CvPoint

基于二维整形坐标轴的点

```
typedef struct CvPoint
 {
     int x; /* X 坐标, 通常以 0 为基点 */
     int y; /* y 坐标，通常以 0 为基点 */
 }
 CvPoint;
 /* 构造函数 */
 inline CvPoint cvPoint( int x, int y );
 /* 从 CvPoint2D32f 类型转换得来 */
```

```
inline CvPoint cvPointFrom32f( CvPoint2D32f point );
```

CvPoint2D32f

二维浮点坐标上的点

```
typedef struct CvPoint2D32f
    {
        float x; /* X 坐标，通常以 0 为基点*/
        float y; /* Y 坐标，通常以 0 为基点*/
    }
    CvPoint2D32f;
     /* 构造函数 */
    inline CvPoint2D32f cvPoint2D32f( double x, double y );
    /* 从 CvPoint 转换来 */
    inline CvPoint2D32f cvPointTo32f( CvPoint point );
```

CvPoint3D32f

三维浮点坐标上的点

```
    typedef struct CvPoint3D32f
    {
        float x; /* x-坐标，通常基于 0 */
        float y; /* y-坐标，通常基于 0*/
        float z; /* z-坐标，通常基于 0 */
    }
    CvPoint3D32f;

    /* 构造函数 */
    inline CvPoint3D32f cvPoint3D32f( double x, double y, double
z );
```

CvSize

矩形框大小，以像素为精度

```
    typedef struct CvSize
    {
        int width; /* 矩形宽 */
        int height; /* 矩形高 */
    }
    CvSize;
    /* 构造函数 */
    inline CvSize cvSize( int width, int height );
```

CvSize2D32f

以低像素精度标量矩形框大小

```
    typedef struct CvSize2D32f
    {
```

```
        float width; /* 矩形宽 */
        float height; /* 矩形高 */
    }
    CvSize2D32f;

    /* 构造函数*/
    inline CvSize2D32f cvSize2D32f( double width, double height );
```

CvRect

矩形框的偏移和大小

```
    typedef struct CvRect
    {
        int x; /* 方形的最左角的 x-坐标 */
      int y; /* 方形的最上或者最下角的 y-坐标 */
      int width; /* 宽 */
        int height; /* 高 */
    }
    CvRect;
    /* 构造函数*/
    inline CvRect cvRect( int x, int y, int width, int height );
```

CvScalar

可存放在 1-，2-，3-，4-TUPLE 类型的捆绑数据的容器

```
    typedef struct CvScalar
    {
        double val[4]
    }
    CvScalar;

   /* 构造函数：用 val0 初始化 val[0]用 val1 初始化 val[1]等等*/
    inline CvScalar cvScalar( double val0, double val1=0,
                                  double val2=0, double val3=0 );
    /* 构造函数：用 val0123 初始化 val0123 */
    inline CvScalar cvScalarAll( double val0123 );

    /* 构造函数：用 val0 初始化 val[0],val[1]…val[3]用 0 初始化 */
    inline CvScalar cvRealScalar( double val0 );
```

CvTermCriteria

迭代算法的终止标准

```
#define CV_TERMCRIT_ITER      1
#define CV_TERMCRIT_NUMBER    CV_TERMCRIT_ITER
#define CV_TERMCRIT_EPS       2
```

```
typedef struct CvTermCriteria
{
    int       type;    /* CV_TERMCRIT_ITER 和
CV_TERMCRIT_EPS 的联合 */
    int       max_iter; /* 迭代的最大数 */
    double epsilon; /* 结果的精确性 */
}
CvTermCriteria;

/* 构造函数 */
inline   CvTermCriteria   cvTermCriteria( int type, int max_iter,
double epsilon );

/* 检查终止标准并且转换使
type=CV_TERMCRIT_ITER+CV_TERMCRIT_EPS，并且满足
max_iter 和 epsilon 限制条件 */
CvTermCriteria cvCheckTermCriteria( CvTermCriteria criteria,
                                    double default_eps,
                                    int default_max_iters );
```

CvMat

多通道矩阵

```
    typedef struct CvMat
    {
        int type; /* CvMat 标识 (CV_MAT_MAGIC_VAL), 元素类型
和标记 */
        int step; /* 以字节为单位的行数据长度*/
        int* refcount; /* 数据参考计数 */
        union
        {
            uchar* ptr;
            short* s;
            int* i;
            float* fl;
            double* db;
        } data; /* data 指针 */

    #ifdef __cplusplus
        union
        {
            int rows;
            int height;
        };

        union
        {
```

```
            int cols;
            int width;
        };
    #else
        int rows; /* 行数 */
        int cols; /* 列数*/
    #endif

    } CvMat;
```

CvMatND

多维、多通道密集数组

```
    typedef struct CvMatND
    {
        int type; /* CvMatND 标识(CV_MATND_MAGIC_VAL), 元素类型和标号*/
        int dims; /* 数组维数 */
        int* refcount; /* 数据参考计数 */

        union
        {
            uchar* ptr;
            short* s;
            int* i;
            float* fl;
            double* db;
        } data; /* data 指针*/

        /* 每维的数据结构 (元素号,以字节为单位的元素之间的距离)是配套定义的 */
        struct
        {
            int size;          int step;
        }
        dim[CV_MAX_DIM];

    } CvMatND;
```

CvSparseMat

多维、多通道稀疏数组

```
    typedef struct CvSparseMat
    {
        int type; /* CvSparseMat 标识 (CV_SPARSE_MAT_MAGIC_VAL), 元素类型和标号 */
        int dims; /* 维数 */
```

```
        int* refcount; /* 参考数量 - 未用 */
        struct CvSet* heap; /* HASH 表节点池 */
        void** hashtable; /* HASH 表:每个入口有一个节点列表，有
相同的 "以 HASH 大小为模板的 HASH 值" */
        int hashsize; /* HASH 表大小 */
        int total; /* 稀疏数组的节点数 */
        int valoffset; /* 数组节点值在字节中的偏移 */
        int idxoffset; /* 数组节点索引在字节中的偏移 */
        int size[CV_MAX_DIM]; /*维大小 */
    } CvSparseMat;
```

IplImage

IPL 图像头

```
    typedef struct _IplImage
    {
        int  nSize;             /* IplImage 大小 */
        int  ID;                /* 版本 (=0)*/
        int  nChannels;         /* 大多数 OPENCV 函数支持 1,2,3 或
4 个通道 */
        int  alphaChannel;      /* 被 OpenCV 忽略 */
        int  depth;             /* 像素的位深度: IPL_DEPTH_8U,
IPL_DEPTH_8S, IPL_DEPTH_16U,
                                   IPL_DEPTH_16S,
IPL_DEPTH_32S, IPL_DEPTH_32F and IPL_DEPTH_64F 可支持
*/
        char colorModel[4]; /* 被 OpenCV 忽略 */
        char channelSeq[4]; /* 同上 */
        int  dataOrder;         /* 0 - 交叉存取颜色通道, 1 - 分开的颜
色通道.
                                   只有 cvCreateImage 可以创建交
叉存取图像 */
        int  origin;           /* 0 - 顶—左结构,
                                   1 - 底—左结构 (Windows
bitmaps 风格) */
        int  align;             /* 图像行排列 (4 or 8). OpenCV 忽略
它，使用 widthStep 代替 */
        int  width;             /* 图像宽像素数 */
        int  height;            /* 图像高像素数*/
        struct _IplROI *roi;/* 图像感兴趣区域. 当该值非空只对该区
域进行处理 */
        struct _IplImage *maskROI; /* 在 OpenCV 中必须置 NULL
*/
        void  *imageId;         /* 同上*/
        struct _IplTileInfo *tileInfo; /*同上*/
        int  imageSize;         /* 图像数据大小(在交叉存取格式下
imageSize=image->height*image->widthStep），单位字节*/
```

```
    char *imageData;   /* 指向排列的图像数据 */
    int   widthStep;   /* 排列的图像行大小，以字节为单位 */
    int   BorderMode[4]; /* 边际结束模式，被 OpenCV 忽略 */
    int   BorderConst[4]; /* 同上 */
    char *imageDataOrigin; /* 指针指向一个不同的图像数据结构（不是必须排列的），是为了纠正图像内存分配准备的 */
}
IplImage;
```

IplImage结构来自于 *Intel Image Processing Library（是其本身所具有的）*. OpenCV 只支持其中的一个子集:

- alpha通道在 OpenCV中被忽略.
- colorModel 和channelSeq 被OpenCV忽略. OpenCV颜色转换的唯一个函数 cvCvtColor把原图像的颜色空间的目标图像的颜色空间作为一个参数.
- 数据顺序 必须是IPL_DATA_ORDER_PIXEL (颜色通道是交叉存取), 然面平面图像的被选择通道可以被处理，就像COI（感兴趣的通道）被设置过一样.
- 当 widthStep 被用于去接近图像行序列，排列是被OpenCV忽略的.
- 不支持maskROI . 处理MASK的函数把他当作一个分离的参数. MASK在 OpenCV 里是 8-bit, 然而在 IPL他是 1-bit.
- 名字信息不支持.
- 边际模式和边际常量是不支持的. 每个 OpenCV 函数处理像素的邻近的像素，通常使用单一的固定代码边际模式.

除了上述限制, OpenCV处理ROI有不同的要求.要求原图像和目标图像的尺寸或 ROI的尺寸必须（根据不同的作操，例如cvPyrDown 目标图像的宽（高）必须等于原图像的宽（高）除 2 ±1)精确匹配,而IPL处理交叉区域，如图像的大小或ROI大小可能是完全独立的。

CvArr

不确定数组

```
typedef void CvArr;
```

CvArr* 仅仅是被用于作函数的参数，用于指示函数接收的数组类型可以不止一个，如 IplImage*, CvMat* 甚至 CvSeq*. 最终的数组类型是在运行时通过分析数组头的前 4 个字节判断。

数组操作

CreateImage

创建头并分配数据

```
IplImage* cvCreateImage( CvSize size, int depth, int channels );
```

`size` 图像宽、高.

depth

图像元素的位深度，可以是下面的其中之一:

IPL_DEPTH_8U - 无符号 8 位整型

IPL_DEPTH_8S - 有符号 8 位整型

IPL_DEPTH_16U - 无符号 16 位整型

IPL_DEPTH_16S - 有符号 16 位整型

IPL_DEPTH_32S - 有符号 32 位整型

IPL_DEPTH_32F - 单精度浮点数

IPL_DEPTH_64F - 双精度浮点数

channels

每个元素（像素）通道号.可以是 1, 2, 3 或 4.通道是交叉存取的，例如通常的彩色图像数据排列是：

b0 g0 r0 b1 g1 r1 ...

虽然通常 IPL 图象格式可以存贮非交叉存取的图像，并且一些 OpenCV 也能处理他, 但是这个函数只能创建交叉存取图像.

函数 cvCreateImage 创建头并分配数据，这个函数是下列的缩写型式

```
header = cvCreateImageHeader(size,depth,channels);
cvCreateData(header);
```

CreateImageHeader

*分配，初始化，并且返回 **IplImage** 结构*

IplImage* cvCreateImageHeader(CvSize size, int depth, int channels);

size 图像宽、高. depth

像深 (见 CreateImage).

channels

通道数 (见 CreateImage).

函数 cvCreateImageHeader 分配, 初始化, 并且返回 IplImage结构.

这个函数相似于：

```
iplCreateImageHeader( channels, 0, depth,
                      channels == 1 ? "GRAY" : "RGB",
                      channels == 1 ? "GRAY" : channels == 3 ?
"BGR" :
                      channels == 4 ? "BGRA" : "",
                      IPL_DATA_ORDER_PIXEL,
IPL_ORIGIN_TL, 4,
                      size.width, size.height,
                      0,0,0,0);
```

然 而 IPL 函 数 不 是 作 为 默 认 的 (见 CV_TURN_ON_IPL_COMPATIBILITY 宏)

ReleaseImageHeader

释放头

void cvReleaseImageHeader(IplImage** image);

`image`
双指针指向头内存分配单元.
函数 cvReleaseImageHeader 释放头. 相似于

```
if( image )
{
    iplDeallocate( *image, IPL_IMAGE_HEADER |
IPL_IMAGE_ROI );
    *image = 0;
}
```

然 而 IPL 函 数 不 是 作 为 默 认 的 (见 CV_TURN_ON_IPL_COMPATIBILITY 宏)

ReleaseImage

释放头和图像数据

```
void cvReleaseImage( IplImage** image );
```

`image`
双指针指向图像内存分配单元。
函数 cvReleaseImage 释放头和图像数据，相似于：

```
if( *image )
{
    cvReleaseData( *image );
    cvReleaseImageHeader( image );
}
```

InitImageHeader

初始他被用图分配的图像头

```
IplImage* cvInitImageHeader( IplImage* image, CvSize size, int
depth,
                             int channels, int origin=0, int
align=4 );
```

`image`
被初始化的图像头.
`size`
图像的宽高.
`depth`
像深(见 CreateImage).
`channels`
通道数(见 CreateImage).
`origin`
IPL_ORIGIN_TL 或 IPL_ORIGIN_BL.
`align`
图像行排列, 典型的 4 或 8 字节.
函数 cvInitImageHeader 初始他图像头结构, 指向用户指定的图像并且返回这个指针。

CloneImage

制作图像的完整拷贝

IplImage* cvCloneImage(const IplImage* image);

`image`

原图像.

函数 cvCloneImage 制作图像的完整拷贝包括头、ROI和数据

SetImageCOI

基于给定的值设置感兴趣通道

void cvSetImageCOI(IplImage* image, int coi);

`image`

图像头.

`coi`

感兴趣通道.

函数 cvSetImageCOI 基于给定的值设置感兴趣的通道。值 0 意味着所有的通道都被选定, 1 意味着第一个通道被选定等等。如果 ROI 是 NULL 并且COI!= 0, ROI 被分配. 然而大多数的 OpenCV 函数不支持 COI, 对于这种状况当处理分离图像/矩阵通道时，可以拷贝（通过 cvCopy 号 cvSplit) 通道来分离图像/矩阵，处理后如果需要可再拷贝（通过cvCopy 或 cvCvtPlaneToPix）回来.

GetImageCOI

返回感兴趣通道号

int cvGetImageCOI(const IplImage* image);

`image`

图像头.

函数cvGetImageCOI 返回图像的感兴趣通道（当所有的通道都被选中返回值是 0).

SetImageROI

基于给定的矩形设置感兴趣区域

void cvSetImageROI(IplImage* image, CvRect rect);

`image`

图像头.

`rect`

ROI 矩形.

函数 cvSetImageROI 基于给定的矩形设置图像的 ROI（感兴趣区域） . 如果ROI是NULL 并且参数RECT的值不等于整个图像, ROI被分配. 不像 COI, 大多数的 OpenCV 函数支持 ROI 并且处理它就行它是一个分离的图像 (例如, 所有的像素坐标从ROI的顶-左或底-左角（基于图像的结构）计算。

ResetImageROI

*释放图偈的 **ROI***

void cvResetImageROI(IplImage* image);

`image`

图像头.

函数 cvResetImageROI 释放图像 ROI. 释放之后整个图像被认为是全部被选中的。相似的结果可以通过下述办法

cvSetImageROI(image, cvRect(0, 0, image->width,
image->height));
cvSetImageCOI(image, 0);

但是后者的变量不分配 image->roi.

GetImageROI

*返回图像的 **ROI** 坐标*

CvRect cvGetImageROI(const IplImage* image);

`image`

图像头.

函数 cvGetImageROI 返回图像ROI 坐标. 如果没有ROI则返回矩形值为 cvRect(0,0,image->width,image->height)

CreateMat

创建矩阵

CvMat* cvCreateMat(int rows, int cols, int type);

`rows`

矩阵行数. `cols` 矩阵列数. `type`

矩阵元素类型. 通常以 CV_<bit_depth>(S|U|F)C<number_of_channels> 型式描述, 例如:
CV_8UC1 意思是一个 8-bit 无符号单通道矩阵, CV_32SC2 意思是一个 32-bit 有符号二个通道的矩阵.

函数 cvCreateMat 为新的矩阵分配头和下面的数据, 并且返回一个指向新创建的矩阵的指针. 是下列操作的缩写型式:

```
CvMat* mat = cvCreateMatHeader( rows, cols, type );
cvCreateData( mat );
```

矩阵按行存贮. 所有的行以 4 个字节排列。

CreateMatHeader

创建新的矩阵头

CvMat* cvCreateMatHeader(int rows, int cols, int type);

`rows`

矩阵行数.

`cols`

矩阵列数.

`type`

矩阵元素类型(见 cvCreateMat).

函数 cvCreateMatHeader 分配新的矩阵头并且返回指向它的指针. 矩阵数据可被进一步的分配，使用cvCreateData 或通过 cvSetData明确的分配数据.

ReleaseMat

删除矩阵

void cvReleaseMat(CvMat** mat);

`mat`

双指针指向矩阵.

函数cvReleaseMat 缩减矩阵数据参考计数并且释放矩阵头 :

```
if( *mat )
    cvDecRefData( *mat );
cvFree( (void**)mat );
```

InitMatHeader

初始化矩阵头

CvMat* cvInitMatHeader(CvMat* mat, int rows, int cols, int type,
void* data=NULL, int
step=CV_AUTOSTEP);

`mat`

指针指向要被初始化的矩阵头.

`rows`

矩阵的行数.

`cols`

矩阵的列数.

`type`

矩阵元素类型.

`data`

可选的，将指向数据指针分配给矩阵头.

`step`

排列后的数据的整个行宽，默认状态下，使用 STEP 的最小可能值。例如假定矩阵的行与行之间无隙

函数 cvInitMatHeader 初始化已经分配了的 CvMat 结构. 它可以被OpenCV矩阵函数用于处理原始数据。

例如, 下面的代码计算通用数组格式存贮的数据的矩阵乘积.

计算两个矩阵的积

```
double a[] = { 1, 2, 3, 4
               5, 6, 7, 8,
               9, 10, 11, 12 };

double b[] = { 1, 5, 9,
               2, 6, 10,
               3, 7, 11,
               4, 8, 12 };
```

```
double c[9];
CvMat Ma, Mb, Mc ;

cvInitMatHeader( &Ma, 3, 4, CV_64FC1, a );
cvInitMatHeader( &Mb, 4, 3, CV_64FC1, b );
cvInitMatHeader( &Mc, 3, 3, CV_64FC1, c );

cvMatMulAdd( &Ma, &Mb, 0, &Mc );
// c 数组存贮 a(3x4) 和 b(4x3) 矩阵的积
```

Mat

初始化矩阵的头(轻磅变量)

```
CvMat cvMat( int rows, int cols, int type, void* data=NULL );
```

`rows`

矩阵行数

`cols`

列数.

`type`

元素类型(见 CreateMat).

`data`

可选的分配给矩阵头的数据指针 .

函数 cvMat 是个一快速内连函数，替代函数 cvInitMatHeader. 也就是说他相当于:

```
CvMat mat;
cvInitMatHeader( &mat, rows, cols, type, data,
CV_AUTOSTEP );
```

CloneMat

创建矩阵拷贝

```
CvMat* cvCloneMat( const CvMat* mat );
```

`mat`

输入矩阵.

函数 cvCloneMat 创建输入矩阵的一个拷贝并且返回 该矩阵的指针.

CreateMatND

创建多维密集数组

```
CvMatND* cvCreateMatND( int dims, const int* sizes, int type );
```

`dims`

数组维数. 但不许超过 CV_MAX_DIM (默认=32，但这个默认值可能在编译时被改变)的定义

`sizes`

数组的维大小.

`type`

数组元素类型. 与 CvMat相同

函数cvCreateMatND 分配头给多维密集数组并且分配下面的数据，返回指向被创建数组的指针 . 是下列的缩减形式:

```
CvMatND* mat = cvCreateMatNDHeader( dims, sizes, type );
cvCreateData( mat );
```

矩阵按行存贮. 所有的行以 4 个字节排列。.

CreateMatNDHeader

创建新的数组头

CvMatND* cvCreateMatNDHeader(int dims, const int* sizes, int type);

`dims`

数组维数.

`sizes`

维大小.

`type`

数组元素类型. 与 CvMat相同

函数cvCreateMatND 分配头给多维密集数组。数组数据可以用 cvCreateData 进一步的被分配或利用cvSetData由用户明确指定.

ReleaseMatND

删除多维数组

void cvReleaseMatND(CvMatND** mat);

`mat`

指向数组的双指针.

函数 cvReleaseMatND 缩减数组参考计数并释放数组头:

```
if( *mat )
    cvDecRefData( *mat );
cvFree( (void**)mat );
```

InitMatNDHeader

初始化多维数组头

CvMatND* cvInitMatNDHeader(CvMatND* mat, int dims, const int* sizes, int type, void* data=NULL);

`mat`

指向要被出初始化的数组头指针.

`dims`

数组维数.

`sizes`

维大小.

`type`

数组元素类型. 与 CvMat相同

`data`

可选的分配给矩阵头的数据指针.

函数 cvInitMatNDHeader 初始化 用户指派的CvMatND 结构.

CloneMatND

创建多维数组的完整拷贝

CvMatND* cvCloneMatND(const CvMatND* mat);

`mat`

输入数组

函数 cvCloneMatND 创建输入数组的拷贝并返回指针.

DecRefData

缩减数组数据的参考计数

void cvDecRefData(CvArr* arr);

`arr`

数组头.

函数 cvDecRefData 缩减CvMat 或CvMatND 数据的参考计数，如参考计数指针非NULL并且计数到 0 就删除数据，在当前的执行中只有当数据是用cvCreateData 分配的参考计算才会是非NULL，换句话说 ：
使用cvSetData指派外部数据给头
矩阵头代表部分大的矩阵或图像
矩阵头是从图像头或N维矩阵头转换过来的。
参考计数如果被设置成NULL就不会被缩减. 无论数据是否被删除,数据指针和参考计数指针都将被这个函数清空。

IncRefData

增加数组数据的参考计数

int cvIncRefData(CvArr* arr);

`arr`

数组头.

函数 cvIncRefData 增加 CvMat 或 CvMatND 数据参考计数，如果参考计数非空返回新的计数值 否则返回 0。

CreateData

分配数组数据

void cvCreateData(CvArr* arr);

`arr`

数组头.

函数 cvCreateData 分配图像,矩阵或多维数组数据. 对于矩阵类型使用OpenCV的分配函数，对于 IplImage类型如果 CV_TURN_ON_IPL_COMPATIBILITY没有被调用也是可以使用这种方法的反之使用 IPL 函数分配数据

ReleaseData

释放数组数据

void cvReleaseData(CvArr* arr);

`arr`

数组头

函数cvReleaseData 释放数组数据. 对于 CvMat 或 CvMatND 结构只需调用 cvDecRefData(), 也就是说这个函数不能删除外部数据。见 cvCreateData.

SetData

指派用户数据给数组头

void cvSetData(CvArr* arr, void* data, int step);

`arr`

数组头.

`data`

用户数据.

`step`

整行字节长.

函数cvSetData 指派用记数据给数组头. 头应该已经使用 cvCreate*Header, cvInit*Header 或 cvMat (对于矩阵)初始化过.

GetRawData

返回组数的底层信息

```
void cvGetRawData( const CvArr* arr, uchar** data,
                   int* step=NULL, CvSize* roi_size=NULL );
```

`arr`

数组头.

`data`

输出指针，指针指向整个图像的结构或 ROI

`step`

输出行字节长

`roi_size`

输出 ROI 尺寸

函数 cvGetRawData 添充给输出变量数组的底层信息。所有的输出参数是可选的 ，因此这些指针可设为NULL. 如果数组是设置了ROI的 IplImage 结构, ROI参数被返回。

接下来的例子展示怎样去接近数组元素。

使用 GetRawData 计算单通道浮点数组的元素绝对值.

```
float* data;
int step;

CvSize size;
int x, y;

cvGetRawData( array, (uchar**)&data, &step, &size );
step /= sizeof(data[0]);

for( y = 0; y < size.height; y++, data += step )
    for( x = 0; x < size.width; x++ )
        data[x] = (float)fabs(data[x]);
```

GetMat

从不确定数组返回矩阵头

CvMat* cvGetMat(const CvArr* arr, CvMat* header, int* coi=NULL, int allowND=0);

`arr`

输入数组.

`header`

指向 CvMat结构的指针，作为临时缓存 .

`coi`

可选的输出参数，用于输出 COI.

`allowND`

如果非 0，函数就接收多维密集数组 (CvMatND*)并且返回 2D (如果 CvMatND 是二维的）或 1D 矩阵(当 CvMatND 是一维或多于二维). 数组必须是连续的.

函数 cvGetMat从输入的数组生成矩阵头，输入的数组可以是 - CvMat 结构, IplImage结构 或多维密集数组 CvMatND* (后者只有当 allowND != 0 时才可以使用) . 如果是矩阵函数只是返回指向矩阵的指针.如果是 IplImage* 或 CvMatND* 函数用当前图像的ROI初始化头结构并且返回指向这个临时结构的指针。因为CvMat不支持COI，所以他们的返回结果是不同的.

这个函数提供了一个简单的方法，用同一代码处理 IplImage 和 CvMat二种数据类型。这个函数的反向转换可以用 cvGetImage将 CvMat 转换成 IplImage .

输入的数组必须有已分配好的底层数据或附加的数据，否则该函数将调用失败

如果输入的数组是IplImage 格式，使用平面式数据编排并设置了COI，函数返回的指针指向被选定的平面并设置COI=0.利用OPENCV函数对于多通道平面编排图像可以处理每个平面。

GetImage

从不确定数组返回图像头

IplImage* cvGetImage(const CvArr* arr, IplImage* image_header);

`arr`

输入数组.

`image_header`

指向IplImage结构的指针，该结构存贮在一个临时缓存 .

函数 cvGetImage 从输出数组获得图头，该数组可是以矩阵- CvMat*, 或图像 - IplImage*. 如果是图像函数只是返回输入参数的指针，如是查 CvMat* 函数用输入参数矩阵初始化图像头。因此如果我们把 IplImage 转换成 CvMat 然后再转换 CvMat 回 IplImage,如果ROI被设置过了我们可能会获得不同的头，这样一些计算图像跨度的IPL函数就会失败。

CreateSparseMat

创建稀疏数组

CvSparseMat* cvCreateSparseMat(int dims, const int* sizes, int type);

`dims`

数组维数。相对于密集型矩阵，稀疏数组的维数是不受限制的（最多可达 2^{16}).

`sizes`

数组的维大小.

`type`

数组元素类型，见 CvMat

函数 cvCreateSparseMat 分配多维稀疏数组. Initially the array contain no elements, that is cvGet*D or cvGetReal*D return zero for every index

---

ReleaseSparseMat

删除稀疏数组

void cvReleaseSparseMat(CvSparseMat** mat);

`mat`

双指针指向数组。

函数 cvReleaseSparseMat释放稀疏数组并清空数组指针

CloneSparseMat

创建稀疏数组的拷贝

CvSparseMat* cvCloneSparseMat(const CvSparseMat* mat);

`mat`

输放的数组。

函数 cvCloneSparseMat 创建输入数组的拷贝并返回指向这个拷贝的指针.

获取元素和数组子集

GetSubRect

根据输入的图像或矩阵的矩形数组子集返回矩阵头

CvMat* cvGetSubRect(const CvArr* arr, CvMat* submat, CvRect rect);

`arr`

输入数组

`submat`

指针指向结果数组头 Pointer to the resultant sub-array header.

`rect`

以 0 坐标为基准的 ROI.

函数 cvGetSubRect 根据指定的数组矩形返回矩阵头，换句话说，函数允许处理输入数组的指定的一个子矩形，就像一个独立的数组一样

进行处理。函数在处理时要考虑进输入数组的ROI，因此数组的ROI是实际上被提取的。

GetRow, GetRows

返回数组的行或在一定跨度内的行

CvMat* cvGetRow(const CvArr* arr, CvMat* submat, int row);

CvMat* cvGetRows(const CvArr* arr, CvMat* submat, int start_row, int end_row, int delta_row=1);

`arr`

输入数组.

`submat`

指向返回的子数组头的指针.

`row`

被选定的行号下标，以 0 为基准.

`start_row`

跨度的开始行（包括此行）索引下标，以 0 为下标基准

`end_row`

跨度的结束行（不包括此行）索引下标，以 0 为下标基准.

`delta_row`

在跨度内的索引下标跨步，从开始行到结束行每隔delta_row行提取一行。

函数GetRow 和 GetRows 根据指定的行或跨度从输入数组中返回对应的头。 GetRow 是 cvGetRows的缩写:

cvGetRow(arr, submat, row) ~ cvGetRows(arr, submat, row, row + 1, 1);

GetCol, GetCols

返回数组的列或一定跨度内的列

CvMat* cvGetCol(const CvArr* arr, CvMat* submat, int col);

CvMat* cvGetCols(const CvArr* arr, CvMat* submat, int start_col, int end_col);

`arr`

输入数组

`submat`

指向结果子数组头指针.

`col`

选定的列索引下标，该下标以 0 为基准。

`start_col`

跨度的开始列（包括该列）索引下标，该下标以 0 为基准。

`end_col`

跨度的结束列（不包括该列）索引下标，该下标以 0 为基准。

函数 GetCol 和 GetCols 根据指定的列/列跨度返回头。GetCol 是 cvGetCols的缩写形式:

cvGetCol(arr, submat, col); // ~ cvGetCols(arr, submat, col, col + 1);

GetDiag

返回一个数组对角线

CvMat* cvGetDiag(const CvArr* arr, CvMat* submat, int diag=0);

`arr`

输入数组.

`submat`

指向结果子集的头指针.

`diag`

数组对角线。0 是主对角线，-1 是主对角线上面对角线，1 是主对角线下对角线，以此类推。

函数 cvGetDiag 根据指定的diag参数返回数组的对角线头。

GetSize

返回图像或矩阵 ROI 大小

CvSize cvGetSize(const CvArr* arr);

`arr`

数组头.

函数 cvGetSize 返回矩阵或图像的行数和列数，如果是图像就返回ROI的大小

InitSparseMatIterator

初始化稀疏数线元素迭代器

CvSparseNode* cvInitSparseMatIterator(const CvSparseMat* mat,
CvSparseMatIterator*
mat_iterator);

`mat`

输入的数组.

`mat_iterator`

被初始化迭代器.

函数 cvInitSparseMatIterator 初始化稀疏数组元素的迭代器并且返回指向第一个元素的指针，如果数组为空则返回NULL。

GetNextSparseNode

初始化稀疏数线元素迭代器

CvSparseNode* cvGetNextSparseNode(CvSparseMatIterator*
mat_iterator);

`mat_iterator`

稀疏数组的迭代器

函数cvGetNextSparseNode 移动迭代器到下一个稀疏矩阵元素并返回指向他的指针。在当前的版本不存在任何元素的特殊顺序，因为元素是按HASH表存贮的下面的列子描述怎样在稀疏矩阵上迭代 ：

利用cvInitSparseMatIterator 和cvGetNextSparseNode 计算浮点稀疏数组的和。

```
double sum;
```

```
    int i, dims = cvGetDims( array );
    CvSparseMatIterator mat_iterator;
    CvSparseNode* node = cvInitSparseMatIterator( array,
&mat_iterator );

    for( ; node != 0; node = cvGetNextSparseNode( &mat_iterator ))
    {
        int* idx = CV_NODE_IDX( array, node ); /* get pointer to the
element indices */
        float val = *(float*)CV_NODE_VAL( array, node ); /* get
value of the element

(assume that the type is CV_32FC1) */
        printf( "(" );
        for( i = 0; i < dims; i++ )
            printf( "%4d%s", idx[i], i < dims - 1 "," : "): " );
        printf( "%g\n", val );

        sum += val;
    }

    printf( "\nTotal sum = %g\n", sum );
```

GetElemType

返回数组元素类型

```
int cvGetElemType( const CvArr* arr );
```

`arr`

输入数组.

