目录

- 10.3 CalcEMD2

梯度、边缘和角点

Sobel

使用扩展 Sobel 算子计算一阶、二阶、三阶或混合图像差分

void cvSobel(const CvArr* src, CvArr* dst, int xorder, int yorder, int aperture_size=3);

src
　　输入图像.

dst
　　输出图像.

xorder
　　x 方向上的差分阶数

yorder
　　y 方向上的差分阶数

aperture_size
　　扩展 Sobel 核的大小，必须是 1, 3, 5 或 7。 除了尺寸为 1， 其它情况下， aperture_size ×aperture_size 可分离内核将用来计算差分。对 aperture_size=1 的情况， 使用 3x1 或 1x3 内核 （不进行高斯平滑操作）。这里有一个特殊变量 CV_SCHARR (=-1)，对应 3x3 Scharr 滤波器，可以给出比 3x3 Sobel 滤波更精确的结果。Scharr 滤波器系数是：

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

对 x-方向 或矩阵转置后对 y-方向。

函数 cvSobel 通过对图像用相应的内核进行卷积操作来计算图像差分：

$$dst(x,y) = \frac{d^{xorder+yorder} src}{dx^{xorder} dy^{yorder}}|(x,y)$$

由于 Sobel 算子结合了 Gaussian 平滑和微分，所以，其结果或多或少对噪声有一定的鲁棒性。通常情况，函数调用采用如下参数 (xorder=1, yorder=0, aperture_size=3) 或 (xorder=0, yorder=1, aperture_size=3) 来计算一阶 x- 或 y- 方向的图像差分。第一种情况对应：

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

第二种对应：

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

或者

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

核的选则依赖于图像原点的定义 (origin 来自 IplImage 结构的定义)。由于该函数不进行图像尺度变换，所以和输入图像(数组)相比，输出图像(数组)的元素通常具有更大的绝对数值（译者注：即像素的位深）。为防止溢出，当输入图像是 8 位的，要求输出图像是 16 位的。当然可以用函数 cvConvertScale 或 cvConvertScaleAbs 转换为 8 位的。除了 8-位 图像，函数也接受 32-位 浮点数图像。所有输入和输出图像都必须是单通道的，并且具有相同的图像尺寸或者 ROI 尺寸。

Laplace

计算图像的 Laplacian 变换

```
void cvLaplace( const CvArr* src, CvArr* dst, int aperture_size=3 );
```

src
　　输入图像.

dst
　　输出图像.

aperture_size
　　核大小 (与 cvSobel 中定义一样).

函数 cvLaplace 计算输入图像的 Laplacian 变换，方法是先用 sobel 算子计算二阶 x- 和 y- 差分，再求和:

$$dst(x,y) = \frac{d^2 src}{dx^2} + \frac{d^2 src}{dy^2}$$

对 aperture_size=1 则给出最快计算结果，相当于对图像采用如下内核做卷积：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

类似于 cvSobel 函数，该函数也不作图像的尺度变换，所支持的输入、输出图像类型的组合和 cvSobel 一致。

Canny

采用 Canny 算法做边缘检测

```
void cvCanny( const CvArr* image, CvArr* edges, double threshold1,
              double threshold2, int aperture_size=3 );
```

image
　　单通道输入图像.
edges
　　单通道存储边缘的输出图像
threshold1
　　第一个阈值
threshold2
　　第二个阈值
aperture_size
　　Sobel 算子内核大小 (见 cvSobel).

函数 cvCanny 采用 CANNY 算法发现输入图像的边缘而且在输出图像中标识这些边缘。threshold1 和 threshold2 当中的小阈值用来控制边缘连接，大的阈值用来控制强边缘的初始分割。

- 注意事项：cvCanny 只接受单通道图像作为输入。
- 外部链接：经典的 canny 自调整阈值算法的一个 opencv 的实现见在 OpenCV 中自适应确定 canny 算法的分割门限

PreCornerDetect

计算用于角点检测的特征图，

```
void cvPreCornerDetect( const CvArr* image, CvArr* corners, int
aperture_size=3 );
```

image
　　输入图像.

corners
保存候选角点的特征图
aperture_size
Sobel 算子的核大小(见 cvSobel).

函数 cvPreCornerDetect 计算函数

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}$$

其中 D 表示一阶图像差分，　　表示二阶图像差分。 角点被认为是函数的局部最大值：

```
// 假设图像格式为浮点数
IplImage* corners = cvCloneImage(image);
IplImage* dilated_corners = cvCloneImage(image);
IplImage* corner_mask = cvCreateImage( cvGetSize(image), 8, 1 );
cvPreCornerDetect( image, corners, 3 );
cvDilate( corners, dilated_corners, 0, 1 );
cvSubS( corners, dilated_corners, corners );
cvCmpS( corners, 0, corner_mask, CV_CMP_GE );
cvReleaseImage( &corners );
cvReleaseImage( &dilated_corners );
```

CornerEigenValsAndVecs

计算图像块的特征值和特征向量，用于角点检测

```
void cvCornerEigenValsAndVecs( const CvArr* image, CvArr* eigenvv,
                               int block_size, int aperture_size=3 );
```

image
输入图像.
eigenvv
保存结果的数组。必须比输入图像宽 6 倍。
block_size
邻域大小 (见讨论).
aperture_size
Sobel 算子的核尺寸(见 cvSobel).

对每个象素，函数 cvCornerEigenValsAndVecs 考虑 block_size × block_size 大小的邻域 S(p)，然后在邻域上计算图像差分的相关矩阵：

$$M = \begin{bmatrix} \sum_{S(p)}(\frac{dI}{dx})^2 & \sum_{S(p)}(\frac{dI}{dx} \cdot \frac{dI}{dy})^2 \\ \sum_{S(p)}(\frac{dI}{dx} \cdot \frac{dI}{dy})^2 & \sum_{S(p)}(\frac{dI}{dy})^2 \end{bmatrix}$$

然后它计算矩阵的特征值和特征向量，并且按如下方式(λ1, λ2, x1, y1, x2, y2)存储这些值到输出图像中，其中

λ1, λ2 - M 的特征值，没有排序
(x1, y1) - 特征向量，对 λ1
(x2, y2) - 特征向量，对 λ2

CornerMinEigenVal

计算梯度矩阵的最小特征值，用于角点检测

```
void cvCornerMinEigenVal( const CvArr* image,
CvArr* eigenval, int block_size, int aperture_size=3 );
```

image
输入图像.

eigenval
保存最小特征值的图像. 与输入图像大小一致

block_size
邻域大小 (见讨论 cvCornerEigenValsAndVecs).

aperture_size
Sobel 算子的核尺寸(见 cvSobel). 当输入图像是浮点数格式时，该参数表示用来计算差分固定的浮点滤波器的个数.

函数 cvCornerMinEigenVal 与 cvCornerEigenValsAndVecs 类似，但是它仅仅计算和存储每个象素点差分相关矩阵的最小特征值，即前一个函数的 min(λ1, λ2)

CornerHarris

哈里斯（Harris）角点检测

```
void cvCornerHarris( const CvArr* image, CvArr* harris_responce,
int block_size, int aperture_size=3, double k=0.04 );
```

image
输入图像。

harris_responce
存储哈里斯（Harris）检测 responces 的图像。与输入图像等大。

block_size

邻域大小（见关于 cvCornerEigenValsAndVecs 的讨论）。

aperture_size

扩展 Sobel 核的大小（见 cvSobel）。格式. 当输入图像是浮点数格式时，该参数表示用来计算差分固定的浮点滤波器的个数。

k

harris 检测器的自由参数。参见下面的公式。

函数 cvCornerHarris 对输入图像进行 Harris 边界检测。类似于 cvCornerMinEigenVal 和 cvCornerEigenValsAndVecs。对每个像素，在 block_size*block_size 大小的邻域上，计算其 2*2 梯度共变矩阵（或相关异变矩阵）M。然后，将

det(M) - k*trace(M)2 （这里 2 是平方）

保存到输出图像中。输入图像中的角点在输出图像中由局部最大值表示。

FindCornerSubPix

精确角点位置

```
void cvFindCornerSubPix( const CvArr* image, CvPoint2D32f* corners,
                         int count, CvSize win, CvSize zero_zone,
                         CvTermCriteria criteria );
```

image

输入图像.

corners

输入角点的初始坐标，也存储精确的输出坐标

count

角点数目

win

搜索窗口的一半尺寸。如果 win=(5,5) 那么使用 5*2+1 × 5*2+1 = 11 × 11 大小的搜索窗口

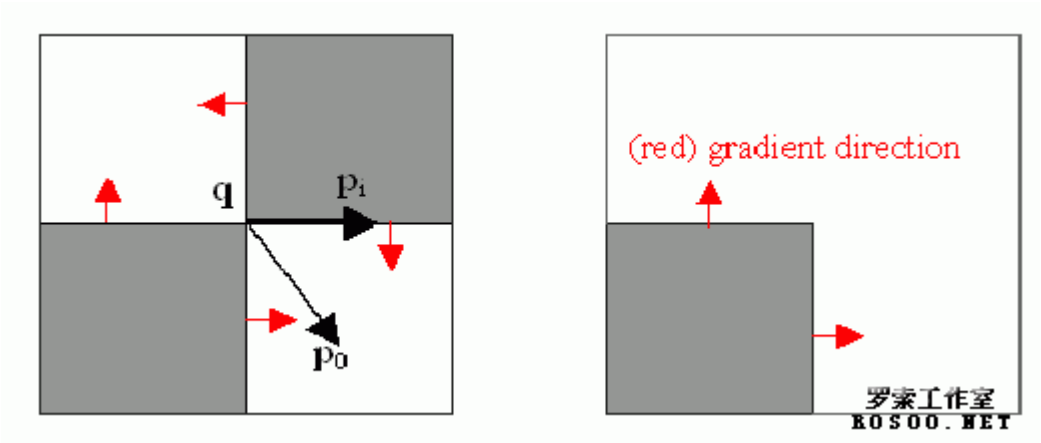
zero_zone

死区的一半尺寸，死区为不对搜索区的中央位置做求和运算的区域。它是用来避免自相关矩阵出现的某些可能的奇异性。当值为 (-1,-1) 表示没有死区。

criteria

求角点的迭代过程的终止条件。即角点位置的确定，要么迭代数大于某个设定值，或者是精确度达到某个设定值。criteria 可以是最大迭代数目，或者是设定的精确度，也可以是它们的组合。

函数 cvFindCornerSubPix 通过迭代来发现具有子象素精度的角点位置，或如图所示的放射鞍点（radial saddle points）。



子象素级角点定位的实现是基于对向量正交性的观测而实现的，即从中央点 q 到其邻域点 p 的向量和 p 点处的图像梯度正交（服从图像和测量噪声）。考虑以下的表达式：

$\varepsilon_i = DIpi^T \cdot (q - pi)$

其中，DIpi 表示在 q 的一个邻域点 pi 处的图像梯度，q 的值通过最小化 ε_i 得到。通过将 ε_i 设为 0，可以建立系统方程如下：

$$\sum_i (DIpi \cdot DIpi^T) \cdot q - \sum_i (DIpi \cdot DIpi^T \cdot pi) = 0$$

其中 q 的邻域（搜索窗）中的梯度被累加。调用第一个梯度参数 G 和第二个梯度参数 b，得到：

$$q = G^{-1} \cdot b$$

该算法将搜索窗的中心设为新的中心 q，然后迭代，直到找到低于某个阈值点的中心位置。

GoodFeaturesToTrack

确定图像的强角点

```
void cvGoodFeaturesToTrack( const CvArr* image, CvArr* eig_image, CvArr* temp_image,
                            CvPoint2D32f* corners, int* corner_count,
                            double quality_level, double min_distance,
                            const CvArr* mask=NULL );
```

image

输入图像，8-位或浮点 32-比特，单通道

eig_image

临时浮点 32-位图像，尺寸与输入图像一致

temp_image
另外一个临时图像，格式与尺寸与 eig_image 一致
corners
输出参数，检测到的角点
corner_count
输出参数，检测到的角点数目
quality_level
最大最小特征值的乘法因子。定义可接受图像角点的最小质量因子。
min_distance
限制因子。得到的角点的最小距离。使用 Euclidian 距离
mask
ROI:感兴趣区域。函数在 ROI 中计算角点，如果 mask 为 NULL，则选择整个图像。 必须为单通道的灰度图，大小与输入图像相同。mask 对应的点不为 0，表示计算该点。

函数 cvGoodFeaturesToTrack 在图像中寻找具有大特征值的角点。该函数，首先用 cvCornerMinEigenVal 计算输入图像的每一个象素点的最小特征值，并将结果存储到变量 eig_image 中。然后进行非最大值抑制（仅保留 3x3 邻域中的局部最大值）。下一步将最小特征值小于 quality_level•max(eig_image(x,y)) 排除掉。最后，函数确保所有发现的角点之间具有足够的距离，（最强的角点第一个保留，然后检查新的角点与已有角点之间的距离大于 min_distance ）。

采样、插值和几何变换

InitLineIterator

初始化线段迭代器

```
int cvInitLineIterator( const CvArr* image, CvPoint pt1, CvPoint pt2,
                        CvLineIterator* line_iterator, int
connectivity=8 );
```

image
带采线段的输入图像.
pt1
线段起始点
pt2
线段结束点
line_iterator
指向线段迭代器状态结构的指针
connectivity
被扫描线段的连通数，4 或 8.

