CV 参考手册

HUNNISH 注:

本翻译是直接根据 OpenCV Beta 4.0 版本的用户手册翻译的，原文件是: <opencv_directory>/doc/ref/opencvref_cv.htm，可以从 SOURCEFORG 上面的 OpenCV 项目下载，也可以直接从 阿须数码 中下载: http://www.assuredigit.com/incoming/sourcecode/opencv/chinese_docs/ref/opencvref_cv.htm 。

翻译中肯定有不少错误，另外也有些术语和原文语义理解不透导致翻译不准确或者错误，也请有心人赐教。

图像处理、结构分析、运动分析和对象跟踪部分由R.Z.LIU翻译，模式识别、照相机定标与三维重建部分由H.M.ZHANG翻译，全文由Y.C.WEI统一修改校正。

图像处理

注意:

本章描述图像处理和分析的一些函数。其中大多数函数都是针对两维象素数组的，这里，我们称这些数组为“图像”，但是它们不一定非得是IplImage 结构，也可以是CvMat或者CvMatND结构。

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:

```
|  -3 0  3|
|-10 0 10|
|  -3 0  3|
```

对 x-方向 以及转置矩阵对 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:

$$dst(x,y) = d^{xorder+yoder} src/dx^{xorder} ?dy^{yorder} |_{(x,y)}$$

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:

```
|-1  0  1|
|-2  0  2|
|-1  0  1|
```

核。第二种对应

```
|-1 -2 -1|
| 0  0  0|
| 1  2  1|
```

or

```
| 1  2  1|
| 0  0  0|
|-1 -2 -1|
```

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: 即象素的深度) 。为防止溢出，当输入图像是 8 位的，要求输出图像是 16

位的。当然可以用函数函数 cvConvertScale 或 cvConvertScaleAbs 转换为 8 位的。除了 8-比特 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:

$$dst(x,y) = d^2src/dx^2 + d^2src/dy^2$$

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:

```
|0  1  0|
|1 -4  1|
|0  1  0|
```

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`
输入图像.

`edges`
输出的边缘图像

`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。

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 $D_x^2D_{yy}+D_y^2D_{xx} - 2D_xD_yD_{xy}$ 其中 $D_?$ 表示一阶图像差分，$D_{??}$ 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:

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

然后它计算矩阵的特征值和特征向量，并且按如下方式(λ_1, λ_2, x_1, y_1, x_2, y_2)存储这些值到输出图像中，其中

λ_1, λ_2 - M 的特征值，没有排序

(x_1, y_1) - 特征向量，对 λ_1

(x_2, y_2) - 特征向量，对 λ_2

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(λ_1, λ_2)

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

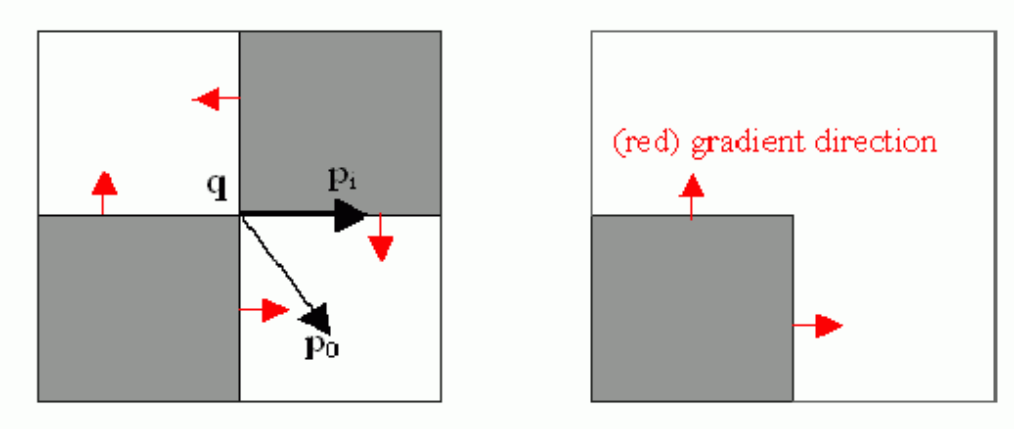
zero_zone

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:

$$\varepsilon_i = DI_{p_i}^T \cdot (q - p_i)$$

其中，DI_{p_i}表示在q的一个邻域点p_i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:

$$sum_i(DI_{p_i} \cdot DI_{p_i}^T) \cdot q - sum_i(DI_{p_i} \cdot DI_{p_i}^T \cdot p_i) = 0$$

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。

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、差值和几何变换

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

将光栅线读入缓冲区

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

`image`

带采线段的输入图像

`pt1`

起点

`pt2`

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。它读取由两点 pt1 和 pt2 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差值提取。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,
                            int fill_outliers=0, CvScalar fill_value=cvScalarAll(0) );
```

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）.

fill_outliers

该标志位指定是否对原始图像边界外面的象素点使用复制模式(fill_outliers=0)进行差值或者将其设置为指定值(fill_outliers=1)。

fill_value

对超出图像边界的矩形象素设定的值(当 fill_outliers=1时的情况).

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:

$$dst(x+width(dst)/2,\ y+height(dst)/2) = src(\ A_{11}x+A_{12}y+b_1,\ A_{21}x+A_{22}y+b_2),$$

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

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

其中在非整数坐标 $A?(x,y)^T+b$ 的象素点值通过双线性变换得到。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`
差值方法:

- CV_INTER_NN - 最近邻差值,
- CV_INTER_LINEAR - 双线性差值 (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 - 立方差值.

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差值。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&apos;,y&apos;)<-src(x,y)
如果没有指定 CV_WARP_INVERSE_MAP ， (x&apos;,y&apos;)^T=map_matrix?(x,y,1)^T+b ，
否则， (x, y)^T=map_matrix?(x&apos;,y&apos,1)^T+b
```

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 。

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，放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差值。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')<-src(x,y)
若指定 CV_WARP_INVERSE_MAP， $(tx',ty',t)^T=map\_matrix?(x,y,1)^T+b$
否则， $(tx, ty, t)^T=map\_matrix?(x',y&apos,1)^T+b$

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:

$(t_i x'_i, t_i y'_i, t_i)^T=matrix?(x_i, y_i, 1)^T$

其中 dst(i)=(x'$_i$,y'$_i$), src(i)=(x$_i$,y$_i$), i=0..3.

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
: 锚点的相对垂直便宜量

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:

$$dst=erode(src,element): \quad dst(x,y)=\min_{((x',y') \text{ in element})} src(x+x',y+y')$$

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): $dst(x,y)=\max_{((x',y') \text{ in element})}src(x+x',y+y')$

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。

滤波器与彩色变换

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 );
```

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.

Filter2D

对图像做卷积

```
void cvFilter2D( const CvArr* src, CvArr* dst,
                 const CvMat* kernel,
                 CvPoint anchor=cvPoint(-1,-1));
#define cvConvolve2D cvFilter2D
```

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差值得到边界外面的象素值。

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:

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

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

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

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0(1)。例如:

$$sum_{x1<=x<x2,y1<=y<y2}image(x,y)=sum(x2,y2)-sum(x1,y2)-sum(x2,y1)+sum(x1,x1)$$

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-比特 或浮点图像.

dst
输出的 8-比特 或浮点图像.

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 B_0 G_0 R_0 B_1 G_1 R_1 ... 层叠的24-位格式，而 RGB 意味着布局为 R_0 G_0 B_0 R_1 G_1 B_1 ... 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)

```
|X|   |0.433910  0.376220  0.189860| |R/255|
|Y| = |0.212649  0.715169  0.072182|*|G/255|
|Z|   |0.017756  0.109478  0.872915| |B/255|

L = 116*Y^1/3      for Y>0.008856
L = 903.3*Y      for Y<=0.008856

a = 500*(f(X)-f(Y))
b = 200*(f(Y)-f(Z))
where f(t)=t^1/3              for t>0.008856
```

```
f(t)=7.787*t+16/116    for t<=0.008856
```

上面的公式可以参考

http://www.cica.indiana.edu/cica/faq/color_spaces/color.spaces.html

- 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

对象素输出的RGB分量由该象素的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：颜色分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-比特 of 32-比特 浮点数）.

`dst`

输出数组，必须与 `src` 的类型一致，或者为 8-比特.

`threshold`

阈值

`max_value`

使用 `CV_THRESH_BINARY` 和 `CV_THRESH_BINARY_INV` 的最大值.

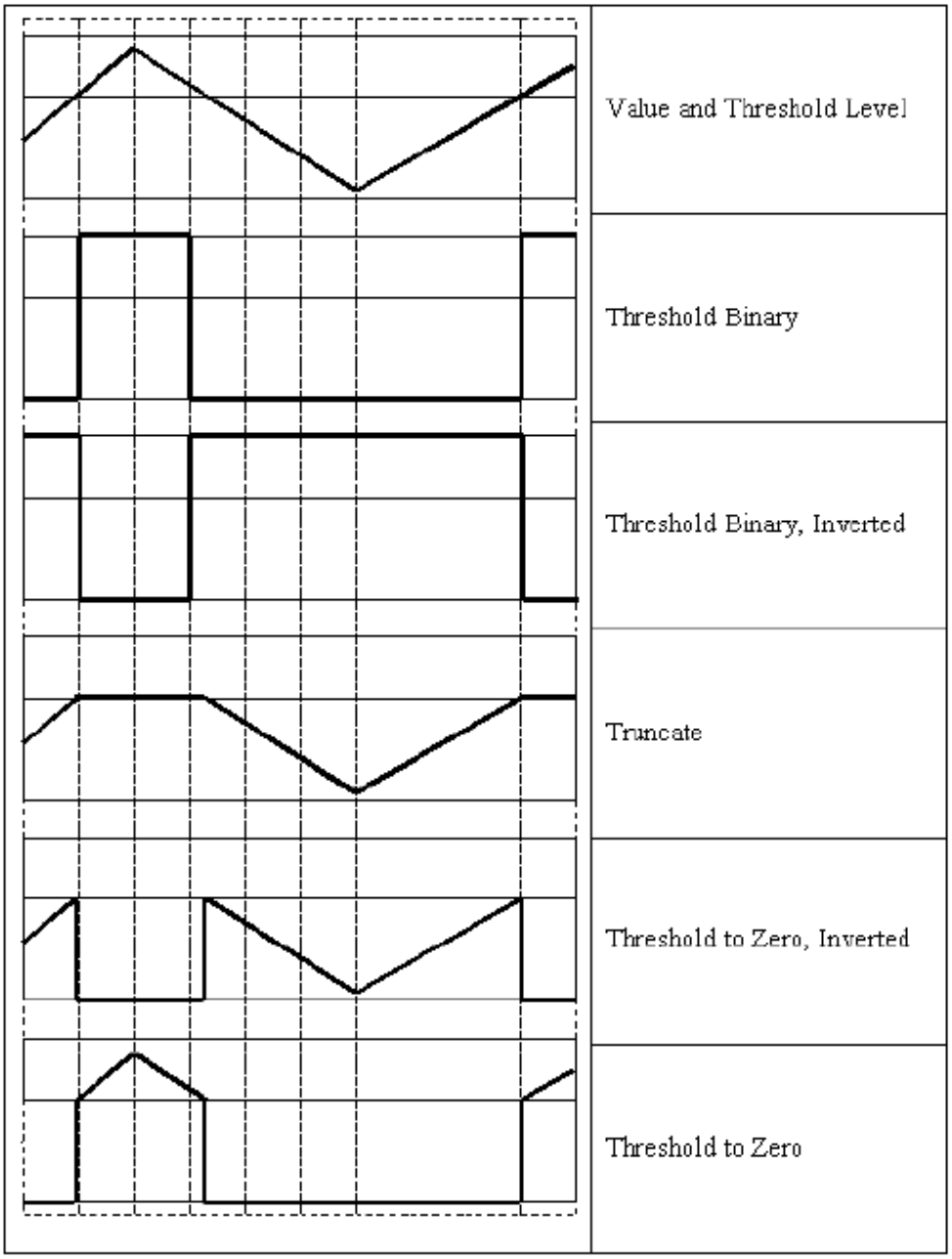
`threshold_type`

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 T_I 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

filter

CV GAUSSIAN 5x5

卷积滤波器的类型，目前仅支持

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做差值。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)

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

$$p(c1,c2)=|c1-c2|.$$

如果输入图像有单个通道(红、绿、兰),那么

$$p(c1,c2)=0,3\cdot(c1_r-c2_r)+0,59\cdot(c1_g-c2_g)+0,11\cdot(c1_b-c2_b)\ .$$

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

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,除非你选则 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`)，但填充面具图像 (这种情况下 MASK 必须是非空的).

`mask`

运算面具，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，则函数使用且更新面具，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。连通性有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:

$src(x',y')-lo\_diff \le src(x,y) \le src(x',y')+up\_diff$, 灰度图像，浮动范围
$src(seed.x,seed.y)-lo \le src(x,y) \le src(seed.x,seed.y)+up\_diff$, 灰度图像，固定范围

$src(x',y')_r-lo\_diff_r \le src(x,y)_r \le src(x',y')_r+up\_diff_r$ 和
$src(x',y')_g-lo\_diff_g \le src(x,y)_g \le src(x',y')_g+up\_diff_g$ 和
$src(x',y')_b-lo\_diff_b \le src(x,y)_b \le src(x',y')_b+up\_diff_b$, 彩色图像，浮动范围

$src(seed.x,seed.y)_r-lo\_diff_r \le src(x,y)_r \le src(seed.x,seed.y)_r+up\_diff_r$ 和
$src(seed.x,seed.y)_g-lo\_diff_g \le src(x,y)_g \le src(seed.x,seed.y)_g+up\_diff_g$ 和
$src(seed.x,seed.y)_b-lo\_diff_b \le src(x,y)_b \le src(seed.x,seed.y)_b+up\_diff_b$, 彩色图像，固定范围

`src(x',y')`

其中 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 使用以提取其余的轮廓。

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。

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ROI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:

$$M_{x\_order,y\_order}=sum_{x,y}(I(x,y)?x^{x\_order}?y^{y\_order})$$

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：

$\mu_{x\_order,y\_order}=\sum_{x,y}(I(x,y)?(x-x_c)^{x\_order}?(y-y_c)^{y\_order})$,

其中 $x_c=M_{10}/M_{00}$, $y_c=M_{01}/M_{00}$ - 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：

$\eta_{x\_order,y\_order}= \mu_{x\_order,y\_order}/M_{00}^{((y\_order+x\_order)/2+1)}$

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：

$h_1=\eta_{20}+\eta_{02}$

$h_2=(\eta_{20}-\eta_{02})2+4\eta_{11}2$

$h_3=(\eta_{30}-3\eta_{12})2+ (3\eta_{21}-\eta_{03})2$

$h_4=(\eta_{30}+\eta_{12})2+ (\eta_{21}+\eta_{03})2$

$h_5=(\eta_{30}-3\eta_{12})(\eta_{30}+\eta_{12})[(\eta_{30}+\eta_{12})2-3(\eta_{21}+\eta_{03})2]+(3\eta_{21}-\eta_{03})(\eta_{21}+\eta_{03})[3(\eta_{30}+\eta_{12})2-(\eta_{21}+\eta_{03})2]$

$h_6=(\eta_{20}-\eta_{02})[(\eta_{30}+\eta_{12})2- (\eta_{21}+\eta_{03})2]+4\eta_{11}(\eta_{30}+\eta_{12})(\eta_{21}+\eta_{03})$

$h_7=(3\eta_{21}-\eta_{03})(\eta_{21}+\eta_{03})[3(\eta_{30}+\eta_{12})2-(\eta_{21}+\eta_{03})2]-(\eta_{30}-3\eta_{12})(\eta_{21}+\eta_{03})[3(\eta_{30}+\eta_{12})2-(\eta_{21}+\eta_{03})2]$

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

合会因为反射而改变。

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，其内容可能被函数所改变

`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(线段没有按照长度、可信度或其它指标排序).

`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 是点与原点 (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.

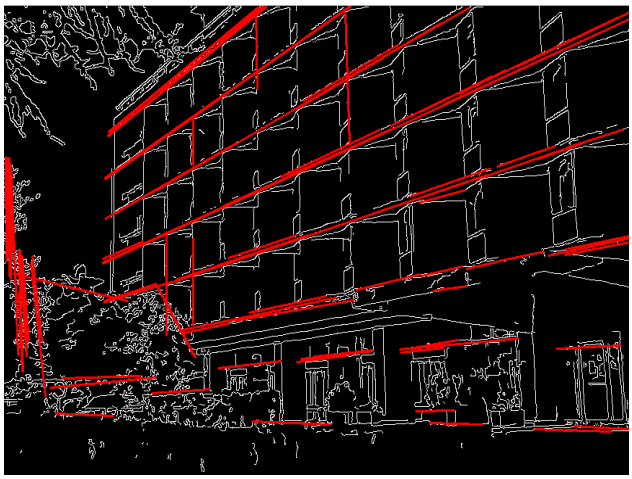
Example. 用 Hough transform 检测线段

