# OpenCV Documentation

***Release 1.1.0***

**OpenCV**

May 21, 2009

# CONTENTS

Welcome! This is the OpenCV documentation.

OpenCV is an Open Source Computer Vision Library. It is a collection of C functions and a few C++ classes that implement many popular image processing and computer vision algorithms.

CHAPTER

ONE

# API DOCUMENTATION

## 1.1 cxcore – Core Functionality

The cxcore module consists of data structures and functions that provide the basis for all computer vision algorithms.

### 1.1.1 Basic Data Structures

**CvPoint**

2D point with integer coordinates:

```
typedef struct CvPoint
{
  int x; /* x-coordinate, usually zero-based */
  int y; /* y-coordinate, usually zero-based */
}
CvPoint;

/* the constructor function */
inline CvPoint cvPoint( int x, int y );

/* conversion from CvPoint2D32f */
inline CvPoint cvPointFrom32f(CvPoint2D32f point);
```

**CvPoint2D32f**

2D point with floating-point coordinates:

```
typedef struct CvPoint2D32f
{
    float x; /* x-coordinate, usually zero-based */
    float y; /* y-coordinate, usually zero-based */
}
CvPoint2D32f;

/* the constructor function */
inline CvPoint2D32f cvPoint2D32f(double x, double y);

/* conversion from CvPoint */
inline CvPoint2D32f cvPointTo32f(CvPoint point);
```

**CvPoint3D32f**

3D point with floating-point coordinates:

```
typedef struct CvPoint3D32f
{
    float x; /* x-coordinate, usually zero-based */
    float y; /* y-coordinate, usually zero-based */
    float z; /* z-coordinate, usually zero-based */
}
CvPoint3D32f;

/* the constructor function */
inline CvPoint3D32f cvPoint3D32f(double x, double y, double z);
```

**Point2D64f**
2D point with double precision floating-point coordinates

```
typedef struct CvPoint2D64f
{
    double x; /* x-coordinate, usually zero-based */
    double y; /* y-coordinate, usually zero-based */
}
CvPoint2D64f;

/* the constructor function */
inline CvPoint2D64f cvPoint2D64f(double x, double y);

/* conversion from CvPoint */
inline CvPoint2D64f cvPointTo64f(CvPoint point);
```

**CvPoint3D64f**
3D point with double precision floating-point coordinates

```
typedef struct CvPoint3D64f
{
    double x; /* x-coordinate, usually zero-based */
    double y; /* y-coordinate, usually zero-based */
    double z; /* z-coordinate, usually zero-based */
}
CvPoint3D64f;

/* the constructor function */
inline CvPoint3D64f cvPoint3D64f(double x, double y, double z);
```

**CvSize**
pixel-accurate size of a rectangle

```
typedef struct CvSize
{
    int width; /* width of the rectangle */
    int height; /* height of the rectangle */
}
CvSize;

/* the constructor function */
inline CvSize cvSize( int width, int height );
```

**CvSize2D32f**
sub-pixel accurate size of a rectangle

```
typedef struct CvSize2D32f
{
    float width; /* width of the box */
    float height; /* height of the box */
}
CvSize2D32f;

/* the constructor function */
inline CvSize2D32f cvSize2D32f(double width, double height);
```

**CvRect**

offset and size of a rectangle

```
typedef struct CvRect
{
    int x; /* x-coordinate of the left-most rectangle
    corner[s] */
    int y; /* y-coordinate of the top-most or bottom-most
              rectangle corner[s] */
    int width; /* width of the rectangle */
    int height; /* height of the rectangle */
}
CvRect;

/* the constructor function */
inline CvRect cvRect( int x, int y, int width, int height );
```

**CvScalar**

A container for 1-,2-,3- or 4-tuples of numbers

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

/* the constructor function: initializes val[0] with val0,
val[1] with val1 etc. */
inline CvScalar cvScalar( double val0, double val1=0,
                          double
                          val2=0, double val3=0 );
/* the constructor function: initializes val[0]...val[3] with
val0123 */
inline CvScalar cvScalarAll( double val0123 );

/* the constructor function: initializes val[0] with val0,
val[1]...val[3] with zeros */
inline CvScalar cvRealScalar( double val0 );
```

**CvTermCriteria**

Termination criteria for iterative algorithms

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

typedef struct CvTermCriteria
```

```
{
    int    type;     /* a combination of CV_TERMCRIT_ITER and CV_TERMCRIT_EPS */
    int    max_iter; /* maximum number of iterations */
    double epsilon;  /* accuracy to achieve */
}
CvTermCriteria;

/* the constructor function */
inline  CvTermCriteria  cvTermCriteria( int type, int max_iter,
double epsilon );

/* check termination criteria and transform it so that
   type = CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
   and both max_iter and epsilon are valid */
CvTermCriteria cvCheckTermCriteria(CvTermCriteria criteria,
                                   double default_eps,
                                   int default_max_iters );
```

**CvMat**

Multi-channel matrix

```
typedef struct CvMat
{
    int type; /* CvMat signature (CV_MAT_MAGIC_VAL),
    element type and flags */
    int step; /* full row length in bytes */

    int* refcount; /* underlying data reference counter
    */

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

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

    union
    {
        int cols;
        int width;
    };
#else
    int rows; /* number of rows */
    int cols; /* number of columns */
#endif

} CvMat;
```

**CvMatND**

Multi-dimensional dense multi-channel array

```
typedef struct CvMatND
{
    int type; /* CvMatND signature (CV_MATND_MAGIC_VAL),
    element type and flags */
    int dims; /* number of array dimensions */

    int* refcount; /* underlying data reference counter
    */

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

    /* pairs (number of elements, distance between
    elements in bytes) for
       every dimension */
    struct
    {
        int size;
        int step;
    }
    dim[CV_MAX_DIM];

} CvMatND;
```

**CvSparseMat**

Multi-dimensional sparse multi-channel array

```
typedef struct CvSparseMat
{
    int type; /* CvSparseMat signature
    (CV_SPARSE_MAT_MAGIC_VAL), element type and flags */
    int dims; /* number of dimensions */
    int* refcount; /* reference counter - not used */
    struct CvSet* heap; /* a pool of hashtable nodes */
    void** hashtable; /* hashtable: each entry has a list
    of nodes
                         having the
                         same "hashvalue modulo hashsize" */
    int hashsize; /* size of hashtable */
    int total; /* total number of sparse array nodes */
    int valoffset; /* value offset in bytes for the array
    nodes */
    int idxoffset; /* index offset in bytes for the array
    nodes */
    int size[CV_MAX_DIM]; /* arr of dimension sizes */

} CvSparseMat;
```

**IplImage**

IPL image header

```
typedef struct _IplImage
{
    int  nSize;         /* sizeof(IplImage) */
    int  ID;            /* version (=0)*/
    int  nChannels;     /* Most of OpenCV functions support 1,2,3 or 4 channels */
    int  alphaChannel;  /* ignored by OpenCV */
    int  depth;         /* pixel depth in bits:
                           IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16U,
                           IPL_DEPTH_16S, IPL_DEPTH_32S,
                           IPL_DEPTH_32F and IPL_DEPTH_64F are supported */
    char colorModel[4]; /* ignored by OpenCV */
    char channelSeq[4]; /* ditto */
    int  dataOrder;     /* 0 - interleaved color channels,
                           1 - separate color channels.
                           cfunc:`CvCreateImage` can only create interleaved images */
    int  origin;        /* 0 - top-left origin,
                           1 - bottom-left origin (Windows bitmaps style) */
    int  align;         /* Alignment of image rows (4 or 8).
                           OpenCV ignores it and uses widthStep instead */
    int  width;         /* image width in pixels */
    int  height;        /* image height in pixels */
    struct _IplROI *roi;/* image ROI. when it is not NULL, this specifies image region to proces
    struct _IplImage *maskROI; /* must be NULL in OpenCV */
    void  *imageId;     /* ditto */
    struct _IplTileInfo *tileInfo; /* ditto */
    int  imageSize;     /* image data size in bytes (= image->height*image->widthStep
                           in case of interleaved data) */
    char *imageData;    /* pointer to aligned image data */
    int  widthStep;     /* size of aligned image row in bytes */
    int  BorderMode[4]; /* border completion mode, ignored by OpenCV */
    int  BorderConst[4]; /* ditto */
    char *imageDataOrigin; /* pointer to a very origin of image data
                              (not necessarily aligned) - it is needed for correct image dealloc
}
IplImage;
```

The structure `IplImage` came from *Intel Image Processing Library* where the format is native. OpenCV supports only a subset of possible `IplImage` formats:

- `alphaChannel` is ignored by OpenCV.
- `colorModel` and `channelSeq` are ignored by OpenCV. The single OpenCV function `cvCvtColor` working with color spaces takes the source and destination color spaces as a parameter.
- `dataOrder` must be `IPL_DATA_ORDER_PIXEL` (the color channels are interleaved), however selected channels of planar images can be processed as well if COI is set.
- `align` is ignored by OpenCV, while `widthStep` is used to access to subsequent image rows.
- `maskROI` is not supported. The function that can work with mask take it as a separate parameter. Also the mask in OpenCV is 8-bit, whereas in IPL it is 1-bit.
- `tileInfo` is not supported.
- `BorderMode` and `BorderConst` are not supported. Every OpenCV function working with a pixel neighborhood uses a single hard-coded border mode (most often, replication).

Besides the above restrictions, OpenCV handles ROI differently. It requires that the sizes or ROI sizes of all source and destination images match exactly (according to the operation, e.g. for `cvPyrDown` destination width(height) must be equal to source width(height) divided by $2 \pm 1$), whereas IPL processes the intersection area - that is, the sizes or ROI sizes of all images may vary independently.

**CvArr**

Arbitrary array

```
typedef void CvArr;
```

The metatype `CvArr*` is used *only* as a function parameter to specify that the function accepts arrays of more than a single type, for example `IplImage*`, `CvMat*` or even `CvSeq*`. The particular array type is determined at runtime by analysing the first 4 bytes of the header.

### 1.1.2 Operations on Arrays

#### Initialization

`IplImage*` **cvCreateImage**(*CvSize size, int depth, int num_channels*)

Creates header and allocates data for an image of size *size*, pixel bit depth *depth* and *num_channels* channels per pixel. The pixel bit depth can be one of the following:

- `IPL_DEPTH_8U` - unsigned 8-bit integers
- `IPL_DEPTH_8S` - signed 8-bit integers
- `IPL_DEPTH_16U` - unsigned 16-bit integers
- `IPL_DEPTH_16S` - signed 16-bit integers
- `IPL_DEPTH_32S` - signed 32-bit integers
- `IPL_DEPTH_32F` - single precision floating-point numbers
- `IPL_DEPTH_64F` - double precision floating-point numbers

The number of channels *num_channels* can be 1, 2, 3 or 4. The channels are interleaved, for example the usual data layout of a color image is:

```
b0 g0 r0 b1 g1 r1 ...
```

Although in general IPL image format can store non-interleaved images as well and some of OpenCV can process it, this function can create interleaved images only.

The function `cvCreateImage()` is a shortened form of

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

`IplImage*` **cvCreateImageHeader**(*CvSize size, int depth, int num_channels*)

Allocates, initializes, and returns the structure `IplImage` for an image of size *size*, pixel bit depth *depth* and *num_channels* channels per pixel (cf. `cvCreateImage`).

The function is an analogue of

```
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);
```

though it does not use IPL functions by default (see also `CV_TURN_ON_IPL_COMPATIBILITY` macro)

`void` **`cvReleaseImageHeader`**`(`*IplImage** image*`)`

Releases the header *image* (double pointer).

The function is an analogue of

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

though it does not use IPL functions by default (see also
:cmacro:`CV_TURN_ON_IPL_COMPATIBILITY`)
```

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

Releases header and image data of *image* (double pointer).

The function call is a shortened form of

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

`IplImage*` **`cvInitImageHeader`**`(`*IplImage* image, CvSize size, int depth, int num_channels[, int origin = 0, int align = 4]*`)`

Initializes the image header structure *image*, which was allocated by the user, and returns the pointer. The header is initialized for an image of size *size*, pixel bit depth *depth* and number of channels *num_channels*. *origin* can be one of `IPL_ORIGIN_TL` or `IPL_ORIGIN_BL`. *align* denotes the alignment for image rows, typically 4 or 8 bytes.

`IplImage*` **`cvCloneImage`**`(`*const IplImage* image*`)`

Makes a full copy of the image *image* including header, ROI and data.

`void` **`cvSetImageCOI`**`(`*IplImage* image, int coi*`)`

Sets channel of interest to *coi*.

`coi = 0` means that all channels are selected, 1 means that the first channel is selected etc. If ROI is `NULL` and `coi != 0`, ROI is allocated. Note that most of OpenCV functions do not support COI, so to process separate image/matrix channel one may copy (via `cvCopy` or `cvSplit`) the channel to separate image/matrix, process it and copy the result back (via `cvCopy` or `cvCvtPlaneToPix`) if need.

`int` **`cvGetImageCOI`**`(`*const IplImage* image*`)`

Returns index of channel of interest of image *image*. It returns 0 if all the channels are selected.

`void` **`cvSetImageROI`**`(`*IplImage* image, CvRect rect*`)`

Sets the image ROI to given rectangle *rect*.

If ROI is `NULL` and the value of the parameter `rect` is not equal to the whole image, ROI is allocated. Unlike COI, most of OpenCV functions do support ROI and treat it in a way as it would be a separate image (for example, all the pixel coordinates are counted from top-left or bottom-left (depending on the image origin) corner of ROI)

`void` **`cvResetImageROI`**`(`*IplImage* image*`)`

Releases image ROI

**Parameter** *image* – Image header.

The function `cvResetImageROI` releases image ROI. After that the whole image is considered selected. The similar result can be achieved by

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

But the latter variant does not deallocate `image->roi`.

`CvRect` **cvGetImageROI** (*const IplImage* image*)

Returns image ROI coordinates

**Parameter** *image* – Image header.

The function `cvGetImageROI` returns image ROI coordinates. The rectangle `cvRect(0, 0, image->width, image->height)` is returned if there is no ROI.

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

Creates new matrix

**Parameters**
- *rows* – Number of rows in the matrix.
- *cols* – Number of columns in the matrix.
- *type* – Type of the matrix elements. Usually it is specified in form `CV_<bit_depth>(S|U|F)C<number_of_channels>`, for example: `CV_8UC1` means an 8-bit unsigned single-channel matrix, `CV_32SC2` means a 32-bit signed matrix with two channels.

The function `cvCreateMat` allocates header for the new matrix and underlying data, and returns a pointer to the created matrix. It is a short form for

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

Matrices are stored row by row. All the rows are aligned by 4 bytes.

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

Creates new matrix header

**Parameters**
- *rows* – Number of rows in the matrix.
- *cols* – Number of columns in the matrix.
- *type* – Type of the matrix elements (see `cvCreateMat`).

The function `cvCreateMatHeader` allocates new matrix header and returns pointer to it. The matrix data can further be allocated using `cvCreateData` or set explicitly to user-allocated data via `cvSetData`.

`void` **cvReleaseMat** (*CvMat** mat*)

Deallocates matrix

**Parameter** *mat* – Double pointer to the matrix.

The function `cvReleaseMat` decrements the matrix data reference counter and releases matrix header

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

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

Initializes matrix header

**Parameters**
- *mat* – Pointer to the matrix header to be initialised.

- *rows* – Number of rows in the matrix.
- *cols* – Number of columns in the matrix.
- *type* – Type of the matrix elements.
- *data* – Optional data pointer assigned to the matrix header.
- *step* – Full row width in bytes of the data assigned. By default, the minimal possible step is used, i.e., no gaps is assumed between subsequent rows of the matrix.

The function `cvInitMatHeader` initializes already allocated `CvMat` structure. It can be used to process raw data with OpenCV matrix functions.

For example, the following code computes matrix product of two matrices, stored as ordinary arrays.

**Example:** Calculating product of two matrices

```
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 array now contains product of a(3x4) and b(4x3) matrices
```

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

Initializes matrix header (light-weight variant)

**Parameters**
- *rows* – Number of rows in the matrix.
- *cols* – Number of columns in the matrix.
- *type* – Type of the matrix elements (see CreateMat).
- *data* – Optional data pointer

assigned to the matrix header.

The function `cvMat` is a fast inline substitution for `cvInitMatHeader`. Namely, it is equivalent to:

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

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

Creates matrix copy

**Parameter** *mat* – Input matrix.

The function `cvCloneMat` creates a copy of input matrix and returns the pointer to it.

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

Creates multi-dimensional dense array

**Parameters**
- *dims* – Number of array dimensions. It must not exceed `CV_MAX_DIM` (=32 by default, though it may be changed at build time)

- *sizes* – Array of dimension sizes.
- *type* – Type of array elements. The same as for `CvMat`

The function `cvCreateMatND` allocates header for multi-dimensional dense array and the underlying data, and returns pointer to the created array. It is a short form for

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

Array data is stored row by row. All the rows are aligned by 4 bytes.

`CvMatND*` **`cvCreateMatNDHeader`** (*int dims, const int* sizes, int type*)
Creates new matrix header

**Parameters**
- *dims* – Number of array dimensions.
- *sizes* – Array of dimension sizes.
- *type* – Type of array elements. The same as for `CvMat`

The function `cvCreateMatND` allocates header for multi-dimensional dense array. The array data can further be allocated using `cvCreateData` or set explicitly to user-allocated data via `cvSetData`.

`void` **`cvReleaseMatND`** (*CvMatND** mat*)
Deallocates multi-dimensional array

**Parameter** *mat* – Double pointer to the array.

The function `cvReleaseMatND` decrements the array data reference counter and releases the array header:

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

`CvMatND*` **`cvInitMatNDHeader`** (*CvMatND* mat, int dims, const int* sizes, int type, void* data=NULL*)
Initializes multi-dimensional array header

**Parameters**
- *mat* – Pointer to the array header to be initialized.
- *dims* – Number of array dimensions.
- *sizes* – Array of dimension sizes.
- *type* – Type of array elements. The same as for `CvMat`
- *data* – Optional data pointer assigned to the matrix header.

The function `cvInitMatNDHeader` initializes `CvMatND` structure allocated by the user.

`CvMatND*` **`cvCloneMatND`** (*const CvMatND* mat*)
Creates full copy of multi-dimensional array

**Parameter** *mat* – Input array.

The function `cvCloneMatND` creates a copy of input array and returns pointer to it.

`void` **`cvDecRefData`** (*CvArr* arr*)
Decrements array data reference counter

**Parameter** *arr* – Array header.

The function `cvDecRefData` decrements `CvMat` or `CvMatND` data reference counter if the reference counter pointer is not NULL and deallocates the data if the counter reaches zero. In the current implementation the reference counter is not NULL only if the data was allocated using `cvCreateData` function, in other cases such as: external data was assigned to the header using `cvSetData` the matrix header presents a part of a larger matrix or image the matrix header was converted from image or n-dimensional matrix header the reference counter is set to NULL and thus it is not decremented. Whenever the data is deallocated or not, the data pointer and reference counter pointers are cleared by the function.

`int` **`cvIncRefData`**(*CvArr* arr*)
Increments array data reference counter

**Parameter** *arr* – array header.

The function `cvIncRefData` increments `CvMat` or `CvMatND` data reference counter and returns the new counter value if the reference counter pointer is not `NULL`, otherwise it returns zero.

`void` **`cvCreateData`**(*CvArr* arr*)
Allocates array data

**Parameter** *arr* – Array header.

The function `cvCreateData` allocates image, matrix or multi-dimensional array data. Note that in case of matrix types OpenCV allocation functions are used and in case of IplImage they are used too unless `CV_TURN_ON_IPL_COMPATIBILITY` was called. In the latter case IPL functions are used to allocate the data

`void` **`cvReleaseData`**(*CvArr* arr*)
Releases array data

**Parameter** *arr* – Array header

The function `cvReleaseData` releases the array data. In case of `CvMat` or `CvMatND` it simply calls `cvDecRefData()`, that is the function cannot deallocate external data. See also the note to `cvCreateData`.

`void` **`cvSetData`**(*CvArr* arr, void* data, int step*)
Assigns user data to the array header

**Parameters**
- *arr* – Array header.
- *data* – User data.
- *step* – Full row length in bytes.

The function `cvSetData` assigns user data to the array header. The header should be initialized before by using `cvCreate*Header`, `cvInit*Header` or `cvMat` (in case of matrix) function.

`cvoid` **`cvGetRawData`**(*const CvArr* arr, uchar** data, int* step=NULL, CvSize* roi_size=NULL*)
Retrieves low-level information about the array

**Parameters**
- *arr* – Array header.
- *data* – Output pointer to the whole image origin or ROI origin if ROI is set.
- *step* – Output full row length in bytes.
- *roi_size* – Output ROI size.

The function `cvGetRawData` fills output variables with low-level information about the array data. All output parameters are optional, so some of the pointers may be set to `NULL`. If the array is `IplImage` with ROI set, parameters of ROI are returned.

The following example shows how to get access to array elements using this function.

**Example:** Using `cvGetRawData` to calculate absolute value of elements of a single-channel floating-point array.

```
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]);
```

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

Returns matrix header for arbitrary array *arr*.

**Parameters**
- *arr* – Input array.
- *header* – Pointer to `CvMat` structure used as a temporary buffer.
- *coi* – Optional output parameter for storing COI.
- *allowND* – If non-zero, the function accepts multi-dimensional dense arrays (`CvMatND*`) and returns 2D (if *arr* has two dimensions) or 1D matrix (when *arr* has 1 dimension or more than 2 dimensions). The array must be continuous.

The input array *arr* can be a matrix - `CvMat`, an image - `IplImage` or multi-dimensional dense array - `CvMatND*` (latter case is allowed only if `allowND != 0`). In the case of matrix the function simply returns the input pointer. In the case of `IplImage*` or `CvMatND*` it initializes *header* with parameters of the current image ROI and returns pointer to this temporary structure. Because COI is not supported by `CvMat`, it is returned separately.

The function provides an easy way to handle both types of array - `IplImage` and `CvMat` -, using the same code. Reverse transform from `CvMat` to `IplImage` can be done using `cvGetImage` function.

The input array must have underlying data allocated or attached, otherwise the function fails.

If the input array is `IplImage` with planar data layout and COI set, the function returns pointer to the selected plane and COI = 0. It enables per- plane processing of multi-channel images with planar data layout using OpenCV functions.

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

Returns image header for arbitrary array given by *arr*. *image_header* is a pointer to an `IplImage` structure which is used as a temporary buffer.

The input array *arr* can be a matrix `CvMat*` or image `IplImage*`. In the case of image the function simply returns the input pointer. In the case of `CvMat*` it initializes *image_header* with parameters of the input matrix. Note that if we transform `IplImage` to `CvMat` and then transform `CvMat` back to `IplImage`, we can get different headers if the ROI is set, and thus some IPL functions that calculate image stride from its width and align may fail on the resultant image.

`CvSparseMat* cvCreateSparseMat(int num_dims, const int* sizes, int type)`

Creates sparse array with *num_dims* dimensions. In contrast to the dense matrix, the number of dimensions is practically unlimited (up to 216). *sizes* is an array denoting the dimension sizes. *type* denotes the type of the array elements (see `cvCreateMat`).

Initially the array contains no elements, that is `cvGet*D` or `cvGetReal*D` return zero for every index.

`void cvReleaseSparseMat(CvSparseMat** mat)`

Deallocates the sparse array *mat* and clears the array pointer upon exit.

`CvSparseMat* cvCloneSparseMat(const CvSparseMat* mat)`

Creates full copy of sparse array *mat*. It returns a pointer to the copy.

### Accessing Elements and Sub-Arrays

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

Returns matrix header corresponding to the rectangular sub-array of input image or matrix

**Parameters**
- *arr* – Input array.
- *submat* – Pointer to the resultant sub-array header.
- *rect* – Zero-based coordinates of the rectangle of interest.

The function `cvGetSubRect` returns header, corresponding to a specified rectangle of the input array. In other words, it allows the user to treat a rectangular part of input array as a stand-alone array. ROI is taken into account by the function so the sub-array of ROI is actually extracted.

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*)

Returns array row or row span

**Parameters**
- *arr* – Input array.
- *submat* – Pointer to the resulting sub-array header.
- *row* – Zero-based index of the selected row.
- *start_row* – Zero-based index of the starting row (inclusive) of the span.
- *end_row* – Zero-based index of the ending row (exclusive) of the span.
- *delta_row* – Index step in the row span. That is, the function extracts every `delta_row`-th row from `start_row` and up to (but not including) `end_row`.

The functions `cvGetRow` and `cvGetRows` return the header, corresponding to a specified row/row span of the input array. Note that `cvGetRow` is a shortcut for `cvGetRows`:

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

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

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

Returns array column or column span

**Parameters**
- *arr* – Input array.
- *submat* – Pointer to the resulting sub-array header.
- *col* – Zero-based index of the selected column.
- *start_col* – Zero-based index of the starting column (inclusive) of the span.
- *end_col* – Zero-based index of the ending column (exclusive) of the span.

The functions `cvGetCol` and `cvGetCols` return the header, corresponding to a specified column/column span of the input array. Note that `cvGetCol` is a shortcut for `cvGetCols`:

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

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

Returns one of array diagonals

**Parameters**
- *arr* – Input array.
- *submat* – Pointer to the resulting sub-array header.
- *diag* – Array diagonal. Zero corresponds to the main diagonal, -1 corresponds to the diagonal above the main etc., 1 corresponds to the diagonal below the main etc.

The function `cvGetDiag` returns the header, corresponding to a specified diagonal of the input array.

CvSize **cvGetSize**(*const CvArr* arr*)

Returns size of matrix or image ROI

**Parameter** *arr* – array header.

The function `cvGetSize` returns number of rows (CvSize::height) and number of columns (CvSize::width) of the input matrix or image. In case of image the size of ROI is returned.

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

Initializes sparse array elements iterator

**Parameters**
- *mat* – Input array.
- *mat_iterator* – Initialized iterator.

The function `cvInitSparseMatIterator` initializes iterator of sparse array elements and returns pointer to the first element, or NULL if the array is empty.

`CvSparseNode*` **cvGetNextSparseNode**(*CvSparseMatIterator* mat_iterator*)

Initializes sparse array elements iterator

**Parameter** *mat_iterator* – Sparse array iterator.

The function `cvGetNextSparseNode` moves iterator to the next sparse matrix element and returns pointer to it. In the current version there is no any particular order of the elements, because they are stored in hash table. The sample below demonstrates how to iterate through the sparse matrix:

Using `cvInitSparseMatIterator` and `cvGetNextSparseNode` to calculate sum of floating-point sparse array.

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

for (; node != 0; node = cvGetNextSparseNode(&mat_iterator)) {
    const 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 );
```

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

Returns type of array elements

**Parameter** *arr* – Input array.

The functions `cvGetElemType` returns type of the array elements as it is described in cvCreateMat discussion: ::CV_8UC1 ... CV_64FC4

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

`int` **cvGetDimSize**(*const CvArr* arr, int index*)

Return number of array dimensions and their sizes or the size of particular dimension

**Parameters**
- *arr* – Input array.
- *sizes* – Optional output vector of the array dimension sizes. For 2d arrays the number of rows (height) goes first, number of columns (width) next.

- *index* – Zero-based dimension index (for matrices 0 means number of rows, 1 means number of columns; for images 0 means height, 1 means width).

The function `cvGetDims` returns number of array dimensions and their sizes. In case of `IplImage` or `CvMat` it always returns 2 regardless of number of image/matrix rows. The function `cvGetDimSize` returns the particular dimension size (number of elements per that dimension). For example, the following code calculates total number of array elements in two ways

```
// 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 );
```

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, const int* idx, int* type=NULL, int create_node=1, unsigned* precalc_hashval=NULL*)

Return pointer to the particular array element

**Parameters**
- *arr* – Input array.
- *idx0* – The first zero-based component of the element index
- *idx1* – The second zero-based component of the element index
- *idx2* – The third zero-based component of the element index
- *idx* – Array of the element indices
- *type* – Optional output parameter: type of matrix elements
- *create_node* – Optional input parameter for sparse matrices. Non-zero value of the parameter means that the requested element is created if it does not exist already.
- *precalc_hashval* – Optional input parameter for sparse matrices. If the pointer is not `NULL`, the function does not recalculate the node hash value, but takes it from the specified location. It is useful for speeding up pair-wise operations (TODO: provide an example)

The functions `cvPtr*D` return pointer to the particular array element. Number of array dimension should match to the number of indices passed to the function except for `cvPtr1D` function that can be used for sequential access to 1D, 2D or nD dense arrays.

The functions can be used for sparse arrays as well - if the requested node does not exist they create it and set it to zero.

All these as well as other functions accessing array elements (`cvGet*D`, `cvGetReal*D`, `cvSet*D`, `cvSetReal*D`) raise an error in case if the element index is out of range.

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, const int* idx*)

Return the particular array element

**Parameters**
- *arr* – Input array.
- *idx0* – The first zero-based component of the element index
- *idx1* – The second zero-based component of the element index
- *idx2* – The third zero-based component of the element index
- *idx* – Array of the element indices

The functions `cvGet*D` return the particular array element. In case of sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions)

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, const int* idx*)

Return the particular element of single-channel array

**Parameters**
- *arr* – Input array. Must have a single channel.
- *idx0* – The first zero-based component of the element index
- *idx1* – The second zero-based component of the element index
- *idx2* – The third zero-based component of the element index
- *idx* – Array of the element indices

The functions `cvGetReal*D` return the particular element of single-channel array. If the array has multiple channels, runtime error is raised. Note that `cvGet*D` function can be used safely for both single-channel and multiple- channel arrays though they are a bit slower.

In case of sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions)

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

Return the particular element of single-channel floating-point matrix

**Parameters**
- *mat* – Input matrix.
- *row* – The zero-based index of row.
- *col* – The zero-based index of column.

The function `cvmGet` is a fast replacement for `cvGetReal2D` in case of single-channel floating-point matrices. It is faster because it is inline, it does less checks for array type and array element type and it checks for the row and column ranges only in debug mode.