函数 cvInitLineIterator 初始化线段迭代器，并返回两点之间的象素点数目。两个点必须在图像内。当迭代器初始化后，连接两点的光栅线上所有点，都可以连续通过调用 CV_NEXT_LINE_POINT 来得到。线段上的点是使用 4－连通或 8－连通利用 Bresenham 算法逐点计算的。

例子：使用线段迭代器计算彩色线上象素值的和

```
CvScalar sum_line_pixels( IplImage* image, CvPoint pt1, CvPoint pt2 )
{
    CvLineIterator iterator;
    int blue_sum = 0, green_sum = 0, red_sum = 0;
    int count = cvInitLineIterator( image, pt1, pt2, &iterator, 8 );

    for( int i = 0; i < count; i++ ){
        blue_sum += iterator.ptr[0];
        green_sum += iterator.ptr[1];
        red_sum += iterator.ptr[2];
        CV_NEXT_LINE_POINT(iterator);

        /* print the pixel coordinates: demonstrates how to calculate the
coordinates */
        {
        int offset, x, y;
        /* assume that ROI is not set, otherwise need to take it into
account. */
        offset = iterator.ptr - (uchar*)(image->imageData);
        y = offset/image->widthStep;
        x = (offset - y*image->widthStep)/(3*sizeof(uchar) /* size of
pixel */);
        printf("(%d,%d)\n", x, y );
        }
    }
    return cvScalar( blue_sum, green_sum, red_sum );
}
```

SampleLine

将图像上某一光栅线上的像素数据读入缓冲区

```
int cvSampleLine( const CvArr* image, CvPoint pt1, CvPoint pt2,
                  void* buffer, int connectivity=8 );
```

image
　　输入图像

pt1
光栅线段的起点
pt2
光栅线段的终点
buffer
存储线段点的缓存区，必须有足够大小来存储点 max(|pt2.x-pt1.x|+1, |pt2.y-pt1.y|+1) ：8－连通情况下，或者
|pt2.x-pt1.x|+|pt2.y-pt1.y|+1 ： 4－连通情况下.
connectivity
线段的连通方式, 4 or 8.

函数 cvSampleLine 实现了线段迭代器的一个特殊应用。它读取由 pt1 和 pt2 两点确定的线段上的所有图像点，包括终点，并存储到缓存中。

GetRectSubPix

从图像中提取象素矩形，使用子象素精度

```
void cvGetRectSubPix( const CvArr* src, CvArr* dst, CvPoint2D32f
center );
```

src
输入图像.
dst
提取的矩形.
center
提取的象素矩形的中心，浮点数坐标。中心必须位于图像内部.

函数 cvGetRectSubPix 从图像 src 中提取矩形:

```
dst(x, y) = src(x + center.x - (width(dst)-1)*0.5, y + center.y -
(height(dst)-1)*0.5)
```

其中非整数象素点坐标采用双线性插值提取。对多通道图像，每个通道独立单独完成提取。尽管函数要求矩形的中心一定要在输入图像之中，但是有可能出现矩形的一部分超出图像边界的情况，这时，该函数复制边界的模识（hunnish:即用于矩形相交的图像边界线段的象素来代替矩形超越部分的象素）。

GetQuadrangleSubPix

提取象素四边形，使用子象素精度

```
void cvGetQuadrangleSubPix( const CvArr* src, CvArr* dst, const CvMat*
map_matrix );
```

src
　　输入图像.
dst
　　提取的四边形.
map_matrix
　　3 × 2 变换矩阵 [A|b] （见讨论）.

函数 cvGetQuadrangleSubPix 以子象素精度从图像 src 中提取四边形,使用子象素精度，并且将结果存储于 dst ,计算公式是：

$dst(x + width(dst) / 2, y + height(dst) / 2) = src(A_{11}x + A_{12}y + b_1, A21x + A22y + b_2)$

其中 A 和 b 均来自映射矩阵(译者注：A, b 为几何形变参数) ，映射矩阵为：

$$map\_matrix = \begin{bmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{bmatrix}$$

其中在非整数坐标 $A \cdot (x,y)^T + b$ 的象素点值通过双线性变换得到。当函数需要图像边界外的像素点时，使用重复边界模式（replication border mode）恢复出所需的值。多通道图像的每一个通道都单独计算。

例子：使用 cvGetQuadrangleSubPix 进行图像旋转

```
#include "cv.h"
#include "highgui.h"
#include "math.h"

int main( int argc, char** argv )
{
    IplImage* src;
    /* the first command line parameter must be image file name */
    if( argc==2 && (src = cvLoadImage(argv[1], -1))!=0)
    {
        IplImage* dst = cvCloneImage( src );
        int delta = 1;
        int angle = 0;

        cvNamedWindow( "src", 1 );
        cvShowImage( "src", src );

        for(;;)
        {
```

```
            float m[6];
            double factor = (cos(angle*CV_PI/180.) + 1.1)*3;
            CvMat M = cvMat( 2, 3, CV_32F, m );
            int w = src->width;
            int h = src->height;

            m[0] = (float)(factor*cos(-angle*2*CV_PI/180.));
            m[1] = (float)(factor*sin(-angle*2*CV_PI/180.));
            m[2] = w*0.5f;
            m[3] = -m[1];
            m[4] = m[0];
            m[5] = h*0.5f;

            cvGetQuadrangleSubPix( src, dst, &M, 1, cvScalarAll(0));

            cvNamedWindow( "dst", 1 );
            cvShowImage( "dst", dst );

            if( cvWaitKey(5) == 27 )
                break;

            angle = (angle + delta) % 360;
        }
    }
    return 0;
}
```

Resize

图像大小变换

```
void cvResize( const CvArr* src, CvArr* dst, int
interpolation=CV_INTER_LINEAR );
```

src

输入图像.

dst

输出图像.

interpolation

插值方法:

- CV_INTER_NN - 最近邻插值,
- CV_INTER_LINEAR - 双线性插值 (缺省使用)
- CV_INTER_AREA - 使用象素关系重采样。当图像缩小时候，该方法可以避免波纹出现。当图像放大时，类似于 CV_INTER_NN 方法..

- CV_INTER_CUBIC - 立方插值.

函数 cvResize 将图像 src 改变尺寸得到与 dst 同样大小。若设定 ROI，函数将按常规支持 ROI.

WarpAffine

对图像做仿射变换

```
void cvWarpAffine( const CvArr* src, CvArr* dst, const CvMat* map_matrix,
                   int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
                   CvScalar fillval=cvScalarAll(0) );
```

src
　　输入图像.

dst
　　输出图像.

map_matrix
　　2×3 变换矩阵

flags
　　插值方法和以下开关选项的组合：

- CV_WARP_FILL_OUTLIERS - 填充所有输出图像的象素。如果部分象素落在输入图像的边界外，那么它们的值设定为 fillval.
- CV_WARP_INVERSE_MAP - 指定 map_matrix 是输出图像到输入图像的反变换，因此可以直接用来做象素插值。否则, 函数从 map_matrix 得到反变换。

fillval
　　用来填充边界外面的值

函数 cvWarpAffine 利用下面指定的矩阵变换输入图像：

$$dst(x',y') \leftarrow src(x,y)$$

- 如果没有指定 CV_WARP_INVERSE_MAP ，
  $$(x',y')^T = map\_matrix \cdot (x,y,1)^T$$
- 否则，
  $$(x,y)^T = map\_matrix \cdot (x',y',1)^T$$

函数与 cvGetQuadrangleSubPix 类似，但是不完全相同。 cvWarpAffine 要求输入和输出图像具有同样的数据类型，有更大的资源开销（因此对小图像不太合

适）而且输出图像的部分可以保留不变。而 cvGetQuadrangleSubPix 可以精确地从 8 位图像中提取四边形到浮点数缓存区中，具有比较小的系统开销，而且总是全部改变输出图像的内容。

要变换稀疏矩阵，使用 cxcore 中的函数 cvTransform 。

GetAffineTransform

由三对点计算仿射变换

```
CvMat* cvGetAffineTransform( const CvPoint2D32f* src,
 const CvPoint2D32f* dst, CvMat* map_matrix );
```

src
　　输入图像的三角形顶点坐标。
dst
　　输出图像的相应的三角形顶点坐标。
map_matrix
　　指向 2×3 输出矩阵的指针。

函数 cvGetAffineTransform 计算满足以下关系的仿射变换矩阵：

$$(x_i', y_i', 1)^T = map\_matrix \cdot (x_i, y_i, 1)^T$$

　　这里, $dst(i) = (x'_i, y'_i)$, $src(i) = (x_i, y_i)$, $i = 0..2$.

2DRotationMatrix

计算二维旋转的仿射变换矩阵

```
CvMat* cv2DRotationMatrix( CvPoint2D32f center, double angle,
                           double scale, CvMat* map_matrix );
```

center
　　输入图像的旋转中心坐标
angle
　　旋转角度（度）。正值表示逆时针旋转(坐标原点假设在左上角).
scale
　　各项同性的尺度因子
map_matrix
　　输出 2×3 矩阵的指针

函数 cv2DRotationMatrix 计算矩阵:

```
[  α   β  |  (1-α)*center.x -  β*center.y ]
[ -β   α  |  β*center.x + (1-α)*center.y ]
```

where α=scale*cos(angle), β=scale*sin(angle)

该变换并不改变原始旋转中心点的坐标，如果这不是操作目的，则可以通过调整平移量改变其坐标(译者注：通过简单的推导可知，仿射变换的实现是首先将旋转中心置为坐标原点，再进行旋转和尺度变换，最后重新将坐标原点设定为输入图像的左上角，这里的平移量是 center.x, center.y).

WarpPerspective

对图像进行透视变换

```
void cvWarpPerspective( const CvArr* src, CvArr* dst, const CvMat*
map_matrix,
                        int
flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
                        CvScalar fillval=cvScalarAll(0) );
```

src
　　输入图像.
dst
　　输出图像.
map_matrix
　　3×3 变换矩阵
flags
　　插值方法和以下开关选项的组合:

- CV_WARP_FILL_OUTLIERS - 填充所有缩小图像的象素。如果部分象素落在输入图像的边界外，那么它们的值设定为 fillval.
- CV_WARP_INVERSE_MAP - 指定 matrix 是输出图像到输入图像的反变换，因此可以直接用来做象素插值。否则, 函数从 map_matrix 得到反变换。

fillval
　　用来填充边界外面的值

函数 cvWarpPerspective 利用下面指定矩阵变换输入图像:

$$dst(x',y') \leftarrow src(x,y)$$

- 如果没有指定 CV_WARP_INVERSE_MAP ，
  $(x',y')^T = map\_matrix \cdot (x,y,1)^T$,
- 否则，$(x,y)^T = map\_matrix \cdot (x',y',1)^T$

要变换稀疏矩阵，使用 cxcore 中的函数 cvTransform 。

WarpPerspectiveQMatrix

用 4 个对应点计算透视变换矩阵

```
CvMat* cvWarpPerspectiveQMatrix( const CvPoint2D32f* src,
                                 const CvPoint2D32f* dst,
                                 CvMat* map_matrix );
```

src
　　输入图像的四边形的 4 个点坐标
dst
　　输出图像的对应四边形的 4 个点坐标
map_matrix
　　输出的 3×3 矩阵

函数 cvWarpPerspectiveQMatrix 计算透视变换矩阵，使得：

(tix'i,tiy'i,ti)T=matrix•(xi,yi,1)T

其中 dst(i)=(x'i,y'i), src(i)=(xi,yi), i=0..3.

GetPerspectiveTransform

由四对点计算透射变换

```
CvMat* cvGetPerspectiveTransform( const CvPoint2D32f* src, const
CvPoint2D32f* dst,
                                  CvMat* map_matrix );

#define cvWarpPerspectiveQMatrix cvGetPerspectiveTransform
```

src
　　输入图像的四边形顶点坐标。
dst
　　输出图像的相应的四边形顶点坐标。
map_matrix
　　指向 3×3 输出矩阵的指针。

函数 cvGetPerspectiveTransform 计算满足以下关系的透射变换矩阵：

$$(t_i x'_i, t_i y'_i, t_i)^T = map\_matrix \cdot (x_i, y_i, 1)^T$$

这里, $dst(i) = (x'_i, y'_i)$, $src(i) = (x_i, y_i)$, $i = 0..3$.

Remap

对图像进行普通几何变换

```
void cvRemap( const CvArr* src, CvArr* dst,
              const CvArr* mapx, const CvArr* mapy,
              int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
              CvScalar fillval=cvScalarAll(0) );
```

src
　　输入图像.

dst
　　输出图像.

mapx
　　x 坐标的映射 (32fC1 image).

mapy
　　y 坐标的映射 (32fC1 image).

flags
　　插值方法和以下开关选项的组合:

- CV_WARP_FILL_OUTLIERS - 填充边界外的像素. 如果输出图像的部分象素落在变换后的边界外，那么它们的值设定为 fillval。

fillval
　　用来填充边界外面的值.

