存是连续的，就把二维空间转换为1维
        if (src_.isContinuous())
        {
            width *= height;
            height = 1;
        }
        //for循环遍历src，计数各个灰度级出现的频次
        for (const uchar* ptr = src_.ptr<uchar>(rowRange.start); height--;
ptr += sstep)
        {
            int x = 0;
            //在width方向上对for循环进行4阶展开
            //如果内存连续，则height方向只循环一次
            for (; x <= width - 4; x += 4)
            {
                //读取原图像像素值
                int t0 = ptr[x], t1 = ptr[x+1];
                //对应灰度级频次计数加以1
                localHistogram[t0]++; localHistogram[t1]++;
                t0 = ptr[x+2]; t1 = ptr[x+3];
```

```
                localHistogram[t0]++; localHistogram[t1]++;
            }
            //处理循环展开的边界
            for (; x < width; ++x)
                localHistogram[ptr[x]]++;
        }
        //对于多线程，此处需要同步
         cv::AutoLock lock(*histogramLock_);
        //汇总每个线程计算分块的局部直方图，变为整个图像的全局直方图
         for( int i = 0; i < HIST_SZ; i++ )
             globalHistogram_[i] += localHistogram[i];
    }
   //该函数用于判断是否使用多线程或者说并行for循环
    static bool isWorthParallel( const cv::Mat& src )
    {
        //如果图像大于640*480就使用并行for循环
        return ( src.total() >= 640*480 );
    }

 private:
    EqualizeHistCalcHist_Invoker& operator=(const
EqualizeHistCalcHist_Invoker&);

    cv::Mat& src_;
    int* globalHistogram_;
    cv::Mutex* histogramLock_;
 };
```

二、opencv源码：高斯滤波

1. 算法原理

数字图像的滤波说白了，就是对原图像的每一个像素滤波，每个像素滤波后的值是根据其相邻像素（包括自己那个点）与一个滤波模板的对应项的乘积和。具体到高斯滤波，我们只要知道这个高斯滤波的模板即可。

高斯函数公式：

$$G(x,y)=\frac{1}{2\pi\sigma_x\sigma_y}e^{-(\frac{x^2}{2\sigma_x^2}+\frac{y^2}{2\sigma_y^2})}=\frac{1}{\sqrt{2\pi}\sigma_x}e^{-(\frac{x^2}{2\sigma_x^2})}*\frac{1}{\sqrt{2\pi}\sigma_y}e^{-\left(\frac{y^2}{2\sigma_y^2}\right)}=G(x)*G(y)$$

由上面这个公式可知，二维高斯函数可以看成两个一维高斯函数乘积，因此可以先计算一维高斯模板，再计算需要的二维高斯模板。正是由于GaussianBlur是一种可分离滤波器，在opencv的高斯滤波实现中，opencv可以先对行滤波,再对列滤波的方式进行滤波。

高斯滤波模板中最重要的参数就是高斯分布的标准差σ。它代表着数据的离散程度，如果σ较小，那么生成的模板中心系数越大，而周围的系数越小，这样对图像的平滑效果就不是很明显；相反，σ较大时，则生成的模板的各个系数相差就不是很大，比较类似于均值模板，对图像的平滑效果就比较明显。

2. API介绍

GaussianBlur函数：

```
void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX,
double sigmaY=0, int borderType=BORDER_DEFAULT);
```

函数参数：

src：输入图像，即源图像，填Mat类的对象即可。它可以是单独的任意通道数的图片，但需要注意，图片深　　度应该为CV_8U,CV_16U, CV_16S, CV_32F 以及 CV_64F之一。

dst：即目标图像，需要和源图片有一样的尺寸和类型。比如可以用Mat::Clone，以源图片为模板，来初始化　　得到如假包换的目标图。

ksize：高斯内核的大小。其中ksize.width和ksize.height可以不同，但他们都必须为正数和奇数（并不能理　　解）。或者，它们可以是零的，它们都是由sigma计算而来。

sigmaX：表示高斯核函数在X方向的的标准偏差。

sigmaY：表示高斯核函数在Y方向的的标准偏差。若sigmaY为零，就将它设为sigmaX，如果sigmaX和　　sigmaY都是0，那么就由ksize.width和ksize.height计算出来。

3. 源码分析

opencv源码：GaussianBlur函数

作用：

输入原图和相关参数，对原图进行高斯滤波处理并将结果输出。

源码说明：

GaussianBlur函数本质上还是调用filter2D,opencv在GaussianBlur函数中只是针对特殊情况进行了GPU和CPU版本的优化,如果输入的维度等信息不满足特殊情况,则选择调用filter2D进行计算而不做优化。opencv调用的filter2D其实是sepFilter2D,这是一种可分离的二维滤波器,这是出于优化考虑。

源码注释：

```
void cv::GaussianBlur( InputArray _src, OutputArray _dst, Size ksize,
                   double sigma1, double sigma2,
                   int borderType )
{
  //初始化及边界类型等的判断
    CV_INSTRUMENT_REGION()

    int type = _src.type();
    Size size = _src.size();
    _dst.create( size, type );

    if( borderType != BORDER_CONSTANT && (borderType & BORDER_ISOLATED) != 0
)
    {
        if( size.height == 1 )
            ksize.height = 1;
        if( size.width == 1 )
            ksize.width = 1;
    }
//高斯滤波器如果尺寸为1,根据高斯函数可以知道该系数为1,即将输入复制到输出
```