```
/* 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;
        cvCanny( src, 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):



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。

直方图

CvHistogram

多维直方图

```
typedef struct CvHistogram
{
    int header_size; /* 头尺寸 */
    CvHistType type; /* 直方图类型 */
    int flags; /* 直方图标识 */
    int c_dims; /* 直方图维数 */
    int dims[CV_HIST_MAX_DIM]; /* 每一维的尺寸 */
    int mdims[CV_HIST_MAX_DIM]; /* 快速访问元素的系数 */
    /* &m[a,b,c] = m + a*mdims[0] + b*mdims[1] + c*mdims[2] */
    float* thresh[CV_HIST_MAX_DIM]; /* 每一维的直方块边界数组 */
    float* array; /* 所有的直方图数据, 扩展为单行 */
struct CvNode* root; /* 存储直方块的平衡树的根结点 */
    CvSet* set; /* 内存存储仓的指针 (对平衡树而言) */
    int* chdims[CV_HIST_MAX_DIM]; /* 快速计算的缓存 */
} CvHistogram;
```

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$lower_0$, $upper_0$, $lower_1$, $upper_1$ == $lower_2$, ..., $upper_{dims[i]-1}$，其中$lower_j$ 和$upper_j$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

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 - 第二个):

相关 (method=CV_COMP_CORREL):
$d(H_1,H_2)=sum_I(H'_1(I)?H'_2(I))/sqrt(sum_I[H'_1(I)^2]?sum_I[H'_2(I)^2])$
其中
$H'_k(I)=H_k(I)-1/N?sum_J H_k(J)$ (N=number of histogram bins)

Chi-square(method=CV_COMP_CHISQR):
$d(H_1,H_2)=sum_I[(H_1(I)-H_2(I))/(H_1(I)+H_2(I))]$

交叉 (method=CV_COMP_INTERSECT):
$d(H_1,H_2)=sum_I max(H_1(I),H_2(I))$

函数返回 $d(H_1,H_2)$ 的值。

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 计算单通道或多通道图像的直方图。 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};
        float h_ranges[] = { 0, 180 }; /* hue varies from 0 (~0°red) to 180
(~360°red again) */
        float s_ranges[] = { 0, 255 }; /* saturation varies from 0 (black-
gray-white) to 255 (pure spectrum color) */
        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 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，比如最大的连同部分。

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, float 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）.

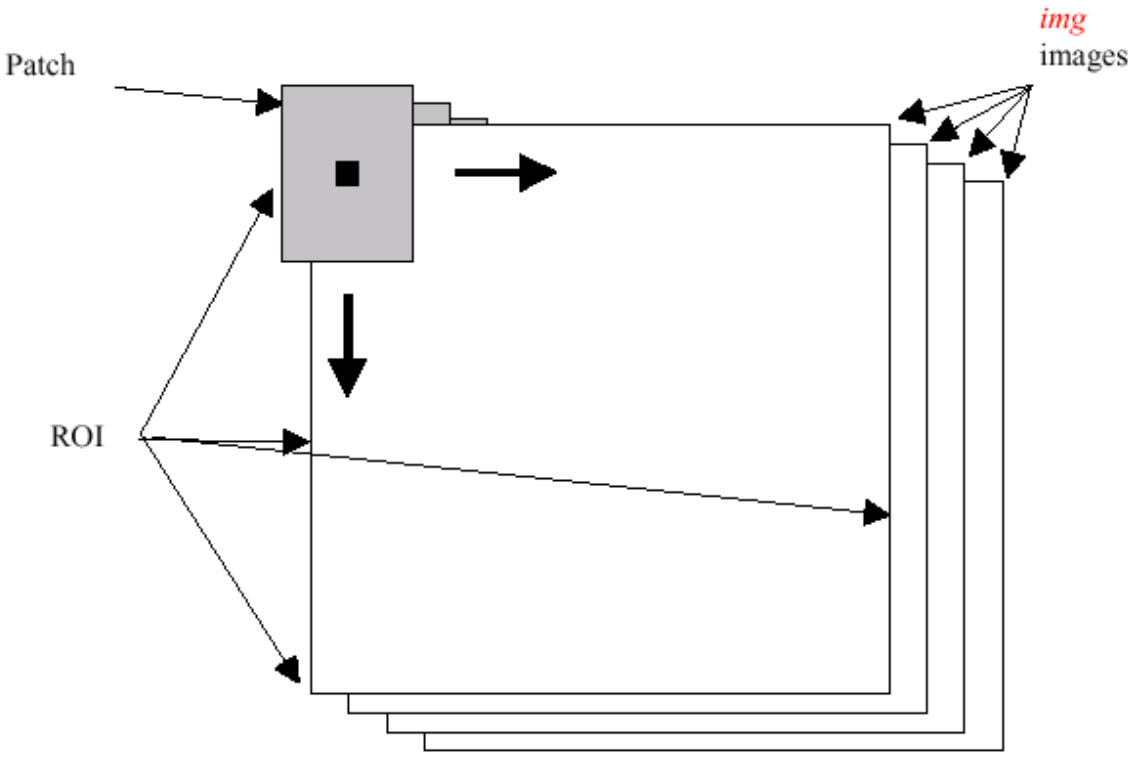
factor

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。

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”已覆盖的尾象素点或者加上新覆盖的象素点来实现，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(分子).

`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。

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

method=CV_TM_CCORR:
$R(x,y)=sum_{x',y'}[T(x',y')?I(x+x',y+y')]$

method=CV_TM_CCORR_NORMED:
$R(x,y)=sum_{x',y'}[T(x',y')?I(x+x',y+y')]/sqrt[sum_{x',y'}T(x',y')^2?sum_{x',y'}I(x+x',y+y')^2]$

method=CV_TM_CCOEFF:
$R(x,y)=sum_{x',y'}[T'(x',y')?I'(x+x',y+y')]$,

where $T'(x',y')=T(x',y') - 1/(w?h)?sum_{x",y"}T(x",y")$ (mean template brightness=>0)
$I'(x+x',y+y')=I(x+x',y+y') - 1/(w?h)?sum_{x",y"}I(x+x",y+y")$ (mean patch brightness=>0)

method=CV_TM_CCOEFF_NORMED:
$R(x,y)=sum_{x',y'}[T'(x',y')?I'(x+x',y+y')]/sqrt[sum_{x',y'}T'(x',y')^2?sum_{x',y'}I'(x+x',y+y')^2]$

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:
$I_1(A,B)=sum_{i=1..7}abs(1/m^A_i - 1/m^B_i)$

method=CV_CONTOUR_MATCH_I2:
$I_2(A,B)=sum_{i=1..7}abs(m^A_i - m^B_i)$

method=CV_CONTOUR_MATCH_I3:
$I_3(A,B)=\sum_{i=1..7}abs(m^A_i - m^B_i)/abs(m^A_i)$

其中
$m^A_i=sign(h^A_i)?log(h^A_i)$,
$m^B_i=sign(h^B_i)?log(h^B_i)$,
h^A_i, h^B_i - 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) : $flow_{ij}$ 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。

结构分析

轮廓处理函数

ApproxChains

用多边形曲线逼近 ***Freeman*** 链

```
CvSeq* cvApproxChains( CvSeq* src_seq, CvMemStorage* storage,
                       int method=CV_CHAIN_APPROX_SIMPLE,
                       double parameter=0, int minimal_perimeter=0, int
recursive=0 );
```

src_seq
涉及其它链的链指针

storage
存储多边形线段位置的缓存

method
逼近方法 (见函数 cvFindContours 的描述).

parameter
方法参数(现在不用).

minimal_perimeter
仅逼近周长大于 minimal_perimeter 轮廓。其它的链从结果中除去。

recursive

如果非 0，函数从 src_seq 中利用 h_next 和 v_next links 连接逼近所有可访问的链。如果为 0，则仅逼近单链。

这是一个单独的逼近程序。 对同样的逼近标识，函数 cvApproxChains 与 cvFindContours 的工作方式一模一样。它返回发现的第一个轮廓的指针。其它的逼近模块，可以用返回结构中的 v_next 和 v_next 域来访问

StartReadChainPoints

初始化链读取

```
void cvStartReadChainPoints( CvChain* chain, CvChainPtReader* reader );
```

chain

链的指针

reader

链的读取状态

函数 cvStartReadChainPoints 初始化一个特殊的读取器（参考 Dynamic Data Structures 以获得关于集合与序列的更多内容）.

ReadChainPoint

得到下一个链的点

```
CvPoint cvReadChainPoint( CvChainPtReader* reader );
```

reader

链的读取状态

函数 cvReadChainPoint 返回当前链的点，并且更新读取位置。

ApproxPoly

用指定精度逼近多边形曲线

```
CvSeq* cvApproxPoly( const void* src_seq, int header_size, CvMemStorage*
storage,
                     int method, double parameter, int parameter2=0 );
```

src_seq

点集数组序列

header_size

逼近曲线的头尺寸

storage
逼近轮廓的容器。如果为 NULL， 则使用输入的序列

method
逼近方法。目前仅支持 CV_POLY_APPROX_DP ， 对应 Douglas-Peucker 算法.

parameter
方法相关参数。对 CV_POLY_APPROX_DP 它是指定的逼近精度

parameter2
如果 src_seq 是序列，它表示要么逼近单个序列，要么在 src_seq 的同一个或低级层次上逼近所有序列（参考 cvFindContours 中对轮廓继承结构的描述).

如果 src_seq 是点集的数组（CvMat*） ， 参数指定曲线是闭合（parameter2!=0）还是非闭合（parameter2=0).

函数 cvApproxPoly 逼近一个或多个曲线，并返回逼近结果。对多个曲线的逼近，生成的树将与输入的具有同样的结构。(1:1 的对应关系).

BoundingRect

计算点集的最外面（***up-right***）矩形边界

```
CvRect cvBoundingRect( CvArr* points, int update=0 );
```

points
二维点集，点的序列或向量（CvMat）

update
更新标识。下面是轮廓类型和标识的一些可能组合:

- update=0, contour – CvContour*: 不计算矩形边界，但直接由轮廓头的 rect 域得到。
- update=1, contour – CvContour*: 计算矩形边界，而且将结果写入到轮廓头的 rect 域中 header.
- update=0, contour – CvSeq* or CvMat*: 计算并返回边界矩形
- update=1, contour – CvSeq* or CvMat*: 产生运行错误 (runtime error is raised)

函数 cvBoundingRect 返回二维点集的最外面 (up-right) 矩形边界。

ContourArea

计算整个轮廓或部分轮廓的面积

```
double cvContourArea( const CvArr* contour, CvSlice slice=CV_WHOLE_SEQ );
```

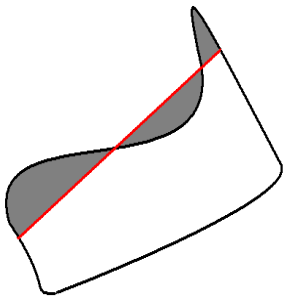
contour

轮廓（定点的序列或数组）.