函数 cvRemap 利用下面指定的矩阵变换输入图像:

```
dst(x,y)<-src(mapx(x,y),mapy(x,y))
```

与其它几何变换类似，可以使用一些插值方法（由用户指定，译者注：同 cvResize）来计算非整数坐标的像素值。

LogPolar

把图像映射到极指数空间

```
void cvLogPolar( const CvArr* src, CvArr* dst,
                 CvPoint2D32f center, double M,
                 int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
```

src
　　输入图像。
dst
　　输出图像。
center
　　变换的中心，输出图像在这里最精确。
M
　　幅度的尺度参数，见下面公式。
flags
　　插值方法和以下选择标志的结合

- CV_WARP_FILL_OUTLIERS -填充输出图像所有像素，如果这些点有和外点对应的，则置零。
- CV_WARP_INVERSE_MAP - 表示矩阵由输出图像到输入图像的逆变换，并且因此可以直接用于像素插值。否则，函数从 map_matrix 中寻找逆变换。

fillval
　　用于填充外点的值。

函数 cvLogPolar 用以下变换变换输入图像：

正变换 (CV_WARP_INVERSE_MAP 未置位):

dst(phi,rho)<-src(x,y)

逆变换 (CV_WARP_INVERSE_MAP 置位):

dst(x,y)<-src(phi,rho),

这里，

rho=M*log(sqrt(x2+y2))
phi=atan(y/x)

此函数模仿人类视网膜中央凹视力，并且对于目标跟踪等可用于快速尺度和旋转变换不变模板匹配。

Example. Log-polar transformation.

```
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv)
{
    IplImage* src;

    if( argc == 2 && (src=cvLoadImage(argv[1],1)  != 0 )
    {
        IplImage* dst = cvCreateImage( cvSize(256,256), 8, 3 );
        IplImage* src2 = cvCreateImage( cvGetSize(src), 8, 3 );
        cvLogPolar( src, dst,
cvPoint2D32f(src->width/2,src->height/2),
 40, CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
        cvLogPolar( dst, src2,
cvPoint2D32f(src->width/2,src->height/2),
 40, CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS+CV_WARP_INVERSE_MAP );
        cvNamedWindow( "log-polar", 1 );
        cvShowImage( "log-polar", dst );
        cvNamedWindow( "inverse log-polar", 1 );
        cvShowImage( "inverse log-polar", src2 );
        cvWaitKey();
    }
    return 0;
}
```

And this is what the program displays when opencv/samples/c/fruits.jpg is passed to it

形态学操作

CreateStructuringElementEx

创建结构元素

```
IplConvKernel* cvCreateStructuringElementEx(
 int cols, int rows, int anchor_x, int anchor_y,
 int shape, int* values=NULL );
```

cols
: 结构元素的列数目

rows
: 结构元素的行数目

anchor_x
: 锚点的相对水平偏移量

anchor_y
: 锚点的相对垂直偏移量

shape
: 结构元素的形状，可以是下列值：

- CV_SHAPE_RECT, 长方形元素;
- CV_SHAPE_CROSS, 交错元素 a cross-shaped element;
- CV_SHAPE_ELLIPSE, 椭圆元素;
- CV_SHAPE_CUSTOM, 用户自定义元素。这种情况下参数 values 定义了 mask,即象素的那个邻域必须考虑。

values
: 指向结构元素的指针，它是一个平面数组，表示对元素矩阵逐行扫描。(非零点表示该点属于结构元)。如果指针为空，则表示平面数组中的所有元素都是非零的，即结构元是一个长方形(该参数仅仅当 shape 参数是 CV_SHAPE_CUSTOM 时才予以考虑)。

函数 cv CreateStructuringElementEx 分配和填充结构 IplConvKernel, 它可作为形态操作中的结构元素。

ReleaseStructuringElement

删除结构元素

```
void cvReleaseStructuringElement( IplConvKernel** element );
```

element
: 被删除的结构元素的指针

函数 cvReleaseStructuringElement 释放结构 IplConvKernel 。如果 *element 为 NULL, 则函数不作用。

Erode

使用任意结构元素腐蚀图像

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

函数 cvErode 对输入图像使用指定的结构元素进行腐蚀，该结构元素决定每个具有最小值象素点的邻域形状：

dst=erode(src,element): dst(x,y)=min((x',y') in element))src(x+x',y+y')

函数可能是本地操作，不需另外开辟存储空间的意思。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。

Dilate

使用任意结构元素膨胀图像

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

函数 cvDilate 对输入图像使用指定的结构元进行膨胀，该结构决定每个具有最小值象素点的邻域形状：

dst=dilate(src,element): dst(x,y)=max((x',y') in element))src(x+x',y+y')

函数支持(in-place)模式。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。

MorphologyEx

高级形态学变换

```
void cvMorphologyEx( const CvArr* src, CvArr* dst, CvArr* temp,
                     IplConvKernel* element, int operation, int
iterations=1 );
```

src
　　输入图像.
dst
　　输出图像.
temp
　　临时图像，某些情况下需要
element
　　结构元素
operation
　　形态操作的类型:
　　CV_MOP_OPEN - 开运算
　　CV_MOP_CLOSE - 闭运算
　　CV_MOP_GRADIENT - 形态梯度
　　CV_MOP_TOPHAT - "顶帽"
　　CV_MOP_BLACKHAT - "黑帽"
iterations
　　膨胀和腐蚀次数.

函数 cvMorphologyEx 在膨胀和腐蚀基本操作的基础上，完成一些高级的形态变换：

开运算

　　dst=open(src,element)=dilate(erode(src,element),element)
闭运算

　　dst=close(src,element)=erode(dilate(src,element),element)
形态梯度

dst=morph_grad(src,element)=dilate(src,element)-erode(src,element)

"顶帽"

dst=tophat(src,element)=src-open(src,element)

"黑帽"

dst=blackhat(src,element)=close(src,element)-src

临时图像 temp 在形态梯度以及对“顶帽”和“黑帽”操作时的 in-place 模式下需要。

滤波器与色彩空间变换

Smooth

各种方法的图像平滑

```
void cvSmooth( const CvArr* src, CvArr* dst,
               int smoothtype=CV_GAUSSIAN,
               int param1=3, int param2=0, double param3=0, double
param4=0 );
```

src

输入图像.

dst

输出图像.

smoothtype

平滑方法:

- CV_BLUR_NO_SCALE (简单不带尺度变换的模糊) - 对每个象素的 param1×param2 领域求和。如果邻域大小是变化的，可以事先利用函数 cvIntegral 计算积分图像。
- CV_BLUR (simple blur) - 对每个象素 param1×param2 邻域 求和并做尺度变换 1/(param1•param2).
- CV_GAUSSIAN (gaussian blur) - 对图像进行核大小为 param1×param2 的高斯卷积
- CV_MEDIAN (median blur) - 对图像进行核大小为 param1×param1 的中值滤波 (i.e. 邻域是方的).
- CV_BILATERAL (双向滤波) - 应用双向 3x3 滤波，彩色 sigma=param1，空间 sigma=param2. 关于双向滤波，可参考 http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1

平滑操作的第一个参数.

param2

平滑操作的第二个参数. 对于简单/非尺度变换的高斯模糊的情况，如果 param2 的值 为零，则表示其被设定为 param1。

param3

对应高斯参数的 Gaussian sigma (标准差). 如果为零，则标准差由下面的核尺寸计算：

sigma = (n/2 - 1)*0.3 + 0.8, 其中 n=param1 对应水平核,
n=param2 对应垂直核.

对小的卷积核 (3×3 to 7×7) 使用如上公式所示的标准 sigma 速度会快。如果 param3 不为零，而 param1 和 param2 为零，则核大小有 sigma 计算 (以保证足够精确的操作).

函数 cvSmooth 可使用上面任何一种方法平滑图像。每一种方法都有自己的特点以及局限。

没有缩放的图像平滑仅支持单通道图像，并且支持 8 位到 16 位的转换(与 cvSobel 和 cvaplace 相似)和 32 位浮点数到 32 位浮点数的变换格式。

简单模糊和高斯模糊支持 1- 或 3-通道, 8-比特 和 32-比特 浮点图像。这两种方法可以（in-place）方式处理图像。

中值和双向滤波工作于 1- 或 3-通道， 8-位图像，但是不能以 in-place 方式处理图像.

中值滤波

中值滤波法是一种非线性平滑技术，它将每一象素点的灰度值设置为该点某邻域窗口内的所有象素点灰度值的中值。实现方法：

1. 通过从图像中的某个采样窗口取出奇数个数据进行排序
2. 用排序后的中值取代要处理的数据即可

中值滤波法对消除椒盐噪音非常有效，在光学测量条纹图象的相位分析处理方法中有特殊作用，但在条纹中心分析方法中作用不大。中值滤波在图像处理中,常用于用来保护边缘信息,是经典的平滑噪声的方法

中值滤波原理

中值滤波是基于排序统计理论的一种能有效抑制噪声的非线性信号处理技术，中值滤波的基本原理是把数字图像或数字序列中一点的值用该点的一个拎域中各点值的中值代替，让周围的像素值接近的值，从而消除孤立的噪声点。方法是去

某种结构的二维滑动模板，将板内像素按照像素值的大小进行排序，生成单调上升（或下降）的为二维数据序列。二维中值滤波输出为 g（x,y）
=med{f(x-k,y-l),(k,l∈W)} ，其中,f(x,y)， g(x,y)分别为原始图像和处理后图像。W 为二维模板，通常为 2*2，3*3 区域，也可以是不同的的形状，如线状，圆形，十字形，圆环形等。

高斯滤波

高斯滤波实质上是一种信号的滤波器，其用途是信号的平滑处理，我们知道数字图像用于后期应用，其噪声是最大的问题，由于误差会累计传递等原因，很多图像处理教材会在很早的时候介绍 Gauss 滤波器，用于得到信噪比 SNR 较高的图像（反应真实信号）。于此相关的有 Gauss-Lapplace 变换，其实就是为了得到较好的图像边缘，先对图像做 Gauss 平滑滤波，剔除噪声，然后求二阶导矢，用二阶导的过零点确定边缘，在计算时也是频域乘积=>空域卷积。

滤波器就是建立的一个数学模型，通过这个模型来将图像数据进行能量转化，能量低的就排除掉，噪声就是属于低能量部分

其实编程运算的话就是一个模板运算，拿图像的八连通区域来说，中间点的像素值就等于八连通区的像素值的均值，这样达到平滑的效果

若使用理想滤波器，会在图像中产生振铃现象。采用高斯滤波器的话，系统函数是平滑的，避免了振铃现象。

Filter2D

对图像做卷积

```
void cvFilter2D( const CvArr* src, CvArr* dst,
                 const CvMat* kernel,
                 CvPoint anchor=cvPoint(-1,-1));
```

src
　　输入图像.
dst
　　输出图像.
kernel
　　卷积核, 单通道浮点矩阵. 如果想要应用不同的核于不同的通道，先用 cvSplit 函数分解图像到单个色彩通道上，然后单独处理。
anchor
　　核的锚点表示一个被滤波的点在核内的位置。 锚点应该处于核内部。缺省值 (-1,-1) 表示锚点在核中心。

函数 cvFilter2D 对图像进行线性滤波，支持 In-place 操作。当核运算部分超出输入图像时，函数从最近邻的图像内部象素插值得到边界外面的象素值。

CopyMakeBorder

复制图像并且制作边界。

```
void cvCopyMakeBorder( const CvArr* src, CvArr* dst, CvPoint offset,
                       int bordertype, CvScalar value=cvScalarAll(0) );
```

src
　　输入图像。

dst
　　输出图像。

offset
　　输入图像（或者其 ROI）欲拷贝到的输出图像长方形的左上角坐标（或者左下角坐标，如果以左下角为原点）。长方形的尺寸要和原图像的尺寸的 ROI 分之一匹配。

bordertype
　　已拷贝的原图像长方形的边界的类型：
　　IPL_BORDER_CONSTANT - 填充边界为固定值，值由函数最后一个参数指定。IPL_BORDER_REPLICATE -边界用上下行或者左右列来复制填充。（其他两种 IPL 边界类型， IPL_BORDER_REFLECT 和 IPL_BORDER_WRAP 现已不支持）。

value
　　如果边界类型为 IPL_BORDER_CONSTANT 的话，那么此为边界像素的值。

函数 cvCopyMakeBorder 拷贝输入 2 维阵列到输出阵列的内部并且在拷贝区域的周围制作一个指定类型的边界。函数可以用来模拟和嵌入在指定算法实现中的边界不同的类型。例如：和 opencv 中大多数其他滤波函数一样，一些形态学函数内部使用复制边界类型，但是用户可能需要零边界或者填充为 1 或 255 的边界。

Integral

计算积分图像

```
void cvIntegral( const CvArr* image, CvArr* sum, CvArr* sqsum=NULL,
CvArr* tilted_sum=NULL );
```

image
　　输入图像, W×H, 单通道，8 位或浮点 (32f 或 64f).

sum
　　积分图像, W+1×H+1(译者注：原文的公式应该写成(W+1)×(H+1),避免误会), 单通道，32 位整数或 double 精度的浮点数(64f).

sqsum

对象素值平方的积分图像，W+1×H+1(译者注：原文的公式应该写成(W+1)×(H+1),避免误会)，单通道，32 位整数或 double 精度的浮点数(64f).

tilted_sum

旋转 45 度的积分图像，单通道，32 位整数或 double 精度的浮点数(64f).

函数 cvIntegral 计算一次或高次积分图像：

$$sum(X,Y) = \sum_{x<X,y<Y} image(x,y)$$

$$sqsum(X,Y) = \sum_{x<X,y<Y} image(x,y)^2$$

$$tilted\_sum(X,Y) = \sum_{y<Y,\ |x-X|<y} image(x,y)$$

利用积分图像，可以计算在某象素的上－右方的或者旋转的矩形区域中进行求和、求均值以及标准方差的计算，并且保证运算的复杂度为 O(1)。例如：

$$\sum_{x_1 \le x < x_2, y_1 \le y < y_2} image(x,y) = sum(x_2,y_2) - sum(x_1,y_2) - sum(x_2,y_1) + sum(x_1,x$$

因此可以在变化的窗口内做快速平滑或窗口相关等操作。

CvtColor

色彩空间转换

```
void cvCvtColor( const CvArr* src, CvArr* dst, int code );
```

src

输入的 8-bit , 16-bit 或 32-bit 单倍精度浮点数影像.

dst

输出的 8-bit , 16-bit 或 32-bit 单倍精度浮点数影像.

code

色彩空间转换,通过定义 CV_<src_color_space>2<dst_color_space> 常数 (见下面).