```
    if( ksize.width == 1 && ksize.height == 1 )
    {
        _src.copyTo(_dst);
        return;
    }
//OpenCV中针对一些ksize = 3和5的情况做了OpenCL优化,所以初始化OpenCL相关函数
    bool useOpenCL = (ocl::isOpenCLActivated() && _dst.isUMat() &&
_src.dims() <= 2 &&
                ((ksize.width == 3 && ksize.height == 3) ||
                (ksize.width == 5 && ksize.height == 5)) &&
                _src.rows() > ksize.height && _src.cols() > ksize.width);
    (void)useOpenCL;

    int sdepth = CV_MAT_DEPTH(type), cn = CV_MAT_CN(type);
//获取gaussianKernels
    Mat kx, ky;
    createGaussianKernels(kx, ky, type, ksize, sigma1, sigma2);
//调用opencl进行计算
    CV_OCL_RUN(useOpenCL, ocl_GaussianBlur_8UC1(_src, _dst, ksize,
CV_MAT_DEPTH(type), kx, ky, borderType));
//如果不是ksize=3或者5的情况,考虑使用filter2D的opencl优化程序计算
    CV_OCL_RUN(_dst.isUMat() && _src.dims() <= 2 && (size_t)_src.rows() >
kx.total() && (size_t)_src.cols() > kx.total(),
                ocl_sepFilter2D(_src, _dst, sdepth, kx, ky, Point(-1, -1), 0,
borderType))
//如果OpenCL版的filter2D依然不能计算,则选择cpu版本的gaussianBlur
    Mat src = _src.getMat();
    Mat dst = _dst.getMat();

    Point ofs;
    Size wsz(src.cols, src.rows);
    if(!(borderType & BORDER_ISOLATED))
        src.locateROI( wsz, ofs );

    CALL_HAL(gaussianBlur, cv_hal_gaussianBlur, src.ptr(), src.step,
dst.ptr(), dst.step, src.cols, src.rows, sdepth, cn,
             ofs.x, ofs.y, wsz.width - src.cols - ofs.x, wsz.height -
src.rows - ofs.y, ksize.width, ksize.height,
             sigma1, sigma2, borderType&~BORDER_ISOLATED);

    CV_OVX_RUN(true,
                openvx_gaussianBlur(src, dst, ksize, sigma1, sigma2,
borderType))

    CV_IPP_RUN_FAST(ipp_GaussianBlur(src, dst, ksize, sigma1, sigma2,
borderType));
//若CPU版本的gaussianBlur仍然不能计算,则选择CPU版本的filter2D
    sepFilter2D(src, dst, sdepth, kx, ky, Point(-1, -1), 0, borderType);
}
```

opencv源码：GaussianKernel函数

源码说明：

根据高斯函数的分布特性,可以知道,函数分布在区间[u - 3 * sigma, u + 3 * sigma]范围内的概率大于99%.因此模板大小的选取往往与sigma有关.。看代码中的公式,ksize = round(2 * 3 * sigma + 1) | 1；注意与1按位或,是保证结果为奇数。

需要注意,opencv认为当图像类型为CV_8U的时候能量集中区域为3 * sigma，其他类型图像的能量集中区域为4*sigma.。接着往下看,可知opencv中获取了两个方向的GaussianKernels，即：kx和ky。当两个方向的sigma相同、尺寸相同的时，两个方向上的kernels是相同的。因为GaussianBlur是一种可分离滤波器，为减少计算量，opencv采用先对行滤波,再对列滤波的方式进行滤波，这是一种优化方式。

源码注释：

```
static void createGaussianKernels( Mat & kx, Mat & ky, int type, Size ksize,
                                   double sigma1, double sigma2 )
{
//初始化
    int depth = CV_MAT_DEPTH(type);
    if( sigma2 <= 0 )
        sigma2 = sigma1;
//如果用户没有设置ksize,则需要根据sigma设定ksize
    if( ksize.width <= 0 && sigma1 > 0 )
        ksize.width = cvRound(sigma1*(depth == CV_8U ?3 :4)*2 + 1)|1;
    if( ksize.height <= 0 && sigma2 > 0 )
        ksize.height = cvRound(sigma2*(depth == CV_8U ?3 :4)*2 + 1)|1;
//判断ksize是否合法
    CV_Assert( ksize.width > 0 && ksize.width % 2 == 1 &&
        ksize.height > 0 && ksize.height % 2 == 1 );
//保证sigma合法
    sigma1 = std::max( sigma1, 0. );
    sigma2 = std::max( sigma2, 0. );
//获取GaussianKernels,其数据类型为float或者double
    kx = getGaussianKernel( ksize.width, sigma1, std::max(depth, CV_32F) );
    if( ksize.height == ksize.width && std::abs(sigma1 - sigma2) <
DBL_EPSILON )
        ky = kx;
    else
        ky = getGaussianKernel( ksize.height, sigma2, std::max(depth,
CV_32F) );
}

}
```

opencv源码：getGaussianKernel函数

源码说明：

这个函数最终确定了gaussianKernels的在不同情形下的两种计算规则：

I. 取固定系数： 当kernels的尺寸为1,3,5,7 且用户没有设置sigma的时候(sigma <= 0),就会取固定的系 数。这是一种默认的值是高斯函数的近似。

II. 按照高斯公式计算： 当kernels尺寸超过7的时,若sigma设置合法(用户设置了sigma),则按高斯公式计 算。若sigma不合法(用户没有设置sigma),按((n-1)0.5 - 1)0.3 + 0.8计算.n为kernels的尺寸。

源码注释:

```
cv::Mat cv::getGaussianKernel( int n, double sigma, int ktype )
{
    const int SMALL_GAUSSIAN_SIZE = 7;
    //定义了固定的filter即Kernels.
    static const float small_gaussian_tab[][SMALL_GAUSSIAN_SIZE] =
    {
        {1.f},
        {0.25f, 0.5f, 0.25f},
        {0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f},
        {0.03125f, 0.109375f, 0.21875f, 0.28125f, 0.21875f, 0.109375f,
0.03125f}
    };
    //对滤波器的类型进行判断,1,尺寸为奇数;2,尺寸小于等于7;3.sigma小于等于0(注)
    const float* fixed_kernel = n % 2 == 1 && n <= SMALL_GAUSSIAN_SIZE &&
sigma <= 0 ?
        small_gaussian_tab[n>>1] :0;
    //Kernels的数据类型为float,double也是ok的.
    CV_Assert( ktype == CV_32F || ktype == CV_64F );
    Mat kernel(n, 1, ktype);
    float* cf = kernel.ptr<float>();
    double* cd = kernel.ptr<double>();
    //确定sigma,如果sigma > 0,ok,不用修改;否则按照公式计算(注)
    double sigmaX = sigma > 0 ?sigma :((n-1)*0.5 - 1)*0.3 + 0.8;
    double scale2X = -0.5/(sigmaX*sigmaX);//高斯公式
    double sum = 0;

    int i;
    for( i = 0; i < n; i++ )
    {
        double x = i - (n-1)*0.5;
        //如果fixed_kernel为真,也就是符合上文中的3个条件,则取固定的系数;否则按照高斯公
式计算
        double t = fixed_kernel ?(double)fixed_kernel[i]
:std::exp(scale2X*x*x);
        //对kernels进行归一化
        if( ktype == CV_32F )
        {
            cf[i] = (float)t;
            sum += cf[i];
        }
        else
        {
            cd[i] = t;
            sum += cd[i];
        }
    }

    sum = 1./sum;
    for( i = 0; i < n; i++ )
    {
        if( ktype == CV_32F )
            cf[i] = (float)(cf[i]*sum);
        else
            cd[i] *= sum;
    }
```