函数 GetElemType 返回数组元素类型就像在cvCreateMat 中讨论的一样:

CV_8UC1 ... CV_64FC4

GetDims, GetDimSize

返回数组维数和他们的大小或者殊维的大小

```
int cvGetDims( const CvArr* arr, int* sizes=NULL );
int cvGetDimSize( const CvArr* arr, int index );
```

`arr`

输入数组.

`sizes`

可选的输出数组维尺寸向量，对于 2D 数组第一位是数组行数（高），第二位是数组列数（宽）

`index`

以 0 为基准的维索引下标（对于矩阵 0 意味着行数，1 意味着列数，对于图象 0 意味着高，1 意味着宽。

函数 cvGetDims 返回维数和他们的大小。如果是 IplImage 或 CvMat 总是返回 2，不管图像/矩阵行数。函数 cvGetDimSize 返回特定的维大小（每维的元素数）。例如，接下来的代码使用二种方法计算数组元素总数。

```
// via cvGetDims()
int sizes[CV_MAX_DIM];
int i, total = 1;
int dims = cvGetDims( arr, size );
for( i = 0; i < dims; i++ )
    total *= sizes[i];

// via cvGetDims() and cvGetDimSize()
int i, total = 1;
int dims = cvGetDims( arr );
for( i = 0; i < dims; i++ )
    total *= cvGetDimsSize( arr, i );
```

Ptr*D

返回指向特殊数组元素的指针

```
uchar* cvPtr1D( const CvArr* arr, int idx0, int* type=NULL );
uchar* cvPtr2D( const CvArr* arr, int idx0, int idx1, int* type=NULL );
uchar* cvPtr3D( const CvArr* arr, int idx0, int idx1, int idx2, int*
type=NULL );
uchar* cvPtrND( const CvArr* arr, int* idx, int* type=NULL, int
create_node=1, unsigned* precalc_hashval=NULL );
```

`arr`

输入数组.

`idx0`

元素下标的第一个以 0 为基准的成员

`idx1`

元素下标的第二个以 0 为基准的成员

`idx2`

元素下标的第三个以 0 为基准的成员

`idx`

数组元素下标

`type`

可选的，矩阵元素类型输出参数

`create_node`

可选的，为稀疏矩阵输入的参数。如果这个参数非零就意味着被需要的元素如果不存在就会被创建。

`precalc_hashval`

可选的，为稀疏矩阵设置的输入参数。如果这个指针非 NULL，函数不会重新计算节点的 HASH 值，而是从指定位置获取。这种方法有利于提高智能组合数据的操作（TODO: 提供了一个例子)

函数cvPtr*D 返回指向特殊数组元素的指针。数组维数应该与转递给函数物下标数相匹配，除了 cvPtr1D 函数，它可以被用于顺序存取的1D，2D或nD密集数组
函数也可以用于稀疏数组，并且如果被需要的节点不存在函数可以创建这个节点并设置为 0
就像其它获取数组元素的函数 (cvGet[Real]*D, cvSet[Real]*D)如果元素的下标超出了范围就会产生错误

Get*D

返回特殊的数组元素

CvScalar cvGet1D(const CvArr* arr, int idx0);
CvScalar cvGet2D(const CvArr* arr, int idx0, int idx1);
CvScalar cvGet3D(const CvArr* arr, int idx0, int idx1, int idx2);
CvScalar cvGetND(const CvArr* arr, int* idx);

`arr`
输入数组.

`idx0`
元素下标第一个以 0 为基准的成员

`idx1`
元素下标第二个以 0 为基准的成员

`idx2`
元素下标第三个以 0 为基准的成员

`idx`
元素下标数组

函数cvGet*D 返回指定的数组元素。对于稀疏数组如果需要的节点不存在函数返回 0 （不会创建新的节点）

GetReal*D

返回单通道数组的指定元素

double cvGetReal1D(const CvArr* arr, int idx0);
double cvGetReal2D(const CvArr* arr, int idx0, int idx1);
double cvGetReal3D(const CvArr* arr, int idx0, int idx1, int idx2);
double cvGetRealND(const CvArr* arr, int* idx);

`arr`
输入数组，必须是单通道.

`idx0`
元素下标的第一个成员，以 0 为基准

`idx1`
元素下标的第二个成员，以 0 为基准

`idx2`
元素下标的第三个成员，以 0 为基准

`idx`
元素下标数组

函数cvGetReal*D 返回单通道数组的指定元素，如果数组是多通道的，就会产生运行时错误，而 cvGet*D 函数可以安全的被用于单通道和多通道数组，但他们运行时会有点慢
如果指定的点不存在对于稀疏数组点会返回 0（不会创建新的节点）。

---

mGet

返回单通道浮点矩阵指定元素

double cvmGet(const CvMat* mat, int row, int col);

`mat`

输入矩阵.

`row`

行下标，以 0 为基点.

`col`

列下标，以 0 为基点

函数 cvmGet 是 cvGetReal2D对于单通道浮点矩阵的快速替代函数，函数运行比较快速因为它是内连函数 ，这个函数对于数组类型、数组元素类型的检查作的很少，并且仅在调式模式下检查数的行和列范围。

Set*D

修改指定的数组

void cvSet1D(CvArr* arr, int idx0, CvScalar value);
void cvSet2D(CvArr* arr, int idx0, int idx1, CvScalar value);
void cvSet3D(CvArr* arr, int idx0, int idx1, int idx2, CvScalar value);
void cvSetND(CvArr* arr, int* idx, CvScalar value);

`arr`

输入数组

`idx0`

元素下标的第一个成员，以 0 为基点

`idx1`

元素下标的第二个成员，以 0 为基点

`idx2`

元素下标的第三个成员，以 0 为基点

`idx`

元素下标数组

`value`

指派的值

函数 cvSet*D 指定新的值给指定的数组元素。对于稀疏矩阵如果指定节点不存在函数创建新的节点

SetReal*D

修改指定数组元素值

void cvSetReal1D(CvArr* arr, int idx0, double value);
void cvSetReal2D(CvArr* arr, int idx0, int idx1, double value);

void cvSetReal3D(CvArr* arr, int idx0, int idx1, int idx2, double value);
void cvSetRealND(CvArr* arr, int* idx, double value);
`arr`
输入数组.
`idx0`
元素下标的第一个成员，以 0 为基点
`idx1`
元素下标的第二个成员，以 0 为基点
`idx2`
元素下标的第三个成员，以 0 为基点
`idx`
元素下标数组
`value`
指派的值
函数 cvSetReal*D 分配新的值给单通道数组的指定元素，如果数组是多通道就会产生运行时错误。然而cvSet*D 可以安全的被用于多通道和单通道数组，只是稍微有点慢。
对于稀疏数组如果指定的节点不存在函数会创建该节点。

---

mSet
返回单通道浮点矩阵的指定元素
void cvmSet(CvMat* mat, int row, int col, double value);
`mat`
矩阵.
`row`
行下标,以 0 为基点.
`col`
列下标,以 0 为基点.
`value`
矩阵元素的新值
函数cvmSet 是cvSetReal2D 快速替代,对于单通道浮点矩阵因为这个函数是内连的所以比较快,函数对于数组类型、数组元素类型的检查作的很少，并且仅在调式模式下检查数的行和列范围。

Clear*D
清除指定数组元素
void cvClearND(CvArr* arr, int* idx);
`arr`
输入数组.
`idx`
数组元素下标
函数cvClearND 清除指定密集型数组的元素(置 0)或删除稀疏数组的元素 ,如果元素不存在函数不作任何事

拷贝和添加

Copy

拷贝一个数组给另一个数组

void cvCopy(const CvArr* src, CvArr* dst, const CvArr* mask=NULL);

`src`

输入数组.

`dst`

输出数组

`mask`

Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

函数 cvCopy copies selected elements from input array to output array:

dst(I)=src(I) if mask(I)!=0.

If any of the passed arrays is of IplImage type, then its ROI and COI fields are used. Both arrays must have the same type, the same number of dimensions and the same size. 函数 can also copy sparse arrays (mask is not supported in this case).

Set

Sets every element of array to given value

void cvSet(CvArr* arr, CvScalar value, const CvArr* mask=NULL);

`arr`

The destination array.

`value`

Fill value.

`mask`

操作覆盖面,8 位单通道数组 ；输出数组中的覆盖面指定元素被除数修改.

函数 cvSet 拷贝数量值到输出数组的每一个被除数选定的元素:

arr(I)=value if mask(I)!=0

如果数组 arr 是 IplImage 类型，那么就会使用ROI,但 COI不能设置 .

SetZero

清空数组

void cvSetZero(CvArr* arr);

#define cvZero cvSetZero

`arr`

要被清空数组.

函数 cvSetZero 清空数组. 对于密集型号数组(CvMat, CvMatND or IplImage) cvZero(array) 就相当于 cvSet(array,cvScalarAll(0),0), 对于稀疏数组所有的元素都将被删除.

变换和置换

Reshape

不拷贝数据修改矩阵/图像形状

CvMat* cvReshape(const CvArr* arr, CvMat* header, int new_cn, int new_rows=0);

arr

输入的数组.

header

被添充的矩阵头

new_cn

新的通道数.new_cn = 0 意味着不修改通道数

new_rows

新的行数. 如果new_rows = 0 保持原行数不修改否则根据 new_cn 值修改输出数组

函数 cvReshape 初始化 CvMat 头header 以便于让头指向修改后的形状（但数据保持原样）-也就是说修改通道数,修改行数或者两者者改变.

例如, 接下来的代码创建一个图像缓存、两个图像头，第一个是 320x240x3 图像第二个是 960x240x1 图像:

```
IplImage* color_img = cvCreateImage( cvSize(320,240),
IPL_DEPTH_8U, 3 );
CvMat gray_mat_hdr;
IplImage gray_img_hdr, *gray_img;
cvReshape( color_img, &gray_mat_hdr, 1 );
gray_img = cvGetImage( &gray_mat_hdr, &gray_img_hdr );
```

下一个例子转换 3x3 矩阵成单向量 1x9

```
CvMat* mat = cvCreateMat( 3, 3, CV_32F );
CvMat row_header, *row;
row = cvReshape( mat, &row_header, 0, 1 );
```

ReshapeMatND

修改多维数组形状，拷贝/不拷贝数据

```
CvArr* cvReshapeMatND( const CvArr* arr,
                       int sizeof_header, CvArr* header,
                       int new_cn, int new_dims, int* new_sizes );

#define cvReshapeND( arr, header, new_cn, new_dims, new_sizes )
\
        cvReshapeMatND( (arr), sizeof(*(header)), (header),
\
                        (new_cn), (new_dims), (new_sizes))
```

arr
输入数组
sizeof_header
输出头的大小，对于 IplImage, CvMat 和 CvMatND 各种结构输出的头巾是不同的.
header
被添充的输出头.
new_cn
新的通道数，如果new_cn = 0 则通道数保持原样
new_dims
新的维数. 如果new_dims = 0 则维数保持原样。
new_sizes
新的维大小.只有当 只有 new_dims-1 值被使用，因为要保持数组的总数一致，因此如果 new_dims = 1, new_sizes 是不被使用的

函数cvReshapeMatND 是 cvReshape 的高级版本，它可以处理多维数组（能够呛处理通用的图像和矩阵）并且修改维数，下面的是使用cvReshapeMatND重写 cvReshape的二个例子 ：

```
IplImage* color_img = cvCreateImage( cvSize(320,240),
IPL_DEPTH_8U, 3 );
IplImage gray_img_hdr, *gray_img;
gray_img = (IplImage*)cvReshapeND( color_img, &gray_img_hdr, 1,
0, 0 );

...

/*second example is modified to convert 2x2x2 array to 8x1 vector */
int size[] = { 2, 2, 2 };
CvMatND* mat = cvCreateMatND( 3, size, CV_32F );
CvMat row_header, *row;
row = cvReshapeND( mat, &row_header, 0, 1, 0 );
```

Repeat

用原数组管道式添充输出数组

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

src
输入数组, 图像或矩阵。
dst
输出数组，图像或矩阵

函数cvRepeat 使用被管道化的原数组添充输出数组：

```
dst(i,j)=src(i mod rows(src), j mod cols(src))
```

因此 ，输出数组可能小于输入数组

Flip

垂直，水平或即垂直又水平翻转二维数组

```
void    cvFlip( const CvArr* src, CvArr* dst=NULL, int flip_mode=0);
```

#define cvMirror cvFlip

`src`
原数组.
`dst`
目标责任制数组. 如果 dst = NULL 翻转是在内部替换.
`flip_mode`
指定怎样去翻转数组。
flip_mode = 0 沿 X-轴翻转, flip_mode > 0 (如 1) 沿 Y-轴翻转， flip_mode < 0 (如 -1) 沿 X-轴和 Y-轴翻转.见下面的公式
函数cvFlip 以三种方式之一翻转数组 (行和列下标是以 0 为基点的):
dst(i,j)=src(rows(src)-i-1,j) if flip_mode = 0
dst(i,j)=src(i,cols(src1)-j-1) if flip_mode > 0
dst(i,j)=src(rows(src)-i-1,cols(src)-j-1) if flip_mode < 0
函数主要使用在:

- 垂直翻转图像(flip_mode > 0)用于 顶-左和底-左图像结构的转换, 主要用于 WIN32 系统下的视频操作处理.
- 水平图像转换，使用连续的水平转换和绝对值差检查垂直轴对称（flip_mode > 0)
- 水平和垂直同时转换，用于连续的水平转换和绝对真理值差检查中心对称 s(flip_mode < 0)
- 翻转 1 维指针数组的顺序(flip_mode > 0)

Split

分割多通道数组成几个单通道数组或者从数组中提取一个通道

void cvSplit(const CvArr* src, CvArr* dst0, CvArr* dst1,
CvArr* dst2, CvArr* dst3);
#define cvCvtPixToPlane cvSplit
`src`
原数组.
`dst0...dst3`
目标通道
函数 cvSplit 分割多通道数组成分离的单通道数组d。可获得两种操作模式 . 如果原数组有N通道且前N输出数组非NULL, 所有的通道都会被从原数组中提取，如果前N个通道只有一个通道非NULL函数只提取该指定通道，否则会产生一个错误，余下的通道（超过前N个通道的以上的）必须被设置成NULL，对于设置了COI的IplImage 结使用cvCopy 也可以从图像中提取单通道。

Merge

从几个单通道数组组合多通道数组或插入一个单通到数组

void cvMerge(const CvArr* src0, const CvArr* src1,
const CvArr* src2, const CvArr* src3, CvArr* dst);
#define cvCvtPlaneToPix cvMerge

src0... src3
输入的通道.
dst
输出数组.
函数cvMerge 是前一个函数的反向操作。如果输出数组有N个通道并且前N个输入通道非NULL，就拷贝所有通道到输出数组，如果在前N个通道中只有一个单通道非NULL ，只拷贝这个通道到输出数组，否则 就会产生错误。除前N通道以外的余下的通道必须置NULL。对于设置了COI的 IplImage结构使用 cvCopy也可以实现向图像中插入一个通道 。

算术，逻辑和比较

LUT

利用搜索表转换数组

void cvLUT(const CvArr* src, CvArr* dst, const CvArr* lut);

src
元素为 8 位的原数组。
dst
与原数组有相同通道数的输出数组，深度不确定
lut
有 256 个元素的搜索表;必须要与原输出数组有想同像深 。
函数cvLUT 使用搜索中的值添充输出数组. 坐标入口来自于原数组，也就是说函数处理每个元素按如下方式：
dst(I)=lut[src(I)+DELTA]
这里当src的深度是CV_8U时DELTA=0 ,src的深度是CV_8S时DELTA=128

ConvertScale

使用线性变换转换数组

void cvConvertScale(const CvArr* src, CvArr* dst, double scale=1, double shift=0);

#define cvCvtScale cvConvertScale
#define cvScale cvConvertScale
#define cvConvert(src, dst) cvConvertScale((src), (dst), 1, 0)

src
原数组.
dst
输出数组
scale
比例因子.
shift
原数组元素按比例缩放后添加的值。

函数 cvConvertScale 有多个不同的目的因此就有多个意义，函数按比例从一个数组中拷贝元素到另一个元素这种操作是最先执行的，或者任意的类型转换，正如下面的操作：

dst(I)=src(I)*scale + (shift,shift,...)

多通道的数组对各个地区通道是独立处理的。

类型转换主要用于舍入和饱合度，也就是如果缩放 后的结果+转换值不能用输出数组元素类型值精确表达，就设置成最接近该数的值到真正的坐标轴上。

如果 scale=1, shift=0 就不会进行比例缩放. 这是一个特殊的优化，相当cvConvert 。如果原和输出数组的类型相同这同样也是一另一种情形，函数可以被用于比例化并且移动矩阵或图像这就变成 cvScale的同义词了。

ConvertScaleAbs

使用线性变换转换输入数组元素成 8 位无符号整型

void cvConvertScaleAbs(const CvArr* src, CvArr* dst, double scale=1, double shift=0);

#define cvCvtScaleAbs cvConvertScaleAbs

`src`

原数组

`dst`

输出数组 (深度为 8u).

`scale`

比例因子.

`shift`

原数组元素按比例缩放后添加的值。

函数 cvConvertScaleAbs 与前一函数是相同的，但它是存贮变换结果的绝对值：

dst(I)=abs(src(I)*scale + (shift,shift,...))

函数只支持目标数数组的深度为 8u (8-bit 无符号)，对于别的类型函数仿效于cvConvertScale 和 cvAbs 函数的联合

Add

计算两个数中每个元素的和

void cvAdd(const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL);

`src1`

第一个原数组

`src2`

第二个原数组

`dst`

输出数组

`mask`

操作的覆盖面, 8-bit 单通道数组; 只有覆盖面指定的输出数组被修改。

函数 cvAdd 加一个数组到别一个数组中：

dst(I)=src1(I)+src2(I) if mask(I)!=0
除覆盖面外所有的数组必须有相同的类型相同的大小（或 ROI 尺寸）。

AddS

计算数量和数组的和

void cvAddS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);

`src`
原数组.

`value`
被加入数量

`dst`
输出数组

`mask`
操作的覆盖面（8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvAddS 用数量值与原数组src1的每个元素想加并存贮结果到
dst(I)=src(I)+value if mask(I)!=0
除覆盖面外所有数组都必须有相同的类型，相同的大小（或 ROI 大小）

AddWeighted

计算两数组的加磅值的和

```
void    cvAddWeighted( const CvArr* src1, double alpha,
                       const CvArr* src2, double beta,
                       double gamma, CvArr* dst );
```

`src1`
第一个原数组.

`alpha`
第一个数组元素的磅值

`src2`
第二个原数组

`beta`
第二个数组元素的磅值

`dst`
输出数组

`gamma`
作和合添加的数量。

函数 cvAddWeighted 计算两数组的加磅值的和：
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
所有的数组必须的相同的类型相同的大小（或 ROI 大小）

Sub

计算两个数组每个元素的差

void cvSub(const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL);

`src1`
第一个原数组
`src2`
第二个原数组.
`dst`
输出数组.
`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改
函数cvSub 从一个数组减去别一个数组:
dst(I)=src1(I)-src2(I) if mask(I)!=0
除覆盖面外所有数组都必须有相同的类型，相同的大小（或 ROI 大小）

SubS
计算数组和数量之间的差
void cvSubS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);
`src`
原数组.
`value`
被减的数量.
`dst`
输出数组.
`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改
函数 cvSubS 从原数组的每个元素中减去一个数量：
dst(I)=src(I)-value if mask(I)!=0
除覆盖面外所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。

SubRS
计算数量和数组之间的差
void cvSubRS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);
`src`
第一个原数组。
`value`
被减的数量
`dst`
输出数组
`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改
函数 cvSubRS 从一个数量减去原数组的每个元素：
dst(I)=value-src(I) if mask(I)!=0
除覆盖面外所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。

Mul

计算两个数组中每个元素的积

void cvMul(const CvArr* src1, const CvArr* src2, CvArr* dst, double scale=1);

`src1`

第一个原数组.

`src2`

第二个原数组.

`dst`

输出数组.

`scale`

设置的比例因子

函数 cvMul 计算两个数组中每个元素的积：

dst(I)=scale•src1(I)•src2(I)

所有的数组必须有相同的类型和相同的大小（或 ROI 大小）

Div

两个数组每个元素相除

void cvDiv(const CvArr* src1, const CvArr* src2, CvArr* dst, double scale=1);

`src1`

第一个原数组。如该指针为 NULL，假高该数组的所有元素都为 1

`src2`

第二个原数组。

`dst`

输出数组

`scale`

设置的比例因子

函数 cvDiv 用一个数组除以另一个数组:

dst(I)=scale•src1(I)/src2(I), if src1!=NULL

dst(I)=scale/src2(I), if src1=NULL

所有的数组必须有相同的类型和相同的大小（或 ROI 大小）

And

计算两个数组的每个元素的按位与

void cvAnd(const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL);

`src1`

第一个原数组

`src2`

第二个原数组.

`dst`

输出数组

`mask`

操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvAnd 计算两个数组的每个元素的按位逻辑与：
dst(I)=src1(I)&src2(I) if mask(I)!=0
对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。

AndS

计算数组每个元素与数量之间的按位与

void cvAndS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);

`src`
原数组.

`value`
操作中用到的数量

`dst`
输出数组

`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 AndS 计算数组中每个元素与数量之量的按位与：
dst(I)=src(I)&value if mask(I)!=0
在实际操作之前首先把数量类型转换成与数组相同的类型。对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。
接下来的例子描述怎样计算浮点数组元素的绝对值，通过清除最前面的符号位：

```
float a[] = { -1, 2, -3, 4, -5, 6, -7, 8, -9 };
CvMat A = cvMat( 3, 3, CV_32F, &a );
int i, abs_mask = 0x7fffffff;
cvAndS( &A, cvRealScalar(*(float*)&abs_mask), &A, 0 );
for( i = 0; i < 9; i++ )
    printf("%.1f ", a[i] );
```

代码结果是:
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

Or

计算两个数组每个元素的按位或

void cvOr(const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL);

`src1`
第一个原数组

`src2`
第二个原数组

`dst`
输出数组.

`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvOr 计算两个数组每个元素的按位或：
dst(I)=src1(I)|src2(I)
对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。

OrS

计算数组中每个元素与数量之间的按位或

void cvOrS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);

`src1`
原数组

`value`
操作中用到的数量

`dst`
目数组.

`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 OrS 计算数组中每个元素和数量之间的按位或：
dst(I)=src(I)|value if mask(I)!=0
在实际操作之前首先把数量类型转换成与数组相同的类型。对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。

Xor

计算两个数组中的每个元素的按位异或

void cvXor(const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL);

`src1`
第一个原数组

`src2`
第二个原数组.

`dst`
输出数组

`mask`
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvXor 计算两个数组元素的按位异或：
dst(I)=src1(I)^src2(I) if mask(I)!=0
对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。

XorS

计算数组元素与数量之间的按位异或

void cvXorS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL);

src
原数组
value
操作中用到的数量
dst
输出数组.
mask
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改。
函数 XorS 计算数组元素与数量之间的按位异或：
dst(I)=src(I)^value if mask(I)!=0
在实际操作之前首先把数量类型转换成与数组相同的类型。对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或 ROI 大小）。
下面例子描述怎样对共轭复数向量转换，通过转换前面的符号位：

```
float a[] = { 1, 0, 0, 1, -1, 0, 0, -1 }; /* 1, j, -1, -j */
CvMat A = cvMat( 4, 1, CV_32FC2, &a );
int i, neg_mask = 0x80000000;
cvXorS( &A, cvScalar( 0, *(float*)&neg_mask, 0, 0 ), &A, 0 );
for( i = 0; i < 4; i++ )
    printf("(%.1f, %.1f) ", a[i*2], a[i*2+1] );
```

The code should print:
(1.0,0.0) (0.0,-1.0) (-1.0,0.0) (0.0,1.0)

Not
计算数组元素的按位取反

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

src1
原数组
dst
输出数组
函数不取反每个数组元素的每一位
dst(I)=~src(I)

Cmp
比较两个数组元素 P

```
void cvCmp( const CvArr* src1, const CvArr* src2, CvArr* dst, int
cmp_op );
```

src1
第一个原数组
src2
第二个原数组，这两个数组必须有一个单通道
dst
输出数组必须是 8u 或 8s 类型.
cmp_op
该标识指定要检查的元素之间的关系：

CV_CMP_EQ - src1(I) "等于" src2(I)
CV_CMP_GT - src1(I) "大于" src2(I)
CV_CMP_GE - src1(I) "大于等于" src2(I)
CV_CMP_LT - src1(I) "小于" src2(I)
CV_CMP_LE - src1(I) "小于等于" src2(I)
CV_CMP_NE - src1(I) "不等于" src2(I)

函数 cvCmp 比较两个数组的对应元素并且添充输出数组：
dst(I)=src1(I) op src2(I),
这里 op 是 '=', '>', '>=', '<', '<=' or '!='.
如果元素之间的关系为真则设置dst(I)为 0xff (也就是所有的位都为 '1')
否则为 0。除了输出数组所有数组必须是相同的类型相同的大小（或ROI大小）。

CmpS

比较数组的每个元素与数量的关系

void cvCmpS(const CvArr* src, double value, CvArr* dst, int cmp_op);

`src`

原数,必须有一个单通道。

`value`

用与数组元素比较的数量值

`dst`

输出数组必须是 8u 或 8s 类型.

`cmp_op`

该标识指定要检查的元素之间的关系：
CV_CMP_EQ - src1(I) "等于" value
CV_CMP_GT - src1(I) "大于" value
CV_CMP_GE - src1(I) "大于等于" value
CV_CMP_LT - src1(I) "小于" value
CV_CMP_LE - src1(I) "小于等于" value
CV_CMP_NE - src1(I) "不等于" value

函数 cvCmpS 比较数组元素与数量并且添充目标覆盖面数组：
dst(I)=src(I) op scalar,
这里 op 是 '=', '>', '>=', '<', '<=' or '!='.
如果元素之间的关系为真则设置dst(I)为 0xff (也就是所有的位都为 '1')
否则为 0。所有的数组必须有相同的大小（或ROI大小）

InRange

检查数组元素是否在两个数组之间

void cvInRange(const CvArr* src, const CvArr* lower, const CvArr* upper, CvArr* dst);

`src`

第一个原数组

`lower`

包括进的下边界数组

upper
不包括进的上边界线数组
dst
输出数组必须是 8u 或 8s 类型.
函数 cvInRange 对输入的数组作范围检查：
$dst(I)=lower(I)_0 <= src(I)_0 < upper(I)_0$
对于单通道数组：
$dst(I)=lower(I)_0 <= src(I)_0 < upper(I)_0$ &&
$lower(I)_1 <= src(I)_1 < upper(I)_1$
对二通道数组，以此类推
如果 src(I) 在范围内dst(I)被设置为 0xff (每一位都是 '1')否则置 0 。
除了输出数组所有数组必须是相同的类型相同的大小（或ROI大小）。

InRangeS
检查数组元素是否在两个数量之间
void cvInRangeS(const CvArr* src, CvScalar lower, CvScalar upper, CvArr* dst);
src
第一个原数组
lower
包括进的下边界.
upper
不包括进的上边界
dst
输出数组必须是 8u 或 8s 类型.
函数 cvInRangeS 检查输入数组元素范围：
$dst(I)=lower_0 <= src(I)_0 < upper_0$
对于单通道数组：
$dst(I)=lower_0 <= src(I)_0 < upper_0$ &&
$lower_1 <= src(I)_1 < upper_1$
对于双通道数组以此类推
如果 src(I) 在范围内dst(I)被设置为 0xff (每一位都是 '1')否则置 0 。所有的数组必须有相同的大小（或ROI大小）

Max
查找两个数组中每个元素的较大值
void cvMax(const CvArr* src1, const CvArr* src2, CvArr* dst);
src1
第一个原数组
src2
第二个原数组
dst
输出数组
函数 cvMax 计算两个数组中每个元素的较大值：
dst(I)=max(src1(I), src2(I))

所有的数组必须的一个单通道，相同的数据类型和相同的大小（或 ROI 大小）

MaxS

查找数组元素与数量之间的较大值

void cvMaxS(const CvArr* src, double value, CvArr* dst);

`src`

第一个原数组.

`value`

数量值.

`dst`

输出数组

函数 cvMaxS 计算数组元素和数量之间的较大值：

dst(I)=max(src(I), value)

所有数组必须有一个单一通道，相同的数据类型相同的大小（或 ROI 大小）

Min

查找两个数组元素之间 的较小值

void cvMin(const CvArr* src1, const CvArr* src2, CvArr* dst);

`src1`

第一个原数组

`src2`

第二个原数组.

`dst`

输出数组.

函数cvMin计算两个数组元素的较大值

dst(I)=min(src1(I),src2(I))

所有数组必须有一个单一通道，相同的数据类型相同的大小（或 ROI 大小）

MinS

查找数组元素和数量之间的较大值

void cvMinS(const CvArr* src, double value, CvArr* dst);

`src`

第一个原数组

`value`

数量值.

`dst`

输出数组..

函数 cvMinS 计算数组元素和数量之量的较小值：

dst(I)=min(src(I), value)

所有数组必须有一个单一通道，相同的数据类型相同的大小（或 ROI 大小）

AbsDiff

计算两个数组差的绝对值

void cvAbsDiff(const CvArr* src1, const CvArr* src2, CvArr* dst);

`src1`

第一个原数组

`src2`

第二个原数组

`dst`

输出数组

函数 cvAbsDiff 计算两个数组差的绝对值

$dst(I)_c = abs(src1(I)_c - src2(I)_c)$.

所有数组必须有相同的数据类型相同的大小（或 ROI 大小）

AbsDiffS

计算数组元素与数量之间差的绝对值

void cvAbsDiffS(const CvArr* src, CvArr* dst, CvScalar value);

#define cvAbs(src, dst) cvAbsDiffS(src, dst, cvScalarAll(0))

`src`

原数组.

`dst`

输出数组

`value`

数量.

函数 cvAbsDiffS 计算数组元素与数量之间差的绝对值

$dst(I)_c = abs(src(I)_c - value_c)$.

所有数组必须有相同的数据类型相同的大小（或 ROI 大小）

统计

CountNonZero

计算非零数组元素

int cvCountNonZero(const CvArr* arr);

`arr`

数组, 必须是单通道数组或者设置 COI（感兴趣通道）的多通道图像。

函数 cvCountNonZero 返回arr中非零元素的数目:

result = sum_I arr(I)!=0

当 IplImage 支持 ROI 和 COI。

Sum

计算数组元素的和

CvScalar cvSum(const CvArr* arr);

`arr`

数组.

函数 cvSum 独立地为每一个通道计算数组元素的和 S :

$S_c = sum_I arr(I)_c$

如果数组是IplImage类型和设置了COI, 该函数只处理选定的通道并将和存储到第一个标量成员 (S_0)。

```
typedef struct CvScalar
    {
        double val[4]
    }
    CvScalar;
```

我个人认为：CvScalar 用来返回 cvSum 函数的计算结果，由于图像最多可能有四个通道，所以要用 CvScalar 来存放。CvScalar 定义在 cxcore 中

//--

Avg

计算数组元素的平均值

CvScalar cvAvg(const CvArr* arr, const CvArr* mask=NULL);

`arr`

数组.