函数 cvCvtColor 将输入图像从一个色彩空间转换为另外一个色彩空间。函数忽略 IplImage 头中定义的 colorModel 和 channelSeq 域,所以输入图像的色彩

空间应该正确指定 (包括通道的顺序,对RGB空间而言,BGR 意味着布局为 B0 G0 R0 B1 G1 R1 ... 层叠的 24-位格式,而 RGB 意味着布局为 R0 G0 B0 R1 G1 B1 ... 层叠的24-位格式. 函数做如下变换：

RGB 空间内部的变换，如增加/删除 alpha 通道，反相通道顺序，到16位 RGB 彩色或者15位RGB彩色的正逆转换(Rx5:Gx6:Rx5),以及到灰度图像的正逆转换，使用：

RGB[A]->Gray: Y=0.212671*R + 0.715160*G + 0.072169*B + 0*A
Gray->RGB[A]: R=Y G=Y B=Y A=0

所有可能的图像色彩空间的相互变换公式列举如下：

RGB<=>XYZ (CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB):

```
|X|   |0.412411  0.357585  0.180454| |R|
|Y| = |0.212649  0.715169  0.072182|*|G|
|Z|   |0.019332  0.119195  0.950390| |B|

|R|   | 3.240479  -1.53715  -0.498535| |X|
|G| = |-0.969256   1.875991  0.041556|*|Y|
|B|   | 0.055648  -0.204043  1.057311| |Z|
```

RGB<=>YCrCb (CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB)

Y=0.299*R + 0.587*G + 0.114*B
Cr=(R-Y)*0.713 + 128
Cb=(B-Y)*0.564 + 128

R=Y + 1.403*(Cr - 128)
G=Y - 0.344*(Cr - 128) - 0.714*(Cb - 128)
B=Y + 1.773*(Cb - 128)

RGB=>HSV (CV_BGR2HSV,CV_RGB2HSV)

V=max(R,G,B)
S=(V-min(R,G,B))*255/V if V!=0, 0 otherwise

```
       (G - B)*60/S,  if V=R
H= 180+(B - R)*60/S,  if V=G
   240+(R - G)*60/S,  if V=B
```

if H<0 then H=H+360

使用上面从 0° 到 360° 变化的公式计算色调（hue）值，确保它们被 2 除后能适用于 8 位。

RGB=>Lab (CV_BGR2Lab, CV_RGB2Lab)

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.433910 & 0.376220 & 0.189860 \\ 0.212649 & 0.715169 & 0.072182 \\ 0.017756 & 0.109478 & 0.872915 \end{bmatrix} * \begin{bmatrix} R/255 \\ G/255 \\ B/255 \end{bmatrix}$$

$$L = \begin{cases} 116*Y1/3 & \text{for } Y>0.008856 \\ 903.3*Y & \text{for } Y<=0.008856 \end{cases}$$

$$a = 500*(f(X)-f(Y))$$
$$b = 200*(f(Y)-f(Z))$$
$$\text{where } f(t)=\begin{cases} t1/3 & \text{for } t>0.008856 \\ 7.787*t+16/116 & \text{for } t<=0.008856 \end{cases}$$

上面的公式可以参考
http://www.cica.indiana.edu/cica/faq/color_spaces/color.spaces.html

RGB=>HLS (CV_BGR2HLS, CV_RGB2HLS)

HSL 表示 hue(色相)、saturation(饱和度)、lightness(亮度)。有的地方也称为 HSI，其中 I 表示 intensity(强度)

转换公式见
http://zh.wikipedia.org/wiki/HSL_%E8%89%B2%E5%BD%A9%E7%A9%BA%E9%97%B4

Bayer=>RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR,

CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerRG2BGR, CV_BayerGB2RGB,

CV_BayerGR2BGR, CV_BayerRG2RGB, CV_BayerBG2BGR, CV_BayerGR2RGB,

CV_BayerGB2BGR)

Bayer 模式被广泛应用于 CCD 和 CMOS 摄像头. 它允许从一个单独平面中得到彩色图像，该平面中的 R/G/B 象素点被安排如下：

R	G	R	G	R
G	B	G	B	G

R	G	R	G	R
G	B	G	B	G
R	G	R	G	R
G	B	G	B	G

对像素输出的 RGB 份量由该像素的 1、2 或者 4 邻域中具有相同颜色的点插值得到。以上的模式可以通过向左或者向上平移一个像素点来作一些修改。转换常量 CV_BayerC1C22{RGB|RGB}中的两个字母 C1 和 C2 表示特定的模式类型：颜色份量分别来自于第二行，第二和第三列。比如说，上述的模式具有很流行的"BG"类型。

Threshold

对数组元素进行固定阈值操作

```
void cvThreshold( const CvArr* src, CvArr* dst, double threshold,
                  double max_value, int threshold_type );
```

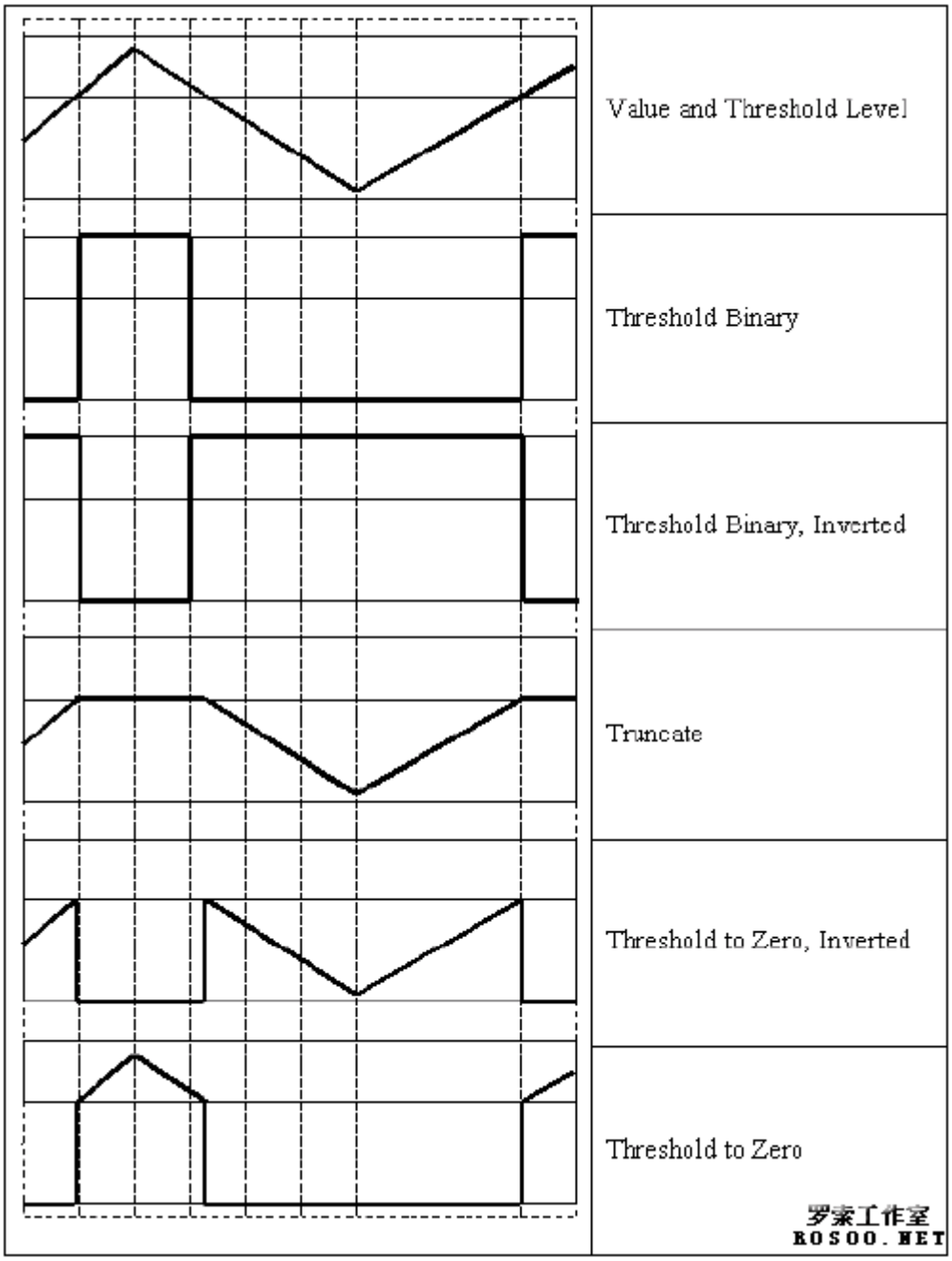
src
　　原始数组 (单通道 , 8-bit of 32-bit 浮点数).
dst
　　输出数组，必须与 src 的类型一致，或者为 8-bit.
threshold
　　阈值
max_value
　　使用 CV_THRESH_BINARY 和 CV_THRESH_BINARY_INV 的最大值.
threshold_type
　　阈值类型 (见讨论)

函数 cvThreshold 对单通道数组应用固定阈值操作。该函数的典型应用是对灰度图像进行阈值操作得到二值图像。(cvCmpS 也可以达到此目的) 或者是去掉噪声，例如过滤很小或很大象素值的图像点。本函数支持的对图像取阈值的方法由 threshold_type 确定:

```
threshold_type=CV_THRESH_BINARY:
dst(x,y) = max_value, if src(x,y)>threshold
           0, otherwise

threshold_type=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>threshold
           max_value, otherwise

threshold_type=CV_THRESH_TRUNC:
dst(x,y) = threshold, if src(x,y)>threshold
           src(x,y), otherwise
```

```
threshold_type=CV_THRESH_TOZERO:
dst(x,y) = src(x,y), if (x,y)>threshold
           0, otherwise

threshold_type=CV_THRESH_TOZERO_INV:
dst(x,y) = 0, if src(x,y)>threshold
           src(x,y), otherwise
```

下面是图形化的阈值描述：



AdaptiveThreshold

自适应阈值方法

```
void cvAdaptiveThreshold( const CvArr* src, CvArr* dst, double max_value,
                          int
adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C,
                          int threshold_type=CV_THRESH_BINARY,
                          int block_size=3, double param1=5 );
```

src
　　输入图像.
dst
　　输出图像.
max_value
　　使用 CV_THRESH_BINARY 和 CV_THRESH_BINARY_INV 的最大值.
adaptive_method
　　自适应阈值算法使用：CV_ADAPTIVE_THRESH_MEAN_C 或
　　CV_ADAPTIVE_THRESH_GAUSSIAN_C （见讨论）.
threshold_type
　　取阈值类型：必须是下者之一

- CV_THRESH_BINARY,
- CV_THRESH_BINARY_INV

block_size
　　用来计算阈值的象素邻域大小: 3, 5, 7, ...
param1
　　与方法有关的参数。对方法 CV_ADAPTIVE_THRESH_MEAN_C 和
　　CV_ADAPTIVE_THRESH_GAUSSIAN_C，它是一个从均值或加权均值提取的常
　　数（见讨论），尽管它可以是负数。

函数 cvAdaptiveThreshold 将灰度图像变换到二值图像，采用下面公式：

```
threshold_type=CV_THRESH_BINARY:
dst(x,y) = max_value, if src(x,y)>T(x,y)
           0, otherwise

threshold_type=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>T(x,y)
           max_value, otherwise
```

其中 TI 是为每一个象素点单独计算的阈值

对方法 CV_ADAPTIVE_THRESH_MEAN_C，先求出块中的均值，再减掉 param1。

对方法 CV_ADAPTIVE_THRESH_GAUSSIAN_C ，先求出块中的加权和(gaussian)，再减掉 param1。

金字塔及其应用

PyrDown

图像的下采样

```
void cvPyrDown( const CvArr* src, CvArr* dst, int
filter=CV_GAUSSIAN_5x5 );
```

src
　　输入图像.
dst
　　输出图像，宽度和高度应是输入图像的一半 ,传入前必须已经完成初始化
filter
　　卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5

函数 cvPyrDown 使用 Gaussian 金字塔分解对输入图像向下采样。首先它对输入图像用指定滤波器进行卷积，然后通过拒绝偶数的行与列来下采样图像。

PyrUp

图像的上采样

```
void cvPyrUp( const CvArr* src, CvArr* dst, int filter=CV_GAUSSIAN_5x5 );
```

src
　　输入图像.
dst
　　输出图像，宽度和高度应是输入图像的 2 倍
filter
　　卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5

函数 cvPyrUp 使用 Gaussian 金字塔分解对输入图像向上采样。首先通过在图像中插入 0 偶数行和偶数列，然后对得到的图像用指定的滤波器进行高斯卷积，其中滤波器乘以 4 做插值。所以输出图像是输入图像的 4 倍大小。(hunnish: 原理不清楚，尚待探讨)

连接部件

CvConnectedComp

连接部件

```
typedef struct CvConnectedComp
{
    double area; /* 连通域的面积 */
    float value; /* 分割域的灰度缩放值 */
    CvRect rect; /* 分割域的 ROI */
} CvConnectedComp;
```

FloodFill

用指定颜色填充一个连接域

```
void cvFloodFill( CvArr* image, CvPoint seed_point, CvScalar new_val,
                  CvScalar lo_diff=cvScalarAll(0), CvScalar
up_diff=cvScalarAll(0),
                  CvConnectedComp* comp=NULL, int flags=4, CvArr*
mask=NULL );
#define CV_FLOODFILL_FIXED_RANGE (1 << 16)
#define CV_FLOODFILL_MASK_ONLY   (1 << 17)
```

image
: 输入的 1- 或 3-通道, 8-比特或浮点数图像。输入的图像将被函数的操作所改变，除非你选择 CV_FLOODFILL_MASK_ONLY 选项 (见下面).

seed_point
: 开始的种子点.

new_val
: 新的重新绘制的象素值

lo_diff
: 当前观察象素值与其部件领域象素或者待加入该部件的种子象素之负差(Lower difference)的最大值。对 8-比特 彩色图像，它是一个 packed value.

up_diff
: 当前观察象素值与其部件领域象素或者待加入该部件的种子象素之正差(upper difference)的最大值。 对 8-比特 彩色图像，它是一个 packed value.

comp
: 指向部件结构体的指针，该结构体的内容由函数用重绘区域的信息填充。

flags
: 操作选项. 低位比特包含连通值, 4 (缺省) 或 8, 在函数执行连通过程中确定使用哪种邻域方式。高位比特可以是 0 或下面的开关选项的组合：

- CV_FLOODFILL_FIXED_RANGE - 如果设置，则考虑当前象素与种子象素之间的差，否则考虑当前象素与其相邻象素的差。(范围是浮点数).
- CV_FLOODFILL_MASK_ONLY - 如果设置，函数不填充原始图像 (忽略 new_val), 但填充掩模图像 (这种情况下 MASK 必须是非空的).

mask

运算掩模, 应该是单通道、8-比特图像, 长和宽上都比输入图像 image 大两个象素点。若非空，则函数使用且更新掩模, 所以使用者需对 mask 内容的初始化负责。填充不会经过 MASK 的非零象素, 例如，一个边缘检测子的输出可以用来作为 MASK 来阻止填充边缘。或者有可能在多次的函数调用中使用同一个 MASK，以保证填充的区域不会重叠。注意: 因为 MASK 比欲填充图像大，所以 mask 中与输入图像(x,y)像素点相对应的点具有(x+1,y+1)坐标。

函数 cvFloodFill 用指定颜色，从种子点开始填充一个连通域。连通性由象素值的接近程度来衡量。在点 (x, y) 的象素被认为是属于重新绘制的区域，如果：

src(x',y')-lo_diff<=src(x,y)<=src(x',y')+up_diff, 灰度图像，浮动范围
src(seed.x,seed.y)-lo<=src(x,y)<=src(seed.x,seed.y)+up_diff, 灰度图像，固定范围
src(x',y')r-lo_diffr<=src(x,y)r<=src(x',y')r+up_diffr 和
src(x',y')g-lo_diffg<=src(x,y)g<=src(x',y')g+up_diffg 和
src(x',y')b-lo_diffb<=src(x,y)b<=src(x',y')b+up_diffb, 彩色图像，浮动范围
src(seed.x,seed.y)r-lo_diffr<=src(x,y)r<=src(seed.x,seed.y)r+up_diffr 和
src(seed.x,seed.y)g-lo_diffg<=src(x,y)g<=src(seed.x,seed.y)g+up_diffg 和
src(seed.x,seed.y)b-lo_diffb<=src(x,y)b<=src(seed.x,seed.y)b+up_diffb, 彩色图像，固定范围

其中 src(x',y') 是象素邻域点的值。也就是说，为了被加入到连通域中，一个象素的彩色/亮度应该足够接近于：

- 它的邻域象素的彩色/亮度值，当该邻域点已经被认为属于浮动范围情况下的连通域。
- 固定范围情况下的种子点的彩色/亮度值

FindContours

在二值图像中寻找轮廓

```
int cvFindContours( CvArr* image, CvMemStorage* storage, CvSeq**
first_contour,
                    int header_size=sizeof(CvContour), int
mode=CV_RETR_LIST,
                    int method=CV_CHAIN_APPROX_SIMPLE, CvPoint
offset=cvPoint(0,0) );
```

image

输入的 8-比特、单通道图像. 非零元素被当成 1， 0 象素值保留为 0 - 从而图像被看成二值的。为了从灰度图像中得到这样的二值图像，可以使用 cvThreshold, cvAdaptiveThreshold 或 cvCanny. 本函数改变输入图像内容。

storage

得到的轮廓的存储容器

first_contour

输出参数：包含第一个输出轮廓的指针

header_size

如果 method=CV_CHAIN_CODE，则序列头的大小 >=sizeof(CvChain)，否则 >=sizeof(CvContour) .

mode

提取模式.