```
    return kernel;
}
```

三、opencv源码：坎尼算子

1. 算法原理

（1）用高斯滤波器平滑图像

进行高斯滤波之前，需要先得到一个高斯滤波器（kernel）。如何得到一个高斯滤波器？其实就是将高　斯函数离散化，将滤波器中对应的横纵坐标索引代入高斯函数，即可得到对应的值。不同尺寸的滤波器，得　到的值也不同，下面是 (2k+1)x(2k+1) 滤波器的计算公式 :

$$H[i,j]=\frac{1}{2\Pi\sigma^2}e^{-\frac{(i-1-k)^2+(j-k-1)^2}{2\sigma^2}}$$

注意：opencv的Canny函数里面没有进行高斯平滑，所以调用该函数之前需要进行平滑。

（2）用sobel算子来计算两个方向的梯度

step1：运用一对卷积阵列近似梯度 (分别作用于 x 和 y 方向):

$$G_x=\begin{bmatrix}-1&0&+1\\-2&0&+2\\-1&0&+1\end{bmatrix}\ G_y=\begin{bmatrix}-1&-2&-1\\0&0&0\\+1&+2&+1\end{bmatrix}$$

step2：使用下列公式计算梯度幅值和方向:

$$G=\sqrt{G_x^{\ 2}+G_y^{\ 2}}$$

$$\theta=\arctan(\frac{G_y}{G_x})$$

opencv的Canny中有***L2gradient参数***:

如果true，计算梯度幅度的时候会使用：（更加精确）

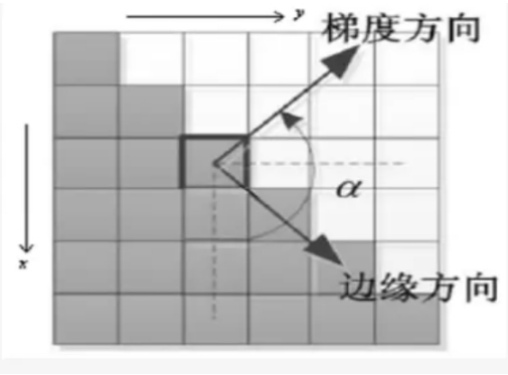
$$G=\sqrt{G_x^{\ 2}+G_y^{\ 2}}$$

如果为false，计算梯度幅度的时候会使用：（计算量小）

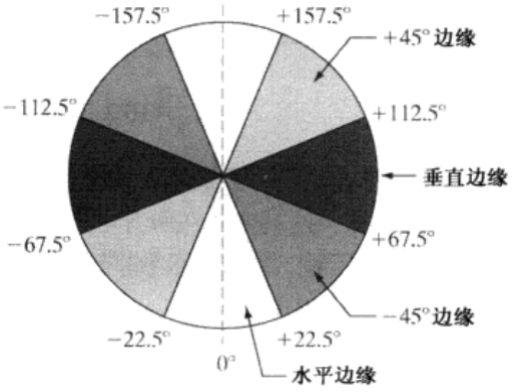
$$G=|G_x|+|G_y|$$

（3）对梯度幅值进行非极大值抑制

理想情况下，最终得到的边缘是很细的。因此，需要执行非极大值抑制以使边缘变细。原理：遍历梯度　矩阵上的所有点，保留边缘方向上具有极大值的像素。本质：指定边缘法线的许多离散方向（梯度向量）。



梯度方向近似：opencv的canny算法中定义4个基本方向，分别是水平、+45度、垂直、-45度，这4个方　向把360度等分为4个子范围，每个子范围以这4个基本方向为对称轴。当边缘法线落在某个子范围就认为边缘　法线的方向是该子范围对应的方向。



非极大抑制的条件：如果某点处的梯度幅度不是局部最大，则对该点进行抑制操作。

（4）用双阈值算法检测和连接边缘

case1：如果某一像素位置的幅值超过 高阈值, 该像素被保留为边缘像素。

case2：如果某一像素位置的幅值小于 低阈值, 该像素被排除。

case3：如果某一像素位置的幅值在两个阈值之间,该像素仅仅在连接到一个高于 高阈值的像素时被保留。

2. API介绍

opencv的Canny函数的原型：

```
void cv::Canny( InputArray _src, OutputArray _dst,
                double low_thresh, double high_thresh,
                int aperture_size, bool L2gradient )
```

必要参数：

- _src：需要处理的原图像，该图像必须为单通道的灰度图；
- _dst：Canny算法处理后的图像；
- low_thresh：低阈值；
- high_thresh：高阈值。
- apertureSize：卷积核的大小（opencv使用的是Sobel算子）；

- L2gradient：一个布尔值，如果为真，则使用更精确的L2范数进行计算（即两个方向的导数的平方和再开方），否则使用计算量小但更加不准确的L1范数（直接将两个方向导数的绝对值相加）。

3. 源码分析

opencv源码：canny算法

```
#include "precomp.hpp"


#ifdef USE_IPP_CANNY
namespace cv
{
static bool ippCanny(const Mat& _src, Mat& _dst, float low,  float high)
{
    int size = 0, size1 = 0;
    IppiSize roi = { _src.cols, _src.rows };

    ippiFilterSobelNegVertGetBufferSize_8u16s_C1R(roi, ippMskSize3x3,
&size);
    ippiFilterSobelHorizGetBufferSize_8u16s_C1R(roi, ippMskSize3x3,
&size1);
    size = std::max(size, size1);
    ippiCannyGetSize(roi, &size1);
    size = std::max(size, size1);