`mask`

可选操作掩模

函数 cvAvg 独立地为每一个通道计算数组元素的平均值 M :

$N = sum_I mask(I)!=0$

$M_c = 1/N \cdot sum_{I,mask(I)!=0} arr(I)_c$

如果数组是 IplImage 类型和设置 COI ，该函数只处理选定的通道并将和存储到第一个标量成员 (S_0)。

AvgSdv

计算数组元素的平均值

void cvAvgSdv(const CvArr* arr, CvScalar* mean, CvScalar* std_dev, const CvArr* mask=NULL);

`arr`

数组

`mean`

指向平均值的指针， 如果不需要的话可以为空（ NULL）。

`std_dev`
指向标准差的指针。
`mask`
可选操作掩模。
函数 cvAvgSdv 独立地为每一个通道计算数组元素的平均值和标准差：
N = sum_I mask(I)!=0

$mean_c$ = 1/N • $sum_{I,mask(I)!=0}$ $arr(I)_c$

std_dev_c = sqrt(1/N • $sum_{I,mask(I)!=0}$ $(arr(I)_c - M_c)^2$)
如果数组是 IplImage 类型和 设置了COI ,该函数只处理选定的通道并将平均值和标准差存储到第一个输出标量成员 (M_0 and S_0)。

MinMaxLoc

查找数组和子数组的全局最小值和最大值

```
void cvMinMaxLoc( const CvArr* arr, double* min_val, double*
max_val,
                  CvPoint* min_loc=NULL, CvPoint*
max_loc=NULL, const CvArr* mask=NULL );
```

`arr`
输入数组， 单通道或者设置了 COI 的多通道。
`min_val`
指向返回的最小值的指针。
`max_val`
指向返回的最大值的指针。
`min_loc`
指向返回的最小值的位置指针。
`max_loc`
指向返回的最大值的位置指针。
`mask`
选择一个子数组的操作掩模。
函数 MinMaxLoc 查找元素中的最小值和最大值以及他们的位置。函数在整个数组、或选定的ROI区域(对IplImage)或当MASK不为NULL时指定的数组区域中，搜索极值 。如果数组不止一个通道，它就必须是设置了 COI 的 IplImage 类型。 如果是多维数组 min_loc->x 和 max_loc->x 将包含极值的原始位置信息 (线性的)。

Norm

计算数组的绝对范数， 绝对差分范数或者相对差分范数

```
double cvNorm( const CvArr* arr1, const CvArr* arr2=NULL, int
norm_type=CV_L2, const CvArr* mask=NULL );
```

`arr1`
第一输入图像
`arr2`

第二输入图像 ，如果为空（NULL），计算 arr1 的绝对范数，否则计算 arr1-arr2 的绝对范数或者相对范数。

`normType`

范数类型，参见“讨论”。

`mask`

可选操作掩模。

如果 arr2 为空（NULL），函数 cvNorm 计算 arr1 的绝对范数：

norm = $||arr1||_C = max_I abs(arr1(I))$, 如果 normType = CV_C

norm = $||arr1||_{L1} = sum_I abs(arr1(I))$, 如果 normType = CV_L1

norm = $||arr1||_{L2} = sqrt(sum_I arr1(I)^2)$, 如果 normType = CV_L2

如果 arr2 不为空（NULL）， 该函数计算绝对差分范数或者相对差分范数：

norm = $||arr1-arr2||_C = max_I abs(arr1(I)-arr2(I))$, 如果 normType = CV_C

norm = $||arr1-arr2||_{L1} = sum_I abs(arr1(I)-arr2(I))$, 如果 normType = CV_L1

norm = $||arr1-arr2||_{L2} = sqrt(sum_I (arr1(I)-arr2(I))^2)$, 如果 normType = CV_L2

或者

norm = $||arr1-arr2||_C/||arr2||_C$, 如果 normType = CV_RELATIVE_C

norm = $||arr1-arr2||_{L1}/||arr2||_{L1}$, 如果 normType = CV_RELATIVE_L1

norm = $||arr1-arr2||_{L2}/||arr2||_{L2}$, 如果 normType = CV_RELATIVE_L2

函数 Norm 返回计算所得的范数。多通道数组被视为单通道处理，因此，所有通道的结果是结合在一起的。

线性代数

SetIdentity

初始化带尺度的单位矩阵

```
void cvSetIdentity( CvArr* mat, CvScalar value=cvRealScalar(1) );
```

`mat`

待初始化的矩阵 (不一定是方阵)。

`value`

赋值给对角线元素的值。

函数 cvSetIdentity 初始化带尺度的单位矩阵:
arr(i,j)=value 如果 i=j,
否则为 0

DotProduct

用欧几里得准则计算两个数组的点积

double cvDotProduct(const CvArr* src1, const CvArr* src2);

`src1`

第一输入数组。

`src2`

第二输入数组。

函数 cvDotProduct 计算并返回两个数组的欧几里得点积。

$src1 \bullet src2 = sum_I(src1(I)*src2(I))$

如果是多通道数组，所有通道的结果是累加在一起的。特别地，cvDotProduct(a,a),将返回 $||a||^2$ ，这里 a 是一个复向量。 该函数可以处理多通道数组,逐行或逐层等等。

CrossProduct

计算两个三维向量的叉积

void cvCrossProduct(const CvArr* src1, const CvArr* src2, CvArr* dst);

`src1`

第一输入向量。

`src2`

第二输入向量。

`dst`

输出向量

函数 cvCrossProduct 计算两个三维向量的差积：

$dst = src1 \times src2$, ($dst_1 = src1_2src2_3 - src1_3src2_2$, $dst_2 = src1_3src2_1 - src1_1src2_3$, $dst_3 = src1_1src2_2 - src1_2src2_1$).

ScaleAdd

计算一个数组缩放后与另一个数组的和

void cvScaleAdd(const CvArr* src1, CvScalar scale, const CvArr* src2, CvArr* dst);

#define cvMulAddS cvScaleAdd

`src1`

第一输入数组

`scale`

第一输入数组的缩放因子

`src2`

第二输入数组

`dst`

输出数组

函数 cvScaleAdd 计算一个数组缩放后与另一个数组的和:
dst(I)=src1(I)*scale + src2(I)
所有的数组参数必须有相同的类型和大小。

GEMM

通用矩阵乘法

```
void    cvGEMM( const CvArr* src1, const CvArr* src2, double alpha,
                const CvArr* src3, double beta, CvArr* dst, int
tABC=0 );
#define cvMatMulAdd( src1, src2, src3, dst ) cvGEMM( src1, src2, 1,
src3, 1, dst, 0 )
#define cvMatMul( src1, src2, dst ) cvMatMulAdd( src1, src2, 0, dst )
```

`src1`
第一输入数组
`src2`
第二输入数组
`src3`
第三输入数组 (偏移量)，如果没有偏移量，可以为空（ NULL） 。
`dst`
输出数组
`tABC`
T 操作标志，可以是 0 或者下面列举的值的组合:
CV_GEMM_A_T - 转置 src1
CV_GEMM_B_T - 转置 src2
CV_GEMM_C_T - 转置 src3
例如, CV_GEMM_A_T+CV_GEMM_C_T 对应
alpha*src1^T*src2 + beta*src3^T

函数 cvGEMM 执行通用矩阵乘法:
dst = alpha*op(src1)*op(src2) + beta*op(src3), 这里 op(X) 是 X 或者 X^T
所有的矩阵应该有相同的数据类型和协调的矩阵大小。支持实数浮点矩阵或者复数浮点矩阵。

Transform

对数组每一个元素执行矩阵变换

```
void cvTransform( const CvArr* src, CvArr* dst, const CvMat*
transmat, const CvMat* shiftvec=NULL );
```

`src`
输入数组
`dst`
输出数组
`transmat`
变换矩阵
`shiftvec`
可选偏移向量

函数 cvTransform 对数组 src 每一个元素执行矩阵变换并将结果存储到 dst:
dst(I)=transmat*src(I) + shiftvec 或者
$dst(I)_k=sum_j(transmat(k,j)*src(I)_j) + shiftvec(k)$
N-通道数组 src 的每一个元素都被视为一个N元向量，使用一个 M×N 的变换矩阵 transmat 和偏移向量 shiftvec 把它变换到一个 M-通道的数组 dst 的一个元素中。 这里可以选择将偏移向量 shiftvec 嵌入到 transmat 中。这样的话 transmat 应该是 M×N+1 的矩阵，并且最右边的一列被看作是偏移向量 。
输入数组和输出数组应该有相同的位深（depth）和同样的大小或者 ROI 大小。 transmat 和 shiftvec 应该是实数浮点矩阵。
该函数可以用来进行 ND 点集的几何变换，任意的线性颜色空间变换，通道转换等。

PerspectiveTransform

向量数组的透视变换

void cvPerspectiveTransform(const CvArr* src, CvArr* dst, const CvMat* mat);

`src`
输入的三通道浮点数组

`dst`
输出三通道浮点数组

`mat`
4 × 4 变换矩阵

函数 cvPerspectiveTransform 用下面的方式变换 src 的每一个元素(通过将其视为二维或者三维的向量):
(x, y, z) -> (x'/w, y'/w, z'/w) 或者
(x, y) -> (x'/w, y'/w),

这里
(x', y', z', w') = mat*(x, y, z, 1) 或者
(x', y', w') = mat*(x, y, 1)

并且 w = w' 如果 w'!=0,
否则 w = inf

MulTransposed

计算数组和数组的转置的乘积

void cvMulTransposed(const CvArr* src, CvArr* dst, int order, const CvArr* delta=NULL);

`src`
输入矩阵

`dst`
目标矩阵

`order`

乘法顺序

`delta`

一个可选数组， 在乘法之前从 src 中减去该数组。

函数 cvMulTransposed 计算 src 和它的转置的乘积。

函数求值公式：

如果 order=0

dst=(src-delta)*(src-delta)^T

否则

dst=(src-delta)^T*(src-delta)

Trace

返回矩阵的迹

CvScalar cvTrace(const CvArr* mat);

`mat`

输入矩阵

函数 cvTrace 返回矩阵mat的对角线元素的和。

tr(src1)=sum$_i$mat(i,i)

Transpose

矩阵的转置

void cvTranspose(const CvArr* src, CvArr* dst);

#define cvT cvTranspose

`src`

输入矩阵

`dst`

目标矩阵

函数 cvTranspose 对矩阵 src 求转置:

dst(i,j)=src(j,i)

注意，假设是复数矩阵不会求得复数的共轭。共轭应该是独立的：查看的 cvXorS 例子代码。

Det

返回矩阵的行列式值

double cvDet(const CvArr* mat);

`mat`

输入矩阵

函数 cvDet 返回方阵 mat 的行列式值。对小矩阵直接计算，对大矩阵用 高斯(GAUSSIAN)消去法。对于对称正定(positive-determined)矩阵也可以用 SVD 函数来求，U=V=NULL ，然后用 w 的对角线元素的乘积来计算行列式。

Invert

查找矩阵的逆矩阵或伪逆矩阵

double cvInvert(const CvArr* src, CvArr* dst, int method=CV_LU);

#define cvInv cvInvert

`src`

输入矩阵

`dst`

目标矩阵

`method`

求逆方法:

CV_LU -最佳主元选取的高斯消除法

CV_SVD - 奇异值分解法 (SVD)

CV_SVD_SYM - 正定对称矩阵的 SVD 方法

函数 cvInvert 对矩阵 src1 求逆并将结果存储到 src2

如果是 LU 方法该函数返回 src1 的行列式值 (src1 必须是方阵)。

如果是 0, 矩阵不求逆， src2 用 0 填充。

如果 SVD 方法该函数返回 src1 的条件数的倒数(最小奇异值和最大奇异值的比值) ，如果 src1 全为 0 则返回 0。 如果 src1 是奇异的，SVD 方法计算一个伪逆矩阵。

Solve

求解线性系统或者最小二乘法问题

int cvSolve(const CvArr* src1, const CvArr* src2, CvArr* dst, int method=CV_LU);

A

输入矩阵

B

线性系统的右部

X

输出解答

`method`

解决方法(矩阵求逆) :

CV_LU - 最佳主元选取的高斯消除法

CV_SVD - 奇异值分解法 (SVD)

CV_SVD_SYM - 对正定对称矩阵的 SVD 方法

函数 cvSolve 解决线性系统或者最小二乘法问题 (后者用 SVD 方法可以解决):

dst = arg $\min_X$||src1*X-src2||

如果使用 CV_LU 方法。 如果 src1 是非奇异的，该函数则返回 1 ，否则返回 0 ，在后一种情况下 dst 是无效的。

SVD

对实数浮点矩阵进行奇异值分解

void cvSVD(CvArr* A, CvArr* W, CvArr* U=NULL, CvArr* V=NULL, int flags=0);

A

M×N 的输入矩阵

`W`

结果奇异值矩阵 (M×N 或者 N×N) 或者 向量 (N×1).

U

可选的左部正交矩阵 (M×M or M×N). 如果 CV_SVD_U_T 被指定， 应该交换上面所说的行与列的数目。

V

可选右部正交矩阵(N×N)

flags

操作标志； 可以是 0 或者下面的值的组合:

- CV_SVD_MODIFY_A 通过操作可以修改矩阵 src1 。这样处理速度会比较快。
- CV_SVD_U_T 意味着会返回转置矩阵 U ，指定这个标志将加快处理速度。
- CV_SVD_V_T 意味着会返回转置矩阵 V ，指定这个标志将加快处理速度。

函数 cvSVD 将矩阵 A 分解成一个对角线矩阵和两个正交矩阵的乘积：

$A=U*W*V^T$

这里 w 是一个奇异值的对角线矩阵，它可以被编码成奇异值的一维向量，u 和 v 也是一样。 所有的奇异值都是非负的并按降序存储。（u 和 v 也相应的存储）。

SVD 算法在数值处理上已经很稳定，它的典型应用包括：

- 当 A 是一个方阵、对称阵和正矩阵时精确的求解特征值问题，例如, 当 A 时一个协方差矩阵时。在这种情况下 w 将是一个特征值的的向量，并且 U=V是矩阵的特征向量(因此，当需要计算特征向量时 u 和 v 只需要计算其中一个就可以了) 。
- 精确的求解病态线性系统
- 超定线性系统的最小二乘求解。上一个问题和这个问题都可以用指定 CV_SVD 的 cvSolve 方法。
- 精确计算矩阵的不同特征，如秩(非零奇异值的数目), 条件数(最大奇异值和最小奇异值的比例), 行列式值(行列式的绝对值等于奇异值的乘积).上述的所有这些值都不要求计算矩阵 u 和 v 。

SVBkSb

奇异值回代算法（back substitution）

```
void    cvSVBkSb( const CvArr* W, const CvArr* U, const CvArr* V,
                  const CvArr* B, CvArr* X, int flags );
```

W

奇异值矩阵或者向量

U

左正交矩阵 (可能是转置的)

V

右正交矩阵 (可能是转置的)

B

原始矩阵 A 的伪逆的乘法矩阵。这个是可选参数。如果它被省略则假定它是一个适当大小的单位矩阵(因此 x 将是 A 的伪逆的重建).。

X

目标矩阵: 奇异值回代算法的结果

flags

操作标志, 和刚刚讨论的 cvSVD 的标志一样。

函数 cvSVBkSb 为被分解的矩阵 A 和矩阵 B 计算回代逆（back substitution） (参见 cvSVD 说明) :

$X=V*W^{-1}*U^{T}*B$

这里

$W^{-1}(i,i)=1/W(i,i)$ 如果 $W(i,i) > epsilon \bullet sum_i W(i,i)$,

否则 0

epsilon 是一个依赖于矩阵数据类型的的很小的数。

该函数和 cvSVD 函数被用来执行 cvInvert 和 cvSolve, 用这些函数(svd & bksb)的原因是初级函数（low-level） 函数可以避免高级函数(inv & solve) 计算中内部分配的临时矩阵。

EigenVV

计算对称矩阵的特征值和特征向量

void cvEigenVV(CvArr* mat, CvArr* evects, CvArr* evals, double eps=0);

mat

输入对称方阵。在处理过程中将被改变。

evects

特征向量输出矩阵， 连续按行存储

evals

特征值输出矩阵，按降序存储(当然特征值和特征向量的排序是同步的)。

eps

对角化的精确度 (典型地， DBL_EPSILON=$\approx 10^{-15}$ 就足够了)。

函数 cvEigenVV 计算矩阵 A 的特征值和特征向量:

mat*evects(i,:)' = evals(i)*evects(i,:)' (在 MATLAB 的记法)

矩阵 A 的数据将会被这个函数修改。

目前这个函数比函数 cvSVD 要慢，精确度要低， 如果已知 A 是正定的，(例如, 它是一个协方差矩阵), 它通常被交给函数 cvSVD 来计算其特征值和特征向量，尤其是在不需要计算特征向量的情况下

CalcCovarMatrix

计算向量集合的协方差矩阵

void cvCalcCovarMatrix(const CvArr** vects, int count, CvArr* cov_mat, CvArr* avg, int flags);

vects

输入向量。他们必须有同样的数据类型和大小。这个向量不一定非是一维的，他们也可以是二维（例如，图像）等等。

count

输入向量的数目

`cov_mat`
输出协方差矩阵，它是浮点型的方阵。
`avg`
输入或者输出数组 (依赖于标记“flags”) - 输入向量的平均向量。
`flags`
操作标志,下面值的组合:
CV_COVAR_SCRAMBLED - 输出协方差矩阵按下面计算:
scale*[vects[0]-avg,vects[1]-avg,...]^T*[vects[0]-avg,vects[1]-avg,...],
即协方差矩阵是 count×count. 这样一个不寻常的矩阵用于一组大型向量的快速 PCA 方法(例如，人脸识别的 EigenFaces 技术)。这个混杂（"scrambled"）矩阵的特征值将和真正的协方差矩阵的特征值匹配，真正的特征向量可以很容易的从混杂（"scrambled"）协方差矩阵的特征向量中计算出来。
CV_COVAR_NORMAL - 输出协方差矩阵被计算成:
scale*[vects[0]-avg,vects[1]-avg,...]*[vects[0]-avg,vects[1]-avg,...]^T,
也就是说, cov_mat 将是一个和每一个输入向量的元素数目具有同样线性大小的通常协方差矩阵。 CV_COVAR_SCRAMBLED 和 CV_COVAR_NORMAL 只能同时指定其中一个。
CV_COVAR_USE_AVG - 如果这个标志被指定， 该函数将不会从输入向量中计算 avg ，而是用过去的 avg 向量，如果 avg 已经以某种方式计算出来了这样做是很有用的。或者如果协方差矩阵是部分计算出来的 - 倘若这样, avg 不是输入向量的子集的平均值，而是整个集合的平均向量。
CV_COVAR_SCALE - 如果这个标志被指定，协方差矩阵被缩放了。 the covariation matrix is scaled.在 "normal" 模式下缩放比例是 1./count, 在 "scrambled" 模式下缩放比例是每一个输入向量的元素总和的倒数。缺省地 (如果没有指定标志) 协方差矩阵不被缩放 (scale=1)。

函数 cvCalcCovarMatrix 计算输入向量的协方差矩阵和平均向量。该函数 可以被运用到主成分分析中（PCA），
以及马氏距离（Mahalanobis distance）比较向量中等等。

Mahalonobis

计算两个向量之间的马氏距离（Mahalonobis distance）

double cvMahalanobis(const CvArr* vec1, const CvArr* vec2, CvArr* mat);

`vec1`
第一个一维输入向量
`vec2`
第二个一维输入向量
`mat`
The inverse covariation matrix. 协方差矩阵的逆矩阵

函数 cvMahalonobis 计算两个向量之间的加权距离，其返回结果是：
d(vec1,vec2)=sqrt($\sum_{i,j}$ {mat(i,j)*(vec1(i)-vec2(i))*(vec1(j)-vec2(j))})
协方差矩阵可以用函数cvCalcCovarMatrix 计算出来，逆矩阵可以用函数 cvInvert 计算出来 (CV_SVD 方法是一个比较好的选择, 因为矩阵可能是奇异的).

数学函数

Round, Floor, Ceil

转换浮点数为整数

int cvRound(double value);
int cvFloor(double value);
int cvCeil(double value);
value
输入浮点值

函数 cvRound, cvFloor, cvCeil 用一种舍入方法将输入浮点数转换成整数。 cvRound 返回和参数最接近的整数值。 cvFloor 返回不大于参数的最大整数值。cvCeil 返回不小于参数的最小整数值。在某些体系结构中该函数 工作起来比标准 C 操作起来还要快。如果参数的绝对值大于 2^{31}，结果是不可预料的。特别值 (±Inf, NaN) 是不可控制的。

Sqrt

计算平方根

float cvSqrt(float value);
value
输入浮点值

函数 cvSqrt 计算输入值的平方根。如果输入的是复数， 结果将不可预料。

InvSqrt

计算平方根的倒数

float cvInvSqrt(float value);
value
输入浮点值

函数 cvInvSqrt 计算输入值的平方根的倒数，大多数情况下它比 1./sqrt(value) 要快。 如果输入的是 0 或者复数，结果将不可预料。特别值 (±Inf, NaN) 是不可控制的。

Cbrt

计算立方根

float cvCbrt(float value);
value
输入浮点值

函数 cvCbrt 计算输入值的立方根，大多数情况下它比 pow(value,1./3) 要快。 另外，负数也是可操作的。特别值 (±Inf, NaN) 是不可控制的。

FastArctan

计算二维向量的角度

float cvFastArctan(float y, float x);

x

二维向量的 x 坐标

y

二维向量的 y 坐标

函数 cvFastArctan 计算二维向量的全范围角度角度，变化范围是 0° 到 360°。 精确度为 ~0.1° 。

IsNaN

判断输入是否是一个数字

int cvIsNaN(double value);

value

输入浮点值

函数 cvIsNaN 发给输入是一个数字则返回 1 (IEEE754 标准)，否则返回 0 。

IsInf

判断输入是否是无穷大

int cvIsInf(double value);

value

输入浮点值

函数 cvIsInf 如果输入是 ±Infinity (IEEE754 标准)则返回 1 ,否则返回 0 .

CartToPolar

计算二维向量的长度和/或者角度

void cvCartToPolar(const CvArr* x, const CvArr* y, CvArr* magnitude,
CvArr* angle=NULL, int angle_in_degrees=0);

x

x 坐标数组

y

y 坐标数组

magnitude

存储向量长度输出数组， 如果不是必要的它可以为空（NULL）

angle

存储角度输出数组, 如果不是必要的它可以为空（NULL）。它可以被标准化为弧度 (0..2π) 或者度数(0..360°)

angle_in_degrees

指示角度是用弧度或者度数表示的标志，缺省模式为弧度

函数 cvCartToPolar 计算二维向量(x(I),y(I))的长度，角度，或者两者同时计算:

magnitude(I)=sqrt($x(I)^2+y(I)^2$),
angle(I)=atan(y(I)/x(I))
角度的精确度 ≈0.1°. (0,0) 点的角度被设置为 0.

PolarToCart

计算极坐标形式的二维向量对应的直角坐标

void cvPolarToCart(const CvArr* magnitude, const CvArr* angle,
CvArr* x, CvArr* y, int angle_in_degrees=0);

`magnitude`
长度数组.如果为空（NULL），长度被假定为全是 1's.

`angle`
角度数组,弧度或者角度表示.

`x`
输出 x 坐标数组, 如果不需要，可以为空（NULL）.

`y`
输出 y 坐标数组, 如果不需要，可以为空（NULL）.

`angle_in_degrees`
指示角度是用弧度或者度数表示的标志，缺省模式为弧度

函数 cvPolarToCart 计算每个向量magnitude(I)*exp(angle(I)*j), j=sqrt(-1)的 x 坐标，y 坐标或者两者都计算:
x(I)=magnitude(I)*cos(angle(I)),
y(I)=magnitude(I)*sin(angle(I))

Pow

对数组内每个元素求幂

void cvPow(const CvArr* src, CvArr* dst, double power);

`src`
输入数组

`dst`
输出数组, 应该和输入数组有相同的类型

`power`
幂指数

函数 cvPow 计算输入数组的每个元素的 p 次幂:
$dst(I)=src(I)^p$, 如果p是整数
否则$dst(I)=abs(src(I))^p$
也就是说，对于非整型的幂指数使用输入数组元素的绝对值进行计算。然而，使用一些额外的操作，负值也可以得到正确的结果，象下面的例子，计算数组元素的立方根：

```
CvSize size = cvGetSize(src);
CvMat* mask = cvCreateMat( size.height, size.width, CV_8UC1 );
cvCmpS( src, 0, mask, CV_CMP_LT ); /* 查找负数 */
cvPow( src, dst, 1./3 );
cvSubRS( dst, cvScalarAll(0), dst, mask ); /* 输入的负值的结果求反 */
cvReleaseMat( &mask );
```

对于一些幂值，例如整数值, 0.5 和 -0.5，优化算法被使用。

Exp

计算数组元素的指数幂

void cvExp(const CvArr* src, CvArr* dst);

`src`

输入数组

`dst`

输出数组，它应该是 double 型的或者和输入数组有相同的类型

函数 cvExp 计算输入数组的每个元素的 e 次幂:

dst(I)=exp(src(I))

最大相对误差为 ≈7e-6. 通常，该函数转换无法输出的值为 0 输出。

Log

***Calculates natural logarithm of every array element absolute value** 计算每个数组元素的绝对值的自然对数*

void cvLog(const CvArr* src, CvArr* dst);

`src`

输入数组

`dst`

输出数组，它应该是 double 型的或者和输入数组有相同的类型

函数 cvLog 计算输入数组每个元素的绝对值的自然对数:

dst(I)=log(abs(src(I))), src(I)!=0

dst(I)=C, src(I)=0

这里 C 是一个大负数 (≈-700 通常的运算中)

随机数生成

RNG

初始化随机数生成器状态

CvRNG cvRNG(int64 seed=-1);

`seed`

64-bit 的值用来初始化一个随机序列

函数 cvRNG 初始化随机数生成器并返回其状态。指向这个状态的指针可以传递给函数 cvRandInt, cvRandReal 和 cvRandArr . 在通常的实现中使用一个 multiply-with-carry generator 。

RandArr

*用随机数填充数组并更新 **RNG** 状态*

void cvRandArr(CvRNG* rng, CvArr* arr, int dist_type, CvScalar param1, CvScalar param2);

`rng`

被 cvRNG 初始化的 RNG 状态.

`arr`

输出数组

`dist_type`

分布类型:

CV_RAND_UNI - 均匀分布

CV_RAND_NORMAL - 正常分布 或者 高斯分布

`param1`

分布的第一个参数。如果是均匀分布它是随机数范围的闭下边界。如果是正态分布它是随机数的平均值。

`param2`

分布的第二个参数。如果是均匀分布它是随机数范围的开上边界。如果是正态分布它是随机数的标准差。

函数 cvRandArr 用均匀分布的或者正态分布的随机数填充输出数组。在下面的例子中该函数被用来添加一些正常分布的浮点数到二维数组的随机位置。

```
/* let's noisy_screen be the floating-point 2d array that is to be
"crapped" */
CvRNG rng_state = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* allocate the array of coordinates of points */
CvMat* locations = cvCreateMat( pointCount, 1, CV_32SC2 );
/* arr of random point values */
CvMat* values = cvCreateMat( pointCount, 1, CV_32FC1 );
CvSize size = cvGetSize( noisy_screen );

cvRandInit( &rng_state,
                0, 1, /* 现在使用虚参数以后再调整 */
                0xffffffff /*这里使用一个确定的种子   */,
                CV_RAND_UNI /* 指定为均匀分布类型 */ );

/* 初始化 locations */
cvRandArr( &rng_state, locations, CV_RAND_UNI, cvScalar(0,0,0,0),
cvScalar(size.width,size.height,0,0) );

/* modify RNG to make it produce normally distributed values */
rng_state.disttype = CV_RAND_NORMAL;
cvRandSetRange( &rng_state,
                    30 /* deviation */,
                    100 /* average point brightness */,
                    -1 /* initialize all the dimensions */ );
/* generate values */
cvRandArr( &rng_state, values, CV_RAND_NORMAL,
              cvRealScalar(100), // average intensity
              cvRealScalar(30) // deviation of the intensity
              );
```

```
/* set the points */
for( i = 0; i < pointCount; i++ )
{
    CvPoint pt = *(CvPoint*)cvPtr1D( locations, i, 0 );
    float value = *(float*)cvPtr1D( values, i, 0 );
    *((float*)cvPtr2D( noisy_screen, pt.y, pt.x, 0 )) += value;
}

/* not to forget to release the temporary arrays */
cvReleaseMat( &locations );
cvReleaseMat( &values );

/* RNG state does not need to be deallocated */
```

RandInt

返回 32-bit 无符号整型并更新 RNG

```
unsigned cvRandInt( CvRNG* rng );
```

`rng`

被 cvRNG 初始化的 RNG 状态，被 RandSetRange (虽然, 后面这个函数对我们正讨论的函数的结果没有什么影响)随意地设置。

函数 cvRandInt 返回均匀分布的随机 32-bit 无符号整型值并更新 RNG 状态。它和 C 运行库里面的 rand() 函数十分相似，但是它产生的总是一个 32-bit 数而 rand() 返回一个 0 到 RAND_MAX （它是 2**16 或者 2**32, 依赖于操作平台）之间的数。

该函数用来产生一个标量随机数，例如点， patch sizes, table indices 等,用模操作可以产生一个确定边界的整数，人和其他特定的边界缩放到 0.. 1 可以产生一个浮点数。下面是用 cvRandInt 重写的前一个函数讨论的例子:

```
/* the input and the task is the same as in the previous sample. */
CvRNG rng_state = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* ... - no arrays are allocated here */
CvSize size = cvGetSize( noisy_screen );
/* make a buffer for normally distributed numbers to reduce call
overhead */
#define bufferSize 16
float normalValueBuffer[bufferSize];
CvMat normalValueMat = cvMat( bufferSize, 1, CV_32F,
normalValueBuffer );
int valuesLeft = 0;

for( i = 0; i < pointCount; i++ )
{
    CvPoint pt;
    /* generate random point */
    pt.x = cvRandInt( &rng_state ) % size.width;
```

```
        pt.y = cvRandInt( &rng_state ) % size.height;

        if( valuesLeft <= 0 )
        {
            /* fulfill the buffer with normally distributed numbers if the
    buffer is empty */
            cvRandArr( &rng_state, &normalValueMat,
    CV_RAND_NORMAL, cvRealScalar(100), cvRealScalar(30) );
            valuesLeft = bufferSize;
        }
        *((float*)cvPtr2D( noisy_screen, pt.y, pt.x, 0 ) =
    normalValueBuffer[--valuesLeft];
    }

    /* there is no need to deallocate normalValueMat because we have
    both the matrix header and the data on stack. It is a common and
    efficient
    practice of working with small, fixed-size matrices */
```