- CV_RETR_EXTERNAL - 只提取最外层的轮廓
- CV_RETR_LIST - 提取所有轮廓，并且放置在 list 中
- CV_RETR_CCOMP - 提取所有轮廓，并且将其组织为两层的 hierarchy: 顶层为连通域的外围边界，次层为洞的内层边界。
- CV_RETR_TREE - 提取所有轮廓，并且重构嵌套轮廓的全部 hierarchy

method

逼近方法 (对所有节点, 不包括使用内部逼近的 CV_RETR_RUNS).

- CV_CHAIN_CODE - Freeman 链码的输出轮廓. 其它方法输出多边形 (定点序列).
- CV_CHAIN_APPROX_NONE - 将所有点由链码形式翻译(转化）为点序列形式
- CV_CHAIN_APPROX_SIMPLE - 压缩水平、垂直和对角分割，即函数只保留末端的象素点;
- CV_CHAIN_APPROX_TC89_L1,
- CV_CHAIN_APPROX_TC89_KCOS - 应用 Teh-Chin 链逼近算法. CV_LINK_RUNS - 通过连接为 1 的水平碎片使用完全不同的轮廓提取算法。仅有 CV_RETR_LIST 提取模式可以在本方法中应用.

offset

每一个轮廓点的偏移量. 当轮廓是从图像 ROI 中提取出来的时候，使用偏移量有用，因为可以从整个图像上下文来对轮廓做分析.

函数 cvFindContours 从二值图像中提取轮廓，并且返回提取轮廓的数目。指针 first_contour 的内容由函数填写。它包含第一个最外层轮廓的指针，如果指针为 NULL，则没有检测到轮廓（比如图像是全黑的）。其它轮廓可以从 first_contour 利用 h_next 和 v_next 链接访问到。 在 cvDrawContours 的样例显示如何使用轮廓来进行连通域的检测。轮廓也可以用来做形状分析和对象识别 - 见 CVPR2001 教程中的 squares 样例。该教程可以在 SourceForge 网站上找到。

StartFindContours

初始化轮廓的扫描过程

```
CvContourScanner cvStartFindContours( CvArr* image, CvMemStorage*
storage,
                                      int
header_size=sizeof(CvContour),
                                      int mode=CV_RETR_LIST,
                                      int
method=CV_CHAIN_APPROX_SIMPLE,
                                      CvPoint offset=cvPoint(0,0) );
```

image
: 输入的 8-比特、单通道二值图像

storage
: 提取到的轮廓容器

header_size
: 序列头的尺寸 >=sizeof(CvChain) 若 method=CV_CHAIN_CODE，否则尺寸 >=sizeof(CvContour) .

mode
: 提取模式，见 cvFindContours.

method
: 逼近方法。它与 cvFindContours 里的定义一样，但是 CV_LINK_RUNS 不能使用。

offset
: ROI 偏移量，见 cvFindContours.

函数 cvStartFindContours 初始化并且返回轮廓扫描器的指针。扫描器在 cvFindNextContour 使用以提取其馀的轮廓。

FindNextContour

Finds next contour in the image

```
CvSeq* cvFindNextContour( CvContourScanner scanner );
```

scanner
被函数 cvStartFindContours 初始化的轮廓扫描器.

函数 cvFindNextContour 确定和提取图像的下一个轮廓，并且返回它的指针。若没有更多的轮廓，则函数返回 NULL.

SubstituteContour

替换提取的轮廓

```
void cvSubstituteContour( CvContourScanner scanner, CvSeq*
new_contour );
```

scanner
被函数 cvStartFindContours 初始化的轮廓扫描器 ..
new_contour
替换的轮廓

函数 cvSubstituteContour 把用户自定义的轮廓替换前一次的函数 cvFindNextContour 调用所提取的轮廓，该轮廓以用户定义的模式存储在边缘扫描状态之中。轮廓，根据提取状态，被插入到生成的结构，List，二层 hierarchy，或 tree 中。如果参数 new_contour=NULL, 则提取的轮廓不被包含入生成结构中，它的所有后代以后也不会被加入到接口中。

EndFindContours

结束扫描过程

```
 CvSeq* cvEndFindContours( CvContourScanner* scanner );
```

scanner
轮廓扫描的指针.

函数 cvEndFindContours 结束扫描过程，并且返回最高层的第一个轮廓的指针。

PyrSegmentation

用金字塔实现图像分割

```
void cvPyrSegmentation( IplImage* src, IplImage* dst,
                        CvMemStorage* storage, CvSeq** comp,
```

int level, double threshold1, double threshold2);

src

输入图像.

dst

输出图像.

storage

Storage: 存储连通部件的序列结果

comp

分割部件的输出序列指针 components.

level

建立金字塔的最大层数

threshold1

建立连接的错误阈值

threshold2

分割簇的错误阈值

函数 cvPyrSegmentation 实现了金字塔方法的图像分割。金字塔建立到 level 指定的最大层数。如果 p(c(a),c(b))<threshold1,则在层 i 的象素点 a 和它的相邻层的父亲象素 b 之间的连接被建立起来，

定义好连接部件后，它们被加入到某些簇中。如果 p(c(A),c(B))<threshold2，则任何两个分割 A 和 B 属于同一簇。

如果输入图像只有一个通道，那么

$p(c^1,c^2)=|c^1-c^2|$.

如果输入图像有单个通道（红、绿、兰），那么

$p(c^1,c^2)=0,3\cdot(c^1r-c^2r)+0,59\cdot(c^1g-c^2g)+0,11\cdot(c^1b-c^2b)$.

每一个簇可以有多个连接部件。 图像 src 和 dst 应该是 8-比特、单通道 或 3-通道图像，且大小一样

PyrMeanShiftFiltering

Does meanshift image segmentation

```
void cvPyrMeanShiftFiltering( const CvArr* src, CvArr* dst,
     double sp, double sr, int max_level=1,
     CvTermCriteria
termcrit=cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,5,1));
```

src

输入的 8-比特,3-信道图象.

dst

和源图象相同大小,相同格式的输出图象.

sp

The spatial window radius.

空间窗的半径

sr

The color window radius.

色彩窗的半径

max_level

Maximum level of the pyramid for the segmentation.

termcrit

Termination criteria: when to stop meanshift iterations.

The function cvPyrMeanShiftFiltering implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X,Y) neighborhood in the joint space-color hyperspace is considered:

{(x,y): X-sp≤x≤X+sp && Y-sp≤y≤Y+sp && ||(R,G,B)-(r,g,b)|| ≤ sr},where (R,G,B) and (r,g,b) are the vectors of color components at (X,Y) and (x,y), respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value (X',Y') and average color vector (R',G',B') are found and they act as the neighborhood center on the next iteration: (X,Y)~(X',Y'), (R,G,B)~(R',G',B').After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration): I(X,Y) <- (R*,G*,B*).Then max_level>0, the gaussian pyramid of max_level+1 levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much (>sr) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when max_level==0).

Watershed

做分水岭图像分割

void cvWatershed(const CvArr* image, CvArr* markers);
image
输入 8 比特 3 通道图像。
markers
输入或输出的 32 比特单通道标记图像。

函数 cvWatershed 实现在[Meyer92]描述的变量分水岭，基于非参数标记的分割算法中的一种。在把图像传给函数之前，用户需要用正指标大致勾画出图像标记的感兴趣区域。比如，每一个区域都表示成一个或者多个像素值 1，2，3 的互联部分。这些部分将作为将来图像区域的种子。标记中所有的其他像素，他们和勾画出的区域关系不明并且应由算法定义，应当被置 0。这个函数的输出则是标记区域所有像素被置为某个种子部分的值，或者在区域边界则置-1。

注：每两个相邻区域也不是必须有一个分水岭边界（-1 像素）分开，例如在初始标记图像里有这样相切的部分。opencv 例程文件夹里面有函数的视觉效果演示和用户例程。见 watershed.cpp。

图像与轮廓矩

Moments

计算多边形和光栅形状的最高达三阶的所有矩

void cvMoments(const CvArr* arr, CvMoments* moments, int binary=0);
arr
图像 (1-通道或 3-通道，有 COI 设置) 或多边形(点的 CvSeq 或一族点的向量).
moments
返回的矩状态接口的指针
binary
(仅对图像) 如果标识为非零，则所有零象素点被当成零，其它的被看成 1.

函数 cvMoments 计算最高达三阶的空间和中心矩，并且将结果存在结构 moments 中。矩用来计算形状的重心，面积，主轴和其它的形状特征，如 7 Hu 不变量等。

GetSpatialMoment

从矩状态结构中提取空间矩

double cvGetSpatialMoment(CvMoments* moments, int x_order, int y_order);
moments
矩状态，由 cvMoments 计算
x_order
提取的 x 次矩, x_order >= 0.
y_order
提取的 y 次矩, y_order >= 0 并且 x_order + y_order <= 3.

函数 cvGetSpatialMoment 提取空间矩，当图像矩被定义为：

Mx_order,y_order=sumx,y(I(x,y)•xx_order•yy_order)

其中 I(x,y) 是象素点 (x, y) 的亮度值.

GetCentralMoment

从矩状态结构中提取中心矩

double cvGetCentralMoment(CvMoments* moments, int x_order, int y_order);
moments
矩状态结构指针
x_order
提取的 x 阶矩, x_order >= 0.
y_order
提取的 y 阶矩， y_order >= 0 且 x_order + y_order <= 3.

函数 cvGetCentralMoment 提取中心矩，其中图像矩的定义是：

μx_order,y_order=sumx,y(I(x,y)•(x-xc)x_order•(y-yc)y_order),

其中 xc=M10/M00, yc=M01/M00 - 重心坐标

GetNormalizedCentralMoment

从矩状态结构中提取归一化的中心矩

double cvGetNormalizedCentralMoment(CvMoments* moments, int x_order, int y_order);
moments
矩状态结构指针
x_order

提取的 x 阶矩, x_order >= 0.

y_order

提取的 y 阶矩， y_order >= 0 且 x_order + y_order <= 3.

函数 cvGetNormalizedCentralMoment 提取归一化中心矩：

ηx_order,y_order= μx_order,y_order/M00((y_order+x_order)/2+1)

GetHuMoments

计算 7 Hu 不变量

```
void cvGetHuMoments( CvMoments* moments, CvHuMoments* hu_moments );
```

moments

矩状态结构的指针

hu_moments

Hu 矩结构的指针.

函数 cvGetHuMoments 计算 7 个 Hu 不变量，它们的定义是：

$h1=\eta 20+\eta 02$

$h2=(\eta 20-\eta 02)^2+4\eta 11^2$

$h3=(\eta 30-3\eta 12)^2+(3\eta 21-\eta 03)^2$

$h4=(\eta 30+\eta 12)^2+(\eta 21+\eta 03)^2$

$h5=(\eta 30-3\eta 12)(\eta 30+\eta 12)[(\eta 30+\eta 12)^2-3(\eta 21+\eta 03)^2]+(3\eta 21-\eta 03)(\eta 21+\eta 03)[3(\eta 30+\eta 12)^2-(\eta 21+\eta 03)^2]$

$h6=(\eta 20-\eta 02)[(\eta 30+\eta 12)^2-(\eta 21+\eta 03)^2]+4\eta 11(\eta 30+\eta 12)(\eta 21+\eta 03)$

$h7=(3\eta 21-\eta 03)(\eta 21+\eta 03)[3(\eta 30+\eta 12)^2-(\eta 21+\eta 03)^2]-(\eta 30-3\eta 12)(\eta 21+\eta 03)[3(\eta 30+\eta 12)^2-(\eta 21+\eta 03)^2]$

这些值被证明为对图像缩放、旋转和反射的不变量。对反射，第 7 个除外，因为它的符号会因为反射而改变。

特殊图像变换

HoughLines

利用 Hough 变换在二值图像中找到直线

```
CvSeq* cvHoughLines2( CvArr* image, void* line_storage, int method,
                      double rho, double theta, int threshold,
                      double param1=0, double param2=0 );
```

image

输入 8-比特、单通道 (二值) 图像，当用 CV_HOUGH_PROBABILISTIC 方法检测的时候其内容会被函数改变

line_storage

检测到的线段存储仓. 可以是内存存储仓 (此种情况下，一个线段序列在存储仓中被创建，并且由函数返回)，或者是包含线段参数的特殊类型(见下面）的具有单行/单列的矩阵(CvMat*)。矩阵头为函数所修改，使得它的 cols/rows 将包含一组检测到的线段。如果 line_storage 是矩阵，而实际线段的数目超过矩阵尺寸，那么最大可能数目的线段被返回(对于标准 hough 变换，线段按照长度降序输出).

method

Hough 变换变量，是下面变量的其中之一：

- CV_HOUGH_STANDARD - 传统或标准 Hough 变换. 每一个线段由两个浮点数 (ρ, θ) 表示，其中 ρ 是直线与原点 (0,0) 之间的距离，θ 线段与 x-轴之间的夹角。因此，矩阵类型必须是 CV_32FC2 type.
- CV_HOUGH_PROBABILISTIC - 概率 Hough 变换(如果图像包含一些长的线性分割，则效率更高). 它返回线段分割而不是整个线段。每个分割用起点和终点来表示，所以矩阵（或创建的序列）类型是 CV_32SC4.
- CV_HOUGH_MULTI_SCALE - 传统 Hough 变换的多尺度变种。线段的编码方式与 CV_HOUGH_STANDARD 的一致。

rho

与象素相关单位的距离精度

theta

弧度测量的角度精度

threshold

阈值参数。如果相应的累计值大于 threshold， 则函数返回的这个线段.

param1

第一个方法相关的参数:

- 对传统 Hough 变换，不使用(0).
- 对概率 Hough 变换，它是最小线段长度.
- 对多尺度 Hough 变换，它是距离精度 rho 的分母 (大致的距离精度是 rho 而精确的应该是 rho / param1).

param2

第二个方法相关参数:

- 对传统 Hough 变换，不使用 (0).