`slice`

感兴趣轮廓部分的起始点，缺省是计算整个轮廓的面积。

函数 cvContourArea 计算整个轮廓或部分轮廓的面积。 对后面的情况，面积表示轮廓部分和起始点连线构成的封闭部分的面积。如下图所示:



NOTE: 轮廓的方向影响面积的符号。因此函数也许会返回负的结果。应用函数 `fabs()` 得到面积的绝对值。

ArcLength

计算轮廓周长或曲线长度

```
double cvArcLength( const void* curve, CvSlice slice=CV_WHOLE_SEQ, int
is_closed=-1 );
```

`curve`

曲线点集序列或数组

`slice`

曲线的起始点，缺省是计算整个曲线的长度

`is_closed`

表示曲线是否闭合，有三种情况:

- is_closed=0 – 假设曲线不闭合
- is_closed>0 – 假设曲线闭合
- is_closed<0 – 若曲线是序列，检查 ((CvSeq*)curve)->flags 中的标识 CV_SEQ_FLAG_CLOSED 来确定曲线是否闭合。否则（曲线由点集的数组 (CvMat*) 表示）假设曲线不闭合。

函数 cvArcLength 通过依次计算序列点之间的线段长度，并求和来得到曲线的长度。

CreateContourTree

创建轮廓的继承表示形式

```
CvContourTree* cvCreateContourTree( const CvSeq* contour, CvMemStorage*
storage, double threshold );
```

`contour`
输入的轮廓

`storage`
输出树的容器

`threshold`
逼近精度

函数 cvCreateContourTree 为输入轮廓 `contour` 创建一个二叉树，并返回树根的指针。如果参数 `threshold` 小于或等于 0 ,则函数创建一个完整的二叉树。如果 `threshold` 大于 0 ， 函数用 `threshold` 指定的精度创建二叉树：如果基线的截断区域顶点小于threshold，该数就停止生长并作为函数的最终结果返回。

ContourFromContourTree

由树恢复轮廓

```
CvSeq* cvContourFromContourTree( const CvContourTree* tree, CvMemStorage*
storage,
                                 CvTermCriteria criteria );
```

`tree`
轮廓树

`storage`
重构的轮廓容器

`criteria`
停止重构的准则

函数 cvContourFromContourTree 从二叉树恢复轮廓。参数 `criteria` 决定了重构的精度和使用树的数目及层次。所以它可建立逼近的轮廓。 函数返回重构的轮廓。

MatchContourTrees

用树的形式比较两个轮廓

```
double cvMatchContourTrees( const CvContourTree* tree1, const CvContourTree*
tree2,
                            int method, double threshold );
```

`tree1`
第一个轮廓树

`tree2`
第二个轮廓树

`method`
相似度。仅支持 `CV_CONTOUR_TREES_MATCH_I1` 。

`threshold`
相似度阈值

函数 cvMatchContourTrees 计算两个轮廓树的匹配值。从树根开始通过逐层比较来计算相似度。如果某层的相似度小于 threshold，则中断比较过程，且返回当前的差值。

计算几何

MaxRect

对两个给定矩形，寻找矩形边界

```
CvRect cvMaxRect( const CvRect* rect1, const CvRect* rect2 );
```

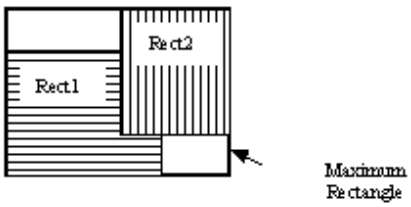
rect1

第一个矩形

rect2

第二个矩形

函数 cvMaxRect 寻找包含两个输入矩形的具有最小面积的矩形边界。



CvBox2D

旋转的二维盒子

```
typedef struct CvBox2D
{
    CvPoint2D32f center;  /* 盒子的中心 */
    CvSize2D32f  size;    /* 盒子的长和宽 */
    float angle;          /* 水平轴与第一个边的夹角，用弧度表示*/
}
CvBox2D;
```

BoxPoints

寻找盒子的顶点

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] );
```

box

盒子

pt

顶点数组

函数 cvBoxPoints 计算输入的二维盒子的定点。下面是函数代码:

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    float a = (float)cos(box.angle)*0.5f;
    float b = (float)sin(box.angle)*0.5f;

    pt[0].x = box.center.x - a*box.size.height - b*box.size.width;
    pt[0].y = box.center.y + b*box.size.height - a*box.size.width;
    pt[1].x = box.center.x + a*box.size.height - b*box.size.width;
    pt[1].y = box.center.y - b*box.size.height - a*box.size.width;
    pt[2].x = 2*box.center.x - pt[0].x;
    pt[2].y = 2*box.center.y - pt[0].y;
    pt[3].x = 2*box.center.x - pt[1].x;
    pt[3].y = 2*box.center.y - pt[1].y;
}
```

FitEllipse

二维点集的椭圆拟合

```
CvBox2D cvFitEllipse2( const CvArr* points );
```

`points`

点集的序列或数组

函数 cvFitEllipse 对给定的一组二维点集作椭圆的最佳拟合(最小二乘意义上的)。返回的结构与 cvEllipse 中的意义类似，除了 `size` 表示椭圆轴的整个长度，而不是一半长度。

FitLine

2D 或 ***3D*** 点集的直线拟合

```
void  cvFitLine( const CvArr* points, int dist_type, double param,
                 double reps, double aeps, float* line );
```

`points`

2D 或 3D 点集，32-比特整数或浮点数坐标

`dist_type`

拟合的距离类型 (见讨论).

`param`

对某些距离的数字参数，如果是 0， 则选择某些最优值

`reps, aeps`

半径 (坐标原点到直线的距离) 和角度的精度，一般设为0.01。

`line`

输出的直线参数。2D 拟合情况下，它是包含 4 个浮点数的数组 `(vx, vy, x0, y0)`，其中 `(vx, vy)` 是线的单位向量而 `(x0, y0)` 是线上的某个点. 对 3D 拟合，它是包含 6 个浮点数的数组 `(vx, vy, vz, x0, y0, z0)`， 其中 `(vx, vy, vz)` 是线的单位向量，而 `(x0, y0, z0)` 是线上某点。

函数 cvFitLine 通过求 $\text{sum}_i \rho(r_i)$ 的最小值方法，用 2D 或 3D 点集拟合直线，其中 r_i 是第 i 个点到直线的距离， $\rho(r)$ 是下面的距离函数之一：

```
dist_type=CV_DIST_L2 (L2):
ρ(r)=r^2/2 (最简单和最快的最小二乘法)

dist_type=CV_DIST_L1 (L1):
ρ(r)=r

dist_type=CV_DIST_L12 (L1-L2):
ρ(r)=2?[sqrt(1+r^2/2) - 1]

dist_type=CV_DIST_FAIR (Fair):
ρ(r)=C^2?[r/C - log(1 + r/C)],  C=1.3998

dist_type=CV_DIST_WELSCH (Welsch):
ρ(r)=C^2/2?[1 - exp(-(r/C)^2)],  C=2.9846

dist_type=CV_DIST_HUBER (Huber):
ρ(r)= r^2/2,   if r < C
      C?(r-C/2),   otherwise;   C=1.345
```

ConvexHull2

发现点集的凸外形

```
CvSeq* cvConvexHull2( const CvArr* input, void* hull_storage=NULL,
                      int orientation=CV_CLOCKWISE, int return_points=0 );
```

`points`

2D 点集的序列或数组，32-比特整数或浮点数坐标

`hull_storage`

输出的数组(CvMat*) 或内存缓存 (CvMemStorage*)，用以存储凸外形。 如果是数组，则它应该是一维的，而且与输入的数组/序列具有同样数目的元素。输出时修改头使得数组裁减到外形的尺寸。输出时，通过修改头结构将数组裁减到凸外形的尺寸。

`orientation`

凸外形的旋转方向：逆时针或顺时针 (`CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`)

`return_points`

如果非零，点集将以外形 (hull) 存储，而不是 `hull_storage` 为数组情况下的顶点形式 (indices) 以及 `hull_storag` 为内存存储模式下的点集形式 (points)。

函数 cvConvexHull2 使用 Sklansky 算法计算 2D 点集的凸外形。如果 `hull_storage` 是内存存储仓， 函数根据 `return_points` 的值，创建一个包含外形的点集或指向这些点的指针的序列。

例子. 由点集序列或数组创建凸外形

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

#define ARRAY  0 /* switch between array/sequence method by replacing 0<=>1
*/

void main( int argc, char** argv )
{
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    cvNamedWindow( "hull", 1 );

#if !ARRAY
        CvMemStorage* storage = cvCreateMemStorage();
#endif

    for(;;)
    {
        int i, count = rand()%100 + 1, hullcount;
        CvPoint pt0;
#if !ARRAY
        CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2,
sizeof(CvContour),
                                    sizeof(CvPoint), storage );
        CvSeq* hull;

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            cvSeqPush( ptseq, &pt0 );
        }
        hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
        hullcount = hull->total;
#else
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
        int* hull = (int*)malloc( count * sizeof(hull[0]));
        CvMat point_mat = cvMat( 1, count, CV_32SC2, points );
        CvMat hull_mat = cvMat( 1, count, CV_32SC1, hull );

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            points[i] = pt0;
        }
        cvConvexHull2( &point_mat, &hull_mat, CV_CLOCKWISE, 0 );
        hullcount = hull_mat.cols;
#endif
        cvZero( img );
        for( i = 0; i < count; i++ )
        {
#if !ARRAY
            pt0 = *CV_GET_SEQ_ELEM( CvPoint, ptseq, i );
#else
            pt0 = points[i];
#endif
            cvCircle( img, pt0, 2, CV_RGB( 255, 0, 0 ), CV_FILLED );
        }

#if !ARRAY
        pt0 = **CV_GET_SEQ_ELEM( CvPoint*, hull, hullcount - 1 );
#else
        pt0 = points[hull[hullcount-1]];
#endif

        for( i = 0; i < hullcount; i++ )
        {
#if !ARRAY
            CvPoint pt = **CV_GET_SEQ_ELEM( CvPoint*, hull, i );
#else
            CvPoint pt = points[hull[i]];
#endif
            cvLine( img, pt0, pt, CV_RGB( 0, 255, 0 ));
            pt0 = pt;
        }

        cvShowImage( "hull", img );

        int key = cvWaitKey(0);
```

```
        if( key == 27 ) // 'ESC'
            break;

#if !ARRAY
        cvClearMemStorage( storage );
#else
        free( points );
        free( hull );
#endif
    }
}
```

CheckContourConvexity

测试轮廓的凸性

```
int cvCheckContourConvexity( const CvArr* contour );
```

contour

被测试轮廓（点序列或数组）.

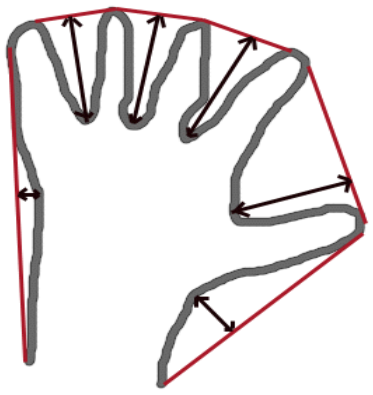
函数 cvCheckContourConvexity 输入的轮廓是否为凸的。必须是简单轮廓，比如没有自交叉。

CvConvexityDefect

用来描述一个简单轮廓凸性缺陷的结构体

```
typedef struct CvConvexityDefect
{
    CvPoint* start; /* 缺陷开始的轮廓点 */
    CvPoint* end; /* 缺陷结束的轮廓点 */
    CvPoint* depth_point; /* 缺陷中距离凸形最远的轮廓点(谷底) */
    float depth; /* 谷底距离凸形的深度*/
} CvConvexityDefect;
```

Picture. Convexity defects of hand contour.



ConvexityDefects

发现轮廓凸形缺陷

```
CvSeq* cvConvexityDefects( const CvArr* contour, const CvArr* convexhull,
                           CvMemStorage* storage=NULL );
```

contour

输入轮廓

convexhull
用 cvConvexHull2 得到的凸外形，它应该包含轮廓的定点或下标，而不是外形点的本身，即cvConvexHull2 中的参数 return_points 应该设置为 0.

storage
凸性缺陷的输出序列容器。如果为 NULL，使用轮廓或外形的存储仓。

函数 cvConvexityDefects 发现输入轮廓的所有凸性缺陷，并且返回 CvConvexityDefect 结构序列。

MinAreaRect2

对给定的 ***2D*** 点集，寻找最小面积的包围矩形

```
CvBox2D  cvMinAreaRect2( const CvArr* points, CvMemStorage* storage=NULL );
```

points
点序列或点集数组

storage
可选的临时存储仓

函数 cvMinAreaRect2 通过建立凸外形并且旋转外形以寻找给定 2D 点集的最小面积的包围矩形.

Picture. Minimal-area bounding rectangle for contour

MinEnclosingCircle

对给定的 ***2D*** 点集，寻找最小面积的包围圆形

```
int cvMinEnclosingCircle( const CvArr* points, CvPoint2D32f* center, float*
radius );
```

points
点序列或点集数组

center
输出参数：圆心

radius
输出参数：半径

函数 cvMinEnclosingCircle 对给定的 2D 点集迭代寻找最小面积的包围圆形。如果

产生的圆包含所有点，返回非零。否则返回零（算法失败）。

CalcPGH

计算轮廓的 ***pair-wise*** 几何直方图

```
void cvCalcPGH( const CvSeq* contour, CvHistogram* hist );
```

`contour`

输入轮廓，当前仅仅支持具有整数坐标的点集

`hist`

计算出的直方图，必须是两维的。

函数 cvCalcPGH 计算轮廓的 2D pair-wise (Hunnish: 不知如何翻译，只好保留) 几何直方图 (pair-wise geometrical histogram : PGH)，算法描述见 [Iivarinen97]. 算法考虑的每一对轮廓边缘。计算每一对边缘之间的夹角以及最大最小距离。具体做法是，轮流考虑每一个边缘做为基准，函数循环遍历所有边缘。在考虑基准边缘和其它边缘的时候， 选择非基准线上的点到基准线上的最大和最小距离。边缘之间的角度定义了直方图的行，而在其中增加对应计算出来的最大和最小距离的所有直方块，(即直方图是 [Iivarninen97] 定义中的转置). 该直方图用来做轮廓匹配。

平面划分

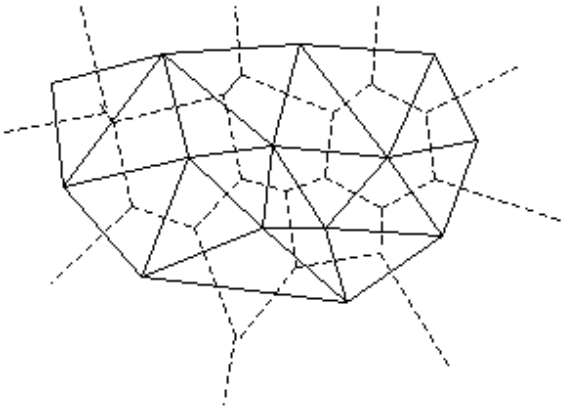
CvSubdiv2D

平面划分

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
CvSubdiv2D;
```

平面划分是将一个平面分割为一组互不重叠的能够覆盖整个平面的区域P(facets)。上面结构描述了建立在 2D 点集上的划分结构，其中点集互相连接并且构成平面图形，该图形通过结合一些无限连接外部划分点(称为凸形点)的边缘，将一个平面用边按照其边缘划分成很多小区域(facets)。

对于每一个划分操作，都有一个对偶划分与之对应，对偶的意思是小区域和点(划分的顶点)变换角色，即在对偶划分中，小区域被当做一个顶点(以下称之为虚拟点)，而原始的划分顶点被当做小区域。在如下所示的图例中，原始的划分用实线来表示，而对偶划分用点线来表示。



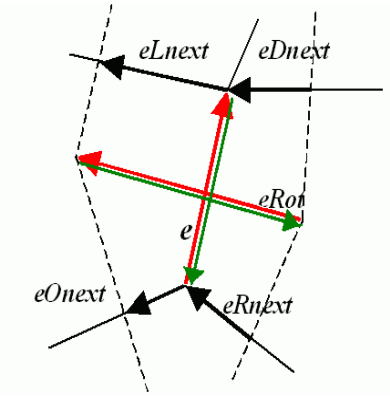
OpenCV 使用Delaunay's 算法将平面分割成小的三角形区域。分割的实现通过从一个假定的三角形(该三角形确保包括所有的分割点)开始不断迭代来完成。在这种情况下，对偶划分就是输入的2d点集的 Voronoi图表。这种划分可以用于对一个平面的3d分段变换、形态变换、平面点的快速定位以及建立特定的图结构（比如 NNG,RNG等等)。