RandReal

*返回浮点型随机数并更新 **RNG***

```
double cvRandReal( CvRNG* rng );
```

`rng`

被 cvRNG 初始化的 RNG 状态

函数 cvRandReal 返回均匀分布的随机浮点数，范围在 0..1 之间(不包括 1)。

离散变换

DFT

执行一维或者二维浮点数组的离散傅立叶正变换或者离散傅立叶逆变换

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_SCALE    2
#define CV_DXT_ROWS     4
#define CV_DXT_INV_SCALE
(CV_DXT_SCALE|CV_DXT_INVERSE)
#define CV_DXT_INVERSE_SCALE CV_DXT_INV_SCALE

void cvDFT( const CvArr* src, CvArr* dst, int flags );
```

`src`

输入数组, 实数或者复数.

`dst`

输出数组，和输入数组有相同的类型和大小。

`flags`

变换标志, 下面的值的组合:

CV_DXT_FORWARD - 正向 1D 或者 2D 变换. 结果不被缩放.

CV_DXT_INVERSE - 逆向 1D 或者 2D 变换. 结果不被缩放.当然 CV_DXT_FORWARD 和 CV_DXT_INVERSE 是互斥的.

CV_DXT_SCALE - 对结果进行缩放: 用数组元素除以它. 通常, 它和 CV_DXT_INVERSE 组合在一起,可以使用缩写 CV_DXT_INV_SCALE.

CV_DXT_ROWS - 输入矩阵的每个独立的行进行整型或者逆向变换。这个标志允许用户同时变换多个向量，减少开销(它往往比处理它自己要快好几倍), 进行 3D 和高维的变换等等。

函数 cvDFT 执行一维或者二维浮点数组的离散傅立叶正变换或者离散傅立叶逆变换:

N 元一维向量的正向傅立叶变换:

$y = F^{(N)} \bullet x$, 这里 $F^{(N)}_{jk}=\exp(-i \bullet 2Pi \bullet j \bullet k/N)$, $i=\sqrt{-1}$

N 元一维向量的逆向傅立叶变换:

$x' = (F^{(N)})^{-1} \bullet y = \mathrm{conj}(F^{(N)}) \bullet y$

$x = (1/N) \bullet x$

M×N 元二维向量的正向傅立叶变换:

$Y = F^{(M)} \bullet X \bullet F^{(N)}$

M×N 元二维向量的逆向傅立叶变换:

$X' = \mathrm{conj}(F^{(M)}) \bullet Y \bullet \mathrm{conj}(F^{(N)})$

$X = (1/(M \bullet N)) \bullet X'$

假设时实数数据 (单通道) ,从 IPL 借鉴过来的压缩格式被用来表现一个正向傅立叶变换的结果或者逆向傅立叶变换的输入：

$\mathrm{Re}\ Y_{0,0}$ $\mathrm{Re}\ Y_{0,1}$ $\mathrm{Im}\ Y_{0,1}$ $\mathrm{Re}\ Y_{0,2}$ $\mathrm{Im}\ Y_{0,2}$ ... $\mathrm{Re}\ Y_{0,N/2-1}$ $\mathrm{Im}\ Y_{0,N/2-1}$ $\mathrm{Re}\ Y_{0,N/2}$

$\mathrm{Re}\ Y_{1,0}$ $\mathrm{Re}\ Y_{1,1}$ $\mathrm{Im}\ Y_{1,1}$ $\mathrm{Re}\ Y_{1,2}$ $\mathrm{Im}\ Y_{1,2}$ ... $\mathrm{Re}\ Y_{1,N/2-1}$ $\mathrm{Im}\ Y_{1,N/2-1}$ $\mathrm{Re}\ Y_{1,N/2}$

$\mathrm{Im}\ Y_{1,0}$ $\mathrm{Re}\ Y_{2,1}$ $\mathrm{Im}\ Y_{2,1}$ $\mathrm{Re}\ Y_{2,2}$ $\mathrm{Im}\ Y_{2,2}$ ... $\mathrm{Re}\ Y_{2,N/2-1}$ $\mathrm{Im}\ Y_{2,N/2-1}$ $\mathrm{Im}\ Y_{2,N/2}$

.....

$\mathrm{Re}\ Y_{M/2-1,0}$ $\mathrm{Re}\ Y_{M-3,1}$ $\mathrm{Im}\ Y_{M-3,1}$ $\mathrm{Re}\ Y_{M-3,2}$ $\mathrm{Im}\ Y_{M-3,2}$ ... $\mathrm{Re}\ Y_{M-3,N/2-1}$ $\mathrm{Im}\ Y_{M-3,N/2-1}$ $\mathrm{Re}\ Y_{M-3,N/2}$

$\mathrm{Im}\ Y_{M/2-1,0}$ $\mathrm{Re}\ Y_{M-2,1}$ $\mathrm{Im}\ Y_{M-2,1}$ $\mathrm{Re}\ Y_{M-2,2}$ $\mathrm{Im}\ Y_{M-2,2}$ ... $\mathrm{Re}\ Y_{M-2,N/2-1}$ $\mathrm{Im}\ Y_{M-2,N/2-1}$ $\mathrm{Im}\ Y_{M-2,N/2}$

$\mathrm{Re}\ Y_{M/2,0}$ $\mathrm{Re}\ Y_{M-1,1}$ $\mathrm{Im}\ Y_{M-1,1}$ $\mathrm{Re}\ Y_{M-1,2}$ $\mathrm{Im}\ Y_{M-1,2}$ ... $\mathrm{Re}\ Y_{M-1,N/2-1}$ $\mathrm{Im}\ Y_{M-1,N/2-1}$ $\mathrm{Im}\ Y_{M-1,N/2}$

注意:如果 N 时偶数最后一列存在（is present），如果 M 时偶数最后一行（is present）.

如果是一维实数的变换结果就像上面矩阵的第一行的形式。

MulSpectrums

*两个傅立叶频谱的每个元素的乘法（**Performs per-element multiplication of two Fourier spectrums**）*

void cvMulSpectrums(const CvArr* src1, const CvArr* src2, CvArr* dst, int flags);

`src1`

第一输入数组

`src2`

第二输入数组

`dst`

输出数组，和输入数组有相同的类型和大小。

`flags`

下面列举的值的组合:

CV_DXT_ROWS - 把数组的每一行视为一个单独的频谱 (参见 cvDFT 的参数讨论).

CV_DXT_MUL_CONJ - 在做乘法之前取第二个输入数组的共轭.

函数 cvMulSpectrums 执行两个 CCS-packed 或者实数或复数傅立叶变换的结果复数矩阵的每个元素的乘法。（performs per-element multiplication of the two CCS-packed or complex matrices that are results of real or complex Fourier transform.）

该函数和 cvDFT 可以用来快速计算两个数组的卷积.

DCT

执行一维或者二维浮点数组的离散余弦变换或者离散反余弦变换

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_ROWS     4
```

void cvDCT(const CvArr* src, CvArr* dst, int flags);

`src`

输入数组, 1D 或者 2D 实数数组.

`dst`

输出数组，和输入数组有相同的类型和大小。

`flags`

变换标志符, 下面值的组合:

CV_DXT_FORWARD - 1D 或者 2D 余弦变换.

CV_DXT_INVERSE - 1D or 2D 反余弦变换.

CV_DXT_ROWS - 对输入矩阵的每个独立的行进行余弦或者反余弦变换.

这个标志允许用户同时进行多个向量的变换，可以用来减少开销（它往往比处理它自己要快好几倍），以及 3D 和高维变换等等。

函数 cvDCT 执行一维或者二维浮点数组的离散余弦变换或者离散反余弦变换：

N 元一维向量的余弦变换:

$y = C^{(N)} \cdot x$, 这里 $C^{(N)}_{jk} = \mathrm{sqrt}((j==0?1:2)/N) \cdot \cos(Pi \cdot (2k+1) \cdot j/N)$

N 元一维向量的反余弦变换:

$x = (C^{(N)})^{-1} \cdot y = (C^{(N)})^{T} \cdot y$

M×N 元二维向量的余弦变换:
$Y = (C^{(M)})\cdot X\cdot (C^{(N)})^T$

M×N 元二维向量的反余弦变换:
$X = (C^{(M)})^T\cdot Y\cdot C^{(N)}$

动态结构

内存存储（memory storage)

CvMemStorage

Growing memory storage

```
typedef struct CvMemStorage
{
    struct CvMemBlock* bottom;/* first allocated block */
    struct CvMemBlock* top; /* the current memory block - top of the
stack */
    struct CvMemStorage* parent; /* borrows new blocks from */
    int block_size; /* block size */
    int free_space; /* free space in the top block (in bytes) */
} CvMemStorage;
```

内存存储器是一个可用来存储诸如序列，轮廓，图形,子划分等动态增长数据结构的底层结构。它是由一系列以同等大小的内存块构成，呈列表型 ---bottom 域指的是列首，top 域指的是当前指向的块但未必是列尾.在 bottom 和 top 之间所有的块(包括 bottom, 不包括 top）被完全占据了空间；在 top 和列尾之间所有的块（包括块尾，不包括 top)则是空的；而 top 块本身则被占据了部分空间 -- free_space 指的是 top 块剩余的空字节数。

新分配的内存缓冲区（或显示的通过 cvMemStorageAlloc 函数分配，或隐示的通过 cvSeqPush, cvGraphAddEdge 等高级函数分配）总是起始于当前块（即 top 块)的剩余那部分，如果剩余那部分能满足要求（够分配的大小）。分配后，free_space 就减少了新分配的那部分内存大小，外加一些用来保存适当列型的附加大小。当 top 块的剩余空间无法满足被分配的块（缓冲区）大小时，top 块的下一个存储块被置为当前块（新的 top 块） -- free_space 被置为先前分配的整个块的大小。

如果已经不存在空的存储块（即：top 块已是列尾），则必须再分配一个新的块（或从 parent 那继承,见 cvCreateChildMemStorage)并将该块加到列尾上去。于是，存储器（memory storage)就如同栈（Stack)那样, bottom 指向栈底，(top, free_space)对指向栈顶。栈顶可通过 cvSaveMemStoragePos 保存，通过 cvRestoreMemStoragePos 恢复指向， 通过 cvClearStorage 重置。

CvMemBlock

内存存储块结构

```
typedef struct CvMemBlock
{
    struct CvMemBlock* prev;
    struct CvMemBlock* next;
} CvMemBlock;
```

CvMemBlock 代表一个单独的内存存储块结构。内存存储块中的实际数据存储在 header 块 之后(即：存在一个头指针 head 指向的块 header ，该块不存储数据）,于是，内存块的第 i 个字节可以通过表达式 ((char*)(mem_block_ptr+1))[i] 获得。然而，通常没必要直接去获得存储结构的域。

CvMemStoragePos

内存存储块地址

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
    int free_space;
} CvMemStoragePos;
```

该结构（如以下所说）保存栈顶的地址，栈顶可以通过 cvSaveMemStoragePos 保存，也可以通过 cvRestoreMemStoragePos 恢复。

CreateMemStorage

创建内存块

```
CvMemStorage* cvCreateMemStorage( int block_size=0 );
```

block_size：存储块的大小以字节表示。如果大小是 0 byte, 则将该块设置成默认值 -- 当前默认大小为 64k.

函数 cvCreateMemStorage 创建一内存块并返回指向块首的指针。起初，存储块是空的。头部（即：header)的所有域值都为 0，除了 block_size 外.

CreateChildMemStorage

创建子内存块

```
CvMemStorage* cvCreateChildMemStorage( CvMemStorage*
parent );
```

parent 父内存块

函数 cvCreateChildMemStorage 创建一类似于普通内存块的子内存块，除了内存分配/释放机制不同外。当一个子存储块需要一个新的块加入时，它就试图从 parent 那得到这样一个块。如果 parent 中 还未被占据空间的那些块中的第一个块是可获得的，就获取第一个块（依

此类推），再将该块从 parent 那里去除。如果不存在这样的块，则 parent 要么分配一个，要么从它自己 parent (即：parent 的 parent) 那借个过来。换句话说，完全有可能形成一个链或更为复杂的结构，其中的内存存储块互为 child/ parent 关系（父子关系）。当子存储结构被释放或清除，它就把所有的块还给各自的 parent. 在其他方面，子存储结构同普通存储结构一样。
子存储结构在下列情况中是非常有用的。想象一下，如果用户需要处理存储在某个块中的动态数据，再将处理的结果存放在该块中。在使用了最简单的方法处理后，临时数据作为输入和输出数据被存放在了同一个存储块中，于是该存储块看上去就类似下面处理后的样子：
Dynamic data processing without using child storage.
结果，在存储块中，出现了垃圾（临时数据）。然而，如果在开始处理数据前就先建立一个子存储块，将临时数据写入子存储块中并在最后释放子存储块，那么最终在 源/目的存储块 （source / destination storage) 中就不会出现垃圾，于是该存储块看上去应该是如下形式：
Dynamic data processing using a child storage.

ReleaseMemStorage

释放内存块

void cvReleaseMemStorage(CvMemStorage** storage);
storage: 指向被释放了的存储块的指针
函数 cvReleaseMemStorage 释放所有的存储（内存）块 或者 将它们返回给各自的 parent（如果需要的话）。 接下来再释放 header 块（即：释放头指针 head 指向的块 = free(head)）并清除指向该块的指针（即：head = NULL)。在释放作为 parent 的块之前，先清除各自的 child 块。

ClearMemStorage

清空内存存储块

void cvClearMemStorage(CvMemStorage* storage);
storage:存储存储块
函数 cvClearMemStorage 将存储块的 top 置到存储块的头部（注：清空存储块中的存储内容）。该函数并不释放内存（仅清空内存）。假使该内存块有一个父内存块（即：存在一内存块与其有父子关系），则函数就将所有的块返回给其 parent.

MemStorageAlloc

在存储块中分配以内存缓冲区

void* cvMemStorageAlloc(CvMemStorage* storage, size_t size);
storage：内存块.
size：缓冲区的大小.
函数 cvMemStorageAlloc 在存储块中分配一内存缓冲区。该缓冲区的大小不能超过内存块的大小，否则就会导致运行时错误。缓冲区的地址被调整为 CV_STRUCT_ALIGN 字节 （当前为 sizeof(double)).

MemStorageAllocString

在存储块中分配一文本字符串

```
typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;
```

CvString cvMemStorageAllocString(CvMemStorage* storage, const char* ptr, int len=-1);

storage：存储块

ptr：字符串

len：字符串的长度（不计算'\0'）。如果参数为负数，函数就计算该字符串的长度。

函数 cvMemStorageAlloString 在存储块中创建了一字符串的拷贝。它返回一结构，该结构包含字符串的长度（该长度或通过用户传递，或通过计算得到）和指向被拷贝了的字符串的指针。

SaveMemStoragePos

保存内存块的位置（地址）

void cvSaveMemStoragePos(const CvMemStorage* storage, CvMemStoragePos* pos);

storage：内存块.

pos：内存块顶部位置。

函数 cvSaveMemStoragePos 将存储块的当前位置保存到参数 pos 中。 函数 cvRestoreMemStoragePos 可进一步获取该位置（地址）。

RestoreMemStoragePos

恢复内存存储块的位置

void cvRestoreMemStoragePos(CvMemStorage* storage, CvMemStoragePos* pos);

storage：内存块.

pos：新的存储块的位置

函数 cvRestoreMemStoragePos 通过参数 pos 恢复内存块的位置。该函数和函数 cvClearMemStorage 是释放被占用内存块的唯一方法。注意：没有什么方法可去释放存储块中被占用的部分内存。

Sequences

CvSeq

Growable sequence of elements

```
#define CV_SEQUENCE_FIELDS() \
```

```
    int flags; /* micsellaneous flags */ \
    int header_size; /* size of sequence header */ \
    struct CvSeq* h_prev; /* previous sequence */ \
    struct CvSeq* h_next; /* next sequence */ \
    struct CvSeq* v_prev; /* 2nd previous sequence */ \
    struct CvSeq* v_next; /* 2nd next sequence */ \
    int total; /* total number of elements */ \
    int elem_size;/* size of sequence element in bytes */ \
    char* block_max;/* maximal bound of the last block */ \
    char* ptr; /* current write pointer */ \
    int delta_elems; /* how many elements allocated when the
sequence grows (sequence granularity) */ \
    CvMemStorage* storage; /* where the seq is stored */ \
    CvSeqBlock* free_blocks; /* free blocks list */ \
    CvSeqBlock* first; /* pointer to the first sequence block */


typedef struct CvSeq
{
    CV_SEQUENCE_FIELDS()
} CvSeq;
```

结构CvSeq是所有OpenCV动态数据结构的基础。

通过不同寻常的宏定义简化了带有附加参数的结构 CvSeq 的扩展。为了扩展 CvSeq, 用户可以定义一新的数据结构或在通过宏 CV_SEQUENCE_FIELDS()所包括的 CvSeq 的域后在放入用户自定义的域。

有两种类型的序列 -- 稠密序列和稀疏序列。稠密序列都派生自 CvSeq, 它们用来代表可扩展的一维数组 -- 向量，栈，队列，双端队列。数据间不存在空隙（即：连续存放）-- 如果元素从序列中间被删除或插入新的元素到序列中（不是两端），那么此元素后边的相关元素会被移动。稀疏序列都派生自 CvSet,后面会有详细的讨论。它们都是由节点所组成的序列,每一个节点要么被占用空间要么是空,由 flag 标志指定。这些序列作为无序的数据结构而被使用，如点集，图，哈希表等。

The field header_size contains the actual size of the sequence header and should be greater or equal to sizeof(CvSeq).

域 header_size 含有序列头部节点的实际大小，此大小大于或等于 sizeof(CvSeq).

域 h_prev, h_next, v_prev, v_next 可用来创建不同序列的层次结构。域 h_prev, h_next 指向同一层次结构前一个和后一个序列，而域 v_prev, v_next 指向在垂直方向上的前一个和后一个序列，即：父亲和子孙。

域 first 指向第一个序列快，块结构在后面描述。

域 total 包含稠密序列的总元素数和稀疏序列被分配的节点数。

域 flags 的高 16 位描述（包含）特定的动态结构类型（CV_SEQ_MAGIC_VAL 表示稠密序列,CV_SET_MAGIC_VAL 表示稀疏序列），同时包含形形色色的信息。

低 CV_SEQ_ELTYPE_BITS 位包含元素类型的 ID(标示符）。大多数处理函数并不会用到元素类型，而会用到存放在 elem_size 中的元素大小 。如果序列中包含 CvMat 中的数据，那么元素的类型就与 CvMat 中的类型相匹配， 如：CV_32SC2 可以被使用为由二维空间中的点序列， CV_32FC1 用为由浮点数组成的序列等。通过宏 CV_SEQ_ELTYPE(seq_header_ptr) 来获取序列中元素的类型。处理数字序列的函数判断： elem.size 等同于序列元素的大小。除了与 CvMat 相兼容的类型外，还有几个在头 cvtypes.h 中定义的额外的类型。

Standard Types of Sequence Elements

```
    #define CV_SEQ_ELTYPE_POINT          CV_32SC2    /*
(x,y) */
    #define CV_SEQ_ELTYPE_CODE           CV_8UC1     /*
freeman code: 0..7 */
    #define CV_SEQ_ELTYPE_GENERIC        0 /* unspecified
type of sequence elements */
    #define CV_SEQ_ELTYPE_PTR            CV_USRTYPE1
/* =6 */
    #define CV_SEQ_ELTYPE_PPOINT
CV_SEQ_ELTYPE_PTR   /* &elem: pointer to element of other
sequence */
    #define CV_SEQ_ELTYPE_INDEX          CV_32SC1    /*
#elem: index of element of some other sequence */
    #define CV_SEQ_ELTYPE_GRAPH_EDGE
CV_SEQ_ELTYPE_GENERIC   /* &next_o, &next_d, &vtx_o,
&vtx_d */
    #define CV_SEQ_ELTYPE_GRAPH_VERTEX
CV_SEQ_ELTYPE_GENERIC   /* first_edge, &(x,y) */
    #define CV_SEQ_ELTYPE_TRIAN_ATR
CV_SEQ_ELTYPE_GENERIC   /* vertex of the binary tree     */
    #define CV_SEQ_ELTYPE_CONNECTED_COMP
CV_SEQ_ELTYPE_GENERIC   /* connected component    */
    #define CV_SEQ_ELTYPE_POINT3D        CV_32FC3    /*
(x,y,z)    */
```

后面的 CV_SEQ_KIND_BITS 字节表示序列的类型：

Standard Kinds of Sequences

```
    /* generic (unspecified) kind of sequence */
    #define CV_SEQ_KIND_GENERIC       (0 <<
CV_SEQ_ELTYPE_BITS)

    /* dense sequence suntypes */
    #define CV_SEQ_KIND_CURVE         (1 <<
CV_SEQ_ELTYPE_BITS)
    #define CV_SEQ_KIND_BIN_TREE      (2 <<
CV_SEQ_ELTYPE_BITS)

    /* sparse sequence (or set) subtypes */
```

```
    #define CV_SEQ_KIND_GRAPH          (3 <<
CV_SEQ_ELTYPE_BITS)
    #define CV_SEQ_KIND_SUBDIV2D       (4 <<
CV_SEQ_ELTYPE_BITS)
```

CvSeqBlock

连续序列块

```
typedef struct CvSeqBlock
{
    struct CvSeqBlock* prev; /* previous sequence block */
    struct CvSeqBlock* next; /* next sequence block */
    int start_index; /* index of the first element in the block +
    sequence->first->start_index */
    int count; /* number of elements in the block */
    char* data; /* pointer to the first element of the block */
} CvSeqBlock;
```

序列块构成一个双向的循环列表，因此指针 prev 和 next 永远不会为 null，而总是指向序列中的前一个和后一个序列块。也就是说：最后一个序列块的 next 指向的就是序列中的第一个块，而第一个块的 prev 指向最后一个块。域 start_index 和 count 有助于跟踪序列中块的位置。 例如，一个含 10 个元素的序列被分成了 3 块，每一块的大小分别为 3， 5， 2，第一块的参数 start_index 为 2， 那么该序列的 (start_index, count) 相应为 （2，3），（5，5），（10，2）。第一个块的参数 start_index 通常为 0，除非一些元素已被插入到序列中。

CvSlice

序列分割

```
typedef struct CvSlice
{
    int start_index;
    int end_index;
} CvSlice;

inline CvSlice cvSlice( int start, int end );
#define CV_WHOLE_SEQ_END_INDEX 0x3fffffff
#define CV_WHOLE_SEQ  cvSlice(0,
CV_WHOLE_SEQ_END_INDEX)

/* calculates the sequence slice length */
int cvSliceLength( CvSlice slice, const CvSeq* seq );
```

有关序列的一些操作函数将 CvSlice 作为输入参数，默认情况下该参数通常被设置成整个序列（CV_WHOLE_SEQ)。start_index 和 end_index 任何一个都可以是负数或 超过序列长度，start_index 是闭界，end_index 是开界。如果两者相等，那么分割被认为是空分割

（即：不包含任何元素）。由于序列被看作是循环结构， 所以分割可以选择序列中靠后的几个元素，靠前的参数反而跟着它们，如 cvSlice（-2，3）。函数用下列发法来规范分割参数：首先， 调用 cvSliceLength 来决定分割的长度，然后， start_index 被使用类似于 cvGetSeqElem 的参数来规范（例如：负数也被允许）。实际的分割操作起始于规范化了的 start_index ，中止于 start_index + cvSliceLength()。（再次假设序列是循环结构）
如果函数并不接受分割参数，但你还是想要处理序列的一部分，那么可以使用函数 cvSeqSlice 获取子序列。

CreateSeq

创建一序列

CvSeq* cvCreateSeq(int seq_flags, int header_size,
int elem_size, CvMemStorage* storage);

seq_flags: 序列的符号标志。如果序列不会被传递给任何使用特定序列的函数，那么将它设为 0， 否则从预定义的序列类型中选择一合适的类型。
header_size: 序列头部的大小；必须大于或等于 sizeof(CvSeq). 如果制定了类型或它的扩展名，则此类型必须适合基类的头部大小。
elem_size: 元素的大小，以字节计。这个大小必须与序列类型相一致。例如，对于一个点的序列，元素类型 CV_SEQ_ELTYPE_POINT 应当被指定， 参数elem_size 必须等同于 sizeof(CvPoint).
函数 cvCreateSeq 创建一序列并且返回指向该序列的指针。函数在存储块中分配序列的头部作为一个连续躯体，并且设置结构的 flags域, elem_size域, header_size域 和 storage域 的值为被传递过来的值，设置 delta_elems 为默认值（可通过函数 cvSetSeqBlockSize 重新对其赋值），清空其他的头 部域，包括前sizeof(CvSeq) 个字节的空间。

SetSeqBlockSize

设置序列块的大小

void cvSetSeqBlockSize(CvSeq* seq, int delta_elems);
seq: 序列
delta_elems: 满足元素所需的块的大小
函数 cvSetSeqBlockSize 会对内存分配的粒度产生影响。 当序列缓冲区中空间消耗完时，函数为 delta_elems 个序列元素分配空间。如果新分配的空间与 之前分配的空间相邻的话，这两个块就合并，否则，就创建了一个新的序列快。因此，参数值越大，序列中出现碎片的可能性就越小，不过内存中更多的空间将被浪费。当序列被创建后，参数 delta_elems 大小将被设置为 默认大小（1K).之后， 就可随时调用该函数，并影响内存分配。 函数可以修改被传递过来的参数值，以满足内存块的大小限制 。

SeqPush

Adds element to sequence end

char* cvSeqPush(CvSeq* seq, void* element=NULL);

seq: 块

element: 添加的元素

函数 cvSeqPush 在序列块的尾部添加一元素并返回指向该元素得指针。如果输入参数为 null, 函数就仅仅分配一空间，留给下一个元素使用。下列代码说明如何使用该函数去创建一空间。

The following code demonstrates how to create a new sequence using this function:

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, /* sequence of integer elements */
                          sizeof(CvSeq), /* header size - no extra fields */
                          sizeof(int), /* element size */
                          storage /* the container storage */ );
int i;
for( i = 0; i < 100; i++ )
{
    int* added = (int*)cvSeqPush( seq, &i );
    printf( "%d is added\n", *added );
}

...
/* release memory storage in the end */
cvReleaseMemStorage( &storage );
```

函数 cvSeqPush 的时间复杂度为 O(1). 如果需要分配并使用的空间比较大，则存在一分配较快的函数（见：cvStartWriteSeq 和相关函数）

SeqPop

删除序列尾部元素

void cvSeqPop(CvSeq* seq, void* element=NULL);

seq: 序列

element: 可选参数。如果该指针不为空，就拷贝被珊元素到指针指向的位置

函数 cvSeqPop 从序列中删除一元素。如果序列已经为空，就报告一错误。函数时间复杂度为 O(1).

SeqPushFront

在序列头部添加元素

char* cvSeqPushFront(CvSeq* seq, void* element=NULL);

seq: 序列

element: 添加的元素

函数 cvSeqPushFront 类似于 cvSeqPush， 不过是在序列头部添加元素。时间复杂度为 O(1).

SeqPopFront

删除序列的头部元素

void cvSeqPopFront(CvSeq* seq, void* element=NULL);

seq: 序列

element: 可选参数。如果该指针不为空，就拷贝被珊元素到指针指向的位置。

函数 cvSeqPopFront 删除序列的头部元素。如果序列已经为空，就报告一错误。函数时间复杂度为 O(1).

SeqPushMulti

添加多个元素到序列尾部或头部。

void cvSeqPushMulti(CvSeq* seq, void* elements, int count, int in_front=0);

seq: 序列

elements: 待添加的元素

count: 添加的元素个数

in_front: 标示在头部还是尾部添加元素

CV_BACK (= 0) -- 在序列尾部添加元素。

CV_FRONT(!= 0) -- 在序列头部添加元素。

函数 cvSeqPushMulti 在序列头部或尾部添加多个元素。 元素按输入数组中的顺序被添加到序列中，不过它们可以添加到不同的序列中

SeqPopMulti

删除多个序列头部或尾部的元素

void cvSeqPopMulti(CvSeq* seq, void* elements, int count, int in_front=0);

seq: 序列

elements: 待删除的元素

count: 删除的元素个数

in_front: 标示在头部还是尾部删除元素

CV_BACK (= 0) -- 删除序列尾部元素。

CV_FRONT(!= 0) -- 删除序列头部元素。

函数 cvSeqPopMulti 删除多个序列头部或尾部的元素。 如果待删除的元素个数超过了序列中的元素总数，则函数删除尽可能多的元素 。

SeqInsert

在序列中添加元素

char* cvSeqInsert(CvSeq* seq, int before_index, void* element=NULL);

seq: 序列

before_index: 元素插入的位置（索引）。如果插入的位置在 0（允许的参数最小值）前，则该函数等同于函数 cvSeqPushFront.如果是在 seq_total（允许的参数最大值）后，则该函数等同于 cvSeqPush.
element: 待插入的元素
函数 cvSeqInsert 移动 从被插入的位置到序列尾部元素所在的位置的所有元素，如果 指针 element 不位 null, 则拷贝 element 中的元素到指定位置。函数返回指向被插入元素的指针。

SeqRemove

删除序列中的元素。

void cvSeqRemove(CvSeq* seq, int index);
seq: 序列
index: 被珊元素的索引。
函数 cvSeqRemove 删除指定的索引元素。如果索引出了序列的范围，就报告发现错误。企图从空序列中删除元素，函数报告错误。函数通过移动序列中的元素来删除索引元素。

ClearSeq

清空序列

void cvClearSeq(CvSeq* seq);
seq
Sequence.
seq: 序列
函数 cvClearSeq 删除序列中的所有元素。函数不会将内存返回到存储器中，当新的元素添加到序列中时，可重新使用该内存。函数时间复杂度为 O(1).

GetSeqElem

返回索引所指定的元素指针

char* cvGetSeqElem(const CvSeq* seq, int index);
#define CV_GET_SEQ_ELEM(TYPE, seq, index)
(TYPE*)cvGetSeqElem((CvSeq*)(seq), (index))
seq: 序列
index: 索引
函数 cvGetSeqElem 查找序列中索引所指定的元素，并返回指向该元素的指针。如果元素不存在，则返回 0。 函数支持负数，即： -1 代表 序列的最后一个元素， -2 代表最后第二个元素，等。如果序列只包含一个块，或者所需的元素在第一个块中，那么应当使用宏，
CV_GET_SEQ_ELEM(elemType, seq, index)宏中的参数
elemType 是序列中元素的类型（如：CvPoint), 参数 seq 表示序列，参数 index 代表所需元素的索引。该宏首先核查所需的元素是否属于第一个块，如果是，则返回该元素，否则，该宏就调用主函数
GetSeqElem. 如果索引为负数的话，则总是调用函数

cvGetSeqElem。函数的时间复杂度为 O(1)，假设块的大小要比元素的数量要小。

SeqElemIdx

返回序列中元素的索引

int cvSeqElemIdx(const CvSeq* seq, const void* element, CvSeqBlock** block=NULL);

seq：序列

element：指向序列中元素的指针

block：可选参数， 如果不为空(NULL),则存放包含该元素的块的地址

函数 cvSeqElemIdx 返回元素的索引，如果该元素不存在这个序列中，则返回一负数。

CvtSeqToArray

拷贝序列中的元素到一个连续的内存块中

void* cvCvtSeqToArray(const CvSeq* seq, void* elements, CvSlice slice=CV_WHOLE_SEQ);

seq：序列

elemenets：指向目的（存放拷贝元素的）数组的指针，指针指向的空间必须足够大。

slice：拷贝到序列中的序列部分。

函数 cvCvtSeqToArray 拷贝整个序列或部分序列到指定的缓冲区中，并返回指向该缓冲区的指针.