- 对概率 Hough 变换，这个参数表示在同一条直线上进行碎线段连接的最大间隔值(gap), 即当同一条直线上的两条碎线段之间的间隔小于 param2 时，将其合二为一。
- 对多尺度 Hough 变换，它是角度精度 theta 的分母 (大致的角度精度是 theta 而精确的角度应该是 theta / param2).

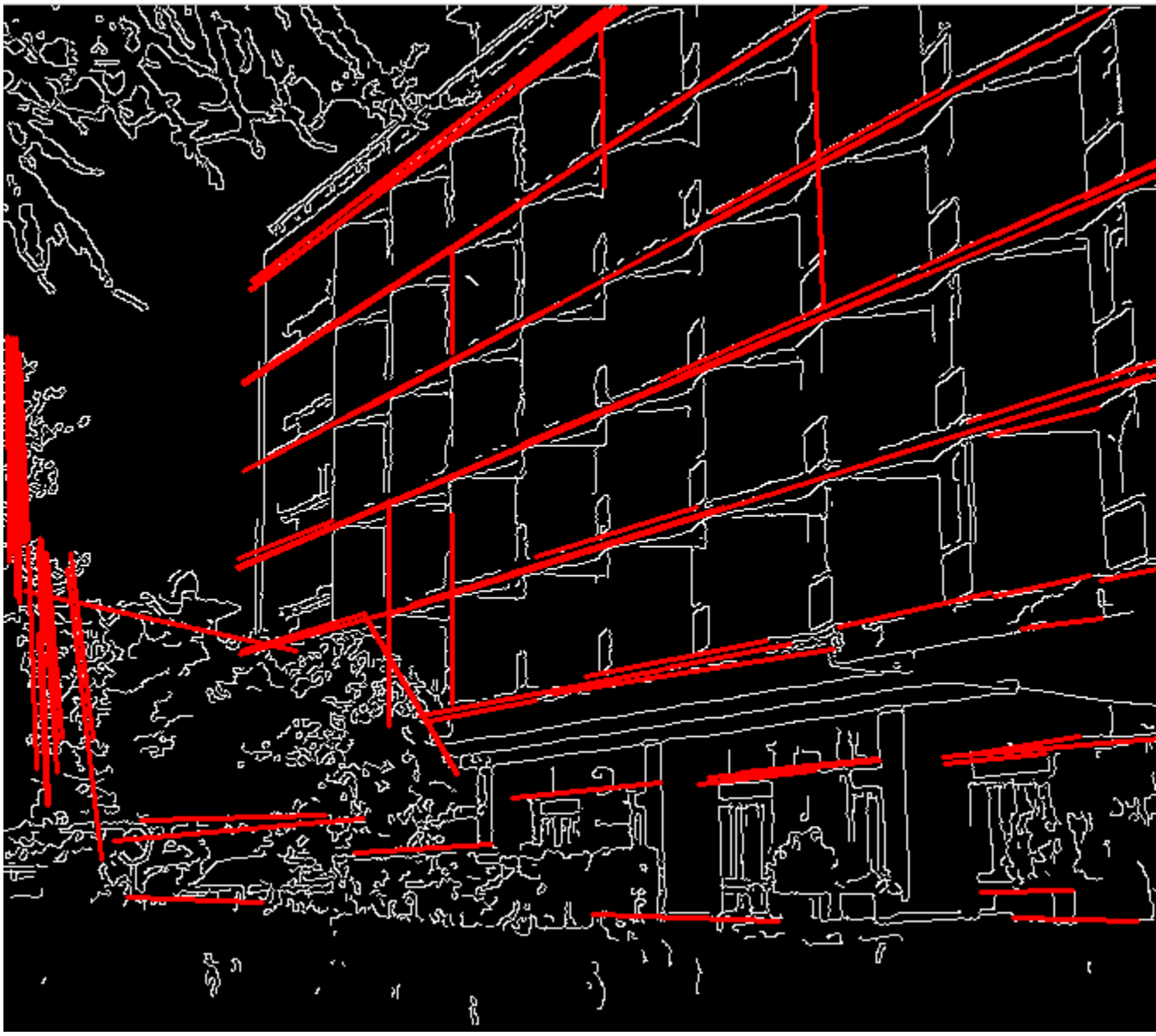
函数 cvHoughLines2 实现了用于线段检测的不同 Hough 变换方法. Example. 用 Hough transform 检测线段

```
/* This is a standalone program. Pass an image name as a first parameter
   of the program.Switch between standard and probabilistic Hough
transform
   by changing "#if 1" to "#if 0" and back */
#include <cv.h>

#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* lines = 0;
        int i;
        IplImage*
src1=cvCreateImage(cvSize(src->width,src->height),IPL_DEPTH_8U,1);

        cvCvtColor(src, src1, CV_BGR2GRAY);
        cvCanny( src1, dst, 50, 200, 3 );

        cvCvtColor( dst, color_dst, CV_GRAY2BGR );
#if 1
        lines = cvHoughLines2( dst, storage, CV_HOUGH_STANDARD, 1,
CV_PI/180, 150, 0, 0 );

        for( i = 0; i < lines->total; i++ )
        {
            float* line = (float*)cvGetSeqElem(lines,i);
            float rho = line[0];
            float theta = line[1];
```

```
            CvPoint pt1, pt2;
            double a = cos(theta), b = sin(theta);
            if( fabs(a) < 0.001 )
            {
                pt1.x = pt2.x = cvRound(rho);
                pt1.y = 0;
                pt2.y = color_dst->height;
            }
            else if( fabs(b) < 0.001 )
            {
                pt1.y = pt2.y = cvRound(rho);
                pt1.x = 0;
                pt2.x = color_dst->width;
            }
            else
            {
                pt1.x = 0;
                pt1.y = cvRound(rho/b);
                pt2.x = cvRound(rho/a);
                pt2.y = 0;
            }
            cvLine( color_dst, pt1, pt2, CV_RGB(255,0,0), 3, 8 );
        }
#else
        lines = cvHoughLines2( dst, storage, CV_HOUGH_PROBABILISTIC, 1,
CV_PI/180, 80, 30, 10 );
        for( i = 0; i < lines->total; i++ )
        {
            CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
            cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3,
8 );
        }
#endif
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "Hough", 1 );
        cvShowImage( "Hough", color_dst );

        cvWaitKey(0);
    }
}
```

这是函数所用的样本图像：

下面是程序的输出，采用概率 Hough transform ("#if 0" 的部分)：



HoughCircles

利用 Hough 变换在灰度图像中找圆

```
CvSeq* cvHoughCircles( CvArr* image, void* circle_storage,
                       int method, double dp, double min_dist,
                       double param1=100, double param2=100,
                       int min_radius=0, int max_radius=0 );
```

image

输入 8-比特、单通道灰度图像.

circle_storage

检测到的圆存储仓. 可以是内存存储仓 (此种情况下,一个线段序列在存储仓中被创建，并且由函数返回)或者是包含圆参数的特殊类型的具有单行/单列的 CV_32FC3 型矩阵(CvMat*). 矩阵头为函数所修改，使得它的 cols/rows 将包含一组检测到的圆。如果 circle_storage 是矩阵，而实际圆的数目超过矩阵尺寸，那么最大可能数目的圆被返回

. 每个圆由三个浮点数表示：圆心坐标(x,y)和半径.

method

Hough 变换方式，目前只支持 CV_HOUGH_GRADIENT, which is basically 21HT, described in [Yuen03].

dp

累加器图像的分辨率。这个参数允许创建一个比输入图像分辨率低的累加器。(这样做是因为有理由认为图像中存在的圆会自然降低到与图像宽高相同数量的范畴）。如果 dp 设置为 1，则分辨率是相同的；如果设置为更大的值(比如 2)，累加器的分辨率受此影响会变小(此情况下为一半)。dp 的值不能比 1 小。

Resolution of the accumulator used to detect centers of the circles. For example, if it is 1, the accumulator will have the same resolution as the input image, if it is 2 - accumulator will have twice smaller width and height, etc.

min_dist

该参数是让算法能明显区分的两个不同圆之间的最小距离。

Minimum distance between centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed.

param1

用于 Canny 的边缘阀值上限，下限被置为上限的一半。

The first method-specific parameter. In case of CV_HOUGH_GRADIENT it is the higher threshold of the two passed to Canny edge detector (the lower one will be twice smaller).

param2

累加器的阀值。

The second method-specific parameter. In case of CV_HOUGH_GRADIENT it is accumulator threshold at the center detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first.

min_radius
　　最小圆半径。

Minimal radius of the circles to search for.

max_radius
　　最大圆半径。

Maximal radius of the circles to search for. By default the maximal radius is set to max(image_width, image_height).

The function cvHoughCircles finds circles in grayscale image using some modification of Hough transform.

Example. Detecting circles with Hough transform.

```
#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* img;
    if( argc == 2 && (img=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* gray = cvCreateImage( cvGetSize(img), 8, 1 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        cvCvtColor( img, gray, CV_BGR2GRAY );
        cvSmooth( gray, gray, CV_GAUSSIAN, 9, 9 );
// smooth it, otherwise a lot of false circles may be detected
        CvSeq* circles = cvHoughCircles( gray, storage,
CV_HOUGH_GRADIENT, 2, gray->height/4, 200, 100 );
        int i;
        for( i = 0; i < circles->total; i++ )
        {
             float* p = (float*)cvGetSeqElem( circles, i );
             cvCircle( img, cvPoint(cvRound(p[0]),cvRound(p[1])),
3, CV_RGB(0,255,0), -1, 8, 0 );
             cvCircle( img, cvPoint(cvRound(p[0]),cvRound(p[1])),
cvRound(p[2]), CV_RGB(255,0,0), 3, 8, 0 );
        }
        cvNamedWindow( "circles", 1 );
        cvShowImage( "circles", img );
    }
    return 0;
```

}

DistTransform

计算输入图像的所有非零元素对其最近零元素的距离

```
void cvDistTransform( const CvArr* src, CvArr* dst, int
distance_type=CV_DIST_L2,
                      int mask_size=3, const float* mask=NULL );
```

src
　　输入 8-比特、单通道 (二值) 图像.

dst
　　含计算出的距离的输出图像(32-比特、浮点数、单通道).

distance_type
　　距离类型; 可以是 CV_DIST_L1, CV_DIST_L2, CV_DIST_C 或 CV_DIST_USER.

mask_size
　　距离变换掩模的大小,可以是 3 或 5. 对 CV_DIST_L1 或 CV_DIST_C 的情况，参数值被强制设定为 3, 因为 3×3 mask 给出 5×5 mask 一样的结果，而且速度还更快。

mask
　　用户自定义距离情况下的 mask。 在 3×3 mask 下它由两个数(水平/垂直位量，对角线位移量）组成， 5×5 mask 下由三个数组成(水平/垂直位移量，对角位移和 国际象棋里的马步(马走日))

函数 cvDistTransform 二值图像每一个象素点到它最邻近零象素点的距离。对零象素，函数设置 0 距离，对其它象素，它寻找由基本位移（水平、垂直、对角线或 knight's move，最后一项对 5×5 mask 有用）构成的最短路径。 全部的距离被认为是基本距离的和。由于距离函数是对称的，所有水平和垂直位移具有同样的代价 (表示为 a), 所有的对角位移具有同样的代价 (表示为 b), 所有的 knight's 移动具有同样的代价 (表示为 c). 对类型 CV_DIST_C 和 CV_DIST_L1，距离的计算是精确的，而类型 CV_DIST_L2 (欧式距离) 距离的计算有某些相对误差 (5×5 mask 给出更精确的结果), OpenCV 使用 [Borgefors86] 推荐的值：

　　CV_DIST_C (3×3):
　　a=1, b=1
　　CV_DIST_L1 (3×3):
　　a=1, b=2
　　CV_DIST_L2 (3×3):
　　a=0.955, b=1.3693
　　CV_DIST_L2 (5×5):
　　a=1, b=1.4, c=2.1969

下面用户自定义距离的的距离域示例（黑点 (0) 在白色方块中间）: 用户自定义 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1	0	1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

用户自定义 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	0	1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

典型的使用快速粗略距离估计 CV_DIST_L2, 3×3 mask , 如果要更精确的距离估计，使用 CV_DIST_L2, 5×5 mask。

When the output parameter labels is not NULL, for every non-zero pixel the function also finds the nearest connected component consisting of zero pixels. The connected components themselves are found as contours in the beginning of the function.

In this mode the processing time is still O(N), where N is the number of pixels. Thus, the function provides a very fast way to compute approximate Voronoi diagram for the binary image.