    AutoBuffer<uchar> buf(size + 64);
    uchar* buffer = alignPtr((uchar*)buf, 32);

    Mat _dx(_src.rows, _src.cols, CV_16S);
    if( ippiFilterSobelNegVertBorder_8u16s_C1R(_src.data, (int)_src.step,
                    _dx.ptr<short>(), (int)_dx.step, roi,
                    ippMskSize3x3, ippBorderRepl, 0, buffer) < 0 )
        return false;

    Mat _dy(_src.rows, _src.cols, CV_16S);
    if( ippiFilterSobelHorizBorder_8u16s_C1R(_src.data, (int)_src.step,
                    _dy.ptr<short>(), (int)_dy.step, roi,
                    ippMskSize3x3, ippBorderRepl, 0, buffer) < 0 )
        return false;

    if( ippiCanny_16s8u_C1R(_dx.ptr<short>(), (int)_dx.step,
                            _dy.ptr<short>(), (int)_dy.step,
                            _dst.data, (int)_dst.step, roi, low, high,
buffer) < 0 )
        return false;
    return true;
}
}
#endif

void cv::Canny( InputArray _src, OutputArray _dst,
                double low_thresh, double high_thresh,
                int aperture_size, bool L2gradient )
{
    Mat src = _src.getMat();            //输入图像，必须为单通道灰度图
```

```
    CV_Assert( src.depth() == CV_8U ); // 8位无符号

    _dst.create(src.size(), CV_8U);    //根据src的大小构造目标矩阵dst
    Mat dst = _dst.getMat();           //输出图像，为单通道黑白图


    // low_thresh 表示低阈值， high_thresh表示高阈值
    // aperture_size 表示算子大小，默认为3
    // L2gradient计算梯度幅值的标识，默认为false

    // 如果L2gradient为false 并且 apeture_size的值为-1（-1的二进制标识为：11111111）
    // L2gradient为false 则计算sobel导数时，用G = |Gx|+|Gy|
    // L2gradient为true  则计算sobel导数时，用G = Math.sqrt((Gx)^2 + (Gy)^2)

    if (!L2gradient && (aperture_size & CV_CANNY_L2_GRADIENT) ==
        CV_CANNY_L2_GRADIENT)
    {
        // CV_CANNY_L2_GRADIENT 宏定义其值为： Value = (1<<31) 1左移31位  即2147483648
        //backward compatibility
        // ~标识按位取反
        aperture_size &= ~CV_CANNY_L2_GRADIENT;//相当于取绝对值
        L2gradient = true;
    }


    // 判别条件1：aperture_size是奇数
    // 判别条件2: aperture_size的范围应当是[3,7]，默认值3
    if ((aperture_size & 1) == 0 || (aperture_size != -1 && (aperture_size < 3 || aperture_size > 7)))
        CV_Error(CV_StsBadFlag, "");  // 报错

    if (low_thresh > high_thresh)           // 如果低阈值 > 高阈值
        std::swap(low_thresh, high_thresh); // 则交换低阈值和高阈值

#ifdef HAVE_TEGRA_OPTIMIZATION
    if (tegra::canny(src, dst, low_thresh, high_thresh, aperture_size, L2gradient))
        return;
#endif

#ifdef USE_IPP_CANNY
    if( aperture_size == 3 && !L2gradient &&
        ippCanny(src, dst, (float)low_thresh, (float)high_thresh) )
        return;
#endif

    const int cn = src.channels();           // cn为输入图像的通道数
    Mat dx(src.rows, src.cols, CV_16SC(cn)); // 存储 x方向 方向导数的矩阵，CV_16SC(cn)：16位有符号cn通道
    Mat dy(src.rows, src.cols, CV_16SC(cn)); // 存储 y方向 方向导数的矩阵
......

    /*Sobel参数说明：(参考cvSobel)
      cvSobel(
            const  CvArr* src,                // 输入图像
            CvArr*        dst,                // 输入图像
```

```
            int           xorder,             // x方向求导的阶数
            int           yorder,             // y方向求导的阶数
            int           aperture_size = 3    // 滤波器的宽和高 必须是奇数
      );
    */

    // BORDER_REPLICATE 表示当卷积点在图像的边界时，原始图像边缘的像素会被复制，并用复制的像素扩展原始图的尺寸
    // 计算x方向的sobel方向导数，计算结果存在dx中
    Sobel(src, dx, CV_16S, 1, 0, aperture_size, 1, 0, cv::BORDER_REPLICATE);
    // 计算y方向的sobel方向导数，计算结果存在dy中
    Sobel(src, dy, CV_16S, 0, 1, aperture_size, 1, 0, cv::BORDER_REPLICATE);

    if (L2gradient)
    {
        low_thresh = std::min(32767.0, low_thresh);
        high_thresh = std::min(32767.0, high_thresh);

        if (low_thresh > 0) low_thresh *= low_thresh;    //低阈值平方运算
        if (high_thresh > 0) high_thresh *= high_thresh; //高阈值平方运算
    }

    int low = cvFloor(low_thresh);   // cvFloor返回不大于参数的最大整数值，相当于取整
    int high = cvFloor(high_thresh);

    // ptrdiff_t 是C/C++标准库中定义的一个数据类型，signed类型，常用于存储两个指针的差（距离），可以是负数
    // mapstep 用于存放
    ptrdiff_t mapstep = src.cols + 2; // +2 表示左右各扩展一条边
    // AutoBuffer<uchar> 会自动分配一定大小的内存，并且指定内存中的数据类型是uchar
    // 列数 +2 表示图像左右各自扩展一条边 （用于复制边缘像素，扩大原始图像）
    // 行数 +2 表示图像上下各自扩展一条边
    AutoBuffer<uchar> buffer((src.cols+2)*(src.rows+2) + cn * mapstep * 3 * sizeof(int));

    int* mag_buf[3];  //定义一个大小为3的int型指针数组，
    mag_buf[0] = (int*)(uchar*)buffer;
    mag_buf[1] = mag_buf[0] + mapstep*cn;
    mag_buf[2] = mag_buf[1] + mapstep*cn;
    memset(mag_buf[0], 0, /* cn* */mapstep*sizeof(int));

    uchar* map = (uchar*)(mag_buf[2] + mapstep*cn);
    memset(map, 1, mapstep);
    memset(map + mapstep*(src.rows + 1), 1, mapstep);

    int maxsize = std::max(1 << 10, src.cols * src.rows / 10); // 2的10次幂 1024
    std::vector<uchar*> stack(maxsize); // 定义指针类型向量，用于存地址
    uchar **stack_top = &stack[0];      // 栈顶指针（指向指针的指针），指向指针 stack[0]
    uchar **stack_bottom = &stack[0];   // 栈底指针 ，初始时 栈底指针 == 栈顶指针