MakeSeqHeaderForArray

构建序列

CvSeq* cvMakeSeqHeaderForArray(int seq_type, int header_size, int elem_size,
void* elements, int total,
CvSeq* seq, CvSeqBlock*
block);

seq_type：序列的类型

header_size：序列的头部大小。大小必须大于等于数组的大小。

elem_size：元素的大小

elements：形成该序列的元素

total：序列中元素的总数。参数值必须等于数据的大小

seq：指向被使用作为序列头部的局部变量

block：指向局部变量的指针

函数 cvMakeSeqHeaderForArray 初始化序列的头部。序列块由用户分配（例如：在栈上）。该函数不拷贝数据。创建的序列只包含一个块，和一个 NULL指针，因此可以读取指针，但试图将元素添加到序列中则多数会引发错误。

SeqSlice

为各个序列碎片建立头

CvSeq* cvSeqSlice(const CvSeq* seq, CvSlice slice,
CvMemStorage* storage=NULL, int
copy_data=0);

seq: 序列

slice: 部分序列块

storage: 存放新的序列和拷贝数据（如果需要）的目的存储空间。如果为 NULL, 则函数使用包含该输入数据的存储空间。

copy_data: 标示是否要拷贝元素， 如果 copy_data != 0, 则需要拷贝；如果 copy_data == 0, 则不需拷贝。

函数 cvSeqSlice 创建一序列，该序列表示输入序列中特定的一部分（slice),。新序列要么与原序列共享元素要么拥有自己的一份拷贝。因此，如果 有人需要去 处理 该部分序列，但函数却没有 slice 参数，则使用该函数去获取该序列。.

CloneSeq

创建序列的一份拷贝

CvSeq* cvCloneSeq(const CvSeq* seq, CvMemStorage*
storage=NULL);

seq: 序列

storage：存放新序列的 header 部分和拷贝数据（如果需要）的目的存储块。如果为 NULL, 则函数使用包含输入序列的存储块 。

函数 cvCloneSeq 创建输入序列的一份完全拷贝。调用函数 cvCloneSeq (seq, storage) 等同于调用 cvSeqSlice(seq, CV_WHOLE_SEQ, storage, 1).

SeqRemoveSlice

*删除序列的 **slice** 部分*

void cvSeqRemoveSlice(CvSeq* seq, CvSlice slice);

seq: 序列

slice: 序列中被移动的那部分

函数 cvSeqRemoveSlice 删除序列中的 slice 部分

SeqInsertSlice

在序列中插入一数组

void cvSeqInsertSlice(CvSeq* seq, int before_index, const CvArr*
from_arr);

seq: 序列

slice: 序列中被移动的那部分

from_arr: 从中获取元素的数组

函数 cvSeqInsertSlice 在指定位置插入 来自数组 from_arr 中 所有元素。数组 from_arr 可以是一个 矩阵也可以是另外一个 序列。

SeqInvert

将序列中的元素进行逆序操作

```
void cvSeqInvert( CvSeq* seq );
```

seq: 序列

函数 cvSeqInvert 对序列进行逆序操作 -- 即：使第一个元素成为最后一个，最后一个元素为第一个。

SeqSort

使用特定的比较函数对序列中的元素进行排序

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b,
void* userdata);

void cvSeqSort( CvSeq* seq, CvCmpFunc func, void*
userdata=NULL );
```

seq: 待排序的序列

func: 比较函数，按照元素间的大小关系返回负数，零，正数（见：上面的声明和下面的例子） --相关函数为 C 运行时库中的 qsort, 后者（qsort)不使用参数 userdata.

userdata: 传递给比较函数的用户参数；有些情况下，可避免全局变量的使用

函数 cvSeqSort 使用特定的标准对序列进行排序。下面是一个 使用该函数的实例

```
/* Sort 2d points in top-to-bottom left-to-right order */
static int cmp_func( const void* _a, const void* _b, void* userdata )
{
    CvPoint* a = (CvPoint*)_a;
    CvPoint* b = (CvPoint*)_b;
    int y_diff = a->y - b->y;
    int x_diff = a->x - b->x;
    return y_diff ? y_diff : x_diff;
}

...

CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq),
sizeof(CvPoint), storage );
int i;

for( i = 0; i < 10; i++ )
{
    CvPoint pt;
    pt.x = rand() % 1000;
    pt.y = rand() % 1000;
    cvSeqPush( seq, &pt );
}
```

```
cvSeqSort( seq, cmp_func, 0 /* userdata is not used here */ );

/* print out the sorted sequence */
for( i = 0; i < seq->total; i++ )
{
    CvPoint* pt = (CvPoint*)cvSeqElem( seq, i );
    printf( "(%d,%d)\n", pt->x, pt->y );
}

cvReleaseMemStorage( &storage );
```

SeqSearch

查询序列中的元素

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b,
void* userdata);

char* cvSeqSearch( CvSeq* seq, const void* elem, CvCmpFunc
func,
                                int is_sorted, int* elem_idx, void*
userdata=NULL );
```

seq: 序列
elem: 待查询的元素
func: 比较函数，按照元素间的大小关系返回负数，零，正数（见：cvSeqSort）
is_sorted: 标示序列是否已经排序
elem_idx: 输出参数；（已查找到）元素的索引值
user_data: 传递到比较函数的用户参数；在某些情况下，有助于避免使用全局变量。
函数 cvSeqSearch 查找序列中的元素。如果序列已被排序，则使用二分查找（时间复杂度为 O(log(N))否则使用简单线性查找。若查找的元素不存在，函数返回 NULL 指针，而索引值设置为序列中的元素数（如果使用的是线性查找）或 满足表达式 seq(i) > elem 的最小的 i.

StartAppendToSeq

将数据写入序列中，并初始化该过程

```
void cvStartAppendToSeq( CvSeq* seq, CvSeqWriter* writer );
```

seq: 指向序列的指针
writer: writer 的状态； 由该函数初始化
函数 cvStartAppendToSeq 初始化将数据写入序列这个过程。通过宏 CV_WRITE_SEQ_ELEM(written_elem, writer)，写入的元素被添加到序列尾部。注意：在写入期间，序列的其他操作可能会产生的错误

的结果，甚至破怀该序列（见 cvFlushSeqWriter 的相关描述，有助于避免这些错误）

StartWriteSeq

创建新序列，并初始化写入部分（writer)

void cvStartWriteSeq(int seq_flags, int header_size, int elem_size,
CvMemStorage* storage, CvSeqWriter* writer);

seq_flags: 标示被创建的序列。如果序列还未传递给任何处理特定序列类型的函数，则序列值等于 0， 否则，必须从之前定义的序列类型中选择一个合适的类型。

header_size: 头部的大小。 参数值不小于 sizeof(CvSeq). 如果定义了某一类型，则该类型不许符合基类的条件。

elem_size: 元素的大小（以字节计）；必须与序列类型相一致。例如：如果创建了包含指针的序列(元素类型为 CV_SEQ_ELTYPE_POINT), 那么 elem_size 必须等同于 sizeof(CvPoint).

storage: 序列的（在内存）位置

writer: 写入部分 writer 的状态； 由该函数初始化

函数 cvStartWriteSeq 是 函数 cvCreateSeq 和函数 cvStartAppendToSeq 的组合。 指向被创建的序列的指针存放在 writer->seq 中， 通过函数 cvEndWriteSeq 返回（因当在最后调用）

EndWriteSeq

完成写入操作

CvSeq* cvEndWriteSeq(CvSeqWriter* writer);

- writer：写入部分 writer 的状态

函数 cvEndWriteSeq 完成写入操作并返回指向被写入元素的序列的地址。同时，函数会截取最后那个不完整的序列块，将块的剩余部分返回到内存中之后，序列就可以被安全的读和写。

FlushSeqWriter

根据写入状态，刷新序列头部

void cvFlushSeqWriter(CvSeqWriter* writer);

writer: 写入部分的状态

函数 cvFlushSeqWriter 用来使用户在写入过程中每当需要时读取序列元素，比如说，核查制定的条件。函数更新序列的头部，从而使读取序列中的数据成为可能。不过，写入并没有被关闭，为的是随时都可以将数据写入序列。在有些算法中，经常需要刷新，考虑使用 cvSeqPush 代替该函数。

StartReadSeq

初始化序列中的读取过程

void cvStartReadSeq(const CvSeq* seq, CvSeqReader* reader, int reverse=0);
seq: 序列
reader: 读取部分的状态； 由该函数初始化
reverse: 决定遍历序列的方向。如果 reverse 为 0，则读取顺序被定位从序列头部元素开始，否则从尾部开始读取
函数 cvStartReadSeq 初始化读取部分的状态。毕竟，顺序读取可通过调用宏 CV_READ_SEQ_ELEM(read_elem, reader)，逆序读取可通过调用宏CV_REV_READ_SEQ_ELEM(read_elem, reader)。这两个宏都将序列元素读进read_elem中， 并将指针移到下一个元素。下面代码显示了如何去使用reader 和 writer.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, sizeof(CvSeq), sizeof(int), storage );
CvSeqWriter writer;
CvSeqReader reader;
int i;

cvStartAppendToSeq( seq, &writer );
for( i = 0; i < 10; i++ )
{
    int val = rand()%100;
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("%d is written\n", val );
}
cvEndWriteSeq( &writer );

cvStartReadSeq( seq, &reader, 0 );
for( i = 0; i < seq->total; i++ )
{
    int val;
#if 1
    CV_READ_SEQ_ELEM( val, reader );
    printf("%d is read\n", val );
#else /* alternative way, that is prefferable if sequence elements are large,
            or their size/type is unknown at compile time */
    printf("%d is read\n", *(int*)reader.ptr );
    CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
#endif
}
...

cvReleaseStorage( &storage );
```

GetSeqReaderPos

返回当前的读取器的位置

int cvGetSeqReaderPos(CvSeqReader* reader);

`reader`

读取器的状态.

函数 cvGetSeqReaderPos 返回当前的 reader 位置 （在 0 到 reader->seq->total - 1 中）

SetSeqReaderPos

移动读取器到指定的位置。

void cvSetSeqReaderPos(CvSeqReader* reader, int index, int is_relative=0);

reader: reader 的状态

index: 目的位置。如果使用了 positioning mode, 则实际位置为 index % reader->seq->total.

is_relative: 如果不位 0, 那么索引（index) 就相对于当前的位置

函数 cvSetSeqReaderPos 将 read 的位置移动到绝对位置，或相对于当前的位置（相对位置）

Sets

CvSet

Collection of nodes

```
typedef struct CvSetElem
{
    int flags; /* it is negative if the node is free and zero or positive otherwise */
    struct CvSetElem* next_free; /* if the node is free, the field is a
                                    pointer to next free node */
}
CvSetElem;

#define CV_SET_FIELDS()    \
    CV_SEQUENCE_FIELDS()   /* inherits from CvSeq */ \
    struct CvSetElem* free_elems; /* list of free nodes */

typedef struct CvSet
{
    CV_SET_FIELDS()
} CvSet;
```

在 OpenCV 的稀疏数据结构中， CvSet 是一基本结构。

从上面的声明中可知：CvSet 继承自 CvSeq， 并在此基础上增加了个 free_elems 域，该域是空节点组成的列表。集合中的每一个节点，无论空否，都是线性表中的一个元素。尽管对于稠密的表中的元素没有限制，集合（派生的结构）元素必须起始于整数域，并与结构

CvSetElem 相吻合，因为这两个域对于（由空节点组成）集合的组织是必要的。如果节点为空，flags 为负，next_free 指向下一个空节点。如果节点已被占据空间，flags 为正， flags 包含节点索引值（使用表达式 set_elem->flags & CV_SET_ELEM_IDX_MASKH 获取), flags 的剩余内容由用户决定。宏 CV_IS_SET_ELEM(set_elem.ptr)用来识别特定的节点是否为空。
起初,集合 set 同表 list 都为空。当需要一个来自集合中的新节点时，就从表 list 中去获取，然后表进行了更新。如果表 list 碰巧为空，于是就分配一内存块，块中的所有节点与表 list 相连。结果，集合的 total 域被设置为空节点和非空节点的和。当非空节点别释放后，就将它加到空节点列表中。最先被释放的节点也就是最先被占用空间的节点
在 OpenCV 中， CvSet 用来代表图形（CvGraph), 稀疏多维数组（CvSparseMat), 平面子划分（planner subdivisions)等

CreateSet

创建空的数据集
CvSet* cvCreateSet(int set_flags, int header_size,
int elem_size, CvMemStorage* storage);
set_flags: 集合的类型
header_size: 头节点的大小；可能小于 sizeof(CvSet)
elem_size: 元素的大小； 可能小于 CvSetElem
storage: 相关容器
函数 CvCreateSet 创建一具有特定头部节点大小和元素类型的空集。并返回指向该集合的指针。

SetAdd

占用集合中的一个节点
int cvSetAdd(CvSet* set_header, CvSetElem* elem=NULL,
CvSetElem** inserted_elem=NULL);
set_header: 集合
elem: 可选的输入参数，被插入的元素。如果不为 NULL, 函数就将数据拷贝到新分配的节点。（拷贝后，清空第一个域的 MSB)
函数 cvSetAdd 分配一新的节点，将输入数据拷贝给它（可选），并且返回指向该节点的指针和节点的索引值。索引值可通过节点的 flags 域的低位中获得。函数的时间复杂度为 O(1), 不过，存在着一个函数可快速的分配内存。（见 cvSetNew)

SetRemove

从点集中删除元素
void cvSetRemove(CvSet* set_header, int index);
set_header： 集合
index：被删元素的索引值
函数 cvSetRemove 从点集中删除一具有特定索引值的元素。如果指定位置的节点为空，函数将什么都不做。函数的时间复杂度为 O(1), 不

过,存在一函数可更快速的完成该操作，该函数就是
cvSetRemoveByPtr

SetNew

添加元素到点集中

CvSetElem* cvSetNew(CvSet* set_header);

- set_header: 集合

函数 cvSetNew 是 cvSetAdd 的变体，内联函数。它占用一新节点，并返回指向该节点的指针而不是索引。

SetRemoveByPtr

删除指针指向的集合元素

void cvSetRemoveByPtr(CvSet* set_header, void* elem);
set_header: 集合
elem: 被删除的元素
函数 cvSetRemoveByPtr 是一内联函数，是函数 cvSetRemove 轻微变化而来的。该函数并不会检查节点是否为空 -- 用户负责这一检查。

GetSetElem

通过索引值查找相应的集合元素

CvSetElem* cvGetSetElem(const CvSet* set_header, int index);
set_header: 集合
index: 索引值
函数 cvGetSetElem 通过索引值查找相应的元素。函数返回指向该元素的指针，如果索引值无效或相应的节点为空，则返回 0。 若函数使用 cvGetSeqElem 去查找节点，则函数支持负的索引值。

ClearSet

清空点集

void cvClearSet(CvSet* set_header);
set_header: 待清空的点集
函数 cvClearSet 删除集合中的所有元素。时间复杂度为 O(1).

Graphs

CvGraph

有向权图和无向权图

```
#define CV_GRAPH_VERTEX_FIELDS()    \
    int flags; /* vertex flags */   \
```

```
    struct CvGraphEdge* first; /* the first incident edge */

typedef struct CvGraphVtx
{
    CV_GRAPH_VERTEX_FIELDS()
}
CvGraphVtx;

#define CV_GRAPH_EDGE_FIELDS()      \
    int flags; /* edge flags */     \
    float weight; /* edge weight */ \
    struct CvGraphEdge* next[2]; /* the next edges in the incidence
lists for staring (0) */ \
                                /* and ending (1) vertices */ \
    struct CvGraphVtx* vtx[2]; /* the starting (0) and ending (1)
vertices */

typedef struct CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS()
}
CvGraphEdge;

#define  CV_GRAPH_FIELDS()                \
    CV_SET_FIELDS() /* set of vertices */   \
    CvSet* edges;   /* set of edges */

typedef struct CvGraph
{
    CV_GRAPH_FIELDS()
}
CvGraph;
```

在 OpenCV 图形结构中，CvGraph 是一基本结构。
图形结构继承自 CvSet -- 该部分描绘了普通图的属性和图的顶点，也包含了一个点集作为其成员 -- 该点集描述了图的边缘。利用宏（可以简化结构扩展和定制）使用与其它 OpenCV 可扩展结构一样的方法和技巧，同样的方法和技巧，我们声明了定点，边和头部结构。虽然顶点结构和边结构无法从 CvSetElem 显式地继承时，但它们满足点集元素的两个条件（在开始是有一个整数域和满足 CvSetElem 结构）。flags 域用来标记顶点和边是否已被占用或者处于其他目的，如：遍历图时（见：cvStartScanGraph 等），因此最好不要去直接使用它们。图代表的就是边的集合。存在有向和无向的区别。对于后者（无向图），在连接顶点 A 到 顶点 B 的边同连接顶点 B 到 顶点 A 的边是没什么区别的，在某一时刻，只可能存在一个，即：要么是<A, B>要么是<B, A>.

CreateGraph

创建一个空树

CvGraph* cvCreateGraph(int graph_flags, int header_size, int vtx_size,
int edge_size, CvMemStorage* storage);

graph_flags: 被创建的图的类型。通常，无向图为 CV_SEQ_KIND_GRAPH,无向图为 CV_SEQ_KIND_GRAPH | CV_GRAPH_FLAG_ORIENTED.

header_size: 头部大小；可能小于 sizeof(CvGraph)

vtx_size: 顶点大小；常规的定点结构必须来自 CvGraphVtx (使用宏 CV_GRAPH_VERTEX_FIELDS())

edge_size: 边的大小；常规的边结构必须来自 CvGraphEdge (使用宏 CV_GRAPH_EDGE_FIELDS())

storage: 图的容器

函数 cvCreateGraph 创建一空图并且返回指向该图的指针。

GraphAddVtx

插入一顶点到图中

int cvGraphAddVtx(CvGraph* graph, const CvGraphVtx* vtx=NULL,
CvGraphVtx** inserted_vtx=NULL);

graph: 图

vtx: 可选输入参数，用来初始化新加入的顶点（仅大小超过 sizeof(CvGraphVtx) 的用户自定义的域才会被拷贝）

inserted_vertex: 可选的输出参数。如果不为 NULL, 则传回新加入顶点的地址

函数 cvGraphAddVtx 将一顶点加入到图中，并返回定点的索引

GraphRemoveVtx

通过索引从图中删除一顶点

int cvGraphRemoveVtx(CvGraph* graph, int index);

graph: 图

vtx_idx: 被珊顶点的索引

函数 cvGraphRemoveAddVtx 从图中删除一顶点，连同删除含有此顶点的边。如果输入的顶点不属于该图的话，将报告删除出错（不存在而无法删除）。返回值为被删除的边数，如果顶点不属于该图的话，返回 -1。

GraphRemoveVtxByPtr

通过指针从图中删除一顶点

int cvGraphRemoveVtxByPtr(CvGraph* graph, CvGraphVtx* vtx);

graph: 图

vtx; 指向被删除的边的指针

函数 cvGraphRemoveVtxByPtr 从图中删除一顶点，连同删除含有此顶点的边。如果输入的顶点不属于该图的话，将报告删除出错（不存

在而无法删除）。返回值为被删除的边数，如果顶点不属于该图的话，返回 -1。

GetGraphVtx

通过索引值查找图的相应顶点

CvGraphVtx* cvGetGraphVtx(CvGraph* graph, int vtx_idx);

graph: 图

vtx_idx: 定点的索引值

函数 cvGetGraphVtx 通过索引值查找对应的顶点，并返回指向该顶点的指针，如果不存在则返回 NULL.

GraphVtxIdx

返回定点相应的索引值

int cvGraphVtxIdx(CvGraph* graph, CvGraphVtx* vtx);

graph: 图

vtx: 指向顶点的指针

函数 cvGraphVtxIdx 返回与顶点相应的索引值

GraphAddEdge

通过索引值在图中加入一条边

int cvGraphAddEdge(CvGraph* graph, int start_idx, int end_idx,
const CvGraphEdge* edge=NULL,
CvGraphEdge** inserted_edge=NULL);

graph: 图

start_idx; 边的起始顶点的索引值

end_idx: 边的尾部顶点的索引值（对于无向图，参数的次序无关紧要，即：start_idx 和 end_idx 可互为起始顶点和尾部顶点）

edge: 可选的输入参数，初始化边的数据

inserted_edge: 可选的输出参数，包含被插入的边的地址。

函数 cvGraphAddEdge 连接两特定的顶点。如果该边成功地加入到图中，返回 1； 如果连接两顶点的边已经存在，返回 0； 如果顶点没被发现（不存在）或者起始顶点和尾部顶点是同一个定点，或其他特殊情况，返回 -1。 如果是后者（即：返回值为负），函数默认的报告一个错误。

GraphAddEdgeByPtr

通过指针在图中加入一条边

int cvGraphAddEdgeByPtr(CvGraph* graph, CvGraphVtx* start_vtx,
CvGraphVtx* end_vtx,
const CvGraphEdge* edge=NULL,
CvGraphEdge** inserted_edge=NULL);
const CvGraphEdge* edge=NULL,
CvGraphEdge** inserted_edge=NULL)

graph:图

start_vtx: 指向起始顶点的指针
end_vtx: 指向尾部顶点的指针。对于无向图来说，顶点参数的次序无关紧要。
edge: 可选的输入参数，初始化边的数据
inserted_edge: 可选的输出参数，包含被插入的边的地址。
函数 cvGraphAddEdge 连接两特定的顶点。如果该边成功地加入到图中，返回 1； 如果连接两顶点的边已经存在，返回 0； 如果顶点没被发现（不存在）或者起始顶点和尾部顶点是同一个定点，或其他特殊情况，返回 -1。 如果是后者（即：返回值为负），函数默认的报告一个错误

GraphRemoveEdge

通过索引值从图中删除顶点

void cvGraphRemoveEdge(CvGraph* graph, int start_idx, int end_idx);
graph: 图
start_idx: 起始顶点的索引值
end_idx; 尾部顶点的索引值。对于无向图来说，顶点参数的次序无关紧要。
函数 cvGraphRemoveEdge 删除连接两特定顶点的边。若两顶点并没有相连接（即：不存在由这两个顶点连接的边），函数什么都不做。

GraphRemoveEdgeByPtr

通过指针从图中删除边

void cvGraphRemoveEdgeByPtr(CvGraph* graph, CvGraphVtx* start_vtx, CvGraphVtx* end_vtx);
graph: 图
start_vtx; 指向起始顶点的指针
end_vtx; 指向尾部顶点的指针。对于无向图来说，顶点参数的次序无关紧要。
函数 cvGraphRemoveEdgeByPtr 删除连接两特定顶点的边。若两顶点并没有相连接（即：不存在由这两个顶点连接的边），函数什么都不做。

FindGraphEdge

通过索引值在图中查找相应的边

CvGraphEdge* cvFindGraphEdge(const CvGraph* graph, int start_idx, int end_idx);
#define cvGraphFindEdge cvFindGraphEdge
graph: 图
start_idx: 其实顶点的索引值
end_idx; 尾部顶点的索引值。对于无向图来说，顶点参数的次序无关紧要

函数 cvFindGraphEdge 查找与两特定顶点相对应的边，并返回指向该边的指针。如果该边不存在，返回 NULL.

FindGraphEdgeByPtr

通过指针在图中查找相应的边

CvGraphEdge* cvFindGraphEdgeByPtr(const CvGraph* graph,
const CvGraphVtx* start_vtx,
const CvGraphVtx* end_vtx);
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
graph: 图
start_vtx; 指向起始顶点的指针
end_vtx; 指向尾部顶点的指针。对于无向图来说，顶点参数的次序无关紧要。
函数 cvFindGraphEdgeByPtr 查找与两特定顶点相对应的边，并返回指向该边的指针。如果该边不存在，返回 NULL

GraphEdgeIdx

返回与该边相应的索引值

int cvGraphEdgeIdx(CvGraph* graph, CvGraphEdge* edge);
graph: 图
edge: 指向该边的指针
函数 cvGraphEdgeIdx 返回与边对应的索引值。

GraphVtxDegree

（通过索引值）统计与顶点相关联的边数

int cvGraphVtxDegree(const CvGraph* graph, int vtx_idx);
graph: 图
vtx_idx: 顶点对应的索引值
函数 cvGraphVtxDegree 返回与特定顶点相关联的边数，包括以该顶点为起始顶点的和尾部顶点的。统计边数，可以适用下列代码：

```
CvGraphEdge* edge = vertex->first; int count = 0;
while( edge )
{
    edge = CV_NEXT_GRAPH_EDGE( edge, vertex );
    count++;
}
```

宏 CV_NEXT_GRAPH_EDGE(edge, vertex) 返回依附于该顶点的下一条边。

GraphVtxDegreeByPtr

（通过指针）统计与顶点相关联的边数

int cvGraphVtxDegreeByPtr(const CvGraph* graph, const CvGraphVtx* vtx);
graph: 图
vtx: 顶点对应的指针
函数 cvGraphVtxDegreeByPtr 返回与特定顶点相关联的边数，包括以该顶点为起始顶点的和尾部顶点的

ClearGraph

删除图
void cvClearGraph(CvGraph* graph);
graph:图
函数 cvClearGraph 删除该图的所有顶点和边。时间复杂度为 O(1).

CloneGraph

克隆图
CvGraph* cvCloneGraph(const CvGraph* graph, CvMemStorage* storage);
graph: 待拷贝的图
storage: 容器，存放拷贝
函数 cvCloneGraph 创建图的完全拷贝。如果顶点和边含有指向外部变量的指针，那么图和它的拷贝共享这些指针。在新的图中，顶点和边可能存在不同，因为函数重新分割了顶点和边的点集。

CvGraphScanner

图的遍历

```
typedef struct CvGraphScanner
{
    CvGraphVtx* vtx;        /* current graph vertex (or current edge origin) */
    CvGraphVtx* dst;        /* current graph edge destination vertex */
    CvGraphEdge* edge;      /* current edge */

    CvGraph* graph;         /* the graph */
    CvSeq*   stack;         /* the graph vertex stack */
    int      index;         /* the lower bound of certainly visited vertices */
    int      mask;          /* event mask */
}
CvGraphScanner;
```

结构 cvGraphScanner 深度遍历整个图。 函数的相关讨论如下（看：StartScanGraph)

StartScanGraph

创建一结构，用来对图进行深度遍历

CvGraphScanner* cvCreateGraphScanner(CvGraph* graph, CvGraphVtx* vtx=NULL, int mask=CV_GRAPH_ALL_ITEMS);

graph: 图

vtx: 开始遍历的（起始）顶点。如果为 NULL, 便利就从第一个顶点开始（指：顶点序列中，具有最小索引值的顶点）

mask: 事件掩码（event mask)代表用户感兴趣的事件（此时 函数 cvNextGraphItem 将控制返回给用户）。这个只可能是 CV_GRAPH_ALL_ITEMS (如果用户对所有的事件都感兴趣的话）或者是下列标志的组合:

CV_GRAPH_VERTEXT -- 在第一次被访问的顶点处停下

CV_GRAPH_TREE_EDGE -- 在 tree edge 处停下(tree edge 指连接最后被访问的顶点与接下来被访问的顶点的边）

CV_GRAPH_BACK_EDGE -- 在 back edge 处停下（back edge 指连接最后被访问的顶点与其在搜索树中祖先的边）

CV_GRAPH_FORWARD_EDGE -- 在 forward edge 处停下（forward edge 指连接最后被访问的顶点与其在搜索树中后裔的边）

CV_GRAPH_CROSS_EDGE -- 在 cross edge 处停下（cross edge 指连接不同搜索树中或同一搜索树中不同分支的边.只有在有向图中， 才存在着一 概念）

CV_GRAPH_ANY_EDGE -- 在 any edge 处停下（any edge 指任何边，包括 tree edge, back edge, forward edge, cross edge)

CV_GRAPH_NEW_TREE -- 在每一个新的搜索树开始处停下。首先遍历从起始顶点开始可以访问到的顶点和边，然后查找搜索图中访问不到的顶点或边并恢复遍历。在开始遍历一颗新的树时（包括第一次调用 cvNextGraphItem 时的树），产生 CV_GRAPH_NEW_TREE 事件。

函数 cvCreateGraphScanner 创建一结构用来深度遍历搜索树。函数 cvNextGraphItem 要使用该初始化了的结构 -- 层层遍历的过程。

NextGraphItem

逐层遍历整个图

int cvNextGraphItem(CvGraphScanner* scanner);

scanner: 图的遍历状态。被此函数更新。

函数 cvNextGraphItem 遍历整个图，直到用户感兴趣的事件发生（即：调用 cvCreateGraphScanner 时， mask 对应的事件）或遍历结束。在前面一种情况下，函数返回 参数 mask 相应的事件，当再次调用函数时，恢复遍历）。在后一种情况下，返回 CV_GRAPH_OVER(-1)。当 mask 相应的事件为 CV_GRAPH_BACKTRACKING 或 CV_GRAPH_NEW_TEEE 时， 当前正在被访问的顶点被存放在 scanner->vtx 中。如果事件与 边 edge 相关，那么 edge 本身被存

放在 scanner->edge，该边的起始顶点存放在 scanner->vtx 中， 尾部节点存放在 scanner->dst 中。

ReleaseGraphScanner

完成图地遍历过程

```
void cvReleaseGraphScanner( CvGraphScanner** scanner );
```

scanner: 指向遍历器的指针.

函数 cvGraphScanner 完成图的遍历过程，并释放遍历器的状态。

Trees

CV_TREE_NODE_FIELDS

用于树结点类型声明的（助手）宏

```
#define CV_TREE_NODE_FIELDS(node_type)
\
    int          flags;             /* micsellaneous flags */              \
    int          header_size;     /* size of sequence header */          \
    struct       node_type* h_prev; /* previous sequence */               \
    struct       node_type* h_next; /* next sequence */                   \
    struct       node_type* v_prev; /* 2nd previous sequence */           \
    struct       node_type* v_next; /* 2nd next sequence */
```

宏 CV_TREE_NODE_FIELDS() 用来声明一层次性结构，例如 CvSeq -- 所有动态结构的基本类型。如果树的节点是由该宏所声明的，那么就可以使用（该部分的）以下函数对树进行相关操作。

CvTreeNodeIterator

打开现存的存储结构或者创建新的文件存储结构

```
typedef struct CvTreeNodeIterator
{
    const void* node;
    int level;
    int max_level;
}
CvTreeNodeIterator;
```

结构 CvTreeNodeIterator 用来对树进行遍历。该树的节点是由宏 CV_TREE_NODE_FIELDS 声明。

InitTreeNodeIterator

用来初始化树结点的迭代器