Inpaint

修复图像中选择区域。

void cvInpaint(const CvArr* src, const CvArr* mask, CvArr* dst,

```
                int flags, double inpaintRadius );
```

src

输入 8 比特单通道或者三通道图像。

mask

修复图像的掩饰，8 比特单通道图像。非零像素表示该区域需要修复。

dst

输出图像，和输入图像相同格式相同大小。

flags

修复方法,以下之一:

CV_INPAINT_NS - 基于 Navier-Stokes 的方法。

CV_INPAINT_TELEA - Alexandru Telea[Telea04]的方法。

inpaintRadius

算法考虑的每个修复点的圆形领域的半径。

函数 cvInpaint 从选择图像区域边界的像素重建该区域。函数可以用来去除扫描相片的灰尘或者刮伤，或者从静态图像或者视频中去除不需要的物体。

直方图

CvHistogram

多维直方图

```
typedef struct CvHistogram
{
    int     type;
    CvArr*  bins;
    float   thresh[CV_MAX_DIM][2]; /* for uniform histograms */
    float** thresh2; /* for non-uniform histograms */
    CvMatND mat; /* embedded matrix header for array histograms */
}
CvHistogram;
```

bins ： 用于存放直方图每个灰度级数目的数组指针，数组在 cvCreateHist 的时候创建，其维数由 cvCreateHist 确定（一般以一维比较常见）

CreateHist

创建直方图

```
CvHistogram* cvCreateHist( int dims, int* sizes, int type,
                           float** ranges=NULL, int uniform=1 );
```

dims

直方图维数的数目

sizes

直方图维数尺寸的数组

type

直方图的表示格式: CV_HIST_ARRAY 意味着直方图数据表示为多维密集数组 CvMatND; CV_HIST_TREE 意味着直方图数据表示为多维稀疏数组 CvSparseMat.

ranges

图中方块范围的数组. 它的内容取决于参数 uniform 的值。这个范围的用处是确定何时计算直方图或决定反向映射（backprojected ），每个方块对应于输入图像的哪个/哪组值。

uniform

归一化标识。 如果不为 0，则 ranges[i]（0<=i<cDims，译者注：cDims 为直方图的维数，对于灰度图为 1，彩色图为 3）是包含两个元素的范围数组，包括直方图第 i 维的上界和下界。在第 i 维上的整个区域 [lower,upper]被分割成 dims[i] 个相等的块（译者注：dims[i]表示直方图第 i 维的块数），这些块用来确定输入象素的第 i 个值（译者注：对于彩色图像，i 确定 R, G,或者 B）的对应的块；如果为 0，则 ranges[i] 是包含 dims[i]+1 个元素的范围数组，包括 lower0, upper0, lower1, upper1 == lower2, ..., upperdims[i]-1, 其中 lowerj 和 upperj 分别是直方图第 i 维上第 j 个方块的上下界（针对输入象素的第 i 个值）。任何情况下，输入值如果超出了一个直方块所指定的范围外，都不会被 cvCalcHist 计数，而且会被函数 cvCalcBackProject 置零。

函数 cvCreateHist 创建一个指定尺寸的直方图，并且返回创建的直方图的指针。 如果数组的 ranges 是 0, 则直方块的范围必须由函数 cvSetHistBinRanges 稍后指定。虽然 cvCalcHist 和 cvCalcBackProject 可以处理 8-比特图像而无需设置任何直方块的范围，但它们都被假设等分 0..255 之间的空间。

SetHistBinRanges

设置直方块的区间

```
void cvSetHistBinRanges( CvHistogram* hist, float** ranges, int
uniform=1 );
```

hist

直方图.

ranges

直方块范围数组的数组，见 cvCreateHist.

uniform

归一化标识，见 cvCreateHist.

函数 cvSetHistBinRanges 是一个独立的函数，完成直方块的区间设置。更多详细的关于参数 ranges 和 uniform 的描述，请参考函数 cvCalcHist ，该函数也可以初始化区间。直方块的区间的设置必须在计算直方图之前，或 在计算直方图的反射图之前。

ReleaseHist

释放直方图结构

```
void cvReleaseHist( CvHistogram** hist );
```

hist

被释放的直方图结构的双指针.

函数 cvReleaseHist 释放直方图 (头和数据). 指向直方图的指针被函数所清空。如果 *hist 指针已经为 NULL, 则函数不做任何事情。

ClearHist

清除直方图

```
void cvClearHist( CvHistogram* hist );
```

hist

直方图.

函数 cvClearHist 当直方图是稠密数组时将所有直方块设置为 0，当直方图是稀疏数组时，除去所有的直方块。

MakeHistHeaderForArray

从数组中创建直方图

```
CvHistogram*  cvMakeHistHeaderForArray( int dims, int* sizes,
CvHistogram* hist,
                                    float* data, float** ranges=NULL, int
uniform=1 );
```

dims

直方图维数.

sizes

直方图维数尺寸的数组

hist

为函数所初始化的直方图头

data

用来存储直方块的数组

ranges

直方块范围，见 cvCreateHist.

uniform

归一化标识，见 cvCreateHist.

函数 cvMakeHistHeaderForArray 初始化直方图，其中头和直方块为用户所分配。以后不需要调用 cvReleaseHist 只有稠密直方图可以采用这种方法，函数返回 hist.

QueryHistValue_1D

查询直方块的值

```
#define cvQueryHistValue_1D( hist, idx0 ) \
    cvGetReal1D( (hist)->bins, (idx0) )
#define cvQueryHistValue_2D( hist, idx0, idx1 ) \
    cvGetReal2D( (hist)->bins, (idx0), (idx1) )
#define cvQueryHistValue_3D( hist, idx0, idx1, idx2 ) \
    cvGetReal3D( (hist)->bins, (idx0), (idx1), (idx2) )
#define cvQueryHistValue_nD( hist, idx ) \
    cvGetRealND( (hist)->bins, (idx) )
```

hist

直方图

idx0, idx1, idx2, idx3

直方块的下标索引

idx

下标数组

宏 cvQueryHistValue_*D 返回 1D, 2D, 3D 或 N-D 直方图的指定直方块的值。对稀疏直方图，如果方块在直方图中不存在，函数返回 0, 而且不创建新的直方块。

GetHistValue_1D

返回直方块的指针

```
#define cvGetHistValue_1D( hist, idx0 ) \
    ((float*)(cvPtr1D( (hist)->bins, (idx0), 0 ))
#define cvGetHistValue_2D( hist, idx0, idx1 ) \
    ((float*)(cvPtr2D( (hist)->bins, (idx0), (idx1), 0 ))
#define cvGetHistValue_3D( hist, idx0, idx1, idx2 ) \
    ((float*)(cvPtr3D( (hist)->bins, (idx0), (idx1), (idx2), 0 ))
#define cvGetHistValue_nD( hist, idx ) \
    ((float*)(cvPtrND( (hist)->bins, (idx), 0 ))
```

hist
　　直方图.
idx0, idx1, idx2, idx3
　　直方块的下标索引.
idx
　　下标数组

宏 cvGetHistValue_*D 返回 1D, 2D, 3D 或 N-D 直方图的指定直方块的指针。对稀疏直方图，函数创建一个新的直方块，且设置其为 0，除非它已经存在。

GetMinMaxHistValue

发现最大和最小直方块

```
void cvGetMinMaxHistValue( const CvHistogram* hist,
                           float* min_value, float* max_value,
                           int* min_idx=NULL, int* max_idx=NULL );
```

hist
　　直方图
min_value
　　直方图最小值的指针
max_value
　　直方图最大值的指针
min_idx
　　数组中最小坐标的指针
max_idx
　　数组中最大坐标的指针

函数 cvGetMinMaxHistValue 发现最大和最小直方块以及它们的位置。任何输出变量都是可选的。在具有同样值几个极值中，返回具有最小下标索引（以字母排列顺序定）的那一个。

NormalizeHist

归一化直方图

```
void cvNormalizeHist( CvHistogram* hist, double factor );
```

hist
　　直方图的指针.
factor
　　归一化因子

函数 cvNormalizeHist 通过缩放来归一化直方块，使得所有块的和等于 factor.

ThreshHist

对直方图取阈值

```
void cvThreshHist( CvHistogram* hist, double threshold );
```

hist
　　直方图的指针.
threshold
　　阈值大小

函数 cvThreshHist 清除那些小于指定阈值得直方块

CompareHist

比较两个稠密直方图

```
double cvCompareHist( const CvHistogram* hist1, const CvHistogram* hist2,
int method );
```

hist1
　　第一个稠密直方图
hist2
　　第二个稠密直方图
method
　　比较方法，采用其中之一：

- CV_COMP_CORREL
- CV_COMP_CHISQR
- CV_COMP_INTERSECT
- CV_COMP_BHATTACHARYYA

函数 cvCompareHist 采用下面指定的方法比较两个稠密直方图(H_1 表示第一个， H_2表示第二个):

- Correlation (method=CV_COMP_CORREL):

$$d(H1,H2) = \sum_i \frac{H'_1(i) \cdot H'_2(i)}{\sqrt{(\sum_j H'_1(j)^2) \cdot (\sum_j H'_2(j)^2)}}$$

其中

$$H'_k(i) = H_k(i) - \frac{1}{N}\sum_j H_k(j)$$ (N 是 number of histogram bins)

- Chi-square(method=CV_COMP_CHISQR):

$$d(H_1,H_2) = \sum_i \frac{H_1(i) - H_2(i)}{H_1(i) + H_2(i)}$$

- 交叉 (method=CV_COMP_INTERSECT):

$$d(H_1,H_2) = \sum_i \min(H_1(i),H_2(i))$$

- Bhattacharyya 距离 (method=CV_COMP_BHATTACHARYYA):

$$d(H_1,H_2) = \sqrt{1 - \sum_i \sqrt{H_1(i) \cdot H_2(i)}}$$

函数返回 d(H1,H2) 的值。

注意：Bhattacharyya 距离只能应用到规一化后的直方图。

为了比较稀疏直方图或更一般的加权稀疏点集(译者注：直方图匹配是图像检索中的常用方法)，考虑使用函数 cvCalcEMD 。

CopyHist

拷贝直方图

```
void cvCopyHist( const CvHistogram* src, CvHistogram** dst );
```

src
　　输入的直方图
dst
　　输出的直方图指针

函数 cvCopyHist 对直方图作拷贝。如果第二个直方图指针 *dst 是 NULL, 则创建一个与 src 同样大小的直方图。否则，两个直方图必须大小和类型一致。然后函数将输入的直方块的值复制到输出的直方图中，并且设置取值范围与 src 的一致。

CalcHist

计算图像 image(s) 的直方图

```
void cvCalcHist( IplImage** image, CvHistogram* hist,
                 int accumulate=0, const CvArr* mask=NULL );
```

image
　　输入图像 s (虽然也可以使用 CvMat**).
hist
　　直方图指针
accumulate
　　累计标识。如果设置，则直方图在开始时不被清零。这个特征保证可以为多个图像计算一个单独的直方图，或者在线更新直方图。
mask
　　操作 mask, 确定输入图像的哪个象素被计数

函数 cvCalcHist 计算一张或多张**单通道**图像的直方图（译者注：若要计算多通道，可像以下例子那样用多个单通道图来表示）。 用来增加直方块的数组元素可从相应输入图像的同样位置提取。 Sample. 计算和显示彩色图像的 2D 色调－饱和度图像

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv )
{
    IplImage* src;
```

```
    if( argc == 2 && (src=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* v_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* planes[] = { h_plane, s_plane };
        IplImage* hsv = cvCreateImage( cvGetSize(src), 8, 3 );
        int h_bins = 30, s_bins = 32;
        int hist_size[] = {h_bins, s_bins};
        /* hue varies from 0 (~0° red) to 180 (~360° red again) */
        float h_ranges[] = { 0, 180 };
        /* saturation varies from 0 (black-gray-white) to 255 (pure
spectrum color) */
        float s_ranges[] = { 0, 255 };
        float* ranges[] = { h_ranges, s_ranges };
        int scale = 10;
        IplImage* hist_img =
cvCreateImage( cvSize(h_bins*scale,s_bins*scale), 8, 3 );
        CvHistogram* hist;
        float max_value = 0;
        int h, s;

        cvCvtColor( src, hsv, CV_BGR2HSV );
        cvCvtPixToPlane( hsv, h_plane, s_plane, v_plane, 0 );
        hist = cvCreateHist( 2, hist_size, CV_HIST_ARRAY, ranges, 1 );
        cvCalcHist( planes, hist, 0, 0 );
        cvGetMinMaxHistValue( hist, 0, &max_value, 0, 0 );
        cvZero( hist_img );

        for( h = 0; h < h_bins; h++ )
        {
            for( s = 0; s < s_bins; s++ )
            {
                float bin_val = cvQueryHistValue_2D( hist, h, s );
                int intensity = cvRound(bin_val*255/max_value);
                cvRectangle( hist_img, cvPoint( h*scale, s*scale ),
                             cvPoint( (h+1)*scale - 1, (s+1)*scale - 1),
                             CV_RGB(intensity,intensity,intensity),
/* graw a grayscale histogram.
if you have idea how to do it
nicer let us know */
                             CV_FILLED );
            }
        }
```