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, const int* idx, CvScalar value*)

Change the particular array element

**Parameters**
- *arr* – Input array.
- *idx0* – The first zero-based component of the element index
- *idx1* – The second zero-based component of the element index
- *idx2* – The third zero-based component of the element index
- *idx* – Array of the element indices
- *value* – The assigned value

The functions `cvSet*D` assign the new value to the particular element of array. In case of sparse array the functions create the node if it does not exist yet 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, const int* idx, double value*)

Change the particular array element

**Parameters**
- *arr* – Input array.
- *idx0* – The first zero-based component of the element index
- *idx1* – The second zero-based component of the element index.
- *idx2* – The third zero-based component of the element index.
- *idx* – Array of the element indices.
- *value* – The assigned value

The functions `cvSetReal*D` assign the new value to the particular element of single-channel array. If the array has multiple channels, runtime error is raised. Note that `cvSet*D` function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In case of sparse array the functions create the node if it does not exist yet.

void **cvmSet** (*CvMat* mat, int row, int col, double value*)

Return the particular element of single-channel floating-point matrix

**param mat** The matrix.

**param row** The zero-based index of row.

**param col** The zero-based index of column.

**param value** The new value of the matrix element

The function `cvmSet` is a fast replacement for `cvSetReal2D` in case of single-channel floating-point matrices. It is faster because it is inline, it does less checks for array type and array element type and it checks for the row and column ranges only in debug mode.

void **cvClearND** (*CvArr* arr, const int* idx*)

Clears the particular array element

**Parameters**
- *arr* – Input array.
- *idx* – Array of the element indices

The function `cvClearND` clears (sets to zero) the particular element of dense array or deletes the element of sparse array. If the element does not exists, the function does nothing.

### Copying and Filling

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

Copies one array to another

**Parameters**
- *src* – The source array.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `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. The function can also copy sparse arrays (mask is not supported in this case).

void **cvSet** (*CvArr* arr, CvScalar value, const CvArr* mask=NULL*)
Sets every element of array to given value

**Parameters**
- *arr* – The destination array.
- *value* – Fill value.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvSet` copies scalar `value` to every selected element of the destination array

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

If array `arr` is of `IplImage` type, then is ROI used, but COI must not be set.

void **cvSetZero** (*CvArr* arr*)

void **cvZero** (*CvArr* arr*)
Clears the array

**Parameter** *arr* – array to be cleared.

The function `cvSetZero` clears the array. In case of dense arrays (CvMat, CvMatND or IplImage) cvZero(array) is equivalent to cvSet(array,cvScalarAll(0),0), in case of sparse arrays all the elements are removed.

void **cvSetIdentity** (*CvArr* mat, CvScalar value=cvRealScalar(1)*)
Initializes scaled identity matrix

**Parameters**
- *arr* – The matrix to initialize (not necessarily square).
- *value* – The value to assign to the diagonal elements.

The function `cvSetIdentity` initializes scaled identity matrix

```
arr(i,j) = value if i = j,
             0 otherwise
```

void **cvRange** (*CvArr* mat, double start, double end*)
Fills matrix with given range of numbers

**Parameters**
- *mat* – The matrix to initialize. It should be single-channel 32-bit, integer or floating-point.
- *start* – The lower inclusive boundary of the range.
- *end* – The upper exclusive boundary of the range.

The function `cvRange` initializes the matrix as following

```
arr(i,j)=(end-start)*(i*cols(arr)+j)/(cols(arr)*rows(arr))
```

For example, the following code will initialize 1D vector with subsequent integer numbers

```
CvMat* A = cvCreateMat( 1, 10, CV_32S );
cvRange( A, 0, A->cols ); // A will be initialized as [0,1,2,3,4,5,6,7,8,9]
```

### Transforms and Permutations

CvMat* **cvReshape** (*const CvArr* arr, CvMat* header, int new_cn, int new_rows=0*)
Changes shape of matrix/image without copying data

**Parameters**
- *arr* – Input array.
- *header* – Output header to be filled.
- *new_cn* – New number of channels. `new_cn = 0` means that number of channels remains unchanged.
- *new_rows* – New number of rows. `new_rows = 0` means that number of rows remains unchanged unless it needs to be changed according to `new_cn` value. destination array to be changed.

The function `cvReshape` initializes CvMat header so that it points to the same data as the original array but has different shape - different number of channels, different number of rows or both.

For example, the following code creates one image buffer and two image headers, first is for 320x240x3 image and the second is for 960x240x1 image

```
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 );
```

And the next example converts 3x3 matrix to a single 1x9 vector

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

CvArr* **cvReshapeMatND** (*const CvArr* arr, int sizeof_header, CvArr* header, int new_cn, int new_dims, int* new_sizes*)
Changes shape of multi-dimensional array w/o copying data

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

**Parameters**
- *arr* – Input array.
- *sizeof_header* – Size of output header to distinguish between IplImage, CvMat and CvMatND output headers.
- *header* – Output header to be filled.
- *new_cn* – New number of channels. `new_cn = 0` means that number of channels remains unchanged.
- *new_dims* – New number of dimensions. `new_dims = 0` means that number of dimensions remains the same.
- *new_sizes* – Array of new dimension sizes. Only `new_dims-1` values are used, because the total number of elements must remain the same. Thus, if `new_dims = 1`, `new_sizes` array is not used

The function `cvReshapeMatND` is an advanced version of `cvReshape` that can work with multi-dimensional arrays as well (though, it can work with ordinary images and matrices) and change the number of dimensions. Below are the two samples from the `cvReshape` description rewritten using `cvReshapeMatND`

```
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 );
```

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

Fill destination array with tiled source array

**Parameters**
- *src* – Source array, image or matrix.
- *dst* – Destination array, image or matrix.

The function `cvRepeat` fills the destination array with source array tiled

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

So the destination array may be as larger as well as smaller than the source array.

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

Flip a 2D array around vertical, horizontal or both axises

**::** #define cvMirror cvFlip

**Parameters**
- *src* – Source array.
- *dst* – Destination array. If `dst = NULL` the flipping is done in-place.
- *flip_mode* – Specifies how to flip the array. flip_mode = 0 means flipping around x-axis, flip_mode > 0 (e.g. 1) means flipping around y-axis and flip_mode < 0 (e.g. -1) means flipping around both axises. See also the discussion below for the formulas

The function `cvFlip` flips the array in one of different 3 ways (row and column indices are 0-based)

```
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
```

The example scenario of the function use are:

- vertical flipping of the image (flip_mode > 0) to switch between top- left and bottom-left image origin, which is typical operation in video processing under Win32 systems.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (flip_mode > 0)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flip_mode < 0)
- reversing the order of 1d point arrays(flip_mode > 0)

void **cvSplit** (*const CvArr* src, CvArr* dst0, CvArr* dst1, CvArr* dst2, CvArr* dst3*)

Divides multi-channel array into several single-channel arrays or extracts a single channel from the array

**::** #define cvCvtPixToPlane cvSplit

**Parameter** *src* – Source array. dst0...dst3Destination channels.

The function cvSplit divides a multi-channel array into separate single- channel arrays. Two modes are available for the operation. If the source array has N channels then if the first N destination channels are not NULL, all they are extracted from the source array, otherwise if only a single destination channel of the first N is not NULL, this particular channel is extracted, otherwise an error is raised. Rest of destination channels (beyond the first N) must always be NULL. For IplImage cvCopy with COI set can be also used to extract a single channel from the image.

void **cvMerge**(*const CvArr* src0, const CvArr* src1, const CvArr* src2, const CvArr* src3, CvArr* dst*)

Composes multi-channel array from several single-channel arrays or inserts a single channel into the array

**::** #define cvCvtPlaneToPix cvMerge

**Parameters**
- *src0 ... src3* – Input channels.
- *dst* – Destination array.

The function cvMerge is the opposite to the previous. If the destination array has N channels then if the first N input channels are not NULL, all they are copied to the destination array, otherwise if only a single source channel of the first N is not NULL, this particular channel is copied into the destination array, otherwise an error is raised. Rest of source channels (beyond the first N) must always be NULL. For IplImage cvCopy with COI set can be also used to insert a single channel into the image.

void **cvMixChannels**(*const CvArr** src, int src_count, CvArr** dst, int dst_count, const int* from_to, int pair_count*)

Copies several channels from input arrays to certain channels of output arrays

**Parameters**
- *src* – The array of input arrays.
- *src_count* – The number of input arrays.
- *dst* – The array of output arrays.
- *dst_count* – The number of output arrays.
- *from_to* – The array of pairs of indices of the planes copied. `from_to[k*2]` is the 0-based index of the input plane, and `from_to[k*2+1]` is the index of the output plane, where the continuous numbering of the planes over all the input and over all the output arrays is used. When `from_to[k*2]` is negative, the corresponding output plane is filled with 0's.
- *pair_count* – The number of pairs in `from_to`, or the number of the planes copied.

The function cvMixChannels is a generalized form of cvSplit and cvMerge and some forms of cvCvtColor. It can be used to change the order of the planes, add/remove alpha channel, extract or insert a single plane or multiple planes etc. Below is the example, how to split 4-channel RGBA image into 3-channel BGR (i.e. with R&B swapped) and separate alpha channel images

```
CvMat* rgba = cvCreateMat( 100, 100, CV_8UC4 );
CvMat* bgr = cvCreateMat( rgba->rows, rgba->cols, CV_8UC3 );
CvMat* alpha = cvCreateMat( rgba->rows, rgba->cols, CV_8UC1);
CvArr* out[] = { bgr, alpha };
int from_to[] = { 0, 2, 1, 1, 2, 0, 3, 3 };
cvSet( rgba, cvScalar(1,2,3,4) );
cvMixChannels( (const CvArr**)&rgba, 1, out, 2, from_to, 4 );
```

void **cvRandShuffle**(*CvArr* mat, CvRNG* rng, double iter_factor=1.*)

Randomly shuffles the array elements

**Parameters**
- *mat* – The input/output matrix. It is shuffled in-place.
- *rng* – The **'Random Number Generator'_** used to shuffle the elements. When the pointer is NULL, a temporary RNG will be created and used.

- *iter_factor* – The relative parameter that characterizes intensity of the shuffling performed. See the description below.

The function `cvRandShuffle` shuffles the matrix by swapping randomly chosen pairs of the matrix elements on each iteration (where each element may contain several components in case of multi-channel arrays). The number of iterations (i.e. pairs swapped) is `round(iter_factor*rows(mat)*cols(mat))`, so `iter_factor=0` means that no shuffling is done, `iter_factor=1` means that the function swaps `rows(mat)*cols(mat)` random pairs etc.

void **cvSort** (*const CvArr* src, CvArr* dst=NULL, CvArr* idxmat=NULL, int flags=0*)
Sort array elements in ascending or descending order by row or by column

**::** #define CV_SORT_EVERY_ROW 0 #define CV_SORT_EVERY_COLUMN 1 #define CV_SORT_ASCENDING 0 #define CV_SORT_DESCENDING 16

**Parameters**
- *src* – The source one-channel array.
- *dst* – The destination (sorted) array. If not NULL, it should be the same size and the same type as the source array.
- *idxmat* – An array of indices. If not NULL, It should be the same size as the source array and the type must be 32SC1.
- *flags* – The operation flags, 0 or a combination of: - CV_SORT_EVERY_ROW: Sort each row independently. CV_SORT_EVERY_ROW and

  CV_SORT_EVERY_COLUMN are mutually exclusive, of course.
  - CV_SORT_EVERY_COLUMN: Sort each column independently.
  - **CV_SORT_ASCENDING: Sort the elements of each row or column in ascending**
    order. CV_SORT_ASCENDING and CV_SORT_DESCENDING are mutually exclusive, of course.
  - CV_SORT_DESCENDING: Sort the elements of each row or column in descending order.

The function cvSort sorts elements of each row or column of the input 2D array in ascending or descending order. The function supports in-place mode (dst == src).

The following sample demonstrates how to sort a matrix

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

int main()
{
    float a[] = { 6,4,8,3,
                  3,5,2,4};
    float b[2*4];
    int c[2*4];

    CvMat src, dst, idx;

    cvInitMatHeader( &src, 2, 4, CV_32FC1, a );
    cvInitMatHeader( &dst, 2, 4, CV_32FC1, b );
    cvInitMatHeader( &idx, 2, 4, CV_32SC1, c );

    cvSort(&src, &dst, &idx);

    int i,j;
    for(i = 0; i < dst.rows; i++)
    {
```

```
        for(j = 0; j < dst.cols; j++)
            printf("%.1f  ", CV_MAT_ELEM(dst, float, i,
            j));
        printf("\n");
    }

    for(i = 0; i < idx.rows; i++)
    {
        for(j = 0; j < idx.cols; j++)
            printf("%d  ", CV_MAT_ELEM(idx, int, i, j));
        printf("\n");
    }
}
```

The code should print

```
3.0  4.0  6.0  8.0
2.0  3.0  4.0  5.0
3  1  0  2
2  0  3  1
```

### Arithmetic, Logic and Comparison

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

Performs look-up table transform of array

**Parameters**
- *src* – Source array of 8-bit elements.
- *dst* – Destination array of arbitrary depth and of the same number of channels as the source array.
- *lut* – Look-up table of 256 elements; should have the same depth as the destination array. In case of multi-channel source and destination arrays, the table should either have a single-channel (in this case the same table is used for all channels), or the same number of channels as the source/destination array.

The function `cvLUT` fills the destination array with values from the look- up table. Indices of the entries are taken from the source array. That is, the function processes each element of `src` as following:

**::** dst(I)=lut[src(I)+DELTA] where `DELTA=0` if `src` has depth `CV_8U`, and `DELTA=128` if `src` has depth `CV_8S`.

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

Converts one array to another with optional linear transformation

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

**Parameters**
- *src* – Source array.
- *dst* – Destination array.
- *scale* – Scale factor.
- *shift* – Value added to the scaled source array elements.

The function `cvConvertScale` has several different purposes and thus has several synonyms. It copies one array to another with optional scaling, which is performed first, and/or optional type conversion, performed after

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

All the channels of multi-channel arrays are processed independently.

The type conversion is done with rounding and saturation, that is if a result of scaling + conversion can not be represented exactly by a value of destination array element type, it is set to the nearest representable value on the real axis.

In case of `scale=1, shift=0` no pre-scaling is done. This is a specially optimized case and it has the appropriate `cvConvert` synonym. If source and destination array types have equal types, this is also a special case that can be used to scale and shift a matrix or an image and that fits to `cvScale` synonym.

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

Converts input array elements to 8-bit unsigned integer another with optional linear transformation

**::** #define cvCvtScaleAbs cvConvertScaleAbs

**Parameters**
- *src* – Source array.
- *dst* – Destination array (should have 8u depth).
- *scale* – ScaleAbs factor.
- *shift* – Value added to the scaled source array elements.

The function `cvConvertScaleAbs` is similar to the previous one, but it stores absolute values of the conversion results

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

The function supports only destination arrays of 8u (8-bit unsigned integers) type, for other types the function can be emulated by combination of `cvConvertScale` and `cvAbs` functions.

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

Computes per-element sum of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvAdd` adds one array to another one

```
dst(I)=src1(I)+src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size)

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

Computes sum of array and scalar

**Parameters**
- *src* – The source array.
- *value* – Added scalar.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvAddS` adds scalar `value` to every element in the source array `src1` and stores the result in `dst`

```
dst(I)=src(I)+value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size)

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

Computes weighted sum of two arrays

**Parameters**
- *src1* – The first source array.
- *alpha* – Weight of the first array elements.
- *src2* – The second source array.
- *beta* – Weight of the second array elements.
- *dst* – The destination array.
- *gamma* – Scalar, added to each sum.

The function `cvAddWeighted` calculated weighted sum of two arrays as following

```
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
```

All the arrays must have the same type and the same size (or ROI size)

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

Computes per-element difference between two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvSub` subtracts one array from another one

```
dst(I)=src1(I)-src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size)

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

Computes difference between array and scalar

**Parameters**
- *src* – The source array.
- *value* – Subtracted scalar.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvSubS` subtracts a scalar from every element of the source array

```
dst(I)=src(I)-value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size)

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

Computes difference between scalar and array

**Parameters**
- *src* – The first source array.
- *value* – Scalar to subtract from.
- *dst* – The destination array.

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

The function `cvSubRS` subtracts every element of source array from a scalar

```
dst(I)=value-src(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size)

void **cvMul**(*const CvArr* src1, const CvArr* src2, CvArr* dst, double scale=1*)
Calculates per-element product of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.
- *scale* – Optional scale factor

The function `cvMul` calculates per-element product of two arrays:

::dst(I)=scale?src1(I)?src2(I)

All the arrays must have the same type, and the same size (or ROI size)

void **cvDiv**(*const CvArr* src1, const CvArr* src2, CvArr* dst, double scale=1*)
Performs per-element division of two arrays

**Parameters**
- *src1* – The first source array. If the pointer is NULL, the array is assumed to be all 1?s.
- *src2* – The second source array.
- *dst* – The destination array.
- *scale* – Optional scale factor

The function `cvDiv` divides one array by another:

**::** dst(I)=scale?src1(I)/src2(I), if src1!=NULL dst(I)=scale/src2(I), if src1=NULL

All the arrays must have the same type, and the same size (or ROI size)

void **cvAnd**(*const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL*)
Calculates per-element bit-wise conjunction of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvAnd` calculates per-element bit-wise logical conjunction of two arrays

```
dst(I)=src1(I)&src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

void **cvAndS**(*const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL*)
Calculates per-element bit-wise conjunction of array and scalar

**Parameters**
- *src* – The source array.
- *value* – Scalar to use in the operation.
- *dst* – The destination array.

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

The function AndS calculates per-element bit-wise conjunction of array and scalar

```
dst(I)=src(I)&value if mask(I)!=0
```

Prior to the actual operation the scalar is converted to the same type as the arrays. In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

The following sample demonstrates how to calculate absolute value of floating-point array elements by clearing the most-significant bit

```
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] );
```

The code should print

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

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

Calculates per-element bit-wise disjunction of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvOr` calculates per-element bit-wise disjunction of two arrays

```
dst(I)=src1(I)|src2(I)
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

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

Calculates per-element bit-wise disjunction of array and scalar

**Parameters**
- *src1* – The source array.
- *value* – Scalar to use in the operation.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function OrS calculates per-element bit-wise disjunction of array and scalar

```
dst(I)=src(I)|value if mask(I)!=0
```

Prior to the actual operation the scalar is converted to the same type as the arrays. In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size `void` **cvXor** (*const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL*)

Performs per-element bit-wise "exclusive or" operation on two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function `cvXor` calculates per-element bit-wise logical conjunction of two arrays

```
dst(I)=src1(I)^src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

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

Performs per-element bit-wise "exclusive or" operation on array and scalar

**Parameters**
- *src* – The source array.
- *value* – Scalar to use in the operation.
- *dst* – The destination array.
- *mask* – Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function XorS calculates per-element bit-wise conjunction of array and scalar

```
dst(I)=src(I)^value if mask(I)!=0
```

Prior to the actual operation the scalar is converted to the same type as the arrays. In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

The following sample demonstrates how to conjugate complex vector by switching the most-significant bit of imaging part

```
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)
```

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

Performs per-element bit-wise inversion of array elements

**Parameters**
- *src1* – The source array.
- *dst* – The destination array.

The function Not inverses every bit of every array element

```
dst(I) =~ src(I)
```

void **cvCmp**(*const CvArr* src1, const CvArr* src2, CvArr* dst, int cmp_op*)
Performs per-element comparison of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array. Both source array must have a single channel.
- *dst* – The destination array, must have 8u or 8s type.
- *cmp_op* – The flag specifying the relation between the elements to be checked: - CV_CMP_EQ - src1(I) "equal to" src2(I) - CV_CMP_GT - src1(I) "greater than" src2(I) - CV_CMP_GE - src1(I) "greater or equal" src2(I) - CV_CMP_LT - src1(I) "less than" src2(I) - CV_CMP_LE - src1(I) "less or equal" src2(I) - CV_CMP_NE - src1(I) "not equal to" src2(I)

The function `cvCmp` compares the corresponding elements of two arrays and fills the destination mask array

```
dst(I)=src1(I) op src2(I),
```

where `op` is '=', '>', '>=', '<', '<=' or '!='.

`dst(I)` is set to 0xff (all '1'-bits) if the particular relation between the elements is true and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

void **cvCmpS**(*const CvArr* src, double value, CvArr* dst, int cmp_op*)
Performs per-element comparison of array and scalar

**Parameters**
- *src* – The source array, must have a single channel.
- *value* – The scalar value to compare each array element with.
- *dst* – The destination array, must have 8u or 8s type.
- *cmp_op* – The flag specifying the relation between the elements to be checked: - CV_CMP_EQ - src1(I) "equal to" value - CV_CMP_GT - src1(I) "greater than" value - CV_CMP_GE - src1(I) "greater or equal" value - CV_CMP_LT - src1(I) "less than" value - CV_CMP_LE - src1(I) "less or equal" value - CV_CMP_NE - src1(I) "not equal" value

The function `cvCmpS` compares the corresponding elements of array and scalar and fills the destination mask array

```
dst(I)=src(I) op scalar,
```

where `op` is '=', '>', '>=', '<', '<=' or '!='.

`dst(I)` is set to 0xff (all '1'-bits) if the particular relation between the elements is true and 0 otherwise. All the arrays must have the same size (or ROI size)

void **cvInRange**(*const CvArr* src, const CvArr* lower, const CvArr* upper, CvArr* dst*)
Checks that array elements lie between elements of two other arrays

**Parameters**
- *src* – The first source array.
- *lower* – The inclusive lower boundary array.
- *upper* – The exclusive upper boundary array.
- *dst* – The destination array, must have 8u or 8s type.

The function `cvInRange` does the range check for every element of the input array

```
dst(I)=lower(I)0 <= src(I)0 < upper(I)0
```

for single-channel arrays

```
dst(I)=lower(I)0 <= src(I)0 < upper(I)0 && lower(I)1 <= src(I)1 < upper(I)1
```

for two-channel arrays etc.

`dst(I)` is set to 0xff (all '1'-bits) if `src(I)` is within the range and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

`void` **cvInRangeS**(*const CvArr* src, CvScalar lower, CvScalar upper, CvArr* dst*)

Checks that array elements lie between two scalars

**Parameters**
- *src* – The first source array.
- *lower* – The inclusive lower boundary.
- *upper* – The exclusive upper boundary.
- *dst* – The destination array, must have 8u or 8s type.

The function `cvInRangeS` does the range check for every element of the input array

```
dst(I)=lower0 <= src(I)0 < upper0
```

for a single-channel array

```
dst(I)=lower0 <= src(I)0 < upper0 && lower1 <= src(I)1 < upper1
```

for a two-channel array etc.

`dst(I)` is set to 0xff (all '1'-bits) if `src(I)` is within the range and 0 otherwise. All the arrays must have the same size (or ROI size)

`void` **cvMax**(*const CvArr* src1, const CvArr* src2, CvArr* dst*)

Finds per-element maximum of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.

The function `cvMax` calculates per-element maximum of two arrays

```
dst(I)=max(src1(I), src2(I))
```

All the arrays must have a single channel, the same data type and the same size (or ROI size).

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

Finds per-element maximum of array and scalar

**Parameters**
- *src* – The first source array.
- *value* – The scalar value.
- *dst* – The destination array.

The function `cvMaxS` calculates per-element maximum of array and scalar

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

All the arrays must have a single channel, the same data type and the same size (or ROI size).

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

Finds per-element minimum of two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.

The function `cvMin` calculates per-element minimum of two arrays

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

All the arrays must have a single channel, the same data type and the same size (or ROI size).

void **cvMinS**(*const CvArr* src, double value, CvArr* dst*)
Finds per-element minimum of array and scalar

**Parameters**
- *src* – The first source array.
- *value* – The scalar value.
- *dst* – The destination array.

The function cvMinS calculates minimum of array and scalar

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

All the arrays must have a single channel, the same data type and the same size (or ROI size).

void **cvAbsDiff**(*const CvArr* src1, const CvArr* src2, CvArr* dst*)
Calculates absolute difference between two arrays

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array.

The function cvAbsDiff calculates absolute difference between two arrays

```
dst(I)c = abs(src1(I)c - src2(I)c).
```

All the arrays must have the same data type and the same size (or ROI size).

void **cvAbsDiffS**(*const CvArr* src, CvArr* dst, CvScalar value*)
Calculates absolute difference between array and scalar

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

**Parameters**
- *src* – The source array.
- *dst* – The destination array.
- *value* – The scalar.

The function cvAbsDiffS calculates absolute difference between array and scalar

```
dst(I)c = abs(src(I)c - valuec).
```

All the arrays must have the same data type and the same size (or ROI size).

## Statistics

int **cvCountNonZero**(*const CvArr* arr*)
Counts non-zero array elements

**Parameter** *arr* – The array, must be single-channel array or multi-channel image with COI set.

The function cvCountNonZero returns the number of non-zero elements in src1

$$result = \sum_I arr(I)! = 0$$

In case of IplImage both ROI and COI are supported.

`CvScalar` **cvSum**(*const CvArr* arr*)
Summarizes array elements

**Parameter** *arr* – The array.

The function `cvSum` calculates sum `S` of array elements, independently for each channel

```
Sc = sumI arr(I)c
```

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the sum to the first scalar component (S0).

`CvScalar` **cvAvg**(*const CvArr* arr, const CvArr* mask=NULL*)
Calculates average (mean) of array elements

**Parameters**
- *arr* – The array.
- *mask* – The optional operation mask.

The function `cvAvg` calculates the average value `M` of array elements, independently for each channel

```
N = sumI mask(I)!=0

Mc = 1/N ? sumI,mask(I)!=0 arr(I)c
```

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average to the first scalar component (S0).

`void` **cvAvgSdv**(*const CvArr* arr, CvScalar* mean, CvScalar* std_dev, const CvArr* mask=NULL*)
Calculates average (mean) of array elements

**Parameters**
- *arr* – The array.
- *mean* – Pointer to the mean value, may be NULL if it is not needed.
- *std_dev* – Pointer to the standard deviation.
- *mask* – The optional operation mask.

The function `cvAvgSdv` calculates the average value and standard deviation of array elements, independently for each channel

```
N = sumI mask(I)!=0

meanc = 1/N ? sumI,mask(I)!=0 arr(I)c

std_devc = sqrt(1/N ? sumI,mask(I)!=0 (arr(I)c - Mc)2)
```

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average and standard deviation to the first components of output scalars (M0 and S0).

`void` **cvMinMaxLoc**(*const CvArr* arr, double* min_val, double* max_val, CvPoint* min_loc=NULL, CvPoint* max_loc=NULL, const CvArr* mask=NULL*)
Finds global minimum and maximum in array or subarray

**Parameters**
- *arr* – The source array, single-channel or multi-channel with COI set.
- *min_val* – Pointer to returned minimum value.
- *max_val* – Pointer to returned maximum value.
- *min_loc* – Pointer to returned minimum location.
- *max_loc* – Pointer to returned maximum location.
- *mask* – The optional mask that is used to select a subarray.

The function `MinMaxLoc` finds minimum and maximum element values and their positions. The extremums are searched over the whole array, selected `ROI` (in case of `IplImage`) or, if `mask` is not `NULL`, in the specified array region. If the array has more than one channel, it must be `IplImage` with `COI` set. In case if multi-dimensional arrays `min_loc->x` and `max_loc->x` will contain raw (linear) positions of the extremums.

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

Calculates absolute array norm, absolute difference norm or relative difference norm

**Parameters**
- *arr1* – The first source image.
- *arr2* – The second source image. If it is NULL, the absolute norm of `arr1` is calculated, otherwise absolute or relative norm of `arr1-arr2` is calculated.
- *norm* – TypeType of norm, see the discussion.
- *mask* – The optional operation mask.

The function `cvNorm` calculates the absolute norm of `arr1` if `arr2` is NULL:

**::** norm = ||arr1||C = maxI abs(arr1(I)), if normType = CV_C

norm = ||arr1||L1 = sumI abs(arr1(I)), if normType = CV_L1

norm = ||arr1||L2 = sqrt( sumI arr1(I)2), if normType = CV_L2

And the function calculates absolute or relative difference norm if `arr2` is not NULL:

**::** norm = ||arr1-arr2||C = maxI abs(arr1(I)-arr2(I)), if normType = CV_C

norm = ||arr1-arr2||L1 = sumI abs(arr1(I)-arr2(I)), if normType = CV_L1

norm = ||arr1-arr2||L2 = sqrt( sumI (arr1(I)-arr2(I))2 ), if normType = CV_L2

or

norm = ||arr1-arr2||C/||arr2||C, if normType = CV_RELATIVE_C

norm = ||arr1-arr2||L1/||arr2||L1, if normType = CV_RELATIVE_L1

norm = ||arr1-arr2||L2/||arr2||L2, if normType = CV_RELATIVE_L2

The function‘‘ Norm‘‘ returns the calculated norm. The multiple-channel array are treated as single-channel, that is, the results for all channels are combined.

void **cvReduce**(*const CvArr* src, CvArr* dst, int op=CV_REDUCE_SUM*)

Reduces matrix to a vector

**Parameters**
- *src* – The input matrix.
- *dst* – The output single-row/single-column vector that accumulates somehow all the matrix rows/columns.
- *dim* – The dimension index along which the matrix is reduce. 0 means that the matrix is reduced to a single row, 1 means that the matrix is reduced to a single column. -1 means that the dimension is chosen automatically by analysing the `dst` size.
- *op* – The reduction operation. It can take of the following values: - `CV_REDUCE_SUM` - the output is the sum of all the matrix rows/columns. - `CV_REDUCE_AVG` - the output is the mean vector of all the matrix rows/columns. - `CV_REDUCE_MAX` - the output is the maximum (column/row-wise) of all the matrix rows/columns. - `CV_REDUCE_MIN` - the output is the minimum (column/row-wise) of all the matrix rows/columns.

The function `cvReduce` reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In case of `CV_REDUCE_SUM` and `CV_REDUCE_AVG` the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

### Linear Algebra

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

Calculates dot product of two arrays in Euclidean metrics

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.

The function `cvDotProduct` calculates and returns the Euclidean dot product of two arrays.

::src1?src2 = sumI(src1(I)*src2(I))

In case of multiple channel arrays the results for all channels are accumulated. In particular, `:cfunc:`cvDotProduct`(a,a)`, where `a` is a complex vector, will return `||a||2`. The function can process multi- dimensional arrays, row by row, layer by layer and so on.

void **cvNormalize**(*const CvArr* src, CvArr* dst, double a=1, double b=0, int norm_type=CV_L2, const CvArr* mask=NULL*)

Normalizes array to a certain norm or value range

**Parameters**
- *src* – The input array.
- *dst* – The output array; in-place operation is supported.
- *a* – The minimum/maximum value of the output array or the norm of output array.
- *b* – The maximum/minimum value of the output array.
- *norm_type* – The normalization type. It can take one of the following values: - `CV_C` - the C-norm (maximum of absolute values) of the array is normalized. - `CV_L1` - the L1-norm (sum of absolute values) of the array is normalized. - `CV_L2` - the (Euclidean) L2-norm of the array is normalized. - `CV_MINMAX` - the array values are scaled and shifted to the specified range.
- *mask* – The operation mask. Makes the function consider and normalize only certain array elements.

The function `cvNormalize` normalizes the input array so that it's norm or value range takes the certain value(s).

**When `norm_type==CV_MINMAX`: ::** dst(i,j)=(src(i,j)-min(src))*(b'-a')/(max(src)-min(src)) + a', if mask(i,j)!=0 dst(i,j)=src(i,j) otherwise

where `b'=MAX(a,b)`, `a'=MIN(a,b)`;

`min(src)` and `max(src)` are the global minimum and maximum, respectively, of the input array, computed over the whole array or the specified subset of it.

**When `norm_type!=CV_MINMAX`: ::** dst(i,j)=src(i,j)*a/:cfunc:'cvNorm'(src,0,norm_type,mask), if mask(i,j)!=0 dst(i,j)=src(i,j) otherwise

Here is the short example