```
void cvInitTreeNodeIterator( CvTreeNodeIterator* tree_iterator,
                             const void* first, int max_level );
```

tree_iterator: 初始化了的迭代器

first: （开始）遍历的第一个节点
max_level: 限制对树进行遍历的最高层（即：第 max_level 层）(假设第一个节点所在的层为第一层）。例如：1 指的是遍历第一个节点所在层，2 指的是遍历第一层和第二层
函数 cvInitTreeNodeIterator 用来初始化树的迭代器。

NextTreeNode

返回当前节点，并将迭代器 iterator 移向当前节点的下一个节点
void* cvNextTreeNode(CvTreeNodeIterator* tree_iterator);
tree_iterator: 初始化了的迭代器
函数 cvNextTreeNode 返回当前节点并且更新迭代器（iterator) -- 并将 iterator 移向（当前节点）下一个节点。换句话说，函数的行为类似于表达式 *p++ (通常的 C 指针 或 C++ 集合迭代器）。 如果没有更多的节点（即：当前节点为最后的节点），则函数返回值为 NULL.

PrevTreeNode

返回当前节点，并将迭代器 iterator 移向当前节点的前一个节点
void* cvPrevTreeNode(CvTreeNodeIterator* tree_iterator);
tree_iterator: 初始化了的迭代器
函数 cvPrevTreeNode 返回当前节点并且更新迭代器（iterator) -- 并将 iterator 移向（当前节点的）前一个节点。换句话说，函数的行为类似于表达式 *p-- (通常的 C 指针 或 C++ 集合迭代器）。 如果没有更多的节点（即：当前节点为头节点），则函数返回值为 NULL.

TreeToNodeSeq

将所有的节点指针（即：指向树结点的指针）收集到线性表 sequence 中
CvSeq* cvTreeToNodeSeq(const void* first, int header_size, CvMemStorage* storage);
first: 初始树结点
header_size: 线性表的表头大小，大小通常为 sizeof(CvSeq)
函数 cvTreeToNodeSeq 将树的节点指针挨个的存放到线性表中。存放的顺序以深度为先。

InsertNodeIntoTree

将新的节点插入到树中
void cvInsertNodeIntoTree(void* node, void* parent, void* frame);
node: 待插入的节点
parent: 树中的父节点（即：含有子节点的节点）
frame: 顶部节点。如果 节点 parent 等同于 节点 frame， 则将节点的域 v_prev 设为 NULL 而不是 parent.
函数 cvInsertNodeIntoTree 将另一个节点插入到树中。函数不分配任何内存，仅仅修改树节点的连接关系。

RemoveNodeFromTree

从树中删除节点

void cvRemoveNodeFromTree(void* node, void* frame);

node：待删除的节点。

frame: 顶部节点。如果 node->v.prev = NULL 且 node->h.prev = NULL, 则将 frame->v.next 设为 node->h.next

函数 cvRemoveNodeFromTree 从树中删除节点。它不会释放任何内存，仅仅修改树中节点的连接关系

绘图函数

绘图函数作用于任何象素深度的矩阵/图像. Antialiasing 技术只能在 8 位图像上实现.所有的函数包括彩色图像的色彩参数(色彩参数是指 rgb 它是由宏 CV_RGB 或 cvScalar 函数构成。）和灰度图像的亮度。.如果一幅绘制图形部分或全部位于图像之外，那么对它先做裁剪。 对于彩色图像正常的色彩通道是 B(蓝),G(绿),R(红)..。如果需要其它的色彩，可以通过 cvScalar 中的特殊色彩通道构造色彩，或者在绘制图像之前或之后 使用 cvCvtColor 或者 cvTransform 来转换。

曲线与形状

CV_RGB

创建 一个色彩值.

#define CV_RGB(r, g, b) cvScalar((b), (g), (r))

Line

绘制连接两个点的线段

void cvLine(CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color,
int thickness=1, int line_type=8, int shift=0);

`img`
图像。

`pt1`
线段的第一个端点。

`pt2`
线段的第二个端点。

`color`
线段的颜色。

`thickness`
线段的粗细程度。

`line_type`

线段的类型。
8 (or 0) - 8-（connected line)连接 线。
4 - 4-(connected line)连接线。
CV_AA - antialiased 线条。
`shift`
坐标点的小数点位数。

函数cvLine 在图像中的点 1 和点 2 之间画一条线段。线段被图像或感兴趣的矩形所裁剪。对于具有整数坐标的 non-antialiasing 线条，使用 8-连接或者 4-连接Bresenham 算法。画粗线条时结尾是圆形的。画 antialiased 线条使用高斯滤波。对于特殊颜色的线条使用宏 CV_RGB(r, g, b)。

Rectangle

绘制简单、指定粗细或者带填充的 矩形

void cvRectangle(CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color,
int thickness=1, int line_type=8, int shift=0);

`img`
图像.
`pt1`
矩形的一个顶点。
`pt2`
矩形对角线上的另一个顶点
`color`
线条颜色 (RGB) 或亮度（灰度图像 ）(grayscale image)。
`thickness`
组成矩形的线条的粗细程度。取负值时（如 CV_FILLED）函数绘制填充了色彩的矩形。
`line_type`
线条的类型。见 cvLine 的描述
`shift`
坐标点的小数点位数。

函数 cvRectangle 通过对角线上的两个顶点绘制矩形。

Circle

绘制圆形。

void cvCircle(CvArr* img, CvPoint center, int radius, CvScalar color,
int thickness=1, int line_type=8, int shift=0);

`img`
图像。
`center`
圆心坐标。
`radius`
圆形的半径。

color
线条的颜色。
thickness
如果是正数，表示组成圆的线条的粗细程度。否则，表示圆是否被填充。
line_type
线条的类型。见 cvLine 的描述
shift
圆心坐标点和半径值的小数点位数。
函数cvCircle绘制或填充一个给定圆心和半径的圆。圆被感兴趣矩形所裁剪。 若指定圆的颜色，可以使用宏 CV_RGB (r, g, b)。

Ellipse

绘制椭圆圆弧和椭圆扇形。

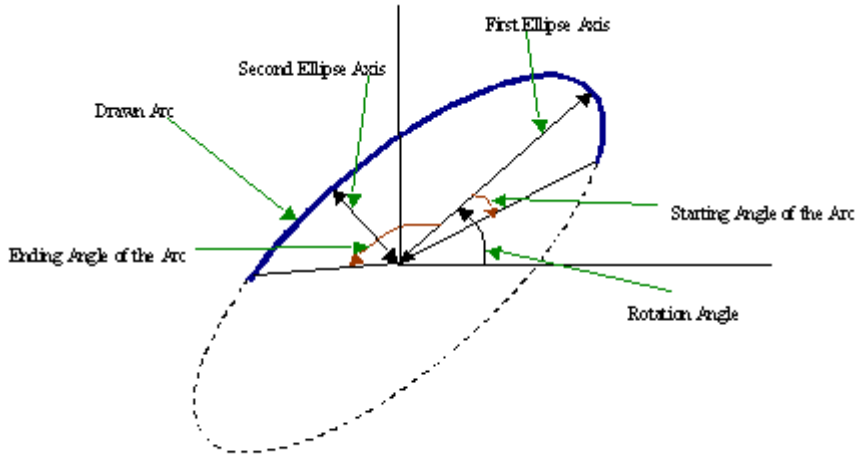
void cvEllipse(CvArr* img, CvPoint center, CvSize axes, double angle,
double start_angle, double end_angle, CvScalar color,
int thickness=1, int line_type=8, int shift=0);

img
图像。
center
椭圆圆心坐标。
axes
轴的长度。
angle
偏转的角度。
start_angle
圆弧起始角的角度。.
end_angle
圆弧终结角的角度。
color
线条的颜色。
thickness
线条的粗细程度。
line_type
线条的类型,见 CVLINE 的描述。
shift

圆心坐标点和数轴的精度。

函数 cvEllipse 用来绘制或者填充一个简单的椭圆弧或椭圆扇形。圆弧被 ROI 矩形所忽略。反走样弧线和粗弧线使用线性分段近似值。所有的角都是以角度的形式给定的。图片下面要解释参数的含义。

Parameters of Elliptic Arc



FillPoly

填充多边形内部

void cvFillPoly(CvArr* img, CvPoint** pts, int* npts, int contours,
CvScalar color, int line_type=8, int shift=0);

`img`

图像初始化。

`pts`

多边形的顶点坐标集合。

`npts`

多边形的顶点个数。

`contours`

组成填充区域的线段的数量。

`color`

多边形的颜色。

`line_type`

组成多边形的线条的类型。

`shift`

顶点坐标的小数点位数。

函数 cvFillPoly 用于一个单独被多变形轮廓所限定的区域内进行填充。
函数可以填充复杂的区域,例如，有漏洞的区域和有交叉点的区域等等。

FillConvexPoly

填充多边形外部

void cvFillConvexPoly(CvArr* img, CvPoint* pts, int npts,
CvScalar color, int line_type=8, int
shift=0);

`img`

图像初始化。

`pts`

多边形的定点坐标集合。

`npts`

多边形的定点个数.

`color`

多边形的颜色。.

`line_type`

组成多边形的线条的类型。

`shift`

顶点坐标的小数点位数。

函数 cvFillConvexPoly 填充多边形限定区域的。这个函数比函数 cvFillPoly 在响应速度上更快。它除了可以填充多边形区域的外部还可以填充任何的单调多边形。例如：一个被水平线（扫描线）至多两次截断的多边形。

PolyLine

绘制多边形。

void cvPolyLine(CvArr* img, CvPoint** pts, int* npts, int contours, int is_closed,

CvScalar color, int thickness=1, int line_type=8, int shift=0);

`img`

图像初始化。

`pts`

多边形的定点集合。

`npts`

多边形的定点个数。

`contours`

多边形的线段数量。

`is_closed`

指出多边形是否绘制完毕。如果完毕，函数将起始点和结束点连线。

`color`

多边形的颜色。

`thickness`

线条的粗细程度。

`line_type`

线段的类型。

`shift`

顶点的小数点位数。

函数 cvPolyLine 绘制一个简单的或多样的多角曲线。

文本

InitFont

字体结构初始化。

void cvInitFont(CvFont* font, int font_face, double hscale,

double vscale, double shear=0,
int thickness=1, int line_type=8);

font

字体初始化。

font_face

字体名称标识符。来源于 Hershey 字体集(http://sources.isc.org/utils/misc/hershey-font.txt) 。

CV_FONT_HERSHEY_SIMPLEX - 正常大小无衬线字体。

CV_FONT_HERSHEY_PLAIN - 小号无衬线字体。

CV_FONT_HERSHEY_DUPLEX - 正常大小无衬线字体。(比CV_FONT_HERSHEY_SIMPLEX更复杂)

CV_FONT_HERSHEY_COMPLEX - 正常大小有衬线字体。

CV_FONT_HERSHEY_TRIPLEX - 正常大小有衬线字体(比CV_FONT_HERSHEY_COMPLEX更复杂)

CV_FONT_HERSHEY_COMPLEX_SMALL - CV_FONT_HERSHEY_COMPLEX 的小译本。

CV_FONT_HERSHEY_SCRIPT_SIMPLEX - 手写风格字体。

CV_FONT_HERSHEY_SCRIPT_COMPLEX - 比CV_FONT_HERSHEY_SCRIPT_SIMPLEX更复杂。

参数能够由一个值和可选择的CV_FONT_ITALIC 字体标记合成。就是斜体字。

hscale

字体宽度。如果等于 1.0f，字符的宽度是最初的字体宽度。如果等于 0.5f，字符的宽度是最初的字体宽度的一半。

vscale

字体高度。如果等于 1.0f，字符的高度是最初的字体高度。如果等于 0.5f，字符的高度是最初的字体高度的一半。

shear

字体的斜度。当值为 0 时 ，字符不倾斜。当值为 1.0f 时，字体倾斜 45 度。

thickness

字体笔划的粗细程度。.

line_type

字体笔划的类型，见 cvLine 的描述。

函数 cvInitFont 完成对文本的描述。

PutText

在图像中加入文本。

void cvPutText(CvArr* img, const char* text, CvPoint org, const CvFont* font, CvScalar color);

img

输入图像。

text

显示字符串。

org

第一个字符左下角的坐标。

font

字体结构初始化。

`color`

文本的字体颜色。

函数 cvPutText 将具有指定字体的和指定颜色的文本加载到图像中。加载到图像中的文本被感兴趣的矩形框圈定。

GetTextSize

设置字符串文本的宽度和高度。

void cvGetTextSize(const char* text_string, const CvFont* font, CvSize* text_size, int* baseline);

`font`

字体结构初始化

`text_string`

输入字符串。

`text_size`

合成字符串的字符的大小。文本的高度不包括基线以下的部分。

`baseline`

基线长度。

函数 cvGetTextSize 是用于当特殊的字体在指定的字符串中被使用的时候计算 binding rectangle 的。

点集和轮廓

DrawContours

在图像中绘制简单的和复杂的轮廓。

```
void cvDrawContours( CvArr *img, CvSeq* contour,
                     CvScalar external_color, CvScalar
hole_color,
                     int max_level, int thickness=1,
                     int line_type=8 );
```

`img`

图像初始化。

`contour`

指针指向初始轮廓。

`external_color`

外层轮廓的颜色。

`hole_color`

内层轮廓的颜色。

`max_level`

绘制轮廓的最大等级。如果等级为 0，绘制单独的轮廓。如果为 1，在相同的级别下绘制轮廓。如果值为 2，所有的轮廓。If 2, all contours after and all contours one level below the contours are drawn, etc.如果值为负数，函数不能绘制轮廓If the value is negative, the function does not draw the contours following after contour but draws child contours of contour up to

abs(max_level)-1 level.

`thickness`

绘制轮廓时所使用的线条的粗细度。如果值为负(e.g. =CV_FILLED),绘制内层轮廓。

`line_type`

线条的类型。

函数cvDrawContours cvDrawContours 如果 thickness>=0 在图像中绘制轮廓。如果 thickness<0 在限定的区域内绘制轮廓。

示例. 通过 contour 函数探测连通分支量

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

int main( int argc, char** argv )
{
    IplImage* src;
    // 第一条命令行参数确定了图像的文件名。
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* contour = 0;

        cvThreshold( src, src, 1, 255, CV_THRESH_BINARY );
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvFindContours( src, storage, &contour, sizeof(CvContour),
CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );
        cvZero( dst );

        for( ; contour != 0; contour = contour->h_next )
        {
            CvScalar color = CV_RGB( rand()&255, rand()&255,
rand()&255 );
            /* 用 1 替代 CV_FILLED 所指示的轮廓外形 */
            cvDrawContours( dst, contour, color, color, -1,
CV_FILLED, 8 );
        }

        cvNamedWindow( "Components", 1 );
        cvShowImage( "Components", dst );
        cvWaitKey(0);
    }
}
```

在样本中用 1 替代 CV_FILLED 所指示的轮廓外形。

数据保存和运行时类型信息

文件存储

CvFileStorage

文件存储器的初始化

```
typedef struct CvFileStorage
{
    ...         // hidden fields
} CvFileStorage;
```

构造函数 CvFileStorage 是将磁盘上存储的文件关联起来的“黑匣子”。在下列函数描述中利用CvFileStorage 输入并允许存储或打开由标量值组成的层次集合,根据 CXCore 对象(例如 矩阵,序列，图表)和用户自定义对象。

CXCore 能将数据读入或写入 XML (http://www.w3c.org/XML) or YAML (http://www.yaml.org) 格式. 下面这个例子是利用CXCore函数将 3×3 单位浮点矩阵存入XML 和 YAML文档。

XML:

```
<?xml version="1.0">
<opencv_storage>
<A type_id="opencv-matrix">
   <rows>3</rows>
   <cols>3</cols>
   <dt>f</dt>
   <data>1. 0. 0. 0. 1. 0. 0. 0. 1.</data>
</A>
</opencv_storage>
```

YAML:

```
%YAML:1.0
A: !!opencv-matrix
   rows: 3
   cols: 3
   dt: f
   data: [ 1., 0., 0., 0., 1., 0., 0., 0., 1.]
```

从例子中可以看到, XML 试用嵌套标签来表现层次，而 YAML 用缩排来表现（类似于 Python 语言） 。

相同的 CXCore 函数也能够在这两种格式下读写数据，特殊的格式决定了文件的扩展名， .xml 是 XML 的扩展名， .yml 或 .yaml 是 YAML 的扩展名。

CvFileNode

文件存储器节点

```
/* 文件节点类型 */
```

```
#define CV_NODE_NONE            0
#define CV_NODE_INT             1
#define CV_NODE_INTEGER         CV_NODE_INT
#define CV_NODE_REAL            2
#define CV_NODE_FLOAT           CV_NODE_REAL
#define CV_NODE_STR             3
#define CV_NODE_STRING          CV_NODE_STR
#define CV_NODE_REF             4 /* not used */
#define CV_NODE_SEQ             5
#define CV_NODE_MAP             6
#define CV_NODE_TYPE_MASK       7

/*  可选标记  */
#define CV_NODE_USER            16
#define CV_NODE_EMPTY           32
#define CV_NODE_NAMED           64

#define CV_NODE_TYPE(tag)    ((tag) & CV_NODE_TYPE_MASK)

#define CV_NODE_IS_INT(tag)             (CV_NODE_TYPE(tag) ==
CV_NODE_INT)
#define CV_NODE_IS_REAL(tag)            (CV_NODE_TYPE(tag) ==
CV_NODE_REAL)
#define CV_NODE_IS_STRING(tag)          (CV_NODE_TYPE(tag)
== CV_NODE_STRING)
#define CV_NODE_IS_SEQ(tag)             (CV_NODE_TYPE(tag) ==
CV_NODE_SEQ)
#define CV_NODE_IS_MAP(tag)             (CV_NODE_TYPE(tag)
== CV_NODE_MAP)
#define CV_NODE_IS_COLLECTION(tag) (CV_NODE_TYPE(tag)
>= CV_NODE_SEQ)
#define CV_NODE_IS_FLOW(tag)            (((tag) &
CV_NODE_FLOW) != 0)
#define CV_NODE_IS_EMPTY(tag)           (((tag) &
CV_NODE_EMPTY) != 0)
#define CV_NODE_IS_USER(tag)            (((tag) &
CV_NODE_USER) != 0)
#define CV_NODE_HAS_NAME(tag)           (((tag) &
CV_NODE_NAMED) != 0)

#define CV_NODE_SEQ_SIMPLE 256
#define CV_NODE_SEQ_IS_SIMPLE(seq) (((seq)->flags &
CV_NODE_SEQ_SIMPLE) != 0)

typedef struct CvString
{
    int len;
```

```
    char* ptr;
}
CvString;

/*关键字 readed 的原理是 文件存储器对存储的无用信息进行加速查找操作 */
typedef struct CvStringHashNode
{
    unsigned hashval;
    CvString str;
    struct CvStringHashNode* next;
}
CvStringHashNode;

/* 文件存储器的基本元素是-标量或集合*/
typedef struct CvFileNode
{
    int tag;
    struct CvTypeInfo* info; /* 类型信息（只能用于用户自定义对象，对于其它对象它为 0） */
    union
    {
        double f; /* 浮点数*/
        int i;    /* 整形数 */
        CvString str; /* 字符文本 */
        CvSeq* seq; /* 序列 (文件节点的有序集合) */
        struct CvMap* map; /*图表 (指定的文件节点的集合 ) */
    } data;
}
CvFileNode;
```

这个构造函数只是用于重新找到文件存储器上的数据(例如 ，从文件中下载数据）。 当数据已经写入文件时利用最小的缓冲继续完成，此时没有数据存放在文件存储器。

相反，当从文件中读数据时，所有文件在内存中像树一样被解析和描绘。树的每一个节点被CvFileNode表现出来。文件节点N的类型能够通过CV_NODE_TYPE(N->tag) 被重新找到。一些节点（叶结点）作为变量：字符串文本，整数，浮点数。其它的文件节点是集合文件节点,有两个类型集合：序列和图表 (我们这里使用 YAML 符号,无论用哪种方法，对于XML是同样有效）。序列(不要与CvSeq混淆） 是由有序的非指定文件节点构成的，图表是由无序的指定文件节点构成的。因而 ,序列的原理是通过索引（cvGetSepElem)来存取，图形的原理是通过姓名（cvGetFileNodeByName)来存取 下表描述不同类型的节点：

Type	CV_NODE_TYPE(node->tag)	Value
Integer	CV_NODE_INT	node->data.i

Floating-point	CV_NODE_REAL	node->data.f
Text string	CV_NODE_STR	node->data.str.ptr
Sequence	CV_NODE_SEQ	node->data.seq
Map	CV_NODE_MAP	node->data.map*

*

这里不需要存取图表内容（顺便说一下CvMapThere 是一个隐藏的构造函数）。图形原理可以被cvGetFileNodeByName函数重新得到通过指针指向图表文件节点。

一个用户对象是一个标准的类型实例，例如 CvMat, CvSeq 等,或者任何一个已注册的类型使用 cvRegisterTypeInfo。这样的对象，例如图表（像表现 XML 和 YAM 示例文件一样）是最初在文件中表现出来的 。在文件存储器打开并分析之后。当用户调用 cvRead 或 cvReadByName 函数时 那么对象将请求被解析 (内存中原来的表述）。

CvAttrList

显示属性

```
typedef struct CvAttrList
{
    const char** attr; /* NULL-将中止一对数组 (attribute_name,attribute_value) */
    struct CvAttrList* next; /* 指针指向下一个属性块 */
}
CvAttrList;

/* initializes 初始化构造函数 CvAttrList */
inline CvAttrList cvAttrList( const char** attr=NULL, CvAttrList* next=NULL );

/* 返回值为属性值，找不到适合的属性则返回值为 0(NULL)*/
const char* cvAttrValue( const CvAttrList* attr, const char* attr_name );
```

在执行当前的属性时通常需通过特定的参数来完成，除了对象类型说明(type_id 属性)以外，标示符不支持 XML 属性。

OpenFileStorage

打开文件存储器读/写数据。

```
CvFileStorage* cvOpenFileStorage( const char* filename, CvMemStorage* memstorage, int flags );
```

`filename`

内存中的相关文件的文件名。

内存中通常存储临时数据和动态结构，例如 CvSeq 和 CvGraph。如果 memstorage 为空,将建立和使用一个暂存器。

`flags`

读/写选择器。

CV_STORAGE_READ - 内存处于读状态。

CV_STORAGE_WRITE - 内存处于写状态。

函数cvOpenFileStorage打开文件存储器读写数据，之后建立文件或继续使用现有的文件 。文件扩展名决定读文件的类型 ： .xml 是 *XML*的扩展名, .yml 或 .yaml 是 *YAML*的扩展名。该函数的返回指针指向CvFileStorage构造函数。

ReleaseFileStorage

释放文件存储单元

void cvReleaseFileStorage(CvFileStorage** fs);

`fs`

双指针指向被关闭的文件存储器。

函数 cvReleaseFileStorage 关闭一个相关的文件存储器并释放所有的临时内存。只有在内存的 I/O 操作完成后才能关闭文件存储器。

写数据

StartWriteStruct

向文件存储器中写数据

void cvStartWriteStruct(CvFileStorage* fs, const char* name,
int struct_flags, const char*
type_name=NULL,
CvAttrList attributes=cvAttrList());

`fs`

初始化文件存储器。

`name`

被写入的数据结构的名称。在内存被读取时可以通过名称访问数据结构。

`struct_flags`

有下列两个值：

CV_NODE_SEQ - 被写入的数据结构为序列结构。这样的数据没有名称。

CV_NODE_MAP - 被写入的数据结构为图表结构。这样的数据含有名称。

有且只有这两个表识符被指定

CV_NODE_FLOW - 这个可选择标识符只能作用于YAML流。被写入的数据结构被看做一个数据流（不是数据块），它更加紧凑是所有标量的基础。

`type_name`

可选参数 - 对象类型名称。如果是XML用打开标识符type_id 属性写入。如果是YAML 用冒号后面的数据结构名写入， 基本上它是伴随用户对象出现的。当存储器读时，编码类型名通常决定对象类型（见Cvtypeinfo和

cvfindtypeinfo） 。

`attributes`

这个参数通常不会被当前执行。

函数 cvStartWriteStruct 开始写复合的数据结构（数据集合）包括序列或图表，标量和结构被写入， cvEndWriteStruct 被指定．该函数能够合并一些对象或写入一些用户对象函数。

EndWriteStruct

中止写数据结构

void cvEndWriteStruct(CvFileStorage* fs);

`fs`

初始化文件存储器。

函数cvEndWriteStruct 中止普通的写数据操作。

WriteInt

写入一个整形值

void cvWriteInt(CvFileStorage* fs, const char* name, int value);

`fs`

初始化文件存储器。

`name`

写入值的名称 。如果母结构是一个序列，name 的值为 NULL。

`value`

写入的整形值。

函数 cvWriteInt 将一个单独的整形值（有名称的或无名称的）写入文件存储器。

WriteReal

写入一个浮点形值

void cvWriteReal(CvFileStorage* fs, const char* name, double value);

`fs`

初始化文件存储器。

`name`

写入值的名称 。如果母结构是一个序列，name 的值为 NULL。

`value`

写入的浮点形值。

函数 cvWriteReal 将一个单独的整形值（有名称的或无名称的）写入文件存储器。 特殊的值被编码：The special values are encoded: NaN (一个编码量的非) 例如 .NaN, ±无穷大 例如 +.Inf (-.Inf)。

下面的实例表明 怎样使用低级写函数存储自定义数据结构。

```
void write_termcriteria( CvFileStorage* fs, const char* struct_name,
                         CvTermCriteria* termcrit )
{
```

```
        cvStartWriteStruct( fs, struct_name, CV_NODE_MAP, NULL,
    cvAttrList(0,0));
        cvWriteComment( fs, "termination criteria", 1 ); //  正确的描述
        if( termcrit->type & CV_TERMCRIT_ITER )
            cvWriteInteger( fs, "max_iterations", termcrit->max_iter );
        if( termcrit->type & CV_TERMCRIT_EPS )
            cvWriteReal( fs, "accuracy", termcrit->epsilon );
        cvEndWriteStruct( fs );
    }
```

WriteString

写入字符串文本

```
void    cvWriteString( CvFileStorage* fs, const char* name,
                               const char* str, int quote=0 );
```

`fs`

初始化文件存储器。

`name`

写入字符串的名称 。如果母结构是一个序列，name 的值为 NULL。

`str`

写入的字符串文本。

`quote`

如果不为 0，不管是否标准，字符串都将被写入 。如果标识符为 0。只有在标准的情况下（字符串的首位是数字或者空格）字符串被写入。

函数 cvWriteString将字符串文本写入文件存储器。

WriteComment

写入注释

```
void    cvWriteComment( CvFileStorage* fs, const char* comment, int
eol_comment );
```

`fs`

初始化文件存储器。

`comment`

写入的注释,单行的或者多行的。

`eol_comment`

如果不为 0，函数将注释加到当前行的后面。 如果为 0，并且是多行注释或者结尾没有注释行，那么注释将从新的一行开始 。

函数 cvWriteComment将注释写入文件存储器。读内存时注释将被跳过，它只能被用于调试和描述。

StartNextStream

打开下一个数据流

```
void    cvStartNextStream( CvFileStorage* fs );
```

`fs`

初始化文件存储器。

函数 cvStartNextStream 从文件存储器中打开下一个数据流。 YAML 和 XML 都支持多数据流。这对连接多个文件和恢复写入的程序很有用。

Write

写入用户对象

```
void    cvWrite( CvFileStorage* fs, const char* name,
                 const void* ptr, CvAttrList attributes=cvAttrList() );
```

`fs`

初始化文件存储器。

`name`

写入对象的名称 。如果母结构是一个序列，name 的值为 NULL。

`ptr`

定义指针指向对象。

`attributes`

定义对象的属性。

函数 cvWrite将对象写入文件存储器 。首先，使用cvTypeOf 查找恰当的类型信息。其次写入指定的方法类型信息。

属性被用于定制输入程序。下面的属性支持标准类型 The standard types support the following attributes (所有的*dt 属性在 cvWriteRawData中都有相同的格式）：

CvSeq

- header_dt -序列首位用户区的描述，它紧跟在CvSeq或CvChain（如果是自由序列）或CvContour（如果是轮廓或点序列）之后。
- dt - 序列原理的描述 。
- recursive - 如果属性没有被引用并且不等于“0”或“false"，则所有的序列树（轮廓）都被存储。

CvGraph

- header_dt - 图表头用户区的描述，它紧跟在 CvGraph之后。
- vertex_dt - 图表顶点用户区的描述。
- edge_dt - 图表边用户区的描述（ 注意磅值经常被写入，所以不需要详细的说明）。

下面的代码的含义是建立YAML文件用来描述CvFileStorage ：

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvMat* mat = cvCreateMat( 3, 3, CV_32F );
    CvFileStorage* fs = cvOpenFileStorage( "example.yml", 0,
CV_STORAGE_WRITE );

    cvSetIdentity( mat );
    cvWrite( fs, "A", mat, cvAttrList(0,0) );
```

```
    cvReleaseFileStorage( &fs );
    cvReleaseMat( &mat );
    return 0;
}
```

WriteRawData

写入重数

```
void    cvWriteRawData( CvFileStorage* fs, const void* src,
                                    int len, const char* dt );
```

`fs`

初始化文件存储器。

`src`

指针指向输入数组。

`len`

写入数组的长度。

`dt`

下面是每一个数组元素说明的格式：([count]{'u'|'c'|'w'|'s'|'i'|'f'|'d'})...，这些特性与C语言的类型相似 ：

- 'u' - 8 位无符号数。
- 'c' - 8 位符号数。
- 'w' - 16 位无符号数。
- 's' - 16 位符号数。
- 'i' - 32 位符号数。
- 'f' - 单精度浮点数。
- 'd' - 双精度浮点数。
- 'r' - 指针。输入的带符号的低 32 位整数。 这个类型常被用来存储数据结构。用来连接两个基本项。

count 是用来选择计数值的确定类型。 例如, dt='2if' 是指任意的一个数组元素的结构是：2 个字节整形数，后面跟一个单精度浮点数。上面的说明与 'iif', '2i1f' 等相同。另外一个例子：dt='u'是指 一个由类型组成的数组，dt='2d'是指由两个双精度浮点数构成的数组。

函数 cvWriteRawData 将重数写入文件存储器。那些元素由单独的重数构成。一些函数命令能够被循环调用替换包括 cvWriteInt 和 cvWriteReal命令，但是一个单独的命令更加有效。 注意，那是因为元素没有名字， 把它们写入序列比写入图表要好。

WriteFileNode

将文件节点写入另一个文件存储器

```
void cvWriteFileNode( CvFileStorage* fs, const char*
new_node_name,
                                    const CvFileNode* node, int embed );
```

`fs`

设置目的文件存储器

new_file_node

在目的文件存储器中设置新的文件节点名。保持现有的文件节点名，使用cvGetFileNodeName(节点).

node

被写入的节点。

embed

如果被写入的节点已经存在并且它的值不为 0，不建立额外的等级。 所有的节点元素被写入常用的输入结构 。图表元素只被写入图表，序列元素只被写入序列

函数 cvWriteFileNode将一个文件节点的拷贝写入文件存储器 可能请求的函数是： 将几个文件存储器合而为一。在XML 和YAML 之间变换格式等。

读取数据

从文件存储器中得到数据有两种方法：第一， 查找文件节点包括那些被请求的数据；其次 ，利用手动或者使用自定义read 方法取得数据。

GetRootFileNode

从文件存储器中得到一个高层节点

CvFileNode* cvGetRootFileNode(const CvFileStorage* fs, int stream_index=0);

fs

初始化文件存储器 。

stream_index

流的零基索引 。参考 cvStartNextStream. 在通常情况下，文件中只有一个流，但是可以拥有多个。

函数 cvGetRootFileNode 返回一个高层文件节点。 高层节点没有名称，它们和流相对应，接连存入文件存储器。如果超出索引， 函数返回NULL指针， 所以高层节点反复调用函数stream_index=0,1,...,直到返回NULL指针。这个函数可以在文件存储器中反复调用。

GetFileNodeByName

在图表或者文件存储器中查找节点

CvFileNode* cvGetFileNodeByName(const CvFileStorage* fs,
const CvFileNode* map,
const char* name);

fs

初始化文件存储器。

map

设置母图表。如果为 NULL，函数 在所有的高层节点（流）中检索， 从第一个开始。

name

设置文件节点名。

函数 cvGetFileNodeByName 文件节点通过name 查找文件节点 该节点在图表中被查找或者 ，如果指针为NULL，在内存中的高层文件节点中查找。 在图表中或者cvGetSeqElem序列中使用这个函数 。可能反复调用文件存储器i 加速确定某个多重表示值（例如 结构数组 ）可能在cvGetHashedKey 和cvGetFileNode之中用到一个。

GetHashedKey

返回一个指向已有名称的唯一指针

CvStringHashNode* cvGetHashedKey(CvFileStorage* fs, const char* name,
int len=-1, int create_missing=0);

`fs`

初始化文件存储器。

`name`

设置文字节点名。

`len`

名称（名称已知）的长度，如果值为-1 长度需要被计算出来 。

`create_missing`

标识符说明 ，是否应该将一个缺省节点的值加入哈希表。

函数 cvGetHashedKey返回指向每一个特殊文件节点名的特殊指针。这个指针通过cvGetFileNode 函数。它比cvGetFileNodeByName快。观察下面例子： 用二维图来表示一个点集，例：