```
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "H-S Histogram", 1 );
        cvShowImage( "H-S Histogram", hist_img );

        cvWaitKey(0);
    }
}
```

CalcBackProject

计算反向投影

```
void cvCalcBackProject( IplImage** image, CvArr* back_project, const
CvHistogram* hist );
```

image
　　输入图像 (也可以传递 CvMat**).

back_project
　　反向投影图像，与输入图像具有同样类型.

hist
　　直方图

函数 cvCalcBackProject 计算直方图的反向投影. 对于所有输入的单通道图像同一位置的象素数组，该函数根据相应的象素数组(RGB)，放置其对应的直方块的值到输出图像中。用统计学术语，输出图像象素点的值是观测数组在某个分布（直方图）下的概率。 例如，为了发现图像中的红色目标，可以这么做：

1. 对红色物体计算色调直方图，假设图像仅仅包含该物体。则直方图有可能有极值，对应着红颜色。
2. 对将要搜索目标的输入图像，使用直方图计算其色调平面的反向投影，然后对图像做阈值操作。
3. 在产生的图像中发现连通部分，然后使用某种附加准则选择正确的部分，比如最大的连通部分。

这是 Camshift 彩色目标跟踪器中的一个逼进算法，除了第三步，CAMSHIFT 算法使用了上一次目标位置来定位反向投影中的目标。

CalcBackProjectPatch

用直方图比较来定位图像中的模板

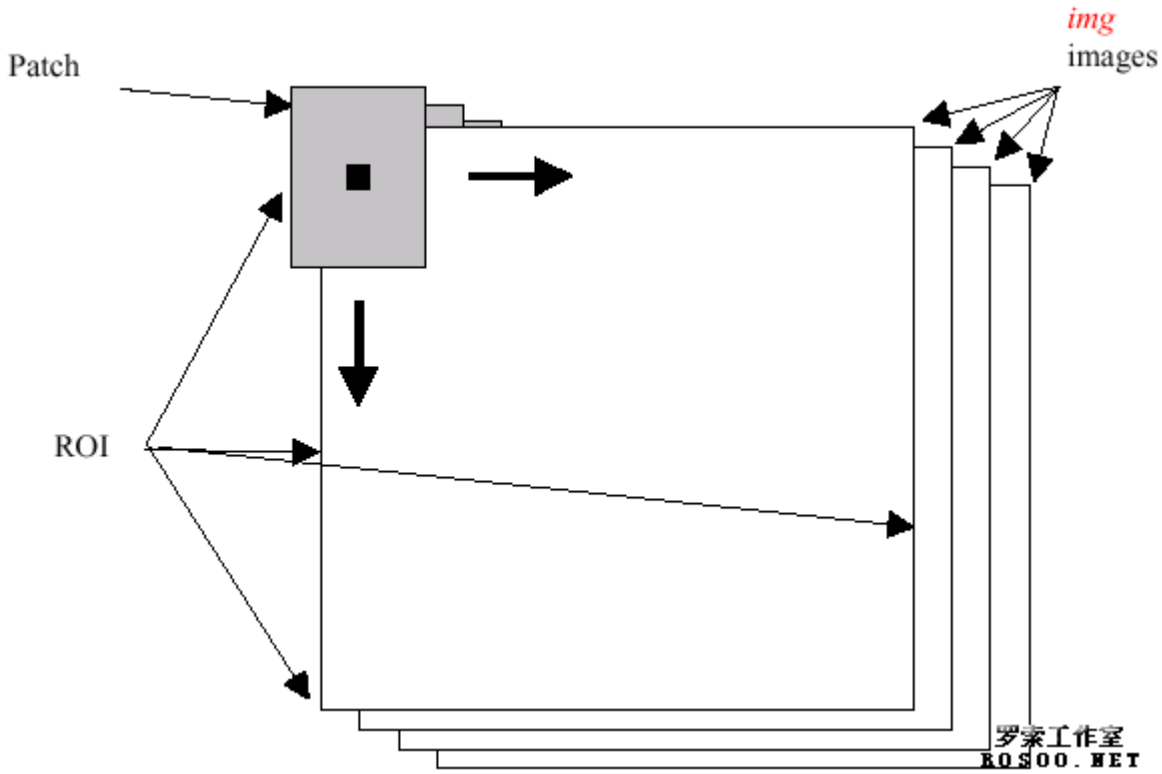
```
void cvCalcBackProjectPatch( IplImage** image, CvArr* dst,
                             CvSize patch_size, CvHistogram* hist,
                             int method, double factor );
```

image
　　输入图像 (可以传递 CvMat**)
dst
　　输出图像.
patch_size
　　扫描输入图像的补丁尺寸
hist
　　直方图
method
　　比较方法，传递给 cvCompareHist (见该函数的描述).
factor
　　直方图的归一化因子，将影响输出图像的归一化缩放。如果为 1，则不定。
　　/*归一化因子的类型实际上是 double，而非 float*/

函数 cvCalcBackProjectPatch 通过输入图像补丁的直方图和给定直方图的比较，来计算反向投影。提取图像在 ROI 中每一个位置的某种测量结果产生了数组 image. 这些结果可以是色调, x 差分, y 差分, Laplacian 滤波器, 有方向 Gabor 滤波器等中 的一个或多个。每种测量输出都被划归为它自己的单独图像。image 图像数组是这些测量图像的集合。一个多维直方图 hist 从这些图像数组中被采样创建。最后直方图被归一化。直方图 hist 的维数通常很大等于图像数组 image 的元素个数。

在选择的 ROI 中，每一个新的图像被测量并且转换为一个图像数组。在以锚点为“补丁”中心的图像 image 区域中计算直方图 （如下图所示）。用参数 norm_factor 来归一化直方图，使得它可以与 hist 互相比较。计算出的直方图与直方图模型互相比较， （hist 使用函数 cvCompareHist ，比较方法是 method=method). 输出结果被放置到概率图像 dst 补丁锚点的对应位置上。这个过程随着补丁滑过整个 ROI 而重复进行。迭代直方图的更新可以通过在原直方图中减除“补丁”已复盖的尾象素点或者加上新复盖的象素点来实现，这种更新方式可以节省大量的操作，尽管目前在函数体中还没有实现。

Back Project Calculation by Patches



CalcProbDensity

两个直方图相除

```
void  cvCalcProbDensity( const CvHistogram* hist1, const CvHistogram*
hist2,
                         CvHistogram* dst_hist, double scale=255 );
```

hist1
　　第一个直方图(分母).
hist2
　　第二个直方图
dst_hist
　　输出的直方图
scale
　　输出直方图的尺度因子

函数 cvCalcProbDensity 从两个直方图中计算目标概率密度：

```
dist_hist(I)=0       if hist1(I)==0
```

```
            scale  if hist1(I)!=0 && hist2(I)>hist1(I)
            hist2(I)*scale/hist1(I) if hist1(I)!=0 &&
hist2(I)<=hist1(I)
```

所以输出的直方块小于尺度因子。

EqualizeHist

灰度图象直方图均衡化

```
void cvEqualizeHist( const CvArr* src, CvArr* dst );
```

src

输入的 8-比特 单信道图像

dst

输出的图像与输入图像大小与数据类型相同

函数 cvEqualizeHist 采用如下法则对输入图像进行直方图均衡化：

1. 计算输入图像的直方图 H
2. 直方图归一化，因此直方块和为 255
3. 计算直方图积分：$$H'(i) = \sum_{0 \le j \le i} H(j)$$
4. 采用 H'作为查询表：dst(x,y)=H'(src(x,y))进行图像变换。

该方法归一化图像亮度和增强对比度。

例：彩色图像的直方图均衡化

```
int i;
IplImage *pImageChannel[4] = { 0, 0, 0, 0 };
pSrcImage = cvLoadImage( "test.jpg", 1 )  ;
IplImage *pImage = cvCreateImage(cvGetSize(pSrcImage),
  pSrcImage->depth, pSrcImage->nChannels);
if( pSrcImage )
{
        for( i = 0; i < pSrcImage->nChannels; i++ )
        {
pImageChannel[i] = cvCreateImage( cvGetSize(pSrcImage),
pSrcImage->depth, 1 );
        }
        // 信道分离
        cvSplit( pSrcImage, pImageChannel[0], pImageChannel[1],
                pImageChannel[2], pImageChannel[3] );
```

```
        for( i = 0; i < pImage->nChannels; i++ )
        {
                // 直方图均衡化
                cvEqualizeHist( pImageChannel[i], pImageChannel[i] );
        }
        // 信道组合
        cvMerge( pImageChannel[0], pImageChannel[1], pImageChannel[2],
                pImageChannel[3], pImage );
        // ……图像显示代码（略）
        // 释放资源
        for( i = 0; i < pSrcImage->nChannels; i++ )
        {
                if ( pImageChannel[i] )
                {
                        cvReleaseImage( pImageChannel[i] );
                        pImageChannel[i] = 0;
                }
        }
        cvReleaseImage( pImage );
        pImage = 0;
}
```

匹配

MatchTemplate

比较模板和重叠的图像区域

```
void cvMatchTemplate( const CvArr* image, const CvArr* templ,
                      CvArr* result, int method );
```

image
　　欲搜索的图像。它应该是单通道、8-比特或 32-比特 浮点数图像
templ
　　搜索模板，不能大于输入图像，且与输入图像具有一样的数据类型
result
　　比较结果的映射图像。单通道、32-比特浮点数. 如果图像是 W×H 而 templ 是 w×h ，则 result 一定是 (W-w+1) ×(H-h+1) .
method
　　指定匹配方法：

函数 cvMatchTemplate 与函数 cvCalcBackProjectPatch 类似。它滑动过整个图像 image, 用指定方法比较 templ 与图像尺寸为 w×h 的重叠区域，并且将

比较结果存到 result 中。下面是不同的比较方法,可以使用其中的一种 (I 表示图像, T - 模板, R - 结果. 模板与图像重叠区域 x'=0..w-1, y'=0..h-1 之间求和):

method=CV_TM_SQDIFF:

$$R(x,y) = \sum_{x',y'} [T(x',y') - I(x+x',y+y')]^2$$

method=CV_TM_SQDIFF_NORMED:

method=CV_TM_CCORR:

$$R(x,y) = \sum_{x',y'} [T(x',y') \cdot I(x+x',y+y')]$$

method=CV_TM_CCORR_NORMED:

$$R(x,y) = \sum_{x',y'} [T(x',y') \cdot I(x+x',y+y')] / \sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}$$

method=CV_TM_CCOEFF:

$$R(x,y) = \sum_{x',y'} [T'(x',y') \cdot I'(x+x',y+y')]$$

其中

$$T'(x',y') = T(x',y') - \frac{1}{w \cdot h} \sum_{x'',y''} T(x'',y'')$$

(mean template brightness=>0)

$$I'(x+x',y+y') = I(x+x',y+y') - \frac{1}{w \cdot h} \sum_{x'',y''} I(x+x'',y+y'')$$

(mean patch brightness=>0)

method=CV_TM_CCOEFF_NORMED:

$$R(x,y) = \sum_{x',y'} [T'(x',y') \cdot I'(x+x',y+y')] / \sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}$$

函数完成比较后，通过使用 cvMinMaxLoc 找全局最小值 CV_TM_SQDIFF*) 或者最大值 (CV_TM_CCORR* and CV_TM_CCOEFF*)。

MatchShapes

比较两个形状

```
double cvMatchShapes( const void* object1, const void* object2,
                      int method, double parameter=0 );
```

object1
: 第一个轮廓或灰度图像

object2
: 第二个轮廓或灰度图像

method
: 比较方法，其中之一 CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3.

parameter
: 比较方法的参数 (目前不用).

函数 cvMatchShapes 比较两个形状。 三个实现方法全部使用 Hu 矩 (见 cvGetHuMoments) (A ~ object1, B - object2):

method=CV_CONTOUR_MATCH_I1:

$$I_1(A,B) = \sum_{i=1}^{7} \left| \frac{1}{m^{A_i}} - \frac{1}{m^{B_i}} \right|$$

method=CV_CONTOUR_MATCH_I2:

$$I_2(A,B) = \sum_{i=1}^{7} \left| m^{A_i} - m^{B_i} \right|$$

method=CV_CONTOUR_MATCH_I3:

$$I_3(A,B) = \sum_{i=1}^{7} \frac{\left| m^{A_i} - m^{B_i} \right|}{\left| m^{A_i} \right|}$$

其中

$$m^{A_i} = sign(h^{A_i}) \cdot \log(h^{A_i}),$$
$$m^{B_i} = sign(h^{B_i}) \cdot \log(h^{B_i}),$$

是 A 和 B 的 Hu 矩.

CalcEMD2

两个加权点集之间计算最小工作距离

```
float cvCalcEMD2( const CvArr* signature1, const CvArr* signature2, int
distance_type,
                  CvDistanceFunction distance_func=NULL, const CvArr*
cost_matrix=NULL,
                  CvArr* flow=NULL, float* lower_bound=NULL, void*
userdata=NULL );
typedef float (*CvDistanceFunction)(const float* f1, const float* f2,
void* userdata);
```

signature1
: 第一个签名，大小为 size1×(dims+1) 的浮点数矩阵，每一行依次存储点的权重和点的坐标。矩阵允许只有一列（即仅有权重），如果使用用户自定义的代价矩阵。

signature2
: 第二个签名，与 signature1 的格式一样 size2×(dims+1)，尽管行数可以不同(列数要相同)。当一个额外的虚拟点加入 signature1 或 signature2 中的时候，权重也可不同。

distance_type
: 使用的准则， CV_DIST_L1, CV_DIST_L2, 和 CV_DIST_C 分别为标准的准则。 CV_DIST_USER 意味着使用用户自定义函数 distance_func 或预先计算好的代价矩阵 cost_matrix 。

distance_func
: 用户自定义的距离函数。用两个点的坐标计算两点之间的距离。

cost_matrix
: 自定义大小为 size1×size2 的代价矩阵。 cost_matrix 和 distance_func 两者至少有一个必须为 NULL. 而且，如果使用代价函数，下边界无法计算，因为它需要准则函数。

flow
: 产生的大小为 size1×size2 流矩阵（flow matrix）: flowij 是从 signature1 的第 i 个点到 signature2 的第 j 个点的流(flow)。

lower_bound
: 可选的输入/输出参数：两个签名之间的距离下边界，是两个质心之间的距离。如果使用自定义代价矩阵，点集的所有权重不等，或者有签名只包含权重（即该签名矩阵只有单独一列），则下边界也许不会计算。用户必

须初始化 *lower_bound. 如果质心之间的距离大于获等于 *lower_bound (这意味着签名之间足够远）， 函数则不计算 EMD. 任何情况下，函数返回时 *lower_bound 都被设置为计算出来的质心距离。因此如果用户想同时计算质心距离和 T EMD, *lower_bound 应该被设置为 0.

userdata

传输到自定义距离函数的可选数据指针

函数 cvCalcEMD2 计算两个加权点集之间的移动距离或距离下界。在 [RubnerSept98] 中所描述的其中一个应用就是图像提取得多维直方图比较。EMD 是一个使用某种单纯形算法（simplex algorithm）来解决的交通问题。其计算复杂度在最坏情况下是指数形式的，但是平均而言它的速度相当快。对实的准则，下边界的计算可以更快（使用线性时间算法），且它可用来粗略确定两个点集是否足够远以至无法联系到同一个目标上