```
float v[3] = { 1, 2, 3 };
CvMat V = cvMat( 1, 3, CV_32F, v );

// make vector v unit-length;
// equivalent to
//
// for (int i=0;i<3;i++)
//     v[i]/=sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
cvNormalize( &V, &V );
```

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

Calculates cross product of two 3D vectors

**Parameters**
- *src1* – The first source vector.
- *src2* – The second source vector.
- *dst* – The destination vector.

The function `cvCrossProduct` calculates the cross product of two 3D vectors:

::dst = src1 ? src2, (dst1 = src12src23 - src13src22 , dst2 = src13src21 - src11src23 , dst3 = src11src22 - src12src21).

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

Calculates sum of scaled array and another array

**::** #define cvMulAddS cvScaleAdd

**Parameters**
- *src1* – The first source array.
- *scale* – Scale factor for the first array.
- *src2* – The second source array.
- *dst* – The destination array

The function `cvScaleAdd` calculates sum of scaled array and another array:

::dst(I)=src1(I)*scale + src2(I)

All array parameters should have the same type and the same size.

`void` **`cvGEMM`**(*const CvArr* src1, const CvArr* src2, double alpha, const CvArr* src3, double beta, CvArr* dst, int tABC=0*)

Performs generalized matrix multiplication

**::** #define cvMatMulAdd( src1, src2, src3, dst ) cvGEMM( src1, src2, 1, src3, 1, dst, 0 ) #define cvMatMul( src1, src2, dst ) cvMatMulAdd( src1, src2, 0, dst )

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *src3* – The third source array (shift). Can be NULL, if there is no shift.
- *dst* – The destination array.
- *tABC* –

  **The operation flags that can be 0 or combination of the following values:** –
  CV_GEMM_A_T - transpose src1
  - CV_GEMM_B_T - transpose src2
  - CV_GEMM_C_T - transpose src3

  for example, CV_GEMM_A_T+CV_GEMM_C_T corresponds to ::alpha*src1T*src2 + beta*srcT

The function `cvGEMM` performs generalized matrix multiplication:

::dst = alpha*op(src1)*op(src2) + beta*op(src3), where op(X) is X or XT

All the matrices should have the same data type and the coordinated sizes. Real or complex floating-point matrices are supported

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

Performs matrix transform of every array element

**Parameters**
- *src* – The first source array.
- *dst* – The destination array.
- *transmat* – Transformation matrix.
- *shiftvec* – Optional shift vector.

The function `cvTransform` performs matrix transformation of every element of array `src` and stores the results in `dst`:

::dst(I)=transmat*src(I) + shiftvec or dst(I)k=sumj(transmat(k,j)*src(I)j) + shiftvec(k)

That is every element of `N`-channel array `src` is considered as `N`-element vector, which is transformed using matrix `M`?‘‘N‘‘ matrix `transmat` and shift vector `shiftvec` into an element of `M`-channel array `dst`. There is an option to embed `shiftvec` into `transmat`. In this case `transmat` should be `M`?‘‘N+1‘‘ matrix and the right-most column is treated as the shift vector.

Both source and destination arrays should have the same depth and the same size or selected ROI size. `transmat` and `shiftvec` should be real floating-point matrices.

The function may be used for geometrical transformation of ND point set, arbitrary linear color space transformation, shuffling the channels etc.

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

Performs perspective matrix transform of vector array

**Parameters**
- *src* – The source three-channel floating-point array.
- *dst* – The destination three- channel floating-point array. mat3?3 or 4?4 transformation matrix.

The function `cvPerspectiveTransform` transforms every element of `src` (by treating it as 2D or 3D vector) in the following way

```
(x, y, z) -> (x?/w, y?/w, z?/w) or
(x, y) -> (x?/w, y?/w),

where
(x?, y?, z?, w?) = mat4x4*(x, y, z, 1) or
(x?, y?, w?) = mat3x3*(x, y, 1)

and w = w?   if w?!=0,
        inf  otherwise
```

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

Calculates product of array and transposed array

**Parameters**
- *src* – The source matrix.
- *dst* – The destination matrix.
- *order* – Order of multipliers.
- *delta* – An optional array, subtracted from `src` before multiplication.

The function `cvMulTransposed` calculates the product of src and its transposition.

The function evaluates

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

if order=0, and ::dst=(src-delta)T*(src-delta)

otherwise.

`CvScalar` **cvTrace**(*const CvArr* mat*)

Returns trace of matrix

**Parameter** *mat* – The source matrix.

The function `cvTrace` returns sum of diagonal elements of the matrix `src1`.

```
tr(src1)=sumimat(i,i)
```

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

Transposes matrix

**::** #define cvT cvTranspose

**Parameters**
- *src* – The source matrix.
- *dst* – The destination matrix.

The function `cvTranspose` transposes matrix `src1`:

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

Note that no complex conjugation is done in case of complex matrix. Conjugation should be done separately: look at the sample code in `cvXorS` for example

double **cvDet**(*const CvArr* mat*)

Returns determinant of matrix

**Parameter** *mat* – The source matrix.

The function `cvDet` returns determinant of the square matrix `mat`. The direct method is used for small matrices and Gaussian elimination is used for larger matrices. For symmetric positive-determined matrices it is also possible to run `SVD` with `U=V=NULL` and then calculate determinant as a product of the diagonal elements of `W`

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

Finds inverse or pseudo-inverse of matrix

**::** #define cvInv cvInvert

**Parameters**
- *src* – The source matrix.
- *dst* – The destination matrix.
- *method* – Inversion method: - CV_LU - Gaussian elimination with optimal pivot element chose - CV_SVD - Singular value decomposition (SVD) method - CV_SVD_SYM - SVD method for a symmetric positively-defined matrix

The function `cvInvert` inverts matrix `src1` and stores the result in `src2`

In case of `LU` method the function returns `src1` determinant (src1 must be square). If it is 0, the matrix is not inverted and `src2` is filled with zeros.

In case of `SVD` methods the function returns the inverted condition number of `src1` (ratio of the smallest singular value to the largest singular value) and 0 if `src1` is all zeros. The SVD methods calculate a pseudo-inverse matrix if `src1` is singular

int **cvSolve**(*const CvArr* A, const CvArr* B, CvArr* X, int method=CV_LU*)

Solves linear system or least-squares problem

**Parameters**
- *A* – The source matrix.
- *B* – The right-hand part of the linear system.
- *X* – The output solution.
- *method* – The solution (matrix inversion) method: - CV_LU - Gaussian elimination with optimal pivot element chose - CV_SVD - Singular value decomposition (SVD) method - CV_SVD_SYM - SVD method for a symmetric positively-defined matrix.

The function `cvSolve` solves linear system or least-squares problem (the latter is possible with SVD methods)

```
dst = arg minX||A*X-B||
```

If `CV_LU` method is used, the function returns 1 if `src1` is non- singular and 0 otherwise, in the latter case `dst` is not valid

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

Performs singular value decomposition of real floating-point matrix

**Parameters**
- *A* – Source `M`?‘‘N‘‘ matrix.
- *W* – Resulting singular value matrix (`M`?‘‘N‘‘ or `N`?‘‘N‘‘) or vector (`N`?‘‘1‘‘).
- *U* – Optional left orthogonal matrix (`M`?‘‘M‘‘ or `M`?‘‘N‘‘). If CV_SVD_U_T is specified, the number of rows and columns in the sentence above should be swapped.
- *V* – Optional right orthogonal matrix (`N`?‘‘N‘‘)
- *flags* – Operation flags; can be 0 or combination of the following values:
  - `CV_SVD_MODIFY_A` enables modification of matrix `src1` during the operation. It speeds up the processing.
  - `CV_SVD_U_T` means that the transposed matrix `U` is returned. Specifying the flag speeds up the processing.
  - `CV_SVD_V_T` means that the transposed matrix `V` is returned. Specifying the flag speeds up the processing.

The function `cvSVD` decomposes matrix `A` into a product of a diagonal matrix and two orthogonal matrices:

**::** A=U*W*VT

Where `W` is diagonal matrix of singular values that can be coded as a 1D vector of singular values and `U` and `V`. All the singular values are non-negative and sorted (together with `U` and and `V` columns) in descending order.

SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix `A` is square, symmetric and positively defined matrix, for example, when it is a covariation matrix. `W` in this case will be a vector of eigenvalues, and `U``=``V` is matrix of eigenvectors (thus, only one of `U` or `V` needs to be calculated if the eigenvectors are required)
- accurate solution of poor-conditioned linear systems
- least-squares solution of overdetermined linear systems. This and previous is done by `cvSolve` function with `CV_SVD` method
- accurate calculation of different matrix characteristics such as rank (number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), determinant (absolute value of determinant is equal to the product of singular values). All the things listed in this item do not require calculation of `U` and `V` matrices.

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

Performs singular value back substitution

**Parameters**
- *W* – Matrix or vector of singular values.
- *U* – Left orthogonal matrix (transposed, perhaps)
- *V* – Right orthogonal matrix (transposed, perhaps)
- *B* – The matrix to multiply the pseudo-inverse of the original matrix `A` by. This is the optional parameter. If it is omitted then it is assumed to be an identity matrix of an appropriate size (So `X` will be the reconstructed pseudo- inverse of `A`).
- *X* – The destination matrix: result of back substitution.
- *flags* – Operation flags, should match exactly to the `flags` passed to `cvSVD`.

The function `cvSVBkSb` calculates back substitution for decomposed matrix `A` (see `cvSVD` description) and matrix `B`:

**::** X=V*W-1*UT*B

Where

**:: W-1(i,i)=1/W(i,i) if W(i,i) > epsilon?sumiW(i,i),** 0 otherwise

And `epsilon` is a small number that depends on the matrix data type.

This function together with `cvSVD` is used inside `cvInvert` and `cvSolve`, and the possible reason to use these (svd & bksb) "low-level" function is to avoid temporary matrices allocation inside the high-level counterparts (inv & solve).

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

Computes eigenvalues and eigenvectors of symmetric matrix

**Parameters**

- *mat* – The input symmetric square matrix. It is modified during the processing.
- *evects* – The output matrix of eigenvectors, stored as a subsequent rows.
- *evals* – The output vector of eigenvalues, stored in the descending order (order of eigenvalues and eigenvectors is synchronized, of course).
- *eps* – Accuracy of diagonalization (typically, DBL_EPSILON=?10-15 is enough).

The function `cvEigenVV` computes the eigenvalues and eigenvectors of the matrix `A`:

::mat*evects(i,:)' = evals(i)*evects(i,:)' (in MATLAB notation)

The contents of matrix `A` is destroyed by the function.

Currently the function is slower than `cvSVD` yet less accurate, so if `A` is known to be positively-defined (for example, it is a covariation matrix), it is recommended to use `cvSVD` to find eigenvalues and eigenvectors of `A`, especially if eigenvectors are not required. That is, instead of

```
cvEigenVV(mat, eigenvals, eigenvects);
 call ::
cvSVD(mat, eigenvals, eigenvects, 0, CV_SVD_U_T + CV_SVD_MODIFY_A);
```

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

Calculates covariation matrix of the set of vectors

**Parameters**

- *vects* – The input vectors. They all must have the same type and the same size. The vectors do not have to be 1D, they can be 2D (e.g. images) etc.
- *count* – The number of input vectors.
- *cov_mat* – The output covariation matrix that should be floating-point and square.
- *avg* – The input or output (depending on the flags) array - the mean (average) vector of the input vectors.
- *flags* – The operation flags, a combination of the following values: - `CV_COVAR_SCRAMBLED` - the output covariation matrix is calculated as:

  `scale*[vects[0]-avg,vects[1]-avg,...]T*[vects[0]-avg,vects[1]-avg,...]`, that is, the covariation matrix is `count?``count``. Such an unusual covariation matrix is used for fast PCA of a set of very large vectors (see, for example, Eigen Faces technique for face recognition). Eigenvalues of this "scrambled" matrix will match to the eigenvalues of the true covariation matrix and the "true" eigenvectors can be easily calculated from the eigenvectors of the "scrambled" covariation matrix.

  – `CV_COVAR_NORMAL` - the output covariation matrix is calculated as: `scale*[vects[0]-avg,vects[1]-avg,...]*[vects[0]-avg,vects[1]-avg,...]T`, that is, `cov_mat` will be a usual covariation matrix with the same linear size as the total number of elements in every input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

- CV_COVAR_USE_AVG - if the flag is specified, the function does not calculate avg from the input vectors, but, instead, uses the passed avg vector. This is useful if avg has been already calculated somehow, or if the covariation matrix is calculated by parts - in this case, avg is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.
- CV_COVAR_SCALE - if the flag is specified, the covariation matrix is scaled by the number of input vectors. CV_COVAR_ROWS - Means that all the input vectors are stored as rows of a single matrix, vects[0]. count is ignored in this case, and avg should be a single-row vector of an appropriate size. CV_COVAR_COLS - Means that all the input vectors are stored as columns of a single matrix, vects[0]. count is ignored in this case, and avg should be a single-column vector of an appropriate size.

The function cvCalcCovarMatrix calculates the covariation matrix and, optionally, mean vector of the set of input vectors. The function can be used for PCA, for comparing vectors using Mahalanobis distance etc.

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

Calculates Mahalonobis distance between two vectors

**Parameters**
- *vec1* – The first 1D source vector.
- *vec2* – The second 1D source vector.
- *mat* – The inverse covariation matrix.

The function cvMahalonobis calculates the weighted distance between two vectors and returns it:

**::** d(vec1,vec2)=sqrt( sumi,j {mat(i,j)*(vec1(i)-vec2(i))*(vec1(j)-vec2(j))} )

The covariation matrix may be calculated using cvCalcCovarMatrix function and further inverted using cvInvert function (CV_SVD method is the preferred one, because the matrix might be singular).

void **cvCalcPCA**(*const CvArr* data, CvArr* avg, CvArr* eigenvalues, CvArr* eigenvectors, int flags*)

Performs Principal Component Analysis of a vector set

**Parameters**
- *data* – The input data; each vector is either a single row (CV_PCA_DATA_AS_ROW) or a single column (CV_PCA_DATA_AS_COL).
- *avg* – The mean (average) vector, computed inside the function or provided by user.
- *eigenvalues* – The output eigenvalues of covariation matrix.
- *eigenvectors* – The output eigenvectors of covariation matrix (i.e. principal components); one vector per row.
- *flags* – The operation flags, a combination of the following values: - CV_PCA_DATA_AS_ROW - the vectors are stored as rows (i.e. all the

  components of a certain vector are stored continuously)

  - CV_PCA_DATA_AS_COL - the vectors are stored as columns (i.e. values of a certain vector component are stored continuously) (the above two flags are mutually exclusive)
  - CV_PCA_USE_AVG - use pre-computed average vector

The function cvCalcPCA performs PCA analysis of the vector set. First, it uses cvCalcCovarMatrix to compute covariation matrix and then it finds its eigenvalues and eigenvectors. The output number of eigenvalues/eigenvectors should be less than or equal to MIN(rows(data),cols(data)).

**void cvProjectPCA( const CvArr* data, const CvArr* avg, const CvArr* eigenvectors, CvArr***

Projects vectors to the specified subspace

**Parameters**
- *data* – The input data; each vector is either a single row or a single column.
- *avg* – The mean (average) vector. If it is a single-row vector, it means that the input vectors are stored as rows of data; otherwise, it should be a single-column vector, then the vectors are stored as columns of data.
- *eigenvectors* – The eigenvectors (principal components); one vector per row.

- *result* – The output matrix of decomposition coefficients. The number of rows must be the same as the number of vectors, the number of columns must be less than or equal to the number of rows in `eigenvectors`. That it is less, the input vectors are projected into subspace of the first `cols(result)` principal components.

The function `cvProjectPCA` projects input vectors to the subspace represented by the orthonormal basis (`eigenvectors`). Before computing the dot products, `avg` vector is subtracted from the input vectors:

**::** result(i,:)=(data(i,:)-avg)*eigenvectors' // for CV_PCA_DATA_AS_ROW layout.

`void` **`cvBackProjectPCA`**(*const CvArr* proj, const CvArr* avg, const CvArr* eigenvects, CvArr* result*)

Reconstructs the original vectors from the projection coefficients

**Parameters**
- *proj* – The input data; in the same format as `result` in `cvProjectPCA`.
- *avg* – The mean (average) vector. If it is a single-row vector, it means that the output vectors are stored as rows of `result`; otherwise, it should be a single-column vector, then the vectors are stored as columns of `result`.
- *eigenvectors* – The eigenvectors (principal components); one vector per row.
- *result* – The output matrix of reconstructed vectors.

The function `cvBackProjectPCA` reconstructs the vectors from the projection coefficients:

**::** result(i,:)=proj(i,:)*eigenvectors + avg // for CV_PCA_DATA_AS_ROW layout.

### Math Functions

`int` **`cvRound`**(*double value*)

`int` **`cvFloor`**(*double value*)

`int` **`cvCeil`**(*double value*)

Converts floating-point number to integer

**Parameter** *value* – The input floating-point value

The functions `cvRound, cvFloor` and `cvCeil` convert input floating- point number to integer using one of the rounding modes. `cvRound` returns the nearest integer value to the argument. `cvFloor` returns the maximum integer value that is not larger than the argument. `cvCeil` returns the minimum integer value that is not smaller than the argument. On some architectures the functions work *much* faster than the standard cast operations in C. If absolute value of the argument is greater than 231, the result is not determined. Special values (?Inf, NaN) are not handled.

`float` **`cvSqrt`**(*float value*)

Calculates square root

**Parameter** *value* – The input floating-point value

The function `cvSqrt` calculates square root of the argument. If the argument is negative, the result is not determined.

`float` **`cvInvSqrt`**(*float value*)

Calculates inverse square root

**Parameter** *value* – The input floating-point value

The function `cvInvSqrt` calculates inverse square root of the argument, and normally it is faster than `1./sqrt(value)`. If the argument is zero or negative, the result is not determined. Special values (?Inf, NaN) are not handled.

`float` **`cvCbrt`** (*float value*)
Calculates cubic root

**Parameter** *value* – The input floating-point value

The function `cvCbrt` calculates cubic root of the argument, and normally it is faster than `pow(value,1./3)`. Besides, negative arguments are handled properly. Special values (?Inf, NaN) are not handled.

`float` **`cvFastArctan`** (*float y, float x*)
Calculates angle of 2D vector

**Parameters**
- *x* – x-coordinate of 2D vector
- *y* – y-coordinate of 2D vector

The function `cvFastArctan` calculates full-range angle of input 2D vector. The angle is measured in degrees and varies from 0? to 360?. The accuracy is ~0.1?

`int` **`cvIsNaN`** (*double value*)
Determines if the argument is Not A Number

**Parameter** *value* – The input floating-point value

The function `cvIsNaN` returns 1 if the argument is Not A Number (as defined by IEEE754 standard), 0 otherwise.

`int` **`cvIsInf`** (*double value*)
Determines if the argument is Infinity

**Parameter** *value* – The input floating-point value

The function `cvIsInf` returns 1 if the argument is ?Infinity (as defined by IEEE754 standard), 0 otherwise.

`void` **`cvCartToPolar`** (*const CvArr* x, const CvArr* y, CvArr* magnitude, CvArr* angle=NULL, int angle_in_degrees=0*)
Calculates magnitude and/or angle of 2d vectors

**Parameters**
- *x* – The array of x-coordinates
- *y* – The array of y-coordinates
- *magnitude* – The destination array of magnitudes, may be set to NULL if it is not needed
- *angle* – The destination array of angles, may be set to NULL if it is not needed. The angles are measured in radians (0..2?) or in degrees (0..360?).
- *angle_in_degrees* – The flag indicating whether the angles are measured in radians, which is default mode, or in degrees.

The function `cvCartToPolar` calculates either magnitude, angle, or both of every 2d vector `(x(I),y(I))`:

**::** magnitude(I)=sqrt( x(I)2+y(I)2 ), angle(I)=atan( y(I)/x(I) )

The angles are calculated with ?0.1? accuracy. For (0,0) point the angle is set to 0.

`void` **`cvPolarToCart`** (*const CvArr* magnitude, const CvArr* angle, CvArr* x, CvArr* y, int angle_in_degrees=0*)
Calculates Cartesian coordinates of 2d vectors represented in polar form

**Parameters**
- *magnitude* – The array of magnitudes. If it is NULL, the magnitudes are assumed all 1?s.
- *angle* – The array of angles, whether in radians or degrees.
- *x* – The destination array of x-coordinates, may be set to NULL if it is not needed.
- *y* – The destination array of y-coordinates, may be set to NULL if it is not needed.

- *angle_in_degrees* – The flag indicating whether the angles are measured in radians, which is default mode, or in degrees.

The function `cvPolarToCart` calculates either x-coordinate, y-coordinate or both of every vector `magnitude(I)*exp(angle(I)*j), j=sqrt(-1)`:

**::** x(I)=magnitude(I)*cos(angle(I)), y(I)=magnitude(I)*sin(angle(I))

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

Raises every array element to power

**Parameters**
- *src* – The source array.
- *dst* – The destination array, should be the same type as the source.
- *power* – The exponent of power.

The function `cvPow` raises every element of input array to `p`:

**::** dst(I)=src(I)p, if p is integer dst(I)=abs(src(I))p, otherwise

That is, for non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following sample, computing cube root of array elements, shows

```
CvSize size = cvGetSize(src);
CvMat* mask = cvCreateMat( size.height, size.width, CV_8UC1 );
cvCmpS( src, 0, mask, CV_CMP_LT ); /* find negative elements */
cvPow( src, dst, 1./3 );
cvSubRS( dst, cvScalarAll(0), dst, mask ); /* negate the results of negative inputs */
cvReleaseMat( &mask );
```

For some values of `power`, such as integer values, 0.5 and -0.5, specialized faster algorithms are used.

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

Calculates exponent of every array element

**Parameters**
- *src* – The source array.
- *dst* – The destination array, it should have `double` type or the same type as the source.

The function `cvExp` calculates exponent of every element of input array:

**::** dst(I)=exp(src(I))

Maximum relative error is ?7e-6. Currently, the function converts denormalized values to zeros on output.

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

Calculates natural logarithm of every array element absolute value

**Parameters**
- *src* – The source array.
- *dst* – The destination array, it should have `double` type or the same type as the source.

The function `cvLog` calculates natural logarithm of absolute value of every element of input array:

**::** dst(I)=log(abs(src(I))), src(I)!=0 dst(I)=C, src(I)=0

Where `C` is large negative number (?-700 in the current implementation)

`int` **`cvSolveCubic`**(*const CvMat* coeffs, CvMat* roots*)

Finds real roots of a cubic equation

**Parameters**
- *coeffs* – The equation coefficients, array of 3 or 4 elements.
- *roots* – The output array of real roots. Should have 3 elements.

The function `cvSolveCubic` finds real roots of a cubic equation:

**::** coeffs[0]*x3 + coeffs[1]*x2 + coeffs[2]*x + coeffs[3] = 0 (if coeffs is 4-element vector)

or

x3 + coeffs[0]*x2 + coeffs[1]*x + coeffs[2] = 0 (if coeffs is 3-element vector)

The function returns the number of real roots found. The roots are stored to `root` array, which is padded with zeros if there is only one root.

`void` **`cvSolvePoly`**(*const CvMat* coeffs, CvMat* roots, int maxiter = 10, int fig = 10*)

Finds real and complex roots of a polynomial equation with real coefficients

**Parameters**
- *coeffs* – The (degree + 1)-length array of equation coefficients (CV_32FC1 or CV_64FC1).
- *roots* – The degree-length output array of real or complex roots (CV_32FC2 or CV_64FC2).
- *maxiter* – The maximum number of iterations.
- *fig* – The required figures of precision required.

The function `cvSolvePoly` finds all real and complex roots of any degree polynomial with real coefficients.

## Random Number Generation

`CvRNG` **`cvRNG`**(*int64 seed=-1*)

Initializes random number generator state

**Parameter** *seed* – 64-bit value used to initiate a random sequence.

The function `cvRNG` initializes random number generator and returns the state. Pointer to the state can be then passed to `cvRandInt`, `cvRandReal` and `cvRandArr` functions. In the current implementation a multiply-with- carry generator is used.

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

Fills array with random numbers and updates the RNG state

**Parameters**
- *rng* – RNG state initialized by `cvRNG`.
- *arr* – The destination array.
- *dist_type* – Distribution type: - `CV_RAND_UNI` - uniform distribution - `CV_RAND_NORMAL` - normal or Gaussian distribution
- *param1* – The first parameter of distribution. In case of uniform distribution it is the inclusive lower boundary of random numbers range. In case of normal distribution it is the mean value of random numbers.
- *param2* – The second parameter of distribution. In case of uniform distribution it is the exclusive upper boundary of random numbers range. In case of normal distribution it is the standard deviation of random numbers.

The function `cvRandArr` fills the destination array with uniformly or normally distributed random numbers. In the sample below the function is used to add a few normally distributed floating-point numbers to random locations within a 2d array

**::** /* let 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, /* use dummy parameters now and adjust them further / *0xffffffff* / just use a fixed seed here /*, CV_RAND_UNI* / specify uniform type */ );

/* initialize the locations using a uniform distribution */ cvRandArr( &rng_state, locations, CV_RAND_UNI, cvScalar(0,0,0,0), cvScalar(size.width,size.height,0,0) );

/* generate normally distributed random 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 */

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

Returns 32-bit unsigned integer and updates RNG

rngRNG state initialized by `cvRandInit` and, optionally, customized by `cvRandSetRange` (though, the latter function does not affect on the discussed function outcome).

The function `cvRandInt` returns uniformly-distributed random 32-bit unsigned integer and updates RNG state. It is similar to rand() function from C runtime library, but it always generates 32-bit number whereas rand() returns a number in between 0 and `RAND_MAX` which is 2**16 or 2**32, depending on the platform.

The function is useful for generating scalar random numbers, such as points, patch sizes, table indices etc, where integer numbers of a certain range can be generated using modulo operation and floating-point numbers can be generated by scaling to 0..1 of any other specific range. Here is the example from the previous function discussion rewritten using `cvRandInt`:

**::** /* 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 */

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

Returns floating-point random number and updates RNG

rngRNG state initialized by `cvRNG`.

The function `cvRandReal` returns uniformly-distributed random floating- point number from 0..1 range (1 is not included).

### Discrete Transforms

void **cvDFT**(*const CvArr* src, CvArr* dst, int flags, int nonzero_rows=0*)

Performs forward or inverse Discrete Fourier transform of 1D or 2D floating-point array

```
#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
```

**Parameters**
- *src* – Source array, real or complex.
- *dst* – Destination array of the same size and same type as the source.
- *flags* – Transformation flags, a combination of the following values:
  - `CV_DXT_FORWARD` - do forward 1D or 2D transform. The result is not scaled.
  - `CV_DXT_INVERSE` - do inverse 1D or 2D transform. The result is not scaled.
  - `CV_DXT_FORWARD` and `CV_DXT_INVERSE` are mutually exclusive, of course.
  - `CV_DXT_SCALE` - scale the result: divide it by the number of array elements. Usually, it is combined with `CV_DXT_INVERSE`, and one may use a shortcut `CV_DXT_INV_SCALE`.
  - `CV_DXT_ROWS` - do forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher- dimensional transforms etc.
- *nonzero_rows* – Number of nonzero rows to in the source array (in case of forward 2d transform), or a number of rows of interest in the destination array (in case of inverse 2d transform). If the value is negative, zero, or greater than the total number of rows, it is ignored. The parameter can be used to speed up 2d convolution/correlation when computing them via DFT. See the sample below.

The function `cvDFT` performs forward or inverse transform of 1D or 2D floating-point array:

```
Forward Fourier transform of 1D vector of N elements:
y = F(N)?x, where F(N)jk=exp(-i?2Pi?j?k/N), i=sqrt(-1)

Inverse Fourier transform of 1D vector of N elements:
x'= (F(N))-1?y = conj(F(N))?y
x = (1/N)?x

Forward Fourier transform of 2D vector of M?N elements:
Y = F(M)?X?F(N)

Inverse Fourier transform of 2D vector of M?N elements:
X'= conj(F(M))?Y?conj(F(N))
X = (1/(M?N))?X'
```

In case of real (single-channel) data, the packed format, borrowed from IPL, is used to to represent a result of forward Fourier transform or input for inverse Fourier transform:

```
Re Y0,0      Re Y0,1    Im Y0,1    Re Y0,2     Im Y0,2  ...  Re
Y0,N/2-1   Im Y0,N/2-1  Re Y0,N/2
Re Y1,0      Re Y1,1    Im Y1,1    Re Y1,2     Im Y1,2  ...  Re
Y1,N/2-1   Im Y1,N/2-1  Re Y1,N/2
```