```
%YAML:1.0
points:
  - { x: 10, y: 10 }
  - { x: 20, y: 20 }
  - { x: 30, y: 30 }
  # ...
```

因而,它使用哈希指针“x”和“y"加速对点的编译。

Example. Reading an array of structures from file storage

例：从一个文件存储器中读取顺序的结构

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvFileStorage* fs = cvOpenFileStorage( "points.yml", 0, CV_STORAGE_READ );
    CvStringHashNode* x_key = cvGetHashedNode( fs, "x", -1, 1 );
    CvStringHashNode* y_key = cvGetHashedNode( fs, "y", -1, 1 );
    CvFileNode* points = cvGetFileNodeByName( fs, 0, "points" );

    if( CV_NODE_IS_SEQ(points->tag) )
    {
        CvSeq* seq = points->data.seq;
```

```
            int i, total = seq->total;
            CvSeqReader reader;
            cvStartReadSeq( seq, &reader, 0 );
            for( i = 0; i < total; i++ )
            {
                CvFileNode* pt = (CvFileNode*)reader.ptr;
#if 1 /* 快变量 */
                CvFileNode* xnode = cvGetFileNode( fs, pt, x_key, 0 );
                CvFileNode* ynode = cvGetFileNode( fs, pt, y_key, 0 );
                assert( xnode && CV_NODE_IS_INT(xnode->tag) &&
                        ynode && CV_NODE_IS_INT(ynode->tag));
                int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
                int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#elif 1 /* 慢变量：不使用 x 值与 y 值 */
                CvFileNode* xnode = cvGetFileNodeByName( fs, pt,
"x" );
                CvFileNode* ynode = cvGetFileNodeByName( fs, pt,
"y" );
                assert( xnode && CV_NODE_IS_INT(xnode->tag) &&
                        ynode && CV_NODE_IS_INT(ynode->tag));
                int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
                int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#else /* 最慢的可以轻松使用的变量 */
                int x = cvReadIntByName( fs, pt, "x", 0 /* default value
*/ );
                int y = cvReadIntByName( fs, pt, "y", 0 /* default value
*/ );
#endif
                CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
                printf("%d: (%d, %d)\n", i, x, y );
            }
        }
        cvReleaseFileStorage( &fs );
        return 0;
}
```

请注意,无论使用那一种方法访问图表 ，都比使用序列慢，例如上面的例子，在单一数字序列中点像数字一样被编译。

GetFileNode

在图表或者文件存储器中查找节点

```
CvFileNode* cvGetFileNode( CvFileStorage* fs, CvFileNode* map,
                           const CvStringHashNode* key, int
create_missing=0 );
```

`fs`

初始化文件存储器 。

`map`

设置母图表。如果为 NULL，函数 在所有的高层节点（流）中检索，如果图表与值都为 NULLs,函数返回到根节点-图表包含高层节点

`key`

指向节点名的特殊节点 ，从cvGetHashedKey中得到 。

`create_missing`

标识符说明 ，是否应该将一个缺省节点加入图表。

函数 cvGetFileNode 查找一个文件节点。函数能够插入一个新的节点，当它不在图表中时

GetFileNodeName

返回文件节点名

```
const char* cvGetFileNodeName( const CvFileNode* node );
```

`node`

初始化文件节点 。

函数 cvGetFileNodeName 返回文件节点名或返回NULL(如果文件节点没有名称或者node为NULL 。

ReadInt

从文件节点中得到整形值

```
int cvReadInt( const CvFileNode* node, int default_value=0 );
```

`node`

初始化文件节点

`default_value`

如果 node 为 NULL，返回一个值。

函数 cvReadInt 从文件节点中返回整数。如果文件节点为NULL，default_value 被返回 。另外如果文件节点有类型 CV_NODE_INT，则 node->data.i 被返回 。如果文件节点有类型 CV_NODE_REAL，则 node->data.f 被修改成整数后返回。 另外一种情况是，结果不确定。

ReadIntByName

查找文件节点返回它的值

```
int cvReadIntByName( const CvFileStorage* fs, const CvFileNode*
map,
                     const char* name, int default_value=0 );
```

`fs`

初始化文件存储器 。

`map`

设置母图表。如果为 NULL，函数 在所有的高层节点（流）中检索。

`name`

设置节点名。

`default_value`

如果文件节点为 NULL，返回一个值。

函数 cvReadIntByName是 cvGetFileNodeByName 和 cvReadInt的简单重叠.

ReadReal

从文件节点中得到浮点形值

double cvReadReal(const CvFileNode* node, double default_value=0.);

`node`

初始化文件节点 。

`default_value`

如果 node 为 NULL，返回一个值。

函数cvReadReal 从文件节点中返回浮点形值。如果文件节点为 NULL， default_value 被返回 。另外如果文件节点有类型 CV_NODE_REAL ，则node->data.f 被返回 。如果文件节点有类型 CV_NODE_INT , 则 node->data.i 被修改成浮点数后返回。 另外一种情况是，结果不确定。 .

ReadRealByName

查找文件节点返回它的浮点形值

double cvReadRealByName(const CvFileStorage* fs, const CvFileNode* map, const char* name, double default_value=0.);

`fs`

初始化文件存储器 。

`map`

设置母图表。如果为 NULL，函数 在所有的高层节点（流）中检索。

`name`

设置节点名。

`default_value`

如果 node 为 NULL，返回一个值。

函数 cvReadRealByName 是 cvGetFileNodeByName 和 cvReadReal 的简单重叠。

ReadString

从文件节点中得到字符串文本

const char* cvReadString(const CvFileNode* node, const char* default_value=NULL);

`node`

初始化文件节点。

`default_value`

如果 node 为 NULL，返回一个值。

函数cvReadString 从文件节点中返回字符串文本。如果文件节点为 NULL， default_value 被返回 。另外如果文件节点有类型

CV_NODE_STR, 则data.str.ptr 被返回 。 另外一种情况是，结果不确定。

ReadStringByName

查找文件节点返回它的字符串文本

```
const char* cvReadStringByName( const CvFileStorage* fs, const
CvFileNode* map,
                                const char* name, const char*
default_value=NULL );
```

`fs`

初始化文件存储器 。

`map`

设置母图表。如果为 NULL，函数 在所有的高层节点（流）中检索。

`name`

设置节点名。

`default_value`

如果文件节点为 NULL，返回一个值。

函数 cvReadStringByName是 cvGetFileNodeByName 和 cvReadString 的简单重叠。

Read

解释对象并返回指向它的指针

```
void* cvRead( CvFileStorage* fs, CvFileNode* node,
              CvAttrList* attributes=NULL );
```

`fs`

初始化文件存储器。

`node`

设置对象根节点。

`attributes`

不被使用的参数.

函数 cvRead 解释用户对象 (在文件存储器子树中建立新的对象）并返回。对象被解释 ，必须按原有的支持读方法的类型 (参考 CvTypeInfo).用类型名决定对象，并在文件中被解释 。如果对象是动态结构，它将在内存中通过cvOpenFileStorage或者使NULL指针被建立。在临时性内存中。当cvReleaseFileStorage 被调用时释放内存。如果对象不是动态结构 ,将在堆中被建立用专用函数或通用函数 cvRelease. 释放内存。

ReadByName

查找对象并解释

```
void* cvReadByName( CvFileStorage* fs, const CvFileNode* map,
                    const char* name, CvAttrList*
attributes=NULL );
```

`fs`

初始化文件存储器。

`map`

设置双亲节点。如果它为 NULL，函数从高层节点中查找。

`name`

设置节点名称。

`attributes`

不被使用的参数.

函数 cvReadByName 是由cvGetFileNodeByName 和 cvRead叠合的。 .

ReadRawData

读重数

void cvReadRawData(const CvFileStorage* fs, const CvFileNode* src,
void* dst, const char* dt);

`fs`

初始化文件存储器 。

`src`

设置文件节点（有序的）来读数。

`dst`

设置指向目的数组的指针。

`dt`

数组元素的说明。格式参考 cvWriteRawData。

函数 cvReadRawData从有序的文件节点中读取元素。

StartReadRawData

初始化文件节点读取器

void cvStartReadRawData(const CvFileStorage* fs, const CvFileNode* src,
CvSeqReader* reader);

`fs`

初始化文件存储器。

`src` 设置文件节点(有序的）来读数。

-

reader 设置顺序读取指针。

函数 cvStartReadRawData 初始化顺序读取器从文件节点中读取数据。能够通过cvReadRawDataSlice.初始化文件读取器 。

ReadRawDataSlice

初始化文件节点按序读取器

void cvReadRawDataSlice(const CvFileStorage* fs, CvSeqReader* reader,

int count, void* dst, const char* dt);

`fs`

初始化文件存储器。

`reader`

设置按序读取器 . 用 cvStartReadRawData.初始化。

`count`

被读取元素的数量。

`dst`

指向目的数组的指针。

`dt`

数组元素的说明。格式参考 cvWriteRawData。

函数 cvReadRawDataSlice 从文件节点读一个或多个元素，组成一个序列用于指定数组。读入元素的总数由其他数组的元素总和构成的。例如 如果 dt='2if', 函数将读是总数三倍的序列元素。

运行时类型信息和通用函数

CvTypeInfo

类型信息

```
typedef int (CV_CDECL *CvIsInstanceFunc)( const void* struct_ptr );
typedef void (CV_CDECL *CvReleaseFunc)( void** struct_dblptr );
typedef void* (CV_CDECL *CvReadFunc)( CvFileStorage* storage,
CvFileNode* node );
typedef void (CV_CDECL *CvWriteFunc)( CvFileStorage* storage,
                                      const char* name,
                                      const void* struct_ptr,
                                      CvAttrList attributes );
typedef void* (CV_CDECL *CvCloneFunc)( const void* struct_ptr );

typedef struct CvTypeInfo
{
    int flags; /* 不常用 */
    int header_size; /*  (CvTypeInfo)的大小 */
    struct CvTypeInfo* prev; /* 在列表中已定义过的类型 */
    struct CvTypeInfo* next; /* 在列表中下一个已定义过的类型 */
    const char* type_name; /*定义类型名，并写入文件存储器 */

    /* methods */
    CvIsInstanceFunc is_instance; /* 选择被传递的对象属于的类型 */
    CvReleaseFunc release; /* 释放对象的内存空间 */
    CvReadFunc read; /* 从文件存储器中读对象 */
    CvWriteFunc write; /* 将对象写入文件存储器 */
    CvCloneFunc clone; /* 复制一个对象 */
}
```

CvTypeInfo;
结构 CvTypeInfo包含的信息包括标准的或用户自定义的类型。类型有没有包含指向相应的CvTypeInfo结构的指针。在已有的对象中查找类型的方法是使用cvTypeOf函数。已有的信息可以通过类型名使用cvFindType来查找，这个方法在从文件存储器中读对象的时候被使用。 用户可以通过cvRegisterType定义一个新的类型 ，并将类型信息结构加到文件列表的开始端， 它可以从标准类型中建立专门的类型，不必考虑基本的方法。

RegisterType

定义新类型

void cvRegisterType(const CvTypeInfo* info);

`info`

类型信息结构。

函数 cvRegisterType 定义一个新类型,可以通过信息来描述它。

UnregisterType

删除定义的类型

void cvUnregisterType(const char* type_name);

`type_name`

被删除的类型的名称。

函数 cvUnregisterType通过指定的名称删除已定义的类型。 如果不知道类型名,可以用cvTypeOf或者连续扫描类型列表，从cvFirstType开始,然后调用 cvUnregisterType(info->type_name)。

FirstType

返回类型列表的首位。

CvTypeInfo* cvFirstType(void);

函数 cvFirstType 返回类型列表中的第一个类型。

FindType

通过类型名查找类型

CvTypeInfo* cvFindType(const char* type_name);

`type_name`

类型名

函数 cvFindType通过类型名查找指定的类型。如果找不到返回值为NULL。

TypeOf

返回对象的类型

CvTypeInfo* cvTypeOf(const void* struct_ptr);

`struct_ptr`

定义对象指针。

函数 cvTypeOf 查找指定对象的类型。它反复扫描类型列表，调用每一个类型信息结构中的函数和方法与对象做比较，直到它们中的一个的返回值不为 0 或者所有的类型都被访问。

Release

删除对象

void cvRelease(void** struct_ptr);

`struct_ptr`

定义指向对象的双指针。

函数 cvRelease 查找指定对象的类型，然后调用release。

Clone

克隆一个对象

void* cvClone(const void* struct_ptr);

`struct_ptr`

定义被克隆的对象

函数 cvClone 查找指定对象的类型，然后调用 clone。

Save

存储对象到文件中

void cvSave(const char* filename, const void* struct_ptr,
const char* name=NULL,
const char* comment=NULL,
CvAttrList attributes=cvAttrList());

`filename`

初始化文件名。

`struct_ptr`

指定要存储的对象。

`name`

可选择的对象名 。如果为 NULL, 对象名将从filename中列出。

`comment`

可选注释。加在文件的开始处。

-

attributes 可选属性。 通过 cvWrite。

函数 cvSave存储对象到文件。它给cvWrite提供一个简单的界面。

Load

从文件中打开对象。

void* cvLoad(const char* filename, CvMemStorage* memstorage=NULL,
const char* name=NULL, const char**
real_name=NULL);

`filename`

初始化文件名
`memstorage`
动态结构的内存，例如CvSeq或CvGraph。不能作用于矩阵或图像。
`name`
可选对象名。如果为 NULL,内存中的第一个高层对象被打开。
`real_name`
可选输出参数 。它包括已打开的对象的名称 (如果 name=NULL时有效)。
函数 cvLoad 从文件中打开对象。它给cvRead提供一个简单的界面.对象被打开之后，文件存储器被关闭，所有的临时缓冲区被删除。因而，打开一个动态结构，如序列，轮廓或图像。 它应该通过一个有效的目的文件内存发挥作用。

其它混合函数

CheckArr

检查输入数组的每一个元素是否是非法值

```
int    cvCheckArr( const CvArr* arr, int flags=0,
                   double min_val=0, double max_val=0);
#define cvCheckArray cvCheckArr
```

`arr`
待检查数组
`flags`
操作标志, 0 或者下面值的组合:
CV_CHECK_RANGE - 如果设置这个标志, 函数检查数组的每一个值是否在范围 [minVal,maxVal) 以内,如果不在范围内它不检查每一个元素是否是 NaN 或者 ±Infinity。
CV_CHECK_QUIET - 如果设置这个标志, 如果一个元素是非法的或者越界的，函数 不会产生一个错误。
`min_val`
有效值范围的闭下边界。只有当 CV_CHECK_RANGE 被设置的时候它才有作用。
`max_val`
有效值范围的开上边界。只有当 CV_CHECK_RANGE 被设置的时候它才有作用。
函数 cvCheckArr 检查每一个数组元素不是 NaN 也不是 ±Infinity 。如果 CV_CHECK_RANGE 被设定, 它也检查每一个元素是大于等于 minVal 并且小于maxVal。如果检查成功函数返回非零值，例如，所有元素都是合法的并且在范围内，如果检查失败则返回 0 。 在后一种情况下如果 CV_CHECK_QUIET 标志没有被设定， 函数报出运行错误。

KMeans2

按照给定的群的数目拆分向量的集合（Splits set of vectors by given number of clusters）

```
void cvKMeans2( const CvArr* samples, int cluster_count,
                CvArr* labels, CvTermCriteria termcrit );
```

samples
输入样例的浮点矩阵，每个样例一行。
cluster_count
拆分集合的群的数目
labels
输出整数向量，为每个样例存储群索引
termcrit
指定最大迭代次数和/或精度(distance the centers move by between the subsequent iterations).
函数 cvKMeans2 执行 k-means 算法 查找 cluster_count 群的中心并沿着群分组输入样例，输出 labels(i) 包含样例的群索引，存储在样例矩阵的第 i 行。

例子. 带 k-means 的高斯分布的随机群样例（Clustering random samples of multi-gaussian distribution with k-means）

```
#include "cxcore.h"
#include "highgui.h"

void main( int argc, char** argv )
{
    #define MAX_CLUSTERS 5
    CvScalar color_tab[MAX_CLUSTERS];
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG rng = cvRNG(0xffffffff);

    color_tab[0] = CV_RGB(255,0,0);
    color_tab[1] = CV_RGB(0,255,0);
    color_tab[2] = CV_RGB(100,100,255);
    color_tab[3] = CV_RGB(255,0,255);
    color_tab[4] = CV_RGB(255,255,0);

    cvNamedWindow( "clusters", 1 );

    for(;;)
    {
        int k, cluster_count = cvRandInt(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandInt(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

        /* generate random sample from multigaussian distribution */
        for( k = 0; k < cluster_count; k++ )
        {
            CvPoint center;
            CvMat point_chunk;
            center.x = cvRandInt(&rng)%img->width;
            center.y = cvRandInt(&rng)%img->height;
```

```
                cvGetRows( points, &point_chunk,
k*sample_count/cluster_count,
                        k == cluster_count - 1 ? sample_count :
(k+1)*sample_count/cluster_count );
                cvRandArr( &rng, &point_chunk, CV_RAND_NORMAL,
                        cvScalar(center.x,center.y,0,0),
                        cvScalar(img->width/6, img->height/6,0,0) );
            }

            /* shuffle samples */
            for( i = 0; i < sample_count/2; i++ )
            {
                CvPoint2D32f* pt1 = (CvPoint2D32f*)points->data.fl +
cvRandInt(&rng)%sample_count;
                CvPoint2D32f* pt2 = (CvPoint2D32f*)points->data.fl +
cvRandInt(&rng)%sample_count;
                CvPoint2D32f temp;
                CV_SWAP( *pt1, *pt2, temp );
            }

            cvKMeans2( points, cluster_count, clusters,

cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

            cvZero( img );

            for( i = 0; i < sample_count; i++ )
            {
                CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
                int cluster_idx = clusters->data.i[i];
                cvCircle( img, cvPointFrom32f(pt), 2, color_tab[cluster_idx],
CV_FILLED );
            }

            cvReleaseMat( &points );
            cvReleaseMat( &clusters );

            cvShowImage( "clusters", img );

            int key = cvWaitKey(0);
            if( key == 27 ) // 'ESC'
                break;
        }
}
```

SeqPartition

Splits sequence into equivalency classes 拆分序列为等效的类

typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);

int cvSeqPartition(const CvSeq* seq, CvMemStorage* storage, CvSeq** labels,

CvCmpFunc is_equal, void* userdata);

`seq`

划分序列

`storage`

存储序列的等效类的存储器，如果为空（NULL），函数用 seq->storage 存储输出标签 labels

`labels`

输出参数。双重指向基于 0 标签的输入序列元素的序列。

`is_equal`

如果两个特殊元素是来自同一个类这个关系函数返回非零值，否则返回 0 。划分算法用关系函数的传递闭包得到等价类。

`userdata`

传递给 is_equal 函数的透明指针。

函数 cvSeqPartition 执行二次方程算法为拆分集合为一个或者更多的等效类。 函数返回等效等效类的数目。

Example. Partitioning 2d point set.

```
#include "cxcore.h"
#include "highgui.h"
#include <stdio.h>

CvSeq* point_seq = 0;
IplImage* canvas = 0;
CvScalar* colors = 0;
int pos = 10;

int is_equal( const void* _a, const void* _b, void* userdata )
{
    CvPoint a = *(const CvPoint*)_a;
    CvPoint b = *(const CvPoint*)_b;
    double threshold = *(double*)userdata;
    return (double)(a.x - b.x)*(a.x - b.x) + (double)(a.y - b.y)*(a.y - b.y) <=
threshold;
}

void on_track( int pos )
{
    CvSeq* labels = 0;
    double threshold = pos*pos;
    int i, class_count = cvSeqPartition( point_seq, 0, &labels, is_equal,
&threshold );
    printf("%4d classes\n", class_count );
```

```
    cvZero( canvas );

    for( i = 0; i < labels->total; i++ )
    {
        CvPoint pt = *(CvPoint*)cvGetSeqElem( point_seq, i, 0 );
        CvScalar color = colors[*(int*)cvGetSeqElem( labels, i, 0 )];
        cvCircle( canvas, pt, 1, color, -1 );
    }

    cvShowImage( "points", canvas );
}

int main( int argc, char** argv )
{
    CvMemStorage* storage = cvCreateMemStorage(0);
    point_seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq),
sizeof(CvPoint), storage );
    CvRNG rng = cvRNG(0xffffffff);

    int width = 500, height = 500;
    int i, count = 1000;
    canvas = cvCreateImage( cvSize(width,height), 8, 3 );

    colors = (CvScalar*)cvAlloc( count*sizeof(colors[0]) );
    for( i = 0; i < count; i++ )
    {
        CvPoint pt;
        int icolor;
        pt.x = cvRandInt( &rng ) % width;
        pt.y = cvRandInt( &rng ) % height;
        cvSeqPush( point_seq, &pt );
        icolor = cvRandInt( &rng ) | 0x00404040;
        colors[i] = CV_RGB(icolor & 255, (icolor >> 8)&255, (icolor >>
16)&255);
    }

    cvNamedWindow( "points", 1 );
    cvCreateTrackbar( "threshold", "points", &pos, 50, on_track );
    on_track(pos);
    cvWaitKey(0);
    return 0;
}
```

错误处理和系统函数

错误处理

在 OpenCV 中错误处理和 IPL (Image Processing Library)很相似。假如错误处理函数不返回错误代码，而是用CV_ERROR 宏调用 cvError 函数报错，按次序地，用 cvSetErrStatus 函数设置错误状态，然后调用标准的或者用户自定义的错误处理器(它可以显示一个消息对话框，写出错误日志等等, 参考函数 cvRedirectError, cvNulDevReport, cvStdErrReport, cvGuiBoxReport)。每个程序的线程都有一个全局变量，它包含了错误状态(一个整数值)。这个状态可以被 cvGetErrStatus 函数检索到。
有三个错误处理模式(参考 cvSetErrMode 和 cvGetErrMode):
`Leaf`
错误处理器被调用以后程序被终止。这是缺省值。它在调试中是很有用的，当错误发生的时候立即产生错误信息。然而对于产生式系统后面两种方法可能会为他们提供更多的控制方式。
`Parent`
错误处理器被调用以后程序不会被终止。栈被清空 (它用 C++ 异常处理机制完成写/输出--w/o)。当调用 CxCore 的函数 cvGetErrStatus 起作用以后用户可以检查错误代码。
`Silent`
和 *Parent* 模式相似, 但是没有错误处理器被调用。
事实上, *Leaf* 和 *Parent* 模式的语义被错误处理器执行，上面的描述对 cvNulDevReport, cvStdErrReport. cvGuiBoxReport 的行为有一些细微的差别，一些自定义的错误处理器可能语义上会有很大的不同。

错误处理宏

报错，检查错误等的宏

```
/* special macros for enclosing processing statements within a function and separating
    them from prologue (resource initialization) and epilogue (guaranteed resource release) */
#define __BEGIN__           {
#define __END__             goto exit; exit: ; }
/* proceeds to "resource release" stage */
#define EXIT                goto exit

/* Declares locally 函数 name for CV_ERROR() use */
#define CV_FUNCNAME( Name )   \
      static char cvFuncName[] = Name

/* Raises an error within the current context */
#define CV_ERROR( Code, Msg )
\
{
    \
       cvError( (Code), cvFuncName, Msg, __FILE__, __LINE__ );
\
```

```
        EXIT;
\
}

/* Checks status after calling CXCORE function */
#define CV_CHECK()
\
{
    \
      if( cvGetErrStatus() < 0 )                                                \
            CV_ERROR( CV_StsBackTrace, "Inner function failed." );
\
}

/* Provies shorthand for CXCORE function call and CV_CHECK() */
#define CV_CALL( Statement )
\
{
    \
      Statement;
\
      CV_CHECK();
\
}

/* Checks some condition in both debug and release configurations */
#define CV_ASSERT( Condition )
\
{
          \
      if( !(Condition) )
\
            CV_ERROR( CV_StsInternal, "Assertion: " #Condition " failed" );
\
}

/* these macros are similar to their CV_... counterparts, but they
    do not need exit label nor cvFuncName to be defined */
#define OPENCV_ERROR(status,func_name,err_msg) ...
#define OPENCV_ERRCHK(func_name,err_msg) ...
#define OPENCV_ASSERT(condition,func_name,err_msg) ...
#define OPENCV_CALL(statement) ...
```

取代上面的讨论，这里有典型的 CXCORE 函数和这些函数使用的样例。

错误处理宏的使用

```
#include "cxcore.h"
#include <stdio.h>
```

```
void cvResizeDCT( CvMat* input_array, CvMat* output_array )
{
    CvMat* temp_array = 0; // declare pointer that should be released
anyway.

    CV_FUNCNAME( "cvResizeDCT" ); // declare cvFuncName

    __BEGIN__; // start processing. There may be some declarations just
after this macro,
                    // but they couldn't be accessed from the epilogue.

    if( !CV_IS_MAT(input_array) || !CV_IS_MAT(output_array) )
        // use CV_ERROR() to raise an error
        CV_ERROR( CV_StsBadArg, "input_array or output_array are
not valid matrices" );

    // some restrictions that are going to be removed later, may be
checked with CV_ASSERT()
    CV_ASSERT( input_array->rows == 1 && output_array->rows == 1 );

    // use CV_CALL for safe function call
    CV_CALL( temp_array = cvCreateMat( input_array->rows,
MAX(input_array->cols,output_array->cols),
                                       input_array->type ));

    if( output_array->cols > input_array->cols )
        CV_CALL( cvZero( temp_array ));

    temp_array->cols = input_array->cols;
    CV_CALL( cvDCT( input_array, temp_array,
CV_DXT_FORWARD ));
    temp_array->cols = output_array->cols;
    CV_CALL( cvDCT( temp_array, output_array, CV_DXT_INVERSE ));
    CV_CALL( cvScale( output_array, output_array,
1./sqrt((double)input_array->cols*output_array->cols), 0 ));

    __END__; // finish processing. Epilogue follows after the macro.

    // release temp_array. If temp_array has not been allocated before an
error occured, cvReleaseMat
    // takes care of it and does nothing in this case.
    cvReleaseMat( &temp_array );
}


int main( int argc, char** argv )
```

```
{
    CvMat* src = cvCreateMat( 1, 512, CV_32F );
#if 1 /* no errors */
    CvMat* dst = cvCreateMat( 1, 256, CV_32F );
#else
    CvMat* dst = 0; /* test error processing mechanism */
#endif
    cvSet( src, cvRealScalar(1.), 0 );
#if 0 /* change 0 to 1 to suppress error handler invocation */
    cvSetErrMode( CV_ErrModeSilent );
#endif
    cvResizeDCT( src, dst ); // if some error occurs, the message box will
popup, or a message will be
                                    // written to log, or some user-defined
processing will be done
    if( cvGetErrStatus() < 0 )
        printf("Some error occured" );
    else
        printf("Everything is OK" );
    return 0;
}
```

GetErrStatus

返回当前错误状态

int cvGetErrStatus(void);

函数 cvGetErrStatus 返回当前错误状态 - 这个状态是被上一步调用的 cvSetErrStatus 设置的。 注意，在 *Leaf* 模式下错误一旦发生程序立即被终止 ，因此对于总是需要调用函数后蔡获得控制的应用，可以调用 cvSetErrMode 函数将错误模式设置为 *Parent* 或 *Silent* 。

SetErrStatus

设置错误状态

void cvSetErrStatus(int status);

`status`

错误状态

函数 cvSetErrStatus 设置错误状态为指定的值。 大多数情况下，该函数被用来重设错误状态(设置为 CV_StsOk) 以从错误中恢复。在其他情况下调用 cvError 或 CV_ERROR 更自然一些。

GetErrMode

返回当前错误模式

int cvGetErrMode(void);

函数 cvGetErrMode 返回当前错误模式 - 这个值是在之前被 cvSetErrMode 函数设定的。

SetErrMode

设置当前错误模式

```
#define CV_ErrModeLeaf      0
#define CV_ErrModeParent    1
#define CV_ErrModeSilent    2
int cvSetErrMode( int mode );
```

`mode`

错误模式

函数 cvSetErrMode 设置指定的错误模式。关于不同的错误模式的讨论参考本节开始.

Error

产生一个错误

```
int cvError( int status, const char* func_name,
                const char* err_msg, const char* file_name, int line );
```

`status`

错误状态

`func_name`

产生错误的函数名

`err_msg`

关于错误的额外诊断信息

`file_name`

产生错误的文件名

`line`

产生错误的行号

函数 cvError 设置错误状态为指定的值(通过 cvSetErrStatus) ，如果错误模式不是 *Silent*，调用错误处理器。

ErrorStr

返回错误状态编码的原文描述

```
const char* cvErrorStr( int status );
```

`status`

错误状态

函数 cvErrorStr 返回指定错误状态编码的原文描述。如果是步知道的状态该函数返回空 （NULL）指针。

RedirectError

设置一个新的错误处理器

```
typedef int (CV_CDECL *CvErrorCallback)( int status, const char*
func_name,
                                const char* err_msg, const char* file_name, int
line );

CvErrorCallback cvRedirectError( CvErrorCallback error_handler,
```

void* userdata=NULL, void**
prev_userdata=NULL);

`error_handler`
新的错误处理器

`userdata`
传给错误处理器的任意透明指针

`prev_userdata`
指向前面分配给用户数据的指针

函数 cvRedirectError 在标准错误处理器或者有确定借口的自定义错误处理器中选择一个新的错误处理器 。错误处理器和 cvError 函数有相同的参数。 如果错误处理器返回非零的值，程序终止，否则，程序继续运行。错误处理器通过 cvGetErrMode 检查当前错误模式而作出决定。

---

cvNulDevReport cvStdErrReport cvGuiBoxReport

提供标准错误操作

```
int cvNulDevReport( int status, const char* func_name,
                    const char* err_msg, const char* file_name,
                    int line, void* userdata );

int cvStdErrReport( int status, const char* func_name,
                    const char* err_msg, const char* file_name,
                    int line, void* userdata );

int cvGuiBoxReport( int status, const char* func_name,
                    const char* err_msg, const char* file_name,
                    int line, void* userdata );
```

`status`
错误状态

`func_name`
产生错误的函数名

`err_msg`
关于错误的额外诊断信息

`file_name`
产生错误的文件名

`line`
产生错误的行号

`userdata`
指向用户数据的指针，被标准错误操作忽略。

函数 cvNullDevReport, cvStdErrReport, cvGuiBoxReport 提供标准错误操作。cvGuiBoxReport 是 Win32 系统缺省的错误处理器，cvStdErrReport - 其他系统. cvGuiBoxReport 弹出错误描述的消息框并提供几个选择。 下面是一个消息框的例子，如果和例子中的错误描述相同，它和上面的例子代码可能是兼容的。

错误消息对话框

如果错误处理器是 cvStdErrReport, 上面的消息将被打印到标准错误输出，程序将要终止和继续依赖于当前错误模式。

错误消息打印到标准错误输出 (在 *Leaf* 模式)

OpenCV ERROR: Bad argument (input_array or output_array are not valid matrices)
in function cvResizeDCT,
D:\User\VP\Projects\avl_proba\a.cpp(75)
Terminating the application...