CvQuadEdge2D

平面划分中的***Quad-edge(***四方边缘结构***)***

```
/* quad-edge中的一条边，低两位表示该边缘的索引号，其它高位表示边缘指针。 */
typedef long CvSubdiv2DEdge;

/* 四方边缘的结构场 */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```

Quad-edge(译者注：以下称之为四方边缘结构)是平面划分的基元，其中包括四个边缘(e, eRot 以及它们的逆)。



CvSubdiv2DPoint

原始和对偶划分点

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int            flags;      \
    CvSubdiv2DEdge first;      \
    CvPoint2D32f   pt;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

Subdiv2DGetEdge

返回给定的边缘之一

```
CvSubdiv2DEdge  cvSubdiv2DGetEdge( CvSubdiv2DEdge edge, CvNextEdgeType type
);
#define cvSubdiv2DNextEdge( edge ) cvSubdiv2DGetEdge( edge,
CV_NEXT_AROUND_ORG )
```

`edge`

划分的边缘（并不是四方边缘结构）

`type`

确定函数返回哪条相关边缘，是下面几种之一：

- CV_NEXT_AROUND_ORG - 边缘原点的下一条（`eOnext` on the picture above if `e` is the input edge）
- CV_NEXT_AROUND_DST - 边缘顶点的下一条（`eDnext`）
- CV_PREV_AROUND_ORG - 边缘原点的前一条（reversed `eRnext`）
- CV_PREV_AROUND_DST - 边缘终点的前一条（reversed `eLnext`）
- CV_NEXT_AROUND_LEFT - 左区域的下一条（`eLnext`）
- CV_NEXT_AROUND_RIGHT - 右区域的下一条(`eRnext`)
- CV_PREV_AROUND_LEFT - 左区域的前一条（reversed `eOnext`）
- CV_PREV_AROUND_RIGHT - 右区域的前一条（reversed `eDnext`）

函数 cvSubdiv2DGetEdge 返回与输入边缘相关的边缘

Subdiv2DRotateEdge

返回同一个四方边缘结构中的另一条边缘

```
CvSubdiv2DEdge  cvSubdiv2DRotateEdge( CvSubdiv2DEdge edge, int rotate );
```

edge

划分的边缘（并不是四方边缘结构）

type

确定函数根据输入的边缘返回同一四方边缘结构中的哪条边缘，是下面几种之一：

- 0 - 输入边缘（上图中的e，如果e是输入边缘）
- 1 - 旋转边缘（eRot）
- 2 –逆边缘（ e的反向边缘）
- 3 - 旋转边缘的反向边缘(eRot的反向边缘， 图中绿色)

函数 cvSubdiv2DRotateEdge 根据输入的边缘返回四方边缘结构中的一条边缘

Subdiv2DEdgeOrg

返回边缘的原点

```
CvSubdiv2DPoint* cvSubdiv2DEdgeOrg( CvSubdiv2DEdge edge );
```

edge

划分的边缘（并不是四方边缘结构）

函数 cvSubdiv2DEdgeOrg 返回边缘的原点。如果该边缘是从对偶划分得到并且虚点坐标还没有计算出来，可能返回空指针。虚点可以用函数来cvCalcSubdivVoronoi2D计算。

Subdiv2DEdgeDst

Returns edge destination

```
CvSubdiv2DPoint* cvSubdiv2DEdgeDst( CvSubdiv2DEdge edge );
```

edge

划分的边缘（并不是四方边缘结构）

函数 cvSubdiv2DEdgeDst 返回边缘的终点。如果该边缘是从对偶划分得到并且虚点坐

标还没有计算出来，可能返回空指针。虚点可以用函数来cvCalcSubdivVoronoi2D计算。

CreateSubdivDelaunay2D

生成的空***Delaunay*** 三角测量

```
CvSubdiv2D* cvCreateSubdivDelaunay2D( CvRect rect, CvMemStorage* storage );
```

rect

Rectangle包括所有待加入划分操作的2d点的四方形。

storage

划分操作的存储器

函数 cvCreateSubdivDelaunay2D 生成一个空的Delaunay 划分，其中2d points可以进一步使用函数 cvSubdivDelaunay2DInsert来添加。所有的点一定要在指定的四方形中添加，否则就会报运行错误。

SubdivDelaunay2DInsert

向 ***Delaunay***三角测量中插入一个点

```
CvSubdiv2DPoint*  cvSubdivDelaunay2DInsert( CvSubdiv2D* subdiv, CvPoint2D32f
pt);
```

subdiv

通过函数 cvCreateSubdivDelaunay2D.生成的Delaunay划分

pt

待插入的点

函数 cvSubdivDelaunay2DInsert 向划分的结构中插入一个点并且正确地改变划分的拓朴结构。如果划分结构中已经存在一个相同的坐标点，则不会有新点插入。该函数返回指向已插入点的指针。在这个截断，不计算任何虚点坐标。

Subdiv2DLocate

在 ***Delaunay***三角测量中定位输入点

```
CvSubdiv2DPointLocation  cvSubdiv2DLocate( CvSubdiv2D* subdiv, CvPoint2D32f
pt,
                                           CvSubdiv2DEdge* edge,
                                           CvSubdiv2DPoint** vertex=NULL );
```

subdiv

Delaunay 或者是其它分割结构.

pt

待定位的输入点

edge

与输入点对应的输入边缘(点在其上或者其右)

vertex
与输入点对应的输出顶点坐标(指向double类型)，可选。

函数 cvSubdiv2DLocate 在划分中定位输入点，共有5种类型:

- 输入点落入某小区域内。 函数返回参数 CV_PTLOC_INSIDE 且*edge 中包含小区域的边缘之一。
- 输入点落p在边缘之上。 函数返回参数 CV_PTLOC_ON_EDGE 且 *edge 包含此边缘。
- 输入点与划分的顶点之一相对应。 函数返回参数 CV_PTLOC_VERTEX 且 *vertex 中包括指向该顶点的指针;
- 输入点落在划分的参考区域之外。 函数返回参数 CV_PTLOC_OUTSIDE_RECT且不填写任何指针。
- 输入参数之一有误。函数报运行错误(如果已经选则了沉默或者父母出错模式，则函数返回CV_PTLOC_ERROR) 。

FindNearestPoint2D

根据输入点，找到其最近的划分顶点

```
CvSubdiv2DPoint* cvFindNearestPoint2D( CvSubdiv2D* subdiv, CvPoint2D32f pt );
```

subdiv
Delaunay或者其它划分方式

pt
输入点

函数 cvFindNearestPoint2D 是另一个定位输入点的函数。该函数找到输入点的最近划分顶点。尽管划分出的小区域(facet)被用来作为起始点，但是输入点不一定非得在最终找到的顶点所在的小区域之内。该函数返回指向找到的划分顶点的指针。

CalcSubdivVoronoi2D

计算*Voronoi*图表的细胞结构

```
void cvCalcSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv
Delaunay 划分,其中所有的点已经添加 。

函数 cvCalcSubdivVoronoi2D 计算虚点的坐标，所有与原划分中的某顶点相对应的虚

点形成了(当他们相互连接时)该顶点的Voronoi 细胞的边界。

ClearSubdivVoronoi2D

移除所有的虚点

```
void cvClearSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv

Delaunay 划分

函数 cvClearSubdivVoronoi2D 移除所有的虚点。当划分的结果被函数cvCalcSubdivVoronoi2D的前一次调用更改时，该函数被cvCalcSubdivVoronoi2D内部调用 。

还有一些其它的底层处理函数与平面划分操作协同工作，参见 cv.h 及源码。生成 delaunay.c 三角测量以及2d随机点集的Voronoi 图表的演示代码可以在 opencv/samples/c目录下的delaunay.c 文件中找到。

运动分析与对象跟踪

背景统计量的累积

Acc

将帧叠加到累积器（***accumulator***）中

```
void cvAcc( const CvArr* image, CvArr* sum, const CvArr* mask=NULL );
```

image

输入图像，1- 或 3-通道，8-比特或32-比特浮点数. (多通道的每一个通道都单独处理).

sum

同一个输入图像通道的累积，32-比特或64-比特浮点数数组.

mask

可选的运算 mask.

函数 cvAcc 将整个图像 image 或某个选择区域叠加到 sum 中:

```
sum(x,y)=sum(x,y)+image(x,y) if mask(x,y)!=0
```

SquareAcc

叠加输入图像的平方到累积器中

```
void cvSquareAcc( const CvArr* image, CvArr* sqsum, const CvArr* mask=NULL
);
```

`image`

输入图像，1- 或 3-通道，8-比特或32-比特浮点数（多通道的每一个通道都单独处理）

`sqsum`

同一个输入图像通道的累积，32-比特或64-比特浮点数数组.

`mask`

可选的运算 mask.

函数 cvSquareAcc 叠加输入图像 `image` 或某个选择区域的二次方，到累积器 `sqsum` 中

$$sqsum(x,y)=sqsum(x,y)+image(x,y)^2 \text{ if } mask(x,y)!=0$$

MultiplyAcc

将两幅输入图像的乘积叠加到累积器中

```
void cvMultiplyAcc( const CvArr* image1, const CvArr* image2, CvArr* acc,
const CvArr* mask=NULL );
```

`image1`

第一个输入图像，1- or 3-通道，8-比特 or 32-比特 浮点数（多通道的每一个通道都单独处理）

`image2`

第二个输入图像，与第一个图像的格式一样

`acc`

同一个输入图像通道的累积，32-比特或64-比特浮点数数组.

`mask`

可选的运算 mask.

函数 cvMultiplyAcc 叠加两个输入图像的乘积到累积器 `acc`:

```
acc(x,y)=acc(x,y) + image1(x,y)?image2(x,y) if mask(x,y)!=0
```

RunningAvg

更新 ***running average*** （***Hunnish:*** 不知道 ***running average*** 如何翻译才恰当）

```
void cvRunningAvg( const CvArr* image, CvArr* acc, double alpha, const
CvArr* mask=NULL );
```

`image`

输入图像，1- or 3-通道，8-比特 or 32-比特 浮点数 (each channel of multi-channel image is processed independently).

`acc`

同一个输入图像通道的累积，32-比特或64-比特浮点数数组.

`alpha`

输入图像权重

`mask`

可选的运算 mask

函数 cvRunningAvg 计算输入图像 `image` 的加权和，以及累积器 `acc` 使得 `acc` 成为帧序列的一个 running average:

```
acc(x,y)=(1-α)?acc(x,y) + α?image(x,y) if mask(x,y)!=0
```

其中 α (alpha) 调节更新速率 (累积器以多快的速率忘掉前面的帧).

运动模板

UpdateMotionHistory

去掉影像***(silhouette)*** 以更新运动历史图像

```
void cvUpdateMotionHistory( const CvArr* silhouette, CvArr* mhi,
                            double timestamp, double duration );
```

`silhouette`

影像 mask，运动发生地方具有非零象素

`mhi`

运动历史图像(单通道，32-比特 浮点数)，为本函数所更新

`timestamp`

当前时间，毫秒或其它单位

`duration`

运动跟踪的最大持续时间，用 `timestamp` 一样的时间单位

函数 cvUpdateMotionHistory 用下面方式更新运动历史图像:

```
mhi(x,y)=timestamp  if silhouette(x,y)!=0
         0          if silhouette(x,y)=0 and mhi(x,y)<timestamp-duration
         mhi(x,y)   otherwise
```

也就是，MHI (motion history image) 中在运动发生的象素点被设置为当前时间戳，而运动发生较久的象素点被清除。

CalcMotionGradient

计算运动历史图像的梯度方向

```
void cvCalcMotionGradient( const CvArr* mhi, CvArr* mask, CvArr*
orientation,
                           double delta1, double delta2, int aperture_size=3
);
```

mhi
运动历史图像

mask
Mask 图像；用来标注运动梯度数据正确的点，为输出参数。

orientation
运动梯度的方向图像，包含从 0 到 360 角度

delta1, delta2
函数在每个象素点 (x,y) 邻域寻找 MHI 的最小值 (m(x,y)) 和最大值 (M(x,y))，并且假设梯度是正确的，当且仅当:

```
min(delta1,delta2) <= M(x,y)-m(x,y) <= max(delta1,delta2).
```

aperture_size
函数所用微分算子的开孔尺寸 CV_SCHARR, 1, 3, 5 or 7 (见 cvSobel).

函数 cvCalcMotionGradient 计算 MHI 的差分 Dx 和 Dy ，然后计算梯度方向如下式:

```
orientation(x,y)=arctan(Dy(x,y)/Dx(x,y))
```

其中都要考虑 Dx(x,y)′ 和 Dy(x,y)′ 的符号 (如 cvCartToPolar 类似). 然后填充 mask 以表示哪些方向是正确的(见 delta1 和delta2 的描述).

CalcGlobalOrientation

计算某些选择区域的全局运动方向

```
double cvCalcGlobalOrientation( const CvArr* orientation, const CvArr* mask,
const CvArr* mhi,
                                double timestamp, double duration );
```

orientation
运动梯度方向图像，由函数 cvCalcMotionGradient 得到

mask
Mask 图像. 它可以是正确梯度 mask (由函数 cvCalcMotionGradient 得到)与区域 mask 的结合，其中区域 mask 确定哪些方向需要计算。

mhi
运动历史图象

timestamp
当前时间 (单位毫秒或其它) 最好在传递它到函数 cvUpdateMotionHistory 之前存储一下以便以后的重用，因为对大图像运行 cvUpdateMotionHistory 和 cvCalcMotionGradient 会花费一些时间

duration
运动跟踪的最大持续时间，用法与 cvUpdateMotionHistory 中的一致

函数 cvCalcGlobalOrientation 在选择的区域内计算整个运动方向，并且返回 0° 到 360° 之间的角度值。首先函数创建运动直方图，寻找基本方向做为直方图最大值的坐标。然后函数计算与基本方向的相对偏移量，做为所有方向向量的加权和：运行越近，权重越大。得到的角度是基本方向和偏移量的循环和。

SegmentMotion
将整个运动分割为独立的运动部分

```
CvSeq* cvSegmentMotion( const CvArr* mhi, CvArr* seg_mask, CvMemStorage*
storage,
                        double timestamp, double seg_thresh );
```

mhi
运动历史图像

seg_mask
发现应当存储的 mask 的图像，单通道，32-比特， 浮点数.

storage
包含运动连通域序列的内存存储仓

timestamp
当前时间，毫秒单位

seg_thresh
分割阈值，推荐等于或大于运动历史“每步”之间的间隔。

函数 cvSegmentMotion 寻找所有的运动分割，并且在seg_mask 用不同的单独数字(1,2,...)标识它们。它也返回一个具有 CvConnectedComp 结构的序列，其中每个结构对应一个运动部件。在这之后，每个运动部件的运动方向就可以被函数 cvCalcGlobalOrientation 利用提取的特定部件的掩模(mask)计算出来(使用 cvCmp)

对象跟踪

MeanShift
在反向投影图中发现目标中心