```
Im Y1,0      Re Y2,1    Im Y2,1    Re Y2,2     Im Y2,2  ...  Re
Y2,N/2-1   Im Y2,N/2-1  Im Y2,N/2
.....................................................................
......................
Re YM/2-1,0   Re YM-3,1   Im YM-3,1  Re YM-3,2   Im YM-3,2 ...  Re
YM-3,N/2-1  Im YM-3,N/2-1 Re YM-3,N/2
Im YM/2-1,0   Re YM-2,1   Im YM-2,1  Re YM-2,2   Im YM-2,2 ...  Re
YM-2,N/2-1  Im YM-2,N/2-1 Im YM-2,N/2
Re YM/2,0    Re YM-1,1   Im YM-1,1  Re YM-1,2   Im YM-1,2  ... Re
YM-1,N/2-1  Im YM-1,N/2-1 Im YM-1,N/2
```

Note: the last column is present if `N` is even, the last row is present if `M` is even.

In case of 1D real transform the result looks like the first row of the above matrix

Computing 2D Convolution using DFT

```
CvMat* A = cvCreateMat( M1, N1, CV_32F );
CvMat* B = cvCreateMat( M2, N2, A->type );

// it is also possible to have only abs(M2-M1)+1?abs(N2-N1)+1
// part of the full convolution result
CvMat* conv = cvCreateMat( A->rows + B->rows - 1, A->cols +
B->cols - 1, A->type );

// initialize A and B
...

int dft_M = cvGetOptimalDFTSize( A->rows + B->rows - 1 );
int dft_N = cvGetOptimalDFTSize( A->cols + B->cols - 1 );

CvMat* dft_A = cvCreateMat( dft_M, dft_N, A->type );
CvMat* dft_B = cvCreateMat( dft_M, dft_N, B->type );
CvMat tmp;

// copy A to dft_A and pad dft_A with zeros
cvGetSubRect( dft_A, &tmp, cvRect(0,0,A->cols,A->rows));
cvCopy( A, &tmp );
cvGetSubRect( dft_A, &tmp, cvRect(A->cols,0,dft_A->cols - A->cols,A->rows));
cvZero( &tmp );
// no need to pad bottom part of dft_A with zeros because of
// use nonzero_rows parameter in cvDFT() call below

cvDFT( dft_A, dft_A, CV_DXT_FORWARD, A->rows );

// repeat the same with the second array
cvGetSubRect( dft_B, &tmp, cvRect(0,0,B->cols,B->rows));
cvCopy( B, &tmp );
cvGetSubRect( dft_B, &tmp, cvRect(B->cols,0,dft_B->cols -
B->cols,B->rows));
cvZero( &tmp );
// no need to pad bottom part of dft_B with zeros because of
// use nonzero_rows parameter in cvDFT() call below

cvDFT( dft_B, dft_B, CV_DXT_FORWARD, B->rows );

cvMulSpectrums( dft_A, dft_B, dft_A, 0 /* or CV_DXT_MUL_CONJ to get correlation rather than conv

cvDFT( dft_A, dft_A, CV_DXT_INV_SCALE, conv->rows ); // calculate only the top part
```

```
cvGetSubRect( dft_A, &tmp, cvRect(0,0,conv->cols,conv->rows) );

cvCopy( &tmp, conv );
```

int **cvGetOptimalDFTSize**(*int size0*)

Returns optimal DFT size for given vector size

**Parameter** *size0* – Vector size.

The function `cvGetOptimalDFTSize` returns the minimum number `N` that is greater to equal to `size0`, such that DFT of a vector of size `N` can be computed fast. In the current implementation `N=2p?3q?5r` for some `p, q, r`.

The function returns a negative number if `size0` is too large (very close to `INT_MAX`)

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

Performs per-element multiplication of two Fourier spectrums

**Parameters**
- *src1* – The first source array.
- *src2* – The second source array.
- *dst* – The destination array of the same type and the same size of the sources.
- *flags* – A combination of the following values: - `CV_DXT_ROWS` - treat each row of the arrays as a

  separate spectrum (see `cvDFT` parameters description).

  – `CV_DXT_MUL_CONJ` - conjugate the second source array before the multiplication.

The function `cvMulSpectrums` performs per-element multiplication of the two CCS-packed or complex matrices that are results of real or complex Fourier transform.

The function, together with `cvDFT`, may be used to calculate convolution of two arrays fast.

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

Performs forward or inverse Discrete Cosine transform of 1D or 2D floating-point array

**::** #define CV_DXT_FORWARD 0 #define CV_DXT_INVERSE 1 #define CV_DXT_ROWS 4

**Parameters**
- *src* – Source array, real 1D or 2D array.
- *dst* – Destination array of the same size and same type as the source.
- *flags* – Transformation flags, a combination of the following values: - `CV_DXT_FORWARD` - do forward 1D or 2D transform. - `CV_DXT_INVERSE` - do inverse 1D or 2D transform. - `CV_DXT_ROWS` - do forward or inverse transform of every individual row of

  the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher- dimensional transforms etc.

The function `cvDCT` performs forward or inverse transform of 1D or 2D floating-point array:

$$ForwardCosinetransformof1DvectorofNelements: y = C(N)?x, whereC(N)jk = sqrt((j == 0?1:2)/N)?cos(Pi?($$
$$InverseCosinetransformof1DvectorofNelements: x = (C(N)) - 1?y = (C(N))T?y$$
$$ForwardCosinetransformof2DvectorofM?Nelements: Y = (C(M))?X?(C(N))T$$
$$InverseCosinetransformof2DvectorofM?Nelements: X = (C(M))T?Y?C(N)$$

### 1.1.3 Dynamic Structures

#### Memory Storages

**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;
```

Memory storage is a low-level structure used to store dynamically growing data structures such as sequences, contours, graphs, subdivisions etc. It is organized as a list of memory blocks of equal size - `bottom` field is the beginning of the list of blocks‘‘ and `top` is the currently used block, but not necessarily the last block of the list. All blocks between `bottom` and `top`, not including the latter, are considered fully occupied; and all blocks between `top` and the last block, not including `top`, are considered free and `top` block itself is partly occupied - `free_space` contains the number of free bytes left in the end of `top`.

New memory buffer that may be allocated explicitly by `cvMemStorageAlloc` function or implicitly by higher-level functions, such as `cvSeqPush`, `cvGraphAddEdge` etc., `always` starts in the end of the current block if it fits there. After allocation `free_space` is decremented by the size of the allocated buffer plus some padding to keep the proper alignment. When the allocated buffer does not fit into the available part of `top`, the next storage block from the list is taken as `top` and `free_space` is reset to the whole block size prior to the allocation.

If there is no more free blocks, a new block is allocated (or borrowed from parent, see `cvCreateChildMemStorage`) and added to the end of list. Thus, the storage behaves as a stack with `bottom` indicating bottom of the stack and the pair (`top`, `free_space`) indicating top of the stack. The stack top may be saved via `cvSaveMemStoragePos`, restored via `cvRestoreMemStoragePos` or reset via `cvClearStorage`.

**CvMemBlock**

Memory storage block

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

The structure `CvMemBlock` represents a single block of memory storage. Actual data of the memory blocks follows the header, that is, the `i-th` byte of the memory block can be retrieved with the expression `((char*)(mem_block_ptr+1))[i]`. However, normally there is no need to access the storage structure fields directly.

**CvMemStoragePos**

Memory storage position

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
```

```
    int free_space;
} CvMemStoragePos;
```

The structure described below stores the position of the stack top that can be saved via cvSaveMemStoragePos and restored via cvRestoreMemStoragePos.

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

Creates memory storage

**Parameter** *block_size* – Size of the storage blocks in bytes. If it is 0, the block size is set to default value - currently it is ?64K.

The function cvCreateMemStorage creates a memory storage and returns pointer to it. Initially the storage is empty. All fields of the header, except the block_size, are set to 0.

CvMemStorage* **cvCreateChildMemStorage**(*CvMemStorage* parent*)

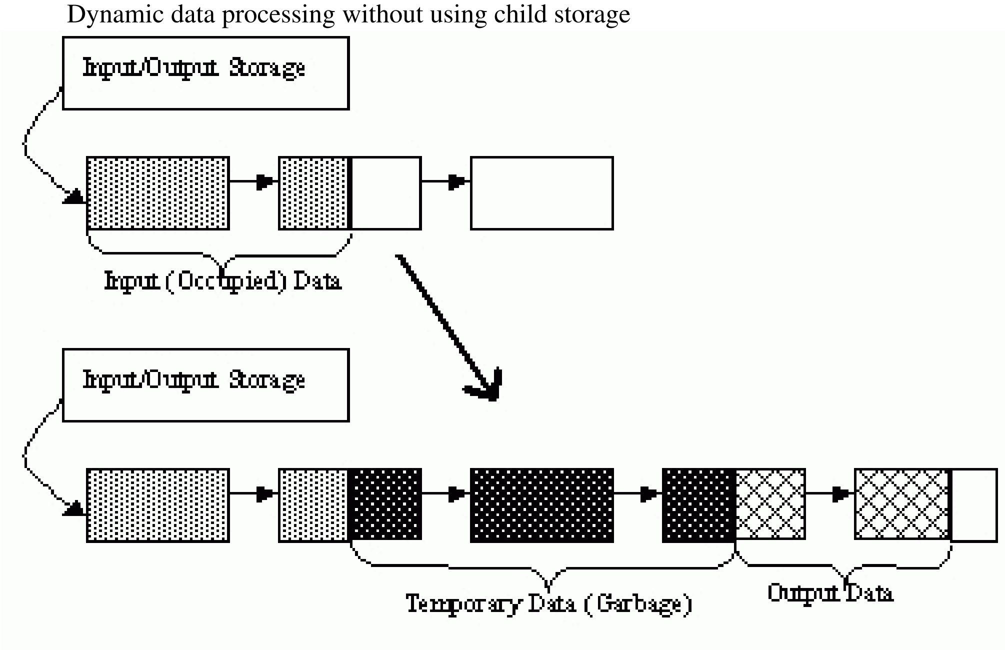
Creates child memory storage

**Parameter** *parent* – Parent memory storage.

The function cvCreateChildMemStorage creates a child memory storage that is similar to simple memory storage except for the differences in the memory allocation/deallocation mechanism. When a child storage needs a new block to add to the block list, it tries to get this block from the parent. The first unoccupied parent block available is taken and excluded from the parent block list. If no blocks are available, the parent either allocates a block or borrows one from its own parent, if any. In other words, the chain, or a more complex structure, of memory storages where every storage is a child/parent of another is possible. When a child storage is released or even cleared, it returns all blocks to the parent. In other aspects, the child storage is the same as the simple storage.

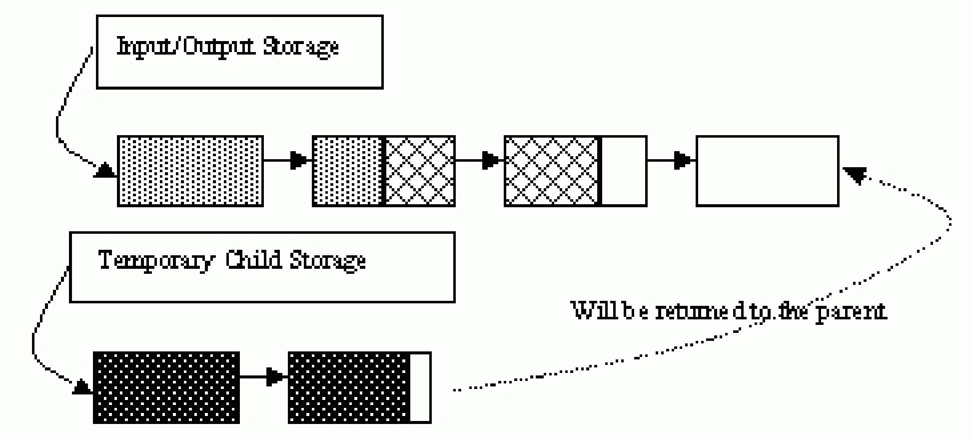
The children storages are useful in the following situation. Imagine that user needs to process dynamical data resided in some storage and put the result back to the same storage. With the simplest approach, when temporary data is resided in the same storage as the input and output data, the storage will look as following after processing:

Dynamic data processing without using child storage




That is, garbage appears in the middle of the storage. However, if one creates a child memory storage in the beginning of the processing, writes temporary data there and releases the child storage in the end, no garbage will appear in the source/destination storage:

Dynamic data processing using a child storage




void **cvReleaseMemStorage** (*CvMemStorage** storage*)
Releases memory storage

**Parameter** *storage* – Pointer to the released storage.

The function `cvReleaseMemStorage` deallocates all storage memory blocks or returns them to the parent, if any. Then it deallocates the storage header and clears the pointer to the storage. All children of the storage must be released before the parent is released.

void **cvClearMemStorage** (*CvMemStorage* storage*)
Clears memory storage

**Parameter** *storage* – Memory storage.

The function `cvClearMemStorage` resets the top (free space boundary) of the storage to the very beginning. This function does not deallocate any memory. If the storage has a parent, the function returns all blocks to the parent.

void* **cvMemStorageAlloc** (*CvMemStorage* storage, size_t size*)
Allocates memory buffer in the storage

**Parameters**
- *storage* – Memory storage.
- *size* – Buffer size.

The function `cvMemStorageAlloc` allocates memory buffer in the storage. The buffer size must not exceed the storage block size, otherwise runtime error is raised. The buffer address is aligned by `CV_STRUCT_ALIGN` (=‘‘sizeof(double)‘‘ for the moment) bytes.

CvString **cvMemStorageAllocString** (*CvMemStorage* storage, const char* ptr, int len=-1*)
Allocates text string in the storage

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

**Parameters**
- *storage* – Memory storage.
- *ptr* – The string.
- *len* – Length of the string (not counting the ending ‘0’). If the parameter is negative, the function computes the length.

The function cvMemStorageAllocString creates copy of the string in the memory storage. It returns the structure that contains user-passed or computed length of the string and pointer to the copied string.

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

Saves memory storage position

**Parameters**
- *storage* – Memory storage.
- *pos* – The output position of the storage top.

The function cvSaveMemStoragePos saves the current position of the storage top to the parameter pos. The function cvRestoreMemStoragePos can further retrieve this position.

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

Restores memory storage position

**Parameters**
- *storage* – Memory storage.
- *pos* – New storage top position.

The function cvRestoreMemStoragePos restores the position of the storage top from the parameter pos. This function and The function cvClearMemStorage are the only methods to release memory occupied in memory blocks. Note again that there is no way to free memory in the middle of the occupied part of the storage.

### Sequences

#### CvSeq

Growable sequence of elements

```
#define CV_SEQUENCE_FIELDS() \
    int flags; /* miscellaneous 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;
```

The structure CvSeq is a base for all of OpenCV dynamic data structures.

Such an unusual definition via a helper macro simplifies the extension of the structure CvSeq with additional parameters. To extend CvSeq the user may define a new structure and put user-defined fields after all CvSeq fields that are included via the macro CV_SEQUENCE_FIELDS().

There are two types of sequences - dense and sparse. Base type for dense sequences is CvSeq and such sequences are used to represent growable 1d arrays - vectors, stacks, queues, deques. They have no gaps in the middle - if an element is removed from the middle or inserted into the middle of the sequence the elements from the closer end are shifted. Sparse sequences have CvSet base class and they are discussed later in more details. They are sequences of nodes each of those may be either occupied or free as indicated by the node flag. Such sequences are used for unordered data structures such as sets of elements, graphs, hash tables etc.

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

The fields h_prev, h_next, v_prev, v_next can be used to create hierarchical structures from separate sequences. The fields h_prev and h_next point to the previous and the next sequences on the same hierarchical level while the fields v_prev and v_next point to the previous and the next sequence in the vertical direction, that is, parent and its first child. But these are just names and the pointers can be used in a different way.

The field first points to the first sequence block, whose structure is described below.

The field total contains the actual number of dense sequence elements and number of allocated nodes in sparse sequence.

The field flags contain the particular dynamic type signature (CV_SEQ_MAGIC_VAL for dense sequences and CV_SET_MAGIC_VAL for sparse sequences) in the highest 16 bits and miscellaneous information about the sequence. The lowest CV_SEQ_ELTYPE_BITS bits contain the ID of the element type. Most of sequence processing functions do not use element type but element size stored in elem_size. If sequence contains the numeric data of one of CvMat type then the element type matches to the corresponding CvMat element type, e.g. CV_32SC2 may be used for sequence of 2D points, CV_32FC1 for sequences of floating-point values etc. CV_SEQ_ELTYPE(seq_header_ptr) macro retrieves the type of sequence elements. Processing function that work with numerical sequences check that elem_size is equal to the calculated from the type element size. Besides CvMat compatible types, there are few extra element types defined in cvtypes.h header:

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)  */
```

The next CV_SEQ_KIND_BITS bits specify the kind of the sequence:

Standard Kinds of Sequences

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

/* dense sequence subtypes */
#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)
```

The remaining bits are used to identify different features specific to certain sequence kinds and element types. For example, curves made of points ( `CV_SEQ_KIND_CURVE|CV_SEQ_ELTYPE_POINT` ), together with the flag `CV_SEQ_FLAG_CLOSED` belong to the type `CV_SEQ_POLYGON` or, if other flags are used, to its subtype. Many contour processing functions check the type of the input sequence and report an error if they do not support this type. The file `cvtypes.h` stores the complete list of all supported predefined sequence types and helper macros designed to get the sequence type of other properties. Below follows the definition of the building block of sequences.

**CvSeqBlock**

Continuous sequence block

```
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;
```

Sequence blocks make up a circular double-linked list, so the pointers `prev` and `next` are never `NULL` and point to the previous and the next sequence blocks within the sequence. It means that `next` of the last block is the first block and `prev` of the first block is the last block. The fields `start_index` and `count` help to track the block location within the sequence. For example, if the sequence consists of 10 elements and splits into three blocks of 3, 5, and 2 elements, and the first block has the parameter `start_index = 2`, then pairs `(start_index, count)` for the sequence blocks are `(2,3), (5, 5)`, and `(10, 2)` correspondingly. The parameter `start_index` of the first block is usually `0` unless some elements have been inserted at the beginning of the sequence.

**CvSlice**

A sequence slice

```
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 );
```

Some of functions that operate on sequences take `CvSlice slice` parameter that is often set to the whole sequence (CV_WHOLE_SEQ) by default. Either of the `start_index` and `end_index` may be negative or exceed the sequence length, `start_index` is inclusive, `end_index` is exclusive boundary. If they are equal, the slice is considered empty (i.e. contains no elements). Because sequences are treated as circular structures, the slice may select a few elements in the end of a sequence followed by a few elements in the beginning of the sequence, for example, `cvSlice(-2, 3)` in case of 10-element sequence will select 5-element slice, containing pre-last (8th), last (9th), the very first (0th), second (1th) and third (2nd) elements. The functions normalize the slice argument in the following way: first, `cvSliceLength` is called to determine the length of the slice, then, `start_index` of the slice is normalized similarly to the argument of `cvGetSeqElem` (i.e. negative indices are allowed). The actual slice to process starts at the normalized `start_index` and lasts `cvSliceLength` elements (again, assuming the sequence is a circular structure).

If a function does not take slice argument, but you want to process only a part of the sequence, the sub-sequence may be extracted using `cvSeqSlice` function, or stored as into a continuous buffer with `cvCvtSeqToArray` (optionally, followed by `cvMakeSeqHeaderForArray`.

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

Creates sequence

**Parameters**
- *seq_flags* – Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be set to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.
- *header_size* – Size of the sequence header; must be greater or equal to `sizeof(CvSeq)`. If a specific type or its extension is indicated, this type must fit the base type header.
- *elem_size* – Size of the sequence elements in bytes. The size must be consistent with the sequence type. For example, for a sequence of points to be created, the element type `CV_SEQ_ELTYPE_POINT` should be specified and the parameter `elem_size` must be equal to `sizeof(CvPoint)`.
- *storage* – Sequence location.

The function `cvCreateSeq` creates a sequence and returns the pointer to it. The function allocates the sequence header in the storage block as one continuous chunk and sets the structure fields `flags`, `elem_size`, `header_size` and `storage` to passed values, sets `delta_elems` to the default value (that may be reassigned using `cvSetSeqBlockSize` function), and clears other header fields, including the space after the first `sizeof(CvSeq)` bytes.

`void` **`cvSetSeqBlockSize`**`(`*CvSeq* seq, int delta_elems*`)`

Sets up sequence block size

**Parameters**
- *seq* – Sequence.
- *delta_elems* – Desirable sequence block size in elements.

The function `cvSetSeqBlockSize` affects memory allocation granularity. When the free space in the sequence buffers has run out, the function allocates the space for `delta_elems` sequence elements. If this block immediately follows the one previously allocated, the two blocks are concatenated, otherwise, a new sequence block is created. Therefore, the bigger the parameter is, the lower the possible sequence fragmentation, but the more space in the storage is wasted. When the sequence is created, the parameter `delta_elems` is set to the default value ?1K. The function can be called any time after the sequence is created and affects future allocations. The function can modify the passed value of the parameter to meet the memory storage constraints.

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

Adds element to sequence end

**Parameters**
- *seq* – Sequence.
- *element* – Added element.

The function `cvSeqPush` adds an element to the end of sequence and returns pointer to the allocated element. If the input `element` is NULL, the function simply allocates a space for one more element.

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 );
```

The function `cvSeqPush` has O(1) complexity, but there is a faster method for writing large sequences (see `cvStartWriteSeq` and related functions).

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

Removes element from sequence end

**Parameters**
- *seq* – Sequence.
- *element* – Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

The function `cvSeqPop` removes an element from the sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

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

Adds element to sequence beginning

**Parameters**
- *seq* – Sequence.
- *element* – Added element.

The function `cvSeqPushFront` is similar to `cvSeqPush` but it adds the new element to the beginning of the sequence. The function has O(1) complexity.

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

Removes element from sequence beginning

**Parameters**
- *seq* – Sequence.
- *element* – Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

The function `cvSeqPopFront` removes an element from the beginning of the sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

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

Pushes several elements to the either end of sequence

**Parameters**
- *seq* – Sequence.
- *elements* – Added elements.
- *count* – Number of elements to push.
- *in_front* – The flags specifying the modified sequence end:
  - CV_BACK (=0) - the elements are added to the end of sequence
  - CV_FRONT(!=0) - the elements are added to the beginning of sequence

The function `cvSeqPushMulti` adds several elements to either end of the sequence. The elements are added to the sequence in the same order as they are arranged in the input array but they can fall into different sequence blocks.

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

Removes several elements from the either end of sequence

**Parameters**
- *seq* – Sequence.
- *elements* – Removed elements.
- *count* – Number of elements to pop.
- *in_front* – The flags specifying the modified sequence end:
  - CV_BACK (=0) - the elements are removed from the end of sequence
  - CV_FRONT(!=0) - the elements are removed from the beginning of sequence

The function `cvSeqPopMulti` removes several elements from either end of the sequence. If the number of the elements to be removed exceeds the total number of elements in the sequence, the function removes as many elements as possible.

`char*` **`cvSeqInsert`**(*CvSeq* seq, int before_index, void* element=NULL*)
Inserts element in sequence middle

**Parameters**
- *seq* – Sequence.
- *before_index* – Index before which the element is inserted. Inserting before 0 (the minimal allowed value of the parameter) is equal to `cvSeqPushFront` and inserting before `seq->total` (the maximal allowed value of the parameter) is equal to `cvSeqPush`.
- *element* – Inserted element.

The function `cvSeqInsert` shifts the sequence elements from the inserted position to the nearest end of the sequence and copies the `element` content there if the pointer is not NULL. The function returns pointer to the inserted element.

`void` **`cvSeqRemove`**(*CvSeq* seq, int index*)
Removes element from sequence middle

**Parameters**
- *seq* – Sequence.
- *index* – Index of removed element.

The function `cvSeqRemove` removes elements with the given index. If the index is out of range the function reports an error. An attempt to remove an element from an empty sequence is a partial case of this situation. The function removes an element by shifting the sequence elements between the nearest end of the sequence and the `index`-th position, not counting the latter.

`void` **`cvClearSeq`**(*CvSeq* seq*)
Clears sequence

**Parameter** *seq* – Sequence.

The function `cvClearSeq` removes all elements from the sequence. The function does not return the memory to the storage, but this memory is reused later when new elements are added to the sequence. This function time complexity is `O(1)`.

`char*` **`cvGetSeqElem`**(*const CvSeq* seq, int index*)
Returns pointer to sequence element by its index

```
#define CV_GET_SEQ_ELEM( TYPE, seq, index )  (TYPE*)cvGetSeqElem(
(CvSeq*)(seq), (index) )
```

**Parameters**
- *seq* – Sequence.
- *index* – Index of element.

The function `cvGetSeqElem` finds the element with the given index in the sequence and returns the pointer to it. If the element is not found, the function returns 0. The function supports negative indices, where -1 stands for the last sequence element, -2 stands for the one before last, etc. If the sequence is most likely to consist of a single sequence block or the desired element is likely to be located in the first block, then the macro

`CV_GET_SEQ_ELEM( elemType, seq, index )` should be used, where the parameter `elemType` is the type of sequence elements ( `CvPoint` for example), the parameter `seq` is a sequence, and the parameter `index` is the index of the desired element. The macro checks first whether the desired element belongs to the first block of the sequence and returns it if it does, otherwise the macro calls the main function `GetSeqElem`. Negative indices always cause the `cvGetSeqElem` call. The function has O(1) time complexity assuming that number of blocks is much smaller than the number of elements.

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

Returns index of concrete sequence element

**Parameters**
- *seq* – Sequence.
- *element* – Pointer to the element within the sequence.
- *block* – Optional argument. If the pointer is not `NULL`, the address of the sequence block that contains the element is stored in this location.

The function `cvSeqElemIdx` returns the index of a sequence element or a negative number if the element is not found.

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

Copies sequence to one continuous block of memory

**Parameters**
- *seq* – Sequence.
- *elements* – Pointer to the destination array that must be large enough. It should be a pointer to data, not a matrix header.
- *slice* – The sequence part to copy to the array.

The function `cvCvtSeqToArray` copies the entire sequence or subsequence to the specified buffer and returns the pointer to the buffer.

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

Constructs sequence from array

**Parameters**
- *seq_type* – Type of the created sequence.
- *header_size* – Size of the header of the sequence. Parameter sequence must point to the structure of that size or greater size.
- *elem_size* – Size of the sequence element.
- *elements* – Elements that will form a sequence.
- *total* – Total number of elements in the sequence. The number of array elements must be equal to the value of this parameter.
- *seq* – Pointer to the local variable that is used as the sequence header.
- *block* – Pointer to the local variable that is the header of the single sequence block.

The function `cvMakeSeqHeaderForArray` initializes sequence header for array. The sequence header as well as the sequence block are allocated by the user (for example, on stack). No data is copied by the function. The resultant sequence will consists of a single block and have NULL storage pointer, thus, it is possible to read its elements, but the attempts to add elements to the sequence will raise an error in most cases.

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

Makes separate header for the sequence slice

**Parameters**
- *seq* – Sequence.
- *slice* – The part of the sequence to extract.
- *storage* – The destination storage to keep the new sequence header and the copied data if any. If it is NULL, the function uses the storage containing the input sequence.
- *copy_data* – The flag that indicates whether to copy the elements of the extracted slice (`copy_data!=0`) or not (`copy_data=0`)

The function `cvSeqSlice` creates a sequence that represents the specified slice of the input sequence. The new sequence either shares the elements with the original sequence or has own copy of the elements. So if one needs to process a part of sequence but the processing function does not have a slice parameter, the required sub-sequence may be extracted using this function.

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

Creates a copy of sequence

**Parameters**
- *seq* – Sequence.
- *storage* – The destination storage to keep the new sequence header and the copied data if any. If it is NULL, the function uses the storage containing the input sequence.

The function `cvCloneSeq` makes a complete copy of the input sequence and returns it. The call `:cfunc:`cvCloneSeq`( seq, storage )` is equivalent to `:cfunc:`cvSeqSlice`( seq, CV_WHOLE_SEQ, storage, 1 )`

`void` **cvSeqRemoveSlice**(*CvSeq* seq, CvSlice slice*)

Removes sequence slice

**Parameters**
- *seq* – Sequence.
- *slice* – The part of the sequence to remove.

The function `cvSeqRemoveSlice` removes slice from the sequence.

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

Inserts array in the middle of sequence

**Parameters**
- *seq* – Sequence.
- *slice* – The part of the sequence to remove.
- *from_arr* – The array to take elements from.

The function `cvSeqInsertSlice` inserts all `from_arr` array elements at the specified position of the sequence. The array `from_arr` can be a matrix or another sequence.

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

Reverses the order of sequence elements

**Parameter** *seq* – Sequence.

The function `cvSeqInvert` reverses the sequence in-place - makes the first element go last, the last element go first etc.

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

Sorts sequence element using the specified comparison function

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

**Parameter** *seq* – The sequence to sort funcThe comparison function that returns negative, zero or positive value depending on the elements relation (see the above declaration and the example below) - similar function is used by `qsort` from C runtime except that in the latter `userdata` is not used userdataThe user parameter passed to the comparison function; helps to avoid global variables in some cases.

The function `cvSeqSort` sorts the sequence in-place using the specified criteria. Below is the example of the function use

```
/* 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 );
```

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

Searches element in sequence

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

**Parameters**
- *seq* – The sequence elemThe element to look for funcThe comparison function that returns negative, zero or positive value depending on the elements relation (see also `cvSeqSort`).
- *is_sorted* – Whether the sequence is sorted or not.
- *elem_idx* – Output parameter; index of the found element.
- *userdata* – The user parameter passed to the comparison function; helps to avoid global variables in some cases.

The function `cvSeqSearch` searches the element in the sequence. If the sequence is sorted, binary O(log(N)) search is used, otherwise, a simple linear search is used. If the element is not found, the function returns NULL pointer and the index is set to the number of sequence elements if the linear search is used, and to the smallest index `i, seq(i)>elem`.

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

Initializes process of writing data to sequence

**Parameters**
- *seq* – Pointer to the sequence.
- *writer* – Writer state; initialized by the function.

The function `cvStartAppendToSeq` initializes the process of writing data to the sequence. Written elements are added to the end of the sequence by `CV_WRITE_SEQ_ELEM( written_elem, writer )` macro. Note that during the writing process other operations on the sequence may yield incorrect result or even corrupt the sequence (see description of `cvFlushSeqWriter` that helps to avoid some of these problems).

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

Creates new sequence and initializes writer for it

**Parameters**
- *seq_flags* – Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be equal to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.
- *header_size* – Size of the sequence header. The parameter value may not be less than `sizeof(CvSeq)`. If a certain type or extension is specified, it must fit the base type header.
- *elem_size* – Size of the sequence elements in bytes; must be consistent with the sequence type. For example, if the sequence of points is created (element type `CV_SEQ_ELTYPE_POINT` ), then the parameter elem_size must be equal to `sizeof(CvPoint)`.
- *storage* – Sequence location.
- *writer* – Writer state; initialized by the function.