    // 梯度的方向被近似到四个角度之一 (0, 45, 90, 135 四选一)
```

```
147
148     // define 定义函数块
149     // CANNY_PUSH(d) 是入栈函数， 参数d表示地址指针，让该指针指向的内容为2（int型强制转换成uchar型），并入栈，栈顶指针+1
150     // 2表示 像素属于某条边缘 可以看下方的注释
151     // CANNY_POP(d) 是出栈函数， 栈顶指针-1，然后将-1后的栈顶指针指向的值，赋给d
152     #define CANNY_PUSH(d)    *(d) = uchar(2), *stack_top++ = (d)
153     #define CANNY_POP(d)     (d) = *--stack_top
154
155     // calculate magnitude and angle of gradient, perform non-maxima suppression.
156     // fill the map with one of the following values:
157     // 0 - the pixel might belong to an edge 可能属于边缘
158     // 1 - the pixel can not belong to an edge 不属于边缘
159     // 2 - the pixel does belong to an edge 一定属于边缘
160
161     // for循环作用：进行非极大值抑制 + 滞后阈值（双阈值）处理
162     for (int i = 0; i <= src.rows; i++) // i 表示第i行
163     {
164
165         // i == 0 时，_norm 指向 mag_buf[1]
166         // i > 0 时， _norm 指向 mag_buf[2]
167         // +1 表示跳过每行的第一个元素，因为是后扩展的边，不可能是边缘
168         int* _norm = mag_buf[(i > 0) + 1] + 1;
169         if (i < src.rows)
170         {
171             short* _dx = dx.ptr<short>(i); // _dx指向dx矩阵的第i行
172             short* _dy = dy.ptr<short>(i); // _dy指向dy矩阵的第i行
173             // 如果 L2gradient为false
174             if (!L2gradient)
175             {
176               for (int j = 0; j < src.cols*cn; j++) // 对第i行里的每一个值都进行计算
177                   _norm[j] = std::abs(int(_dx[j])) + std::abs(int(_dy[j])); // ||+||
178             }
179             else
180             {
181               for (int j = 0; j < src.cols*cn; j++)
182               //用平方计算,L2gradient为 true时，高低阈值都被平方了，所以此处_norm[j]无需开方
183                    _norm[j] = int(_dx[j])*_dx[j] + int(_dy[j])*_dy[j];
184             }
185
186             // 如果不是单通道
187             if (cn > 1)
188             {
189                 for(int j = 0, jn = 0; j < src.cols; ++j, jn += cn)
190                 {
191                     int maxIdx = jn;
192                     for(int k = 1; k < cn; ++k)
193                         if(_norm[jn + k] > _norm[maxIdx]) maxIdx = jn + k;
194                     _norm[j] = _norm[maxIdx];
195                     _dx[j] = _dx[maxIdx];
196                     _dy[j] = _dy[maxIdx];
197                 }
198             }
```

```
            _norm[-1] = _norm[src.cols] = 0; // 最后一列和第一列的梯度幅值设置为0
        }
        // 当i == src.rows （最后一行）时，申请空间并且每个空间的值初始化为0，存储在mag_buf[2]
        else
            memset(_norm-1, 0, /* cn* */mapstep*sizeof(int));

        // at the very beginning we do not have a complete ring
        // buffer of 3 magnitude rows for non-maxima suppression
        if (i == 0)
            continue;

        uchar* _map = map + mapstep*i + 1; // _map 指向第 i+1 行，+1表示跳过该行第一个元素
        _map[-1] = _map[src.cols] = 1; // 第一列和最后一列不是边缘，所以设置为1
        int* _mag = mag_buf[1] + 1; // take the central row 中间那一行
        ptrdiff_t magstep1 = mag_buf[2] - mag_buf[1];
        ptrdiff_t magstep2 = mag_buf[0] - mag_buf[1];

        const short* _x = dx.ptr<short>(i-1);
        const short* _y = dy.ptr<short>(i-1);

        // 如果栈的大小不够，则重新为栈分配内存（相当于扩大容量）
        if ((stack_top - stack_bottom) + src.cols > maxsize)
        {
            int sz = (int)(stack_top - stack_bottom);
            maxsize = maxsize * 3/2;
            stack.resize(maxsize);
            stack_bottom = &stack[0];
            stack_top = stack_bottom + sz;
        }

        int prev_flag = 0; //前一个像素点 0：非边缘点 ；1：边缘点
        for (int j = 0; j < src.cols; j++) // 第 j 列
        {
          #define CANNY_SHIFT 15
          // tan22.5
          const int TG22 = (int)(0.4142135623730950488016887242097*(1<<CANNY_SHIFT) + 0.5);
            int m = _mag[j];
            if (m > low) // 如果大于低阈值
            {
                int xs = _x[j];    // dx中 第i-1行 第j列
                int ys = _y[j];    // dy中 第i-1行 第j列
                int x = std::abs(xs);
                int y = std::abs(ys) << CANNY_SHIFT;
                int tg22x = x * TG22;
                if (y < tg22x) //角度小于22.5 用区间表示：[0, 22.5)
                {
                // 与左右两点的梯度幅值比较，如果比左右都大
                //（此时当前点是左右邻域内的极大值），则 goto __ocv_canny_push 执行入栈操作
                    if (m > _mag[j-1] && m >= _mag[j+1]) goto __ocv_canny_push;
                }
                else //角度大于22.5
                {
```