```
int cvMeanShift( const CvArr* prob_image, CvRect window,
                 CvTermCriteria criteria, CvConnectedComp* comp );
```

prob_image
目标直方图的反向投影(见 cvCalcBackProject).

window

初始搜索窗口

criteria

确定窗口搜索停止的准则

comp

生成的结构，包含收敛的搜索窗口坐标（comp->rect 字段）与窗口内部所有象素点的和（comp->area 字段）.

函数 cvMeanShift 在给定反向投影和初始搜索窗口位置的情况下，用迭代方法寻找目标中心。当搜索窗口中心的移动小于某个给定值时或者函数已经达到最大迭代次数时停止迭代。 函数返回迭代次数。

CamShift

发现目标中心，尺寸和方向

```
int cvCamShift( const CvArr* prob_image, CvRect window, CvTermCriteria
criteria,
                CvConnectedComp* comp, CvBox2D* box=NULL );
```

prob_image

目标直方图的反向投影（见 cvCalcBackProject）.

window

初始搜索窗口

criteria

确定窗口搜索停止的准则

comp

生成的结构，包含收敛的搜索窗口坐标（comp->rect 字段）与窗口内部所有象素点的和（comp->area 字段）.

box

目标的带边界盒子。如果非 NULL，则包含目标的尺寸和方向。

函数 cvCamShift 实现了 CAMSHIFT 目标跟踪算法([Bradski98]). 首先它调用函数 cvMeanShift 寻找目标中心，然后计算目标尺寸和方向。最后返回函数 cvMeanShift 中的迭代次数。

CvCamShiftTracker 类在 cv.hpp 中被声明，函数实现了彩色目标的跟踪。

SnakeImage

改变轮廓位置使得它的能量最小

```
void cvSnakeImage( const IplImage* image, CvPoint* points, int length,
                   float* alpha, float* beta, float* gamma, int coeff_usage,
                   CvSize win, CvTermCriteria criteria, int calc_gradient=1
);
```

image

输入图像或外部能量域

`points`

轮廓点 (snake).

`length`

轮廓点的数目

`alpha`

连续性能量的权 Weight[s], 单个浮点数或长度为 `length` 的浮点数数组, 每个轮廓点有一个权

`beta`

曲率能量的权 Weight[s], 与 `alpha` 类似

`gamma`

图像能量的权 Weight[s], 与 `alpha` 类似

`coeff_usage`

前面三个参数的不同使用方法:

- `CV_VALUE` 表示每个 `alpha, beta, gamma` 都是指向为所有点所用的一个单独数值;
- `CV_ARRAY` 表示每个 `alpha, beta, gamma` 是一个指向系数数组的指针, snake 上面各点的系数都不相同。因此, 各个系数数组必须与轮廓具有同样的大小。所有数组必须与轮廓具有同样大小

`win`

每个点用于搜索最小值的邻域尺寸, 两个 `win.width` 和 `win.height` 都必须是奇数

`criteria`

终止条件

`calc_gradient`

梯度符号。如果非零, 函数为每一个图像象素计算梯度幅值, 且把它当成能量场, 否则考虑输入图像本身。

函数 cvSnakeImage 更新 snake 是为了最小化 snake 的整个能量, 其中能量是依赖于轮廓形状的内部能量(轮廓越光滑, 内部能量越小)以及依赖于能量场的外部能量之和, 外部能量通常在哪些局部能量极值点中达到最小值(这些局部能量极值点与图像梯度表示的图像边缘相对应)。

参数 `criteria.epsilon` 用来定义必须从迭代中除掉以保证迭代正常运行的点的最少数目。

如果在迭代中去掉的点数目小于 `criteria.epsilon` 或者函数达到了最大的迭代次数 `criteria.max_iter` ，则终止函数。

光流

CalcOpticalFlowHS

计算两幅图像的光流

```
void cvCalcOpticalFlowHS( const CvArr* prev, const CvArr* curr, int
use_previous,
                          CvArr* velx, CvArr* vely, double lambda,
                          CvTermCriteria criteria );
```

`prev`
　　第一幅图像，8-比特，单通道.

`curr`
　　第二幅图像，8-比特，单通道.

`use_previous`
　　使用以前的（输入）速度域

`velx`
　　光流的水平部分，与输入图像大小一样，32-比特，浮点数，单通道.

`vely`
　　光流的垂直部分，与输入图像大小一样，32-比特， 浮点数，单通道.

`lambda`
　　Lagrangian 乘子

`criteria`
　　速度计算的终止条件

函数 cvCalcOpticalFlowHS 为输入图像的每一个象素计算光流，使用 Horn & Schunck 算法 [Horn81].

CalcOpticalFlowLK

计算两幅图像的光流

```
void cvCalcOpticalFlowLK( const CvArr* prev, const CvArr* curr, CvSize
win_size,
                          CvArr* velx, CvArr* vely );
```

`prev`
　　第一幅图像，8-比特，单通道.

`curr`
　　第二幅图像，8-比特，单通道.

`win_size`
　　用来归类象素的平均窗口尺寸 (Size of the averaging window used for

grouping pixels)

velx
光流的水平部分，与输入图像大小一样，32-比特， 浮点数，单通道.

vely
光流的垂直部分，与 输入图像大小一样，32-比特， 浮点数，单通道.

函数 cvCalcOpticalFlowLK 为输入图像的每一个象素计算光流，使用 Lucas & Kanade 算法 [Lucas81].

CalcOpticalFlowBM

用块匹配方法计算两幅图像的光流

```
void cvCalcOpticalFlowBM( const CvArr* prev, const CvArr* curr, CvSize
block_size,
                          CvSize shift_size, CvSize max_range, int
use_previous,
                          CvArr* velx, CvArr* vely );
```

prev
第一幅图像，8-比特，单通道.

curr
第二幅图像，8-比特，单通道.

block_size
比较的基本块尺寸

shift_size
块坐标的增量

max_range
块周围象素的扫描邻域的尺寸

use_previous
使用以前的（输入）速度域

velx
光流的水平部分，尺寸为 floor((prev->width - block_size.width)/shiftSize.width) × floor((prev->height - block_size.height)/shiftSize.height) ，32-比特，浮点数，单通道.

vely
光流的垂直部分，与 velx 大小一样，32-比特，浮点数，单通道.

函数 cvCalcOpticalFlowBM 为重叠块 block_size.width×block_size.height 中的每一个象素计算光流，因此其速度域小于整个图像的速度域。对每一个在图像 prev 中的块，函数试图在 curr 中某些原始块或其偏移 (velx(x0,y0),vely(x0,y0)) 块的邻域里寻找类似的块，如同在前一个函数调用中所计算的类似(如果 use_previous=1)

CalcOpticalFlowPyrLK

计算一个稀疏特征集的光流，使用金字塔中的迭代 ***Lucas-Kanade*** 方法

```
void cvCalcOpticalFlowPyrLK( const CvArr* prev, const CvArr* curr, CvArr*
prev_pyr, CvArr* curr_pyr,
                             const CvPoint2D32f* prev_features,
CvPoint2D32f* curr_features,
                             int count, CvSize win_size, int level, char*
status,
                             float* track_error, CvTermCriteria criteria,
int flags );
```

`prev`
在时间 `t` 的第一帧

`curr`
在时间 `t + dt` 的第二帧

`prev_pyr`
第一帧的金字塔缓存. 如果指针非 `NULL` ，则缓存必须有足够的空间来存储金字塔从层 1 到层 `#level` 的内容。尺寸 `(image_width+8)*image_height/3` 比特足够了

`curr_pyr`
与 `prev_pyr` 类似， 用于第二帧

`prev_features`
需要发现光流的点集

`curr_features`
包含新计算出来的位置的 点集

`features`
第二幅图像中

`count`
特征点的数目

`win_size`
每个金字塔层的搜索窗口尺寸

`level`
最大的金字塔层数。如果为 `0` ，不使用金字塔 (即金字塔为单层)，如果为 `1` ，使用两层，下面依次类推。

`status`
数组。如果对应特征的光流被发现，数组中的每一个元素都被设置为 1， 否则设置为 0。

`error`
双精度数组，包含原始图像碎片与移动点之间的差。为可选参数，可以是 `NULL` .

`criteria`
准则，指定在每个金字塔层，为某点寻找光流的迭代过程的终止条件。

`flags`
其它选项:

- `CV_LKFLOW_PYR_A_READY` ，在调用之前，先计算第一帧的金字塔

- CV_LKFLOW_PYR_B_READY ，在调用之前，先计算第二帧的金字塔
- CV_LKFLOW_INITIAL_GUESSES ，在调用之前，数组 B 包含特征的初始坐标

(Hunnish：在本节中没有出现数组 B，不知是指的哪一个)

函数 cvCalcOpticalFlowPyrLK 实现了金字塔中 Lucas-Kanade 光流计算的稀疏迭代版本 ([Bouguet00])。 它根据给出的前一帧特征点坐标计算当前视频帧上的特征点坐标。 函数寻找具有子象素精度的坐标值。

两个参数 prev_pyr 和 curr_pyr 都遵循下列规则： 如果图像指针为 0，函数在内部为其分配缓存空间，计算金字塔，然后再处理过后释放缓存。 否则，函数计算金字塔且存储它到缓存中，除非设置标识 CV_LKFLOW_PYR_A[B]_READY 。 图像应该足够大以便能够容纳 Gaussian 金字塔数据。 调用函数以后，金字塔被计算而且相应图像的标识也被设置，为下一次调用准备就绪（比如：对除了第一个图像的所有图像序列，标识 CV_LKFLOW_PYR_A_READY 被设置).

预估器

CvKalman

Kalman 滤波器状态

```
typedef struct CvKalman
{
    int MP;                     /* 测量向量维数 */
    int DP;                     /* 状态向量维数 */
    int CP;                     /* 控制向量维数 */

    /* 向后兼容字段 */
#if 1
    float* PosterState;         /* =state_pre->data.fl */
    float* PriorState;          /* =state_post->data.fl */
    float* DynamMatr;           /* =transition_matrix->data.fl */
    float* MeasurementMatr;     /* =measurement_matrix->data.fl */
    float* MNCovariance;        /* =measurement_noise_cov->data.fl */
    float* PNCovariance;        /* =process_noise_cov->data.fl */
    float* KalmGainMatr;        /* =gain->data.fl */
    float* PriorErrorCovariance;/* =error_cov_pre->data.fl */
    float* PosterErrorCovariance;/* =error_cov_post->data.fl */
    float* Temp1;               /* temp1->data.fl */
    float* Temp2;               /* temp2->data.fl */
#endif

    CvMat* state_pre;           /* 预测状态 (x'(k)):
                                   x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* 矫正状态 (x(k)):
                                   x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* 状态传递矩阵 state transition matrix (A) */
    CvMat* control_matrix;      /* 控制矩阵 control matrix (B)
                                   (如果没有控制，则不使用它)*/
    CvMat* measurement_matrix;  /* 测量矩阵 measurement matrix (H) */
    CvMat* process_noise_cov;   /* 处理噪声相关矩阵 process noise covariance
matrix (Q) */
    CvMat* measurement_noise_cov; /* 测量噪声相关矩阵 measurement noise
covariance matrix (R) */
```

```
    CvMat* error_cov_pre;       /* 先验错误估计相关矩阵 priori error estimate
covariance matrix (P'(k)):
                                    P'(k)=A*P(k-1)*At + Q)*/
    CvMat* gain;                /* Kalman 增益矩阵 gain matrix (K(k)):
                                    K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* 后验错误估计相关矩阵 posteriori error
estimate covariance matrix (P(k)):
                                    P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* 临时矩阵 temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

(Hunnish: 害怕有翻译上的不准确，因此在翻译注释时，保留了原来的英文术语，以便大家对照)

结构 CvKalman 用来保存 Kalman 滤波器状态。它由函数 cvCreateKalman 创建，由函数f cvKalmanPredict 和 cvKalmanCorrect 更新，由 cvReleaseKalman 释放. 通常该结构是为标准 Kalman 所使用的 (符号和公式都借自非常优秀的 Kalman 教程 [Welch95]):

$x_k=A?x_{k-1}+B?u_k+w_k$ 译者注: 系统运动方程
$z_k=H?x_k+v_k,$ 译者注: 系统观测方程

其中:

x_k (x_{k-1}) - 系统在时刻 k (k-1) 的状态向量 (state of the system at the moment k (k-1))
z_k - 在时刻 k 的系统状态测量向量 (measurement of the system state at the moment k)
u_k - 应用于时刻 k 的外部控制 (external control applied at the moment k)

w_k 和 v_k 分别为正态分布的运动和测量噪声

$p(w) \sim N(0,Q)$
$p(v) \sim N(0,R),$

即,
Q - 运动噪声的相关矩阵, 常量或变量
R - 测量噪声的相关矩阵, 常量或变量

对标准 Kalman 滤波器, 所有矩阵: A, B, H, Q 和 R 都是通过 cvCreateKalman 在分配结构 CvKalman 时初始化一次。但是，同样的结构和函数，通过在当前系统状态邻域中线性化扩展 Kalman 滤波器方程，可以用来模拟扩展 Kalman 滤波器，在这种情况下， A, B, H (也许还有 Q 和 R) 在每一步中都被更新。

CreateKalman

分配 ***Kalman*** 滤波器结构

```
CvKalman* cvCreateKalman( int dynam_params, int measure_params, int
control_params=0 );
```

dynam_params

状态向量维数

`measure_params`

测量向量维数

`control_params`

控制向量维数

函数 cvCreateKalman 分配 CvKalman 以及它的所有矩阵和初始参数

ReleaseKalman

释放 ***Kalman*** 滤波器结构

```
void cvReleaseKalman( CvKalman** kalman );
```

`kalman`

指向 Kalman 滤波器结构的双指针

函数 cvReleaseKalman 释放结构 CvKalman 和里面所有矩阵

KalmanPredict

估计后来的模型状态

```
const CvMat* cvKalmanPredict( CvKalman* kalman, const CvMat* control=NULL );
#define cvKalmanUpdateByTime cvKalmanPredict
```

`kalman`

Kalman 滤波器状态

`control`

控制向量 (u_k), 如果没有外部控制 (`control_params`=0) 应该为 NULL

函数 cvKalmanPredict 根据当前状态估计后来的随机模型状态, 并存储于 `kalman->state_pre`:

$$x'_k=A?x_k+B?u_k$$
$$P'_k=A?P_{k-1}*A^T + Q,$$

其中
x'_k 是预测状态 (kalman->state_pre),
x_{k-1} 是前一步的矫正状态 (kalman->state_post)
(应该在开始的某个地方初始化, 即缺省为零向量),
u_k 是外部控制(control 参数),
P'_k 是先验误差相关矩阵 (kalman->error_cov_pre)
P_{k-1} 是前一步的后验误差相关矩阵(kalman->error_cov_post)
(应该在开始的某个地方初始化, 即缺省为单位矩阵),