The function `cvStartWriteSeq` is a composition of `cvCreateSeq` and `cvStartAppendToSeq`. The pointer to the created sequence is stored at `writer->seq` and is also returned by `cvEndWriteSeq` function that should be called in the end.

`CvSeq*` **cvEndWriteSeq**(*CvSeqWriter* writer*)

Finishes process of writing sequence

**Parameter** *writer* – Writer state

The function `cvEndWriteSeq` finishes the writing process and returns the pointer to the written sequence. The function also truncates the last incomplete sequence block to return the remaining part of the block to the memory storage. After that the sequence can be read and modified safely.

`void` **cvFlushSeqWriter**(*CvSeqWriter* writer*)

Updates sequence headers from the writer state

**Parameter** *writer* – Writer state

The function `cvFlushSeqWriter` is intended to enable the user to read sequence elements, whenever required, during the writing process, e.g., in order to check specific conditions. The function updates the sequence headers to make reading from the sequence possible. The writer is not closed, however, so that the writing process can be continued any time. If some algorithm requires often flushes, consider using `cvSeqPush` instead.

`void` **cvStartReadSeq**(*const CvSeq* seq, CvSeqReader* reader, int reverse=0*)

Initializes process of sequential reading from sequence

**Parameters**
- *seq* – Sequence.
- *reader* – Reader state; initialized by the function.
- *reverse* – Determines the direction of the sequence traversal. If `reverse` is 0, the reader is positioned at the first sequence element, otherwise it is positioned at the last element.

The function `cvStartReadSeq` initializes the reader state. After that all the sequence elements from the first down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM( read_elem, reader )` in case of forward reading and by using `CV_REV_READ_SEQ_ELEM( read_elem, reader )` in case of reversed reading. Both macros put the sequence element to `read_elem` and move the reading pointer toward the next element. A circular structure of sequence blocks is used for the reading process, that is, after the last element has been read by the macro `CV_READ_SEQ_ELEM`, the first element is read when the macro is called again. The same applies to `CV_REV_READ_SEQ_ELEM ``. There is no function to finish the reading process, since it neither changes the sequence nor creates any temporary buffers. The reader field ``ptr` points to the current element of the sequence that is to be read next.

The code below demonstrates how to use sequence writer and reader

```
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 preferable 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 );
```

`int` **`cvGetSeqReaderPos`**(*CvSeqReader* reader*)

Returns the current reader position

**Parameter** *reader* – Reader state.

The function `cvGetSeqReaderPos` returns the current reader position (within 0 ... `reader->seq->total` - 1).

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

Moves the reader to specified position

**Parameters** • *reader* – Reader state.

- *index* – The destination position. If the positioning mode is used (see the next parameter) the actual position will be `index` mod `reader->seq->total`.
- *is_relative* – If it is not zero, then `index` is a relative to the current position.

The function `cvSetSeqReaderPos` moves the read position to the absolute position or relative to the current position.

## 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 :ctype:`CvSeq` */ \
    struct CvSetElem* free_elems; /* list of free nodes
    */

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

The structure `CvSet` is a base for OpenCV sparse data structures.

As follows from the above declaration `CvSet` inherits from `CvSeq` and it adds `free_elems` field it to, which is a list of free nodes. Every set node, whether free or not, is the element of the underlying sequence. While there is no restrictions on elements of dense sequences, the set (and derived structures) elements must start with integer field and be able to fit CvSetElem structure, because these two fields (integer followed by the pointer) are required for organization of node set with the list of free nodes. If a node is free, `flags` field is negative (the most-significant bit, or MSB, of the field is set), and `next_free` points to the next free node (the first free node is referenced by `free_elems` field of `CvSet`). And if a node is occupied, `flags` field is positive and contains the node index that may be retrieved using (set_elem->flags & CV_SET_ELEM_IDX_MASK) expression, the rest of the node content is determined by the user. In particular, the occupied nodes are not linked as the free nodes are, so the second field can be used for such a link as well as for some different purpose. The macro `CV_IS_SET_ELEM(set_elem_ptr)` can be used to determined whether the specified node is occupied or not.

Initially the set and the list are empty. When a new node is requested from the set, it is taken from the list of free nodes, which is updated then. If the list appears to be empty, a new sequence block is allocated and all the nodes within the block are joined in the list of free nodes. Thus, `total` field of the set is the total number of nodes both occupied and free. When an occupied node is released, it is added to the list of free nodes. The node released last will be occupied first.

In OpenCV `CvSet` is used for representing graphs (`CvGraph`), sparse multi-dimensional arrays (`CvSparseMat`), planar subdivisions (`CvSubdiv2D`) etc.

`CvSet*` **`cvCreateSet`**(*int set_flags, int header_size, int elem_size, CvMemStorage* storage*)
Creates empty set

**Parameters**
- *set_flags* – Type of the created set.
- *header_size* – Set header size; may not be less than `sizeof(CvSet)`.
- *elem_size* – Set element size; may not be less than `CvSetElem`.
- *storage* – Container for the set.

The function `cvCreateSet` creates an empty set with a specified header size and element size, and returns the pointer to the set. The function is just a thin layer on top of `cvCreateSeq`.

`int` **`cvSetAdd`**(*CvSet* set_header, CvSetElem* elem=NULL, CvSetElem** inserted_elem=NULL*)
Occupies a node in the set

**Parameters**
- *set_header* – Set.
- *elem* – Optional input argument, inserted element. If not NULL, the function copies the data to the allocated node (The MSB of the first integer field is cleared after copying).
- *inserted_elem* – Optional output argument; the pointer to the allocated cell.

The function `cvSetAdd` allocates a new node, optionally copies input element data to it, and returns the pointer and the index to the node. The index value is taken from the lower bits of `flags` field of the node. The function has O(1) complexity, however there exists a faster function for allocating set nodes (see `cvSetNew`).

`void` **`cvSetRemove`**(*CvSet* set_header, int index*)
Removes element from set

**Parameters**
- *set_header* – Set.
- *index* – Index of the removed element.

The function `cvSetRemove` removes an element with a specified index from the set. If the node at the specified location is not occupied the function does nothing. The function has O(1) complexity, however, `cvSetRemoveByPtr` provides yet faster way to remove a set element if it is located already.

`CvSetElem*` **`cvSetNew`**(*CvSet* set_header*)
Adds element to set (fast variant)

**Parameter** *set_header* – Set.

The function `cvSetNew` is inline light-weight variant of `cvSetAdd`. It occupies a new node and returns pointer to it rather than index.

`void` **`cvSetRemoveByPtr`**(*CvSet* set_header, void* elem*)
Removes set element given its pointer

**Parameters**
- *set_header* – Set.
- *elem* – Removed element.

The function `cvSetRemoveByPtr` is inline light-weight variant of `cvSetRemove` that takes element pointer. The function does not check whether the node is occupied or not - the user should take care of it.

`CvSetElem*` **`cvGetSetElem`**(*const CvSet* set_header, int index*)
Finds set element by its index

**Parameters**
- *set_header* – Set.
- *index* – Index of the set element within a sequence.

The function `cvGetSetElem` finds a set element by index. The function returns the pointer to it or 0 if the index is invalid or the corresponding node is free. The function supports negative indices as it uses `cvGetSeqElem` to locate the node.

void **cvClearSet**(*CvSet* set_header*)

Clears set

**Parameter** *set_header* – Cleared set.

The function `cvClearSet` removes all elements from set. It has O(1) time complexity.

## Graphs

### CvGraph

Oriented or undirected weighted graph

```
#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;
```

The structure `CvGraph` is a base for graphs used in OpenCV.

Graph structure inherits from `CvSet` - this part describes common graph properties and the graph vertices, and contains another set as a member - this part describes the graph edges.

The vertex, edge and the graph header structures are declared using the same technique as other extendible OpenCV structures - via macros, that simplifies extension and customization of the structures. While the vertex and edge structures do not inherit from `CvSetElem` explicitly, they satisfy both conditions on the set elements - have an integer field in the beginning and fit CvSetElem structure. The `flags` fields are used as for indicating occupied vertices and edges as well as for other purposes, for example, for graph traversal (see `cvCreateGraphScanner` et al.), so it is better not to use them directly.

The graph is represented as a set of edges each of whose has the list of incident edges. The incidence lists for different vertices are interleaved to avoid information duplication as much as possible.

The graph may be oriented or undirected. In the latter case there is no distinction between edge connecting vertex A with vertex B and the edge connecting vertex B with vertex A - only one of them can exist in the graph at the same moment and it represents both <A, B> and <B, A> edges..

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

Creates empty graph

**Parameters**
- *graph_flags* – Type of the created graph. Usually, it is either `CV_SEQ_KIND_GRAPH` for generic undirected graphs and `CV_SEQ_KIND_GRAPH | CV_GRAPH_FLAG_ORIENTED` for generic oriented graphs.
- *header_size* – Graph header size; may not be less than `sizeof(CvGraph)`. vtx_sizeGraph vertex size; the custom vertex structure must start with `CvGraphVtx` (use `CV_GRAPH_VERTEX_FIELDS()`) edge_sizeGraph edge size; the custom edge structure must start with `CvGraphEdge` (use `CV_GRAPH_EDGE_FIELDS()`)
- *storage* – The graph container.

The function `cvCreateGraph` creates an empty graph and returns pointer to it.

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

Adds vertex to graph

**Parameters**
- *graph* – Graph.
- *vtx* – Optional input argument used to initialize the added vertex (only user-defined fields beyond `sizeof(CvGraphVtx)` are copied).
- *inserted_vertex* – Optional output argument. If not `NULL`, the address of the new vertex is written there.

The function `cvGraphAddVtx` adds a vertex to the graph and returns the vertex index.

`int` **`cvGraphRemoveVtx`**`(`*CvGraph* graph, int index*`)`

Removes vertex from graph

**Parameters**
- *graph* – Graph.
- *vtx_idx* – Index of the removed vertex.

The function `cvGraphRemoveAddVtx` removes a vertex from the graph together with all the edges incident to it. The function reports an error, if the input vertex does not belong to the graph. The return value is number of edges deleted, or -1 if the vertex does not belong to the graph.

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

Removes vertex from graph

**Parameters**
- *graph* – Graph.
- *vtx* – Pointer to the removed vertex.

The function `cvGraphRemoveVtxByPtr` removes a vertex from the graph together with all the edges incident to it. The function reports an error, if the vertex does not belong to the graph. The return value is number of edges deleted, or -1 if the vertex does not belong to the graph.

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

Finds graph vertex by index

**Parameters**
- *graph* – Graph.
- *vtx_idx* – Index of the vertex.

The function `cvGetGraphVtx` finds the graph vertex by index and returns the pointer to it or NULL if the vertex does not belong to the graph.

int **cvGraphVtxIdx** (*CvGraph* graph, CvGraphVtx* vtx*)
Returns index of graph vertex

**Parameters**
- *graph* – Graph.
- *vtx* – Pointer to the graph vertex.

The function `cvGraphVtxIdx` returns index of the graph vertex.

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

**Parameters**
- *graph* – Graph.
- *start_idx* – Index of the starting vertex of the edge.
- *end_idx* – Index of the ending vertex of the edge. For undirected graph the order of the vertex parameters does not matter.
- *edge* – Optional input parameter, initialization data for the edge.
- *inserted_edge* – Optional output parameter to contain the address of the inserted edge.

The function `cvGraphAddEdge` connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e. when the result is negative) the function also reports an error by default.

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

**Parameters**
- *graph* – Graph.
- *start_vtx* – Pointer to the starting vertex of the edge.
- *end_vtx* – Pointer to the ending vertex of the edge. For undirected graph the order of the vertex parameters does not matter.
- *edge* – Optional input parameter, initialization data for the edge.
- *inserted_edge* – Optional output parameter to contain the address of the inserted edge within the edge set.

The function `cvGraphAddEdge` connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e. when the result is negative) the function also reports an error by default.

void **cvGraphRemoveEdge** (*CvGraph* graph, int start_idx, int end_idx*)
Removes edge from graph

**Parameters**
- *graph* – Graph.
- *start_idx* – Index of the starting vertex of the edge.
- *end_idx* – Index of the ending vertex of the edge. For undirected graph the order of the vertex parameters does not matter.

The function `cvGraphRemoveEdge` removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

void **cvGraphRemoveEdgeByPtr** (*CvGraph* graph, CvGraphVtx* start_vtx, CvGraphVtx* end_vtx*)
Removes edge from graph

**Parameters**
- *graph* – Graph.
- *start_vtx* – Pointer to the starting vertex of the edge.

- *end_vtx* – Pointer to the ending vertex of the edge. For undirected graph the order of the vertex parameters does not matter.

The function `cvGraphRemoveEdgeByPtr` removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

CvGraphEdge* **cvFindGraphEdge**(*const CvGraph* graph, int start_idx, int end_idx*)
Finds edge in graph

```
#define cvGraphFindEdge cvFindGraphEdge
```

**Parameters**
- *graph* – Graph.
- *start_idx* – Index of the starting vertex of the edge.
- *end_idx* – Index of the ending vertex of the edge. For undirected graph the order of the vertex parameters does not matter.

The function `cvFindGraphEdge` finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

CvGraphEdge* **cvFindGraphEdgeByPtr**(*const CvGraph* graph, const CvGraphVtx* start_vtx, const CvGraphVtx* end_vtx*)
Finds edge in graph

```
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

**Parameters**
- *graph* – Graph.
- *start_vtx* – Pointer to the starting vertex of the edge.
- *end_vtx* – Pointer to the ending vertex of the edge. For undirected graph the order of the vertex parameters does not matter.

The function `cvFindGraphEdge` finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

int **cvGraphEdgeIdx**(*CvGraph* graph, CvGraphEdge* edge*)
Returns index of graph edge

**Parameters**
- *graph* – Graph.
- *edge* – Pointer to the graph edge.

The function `cvGraphEdgeIdx` returns index of the graph edge.

int **cvGraphVtxDegree**(*const CvGraph* graph, int vtx_idx*)
Counts edges incident to the vertex

**Parameters**
- *graph* – Graph.
- *vtx* – Index of the graph vertex.

The function `cvGraphVtxDegree` returns the number of edges incident to the specified vertex, both incoming and outgoing. To count the edges, the following code is used

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

The macro `CV_NEXT_GRAPH_EDGE( edge, vertex )` returns the edge incident to `vertex` that follows after `edge`.

`int` **`cvGraphVtxDegreeByPtr`**(*const CvGraph* graph, const CvGraphVtx* vtx*)
Finds edge in graph

**Parameters**
- *graph* – Graph.
- *vtx* – Pointer to the graph vertex.

The function `cvGraphVtxDegree` returns the number of edges incident to the specified vertex, both incoming and outgoing.

`void` **`cvClearGraph`**(*CvGraph* graph*)
Clears graph

**Parameter** *graph* – Graph.

The function `cvClearGraph` removes all vertices and edges from the graph. The function has O(1) time complexity.

`CvGraph*` **`cvCloneGraph`**(*const CvGraph* graph, CvMemStorage* storage*)
Clone graph

**Parameters**
- *graph* – The graph to copy.
- *storage* – Container for the copy.

The function `cvCloneGraph` creates full copy of the graph. If the graph vertices or edges have pointers to some external data, it still be shared between the copies. The vertex and edge indices in the new graph may be different from the original, because the function defragments the vertex and edge sets.

**`CvGraphScanner`**
Graph traversal state

```
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;
```

The structure `CvGraphScanner` is used for depth-first graph traversal. See discussion of the functions below.

`CvGraphScanner*` **`cvCreateGraphScanner`**(*CvGraph* graph, CvGraphVtx* vtx=NULL, int mask=CV_GRAPH_ALL_ITEMS*)
Creates structure for depth-first graph traversal

**Parameters**
- *graph* – Graph.
- *vtx* – Initial vertex to start from. If NULL, the traversal starts from the first vertex (a vertex with the minimal index in the sequence of vertices).

- *mask* – Event mask indicating which events are interesting to the user (where cvNextGraphItem function returns control to the user) It can be CV_GRAPH_ALL_ITEMS (all events are interesting) or combination of the following flags:
  - CV_GRAPH_VERTEX - stop at the graph vertices visited for the first time
  - CV_GRAPH_TREE_EDGE - stop at tree edges (`tree edge` is the edge connecting the last visited vertex and the vertex to be visited next)
  - CV_GRAPH_BACK_EDGE - stop at back edges (`back edge` is an edge connecting the last visited vertex with some of its ancestors in the search tree)
  - CV_GRAPH_FORWARD_EDGE - stop at forward edges (`forward edge` is an edge connecting the last visited vertex with some of its descendants in the search tree). The `forward edges` are only possible during oriented graph traversal)
  - CV_GRAPH_CROSS_EDGE - stop at cross edges (`cross edge` is an edge connecting different search trees or branches of the same tree. The `cross edges` are only possible during oriented graphs traversal)
  - CV_GRAPH_ANY_EDGE - stop and any edge (`tree, back, forward` and `cross edges`)
  - CV_GRAPH_NEW_TREE - stop in the beginning of every new search tree. When the traversal procedure visits all vertices and edges reachable from the initial vertex (the visited vertices together with tree edges make up a tree), it searches for some unvisited vertex in the graph and resumes the traversal process from that vertex. Before starting a new tree (including the very first tree when cvNextGraphItem is called for the first time) it generates `CV_GRAPH_NEW_TREE` event. For undirected graphs each search tree corresponds to a connected component of the graph.
  - CV_GRAPH_BACKTRACKING - stop at every already visited vertex during backtracking - returning to already visited vertexes of the traversal tree.

The function `cvCreateGraphScanner` creates structure for depth-first graph traversal/search. The initialized structure is used in `cvNextGraphItem` function - the incremental traversal procedure.

`int` **cvNextGraphItem**(*CvGraphScanner* scanner*)

Makes one or more steps of the graph traversal procedure

**Parameter** *scanner* – Graph traversal state. It is updated by the function.

The function `cvNextGraphItem` traverses through the graph until an event interesting to the user (that is, an event, specified in the `mask` in `cvCreateGraphScanner` call) is met or the traversal is over. In the first case it returns one of the events, listed in the description of `mask` parameter above and with the next call it resumes the traversal. In the latter case it returns CV_GRAPH_OVER (-1). When the event is `CV_GRAPH_VERTEX`, or `CV_GRAPH_BACKTRACKING` or `CV_GRAPH_NEW_TREE`, the currently observed vertex is stored in `scanner->vtx`. And if the event is edge-related, the edge itself is stored at `scanner->edge`, the previously visited vertex - at `scanner->vtx` and the other ending vertex of the edge - at `scanner->dst`.

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

Finishes graph traversal procedure

**Parameter** *scanner* – Double pointer to graph traverser.

The function `cvGraphScanner` finishes graph traversal procedure and releases the traverser state.

### Trees

**CV_TREE_NODE_FIELDS**

Helper macro for a tree node type declaration

```
#define CV_TREE_NODE_FIELDS(node_type)                          \
    int       flags;         /* miscellaneous 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 */
```

The macro `CV_TREE_NODE_FIELDS()` is used to declare structures that can be organized into hierarchical structures (trees), such as `CvSeq` - the basic type for all dynamical structures. The trees made of nodes declared using this macro can be processed using the functions described below in this section.

**CvTreeNodeIterator**

Opens existing or creates new file storage

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

The structure `CvTreeNodeIterator` is used to traverse trees. The tree node declaration should start with `CV_TREE_NODE_FIELDS(...)` macro.

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

Initializes tree node iterator

**Parameters**
- *tree_iterator* – Tree iterator initialized by the function.
- *first* – The initial node to start traversing from.
- *max_level* – The maximal level of the tree (`first` node assumed to be at the first level) to traverse up to. For example, 1 means that only nodes at the same level as `first` should be visited, 2 means that the nodes on the same level as `first` and their direct children should be visited etc.

The function `cvInitTreeNodeIterator` initializes tree iterator. The tree is traversed in depth-first order.

void* **cvNextTreeNode**(*CvTreeNodeIterator* tree_iterator*)

Returns the currently observed node and moves iterator toward the next node

**Parameter** *tree_iterator* – Tree iterator initialized by the function.

The function `cvNextTreeNode` returns the currently observed node and then updates the iterator - moves it toward the next node. In other words, the function behavior is similar to `*p++` expression on usual C pointer or C++ collection iterator. The function returns NULL if there is no more nodes.

void* **cvPrevTreeNode**(*CvTreeNodeIterator* tree_iterator*)

Returns the currently observed node and moves iterator toward the previous

**Parameter** *tree_iterator* – Tree iterator initialized by the function.

The function `cvPrevTreeNode` returns the currently observed node and then updates the iterator - moves it toward the previous node. In other words, the function behavior is similar to `*p--` expression on usual C pointer or C++ collection iterator. The function returns NULL if there is no more nodes.

CvSeq* **cvTreeToNodeSeq**(*const void* first, int header_size, CvMemStorage* storage*)

Gathers all node pointers to the single sequence

**Parameters**
- *first* – The initial tree node.

- *header_size* – Header size of the created sequence (sizeof(CvSeq) is the most used value).
- *storage* – Container for the sequence.

The function `cvTreeToNodeSeq` puts pointers of all nodes reachable from `first` to the single sequence. The pointers are written subsequently in the depth-first order.

void **cvInsertNodeIntoTree**(*void* node, void* parent, void* frame*)
Adds new node to the tree

**Parameters**
- *node* – The inserted node.
- *parent* – The parent node that is already in the tree.
- *frame* – The top level node. If `parent` and `frame` are the same, `v_prev` field of `node` is set to NULL rather than `parent`.

The function `cvInsertNodeIntoTree` adds another node into tree. The function does not allocate any memory, it can only modify links of the tree nodes.

void **cvRemoveNodeFromTree**(*void* node, void* frame*)
Removes node from tree

**Parameters**
- *node* – The removed node.
- *frame* – The top level node. If `node->v_prev = NULL` and `node->h_prev` is NULL (i.e. if `node` is the first child of `frame`), `frame->v_next` is set to `node->h_next` (i.e. the first child or frame is changed).

The function `cvRemoveNodeFromTree` removes node from tree. The function does not deallocate any memory, it can only modify links of the tree nodes.

## 1.1.4 Drawing Functions

Drawing functions work with matrices/images or arbitrary depth. Antialiasing is implemented only for 8-bit images. All the functions include parameter color that means rgb value (that may be constructed with `CV_RGB` macro or `cvScalar` function) for color images and brightness for grayscale images.

If a drawn figure is partially or completely outside the image, it is clipped. For color images the order channel is: Blue Green Red ... If one needs a different channel order, it is possible to construct color via `cvScalar` with the particular channel order, or convert the image before and/or after drawing in it with `cvCvtColor` or `cvTransform`.

### Curves and Shapes

**CV_RGB**
Constructs a color value

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

void **cvLine**(*CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness=1, int line_type=8, int shift=0*)
Draws a line segment connecting two points

**Parameters**
- *img* – The image.
- *pt1* – First point of the line segment.
- *pt2* – Second point of the line segment.
- *color* – Line color.
- *thickness* – Line thickness.
- *line_type* – Type of the line: - `8` (or `0`) - 8-connected line. - `4` - 4-connected line. - `CV_AA` - antialiased line.

- *shift* – Number of fractional bits in the point coordinates.

The function `cvLine` draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB( r, g, b )`.

void **cvRectangle**(*CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness=1, int line_type=8, int shift=0*)
Draws simple, thick or filled rectangle

**Parameters**
- *img* – Image.
- *pt1* – One of the rectangle vertices.
- *pt2* – Opposite rectangle vertex.
- *color* – Line color (RGB) or brightness (grayscale image).
- *thickness* – Thickness of lines that make up the rectangle. Negative values, e.g. CV_FILLED, make the function to draw a filled rectangle.
- *line_type* – Type of the line, see `cvLine` description.
- *shift* – Number of fractional bits in the point coordinates.

The function `cvRectangle` draws a rectangle with two opposite corners `pt1` and `pt2`.

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

**Parameters**
- *img* – Image where the circle is drawn.
- *center* – Center of the circle.
- *radius* – Radius of the circle.
- *color* – Circle color.
- *thickness* – Thickness of the circle outline if positive, otherwise indicates that a filled circle has to be drawn.
- *line_type* – Type of the circle boundary, see `cvLine` description.
- *shift* – Number of fractional bits in the center coordinates and radius value.

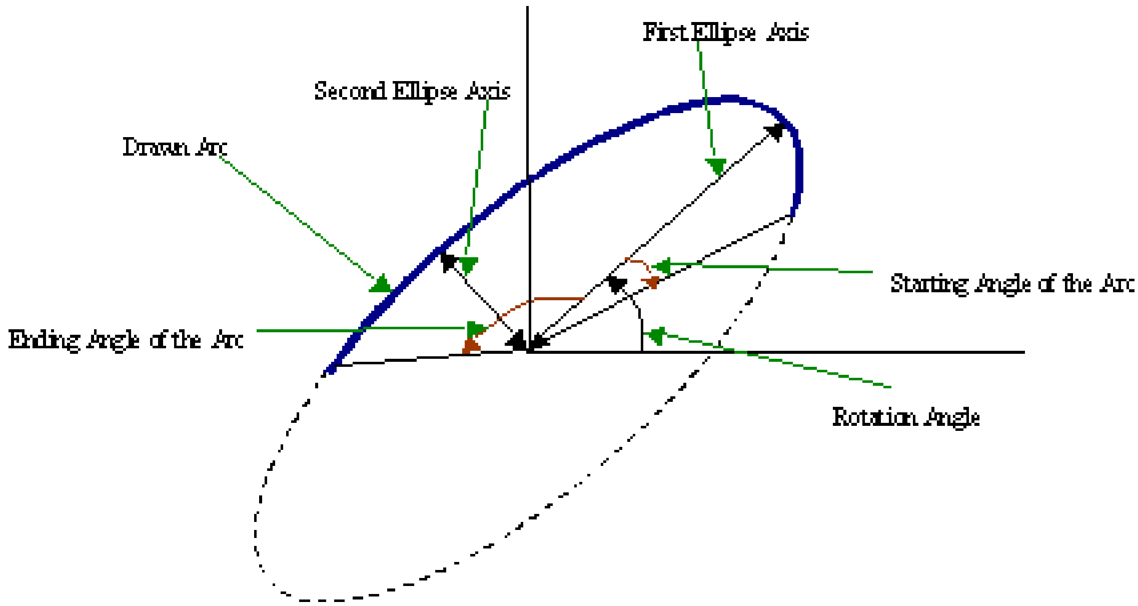
The function `cvCircle` draws a simple or filled circle with given center and radius. The circle is clipped by ROI rectangle. To specify the circle color, the user may use the macro `CV_RGB ( r, g, b )`.

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*)
Draws simple or thick elliptic arc or fills ellipse sector

**Parameters**
- *img* – Image.
- *center* – Center of the ellipse.
- *axes* – Length of the ellipse axes.
- *angle* – Rotation angle.
- *start_angle* – Starting angle of the elliptic arc.
- *end_angle* – Ending angle of the elliptic arc.
- *color* – Ellipse color.
- *thickness* – Thickness of the ellipse arc.
- *line_type* – Type of the ellipse boundary, see `cvLine` description.
- *shift* – Number of fractional bits in the center coordinates and axes' values.

The function cvEllipse draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are given in degrees. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc




void **cvEllipseBox**(*CvArr* img, CvBox2D box, CvScalar color, int thickness=1, int line_type=8, int shift=0*)
Draws simple or thick elliptic arc or fills ellipse sector

**Parameters**
- *img* – Image.
- *box* – The enclosing box of the ellipse drawn
- *thickness* – Thickness of the ellipse boundary.
- *line_type* – Type of the ellipse boundary, see cvLine description.
- *shift* – Number of fractional bits in the box vertex coordinates.

The function cvEllipseBox draws a simple or thick ellipse outline, or fills an ellipse. The functions provides a convenient way to draw an ellipse approximating some shape; that is what cvCamShift and cvFitEllipse do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

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

**Parameters**
- *img* – Image.
- *pts* – Array of pointers to polygons.
- *npts* – Array of polygon vertex counters.
- *contours* – Number of contours that bind the filled region.
- *color* – Polygon color.
- *line_type* – Type of the polygon boundaries, see cvLine description.
- *shift* – Number of fractional bits in the vertex coordinates.

The function cvFillPoly fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, etc.

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

**Parameters**
- *img* – Image.
- *pts* – Array of pointers to a single polygon.
- *npts* – Polygon vertex counter.
- *color* – Polygon color.
- *line_type* – Type of the polygon boundaries, see `cvLine` description.
- *shift* – Number of fractional bits in the vertex coordinates.

The function `cvFillConvexPoly` fills convex polygon interior. This function is much faster than The function `cvFillPoly` and can fill not only the convex polygons but any monotonic polygon, i.e. a polygon whose contour intersects every horizontal line (scan line) twice at the most.

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*)
Draws simple or thick polygons

**Parameters**
- *img* – Image.
- *pts* – Array of pointers to polylines.
- *npts* – Array of polyline vertex counters.
- *contours* – Number of polyline contours.
- *is_closed* – Indicates whether the polylines must be drawn closed. If closed, the function draws the line from the last vertex of every contour to the first vertex.
- *color* – Polyline color.
- *thickness* – Thickness of the polyline edges.
- *line_type* – Type of the line segments, see `cvLine` description.
- *shift* – Number of fractional bits in the vertex coordinates.

The function `cvPolyLine` draws a single or multiple polygonal curves.

### Text

void **cvInitFont** (*CvFont* font, int font_face, double hscale, double vscale, double shear=0, int thickness=1, int line_type=8*)
Initializes font structure

**Parameters**
- *font* – Pointer to the font structure initialized by the function.
- *font_face* – Font name identifier. Only a subset of Hershey fonts (**'http://sources.isc.org/utils/misc/hershey-font.txt'_**) are supported now:
  - `CV_FONT_HERSHEY_SIMPLEX` - normal size sans-serif font
  - `CV_FONT_HERSHEY_PLAIN` - small size sans-serif font
  - `CV_FONT_HERSHEY_DUPLEX` - normal size sans-serif font (more complex than
  - `CV_FONT_HERSHEY_SIMPLEX`)
  - `CV_FONT_HERSHEY_COMPLEX` - normal size serif font
  - `CV_FONT_HERSHEY_TRIPLEX` - normal size serif font (more complex than
  - `CV_FONT_HERSHEY_COMPLEX`)
  - `CV_FONT_HERSHEY_COMPLEX_SMALL` - smaller version of
  - `CV_FONT_HERSHEY_COMPLEX`
  - `CV_FONT_HERSHEY_SCRIPT_SIMPLEX` - hand-writing style font
  - `CV_FONT_HERSHEY_SCRIPT_COMPLEX` - more complex variant of
  - `CV_FONT_HERSHEY_SCRIPT_SIMPLEX`

The parameter can be composed from one of the values above and optional `CV_FONT_ITALIC` flag, that means italic or oblique font.

- *hscale* – Horizontal scale. If equal to `1.0f`, the characters have the original width depending on the font type. If equal to `0.5f`, the characters are of half the original width.
- *vscale* – Vertical scale. If equal to `1.0f`, the characters have the original height depending on the font type. If equal to `0.5f`, the characters are of half the original height.
- *shear* – Approximate tangent of the character slope relative to the vertical line. Zero value means a non- italic font, `1.0f` means `?45?` slope, etc. thickness Thickness of lines composing letters outlines. The function `cvLine` is used for drawing letters.
- *thickness* – Thickness of the text strokes.
- *line_type* – Type of the strokes, see `cvLine` description.

The function `cvInitFont` initializes the font structure that can be passed to text rendering functions.

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

Draws text string

**Parameters**
- *img* – Input image.
- *text* – String to print.
- *org* – Coordinates of the bottom-left corner of the first letter.
- *font* – Pointer to the font structure.
- *color* – Text color.

The function `cvPutText` renders the text in the image with the specified font and color. The printed text is clipped by ROI rectangle. Symbols that do not belong to the specified font are replaced with the rectangle symbol.

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

Retrieves width and height of text string

**Parameters**
- *font* – Pointer to the font structure.
- *text_string* – Input string.
- *text_size* – Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.
- *baseline* – y-coordinate of the baseline relatively to the bottom-most text point.

The function `cvGetTextSize` calculates the binding rectangle for the given text string when a specified font is used.

### Point Sets and Contours

void **cvDrawContours**(*CvArr* img, CvSeq* contour, CvScalar external_color, CvScalar hole_color, int max_level, int thickness=1, int line_type=8, CvPoint offset=cvPoint(0, 0)*)

Draws contour outlines or interiors in the image

**Parameters**
- *img* – Image where the contours are to be drawn. Like in any other drawing function, the contours are clipped with the ROI.
- *contour* – Pointer to the first contour.
- *external_color* – Color of the external contours.
- *hole_color* – Color of internal contours (holes).
- *max_level* – Maximal level for drawn contours. If 0, only `contour` is drawn. If 1, the contour and all contours after it on the same level are drawn. If 2, all contours after and all contours one level below the contours are drawn, etc. 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* – Thickness of lines the contours are drawn with. If it is negative (e.g. =CV_FILLED), the contour interiors are drawn.
- *line_type* – Type of the contour segments, see `cvLine` description.
- *offset* – Shift all the point coordinates by the specified value. It is useful in case if the contours retrieved in some image ROI and then the ROI offset needs to be taken into account during the rendering.

The function `cvDrawContours` draws contour outlines in the image if `thickness`>=0 or fills area bounded by the contours if ‘‘thickness‘‘<0.

**Example:** Connected component detection via contour functions:

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

int main( int argc, char** argv )
{
    IplImage* src;
    // the first command line parameter must be file name of
    binary (black-n-white) image
    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 );
            /* replace CV_FILLED with 1 to see the
            outlines */
            cvDrawContours( dst, contour, color, color,
            -1, CV_FILLED, 8 );
        }

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

Replace CV_FILLED with 1 in the sample below to see the contour outlines

`int` **`cvInitLineIterator`**(*const CvArr* image, CvPoint pt1, CvPoint pt2, CvLineIterator* line_iterator, int connectivity=8, int left_to_right=0*)

Initializes line iterator

**Parameters**
- *image* – Image to sample the line from.
- *pt1* – First ending point of the line segment.

- *pt2* – Second ending point of the line segment.
- *line_iterator* – Pointer to the line iterator state structure.
- *connectivity* – The scanned line connectivity, 4 or 8.
- *left_to_right* – The flag, indicating whether the line should be always scanned from the left-most point to the right-most out of `pt1` and `pt2` (`left_to_right?0`), or it is scanned in the specified order, from `pt1` to `pt2` (`left_to_right=0`).

The function `cvInitLineIterator` initializes the line iterator and returns the number of pixels between two end points. Both points must be inside the image. After the iterator has been initialized, all the points on the raster line that connects the two ending points may be retrieved by successive calls of `CV_NEXT_LINE_POINT` point. The points on the line are calculated one by one using 4-connected or 8-connected Bresenham algorithm.

**Example:** Using line iterator to calculate sum of pixel values along the color line