```
                    int tg67x = tg22x + (x << (CANNY_SHIFT+1));
                    if (y > tg67x) //(67.5, 90)
                    {
                  //与上下两点的梯度幅值比较，如果比上下都大
                  //（此时当前点是左右邻域内的极大值），则 goto __ocv_canny_push 执行入栈操作
                        if (m > _mag[j+magstep2] && m >= _mag[j+magstep1])
                            goto __ocv_canny_push;
                    }
                    else //[22.5, 67.5]
                    {
                        // ^ 按位异或 如果xs与ys异号 则取-1 否则取1
                        int s = (xs ^ ys) < 0 ? -1 : 1;
                        //比较对角线邻域
                        if (m > _mag[j+magstep2-s] && m > _mag[j+magstep1+s])
                            goto __ocv_canny_push;
                    }
                }
            }

            //比当前的梯度幅值低阈值还低，直接被确定为非边缘
            prev_flag = 0;
            _map[j] = uchar(1); // 1 表示不属于边缘

            continue;
__ocv_canny_push:
        // 前一个点非边缘点且当前点的幅值大于高阈值（大于高阈值被视为边缘像素）且正上方的点非边缘点
            if (!prev_flag && m > high && _map[j-mapstep] != 2)
            {
               //将当前点的地址入栈，入栈前，将该点地址指向的值设置为2（查看上面的宏定义函数块里）
                CANNY_PUSH(_map + j);
                prev_flag = 1;
            }
            else
                _map[j] = 0;
        }

        // scroll the ring buffer
        // 交换指针指向的位置，向上覆盖，把mag_[1]的内容覆盖到mag_buf[0]上
        // 把mag_[2]的内容覆盖到mag_buf[1]上
        // 最后 让mag_buf[2]指向_mag指向的那一行
        _mag = mag_buf[0];
        mag_buf[0] = mag_buf[1];
        mag_buf[1] = mag_buf[2];
        mag_buf[2] = _mag;
    }


    // 通过上面的for循环，确定了各个邻域内的极大值点为边缘点（标记为2）
    // 现在，在这些边缘点的8邻域内（上下左右+4个对角）,将可能的边缘点（标记为0）确定为边缘
    while (stack_top > stack_bottom)
    {
        uchar* m;
        if ((stack_top - stack_bottom) + 8 > maxsize)
```

```
        {
            int sz = (int)(stack_top - stack_bottom);
            maxsize = maxsize * 3/2;
            stack.resize(maxsize);
            stack_bottom = &stack[0];
            stack_top = stack_bottom + sz;
        }

        CANNY_POP(m); // 出栈

        if (!m[-1])         CANNY_PUSH(m - 1);
        if (!m[1])          CANNY_PUSH(m + 1);
        if (!m[-mapstep-1]) CANNY_PUSH(m - mapstep - 1);
        if (!m[-mapstep])   CANNY_PUSH(m - mapstep);
        if (!m[-mapstep+1]) CANNY_PUSH(m - mapstep + 1);
        if (!m[mapstep-1])  CANNY_PUSH(m + mapstep - 1);
        if (!m[mapstep])    CANNY_PUSH(m + mapstep);
        if (!m[mapstep+1])  CANNY_PUSH(m + mapstep + 1);
    }

    // 最终：生成边缘图像
    const uchar* pmap = map + mapstep + 1;
    uchar* pdst = dst.ptr();
    for (int i = 0; i < src.rows; i++, pmap += mapstep, pdst += dst.step)
    {
        for (int j = 0; j < src.cols; j++)
            pdst[j] = (uchar)-(pmap[j] >> 1);
    }
}

void cvCanny( const CvArr* image, CvArr* edges, double threshold1,
              double threshold2, int aperture_size )
{
    cv::Mat src = cv::cvarrToMat(image), dst = cv::cvarrToMat(edges);
    CV_Assert( src.size == dst.size && src.depth() == CV_8U && dst.type()
== CV_8U );