系统函数

Alloc

分配内存缓冲区

void* cvAlloc(size_t size);

`size`

以字节为单位的缓冲区大小

函数 cvAlloc 分配字节缓冲区大小并返回分配的缓冲区的指针。如果错误处理函数产生了一个错误报告 则返回一个空（NULL）指针。 缺省地 cvAlloc 调用 icvAlloc 而 icvAlloc 调用 malloc ，然而用 cvSetMemoryManager 调用用户自定义的内存分配和释放函数也是可能的。

Free

释放内存缓冲区

void cvFree(void** ptr);

`buffer`

指向被释放的缓冲区的双重指针

函数 cvFree 释放被 cvAlloc 分配的缓冲区。在退出的时候它清除缓冲区指针，这就是为什么要使用双重指针的原因 。 如果 *buffer 已经是空（NULL），函数什么也不做。

GetTickCount

Returns number of tics

int64 cvGetTickCount(void);

函数 cvGetTickCount 返回从依赖于平台的事件(从启动开始 CPU 的 ticks 数目，从 1970 年开始的微秒数目等等)开始的 tics 的数目 。 该函数对于精确测量函数/用户代码的执行时间是很有用的。要转化 tics 的数目为时间单位，使用函数 cvGetTickFrequency 。

GetTickFrequency

*返回每个微秒的 **tics** 的数目*

double cvGetTickFrequency(void);

函数 cvGetTickFrequency 返回每个微秒的 tics 的数目。 因此，cvGetTickCount() 和 cvGetTickFrequency() 将给出从依赖于平台的事件开始的 tics 的数目 。

RegisterModule

***Registers another module** 注册另外的模块*

```
typedef struct CvPluginFuncInfo
{
    void** func_addr;
    void* default_func_addr;
    const char* func_names;
    int search_modules;
    int loaded_from;
}
CvPluginFuncInfo;

typedef struct CvModuleInfo
{
    struct CvModuleInfo* next;
    const char* name;
    const char* version;
    CvPluginFuncInfo* func_tab;
}
CvModuleInfo;

int cvRegisterModule( const CvModuleInfo* module_info );
```

`module_info`

模块信息

函数 cvRegisterModule 添加模块到已注册模块列表中。模块被注册后，用 cvGetModuleInfo 函数可以检索到它的信息。注册模块可以通过 CXCORE 的 支持利用优化插件 (IPP, MKL, ...)。 CXCORE ， CV (computer vision), CVAUX (auxilary computer vision) 和 HIGHGUI (visualization & image/video acquisition) 自身就是模块的例子。 通常注册后共享库就被载入。参考 cxcore/src/cxswitcher.cpp and cv/src/cvswitcher.cpp 获取细节

信息，怎样注册的参考 cxcore/src/cxswitcher.cpp ， cxcore/src/_cxipp.h 显示了 IPP 和 MKL 是怎样连接到模块的。

GetModuleInfo

检索注册模块和插件的信息

```
void    cvGetModuleInfo( const char* module_name,
                         const char** version,
                         const char** loaded_addon_plugins );
```

`module_name`

模块名，或者 NULL ，则代表所有的模块

`version`

输出参数，模块的信息，包括版本信息

`loaded_addon_plugins`

优化插件的名字和版本列表，这里 CXCORE 可以被找到和载入

函数 cvGetModuleInfo 返回一个或者所有注册模块的信息。返回信息被存储到库当中，因此，用户不用释放或者修改返回的文本字符。

UseOptimized

在优化/不优化两个模式之间切换

```
int cvUseOptimized( int on_off );
```

`on_off`

优化(<>0) 或者 不优化 (0).

函数 cvUseOptimized 在两个模式之间切换,这里只有纯 C 才从 cxcore, OpenCV 等执行。如果可用 IPP 和 MKL 函数也可使用。 当 cvUseOptimized(0) 被调用，所有的优化库都不被载入。该函数在调试模式下是很有用的， IPP&MKL 不工作，在线跨速比较等。它返回载入的优化函数的数目。注意，缺省地优化插件是被载入的，因此在程序开始调用 cvUseOptimized(1) 是没有必要的(事实上，它只会增加启动时间)

SetMemoryManager

分配自定义/缺省内存管理函数

```
typedef void* (CV_CDECL *CvAllocFunc)(size_t size, void* userdata);
typedef int (CV_CDECL *CvFreeFunc)(void* pptr, void* userdata);

void cvSetMemoryManager( CvAllocFunc alloc_func=NULL,
                         CvFreeFunc free_func=NULL,
                         void* userdata=NULL );
```

`alloc_func`

分配函数; 除了 userdata 可能用来确定上下文关系外，接口和 malloc 相似

`free_func`

释放函数; 接口和 free 相似

`userdata`

透明的传给自定义函数的用户数据

函数 cvSetMemoryManager 设置将被 cvAlloc，cvFree 和高级函数 (例如. cvCreateImage) 调用的用户自定义内存管理函数(代替 malloc 和 free)。 注意，当用 cvAlloc 分配数据的时候该函数被调用。 当然，为了避免无限循环调用， 它不允许从自定义分配/释放函数调用 cvAlloc 和 cvFree 。
如果 alloc_func 和 free_func 指针是 NULL，恢复缺省的内存管理函数。

SetIPLAllocators

*切换图像 **IPL** 函数的分配/释放*

```
typedef IplImage* (CV_STDCALL* Cv_iplCreateImageHeader)
                                    (int,int,int,char*,char*,int,int,int,int,int,
                                    IplROI*,IplImage*,void*,IplTileInfo*);
typedef void (CV_STDCALL* Cv_iplAllocateImageData)(IplImage*,int,int);
typedef void (CV_STDCALL* Cv_iplDeallocate)(IplImage*,int);
typedef IplROI* (CV_STDCALL* Cv_iplCreateROI)(int,int,int,int,int);
typedef IplImage* (CV_STDCALL* Cv_iplCloneImage)(const IplImage*);

void cvSetIPLAllocators( Cv_iplCreateImageHeader create_header,
                         Cv_iplAllocateImageData allocate_data,
                         Cv_iplDeallocate deallocate,
                         Cv_iplCreateROI create_roi,
                         Cv_iplCloneImage clone_image );

#define CV_TURN_ON_IPL_COMPATIBILITY()
\
    cvSetIPLAllocators( iplCreateImageHeader, iplAllocateImage,
\
                        iplDeallocate, iplCreateROI, iplCloneImage )
```

`create_header`
指向 iplCreateImageHeader 的指针
`allocate_data`
指向 iplAllocateImage 的指针
`deallocate`
指向 iplDeallocate 的指针
`create_roi`
指向 iplCreateROI 的指针
`clone_image`
指向 iplCloneImage 的指针

函数 cvSetIPLAllocators 使用 CXCORE 来进行图像 IPL 函数的 分配/释放 操作。 为了方便，这里提供了环绕宏 CV_TURN_ON_IPL_COMPATIBILITY。 当 IPL 和 CXCORE/OpenCV 同时使用以及调用 iplCreateImageHeader 等情况该函数很有用。如果 IPL 仅仅是被调用来进行数据处理，该函数就必要了，因为所有的分配/释放都由 CXCORE 来完成，或者所有的分配/释放都由 IPL 和一些 OpenCV 函数来处理数据。

依字母顺序函数列表

A

AbsDiff
AbsDiffS
Add
AddS
AddWeighted
Alloc
And
AndS
Avg
AvgSdv

C

CalcCovarMatrix
CartToPolar
Cbrt
CheckArr
Circle
Clear*D
ClearGraph
ClearMemStorage
ClearSeq
ClearSet
Clone
CloneGraph
CloneImage
CloneMat
CloneMatND
CloneSeq
CloneSparseMat
Cmp
CmpS
ConvertScale
ConvertScaleAbs
Copy
CountNonZero
CreateChildMemStorage
CreateData
CreateGraph
CreateImage
CreateImageHeader
CreateMat
CreateMatHeader
CreateMatND
CreateMatNDHeader
CreateMemStorage
CreateSeq
CreateSet
CreateSparseMat
CrossProduct
CvtSeqToArray

D

DCT
DFT
DecRefData
Det
Div
DotProduct
DrawContours

E

EigenVV
Ellipse
EndWriteSeq
EndWriteStruct
Error
ErrorStr
Exp

F

FastArctan
FillConvexPoly
FindGraphEdgeByPtr
FindType
FlushSeqWriter
Free

G

I

K

L

M

N

O

P

R

S

SeqInsert
SeqInsertSlice
SeqInvert
SeqPartition
SeqPop
SeqPopFront
SeqPopMulti
SeqPush
SeqPushFront
SeqPushMulti
SeqRemove
SeqRemoveSlice
SeqSearch
SetErrMode
SetErrStatus
SetIPLAllocators
SetIdentity
SetImageCOI
SetImageROI
SetMemoryManager
SetNew
SetReal*D
SetRemove
SetRemoveByPtr
SetSeqBlockSize
SetSeqReaderPos
StartReadRawData
StartReadSeq
StartScanGraph
StartWriteSeq
StartWriteStruct
StdErrReport
Sub
SubRS
SubS
Sum
Set

T

Trace
Transform
Transpose
TreeToNodeSeq
TypeOf

U

UnregisterType
UseOptimized

W

Write
WriteComment
WriteFileNode
WriteInt
WriteRawData
WriteReal
WriteString

X

Xor
XorS

例子列表

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。

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：

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

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 $\min(\lambda_1, \lambda_2)$

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`

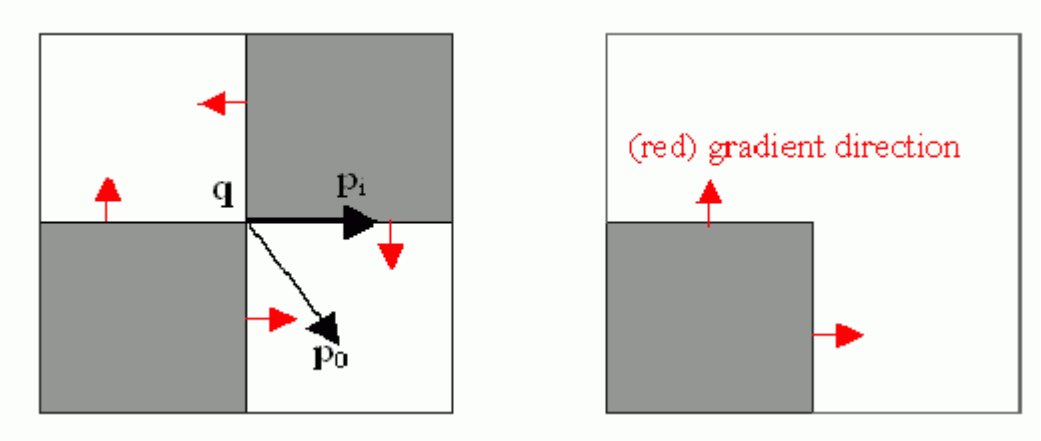
死区的一半尺寸，死区为不对搜索区的中央位置做求和运算的区域。它是用来避免自相关矩阵出现的某些可能的奇异性。当值为 (-1,-1) 表示没有死区。

`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：

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

其中，DI_{pi}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：

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

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) ：8一连通情况下，或者 |pt2.x-pt1.x|+|pt2.y-pt1.y|+1 ： 4一连通情况下.

`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

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

其中在非整数坐标 $A\bullet(x,y)^T+b$ 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',y')<-src(x,y)
如果没有指定 CV_WARP_INVERSE_MAP ，
$(x',y')^T$=map_matrix•$(x,y,1)^T$+b ，
否则， $(x, y)^T$=map_matrix•$(x',y&apos,1)^T$+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:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center.x - \beta\cdot center.y \\ -\beta & \alpha & \beta\cdot center.x + (1-\alpha)\cdot center.y \end{bmatrix}$$

where $\alpha=scale\cdot\cos(angle)$, $\beta=scale\cdot\sin(angle)$
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，

$(tx',ty',t)^T=map\_matrix \cdot (x,y,1)^T+b$

否则， $(tx, ty, t)^T=map\_matrix \cdot (x',y',1)^T+b$

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：

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

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

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 O(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

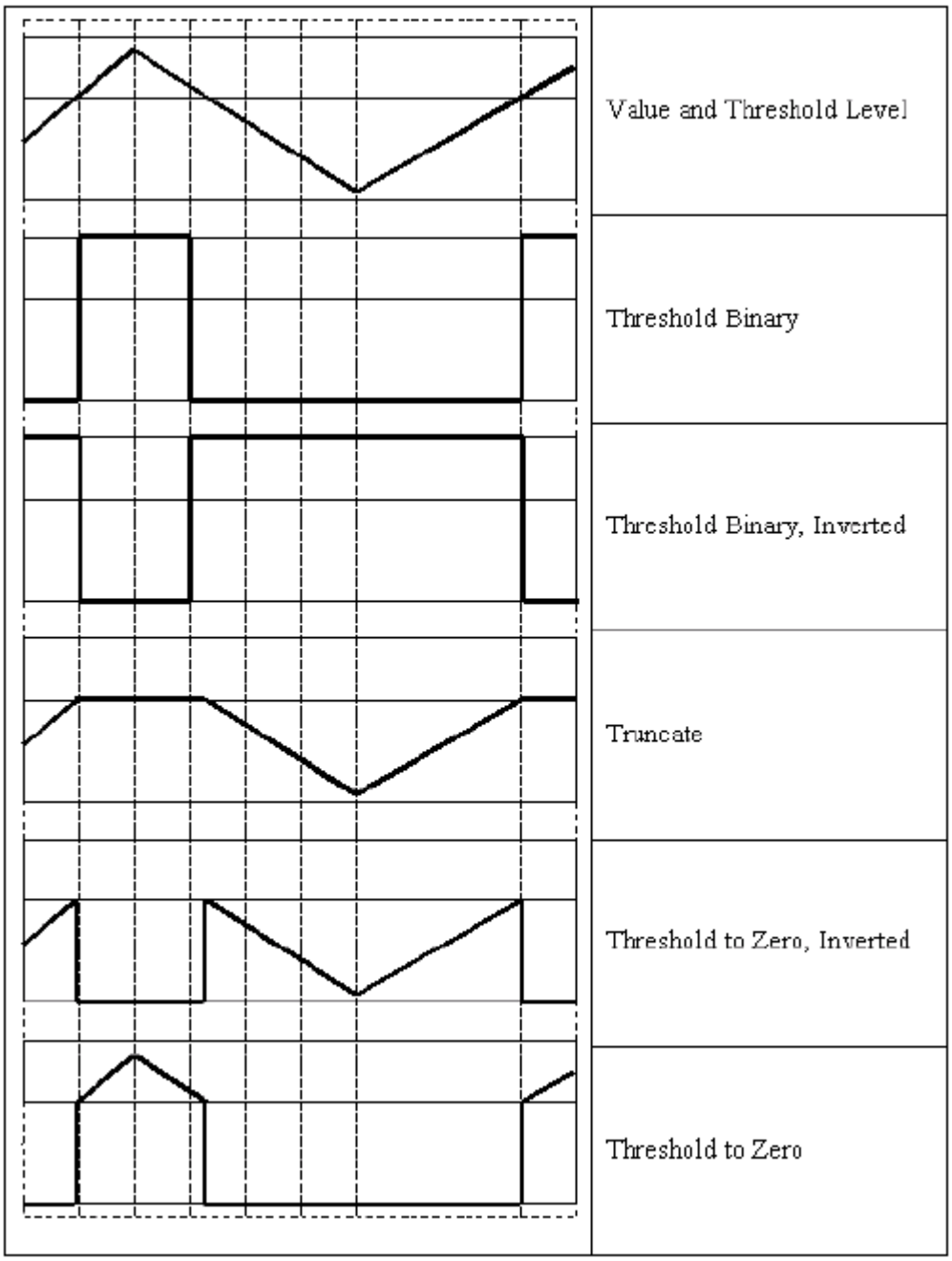
threshold_type=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>threshold
max_value, otherwise

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_l 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`
卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5

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

PyrUp

图像的上采样

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

`src`

输入图像.

`dst`

输出图像, 宽度和高度应是输入图像的 2 倍

`filter`

卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5

函数 cvPyrUp 使用Gaussian 金字塔分解对输入图像向上采样。首先通过在图像中插入 0 偶数行和偶数列，然后对得到的图像用指定的滤波器进行高斯卷积，其中滤波器乘以 4 做差值。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

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

如果输入图像有单个通道（红、绿、兰），那么

$p(c^1,c^2)=0,3\cdot(c^1_r-c^2_r)+0,59\cdot(c^1_g-c^2_g)+0,11\cdot(c^1_b-c^2_b)$.

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<=src(x,y)<=src(x',y')+up_diff, 灰度图像，浮动范围
src(seed.x,seed.y)-lo<=src(x,y)<=src(seed.x,seed.y)+up_diff, 灰度图像，固定范围

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

$src(seed.x,seed.y)_r-lo\_diff_r<=src(x,y)_r<=src(seed.x,seed.y)_r+up\_diff_r$ 和
$src(seed.x,seed.y)_g-lo\_diff_g<=src(x,y)_g<=src(seed.x,seed.y)_g+up\_diff_g$ 和
$src(seed.x,seed.y)_b-lo\_diff_b<=src(x,y)_b<=src(seed.x,seed.y)_b+up\_diff_b$, 彩色图像，固定范围
其中 src(x',y') 是象素邻域点的值。也就是说，为了被加入到连通域中，一个象素的彩色/亮度应该足够接近于：

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

FindContours
在二值图像中寻找轮廓

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

`image`

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

`storage`

得到的轮廓的存储容器

`first_contour`

输出参数：包含第一个输出轮廓的指针

`header_size`

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

`mode`

提取模式.

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

`method`

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

- CV_CHAIN_CODE - Freeman 链码的输出轮廓. 其它方法输出多边形(定点序列).
- CV_CHAIN_APPROX_NONE - 将所有点由链码形式翻译为点序列形式
- 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：
$M_{x\_order,y\_order}=sum_{x,y}(I(x,y)\cdot x^{x\_order}\cdot y^{y\_order})$
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：
$\mu_{x\_order,y\_order}=sum_{x,y}(I(x,y)\cdot(x-x_c)^{x\_order}\cdot(y-y_c)^{y\_order})$,
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

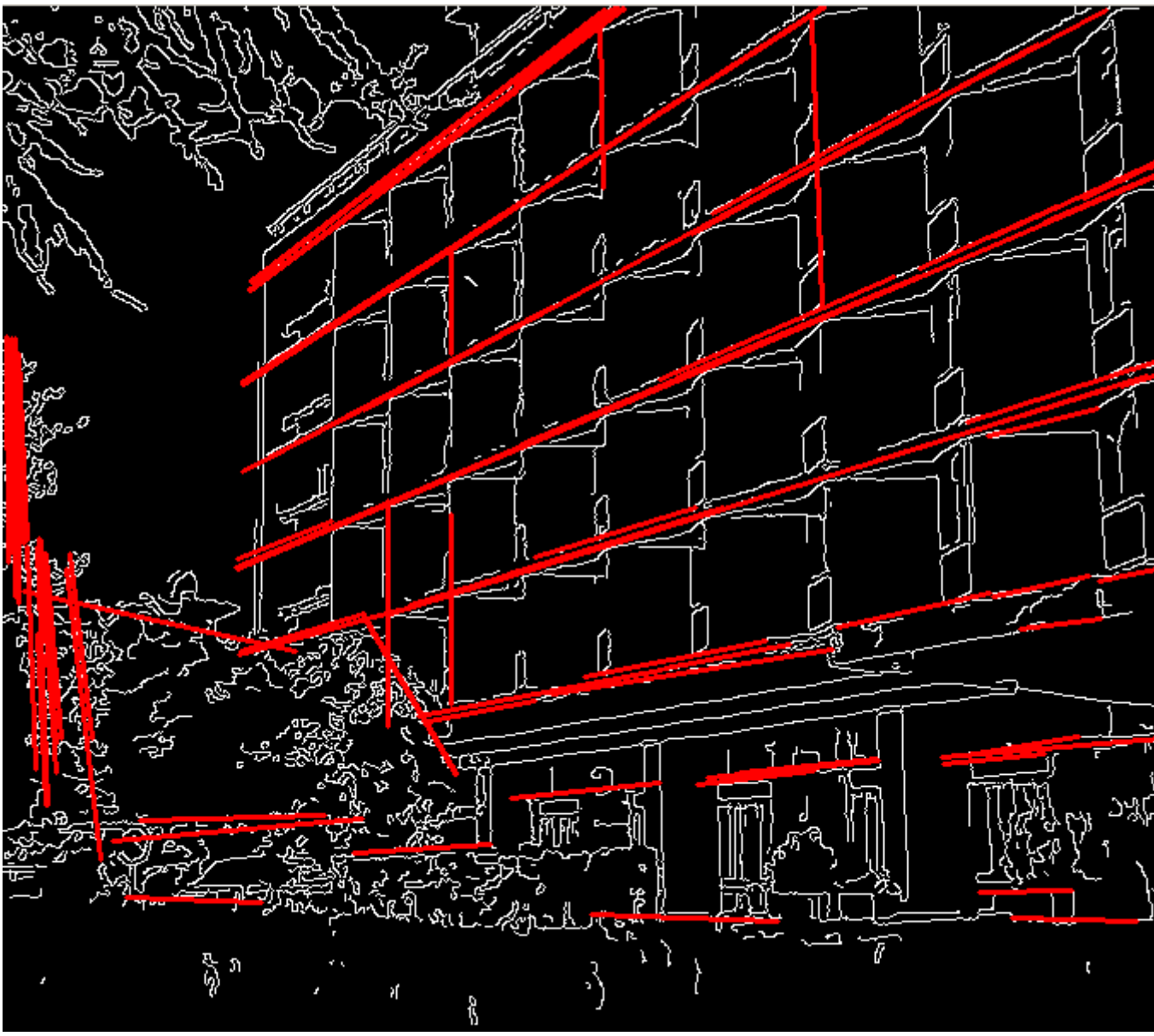
```
/* This is a standalone program. Pass an image name as a first parameter
of the program.
    Switch between standard and probabilistic Hough transform by
changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

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):
$d(H_1,H_2)=sum_I(H'_1(I)\bullet H'_2(I))/sqrt(sum_I[H'_1(I)^2]\bullet sum_I[H'_2(I)^2])$
其中
$H'_k(I)=H_k(I)-1/N\bullet sum_J H_k(J)$ (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`

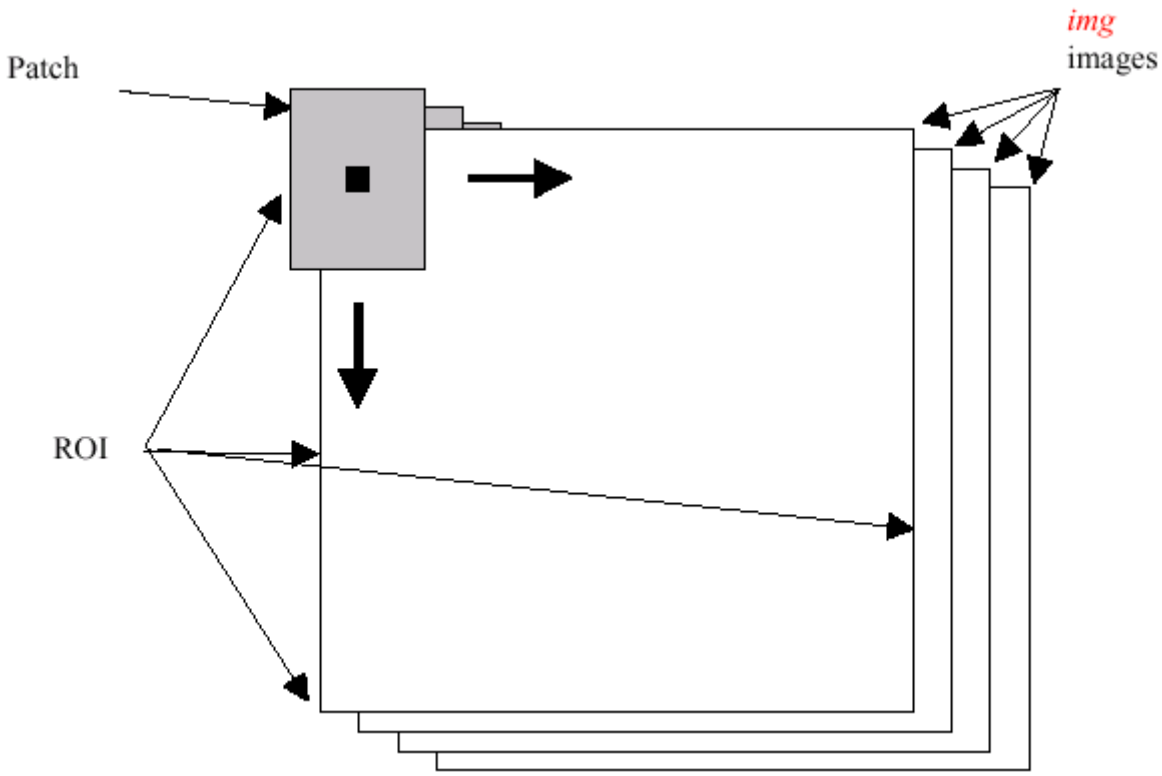
比较方法，传递给 cvCompareHist (见该函数的描述).

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

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

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

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

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。

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);

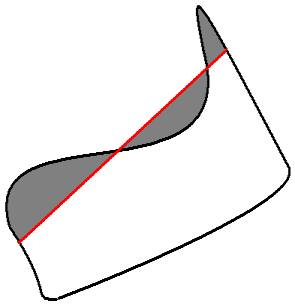
`contour`

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);

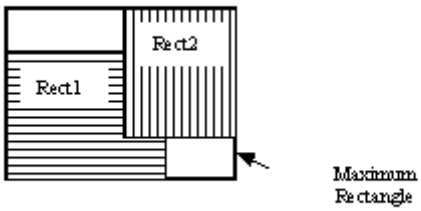
`rect1`

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， ρ(r) 是下面的距离函数之一：

dist_type=CV_DIST_L2 (L_2):
$\rho(r)=r^2/2$ (最简单和最快的最小二乘法)

dist_type=CV_DIST_L1 (L_1):
$\rho(r)=r$

dist_type=CV_DIST_L12 (L_1-L_2):
$\rho(r)=2 \cdot [\text{sqrt}(1+r^2/2) - 1]$

dist_type=CV_DIST_FAIR (Fair):

$\rho(r)=C^2 \cdot [r/C - \log(1 + r/C)], \quad C=1.3998$

dist_type=CV_DIST_WELSCH (Welsch):
$\rho(r)=C^2/2 \cdot [1 - \exp(-(r/C)^2)], \quad C=2.9846$

dist_type=CV_DIST_HUBER (Huber):
$\rho(r)= r^2/2, \quad \text{if } r < C$
$C \cdot (r-C/2), \quad \text{otherwise;} \quad C=1.345$

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。

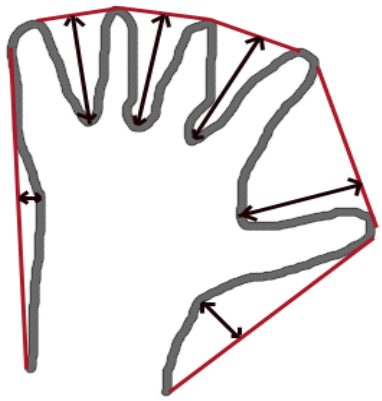
CvConvexityDefect

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

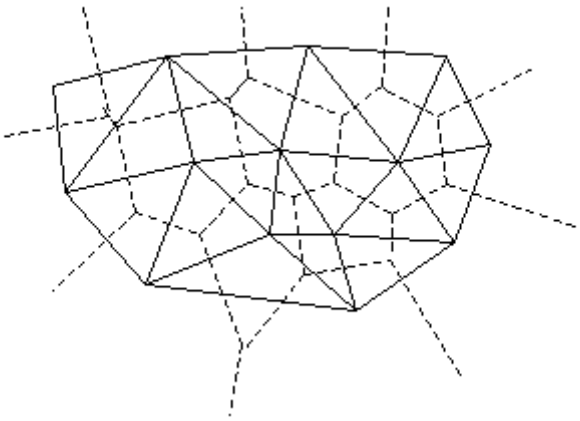
CvSubdiv2D

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)。

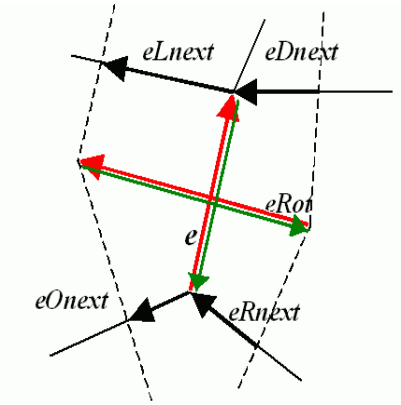
CvQuadEdge2D

平面划分中的 Quad-edge(四方边缘结构)

```
/* quad-edge 中的一条边缘，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

$sqsum(x,y)=sqsum(x,y)+image(x,y)^2$ 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]):

$x_k=A \cdot x_{k-1}+B \cdot u_k+w_k$ 译者注：系统运动方程

$z_k=H \cdot x_k+v_k,$ 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:

$$x'_k=A \cdot x_k+B \cdot u_k$$

$$P'_k=A \cdot P_{k-1} \cdot A^T + Q,$$

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：

$$K_k=P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T+R)^{-1}$$

$$x_k=x'_k+K_k \cdot (z_k-H \cdot x'_k)$$

$$P_k=(I-K_k \cdot H) \cdot P'_k$$

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`

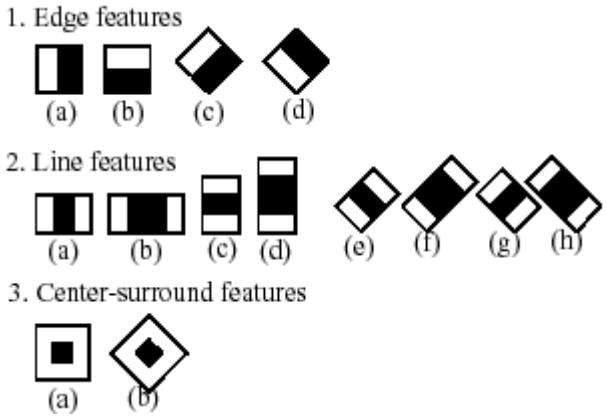
要更新的机构指针

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:
    $Stage_1$:
        $Classifier_{11}$:
            $Feature_{11}$
        $Classifier_{12}$:
            $Feature_{12}$
        ...
    $Stage_2$:
        $Classifier_{21}$:
            $Feature_{21}$
        ...
    ...

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,
CvPoint2D32f* image_points,
CvPoint3D32f* object_points,
CvVect32f focal_length,
CvPoint2D32f principal_point,
CvVect32f distortion_coeffs,
CvVect32f rotation_vector,
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,
CvPoint2D64d* image_points,
CvPoint3D64d* object_points,
CvVect64d focal_length,
CvPoint2D64d principal_point,
CvVect64d distortion_coeffs,
CvVect64d rotation_vector,
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_RA
NSAC,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_7P
OINT,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

D

E

F

G

H

I

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.