```
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, 0 );

    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 );
}
```

int **cvClipLine**(*CvSize img_size, CvPoint* pt1, CvPoint* pt2*)

Clips the line against the image rectangle

**Parameters**
- *img_size* – Size of the image.
- *pt1* – First ending point of the line segment. It is modified by the function.
- *pt2* – Second ending point of the line segment. It is modified by the function.

The function `cvClipLine` calculates a part of the line segment which is entirely in the image. It returns 0 if the line segment is completely outside the image and 1 otherwise.

int **cvEllipse2Poly**(*CvPoint center, CvSize axes, int angle, int arc_start, int arc_end, CvPoint* pts, int delta*)

Approximates elliptic arc with polyline

**Parameters**
- *center* – Center of the arc.

- *axes* – Half-sizes of the arc. See `cvEllipse`.
- *angle* – Rotation angle of the ellipse in degrees. See `cvEllipse`.
- *start_angle* – Starting angle of the elliptic arc.
- *end_angle* – Ending angle of the elliptic arc.
- *pts* – The array of points, filled by the function.
- *delta* – Angle between the subsequent polyline vertices, approximation accuracy. So, the total number of output points will ceil((end_angle - start_angle)/delta) + 1 at max.

The function `cvEllipse2Poly` computes vertices of the polyline that approximates the specified elliptic arc. It is used by `cvEllipse`.

### 1.1.5 Data Persistence and RTTI

#### File Storage

**`CvFileStorage`**

File Storage

:: typedef struct CvFileStorage {

... // hidden fields

} CvFileStorage;

The structure `CvFileStorage` is "black box" representation of file storage that is associated with a file on disk. Several functions that are described below take `cvFileStorage` on input and allow user to save or to load hierarchical collections that consist of scalar values, standard CXCORE objects (such as matrices, sequences, graphs) and user-defined objects.

CXCORE can read and write data in XML (**'http://www.w3c.org/XML'_**) or YAML (**'http://www.yaml.org'_**) formats. Below is the example of 3x3 floating-point identity matrix `A`, stored in XML and YAML files using CXCORE functions:

**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.]

As it can be seen from the examples, XML uses nested tags to represent hierarchy, while YAML uses indentation for that purpose (similarly to Python programming language).

The same CXCORE functions can read and write data in both formats, the particular format is determined by the extension of the opened file, .xml for XML files and .yml or .yaml for YAML.

**`CvFileNode`**

File Storage Node

:: /* file node type / *#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

/* optional flags */ #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;

**/* all the keys (names) of elements in the read file storage** are stored in the hash to speed up the lookup operations */

typedef struct CvStringHashNode {

unsigned hashval; CvString str; struct CvStringHashNode* next;

} CvStringHashNode;

/* basic element of the file storage - scalar or collection */ typedef struct CvFileNode {

int tag; struct CvTypeInfo* info; /* type information

(only for user-defined object, for others it is 0) */

union {

double f; /* scalar floating-point number / *int i; /* scalar integer number / *CvString str;* / text string / *CvSeq* seq; /* sequence (ordered collection of file nodes) / *struct CvMap* map; /* map (collection of named file nodes) */

} data;

} CvFileNode;

The structure is used only for retrieving data from file storage (i.e. for loading data from file). When data is written to file, it is done sequentially, with minimal buffering. No data is stored in the file storage.

In opposite, when data is read from file, the whole file is parsed and represented in memory as a tree. Every node of the tree is represented by `CvFileNode`. Type of the file node `N` can be retrieved as `CV_NODE_TYPE(N->tag)`. Some file nodes (leaves) are scalars: text strings, integer or floating-point numbers. Other file nodes are collections of file nodes, which can be scalars or collections in their turn. There are two types of collections: sequences and maps (we use YAML notation, however, the same is true for XML streams). Sequences (do not mix them with `CvSeq`) are ordered collections of unnamed file nodes, maps are unordered collections of named file nodes. Thus, elements of sequences are accessed by index (`cvGetSeqElem`), while elements of maps are accessed by name (`cvGetFileNodeByName`). The table below describes the different types of a file node:

| 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 |

**Note:** There is no need to access `map` field directly (BTW, `cvMap` is a hidden structure). The elements of the map can be retrieved with `cvGetFileNodeByName` function that takes pointer to the “map” file node.

A user (custom) object is instance of either one of standard CXCORE types, such as `CvMat`, `CvSeq` etc., or any type registered with `cvRegisterTypeInfo`. Such an object is initially represented in file as a map (as shown in XML and YAML sample files above), after file storage has been opened and parsed. Then the object can be decoded (converted to the native representation) by request - when user calls `cvRead` or `cvReadByName` function.

**CvAttrList**

List of attributes

:: typedef struct CvAttrList {

const char** attr; /* NULL-terminated array of (attribute_name,attribute_value) pairs / *struct CvAttrList* next; /* pointer to next chunk of the attributes list */

} CvAttrList;

/* initializes CvAttrList structure / *inline CvAttrList cvAttrList( const char** attr=NULL, CvAttrList* next=NULL );

/* returns attribute value or 0 (NULL) if there is no such attribute / *const char* cvAttrValue( const CvAttrList* attr, const char* attr_name );

In the current implementation attributes are used to pass extra parameters when writing user objects (see `cvWrite`). XML attributes inside tags are not supported, besides the object type specification (`type_id` attribute).

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

Opens file storage for reading or writing data

**Parameters**
- *filename* – Name of the file associated with the storage.
- *memstorage* – Memory storage used for temporary data and for storing dynamic structures, such as `CvSeq` or `CvGraph`. If it is NULL, a temporary memory storage is created and used.
- *flags* – Can be one of the following: - `CV_STORAGE_READ` - the storage is open for reading - `CV_STORAGE_WRITE` - the storage is open for writing

The function `cvOpenFileStorage` opens file storage for reading or writing data. In the latter case a new file is created or existing file is rewritten. Type of the read of written file is determined by the filename extension: `.xml` for *XML*, and `.yml` or `.yaml` for *YAML*. The function returns pointer to `CvFileStorage` structure.

`void` **cvReleaseFileStorage**(*CvFileStorage** fs*)

Releases file storage

**Parameter** *fs* – Double pointer to the released file storage.

The function `cvReleaseFileStorage` closes the file associated with the storage and releases all the temporary structures. It must be called after all I/O operations with the storage are finished.

## Writing Data

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

Starts writing a new structure

**Parameters**
- *fs* – File storage.
- *name* –

  **Name of the written structure. The structure can be** accessed by this name when the storage is read. struct_flagsA combination one of the following values:
  - `CV_NODE_SEQ` - the written structure is a sequence (see discussion of `CvFileStorage`), that is, its elements do not have a name.
  - `CV_NODE_MAP` - the written structure is a map (see discussion of `CvFileStorage`), that is, all its elements have names.

*One and only one of the two above flags must be specified*

– `CV_NODE_FLOW` - the optional flag that has sense only for YAML streams. It means that the structure is written as a flow (not as a block), which is more compact. It is recommended to use this flag for structures or arrays whose elements are all scalars.

- *type_name* – Optional parameter - the object type name. In case of XML it is written as `type_id` attribute of the structure opening tag. In case of YAML it is written after a colon following the structure name (see the example in `CvFileStorage` description). Mainly it comes with user objects. When the storage is read, the encoded type name is used to determine the object type (see `CvTypeInfo` and `cvFindTypeInfo`).
- *attributes* – This parameter is not used in the current implementation.

The function `cvStartWriteStruct` starts writing a compound structure (collection) that can be a sequence or a map. After all the structure fields, which can be scalars or structures, are written, `cvEndWriteStruct` should be called. The function can be used to group some objects or to implement *write* function for a some user object (see `CvTypeInfo`).

void **cvEndWriteStruct**(*CvFileStorage* fs*)

Ends writing a structure

**Parameter** *fs* – File storage.

The function `cvEndWriteStruct` finishes the currently written structure.

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

Writes an integer value

**Parameters**
- *fs* – File storage.
- *name* – Name of the written value. Should be NULL if and only if the parent structure is a sequence.
- *value* – The written value.

The function `cvWriteInt` writes a single integer value (with or without a name) to the file storage.

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

Writes a floating-point value

**Parameters**
- *fs* – File storage.
- *name* – Name of the written value. Should be NULL if and only if the parent structure is a sequence.
- *value* – The written value.

The function `cvWriteReal` writes a single floating-point value (with or without a name) to the file storage. The special values are encoded: NaN (Not A Number) as .NaN, ?Infinity as +.Inf (-.Inf).

The following example shows how to use the low-level writing functions to store custom structures, such as termination criteria, without registering a new type.

**:: 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 ); // just a description if( termcrit->type & CV_TERMCRIT_ITER )

cvWriteInt( fs, "max_iterations", termcrit->max_iter );

**if( termcrit->type & CV_TERMCRIT_EPS )** cvWriteReal( fs, "accuracy", termcrit->epsilon );

cvEndWriteStruct( fs );

}

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

Writes a text string

**Parameters**
- *fs* – File storage.
- *name* – Name of the written string. Should be NULL if and only if the parent structure is a sequence.
- *str* – The written text string.
- *quote* – If non-zero, the written string is put in quotes, regardless of whether they are required or not. Otherwise, if the flag is zero, quotes are used only when they are required (e.g. when the string starts with a digit or contains spaces).

The function `cvWriteString` writes a text string to the file storage.

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

Writes comment

**Parameters**
- *fs* – File storage.
- *comment* – The written comment, single-line or multi-line.
- *eol_comment* – If non-zero, the function tries to put the comment in the end of current line. If the flag is zero, if the comment is multi-line, or if it does not fit in the end of the current line, the comment starts from a new line.

The function `cvWriteComment` writes a comment into the file storage. The comments are skipped when the storage is read, so they may be used only for debugging or descriptive purposes.

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

Starts the next stream

**Parameter** *fs* – File storage.

The function `cvStartNextStream` starts the next stream in the file storage. Both YAML and XML supports multiple "streams". This is useful for concatenating files or for resuming the writing process.

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

Writes user object

**Parameters**
- *fs* – File storage.
- *name* – Name, of the written object. Should be NULL if and only if the parent structure is a sequence.
- *ptr* – Pointer to the object.
- *attributes* – The attributes of the object. They are specific for each particular type (see the discussion).

The function `cvWrite` writes the object to file storage. First, the appropriate type info is found using `cvTypeOf`. Then, `write` method of the type info is called.

Attributes are used to customize the writing procedure. The standard types support the following attributes (all the `*dt` attributes have the same format as in `cvWriteRawData`):

`CvSeq`

- `header_dt` - description of user fields of the sequence header that follow CvSeq, or CvChain (if the sequence is Freeman chain) or CvContour (if the sequence is a contour or point sequence)
- `dt` - description of the sequence elements.
- `recursive` - if the attribute is present and is not equal to "0" or "false", the whole tree of sequences (contours) is stored.

`CvGraph`

- `header_dt` - description of user fields of the graph header that follow CvGraph;
- `vertex_dt` - description of user fields of graph vertices

- `edge_dt` - description of user fields of graph edges (note, that edge weight is always written, so there is no need to specify it explicitly)

Below is the code that creates the YAML file shown in `cvFileStorage` description

```
#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;
}
```

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

Writes multiple numbers

**Parameters**
- *fs* – File storage.
- *src* – Pointer to the written array
- *len* – Number of the array elements to write.
- *dt* –

  **Specification of each array element that has the** following format: `([count]{'u'|'c'|'w'|'s'|'i'|'f'|'d'})...`, where the characters correspond to fundamental C types:
  - 'u' - 8-bit unsigned number
  - 'c' - 8-bit signed number
  - 'w' - 16-bit unsigned number
  - 's' - 16-bit signed number
  - 'i' - 32-bit signed number
  - 'f' - single precision floating-point number
  - 'd' - double precision floating-point number
  - 'r' - pointer. 32 lower bits of it are written as a signed integer.

The type can be used to store structures with links between the elements.

`count` is the optional counter of values of the certain type. For example, `dt='2if'` means that each array element is a structure of 2 integers, followed by a single-precision floating-point number. The equivalent notations of the above specification are `'iif'`, `'2i1f'` etc. Other examples: `dt='u'` means that the array consists of bytes, `dt='2d'` - the array consists of pairs of double?s.

The function `cvWriteRawData` writes array, which elements consist of a single of multiple numbers. The function call can be replaced with a loop containing a few `cvWriteInt` and `cvWriteReal` calls, but a single call is more efficient. Note, that because none of the elements have a name, they should be written to a sequence rather than a map.

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

Writes file node to another file storage :param fs: Destination file storage. :param new_file_node: New name of the file node

in the destination file storage. To keep the existing name, use :cfunc:‘cvGetFileNodeName‘(node).

**Parameter** *node* – The written node embedIf the written node is a collection and this parameter is not zero, no extra level of hierarchy is created. Instead, all the elements of `node` are written into the currently written structure. Of course, map elements may be written only to map, and sequence elements may be written only to sequence.

The function `cvWriteFileNode` writes a copy of file node to file storage. The possible application of the function are: merging several file storages into one. Conversion between XML and YAML formats etc.

### Reading Data

Data are retrieved from file storage in 2 steps: first, the file node containing the requested data is found; then, data is extracted from the node manually or using custom `read` method.

`CvFileNode*` **`cvGetRootFileNode`**(*const CvFileStorage* fs, int stream_index=0*)
Retrieves one of top-level nodes of the file storage

**Parameters**
- *fs* – File storage.
- *stream_index* – Zero-based index of the stream. See `cvStartNextStream`. In most cases, there is only one stream in the file, however there can be several.

The function `cvGetRootFileNode` returns one of top-level file nodes. The top-level nodes do not have a name, they correspond to the streams, that are stored one after another in the file storage. If the index is out of range, the function returns NULL pointer, so all the top-level nodes may be iterated by subsequent calls to the function with `stream_index=0,1,...`, until NULL pointer is returned. This function may be used as a base for recursive traversal of the file storage.

`CvFileNode*` **`cvGetFileNodeByName`**(*const CvFileStorage* fs, const CvFileNode* map, const char* name*)
Finds node in the map or file storage

**Parameters**
- *fs* – File storage.
- *map* – The parent map. If it is NULL, the function searches in all the top-level nodes (streams), starting from the first one.
- *name* – The file node name.

The function `cvGetFileNodeByName` finds a file node by `name`. The node is searched either in `map` or, if the pointer is NULL, among the top-level file nodes of the storage. Using this function for maps and `cvGetSeqElem` (or sequence reader) for sequences, it is possible to navigate through the file storage. To speed up multiple queries for a certain key (e.g. in case of array of structures) one may use a pair of `cvGetHashedKey` and `cvGetFileNode`.

`CvStringHashNode*` **`cvGetHashedKey`**(*CvFileStorage* fs, const char* name, int len=-1, int create_missing=0*)
Returns a unique pointer for given name

**Parameters**
- *fs* – File storage.
- *name* – Literal node name.
- *len* – Length of the name (if it is known a priori), or -1 if it needs to be calculated.
- *create_missing* – Flag that specifies, whether an absent key should be added into the hash table, or not.

The function `cvGetHashedKey` returns the unique pointer for each particular file node name. This pointer can be then passed to `cvGetFileNode` function that is faster than `cvGetFileNodeByName` because it compares text strings by comparing pointers rather than the strings' content.

Consider the following example: an array of points is encoded as a sequence of 2-entry maps, e.g.

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

Then, it is possible to get hashed "x" and "y" pointers to speed up
decoding of the points.
```

**Example:** Reading an array of structures from file storage:

```
#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 /* faster variant */
            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 /* slower variant; does not use x_key & y_key */
            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 /* the slowest yet the easiest to use variant */
```

```
            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;
}
```

Please note that, whatever method of accessing map you are using, it is still *much* slower than using plain sequences, for example, in the above sample, it is more efficient to encode the points as pairs of integers in the single numeric sequence.

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

Finds node in the map or file storage

**Parameters**
- *fs* – File storage.
- *map* – The parent map. If it is NULL, the function searches a top-level node. If both `map` and `key` are NULLs, the function returns the root file node - a map that contains top-level nodes.
- *key* – Unique pointer to the node name, retrieved with `cvGetHashedKey`.
- *create_missing* – Flag that specifies, whether an absent node should be added to the map, or not.

The function `cvGetFileNode` finds a file node. It is a faster version `cvGetFileNodeByName` (see `cvGetHashedKey` discussion). Also, the function can insert a new node, if it is not in the map yet (which is used by parsing functions).

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

Returns name of file node

**Parameter** *node* – File node

The function `cvGetFileNodeName` returns name of the file node or NULL, if the file node does not have a name, or if `node` is `NULL`.

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

Retrieves integer value from file node

**Parameters**
- *node* – File node.
- *default_value* – The value that is returned if `node` is NULL.

The function `cvReadInt` returns integer that is represented by the file node. If the file node is NULL, `default_value` is returned (thus, it is convenient to call the function right after `cvGetFileNode` without checking for NULL pointer), otherwise if the file node has type `CV_NODE_INT`, then `node->data.i` is returned, otherwise if the file node has type `CV_NODE_REAL`, then `node->data.f` is converted to integer and returned, otherwise the result is not determined.

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

Finds file node and returns its value

**Parameters**
- *fs* – File storage.
- *map* – The parent map. If it is NULL, the function searches a top-level node.

- *name* – The node name.
- *default_value* – The value that is returned if the file node is not found.

The function `cvReadIntByName` is a simple superposition of `cvGetFileNodeByName` and `cvReadInt`.

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

Retrieves floating-point value from file node

**Parameters**
- *node* – File node.
- *default_value* – The value that is returned if `node` is NULL.

The function `cvReadReal` returns floating-point value that is represented by the file node. If the file node is NULL, `default_value` is returned (thus, it is convenient to call the function right after `cvGetFileNode` without checking for NULL pointer), otherwise if the file node has type `CV_NODE_REAL`, then `node->data.f` is returned, otherwise if the file node has type `CV_NODE_INT`, then `node->data.f` is converted to floating-point and returned, otherwise the result is not determined.

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

Finds file node and returns its value

**Parameters**
- *fs* – File storage.
- *map* – The parent map. If it is NULL, the function searches a top-level node.
- *name* – The node name.
- *default_value* – The value that is returned if the file node is not found.

The function `cvReadRealByName` is a simple superposition of `cvGetFileNodeByName` and `cvReadReal`.

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

Retrieves text string from file node

**Parameters**
- *node* – File node.
- *default_value* – The value that is returned if `node` is NULL.

The function `cvReadString` returns text string that is represented by the file node. If the file node is NULL, `default_value` is returned (thus, it is convenient to call the function right after `cvGetFileNode` without checking for NULL pointer), otherwise if the file node has type `CV_NODE_STR`, then `node->data.str.ptr` is returned, otherwise the result is not determined.

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

Finds file node and returns its value

**Parameters**
- *fs* – File storage.
- *map* – The parent map. If it is NULL, the function searches a top-level node.
- *name* – The node name.
- *default_value* – The value that is returned if the file node is not found.

The function `cvReadStringByName` is a simple superposition of `cvGetFileNodeByName` and `cvReadString`.

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

Decodes object and returns pointer to it

**Parameters**
- *fs* – File storage.
- *node* – The root object node.
- *attributes* – Unused parameter.

The function `cvRead` decodes user object (creates object in a native representation from the file storage subtree) and returns it. The object to be decoded must be an instance of registered type that supports `read` method (see `CvTypeInfo`). Type of the object is determined by the type name that is encoded in the file. If the object is dynamic structure, it is created either in memory storage, passed to `cvOpenFileStorage` or, if NULL pointer was passed, in temporary memory storage, which is release when `cvReleaseFileStorage` is called. Otherwise, if the object is not a dynamic structure, it is created in heap and should be released with a specialized function or using generic `cvRelease`.

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

Finds object and decodes it

**Parameters**
- *fs* – File storage.
- *map* – The parent map. If it is NULL, the function searches a top-level node.
- *name* – The node name.
- *attributes* – Unused parameter.

The function `cvReadByName` is a simple superposition of `cvGetFileNodeByName` and `cvRead`.

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

Reads multiple numbers

**Parameters**
- *fs* – File storage.
- *src* – The file node (a sequence) to read numbers from.
- *dst* – Pointer to the destination array.
- *dt* – Specification of each array element. It has the same format as in `cvWriteRawData`.

The function `cvReadRawData` reads elements from a file node that represents a sequence of scalars

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

Initializes file node sequence reader

**Parameters**
- *fs* – File storage.
- *src* – The file node (a sequence) to read numbers from.
- *reader* – Pointer to the sequence reader.

The function `cvStartReadRawData` initializes sequence reader to read data from file node. The initialized reader can be then passed to `cvReadRawDataSlice`.

`void` **cvReadRawDataSlice**(*const CvFileStorage* fs, CvSeqReader* reader, int count, void* dst, const char* dt*)

Initializes file node sequence reader

**Parameters**
- *fs* – File storage.
- *reader* – The sequence reader. Initialize it with `cvStartReadRawData`.
- *count* – The number of elements to read.
- *dst* – Pointer to the destination array.
- *dt* – Specification of each array element. It has the same format as in `cvWriteRawData`.

The function `cvReadRawDataSlice` reads one or more elements from the file node, representing a sequence, to user-specified array. The total number of read sequence elements is a product of `total` and the number of components in each array element. For example, if `dt='2if'`, the function will read `total*3` sequence elements. As with any sequence, some parts of the file node sequence may be skipped or read repeatedly by repositioning the reader using `cvSetSeqReaderPos`.

### RTTI and Generic Functions

**CvTypeInfo**

Type information

```
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; /* not used */
    int header_size; /* sizeof(CvTypeInfo) */
    struct CvTypeInfo* prev; /* previous registered type in the
    list */
    struct CvTypeInfo* next; /* next registered type in the list
    */
    const char* type_name; /* type name, written to file storage
    */

    /* methods */
    CvIsInstanceFunc is_instance; /* checks if the passed object
    belongs to the type */
    CvReleaseFunc release; /* releases object (memory etc.) */
    CvReadFunc read; /* reads object from file storage */
    CvWriteFunc write; /* writes object to file storage */
    CvCloneFunc clone; /* creates a copy of the object */
}
CvTypeInfo;
```

The structure `CvTypeInfo` contains information about one of standard or user-defined types. Instances of the type may or may not contain pointer to the corresponding `CvTypeInfo` structure. In any case there is a way to find type info structure for given object - using `cvTypeOf` function. Alternatively, type info can be found by the type name using `cvFindType`, which is used when object is read from file storage. User can register a new type with `cvRegisterType` that adds the type information structure into the beginning of the type list - thus, it is possible to create specialized types from generic standard types and override the basic methods.

void **cvRegisterType**(*const CvTypeInfo* info*)

Registers new type

**Parameter** *info* – Type info structure.

The function `cvRegisterType` registers a new type, which is described by `info`. The function creates a copy of the structure, so user should delete it after calling the function.

void **cvUnregisterType**(*const char* type_name*)

Unregisters the type

**Parameter** *type_name* – Name of the unregistered type.

The function `cvUnregisterType` unregisters the type with the specified name. If the name is unknown, it is possible to locate the type info by an instance of the type using `cvTypeOf` or by iterating the type list, starting from `cvFirstType`, and then call `:cfunc:`cvUnregisterType`(info->type_name)`.

`CvTypeInfo*` **cvFirstType**(*void*)
Returns the beginning of type list

The function `cvFirstType` returns the first type of the list of registered types. Navigation through the list can be done via `prev` and `next` fields of `CvTypeInfo` structure.

`CvTypeInfo*` **cvFindType**(*const char* type_name*)
Finds type by its name

**Parameter** *type_name* – Type name.

The function `cvFindType` finds a registered type by its name. It returns NULL, if there is no type with the specified name.

`CvTypeInfo*` **cvTypeOf**(*const void* struct_ptr*)
Returns type of the object

**Parameter** *struct_ptr* – The object pointer.

The function `cvTypeOf` finds the type of given object. It iterates through the list of registered types and calls `is_instance` function/method of every type info structure with the object until one of them return non-zero or until the whole list has been traversed. In the latter case the function returns NULL.

`void` **cvRelease**(*void** struct_ptr*)
Releases the object

**Parameter** *struct_ptr* – Double pointer to the object.

The function `cvRelease` finds the type of given object and calls `release` with the double pointer.

`void*` **cvClone**(*const void* struct_ptr*)
Makes a clone of the object

**Parameter** *struct_ptr* – The object to clone.

The function `cvClone` finds the type of given object and calls `clone` with the passed object.

`void` **cvSave**(*const char* filename, const void* struct_ptr, const char* name=NULL, const char* comment=NULL, CvAttrList attributes=cvAttrList()*)
Saves object to file

**Parameters**
- *filename* – File name.
- *struct_ptr* – Object to save.
- *name* – Optional object name. If it is NULL, the name will be formed from `filename`.
- *comment* – Optional comment to put in the beginning of the file.
- *attributes* – Optional attributes passed to `cvWrite`.

The function `cvSave` saves object to file. It provides a simple interface to `cvWrite`.

`void*` **cvLoad**(*const char* filename, CvMemStorage* memstorage=NULL, const char* name=NULL, const char** real_name=NULL*)
Loads object from file

**Parameters**
- *filename* – File name.
- *memstorage* – Memory storage for dynamic structures, such as `CvSeq` or `CvGraph`. It is not used for matrices or images.
- *name* – Optional object name. If it is NULL, the first top-level object in the storage will be loaded.
- *real_name* – Optional output parameter that will contain name of the loaded object (useful if `name=NULL`).