函数返回估计得到的状态值

KalmanCorrect

调节模型状态

```
const CvMat* cvKalmanCorrect( CvKalman* kalman, const CvMat* measurement );
#define cvKalmanUpdateByMeasurement cvKalmanCorrect
```

kalman
被更新的 Kalman 结构的指针

measurement
指向测量向量的指针，向量形式为 CvMat

函数 cvKalmanCorrect 在给定的模型状态的测量基础上，调节随机模型状态：

$K_k=P'_k?H^T?(H?P'_k?H^T+R)^{-1}$
$x_k=x'_k+K_k?(z_k-H?x'_k)$
$P_k=(I-K_k?H)?P'_k$
其中
z_k - 给定测量(mesurement parameter)
K_k - Kalman "增益" 矩阵

函数存储调节状态到 kalman->state_post 中并且输出时返回它

例子. 使用 Kalman 滤波器跟踪一个旋转的点

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

int main(int argc, char** argv)
{
    /* A matrix data */
    const float A[] = { 1, 1, 0, 1 };

    IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
    CvKalman* kalman = cvCreateKalman( 2, 1, 0 );
    /* state is (phi, delta_phi) - angle and angle increment */
    CvMat* state = cvCreateMat( 2, 1, CV_32FC1 );
    CvMat* process_noise = cvCreateMat( 2, 1, CV_32FC1 );
    /* only phi (angle) is measured */
    CvMat* measurement = cvCreateMat( 1, 1, CV_32FC1 );
    CvRandState rng;
    int code = -1;

    cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

    cvZero( measurement );
    cvNamedWindow( "Kalman", 1 );

    for(;;)
    {
        cvRandSetRange( &rng, 0, 0.1, 0 );
        rng.disttype = CV_RAND_NORMAL;

        cvRand( &rng, state );

        memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
        cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1) );
        cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-5) );
        cvSetIdentity( kalman->measurement_noise_cov, cvRealScalar(1e-1) );
        cvSetIdentity( kalman->error_cov_post, cvRealScalar(1));
        /* choose random initial state */
        cvRand( &rng, kalman->state_post );

        rng.disttype = CV_RAND_NORMAL;

        for(;;)
        {
            #define calc_point(angle)                                      \
                cvPoint( cvRound(img->width/2 + img->width/3*cos(angle)),  \
                         cvRound(img->height/2 - img->width/3*sin(angle)))

            float state_angle = state->data.fl[0];
            CvPoint state_pt = calc_point(state_angle);

            /* predict point position */
            const CvMat* prediction = cvKalmanPredict( kalman, 0 );
```

```
            float predict_angle = prediction->data.fl[0];
            CvPoint predict_pt = calc_point(predict_angle);
            float measurement_angle;
            CvPoint measurement_pt;

            cvRandSetRange( &rng, 0, sqrt(kalman->measurement_noise_cov-
>data.fl[0]), 0 );
            cvRand( &rng, measurement );

            /* generate measurement */
            cvMatMulAdd( kalman->measurement_matrix, state, measurement,
measurement );

            measurement_angle = measurement->data.fl[0];
            measurement_pt = calc_point(measurement_angle);

            /* plot points */
            #define draw_cross( center, color, d )
\
                cvLine( img, cvPoint( center.x - d, center.y - d ),
\
                             cvPoint( center.x + d, center.y + d ), color,
1, 0 ); \
                cvLine( img, cvPoint( center.x + d, center.y - d ),
\
                             cvPoint( center.x - d, center.y + d ), color,
1, 0 )

            cvZero( img );
            draw_cross( state_pt, CV_RGB(255,255,255), 3 );
            draw_cross( measurement_pt, CV_RGB(255,0,0), 3 );
            draw_cross( predict_pt, CV_RGB(0,255,0), 3 );
            cvLine( img, state_pt, predict_pt, CV_RGB(255,255,0), 3, 0 );

            /* adjust Kalman filter state */
            cvKalmanCorrect( kalman, measurement );

            cvRandSetRange( &rng, 0, sqrt(kalman->process_noise_cov-
>data.fl[0]), 0 );
            cvRand( &rng, process_noise );
            cvMatMulAdd( kalman->transition_matrix, state, process_noise,
state );

            cvShowImage( "Kalman", img );
            code = cvWaitKey( 100 );

            if( code > 0 ) /* break current simulation by pressing a key */
                break;
        }
        if( code == 27 ) /* exit by ESCAPE */
            break;
    }

    return 0;
}
```

CvConDensation

ConDensaation 状态

```
    typedef struct CvConDensation
    {
        int MP;     // 测量向量的维数: Dimension of measurement vector
        int DP;     // 状态向量的维数: Dimension of state vector
        float* DynamMatr;       // 线性动态系统矩阵: Matrix of the linear
Dynamics system
        float* State;           // 状态向量: Vector of State
        int SamplesNum;         // 粒子数: Number of the Samples
        float** flSamples;      // 粒子向量数组: array of the Sample Vectors
        float** flNewSamples;   // 粒子向量临时数组: temporary array of the
Sample Vectors
        float* flConfidence;    // 每个粒子的置信度(译者注: 也就是粒子的权
值): Confidence for each Sample
        float* flCumulative;    // 权值的累计: Cumulative confidence
        float* Temp;            // 临时向量: Temporary vector
        float* RandomSample;    // 用来更新粒子集的随机向量: RandomVector to
update sample set
        CvRandState* RandS;     // 产生随机向量的结构数组: Array of structures
to generate random vectors
    } CvConDensation;
```

结构 CvConDensation中条件概率密度传播(译者注：粒子滤波的一种特例)(Con-Dens-Aation: 单词 CONditional DENSity propagATION 的缩写) 跟踪器的状态。该算法描述可参考

http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/ISARD1/condensation.html

---

CreateConDensation

分配 ***ConDensation*** 滤波器结构

```
CvConDensation* cvCreateConDensation( int dynam_params, int measure_params,
int sample_count );
```

`dynam_params`
　　状态向量的维数

`measure_params`
　　测量向量的维数

`sample_count`
　　粒子数

函数 cvCreateConDensation 创建结构 CvConDensation 并且返回结构指针。

ReleaseConDensation

释放 ***ConDensation*** 滤波器结构

```
void cvReleaseConDensation( CvConDensation** condens );
```

`condens`
　　要释放结构的双指针

函数 cvReleaseConDensation 释放结构 CvConDensation (见cvConDensation) 并且清空所有事先被开辟的内存空间。

ConDensInitSampleSet

初始化 ***ConDensation*** 算法中的粒子集

```
void cvConDensInitSampleSet( CvConDensation* condens, CvMat* lower_bound,
CvMat* upper_bound );
```

`condens`
　　需要初始化的结构指针

`lower_bound`
　　每一维的下界向量

`upper_bound`
　　每一维的上界向量

函数 cvConDensInitSampleSet 在指定区间内填充结构 CvConDensation 中的样例数组。

ConDensUpdateByTime

估计下个模型状态

```
void cvConDensUpdateByTime( CvConDensation* condens );
```

condens

要更新的机构指针

函数 cvConDensUpdateByTime 从当前状态估计下一个随机模型状态。

模式识别

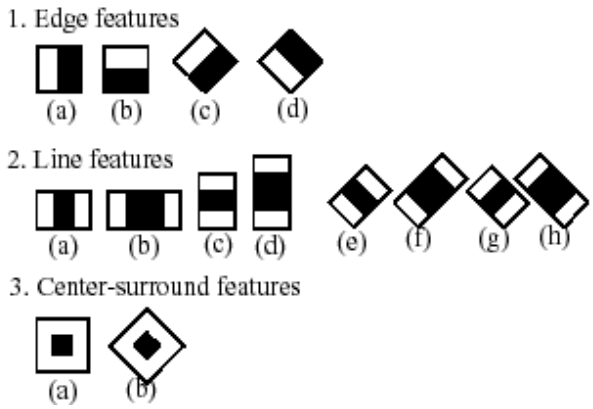
目标检测

目标检测方法最初由Paul Viola [Viola01]提出，并由Rainer Lienhart [Lienhart02]对这一方法进行了改善. 首先，利用样本（大约几百幅样本图片）的harr 特征进行分类器训练，得到一个级联的boosted分类器。训练样本分为正例样本和反例样本，其中正例样本是指待检目标样本(例如人脸或汽车等)，反例样本指其它任意图片，所有的样本图片都被归一化为同样的尺寸大小(例如，20x20)。

分类器训练完以后，就可以应用于输入图像中的感兴趣区域(与训练样本相同的尺寸)的检测。检测到目标区域(汽车或人脸)分类器输出为1，否则输出为0。为了检测整副图像，可以在图像中移动搜索窗口，检测每一个位置来确定可能的目标。 为了搜索不同大小的目标物体，分类器被设计为可以进行尺寸改变，这样比改变待检图像的尺寸大小更为有效。所以，为了在图像中检测未知大小的目标物体，扫描程序通常需要用不同比例大小的搜索窗口对图片进行几次扫描。

分类器中的“级联”是指最终的分类器是由几个简单分类器级联组成。在图像检测中，被检窗口依次通过每一级分类器， 这样在前面几层的检测中大部分的候选区域就被排除了，全部通过每一级分类器检测的区域即为目标区域。 目前支持这种分类器的

boosting技术有四种： Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost。"boosted" 即指级联分类器的每一层都可以从中选取一个boosting算法(权重投票)，并利用基础分类器的自我训练得到。基础分类器是至少有两个叶结点的决策树分类器。 海尔特征是基础分类器的输入，主要描述如下。目前的算法主要利用下面的海尔特征。



每个特定分类器所使用的特征用形状、感兴趣区域中的位置以及比例系数（这里的比例系数跟检测时候采用的比例系数是不一样的，尽管最后会取两个系数的乘积值）来定义。例如在第三行特征(2c)的情况下，响应计算为覆盖全部特征整个矩形框(包括两个白色矩形框和一个黑色矩形框)象素的和减去黑色矩形框内象素和的三倍 。每个矩形框内的象素和都可以通过积分图象很快的计算出来。(察看下面和对cvIntegral的描述).

通过HaarFaceDetect 的演示版可以察看目标检测的工作情况。

下面只是检测部分的参考手册。 haartraining是它的一个单独的应用，可以用来对系列样本训练级联的 boosted分类器。详细察看opencv/apps/haartraining。

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade

Boosted Haar 分类器结构

```
#define CV_HAAR_FEATURE_MAX  3

/* 一个 harr 特征由 2-3 个具有相应权重的矩形组成a haar feature consists of 2-3
rectangles with appropriate weights */
typedef struct CvHaarFeature
{
    int  tilted;  /* 0 means up-right feature, 1 means 45--rotated feature
*/

    /* 2-3 rectangles with weights of opposite signs and
       with absolute values inversely proportional to the areas of the
```

```
rectangles.
       if rect[2].weight !=0, then
       the feature consists of 3 rectangles, otherwise it consists of 2 */
    struct
    {
        CvRect r;
        float weight;
    } rect[CV_HAAR_FEATURE_MAX];
}
CvHaarFeature;

/* a single tree classifier (stump in the simplest case) that returns the
response for the feature
   at the particular image location (i.e. pixel sum over subrectangles of
the window) and gives out
   a value depending on the responce */
typedef struct CvHaarClassifier
{
    int count;  /* number of nodes in the decision tree */

    /* these are "parallel" arrays. Every index i
       corresponds to a node of the decision tree (root has 0-th index).

       left[i] - index of the left child (or negated index if the left child
is a leaf)
       right[i] - index of the right child (or negated index if the right
child is a leaf)
       threshold[i] - branch threshold. if feature responce is <= threshold,
left branch
                      is chosen, otherwise right branch is chosed.
       alpha[i] - output value correponding to the leaf. */
    CvHaarFeature* haar_feature;
    float* threshold;
    int* left;
    int* right;
    float* alpha;
}
CvHaarClassifier;

/* a boosted battery of classifiers(=stage classifier):
   the stage classifier returns 1
   if the sum of the classifiers' responces
   is greater than threshold and 0 otherwise */
typedef struct CvHaarStageClassifier
{
    int  count;  /* number of classifiers in the battery */
    float threshold; /* threshold for the boosted classifier */
    CvHaarClassifier* classifier; /* array of classifiers */

    /* these fields are used for organizing trees of stage classifiers,
       rather than just stright cascades */
    int next;
    int child;
    int parent;
}
CvHaarStageClassifier;

typedef struct CvHidHaarClassifierCascade CvHidHaarClassifierCascade;

/* cascade or tree of stage classifiers */
typedef struct CvHaarClassifierCascade
{
    int  flags; /* signature */
    int  count; /* number of stages */
    CvSize orig_window_size; /* original object size (the cascade is trained
for) */

    /* these two parameters are set by cvSetImagesForHaarClassifierCascade
*/
    CvSize real_window_size; /* current object size */
    double scale; /* current scale */
    CvHaarStageClassifier* stage_classifier; /* array of stage classifiers
*/
    CvHidHaarClassifierCascade* hid_cascade; /* hidden optimized
representation of the cascade,
                                                created by
cvSetImagesForHaarClassifierCascade */
}
CvHaarClassifierCascade;
```

所有的结构都代表一个级联boosted Haar分类器。级联有下面的等级结构:

```
    Cascade:
        Stage1:
            Classifier11:
                Feature11
            Classifier12:
                Feature12
            ...
        Stage2:
            Classifier21:
                Feature21
            ...
        ...
```

整个等级可以手工构建，也可以利用函数cvLoadHaarClassifierCascade从已有的磁盘文件或嵌入式基中导入。

---

cvLoadHaarClassifierCascade

从文件中装载训练好的级联分类器或者从***OpenCV***中嵌入的分类器数据库中导入

```
CvHaarClassifierCascade* cvLoadHaarClassifierCascade(
                        const char* directory,
                        CvSize orig_window_size );
```

directory

训练好的级联分类器的路径

orig_window_size

级联分类器训练中采用的检测目标的尺寸。因为这个信息没有在级联分类器中存储，所有要单独指出。

函数 cvLoadHaarClassifierCascade 用于从文件中装载训练好的利用海尔特征的级联分类器，或者从OpenCV中嵌入的分类器数据库中导入。分类器的训练可以应用函数haartraining(详细察看opencv/apps/haartraining)

函数 已经过时了。现在的目标检测分类器通常存储在 XML 或 YAML 文件中,而不是通过路径导入。从文件中导入分类器，可以使用函数 cvLoad 。

---

cvReleaseHaarClassifierCascade

释放***haar classifier cascade***。

```
void cvReleaseHaarClassifierCascade( CvHaarClassifierCascade** cascade );
```

cascade

双指针类型指针指向要释放的cascade. 指针由函数声明。

函数 cvReleaseHaarClassifierCascade 释放cascade的动态内存，其中cascade的动态内存或者是手工创建，或者通过函数 cvLoadHaarClassifierCascade 或 cvLoad分

配。

---

cvHaarDetectObjects

检测图像中的目标

```
typedef struct CvAvgComp
{
    CvRect rect; /* bounding rectangle for the object (average rectangle of
a group) */
    int neighbors; /* number of neighbor rectangles in the group */
}
CvAvgComp;