    cv::Canny(src, dst, threshold1, threshold2, aperture_size & 255,
              (aperture_size & CV_CANNY_L2_GRADIENT) != 0);
}
```

四、opencv源码：形态学腐蚀和膨胀

1. 算法原理

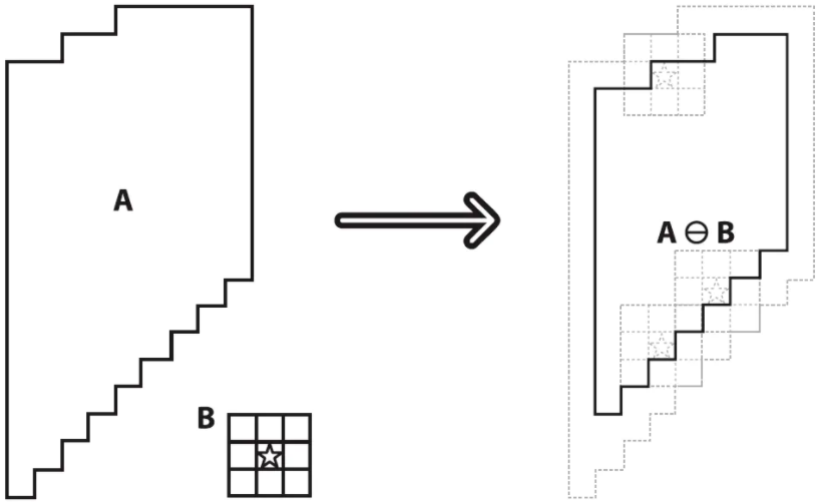
- 腐蚀的原理

  二值图像前景物体为1，背景为0。假设原图像中有一个前景物体，那么我们用一个结构元素去腐蚀原图的过程是这样的：遍历原图像的每一个像素，然后用结构元素的中心点对准当前正在遍历的这个像素，然后取当前结构元素所覆盖下的原图对应区域内的所有像素的最小值，用这个最小值替换当前像素值。由于二值图像最小值就是0，所以就是用0替换，即变成了黑色背景。从而也可以

看出，如果当前结构元素覆盖下，全部都是背景，那么就不会对原图做出改动，因为都是0.如果全部都是前景像素，也不会对原图做出改动，因为都是1.只有结构元素位于前景物体边缘的时候，它覆盖的区域内才会出现0和1两种不同的像素值，这个时候把当前像素替换成0就有变化了。因此腐蚀看起来的效果就是让前景物体缩小了一圈一样。对于前景物体中一些细小的连接处，如果结构元素大小相等，这些连接处就会被断开。

腐蚀的示例图如下所示：

从左到右依次是，原图、结构元、腐蚀后的图像。可以明显发现腐蚀后的图像比原图小了一圈。

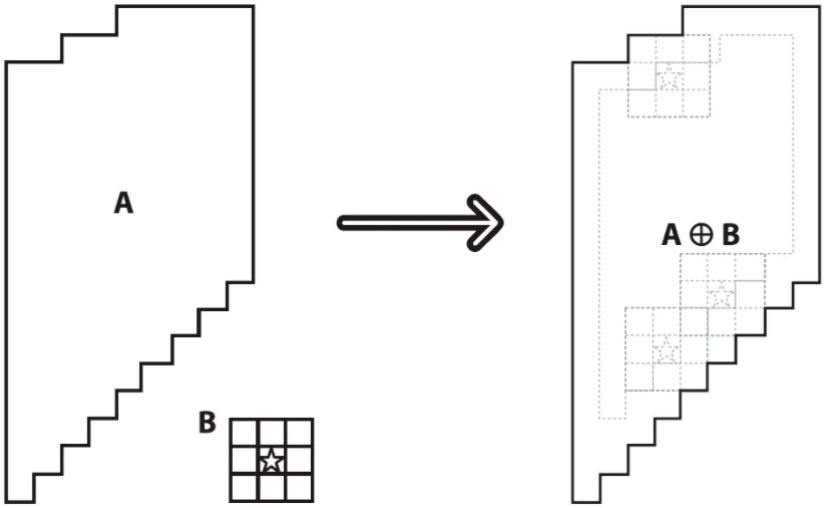


- 膨胀的原理

二值图像前景物体为1，背景为0。假设原图像中有一个前景物体，那么我们用一个结构元素去膨胀原图的过程是这样的：遍历原图像的每一个像素，然后用结构元素的中心点对准当前正在遍历的这个像素，然后取当前结构元素所覆盖下的原图对应区域内的所有像素的最大值，用这个最大值替换当前像素值。由于二值图像最大值就是1，所以就是用1替换，即变成了白色前景物体。从而也可以看出，如果当前结构元素覆盖下，全部都是背景，那么就不会对原图做出改动，因为都是0。如果全部都是前景像素，也不会对原图做出改动，因为都是1。只有结构元素位于前景物体边缘的时候，它覆盖的区域内才会出现0和1两种不同的像素值，这个时候把当前像素替换成1就有变化了。因此膨胀看起来的效果就是让前景物体胀大了一圈一样。对于前景物体中一些细小的断裂处，如果结构元素大小相等，这些断裂的地方就会被连接起来。

膨胀的示例图如下所示：

从左到右依次是，原图、结构元、膨胀后的图像。可以明显发现膨胀后的图像比原图大了一圈。



2. API介绍

腐蚀函数cvErode：

```
void cvErode( const CvArr* src, CvArr* dst, IplConvKernel* element=NULL, int
iterations=1 )
```

函数参数：

src： 输入图像.
dst： 输出图像.
element： 用于腐蚀的结构元素。若为 NULL, 则使用 3×3 长方形的结构元素
iterations： 腐蚀的次数。 腐蚀可以重复进行 (iterations) 次. 对彩色图像，每个彩色通道单独处理。

膨胀函数cvDilate：

```
void cvDilate( const CvArr* src, CvArr* dst, IplConvKernel* element=NULL, int
iterations=1 )
```

函数参数：

src： 输入图像.
dst： 输出图像.
element： 用于膨胀的结构元素。若为 NULL, 则使用 3×3 长方形的结构元素
iterations： 膨胀的次数。 膨胀可以重复进行 (iterations) 次. 对彩色图像，每个彩色通道单独处理。

3. 源码分析

opencv源码： `cvErode`和`cvDilate`

源码说明：

这两个函数无论是从参数还是对图像的处理方式，都高度类似，其本质原因应该是腐蚀和膨胀操作原理的高度类似性。这两个的区别只在于最后分别调用了`cv::erode()`和`cv::dilate()`函数。

源码注释：

```
CV_IMPL void
cvErode( const CvArr* srcarr, CvArr* dstarr, IplConvKernel* element,int
iterations)
{
    cv::Mat src = cv::cvarrToMat(srcarr), dst = cv::cvarrToMat(dstarr),
kernel;
    // 输入输出必须是同等尺寸、同类型的
    CV_Assert(src.size()== dst.size()&& src.type()== dst.type());
    cv::Point anchor;
    // 若没有结构元素输入，kernel=NULL，anchor=(1,1)
    // 否则将结构元素中的值读入kernel和anchor
    convertConvKernel(element, kernel,anchor );
    // 边界差值方法采用边界复制
    cv::erode( src, dst, kernel, anchor, iterations,cv::BORDER_REPLICATE);
}
```

```
CV_IMPL void
cvDilate( const CvArr* srcarr, CvArr* dstarr, IplConvKernel* element,int
iterations)
{
    cv::Mat src = cv::cvarrToMat(srcarr), dst = cv::cvarrToMat(dstarr),
kernel;
    // 输入输出必须是同等尺寸、同类型的
    CV_Assert(src.size()== dst.size()&& src.type()== dst.type());
    cv::Point anchor;
    // 若没有结构元素输入，kernel=NULL，anchor=(1,1)
    // 否则将结构元素中的值读入kernel和anchor
    convertConvKernel(element, kernel,anchor );
    // 边界差值方法采用边界复制
    cv::dilate( src, dst, kernel, anchor, iterations,cv::BORDER_REPLICATE);
}
```

opencv源码：腐蚀函数`cv::erode`、膨胀函数`cv::dilate`

源码说明：

`cv::erode`和`cv::dilate`这两个函数在内部其实是调用`morphOp`，只是调用`morphOp`时，第一个参数标识符不同，一个为MORPH_ERODE（腐蚀），一个为MORPH_DILATE（膨胀）。

函数的第三个参数：InputArray类型的kernel，腐蚀和膨胀操作的核。若为NULL时，表示的是使用参考点位于中心3x3的核。我们一般使用函数`getStructuringElement`返回指定形状和尺寸的 **结构元

源码注释：