The function `cvLoad` loads object from file. It provides a simple interface to `cvRead`. After object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as sequence, contour or graph, one should pass a valid destination memory storage to the function.

### 1.1.6 Miscellaneous Functions

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

Checks every element of input array for invalid values

**::** #define cvCheckArray cvCheckArr

**Parameters**
- *arr* – The array to check.
- *flags* – The operation flags, 0 or combination of: - `CV_CHECK_RANGE` - if set, the function checks that every value of array is

  within [minVal,maxVal) range, otherwise it just checks that every element is neither NaN nor ?Infinity.

  – **`CV_CHECK_QUIET` - if set, the function does not raises an error if an** element is invalid or out of range min_valThe inclusive lower boundary of valid values range. It is used only if `CV_CHECK_RANGE` is set.
- *max_val* – The exclusive upper boundary of valid values range. It is used only if `CV_CHECK_RANGE` is set.

The function `cvCheckArr` checks that every array element is neither NaN nor ?Infinity. If `CV_CHECK_RANGE` is set, it also checks that every element is greater than or equal to `minVal` and less than `maxVal`. The function returns nonzero if the check succeeded, i.e. all elements are valid and within the range, and zero otherwise. In the latter case if `CV_CHECK_QUIET` flag is not set, the function raises runtime error.

`int` **`cvKMeans2`** (*const CvArr* samples, int cluster_count, CvArr* labels, CvTermCriteria termcrit, int attempts=1, CvRNG* rng=0, int flags=0, CvArr* centers=0, double* compactness=0*)

Splits set of vectors by given number of clusters

**Parameters**
- *samples* – Floating-point matrix of input samples, one row per sample.
- *cluster_count* – Number of clusters to split the set by.
- *labels* – Output integer vector storing cluster indices for every sample.
- *termcrit* – Specifies maximum number of iterations of the core kmeans loop and/or accuracy (distance the centers move by between the subsequent iterations).
- *attempts* – How many times the algorithm is executed using different initial labelings. The algorithm returns labels that yield the best compactness (see the last function parameter)
- *rng* – Optional external random number generator; can be used to fully control the function behaviour.
- *flags* – Can be 0 or CV_KMEANS_USE_INITIAL_LABELS. The latter value means that during the first (and possibly the only) attempt the function uses the user-supplied labels as the initial approximation, instead of generating random labels. For the second and further attempts the function will use randomly generated labels in any case.
- *centers* – The optional output array of the cluster centers.
- *compactness* – The optional output parameter, which is computed as sumi||samples(i)-centers(labels(i))||2

after every attempt; the best (minimum) value is chosen and the corresponding labels are returned by the function. Basically, user can use only the core of the function, set the number of attempts to 1, initialize labels each time using a custom algorithm (flags=:const:*CV_KMEAN_USE_INITIAL_LABELS*) and, based on the output compactness or any other criteria, choose the best clustering.

The function `cvKMeans2` implements k-means algorithm that finds centers of `cluster_count` clusters and groups the input samples around the clusters. On output `labels(i)` contains a cluster index for sample stored in the i-th row of `samples` matrix.

**Example:** Clustering random samples of k-variate Gaussian distribution

```
#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 multivariate Gaussian
        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 ),
                   3, 0, 0, 0, 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;
    }
}
```

int **cvSeqPartition**(*const CvSeq* seq, CvMemStorage* storage, CvSeq** labels, CvCmpFunc is_equal, void* userdata*)

Splits sequence into equivalence classes

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

**Parameters**

- *seq* – The sequence to partition.
- *storage* – The storage to store the sequence of equivalence classes. If it is NULL, the function uses `seq->storage` for output labels.
- *labels* – Output parameter. Double pointer to the sequence of 0-based labels of input sequence elements.
- *is_equal* – The relation function that should return non-zero if the two particular sequence elements are from the same class, and zero otherwise. The partitioning algorithm uses transitive closure of the relation function as equivalence criteria.
- *userdata* – Pointer that is transparently passed to the `is_equal` function.

The function `cvSeqPartition` implements quadratic algorithm for splitting a set into one or more classes of equivalence. The function returns the number of equivalence classes.

**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;
}
```

### 1.1.7 Error Handling and System Functions

Error handling in OpenCV is similar to IPL (Image Processing Library). In case of error functions do not return the error code. Instead, they raise an error using `CV_ERROR` macro that calls `cvError` that, in its turn, sets the error status with `cvSetErrStatus` and calls a standard or user-defined error handler (that can display a message box, write to log etc., see `cvRedirectError`, `cvNulDevReport`, `cvStdErrReport`, `cvGuiBoxReport`). There is global variable, one per each program thread, that contains current error status (an integer value). The status can be retrieved with `cvGetErrStatus` function.

There are three modes of error handling (see `cvSetErrMode` and `cvGetErrMode`):

- **Leaf: The program is terminated after error handler is** called. *This is the default value*. It is useful for debugging, as the error is signalled immediately after it occurs. However, for production systems other two methods may be preferable as they provide more control.
- **Parent: The program is not terminated, but the error handler is** called. The stack is unwinded (it is done w/o using C++ exception mechanism). User may check error code after calling Cxcore function with `cvGetErrStatus` and react.
- **Silent:** Similar to *Parent* mode, but no error handler is called.

Actually, the semantics of *Leaf* and *Parent* modes is implemented by error handlers and the above description is true for `cvNulDevReport`, `cvStdErrReport`. `cvGuiBoxReport` behaves slightly differently, and some custom error handler may implement quite different semantics.

#### Error Handling Macros

Macros for raising an error, checking for errors etc.

**::** /* 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 the function 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." );

}

/* Provides 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) ...

Instead of a discussion, here are the documented example of typical CXCORE function and the example of the function use.

Use of Error Handling Macros:

```
#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 occurred, 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 pop up, or a message will be
                             // written
                             to log, or some user-defined processing will
                             be done
    if( cvGetErrStatus() < 0 )
        printf("Some error occurred" );
    else
        printf("Everything is OK" );
    return 0;
}
```

### Error Handling Functions

int **cvGetErrStatus**(*void*)

Returns the current error status

The function `cvGetErrStatus` returns the current error status - the value set with the last `cvSetErrStatus` call. Note, that in *Leaf* mode the program terminates immediately after error occurred, so to always get control after the function call, one should call `cvSetErrMode` and set *Parent* or *Silent* error mode.

void **cvSetErrStatus**(*int status*)
Sets the error status

**Parameter** *status* – The error status.

The function `cvSetErrStatus` sets the error status to the specified value. Mostly, the function is used to reset the error status (set to it `CV_StsOk`) to recover after error. In other cases it is more natural to call `cvError` or **'CV_ERROR'_**.

int **cvGetErrMode**(*void*)
Returns the current error mode

The function `cvGetErrMode` returns the current error mode - the value set with the last `cvSetErrMode` call.

int **cvSetErrMode**(*int mode*)
Sets the error mode

**::** #define CV_ErrModeLeaf 0 #define CV_ErrModeParent 1 #define CV_ErrModeSilent 2

**Parameter** *mode* – The error mode.

The function `cvSetErrMode` sets the specified error mode. For description of different error modes see the beginning of the `section`.

int **cvError**(*int status, const char* func_name, const char* err_msg, const char* file_name, int line*)
Raises an error

**Parameters**
- *status* – The error status.
- *func_name* – Name of the function where the error occurred.
- *err_msg* – Additional information/diagnostics about the error.
- *file_name* – Name of the file where the error occurred.
- *line* – Line number, where the error occurred.

The function `cvError` sets the error status to the specified value (via `cvSetErrStatus`) and, if the error mode is not *Silent*, calls the error handler.

const char* **cvErrorStr**(*int status*)
Returns textual description of error status code

**Parameter** *status* – The error status.

The function `cvErrorStr` returns the textual description for the specified error status code. In case of unknown status the function returns NULL pointer.

CvErrorCallback **cvRedirectError**(*CvErrorCallback error_handler, void* userdata=NULL, void** prev_userdata=NULL*)
Sets a new error handler

**::** typedef int (CV_CDECL *CvErrorCallback)( int status, const char* func_name, const char* err_msg, const char* file_name, int line );

**Parameters**
- *error_handler* – The new error_handler.
- *userdata* – Arbitrary pointer that is transparently passed to the error handler.
- *prev_userdata* – Pointer to the previously assigned user data pointer.

The function `cvRedirectError` sets a new error handler that can be one of standard handlers (`cvNullDevReport`, `cvStdErrReport` or `cvGuiBoxReport`) or a custom handler that has the certain interface. The handler takes the same parameters as `cvError` function. If the handler returns non-zero value, the program is terminated, otherwise, it continues. The error handler may check the current error mode with `cvGetErrMode` to make a decision.

```
int cvNulDevReport( int status, const char* func_name, const char* err_msg, const char* f
int cvStdErrReport( int status, const char* func_name, const char* err_msg, const char* f
int cvGuiBoxReport( int status, const char* func_name, const char* err_msg, const char* f
```

Provide standard error handling

**Parameters**
- *status* – The error status.
- *func_name* – Name of the function where the error occurred.
- *err_msg* – Additional information/diagnostics about the error.
- *file_name* – Name of the file where the error occurred.
- *line* – Line number, where the error occurred.
- *userdata* – Pointer to the user data. Ignored by the standard handlers.

The functions `cvNullDevReport`, `cvStdErrReport` and `cvGuiBoxReport` provide standard error handling. `cvGuiBoxReport` is the default error handler on Win32 systems, `cvStdErrReport` - on other systems. `cvGuiBoxReport` pops up message box with the error description and suggest a few options. Below is the sample message box that may be received with the sample code above, if one introduce an error as described in the sample

Error Message Box


If the error handler is set `cvStdErrReport`, the above message will be printed to standard error output and program will be terminated or continued, depending on the current error mode.

Error Message printed to Standard Error Output (in *Leaf* mode):

```
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...
```

## 1.1.8 System and Utility Functions

`void* cvAlloc(size_t size)`

Allocates memory buffer

**Parameter** *size* – Buffer size in bytes.

The function `cvAlloc` allocates *size* bytes and returns pointer to the allocated buffer. In case of error the function reports an error and returns NULL pointer. By default cvAlloc calls icvAlloc which itself calls malloc, however it is possible to assign user-defined memory allocation/deallocation functions using `cvSetMemoryManager` function.

`void` **`cvFree`**`(`*T** ptr*`)`

Deallocates memory buffer

**Parameter** *ptr* – Double pointer to released buffer.

The function `cvFree` deallocates memory buffer allocated by `cvAlloc`. It clears the pointer to buffer upon exit, that is why the double pointer is used. If *buffer is already NULL, the function does nothing.

`int64` **`cvGetTickCount`**`(`*void*`)`

Returns number of tics

The function `cvGetTickCount` returns number of tics starting from some platform-dependent event (number of CPU ticks from the startup, number of milliseconds from 1970th year etc.). The function is useful for accurate measurement of a function/user-code execution time. To convert the number of tics to time units, use `cvGetTickFrequency`.

`double` **`cvGetTickFrequency`**`(`*void*`)`

Returns number of tics per microsecond

The function `cvGetTickFrequency` returns number of tics per microsecond. Thus, the quotient of :cfunc:'cvGetTickCount'() and :cfunc:'cvGetTickFrequency'() will give a number of microseconds starting from the platform-dependent event.

**`cvRegisterModule`**`(`*const CvModuleInfo* module_info*`)`

Registers another module

**Parameter** *module_info* – Information about the module:

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;

The function `cvRegisterModule` adds module to the list of registered modules. After the module is registered, information about it can be retrieved using `cvGetModuleInfo` function. Also, the registered module makes full use of optimized plugins (IPP, MKL, ...), supported by CXCORE. CXCORE itself, CV (computer vision), CVAUX (auxiliary computer vision) and HIGHGUI (visualization & image/video acquisition) are examples of modules. Registration is usually done then the shared library is loaded. See cxcore/src/cxswitcher.cpp and cv/src/cvswitcher.cpp for details, how registration is done and look at cxcore/src/cxswitcher.cpp, cxcore/src/_cxipp.h on how IPP and MKL are connected to the modules.

`void` **`cvGetModuleInfo`**`(`*const char* module_name, const char** version, const char** loaded_addon_plugins*`)`

Retrieves information about the registered module(s) and plugins

**Parameters**
- *module_name* – Name of the module of interest, or NULL, which means all the modules.
- *version* – The output parameter. Information about the module(s), including version.
- *loaded_addon_plugins* – The list of names and versions of the optimized plugins that CXCORE was able to find and load.

The function `cvGetModuleInfo` returns information about one of or all of the registered modules. The returned information is stored inside the libraries, so user should not deallocate or modify the returned text strings.

`int` **`cvUseOptimized`**`(`*int on_off*`)`

Switches between optimized/non-optimized modes

**Parameter** *on_off* – Use optimized (<>0) or not (0).

The function `cvUseOptimized` switches between the mode, where only pure C implementations from cxcore, OpenCV etc. are used, and the mode, where IPP and MKL functions are used if available. When `cvUseOptimized(0)` is called, all the optimized libraries are unloaded. The function may be useful for debugging, IPP&MKL upgrade on the fly, online speed comparisons etc. It returns the number of optimized functions loaded. Note that by default the optimized plugins are loaded, so it is not necessary to call `cvUseOptimized(1)` in the beginning of the program (actually, it will only increase the startup time)

`void` **`cvSetMemoryManager`**`(`*CvAllocFunc alloc_func=NULL, CvFreeFunc free_func=NULL, void* userdata=NULL*`)`

Assigns custom/default memory managing functions

**::** typedef void* (CV_CDECL *CvAllocFunc)(size_t size, void* userdata); typedef int (CV_CDECL *CvFreeFunc)(void* pptr, void* userdata);

**Parameters**
- *alloc_func* – Allocation function; the interface is similar to `malloc`, except that `userdata` may be used to determine the context.
- *free_func* – Deallocation function; the interface is similar to `free`.
- *userdata* – User data that is transparently passed to the custom functions.

The function `cvSetMemoryManager` sets user-defined memory management functions (replacements for malloc and free) that will be called by `cvAlloc`, `cvFree` and higher-level functions (e.g. `cvCreateImage`). Note, that the function should be called when there is data allocated using *cvAlloc. Also, to avoid infinite recursive calls, it is not allowed to call :cfunc:'cvAlloc* and `cvFree` from the custom allocation/deallocation functions.

If `alloc_func` and `free_func` pointers are `NULL`, the default memory managing functions are restored.

**`void cvSetIPLAllocators( Cv_iplCreateImageHeader create_header, Cv_iplAllocateImageData a`**

Switches to IPL functions for image allocation/deallocation

**::** 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*);

**#define CV_TURN_ON_IPL_COMPATIBILITY() cvSetIPLAllocators( iplCreateImageHeader, iplAllocateImage,** iplDeallocate, iplCreateROI, iplCloneImage )

**Parameters**
- *create_header* – Pointer to iplCreateImageHeader.
- *allocate_data* – Pointer to iplAllocateImage.
- *deallocate* – Pointer to iplDeallocate.
- *create_roi* – Pointer to iplCreateROI.
- *clone_image* – Pointer to iplCloneImage.

The function `cvSetIPLAllocators` makes CXCORE to use IPL functions for image allocation/deallocation operations. For convenience, there is the wrapping macro `CV_TURN_ON_IPL_COMPATIBILITY`. The function is useful for applications where IPL and CXCORE/OpenCV are used together and still there are calls to iplCreateImageHeader() etc. The function is not necessary if IPL is called only for data processing and all the allocation/deallocation is done by CXCORE, or if all the allocation/deallocation is done by IPL and some of OpenCV functions are used to process the data.

int **cvGetNumThreads**(*void*)
Returns the current number of threads used

The function cvGetNumThreads() return the current number of threads that are used by parallelized (via OpenMP) OpenCV functions.

void **cvSetNumThreads**(*int threads=0*)
Sets the number of threads

**Parameter** *threads* – The number of threads.

The function cvSetNumThreads sets the number of threads that are used by parallelized OpenCV functions. When the argument is zero or negative, and at the beginning of the program, the number of threads is set to the number of processors in the system, as returned by the function omp_get_num_procs() from OpenMP runtime.

int **cvGetThreadNum**(*void*)
Returns index of the current thread

The function cvGetThreadNum returns the index, from 0 to cvGetNumThreads()-1, of the thread that called the function. It is a wrapper for the function omp_get_thread_num() from OpenMP runtime. The retrieved index may be used to access local-thread data inside the parallelized code fragments.

## 1.2 cv – Computer Vision Algorithms

The cv module contains functions for image processing and analysis. Most of the functions work with 2d arrays of pixels. We refer the arrays as “images” however they do not necessarily have to be IplImages, they may be CvMats or CvMatNDs as well.

### 1.2.1 Image Processing

**Contents**

### Gradients, Edges, Corners and Features

void **cvSobel**(*const CvArr* src, CvArr* dst, int xorder, int yorder, int aperture_size=3*)
Calculates first, second, third or mixed image derivatives using extended Sobel operator

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *xorder* – Order of the derivative x .
- *yorder* – Order of the derivative y .
- *aperture_size* – Size of the extended Sobel kernel, must be 1, 3, 5 or 7. Except for `aperture_size = 1`, an `aperture_size` x `aperture_size` separable kernel will be used to calculate the derivative. For `aperture_size = 1` either a 3x1 or 1x3 kernel is used (Gaussian smoothing is not done). There is also the special value `CV_SCHARR` (= -1) that corresponds to the 3x3 Scharr filter that may give more accurate results than the 3x3 Sobel. Scharr aperture is

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

for x-derivative or transposed for y-derivative.

The function `cvSobel()` calculates the image derivative by convolving the image with the appropriate kernel

$$dst(x,y) = \frac{\partial^{xorder+yorder} src}{\partial x^{xorder} \partial y^{yorder}} \Big|_{(x,y)}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less robust to the noise. Most often, the function is called with (`xorder=1`, `yorder=0`, `aperture_size=3`) or (`xorder=0`, `yorder=1`, `aperture_size=3`) to calculate first x- or y- image derivative. The first case corresponds to

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

kernel and the second one corresponds to

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

or

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

kernel, depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger by absolute value numbers than the source image. To avoid overflow, the function requires 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using `cvConvertScale()` or `cvConvertScaleAbs()` functions. Besides 8-bit images the function can process 32-bit floating-point images. Both source and destination must be single-channel images of equal size or ROI size.

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

Calculates Laplacian of the image

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *aperture_size* – Aperture size (it has the same meaning as in `cvSobel()`).

The function `cvLaplace()` calculates Laplacian of the source image by summing second x- and y- derivatives calculated using Sobel operator:

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

Specifying `aperture_size=1` gives the fastest variant that is equal to convolving the image with the following kernel:

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

Similar to `cvSobel()` function, no scaling is done and the same combinations of input and output formats are supported.

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

Implements Canny algorithm for edge detection

**Parameters**
- *image* – Input image.
- *edges* – Image to store the edges found by the function.
- *threshold1* – The first threshold.
- *threshold2* – The second threshold.
- *aperture_size* – Aperture parameter for Sobel operator (see `cvSobel`).

The function `cvCanny` finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest of `threshold1` and `threshold2` is used for edge linking, the largest - to find initial segments of strong edges.

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

Calculates feature map for corner detection

**Parameters**
- *image* – Input image.
- *corners* – Image to store the corner candidates.
- *aperture_size* – Aperture parameter for Sobel operator (see `cvSobel`).

The function `cvPreCornerDetect` calculates the function Dx2Dyy+Dy2Dxx - 2DxDyDxy where D? denotes one of the first image derivatives and D?? denotes a second image derivative. The corners can be found as local maximums of the function

```
// assume that the image is floating-point
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 );
```

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

Calculates eigenvalues and eigenvectors of image blocks for corner detection

**Parameters**
- *image* – Input image.
- *eigenvv* – Image to store the results. It must be 6 times wider than the input image.
- *block_size* – Neighborhood size (see discussion).
- *aperture_size* – Aperture parameter for Sobel operator (see `cvSobel`).

For every pixel The function `cvCornerEigenValsAndVecs` considers `block_size` ?‘‘block_size‘‘ neighborhood S(p). It calculates covariation matrix of derivatives over the neighborhood as

```
    | sumS(p)(dI/dx)2    sumS(p)(dI/dx?dI/dy)|
M = |                                        |
    | sumS(p)(dI/dx?dI/dy)   sumS(p)(dI/dy)2 |
```

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form (?1, ?2, x1, y1, x2, y2), where

- ?1, ?2 - eigenvalues of `M`; not sorted
- (x1, y1) - eigenvector corresponding to ?1
- (x2, y2) - eigenvector corresponding to ?2

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

Calculates minimal eigenvalue of gradient matrices for corner detection

**Parameters**
- *image* – Input image.
- *eigenval* – Image to store the minimal eigenvalues. Should have the same size as `image`
- *block_size* – Neighborhood size (see discussion of `cvCornerEigenValsAndVecs`).
- *aperture_size* – Aperture parameter for Sobel operator (see `cvSobel`). format. In the case of floating-point input format this parameter is the number of the fixed float filter used for differencing.

The function `cvCornerMinEigenVal` is similar to `cvCornerEigenValsAndVecs` but it calculates and stores only the minimal eigenvalue of derivative covariation matrix for every pixel, i.e. min(?1, ?2) in terms of the previous function.

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

Harris edge detector

**Parameters**
- *image* – Input image.
- *harris_dst* – Image to store the Harris detector responses. Should have the same size as `image`
- *block_size* – Neighborhood size (see discussion of `cvCornerEigenValsAndVecs()`).
- *aperture_size* – Aperture parameter for Sobel operator (see `cvSobel()`). format. In the case of floating-point input format this parameter is the number of the fixed float filter used for differencing.
- *k* – Harris detector free parameter. See the formula below.

The function `cvCornerHarris()` finds feature points (corners) in the image using Harris' method. Similarly to `cvCornerMinEigenVal` and `cvCornerEigenValsAndVecs`, for each pixel it calculates 2x2 gradient covariation matrix `M` over ?‘‘block_sizeblock_size‘‘ neighborhood. Then, it stores

```
det(M) - k*trace(M)2
```

to the corresponding pixel of the destination image. The corners can be found as local maxima in the destination image.

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

Refines corner locations

**Parameters**
- *image* – Input image.
- *corners* – Initial coordinates of the input corners and refined coordinates on output.
- *count* – Number of corners.
- *win* – Half sizes of the search window. For example, if `win=(5,5)` then 5*2+1 ? 5*2+1 = 11 ? 11 search window is used.
- *zero_zone* – Half size of the dead region in the middle of the search zone over which the summation in formulae below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size.
- *criteria* – Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after certain number of iteration or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy.

The function `cvFindCornerSubPix` iterates to find the sub- pixel accurate location of corners, or radial saddle points, as shown in on the picture below.

Sub-pixel accurate corner locator is based on the observation that every vector from the center `q` to a point `p` located within a neighborhood of `q` is orthogonal to the image gradient at `p` subject to image and measurement noise. Consider the expression

```
?i=DIpiT?(q-pi)
```

where `DIpi` is the image gradient at the one of the points `pi` in a neighborhood of `q`. The value of `q` is to be found such that ?‘‘i‘‘ is minimized. A system of equations may be set up with ?‘‘i‘’ set to zero

```
sumi(DIpi?DIpiT)?q - sumi(DIpi?DIpiT?pi) = 0
```

where the gradients are summed within a neighborhood (“search window”) of `q`. Calling the first gradient term `G` and the second gradient term `b` gives

```
q=G-1?b
```

The algorithm sets the center of the neighborhood window at this new center `q` and then iterates until the center keeps within a set threshold.

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, int block_size=3, int use_harris=0, double k=0.04*)

Determines strong corners on image

**Parameters**
- *image* – The source 8-bit or floating-point 32-bit, single-channel image.
- *eig_image* – Temporary floating-point 32-bit image of the same size as `image`.
- *temp_image* – Another temporary image of the same size and same format as `eig_image`.
- *corners* – Output parameter. Detected corners.
- *corner_count* – Output parameter. Number of detected corners.
- *quality_level* – Multiplier for the maxmin eigenvalue; specifies minimal accepted quality of image corners.
- *min_distance* – Limit, specifying minimum possible distance between returned corners; Euclidean distance is used.
- *mask* – Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL.
- *block_size* – Size of the averaging block, passed to underlying `cvCornerMinEigenVal` or `cvCornerHarris` used by the function.
- *use_harris* – If nonzero, Harris operator (`cvCornerHarris`) is used instead of default `cvCornerMinEigenVal`.
- *k* – Free parameter of Harris detector; used only if ?‘‘use_harris0‘‘

The function `cvGoodFeaturesToTrack` finds corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using `cvCornerMinEigenVal` function and stores them in `eig_image`. Then it performs non-maxima suppression (only local maxima in 3x3 neighborhood remain). The next step is rejecting the corners with the minimal eigenvalue less than `quality_level`?max(`eig_image‘‘(x,y)). Finally, the function ensures that all the corners found are distanced enough one from another by considering the corners (the most strongest corners are considered first) and checking that the distance between the newly considered feature and the features considered earlier is larger than ‘‘min_distance`. So, the function removes the features than are too close to the stronger features.

void **cvExtractSURF** (*const CvArr* image, const CvArr* mask, CvSeq** keypoints, CvSeq** descriptors, CvMemStorage* storage, CvSURFParams params*)

Extracts Speeded Up Robust Features from image

**Parameters**
- *image* – The input 8-bit grayscale image.
- *mask* – The optional input 8-bit mask. The features are only found in the areas that contain more than 50% of non-zero mask pixels.
- *keypoints* – The output parameter; double pointer to the sequence of keypoints. This will be the sequence of `CvSURFPoint` structures:

```
typedef struct CvSURFPoint
{
   CvPoint2D32f pt; // position of the feature within
   the image
   int laplacian;   // -1, 0 or +1. sign of the
   laplacian at the point.
```

```
                    // can be used to speedup feature comparison
                    // (normally features with laplacians of different signs can not b
    int size;       // size of the feature
    float dir;      // orientation of the feature: 0..360 degrees
    float hessian;  // value of the hessian (can be used to approximately estimate the
                    // see also params.hessianThreshold)
}
CvSURFPoint;
```

- *descriptors* – The optional output parameter; double pointer to the sequence of descriptors; Depending on the `params.extended` value, each element of the sequence will be either 64-element or 128-element floating-point (CV_32F) vector. If the parameter is NULL, the descriptors are not computed.
- *storage* – Memory storage where keypoints and descriptors will be stored.
- *params* – Various algorithm parameters put to the structure `CvSURFParams`

```
typedef struct CvSURFParams
{
    int extended; // 0 means basic descriptors (64
    elements each),
                  // 1 means extended descriptors (128 elements each)
    double hessianThreshold; // only features with keypoint.hessian larger than that a
                  // good default value is ~300-500 (can depend on the average
                  // local contrast and sharpness of the image).
                  // user can further filter out some features based on their hessian
                  // and other characteristics
    int nOctaves; // the number of octaves to be used for extraction.
                  // With each next octave the feature size is doubled (3 by default)
    int nOctaveLayers; // The number of layers within each octave (4 by default)
}
CvSURFParams;

CvSURFParams cvSURFParams(double hessianThreshold, int extended=0); // returns defaul
```

The function `cvExtractSURF` finds robust features in the image, as described in **'[Bay06]'_**. For each feature it returns its location, size, orientation and optionally the descriptor, basic or extended. The function can be used for object tracking and localization, image stitching etc. See find_obj.cpp demo in OpenCV samples directory.

`CvSeq*` **`cvGetStarKeypoints`**(*const CvArr* image, CvMemStorage* storage, CvStarDetectorParams params=cvStarDetectorParams()*)

Retrieves keypoints using StarDetector algorithm

**Parameters**
- *image* – The input 8-bit grayscale image.
- *storage* – Memory storage where the keypoints will be stored.
- *params* – Various algorithm parameters put to the structure `CvStarDetectorParams`:

```
typedef struct CvStarDetectorParams
{
    int maxSize; // maximal size of the features detected. The following values
                 // of the parameter are supported:
                 // 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
    int responseThreshold; // threshold for the approximatd laplacian,
                           // used to eliminate weak features
    int lineThresholdProjected; // another threshold for laplacian to eliminate edges
    int lineThresholdBinarized; // another threshold for the feature scale to eliminat
    int suppressNonmaxSize; // linear size of a pixel neighborhood for non-maxima supp
}
CvStarDetectorParams;
```

```
inline CvStarDetectorParams cvStarDetectorParams(int maxSize=45, int responseThreshol
                                                 int lineThresholdBinarized=8, int su
```

The function `cvGetStarKeypoints` extracts keypoints that are local scale-space extremas. The scale-space is constructed by computing approximate values of laplacians with different sigma's at each pixel. Instead of using pyramids, a popular approach to save computing time, all the laplacians are computed at each pixel of the original high-resolution image. But each approximate laplacian value is computed in O(1) time regardless of the sigma, thanks to the use of integral images. The algorithm is based on the paper **'[Agrawal08]'_**, but instead of square, hexagon or octagon it uses 8-end star shape, hence the name, consisting of overlapping upright and tilted squares.

Each computed feature is represented by the following structure

```
typedef struct CvStarKeypoint
{
    CvPoint pt; // coordinates of the feature
    int size; // feature size, see
    CvStarDetectorParams::maxSize
    float response; // the approximated laplacian value
    at that point.
}
CvStarKeypoint;

inline CvStarKeypoint cvStarKeypoint(CvPoint pt, int size,
float response);
```

Below is the small usage sample

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

int main(int argc, char** argv)
{
    const char* filename = argc > 1 ? argv[1] :
    "lena.jpg";
    IplImage* img = cvLoadImage( filename, 0 ), *cimg;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* keypoints = 0;
    int i;

    if( !img )
        return 0;
    cvNamedWindow( "image", 1 );
    cvShowImage( "image", img );
    cvNamedWindow( "features", 1 );
    cimg = cvCreateImage( cvGetSize(img), 8, 3 );
    cvCvtColor( img, cimg, CV_GRAY2BGR );

    keypoints = cvGetStarKeypoints( img, storage,
    cvStarDetectorParams(45) );

    for( i = 0; i < (keypoints ? keypoints->total : 0);
    i++ )
    {
        CvStarKeypoint kpt =
        *(CvStarKeypoint*)cvGetSeqElem(keypoints, i);
        int r = kpt.size/2;
```

```
        cvCircle( cimg, kpt.pt, r, CV_RGB(0,255,0));
        cvLine( cimg, cvPoint(kpt.pt.x + r, kpt.pt.y
        + r),
            cvPoint(kpt.pt.x - r, kpt.pt.y - r),
            CV_RGB(0,255,0));
        cvLine( cimg, cvPoint(kpt.pt.x - r, kpt.pt.y
        + r),
            cvPoint(kpt.pt.x + r, kpt.pt.y - r),
            CV_RGB(0,255,0));
    }
    cvShowImage( "features", cimg );
    cvWaitKey();
}
```

## Sampling, Interpolation and Geometrical Transforms

int **cvSampleLine**(*const CvArr* image, CvPoint pt1, CvPoint pt2, void* buffer, int connectivity=8*)
Reads raster line to buffer

**Parameters**
- *image* – Image to sample the line from.
- *pt1* – Starting the line point.
- *pt2* – Ending the line point.
- *buffer* – Buffer to store the line points; must have enough size to store max( |`pt2.x`-`pt1.x`|+1, |`pt2.y`-`pt1.y`|+1) points in case of 8-connected line and |`pt2.x`-`pt1.x``| +|``pt2.y`-“pt1.y“|+1 in case of 4-connected line.
- *connectivity* – The line connectivity, 4 or 8.