CvSeq* cvHaarDetectObjects( const CvArr* image, CvHaarClassifierCascade*
cascade,
                            CvMemStorage* storage, double scale_factor=1.1,
                            int min_neighbors=3, int flags=0,
                            CvSize min_size=cvSize(0,0) );
```

image

被检图像

cascade

harr 分类器级联的内部标识形式

storage

用来存储检测到的一序列候选目标矩形框的内存区域。

scale_factor

在前后两次相继的扫描中，搜索窗口的比例系数。例如1.1指将搜索窗口依次扩大10%。

min_neighbors

构成检测目标的相邻矩形的最小个数(缺省－1)。如果组成检测目标的小矩形的个数和小于min_neighbors-1 都会被排除。如果min_neighbors 为 0，则函数不做任何操作就返回所有的被检候选矩形框，这种设定值一般用在用户自定义对检测结果的组合程序上。

flags

操作方式。当前唯一可以定义的操作方式是 CV_HAAR_DO_CANNY_PRUNING。如果被设定，函数利用Canny边缘检测器来排除一些边缘很少或者很多的图像区域，因为这样的区域一般不含被检目标。人脸检测中通过设定阈值使用了这种方法，并因此提高了检测速度。

min_size

检测窗口的最小尺寸。缺省的情况下被设为分类器训练时采用的样本尺寸(人脸检测中缺省大小是20×20)。

函数 cvHaarDetectObjects 使用针对某目标物体训练的级联分类器在图像中找到包含目标物体的矩形区域，并且将这些区域作为一序列的矩形框返回。函数以

不同比例大小的扫描窗口对图像进行几次搜索(察看cvSetImagesForHaarClassifierCascade)。 每次都要对图像中的这些重叠区域利用cvRunHaarClassifierCascade进行检测。 有时候也会利用某些继承(heuristics)技术以减少分析的候选区域，例如利用 Canny 裁减(prunning)方法。 函数在处理和收集到候选的方框(全部通过级联分类器各层的区域)之后，接着对这些区域进行组合并且返回一系列各个足够大的组合中的平均矩形。调节程序中的缺省参数(scale_factor=1.1, min_neighbors=3, flags=0)用于对目标进行更精确同时也是耗时较长的进一步检测。为了能对视频图像进行更快的实时检测，参数设置通常是: scale_factor=1.2, min_neighbors=2, flags=CV_HAAR_DO_CANNY_PRUNING, min_size=<minimum possible face size> (例如, 对于视频会议的图像区域).

例子:利用级联的**Haar classifiers**寻找检测目标**(e.g. faces).**

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

CvHaarClassifierCascade* load_object_detector( const char* cascade_path )
{
    return (CvHaarClassifierCascade*)cvLoad( cascade_path );
}

void detect_and_draw_objects( IplImage* image,
                              CvHaarClassifierCascade* cascade,
                              int do_pyramids )
{
    IplImage* small_image = image;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* faces;
    int i, scale = 1;

    /* if the flag is specified, down-scale the 输入图像 to get a
       performance boost w/o loosing quality (perhaps) */
    if( do_pyramids )
    {
        small_image = cvCreateImage( cvSize(image->width/2,image->height/2),
IPL_DEPTH_8U, 3 );
        cvPyrDown( image, small_image, CV_GAUSSIAN_5x5 );
        scale = 2;
    }

    /* use the fastest variant */
    faces = cvHaarDetectObjects( small_image, cascade, storage, 1.2, 2,
CV_HAAR_DO_CANNY_PRUNING );

    /* draw all the rectangles */
    for( i = 0; i < faces->total; i++ )
    {
        /* extract the rectanlges only */
        CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i, 0 );
        cvRectangle( image, cvPoint(face_rect.x*scale,face_rect.y*scale),
                     cvPoint((face_rect.x+face_rect.width)*scale,
                             (face_rect.y+face_rect.height)*scale),
                     CV_RGB(255,0,0), 3 );
    }

    if( small_image != image )
        cvReleaseImage( &small_image );
    cvReleaseMemStorage( &storage );
}

/* takes image filename and cascade path from the command line */
```

```
int main( int argc, char** argv )
{
    IplImage* image;
    if( argc==3 && (image = cvLoadImage( argv[1], 1 )) != 0 )
    {
        CvHaarClassifierCascade* cascade = load_object_detector(argv[2]);
        detect_and_draw_objects( image, cascade, 1 );
        cvNamedWindow( "test", 0 );
        cvShowImage( "test", image );
        cvWaitKey(0);
        cvReleaseHaarClassifierCascade( &cascade );
        cvReleaseImage( &image );
    }

    return 0;
}
```

---

cvSetImagesForHaarClassifierCascade

为隐藏的***cascade(hidden cascade)***指定图像

```
void cvSetImagesForHaarClassifierCascade( CvHaarClassifierCascade* cascade,
                                          const CvArr* sum, const CvArr*
sqsum,
                                          const CvArr* tilted_sum, double
scale );
```

cascade

隐藏 Harr 分类器级联 (Hidden Haar classifier cascade)，由函数 cvCreateHidHaarClassifierCascade生成

sum

32－比特，单通道图像的积分图像 (Integral (sum) 单通道 image of 32-比特 integer format)．这幅图像以及随后的两幅用于对快速特征的评价和亮度/对比度的归一化。它们都可以利用函数 cvIntegral从8-比特或浮点数 单通道的输入图像中得到。

sqsum

单通道64比特图像的平方和图像

tilted_sum

单通道32比特整数格式的图像的倾斜和 (Tilted sum)

scale

cascade的窗口比例．如果 scale=1，就只用原始窗口尺寸检测 (只检测同样尺寸大小的目标物体) - 原始窗口尺寸在函数cvLoadHaarClassifierCascade中定义 (在 "<default_face_cascade>"中缺省为24x24)，如果scale=2，使用的窗口是上面的两倍 (在face cascade中缺省值是48x48)。这样尽管可以将检测速度提高四倍，但同时尺寸小于48x48的人脸将不能被检测到。

函数 cvSetImagesForHaarClassifierCascade 为hidden classifier cascade 指定图像 and/or 窗口比例系数。如果图像指针为空，会继续使用原来的图像(i.e. NULLs 意味这"不改变图像")。比例系数没有 "protection" 值,但是原来的值可以通过函数 cvGetHaarClassifierCascadeScale 重新得到并使用。这个函数用于对特定图像中检

测特定目标尺寸的cascade分类器的设定。函数通过cvHaarDetectObjects进行内部调用，但当需要在更低一层的函数cvRunHaarClassifierCascade中使用的时候，用户也可以自行调用。

---

cvRunHaarClassifierCascade

在给定位置的图像中运行 ***cascade of boosted classifier***

```
int cvRunHaarClassifierCascade( CvHaarClassifierCascade* cascade,
                                CvPoint pt, int start_stage=0 );
```

cascade

Haar 级联分类器

pt

待检测区域的左上角坐标。待检测区域大小为原始窗口尺寸乘以当前设定的比例系数。当前窗口尺寸可以通过cvGetHaarClassifierCascadeWindowSize重新得到。

start_stage

级联层的初始下标值（从0开始计数）。函数假定前面所有每层的分类器都已通过。这个特征通过函数cvHaarDetectObjects内部调用，用于更好的处理器高速缓冲存储器。

函数 cvRunHaarHaarClassifierCascade 用于对单幅图片的检测。在函数调用前首先利用 cvSetImagesForHaarClassifierCascade设定积分图和合适的比例系数 (=> 窗口尺寸)。当分析的矩形框全部通过级联分类器每一层的时返回正值(这是一个候选目标)，否则返回0或负值。

照相机定标和三维重建

照相机定标

CalibrateCamera

对相机单精度定标

```
void cvCalibrateCamera( int image_count, int* point_counts, CvSize
image_size,
                        CvPoint2D32f* image_points, CvPoint3D32f*
object_points,
                        CvVect32f distortion_coeffs, CvMatr32f
```

```
camera_matrix,
                        CvVect32f translation_vectors, CvMatr32f
rotation_matrixes,
                        int use_intrinsic_guess );
```

image_count
图像个数。

point_counts
每副图像定标点个数的数组

image_size
图像大小

image_points
指向图像的指针

object_points
指向检测目标的指针

distortion_coeffs
寻找到四个变形系数的数组。

camera_matrix
相机参数矩阵。Camera matrix found.

translation_vectors
每个图像中模式位置的平移矢量矩阵。

rotation_matrixes
图像中模式位置的旋转矩阵。

use_intrinsic_guess
是否使用内部猜测（Intrinsic guess）。设为1，则使用

函数 cvCalibrateCamera 利用目标图像模式和目标模式的象素点信息计算相机参数。

CalibrateCamera_64d

相机双精度定标

```
void cvCalibrateCamera_64d( int image_count, int* point_counts, CvSize
image_size,
                            CvPoint2D64d* image_points, CvPoint3D64d*
object_points,
                            CvVect64d distortion_coeffs, CvMatr64d
camera_matrix,
                            CvVect64d translation_vectors, CvMatr64d
rotation_matrixes,
                            int use_intrinsic_guess );
```

image_count
图像个数

point_counts
每副图像定标点数目的数组

image_size
图像大小

image_points
图像指针

object_points
模式对象指针Pointer to the pattern.

`distortion_coeffs`
寻找变形系数Distortion coefficients found.

`camera_matrix`
相机矩阵Camera matrix found.

`translation_vectors`
图像中模式位置的平移矢量矩阵

`rotation_matrixes`
图像中每个模式位置的旋转矩阵

`use_intrinsic_guess`
Intrinsic guess. If equal to 1, intrinsic guess is needed.

函数 cvCalibrateCamera_64d 跟 函数 cvCalibrateCamera用法基本相同，不过cvCalibrateCamera_64d使用的是双精度类型。

Rodrigues

进行旋转矩阵和旋转向量间的转换，采用单精度。

```
void  cvRodrigues( CvMat* rotation_matrix, CvMat* rotation_vector,
                   CvMat* jacobian, int conv_type);
```

`rotation_matrix`
旋转矩阵(3x3), 32-比特 or 64-比特 浮点数

`rotation_vector`
跟旋转矩阵相同类型的旋转向量(3x1 or 1x3)

`jacobian`
雅可比矩阵3 × 9

`conv_type`
转换方式；设定为CV_RODRIGUES_M2V，矩阵转化为向量。CV_RODRIGUES_V2M向量转化为矩阵

函数 cvRodrigues 进行旋转矩阵和旋转向量之间的相互转换。

UnDistortOnce

校正相机镜头变形 ***Corrects camera lens distortion***

```
void cvUnDistortOnce( const CvArr* src, CvArr* dst,
                      const float* intrinsic_matrix,
                      const float* distortion_coeffs,
                      int interpolate=1 );
```

`src`
原图像(变形图像)

`dst`
目标图像 (校正)图像.

`intrinsic_matrix`
相机的内参数矩阵(3x3).

distortion_coeffs
向量的四个变形系数 k_1, k_2, p_1 和 p_2.

interpolate
双线性卷积标志。

函数 cvUnDistortOnce 校正单幅图像的镜头变形。相机的内置参数矩阵和变形系数k_1, k_2 , p_1 ,和 p_2事先由函数cvCalibrateCamera计算得到。

UnDistortInit

计算畸变点数组和插值系数 ***Calculates arrays of distorted points indices and interpolation coefficients***

```
void cvUnDistortInit( const CvArr* src, CvArr* undistortion_map,
                      const float* intrinsic_matrix,
                      const float* distortion_coeffs,
                      int interpolate=1 );
```

src
任意的原图像(变形图形)，图像的大小要和通道数匹配。

undistortion_map
32-比特整数的图像，如果interpolate=0，与输入图像尺寸相同，如果interpolate=1，是输入图像的3倍

intrinsic_matrix
摄像机的内参数矩阵

distortion_coeffs
向量的4个变形系数 k_1, k_2, p_1 and p_2

interpolate
双线性卷积标志

函数 cvUnDistortInit 利用摄像机内参数和变形系数计算畸变点指数数组和插值系数。为函数cvUnDistort计算非畸变图。

摄像机的内参数矩阵和变形系数可以由函数cvCalibrateCamera计算。

UnDistort

校正相机镜头变形 ***Corrects camera lens distortion***

```
void cvUnDistort( const CvArr* src, CvArr* dst,
                  const CvArr* undistortion_map, int interpolate=1 );
```

src
原图像(变形图像)

dst
目标图像(校正后图像)

undistortion_map
未变形图像，由函数cvUnDistortInit提前计算

`interpolate`
双线性卷积标志，与函数cvUnDistortInit中相同。

函数 cvUnDistort 提前计算出的未变形图校正相机镜头变形，速度比函数cvUnDistortOnce 快。（利用先前计算出的非变形图来校正摄像机的镜头变形。速度比函数 cvUnDistortOnce快）

FindChessBoardCornerGuesses
发现棋盘内部角点的大概位置

```
int cvFindChessBoardCornerGuesses( const CvArr* image, CvArr* thresh,
                                   CvMemStorage* storage, CvSize board_size,
                                   CvPoint2D32f* corners, int*
corner_count=NULL );
```

`image`
输入的棋盘图像，象素位数为 IPL_DEPTH_8U.

`thresh`
临时图像，与输入图像的大小、格式一样

`storage`
中间数据的存储区，如果为空，函数生成暂时的内存区域。

`board_size`
棋盘每一行和每一列的内部角点数目。宽（列数目）必须小于等于高（行数目）

`corners`
角点数组的指针

`corner_count`
带符号的已发现角点数目. 正值表示整个棋盘的角点都以找到，负值表示不是所有的角点都被找到。

函数 cvFindChessBoardCornerGuesses 试图确定输入图像是否是棋盘视图，并且确定棋盘的内部角点。如果所有角点都被发现，函数返回非零值，并且将角点按一定顺序排列（逐行由左到右），否则，函数返回零。例如一个简单棋盘有 8 x 8 方块和 7 x 7 内部方块相切的角点。单词“大概 approximate" 表示发现的角点坐标与实际坐标会有几个象素的偏差。为得到更精确的坐标，可使用函数 cvFindCornerSubPix.