```
void cv::erode( InputArray src, OutputArray dst, InputArray kernel,
                Point anchor, int iterations,
                int borderType, const Scalar& borderValue )
{
    //调用morphOp函数，并设定标识符为MORPH_ERODE
    morphOp( MORPH_ERODE, src, dst, kernel, anchor, iterations, borderType,
borderValue );
}
```

```
void cv::dilate( InputArray src, OutputArray dst, InputArray kernel,
                 Point anchor, int iterations,
                 int borderType, const Scalar& borderValue )
{
    //调用morphOp函数，并设定标识符为MORPH_DILATE
    morphOp( MORPH_DILATE, src, dst, kernel, anchor, iterations, borderType,
borderValue );
}
```

opencv源码：`getStructuringElement`

源码说明：

函数`getStructuringElement`的作用是获取形态学腐蚀和膨胀所需的结构元，在函数`cv::erode`和`cv::dilate`里面被调用。

源码注释：

```
Mat getStructuringElement(int shape, Size ksize, Point anchor =
Point(-1,-1));
# 函数作用：获取腐蚀或膨胀的结构元

# 函数参数：
# 第一个参数表示内核的形状，我们可以选择如下三种形状之一
            # 矩形：MORPH_RECT
            # 交叉形：MORPH_CROSS
            # 椭圆形：MORPH_ELLIPSE

# 第二和第三个参数分别是内核的尺寸以及锚点的位置。
            # 一般在调erode和dilate函数前，先定义一个Mat类型的变量来
            # 获得getStructuringElement函数的返回值。
            # 对于锚点的位置，有默认值Point(-1,-1)，表示锚点位于中心。
            # 且需要注意，交叉形的element形状唯一依赖于锚点的位置。
            # 而在其他情况下，锚点只是影响了形态学运算结果的偏移。

# 第四个参数，Point类型的anchor，锚的位置，默认值（-1，-1），表示锚位于中心。
# 第五个参数，int类型的iterations，迭代使用erode（）函数的次数，默认值为1。
# 第六个参数，int类型的borderType，用于推断图像外部像素的某种边界模式。有默认值
BORDER_DEFAULT。
# 第七个参数，const Scalar&类型的borderValue，当边界为常数时的边界值，有默认值
morphologyDefaultBorderValue()
```

opencv源码：重要函数`morphOp`

源码说明：

`cv::erode`和`cv::dilate`这两个函数在内部其实是调用`morphOp`，只是调用`morphOp`时，第一个参数标识符不同，一个为MORPH_ERODE（腐蚀），一个为MORPH_DILATE（膨胀）。morphop根据参数op的不同分别指向腐蚀和膨胀。

源码注释：

```
static void morphOp( int op, InputArray _src, OutputArray_dst,
                     InputArray_kernel,
                     Pointanchor, int iterations,
                     intborderType, constScalar& borderValue)
{
    Mat src = _src.getMat(), kernel= _kernel.getMat();
    // 如果输入的时候不输入kernel，则kernel.data=NULL，那么ksize=(3,3)
    Size ksize = kernel.data ? kernel.size() : Size(3,3);
    // ifanchor.x=-1,anchor.x=ksize.width/2; if
anchor.y=-1,anchor.y=ksize.height/2
    // 并判断是否在rect(0, 0, ksize.width, ksize.height)内
    anchor =normalizeAnchor(anchor,ksize);
    // 这一句是多余的，因为在上面normalizeAnchor已经判断了
    CV_Assert(anchor.inside(Rect(0, 0, ksize.width, ksize.height)) );
    _dst.create( src.size(), src.type() );
    Mat dst = _dst.getMat();
    // 如果迭代步数为或者结构元素的尺寸为下面条件的值，不进行处理，直接输出
    if( iterations == 0 || kernel.rows*kernel.cols == 1 )
    {
```

```
19          src.copyTo(dst);
20          return;
21      }
22      // 如果没有输入结构元素，那么创建(1+iterations*2)*(1+iterations*2)的长方形结构
    元素
23      // 结构元素的中心点为(iterations, iterations)，并将迭代步数设置为
24      if( !kernel.data )
25      {
26          kernel= getStructuringElement(MORPH_RECT,
    Size(1+iterations*2,1+iterations*2));
27          anchor= Point(iterations,iterations);
28          iterations= 1;
29      }
30      // 如果结构步数大于的话并且kernel为长方形的结构元素，重新创建结构元素
31      else if( iterations> 1 && countNonZero(kernel) ==
    kernel.rows*kernel.cols )
32      {
33          anchor= Point(anchor.x*iterations, anchor.y*iterations);
34          kernel= getStructuringElement(MORPH_RECT,
35                      Size(ksize.width + (iterations-1)*(ksize.width-1),
36                          ksize.height +(iterations-1)*(ksize.height-1)),
37                              anchor);
38          iterations= 1;
39      }
40      int nStripes = 1;
41
42      // Tegra是NVIDIA公司于2008年推出的基于ARM构架通用处理器品牌（即CPU，NVIDIA称
    为“Computer on          a chip”片上计算机），能够为便携设备提供高性能、低功耗体验。
43  #if definedHAVE_TEGRA_OPTIMIZATION
44      if (src.data != dst.data &&iterations == 1 &&  /
45         (borderType & BORDER_ISOLATED) == 0&& //检查边界类型
46         src.rows >= 64 ) //NOTE: justheuristics
47         nStripes = 4;
48
49  #endif
50
51      //并行处理
52      //op由腐蚀函数cv::erode和膨胀函数cv::dilate传入：
53          //op=MORPH_ERODE（腐蚀）、op=MORPH_DILATE（膨胀）
54      parallel_for_(Range(0, nStripes),
55                    MorphologyRunner(src, dst, nStripes, iterations,op,
    kernel,anchor,    borderType,borderType, borderValue));
56  }
```