姿态估计

FindExtrinsicCameraParams
为模式寻找摄像机外参数矩阵

```
void cvFindExtrinsicCameraParams( int point_count, CvSize image_size,
```

```
object_points,
principal_point,
rotation_vector,
```

```
CvPoint2D32f* image_points, CvPoint3D32f*
CvVect32f focal_length, CvPoint2D32f
CvVect32f distortion_coeffs, CvVect32f
CvVect32f translation_vector );
```

`point_count`
　　点的个数
`ImageSize`
　　图像大小
`image_points`
　　图像指针
`object_points`
　　模式指针Pointer to the pattern.
`focal_length`
　　焦距
`principal_point`
　　基点
`distortion_coeffs`
　　变形系数。
`rotation_vector`
　　旋转向量
`translation_vector`
　　平移向量

函数 cvFindExtrinsicCameraParams 寻找模式的摄像机外参数矩阵

FindExtrinsicCameraParams_64d

以双精度形式寻找照相机外参数 ***Finds extrinsic camera parameters for pattern with double precision***

```
void cvFindExtrinsicCameraParams_64d( int point_count, CvSize image_size,
CvPoint3D64d* object_points,
principal_point,
rotation_vector,
```

```
CvPoint2D64d* image_points,
CvVect64d focal_length, CvPoint2D64d
CvVect64d distortion_coeffs, CvVect64d
CvVect64d translation_vector );
```

`point_count`
　　点的个数
`ImageSize`
　　图像尺寸
`image_points`
　　图像指针
`object_points`
　　模式指针Pointer to the pattern.
`focal_length`
　　焦距

`principal_point`
基点

`distortion_coeffs`
变形系数

`rotation_vector`
旋转向量

`translation_vector`
平移向量

函数 cvFindExtrinsicCameraParams_64d 建立对象模式的外参数，双精度 。

CreatePOSITObject

初始化目标信息 ***Initializes structure containing object information***

```
CvPOSITObject* cvCreatePOSITObject( CvPoint3D32f* points, int point_count );
```

`points`
指向3D对象模型的指针

`point_count`
目标对象的点数

函数 cvCreatePOSITObject 为对象结构分配内存并计算对象的逆矩阵。

预处理的对象数据存储在结构CvPOSITObject中，通过OpenCV内部调用，即用户不能直接得到得到数据结构。用户只可以创建这个结构并将指针传递给函数。

对象是一系列给定坐标的像素点，函数 cvPOSIT计算从照相机坐标系到目标点`points[0]` 之间的向量。

当完成上述对象的工作以后，必须使用函数cvReleasePOSITObject释放内存。

POSIT

执行***POSIT***算法。***Implements POSIT algorithm***

```
void cvPOSIT( CvPOSITObject* posit_object, CvPoint2D32f* image_points, double
focal_length,
              CvTermCriteria criteria, CvMatr32f rotation_matrix, CvVect32f
translation_vector );
```

`posit_object`
指向对象结构的指针

`image_points`
指针，指向目标像素点在二维平面图上的投影。

`focal_length`
使用的摄像机的焦距

`criteria`

POSIT迭代算法程序终止的条件

`rotation_matrix`

旋转矩阵

`translation_vector`

平移矩阵Translation vector.

函数 cvPOSIT 执行POSIT算法。图像坐标在摄像机坐标系统中给出。焦距可以通过摄像机标定得到。算法每一次迭代都会重新计算在估计位置的透视投影。

两次投影之间的范式差值是对应点间的最大距离。如果差值过小，参数`criteria.epsilon`就会终止程序。

ReleasePOSITObject

释放***3D***对象结构

```
void cvReleasePOSITObject( CvPOSITObject** posit_object );
```

`posit_object`

指向 `CvPOSIT` structure双指针

函数 cvReleasePOSITObject 释放函数 cvCreatePOSITObject分配的内存。

CalcImageHomography

计算长方形或椭圆形平面对象的单应矩阵***(***例如，胳膊***)*** ***Calculates homography matrix for oblong planar object (e.g. arm)***

```
void cvCalcImageHomography( float* line, CvPoint3D32f* center,
                            float* intrinsic, float* homography );
```

`line`

主要对象的轴方向 (vector (dx,dy,dz)).

`center`

对象坐标中心 ((cx,cy,cz)).

`intrinsic`

摄像机内参数 (3x3 matrix).

`homography`

输出的单应矩阵(3x3).

函数 cvCalcImageHomography 计算初始图像由图像平面到3D oblong object line界定的平面转换的单应矩阵。 (察看OpenCV指南中的3D重建一章Figure 6-10)

外极线几何

FindFundamentalMat

计算图像中对应点的基本矩阵

```
int cvFindFundamentalMat( CvMat* points1,
                          CvMat* points2,
                          CvMat* fundamental_matrix,
                          int    method,
                          double param1,
                          double param2,
                          CvMat* status=0);
```

points1

第一幅图像点的数组2xN/Nx2 或 3xN/Nx3 (N 点的个数). 点坐标应该是浮点数(双精度或单精度)。

points2

第二副图像的点的数组，格式、大小与第一幅图像相同。

fundamental_matrix

输出的基本矩阵。大小是 3x3 or 9x3 (7-point method can returns up to 3 matrices).

method

计算基本矩阵的方法

CV_FM_7POINT - for 7-point algorithm. Number of points == 7

CV_FM_8POINT - for 8-point algorithm. Number of points >= 8

CV_FM_RANSAC - for RANSAC algorithm. Number of points >= 8

CV_FM_LMEDS - for LMedS algorithm. Number of points >= 8

param1

这个参数只用于方法RANSAC 或 LMedS 。它是点到外极线的最大距离，超过这个值的点将被舍弃，不用于后面的计算。通常这个值的设定是0.5 or 1.0 。

param2

这个参数只用于方法RANSAC 或 LMedS 。 它表示矩阵在某种精度上的正确的理想值。例如可以被设为0.99 。

status

N个元素的数组，在计算过程中没有被舍弃的点，元素被被置为1。否则置为0。数组由方法RANSAC and LMedS 计算。其它方法的时候status被置为1's。这个参数是可选的。 Th

外极线几何可以用下面的等式描述:

$p_2^T*F*p_1=0$,

其中 F 是基本矩阵，ip_1 和 p_2 分别是两幅图上的对应点。

函数 `FindFundamentalMat` 利用上面列出的四种方法之一计算基本矩阵，并返回基本矩阵的值：没有找到矩阵，返回0，找到一个矩阵返回1，多个矩阵返回3。

基本矩阵可以用来进一步计算两幅图像的对应外极点的坐标。

7点法使用确定的7个点。这种方法能找到1个或者3个基本矩阵，返回矩阵的个数；如果目标数组中有足够的空间存储所有检测到的矩阵，该函数将所有矩阵存储，否则，只存贮其中之一。其它方法使用8点或者更多点并且返回一个基本矩阵。

Example. Fundamental matrix calculation

```
int point_count = 100;
CvMat* points1;
CvMat* points2;
CvMat* status;
CvMat* fundamental_matrix;

points1  = cvCreateMat(2,point_count,CV_32F);
points2  = cvCreateMat(2,point_count,CV_32F);
status   = cvCreateMat(1,point_count,CV_32F);

/* Fill the points here ... */

fundamental_matrix = cvCreateMat(3,3,CV_32F);
int num =
cvFindFundamentalMat(points1,points2,fundamental_matrix,CV_FM_RANSAC,1.0,0.99,status);

if( num == 1 )
{
    printf("Fundamental matrix was found\n");
}
else
{
    printf("Fundamental matrix was not found\n");
}


/*====== Example of code for three matrixes ======*/
CvMat* points1;
CvMat* points2;
CvMat* fundamental_matrix;

points1  = cvCreateMat(2,7,CV_32F);
points2  = cvCreateMat(2,7,CV_32F);

/* Fill the points here... */

fundamental_matrix = cvCreateMat(9,3,CV_32F);
int num =
cvFindFundamentalMat(points1,points2,fundamental_matrix,CV_FM_7POINT,0,0,0);
printf("Found %d matrixes\n",num);
```

ComputeCorrespondEpilines

根据一幅图像中的点计算其在另一幅图像中对应的外极线。

```
void cvComputeCorrespondEpilines( const CvMat* points,
                                  int which_image,
                                  const CvMat* fundamental_matrix,
                                  CvMat* correspondent_lines);
```

`points`

输入点：2xN or 3xN array (N 点的个数)

`which_image`

包含点的图像指数(1 or 2)

`fundamental_matrix`
基本矩阵
`correspondent_lines`
计算外极点, 3xN array

函数 `ComputeCorrespondEpilines` 根据外级线几何的基本方程计算每个输入点的对应外级线。如果点位于第一幅图像(`which_image`=1),对应的外极线可以如下计算 :

$l_2=F*p_1$

其中F是基本矩阵, p_1 是第一幅图像中的点, l_2 - 是与第二幅对应的外极线。如果点位于第二副图像中 `which_image`=2), 计算如下:

$l_1=F^T*p_2$

其中p_2 是第二幅图像中的点, l_1 是对应于第一幅图像的外极线

每条外极线都可以用三个系数表示 *a*, *b*, *c*:

```
a*x + b*y + c = 0
```

归一化后的外极线系数存储在`correspondent_lines`.

按字母顺序的函数列表

2

2DRotationMatrix

A

Acc　　ApproxChains　　ArcLength

AdaptiveThreshold　　ApproxPoly

B

BoundingRect　　BoxPoints

C

CalcBackProject　　CalibrateCamera　　ConvexityDefects

D

E

F

G

S

T

U

W

参考书目

下面的参考书目列出了对于Intel Computer Vision Library用户的一些有用的出版物。目录不是很全，只作为一个学习的起点。

1. [Borgefors86] Gunilla Borgefors, "Distance Transformations in Digital Images". Computer Vision, Graphics and Image Processing 34, 344-371

(1986).

2. [Bouguet00] Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker.
   The paper is included into OpenCV distribution (algo_tracking.pdf)
3. [Bradski98] G.R. Bradski. Computer vision face tracking as a component of a perceptual user interface. In Workshop on Applications of Computer Vision, pages 214?19, Princeton, NJ, Oct. 1998.
   Updated version can be found at
   http://www.intel.com/technology/itj/q21998/articles/art_2.htm.
   Also, it is included into OpenCV distribution (camshift.pdf)
4. [Bradski00] G. Bradski and J. Davis. Motion Segmentation and Pose Recognition with Motion History Gradients. IEEE WACV'00, 2000.
5. [Burt81] P. J. Burt, T. H. Hong, A. Rosenfeld. Segmentation and Estimation of Image Region Properties Through Cooperative Hierarchical Computation. IEEE Tran. On SMC, Vol. 11, N.12, 1981, pp. 802-809.
6. [Canny86] J. Canny. A Computational Approach to Edge Detection, IEEE Trans. on Pattern Analysis and Machine Intelligence, 8(6), pp. 679-698 (1986).
7. [Davis97] J. Davis and Bobick. The Representation and Recognition of Action Using Temporal Templates. MIT Media Lab Technical Report 402, 1997.
8. [DeMenthon92] Daniel F. DeMenthon and Larry S. Davis. Model-Based Object Pose in 25 Lines of Code. In Proceedings of ECCV '92, pp. 335-343, 1992.
9. [Fitzgibbon95] Andrew W. Fitzgibbon, R.B.Fisher. A Buyer's Guide to Conic Fitting. Proc.5th British Machine Vision Conference, Birmingham, pp. 513-522, 1995.
10. [Horn81] Berthold K.P. Horn and Brian G. Schunck. Determining Optical Flow. Artificial Intelligence, 17, pp. 185-203, 1981.
11. [Hu62] M. Hu. Visual Pattern Recognition by Moment Invariants, IRE

Transactions on Information Theory, 8:2, pp. 179-187, 1962.

12. [Iivarinen97] Jukka Iivarinen, Markus Peura, Jaakko Srel, and Ari Visa. Comparison of Combined Shape Descriptors for Irregular Objects, 8th British Machine Vision Conference, BMVC'97. http://www.cis.hut.fi/research/IA/paper/publications/bmvc97/bmvc97.html

13. [Jahne97] B. Jahne. Digital Image Processing. Springer, New York, 1997.

14. [Lucas81] Lucas, B., and Kanade, T. An Iterative Image Registration Technique with an Application to Stereo Vision, Proc. of 7th International Joint Conference on Artificial Intelligence (IJCAI), pp. 674-679.

15. [Kass88] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models, International Journal of Computer Vision, pp. 321-331, 1988.

16. [Lienhart02] Rainer Lienhart and Jochen Maydt. An Extended Set of Haar-like Features for Rapid Object Detection. IEEE ICIP 2002, Vol. 1, pp. 900-903, Sep. 2002.
This paper, as well as the extended technical report, can be retrieved at http://www.lienhart.de/Publications/publications.html

17. [Matas98] J.Matas, C.Galambos, J.Kittler. Progressive Probabilistic Hough Transform. British Machine Vision Conference, 1998.

18. [Rosenfeld73] A. Rosenfeld and E. Johnston. Angle Detection on Digital Curves. IEEE Trans. Computers, 22:875-878, 1973.

19. [RubnerJan98] Y. Rubner. C. Tomasi, L.J. Guibas. Metrics for Distributions with Applications to Image Databases. Proceedings of the 1998 IEEE International Conference on Computer Vision, Bombay, India, January 1998, pp. 59-66.

20. [RubnerSept98] Y. Rubner. C. Tomasi, L.J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. Technical Report STAN-CS-TN-98-86, Department of Computer Science, Stanford University, September

1998.

21. [RubnerOct98] Y. Rubner. C. Tomasi. Texture Metrics. Proceeding of the IEEE International Conference on Systems, Man, and Cybernetics, San-Diego, CA, October 1998, pp. 4601-4607.
http://robotics.stanford.edu/~rubner/publications.html
22. [Serra82] J. Serra. Image Analysis and Mathematical Morphology. Academic Press, 1982.
23. [Schiele00] Bernt Schiele and James L. Crowley. Recognition without Correspondence Using Multidimensional Receptive Field Histograms. In International Journal of Computer Vision 36 (1), pp. 31-50, January 2000.
24. [Suzuki85] S. Suzuki, K. Abe. Topological Structural Analysis of Digital Binary Images by Border Following. CVGIP, v.30, n.1. 1985, pp. 32-46.
25. [Teh89] C.H. Teh, R.T. Chin. On the Detection of Dominant Points on Digital Curves. - IEEE Tr. PAMI, 1989, v.11, No.8, p. 859-872.
26. [Trucco98] Emanuele Trucco, Alessandro Verri. Introductory Techniques for 3-D Computer Vision. Prentice Hall, Inc., 1998.
27. [Viola01] Paul Viola and Michael J. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. IEEE CVPR, 2001.
The paper is available online at http://www.ai.mit.edu/people/viola/
28. [Welch95] Greg Welch, Gary Bishop. An Introduction To the Kalman Filter. Technical Report TR95-041, University of North Carolina at Chapel Hill, 1995.
Online version is available at
http://www.cs.unc.edu/~welch/kalman/kalman_filter/kalman.html
29. [Williams92] D. J. Williams and M. Shah. A Fast Algorithm for Active Contours and Curvature Estimation. CVGIP: Image Understanding, Vol. 55, No. 1, pp. 14-26, Jan., 1992.
http://www.cs.ucf.edu/~vision/papers/shah/92/WIS92A.pdf.

30. [Yuille89] A.Y.Yuille, D.S.Cohen, and P.W.Hallinan. Feature Extraction from Faces Using Deformable Templates in CVPR, pp. 104–109, 1989.

31. [Zhang96] Z. Zhang. Parameter Estimation Techniques: A Tutorial with Application to Conic Fitting, Image and Vision Computing Journal, 1996.

32. [Zhang99] Z. Zhang. Flexible Camera Calibration By Viewing a Plane From Unknown Orientations. International Conference on Computer Vision (ICCV′99), Corfu, Greece, pages 666–673, September 1999.

33. [Zhang00] Z. Zhang. A Flexible New Technique for Camera Calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(11):1330–1334, 2000.