The function `cvSampleLine` implements a particular case of application of line iterators. The function reads all the image points lying on the line between `pt1` and `pt2`, including the ending points, and stores them into the buffer.

void **cvGetRectSubPix**(*const CvArr* src, CvArr* dst, CvPoint2D32f center*)
Retrieves pixel rectangle from image with sub-pixel accuracy

**Parameters**
- *src* – Source image.
- *dst* – Extracted rectangle.
- *center* – Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image.

The function `cvGetRectSubPix` extracts pixels from `src`

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

where the values of pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. Whereas the rectangle center must be inside the image, the whole rectangle may be partially occluded. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

void **cvGetQuadrangleSubPix**(*const CvArr* src, CvArr* dst, const CvMat* map_matrix*)
Retrieves pixel quadrangle from image with sub-pixel accuracy

**Parameters**
- *src* – Source image.
- *dst* – Extracted quadrangle.
- *map_matrix* – The transformation 2 ? 3 matrix [`A``|``b`] (see the discussion).

The function cvGetQuadrangleSubPix extracts pixels from `src` at sub-pixel accuracy and stores them to `dst` as follows

```
dst(x, y)= src( A11x'+A12y'+b1, A21x'+A22y'+b2),
```

where A and b are taken from map_matrix

```
             | A11 A12  b1 |
map_matrix = |             |
             | A21 A22  b2 | ,

x'=x-(width(dst)-1)*0.5, y'=y-(height(dst)-1)*0.5
```

where the values of pixels at non-integer coordinates A?(x,y)T+b are retrieved using bilinear interpolation. When the function needs pixels outside of the image, it uses replication border mode to reconstruct the values. Every channel of multiple-channel images is processed independently.

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

Resizes image

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *interpolation* – Interpolation method:
  - CV_INTER_NN - nearest-neighbor interpolation,
  - CV_INTER_LINEAR - bilinear interpolation (used by default)
  - CV_INTER_AREA - resampling using pixel area relation. It is the preferred method for image decimation that gives moire-free results. In case of zooming it is similar to `CV_INTER_NN` method.
  - CV_INTER_CUBIC - bicubic interpolation.

The function cvResize resizes image `src` (or its ROI) so that it fits exactly to `dst` (or its ROI).

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

Applies affine transformation to the image

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *map_matrix* – 2?3 transformation matrix.
- *flags* – A combination of interpolation method and the following optional flags:
  - CV_WARP_FILL_OUTLIERS - fill all the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`.
  - CV_WARP_INVERSE_MAP - indicates that `matrix` is inverse transform from destination image to source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `map_matrix`.
- *fillval* – A value used to fill outliers.

The function cvWarpAffine transforms source image using the specified matrix

```
dst(x?,y?)<-src(x,y)
(x?,y?)T=map_matrix?(x,y,1)T if CV_WARP_INVERSE_MAP is not
set,
(x, y)T=map_matrix?(x?,y',1)T otherwise
```

The function is similar to `cvGetQuadrangleSubPix` but they are not exactly the same. `cvWarpAffine` requires input and output image have the same data type, has larger overhead (so it is not quite suitable for small images) and can leave part of destination image unchanged. While `cvGetQuadrangleSubPix` may extract quadrangles from 8-bit images into floating-point buffer, has smaller overhead and always changes the whole destination image content.

To transform a sparse set of points, use `cvTransform` function from cxcore.

`CvMat*` **cvGetAffineTransform**(*const CvPoint2D32f* src, const CvPoint2D32f* dst, CvMat* map_matrix*)
Calculates affine transform from 3 corresponding points

**Parameters**
- *src* – Coordinates of 3 triangle vertices in the source image.
- *dst* – Coordinates of the 3 corresponding triangle vertices in the destination image.
- *map_matrix* – Pointer to the destination 2?3 matrix.

The function `cvGetAffineTransform` calculates the matrix of an affine transform such that

```
(x'i,y'i)T=map_matrix?(xi,yi,1)T
```

where `dst(i)=(x'i,y'i), src(i)=(xi,yi), i=0..2.`

`CvMat*` **cv2DRotationMatrix**(*CvPoint2D32f center, double angle, double scale, CvMat* map_matrix*)
Calculates affine matrix of 2d rotation

**Parameters**
- *center* – Center of the rotation in the source image.
- *angle* – The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed at top- left corner).
- *scale* – Isotropic scale factor.
- *map_matrix* – Pointer to the destination 2?3 matrix.

The function `cv2DRotationMatrix` calculates matrix

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

where ?=scale*cos(angle), ?=scale*sin(angle)
```

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

`void` **cvWarpPerspective**(*const CvArr* src, CvArr* dst, const CvMat* map_matrix, int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar fillval=cvScalarAll(0)*)
Applies perspective transformation to the image

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *map_matrix* – 3?3 transformation matrix.
- *flags* – A combination of interpolation method and the following optional flags:
  - CV_WARP_FILL_OUTLIERS - fill all the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`.
  - CV_WARP_INVERSE_MAP - indicates that `matrix` is inverse transform from destination image to source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `map_matrix`.
- *fillval* – A value used to fill outliers.

The function `cvWarpPerspective` transforms source image using the specified matrix

```
dst(x?,y?)<-src(x,y)
(t?x?,t?y?,t)T=map_matrix?(x,y,1)T+b if CV_WARP_INVERSE_MAP
is not set,
(t?x, t?y, t)T=map_matrix?(x?,y',1)T+b otherwise
```

For a sparse set of points use `cvPerspectiveTransform` function from cxcore.

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

Calculates perspective transform from 4 corresponding points

**Parameters**
- *src* – Coordinates of 4 quadrangle vertices in the source image.
- *dst* – Coordinates of the 4 corresponding quadrangle vertices in the destination image.
- *map_matrix* – Pointer to the destination 3?3 matrix.

The function `cvGetPerspectiveTransform` calculates matrix of perspective transform such that

```
(ti?x'i,ti?y'i,ti)T=map_matrix?(xi,yi,1)T
```

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

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

Applies generic geometrical transformation to the image

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *mapx* – The map of x-coordinates (32FC1 image).
- *mapy* – The map of y-coordinates (32FC1 image).
- *flags* – A combination of interpolation method and the following optional flag(s):
  - CV_WARP_FILL_OUTLIERS - fill all the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`.
- *fillval* – A value used to fill outliers.

The function `cvRemap` transforms source image using the specified map

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

Similar to other geometrical transformations, some interpolation method (specified by user) is used to extract pixels with non-integer coordinates.

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

Remaps image to log-polar space

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *center* – The transformation center, where the output precision is maximal.
- *M* – Magnitude scale parameter. See below.
- *flags* – A combination of interpolation method and the following optional flags:
  - CV_WARP_FILL_OUTLIERS - fill all the destination image pixels. If some of them correspond to outliers in the source image, they are set to zeros.
  - CV_WARP_INVERSE_MAP - indicates that `matrix` is inverse transform from destination image to source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `map_matrix`.
- *fillval* – A value used to fill outliers.

The function `cvLogPolar` transforms source image using the following transformation

```
Forward transformation (CV_WARP_INVERSE_MAP is not set):
dst(phi,rho)<-src(x,y)

Inverse transformation (CV_WARP_INVERSE_MAP is set):
dst(x,y)<-src(phi,rho),

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

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking etc.

**Example:** Log-polar transformation

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

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

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

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

### Morphological Operations

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

Creates structuring element

**Parameters**
- *cols* – Number of columns in the structuring element.
- *rows* – Number of rows in the structuring element.
- *anchor_x* – Relative horizontal offset of the anchor point.
- *anchor_y* – Relative vertical offset of the anchor point.
- *shape* – Shape of the structuring element; may have the following values:
  - `CV_SHAPE_RECT`, a rectangular element;
  - `CV_SHAPE_CROSS`, a cross-shaped element;
  - `CV_SHAPE_ELLIPSE`, an elliptic element;
  - `CV_SHAPE_CUSTOM`, a user-defined element. In this case the parameter `values` specifies the mask, that is, which neighbors of the pixel must be considered.

- *values* – Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non- zero values indicate points that belong to the element. If the pointer is `NULL`, then all values are considered non-zero, that is, the element is of a rectangular shape. This parameter is considered only if the shape is `CV_SHAPE_CUSTOM` .

The function `cvCreateStructuringElementEx` allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the morphological operations.

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

Deletes structuring element

**Parameter** *element* – Pointer to the deleted structuring element.

The function `cvReleaseStructuringElement` releases the structure `IplConvKernel` that is no longer needed. If `*element` is `NULL`, the function has no effect.

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

Erodes image by using arbitrary structuring element

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *element* – Structuring element used for erosion. If it is `NULL`, a 3?3 rectangular structuring element is used.
- *iterations* – Number of times erosion is applied.

The function `cvErode` erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken

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

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. In case of color image each channel is processed independently.

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

Dilates image by using arbitrary structuring element

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *element* – Structuring element used for erosion. If it is `NULL`, a 3?3 rectangular structuring element is used.
- *iterations* – Number of times erosion is applied.

The function `cvDilate` dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken

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

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. In case of color image each channel is processed independently.

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

Performs advanced morphological transformations

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *temp* – Temporary image, required in some cases.

- *element* – Structuring element.
- *operation* – Type of morphological operation, one of:
  - CV_MOP_OPEN- opening
  - CV_MOP_CLOSE- closing
  - CV_MOP_GRADIENT- morphological gradient
  - CV_MOP_TOPHAT- “top hat”
  - CV_MOP_BLACKHAT- “black hat”
- *iterations* – Number of times erosion and dilation are applied.

The function cvMorphologyEx can perform advanced morphological transformations using erosion and dilation as basic operations

```
Opening:
dst=open(src,element)=dilate(erode(src,element),element)

Closing:
dst=close(src,element)=erode(dilate(src,element),element)

Morphological gradient:
dst=morph_grad(src,element)=dilate(src,element)-erode(src,element)

"Top hat":
dst=tophat(src,element)=src-open(src,element)

"Black hat":
dst=blackhat(src,element)=close(src,element)-src
```

The temporary image `temp` is required for morphological gradient and, in case of in-place operation, for “top hat” and “black hat”.

## Filters and Color Conversion

void **cvSmooth**(*const CvArr* src, CvArr* dst, int smoothtype=CV_GAUSSIAN, int size1=3, int size2=0, double sigma1=0, double sigma2=0*)

Smoothes the image in one of several ways

**Parameters**
- *src* – The source image.
- *dst* – The destination image.
- *smoothtype* – Type of the smoothing operation:
  - CV_BLUR_NO_SCALE (simple blur with no scaling) - for each pixel the result is a sum of pixels values in `size1`?‘‘size2‘‘ neighborhood of the pixel. If the neighborhood size varies from pixel to pixel, compute the sums using integral image (cvIntegral).
  - CV_BLUR (simple blur) - for each pixel the result is the average value (brightness/color) of `size1`?‘‘size2‘‘ neighborhood of the pixel.
  - CV_GAUSSIAN (Gaussian blur) - the image is smoothed using the Gaussian kernel of aperture size `size1`?‘‘size2‘‘. `sigma1` and `sigma2` may optionally be used to specify shape of the kernel.
  - CV_MEDIAN (median blur) - the image is smoothed using medial filter of size `size1`?‘‘size1‘‘ (i.e. only square aperture can be used). That is, for each pixel the result is the median computed over `size1`?‘‘size1‘‘ neighborhood.
  - CV_BILATERAL (bilateral filter) - the image is smoothed using a bilateral 3x3 filter with color sigma=‘‘sigma1‘‘ and spatial sigma=‘‘sigma2‘‘. If `size1!=0`, then a circular kernel with diameter `size1` is used; otherwise the diameter of the kernel is computed from `sigma2`. Information about bilateral filtering can be

found at ‘ http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1 /Bilateral_Filtering.html‘_

- *size1* – The first parameter of smoothing operation. It should be odd (1, 3, 5, ...), so that a pixel neighborhood used for smoothing operation is symmetrical relative to the pixel.
- *size2* – The second parameter of smoothing operation. In case of simple scaled/non-scaled and Gaussian blur if `size2` is zero, it is set to `size1`. When not 0, it should be odd too.
- *sigma1* – In case of Gaussian kernel this parameter may specify Gaussian sigma (standard deviation). If it is zero, it is calculated from the kernel size

```
sigma = (n/2 - 1)*0.3 + 0.8,
where n=param1 for horizontal kernel,
      n=param2 for vertical kernel.
```

  With the standard sigma for small kernels (3?3 to 7?7) the performance is better. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).
  In case of Bilateral filter the parameter specifies color sigma; the larger the value, the stronger the pasterization effect of the filter is.
- *sigma2* – In case of non-square Gaussian kernel the parameter may be used to specify a different (from `param3`) sigma in the vertical direction.
  In case of Bilateral filter the parameter specifies spatial sigma; the larger the value, the stronger the blurring effect of the filter. Note that with large `sigma2` the processing speed decreases substantionally, so it is recommended to limit the kernel size using the parameter `size1`.

The function `cvSmooth` smoothes image using one of the pre- defined methods. Every of the methods has some features and restrictions listed below:

- Blur with no scaling works with single-channel images only and supports accumulation of 8u to 16s format (similar to `cvSobel` and `cvLaplace`) and accumulation of 32f to 32f format.
- Simple blur and Gaussian blur support 1- or 3-channel, 8-bit, 16-bit and 32-bit floating-point images. These two methods can process images in-place.
- Median filter works with 1- or 3-channel 8-bit images and can not process images in-place.
- Bilateral filter works with 1- or 3-channel, 8-bit or 32-bit floating-point images and can not process images in-place.

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

Applies linear filter to image

**Parameters**
- *src* – The source image.
- *dst* – The destination image.
- *kernel* – The filter mask, single-channel 2d floating point matrix of coefficients. In case of multi-channel images every channel is processed independently using the same kernel. To process different channels differently, split the image using `cvSplit` and filter the channels one by one.
- *anchor* – The anchor of the kernel. It is relative position of the filtered pixel inside its neighborhood covered by the kernel mask. The anchor should be inside the kernel. The special default value (-1,-1) means that the anchor is at the kernel center.

The function `cvFilter2D` applies the specified linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values from the nearest pixels at the image boundary. If ROI is set in the input image, cvFilter2D treats it, similarly to many other OpenCV functions, as if it were an isolated image, i.e. pixels inside the image but outside of the ROI are ignored. If it is undesirable, consider new C++ filtering classes declared in cv.hpp.

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

Copies image and makes border around it

**Parameters**
- *src* – The source image.
- *dst* – The destination image.
- *offset* – Coordinates of the top-left corner (or bottom-left in case of images with bottom-left origin) of the destination image rectangle where the source image (or its ROI) is copied. Size of the rectangle matches the source image size/ROI size.
- *bordertype* – Type of the border to create around the copied source image rectangle:
  - `IPL_BORDER_CONSTANT`- border is filled with the fixed value, passed as last parameter of the function.
  - `IPL_BORDER_REPLICATE`- the pixels from the top and bottom rows, the left-most and right-most columns are replicated to fill the border. (The other two border types from IPL, `IPL_BORDER_REFLECT` and
  - `IPL_BORDER_WRAP`, are currently unsupported).
- *value* – Value of the border pixels if `bordertype=IPL_BORDER_CONSTANT`.

The function `cvCopyMakeBorder` copies the source 2D array into interior of destination array and makes a border of the specified type around the copied area. The function is useful when one needs to emulate border type that is different from the one embedded into a specific algorithm implementation. For example, morphological functions, as well as most of other filtering functions in OpenCV, internally use replication border type, while the user may need zero border or a border, filled with 1's or 255's.

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

Calculates integral images

**Parameters**
- *image* – The source image, `W`?‘‘H‘‘, 8-bit or floating- point (32f or 64f) image.
- *sum* – The integral image, `W+1`?‘‘H+1‘‘, 32-bit integer or double precision floating-point (64f).
- *sqsum* – The optional integral image for squared pixel values, `W+1`?‘‘H+1‘‘, double precision floating-point (64f).
- *tilted_sum* – The optional integral for the image rotated by 45 degrees, `W+1`?‘‘H+1‘‘, of the same data type as `sum`.

The function `cvIntegral` calculates one or more integral images for the source image as following

```
sum(X,Y)=sumx<X,y<Yimage(x,y)

sqsum(X,Y)=sumx<X,y<Yimage(x,y)2

tilted_sum(X,Y)=sumy<Y,abs(x-X)<yimage(x,y)
```

Using these integral images, one may calculate sum, mean, standard deviation over arbitrary up-right or rotated rectangular region of the image in a constant time, for example

```
sumx1<=x<x2,y1<=y<y2image(x,y)=sum(x2,y2)-sum(x1,y2)-sum(x2,y
1)+sum(x1,x1)
```

Using integral images it is possible to do variable-size image blurring, block correlation etc. In case of multi-channel input images the integral images must have the same number of channels, and every channel is processed independently.

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

Converts image from one color space to another

**Parameters**
- *src* – The source 8-bit (8u), 16-bit (16u) or single- precision floating-point (32f) image.
- *dst* – The destination image of the same data type as the source one. The number of channels may be different.
- *code* – Color conversion operation that can be specified using CV_<src_color_space>2<dst_color_space> constants (see below).

The function `cvCvtColor` converts input image from one color space to another. The function ignores `colorModel` and `channelSeq` fields of `IplImage` header, so the source image color space should be specified correctly (including order of the channels in case of RGB space, e.g. BGR means 24-bit format with B0 G0 R0 B1 G1 R1 ... layout, whereas RGB means 24-bit format with R0 G0 B0 R1 G1 B1 ... layout).

The conventional range for R,G,B channel values is:

- 0..255 for 8-bit images
- 0..65535 for 16-bit images and
- 0..1 for floating-point images.

Of course, in case of linear transformations the range can be arbitrary, but in order to get correct results in case of non-linear transformations, the input image should be scaled if necessary.

The function can do the following transformations:

- Transformations within RGB space like adding/removing alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5) color, as well as conversion to/from grayscale using:

```
RGB[A]->Gray: Y<-0.299*R + 0.587*G + 0.114*B
Gray->RGB[A]: R<-Y G<-Y B<-Y A<-0
```

- RGB<=>CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ`, `CV_RGB2XYZ`, `CV_XYZ2BGR`, `CV_XYZ2RGB`):

```
|X|    |0.412453  0.357580  0.180423| |R|
|Y| <- |0.212671  0.715160  0.072169|*|G|
|Z|    |0.019334  0.119193  0.950227| |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|

X, Y and Z cover the whole value range (in case of
floating-point images Z may exceed 1).
```

- RGB<=>YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb`, `CV_RGB2YCrCb`, `CV_YCrCb2BGR`, `CV_YCrCb2RGB`)

```
Y <- 0.299*R + 0.587*G + 0.114*B
Cr <- (R-Y)*0.713 + delta
Cb <- (B-Y)*0.564 + delta

R <- Y + 1.403*(Cr - delta)
G <- Y - 0.344*(Cr - delta) - 0.714*(Cb - delta)
B <- Y + 1.773*(Cb - delta),

              { 128 for 8-bit images,
where delta = { 32768 for 16-bit images
              { 0.5 for floating-point
              images
```

```
Y, Cr and Cb cover the whole value range.
```

•RGB<=>HSV (CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB)

```
// In case of 8-bit and 16-bit images
// R, G and B are converted to floating-point format
and scaled to fit 0..1 range

V <- max(R,G,B)
S <- (V-min(R,G,B))/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

On output 0?V?1, 0?S?1, 0?H?360.
The values are then converted to the destination data
type:
    8-bit images:
        V <- V*255, S <- S*255, H <- H/2 (to
        fit to 0..255)
    16-bit images (currently not supported):
        V <- V*65535, S <- S*65535, H <- H
    32-bit images:
        H, S, V are left as is
```

•RGB<=>HLS (CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB)

```
// In case of 8-bit and 16-bit images
// R, G and B are converted to floating-point
format and scaled to fit 0..1 range

Vmax <- max(R,G,B)
Vmin <- min(R,G,B)

L <- (Vmax + Vmin)/2

S <- (Vmax - Vmin)/(Vmax + Vmin)  if L < 0.5
     (Vmax - Vmin)/(2 - (Vmax + Vmin))
     if L ? 0.5

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

if H<0 then H<-H+360

On output 0?L?1, 0?S?1, 0?H?360.
The values are then converted to the
destination data type:
    8-bit images:
        L <- L*255, S <- S*255, H <-
        H/2
    16-bit images (currently not
        supported):
        L <- L*65535, S <- S*65535, H
        <- H
```

```
    32-bit images:
        H, L, S are left as is
```

•RGB<=>CIE L*a*b* (CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB)

```
// In case of 8-bit and 16-bit images
// R, G and B are converted to
floating-point format and scaled to fit 0..1 range

// convert R,G,B to CIE XYZ
|X|    |0.412453  0.357580  0.180423|
|R|
|Y| <- |0.212671  0.715160
0.072169|*|G|
|Z|    |0.019334  0.119193  0.950227|
|B|

X <- X/Xn, where Xn = 0.950456
Z <- Z/Zn, where Zn = 1.088754

L <- 116*Y1/3      for Y>0.008856
L <- 903.3*Y      for Y<=0.008856

a <- 500*(f(X)-f(Y)) + delta
b <- 200*(f(Y)-f(Z)) + delta
where f(t)=t1/3              for
t>0.008856
      f(t)=7.787*t+16/116   for
      t<=0.008856


where delta = 128 for 8-bit images,
              0 for
                  floating-point images

On output 0?L?100, -127?a?127,
-127?b?127
The values are then converted to the
destination data type:
    8-bit images:
        L <- L*255/100, a <-
        a + 128, b <- b + 128
    16-bit images are
        currently not supported
    32-bit images:
        L, a, b are left as
        is
```

•RGB<=>CIE L*u*v* (CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB)

```
// In case of 8-bit and
16-bit images
// R, G and B are converted
to floating-point format and scaled to fit 0..1 range

// convert R,G,B to CIE XYZ
|X|    |0.412453  0.357580
0.180423| |R|
|Y| <- |0.212671  0.715160
```

```
0.072169|*|G|
|Z|    |0.019334  0.119193
0.950227| |B|

L <- 116*Y1/3-16   for
Y>0.008856
L <- 903.3*Y      for
Y<=0.008856

u' <- 4*X/(X + 15*Y + 3*Z)
v' <- 9*Y/(X + 15*Y + 3*Z)

u <- 13*L*(u' - un), where
un=0.19793943
v <- 13*L*(v' - vn), where
vn=0.46831096

On output 0?L?100,
-134?u?220, -140?v?122
The values are then converted
to the destination data type:
    8-bit images:
        L <-
        L*255/100, u <- (u + 134)*255/354, v <- (v +
        140)*255/256
    16-bit images are
        currently not supported
    32-bit images:
        L, u, v are
        left as is
 The above formulae for
 converting RGB to/from various color spaces have
 been taken from multiple sources on Web, primarily
 from 'Color Space Conversions (**[Ford98]**)'_
 document at Charles Poynton site.
```

- **Bayer=>RGB (`CV_BayerBG2BGR`, `CV_BayerGB2BGR`, `CV_BayerRG2BGR`, `CV_BayerGR2BGR`,** `CV_BayerBG2RGB`, `CV_BayerGB2RGB`, `CV_BayerRG2RGB`, `CV_BayerGR2RGB`)

Bayer pattern is widely used in CCD and CMOS cameras. It allows to get color picture out of a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this

```
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
```

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C1 and C2 in the conversion constants CV_BayerC1C22{BGR|RGB} indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

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

Applies fixed-level threshold to array elements

**Parameters**
- *src* – Source array (single-channel, 8-bit of 32-bit floating point).
- *dst* – Destination array; must be either the same type as `src` or 8-bit.
- *threshold* – Threshold value.
- *max_value* – Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types.

- *threshold_type* – Thresholding type (see the discussion)

The function `cvThreshold` applies fixed-level thresholding to single-channel array. The function is typically used to get bi-level (binary) image out of grayscale image (`cvCmpS` could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding the function supports that are determined by `threshold_type`

```
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 src(x,y)>threshold
           0, otherwise

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

And this is the visual description of thresholding types:

Also, the special value `CV_THRESH_OTSU` may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu algorithm and uses it instead of the specified `thresh`. The function returns the computed threshold value. Currently, Otsu method is implemented only for 8-bit images.

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*)

Applies adaptive threshold to array

**Parameters**
- *src* – Source image.
- *dst* – Destination image.
- *max_value* – Maximum value that is used with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV`.
- *adaptive_method* – Adaptive thresholding algorithm to use: `CV_ADAPTIVE_THRESH_MEAN_C` or `CV_ADAPTIVE_THRESH_GAUSSIAN_C` (see the discussion).
- *threshold_type* – Thresholding type; must be one of

•`CV_THRESH_BINARY`,- `CV_THRESH_BINARY_INV`

**Parameters**
- *block_size* – The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, ...
- *param1* – The method-dependent parameter. For the methods `CV_ADAPTIVE_THRESH_MEAN_C` and `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a constant subtracted from mean or weighted mean (see the discussion), though it may be negative.

The function `cvAdaptiveThreshold` transforms grayscale image to binary image according to the formulae

```
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
```

where TI is a threshold calculated individually for each pixel.

For the method `CV_ADAPTIVE_THRESH_MEAN_C` it is a mean of `block_size` ?‘‘block_size‘‘ pixel neighborhood, subtracted by `param1`.

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a weighted sum (Gaussian) of `block_size` ?‘‘block_size‘‘ pixel neighborhood, subtracted by `param1`.

### Pyramids and the Applications

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

Downsamples image

**Parameters**
- *src* – The source image.
- *dst* – The destination image, should have 2x smaller width and height than the source.
- *filter* – Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported.

The function `cvPyrDown` performs downsampling step of Gaussian pyramid decomposition. First it convolves source image with the specified filter and then downsamples the image by rejecting even rows and columns.

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

Upsamples image

**Parameters**
- *src* – The source image.
- *dst* – The destination image, should have 2x smaller width and height than the source.
- *filter* – Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported.

The function `cvPyrUp` performs up-sampling step of Gaussian pyramid decomposition. First it upsamples the source image by injecting even zero rows and columns and then convolves result with the specified filter multiplied by 4 for interpolation. So the destination image is four times larger than the source image.

### Image Segmentation, Connected Components and Contour Retrieval

**`CvConnectedComp`**

Connected component

```
typedef struct CvConnectedComp
{
    double area; /* area of the segmented component */
    float value; /* gray scale value of the segmented component
    */
    CvRect rect; /* ROI of the segmented component */
} CvConnectedComp;
```

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*)

Fills a connected component with given color

**Parameters**
- *image* – Input 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless CV_FLOODFILL_MASK_ONLY flag is set (see below).
- *seed_point* – The starting point.
- *new_val* – New value of repainted domain pixels.
- *lo_diff* – Maximal lower brightness/color difference between the currently observed pixel and one of its neighbor belong to the component or seed pixel to add the pixel to component. In case of 8-bit color images it is packed value.
- *up_diff* – Maximal upper brightness/color difference between the currently observed pixel and one of its neighbor belong to the component or seed pixel to add the pixel to component. In case of 8-bit color images it is packed value.
- *comp* – Pointer to structure the function fills with the information about the repainted domain.
- *flags* – The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or combination of the following flags:
  - CV_FLOODFILL_FIXED_RANGE - if set the difference between the current pixel and seed pixel is considered, otherwise difference between neighbor pixels is considered (the range is floating).
  - CV_FLOODFILL_MASK_ONLY - if set, the function does not fill the image (`new_val` is ignored), but the fills mask (that must be non-NULL in this case).
- *mask* – Operation mask, should be singe-channel 8-bit image, 2 pixels wider and 2 pixels taller than `image`. If not NULL, the function uses and updates the mask, so user takes responsibility of initializing `mask` content. Floodfilling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. Or it is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap. *Note*: because mask is larger than the filled image, pixel in `mask` that corresponds to `(x,y)` pixel in `image` will have coordinates `(x+1,y+1)`.

The function `cvFloodFill` fills a connected component starting from the seed point with the specified color. The connectivity is determined by the closeness of pixel values. The pixel at `(x, y)` is considered to belong to the repainted domain if

```
src(x',y')-lo_diff<=src(x,y)<=src(x',y')+up_diff,
grayscale image, floating range
src(seed.x,seed.y)-lo<=src(x,y)<=src(seed.x,seed.y)+up_diff,
grayscale image, fixed range

src(x',y')r-lo_diffr<=src(x,y)r<=src(x',y')r+up_diffr and
src(x',y')g-lo_diffg<=src(x,y)g<=src(x',y')g+up_diffg and
src(x',y')b-lo_diffb<=src(x,y)b<=src(x',y')b+up_diffb, color
image, floating range

src(seed.x,seed.y)r-lo_diffr<=src(x,y)r<=src(seed.x,seed.y)r+up_diffr
and
src(seed.x,seed.y)g-lo_diffg<=src(x,y)g<=src(seed.x,seed.y)g+up_diffg
and
src(seed.x,seed.y)b-lo_diffb<=src(x,y)b<=src(seed.x,seed.y)b+
up_diffb, color image, fixed range
 where ``src(x',y')`` is value of one of pixel neighbors.
 That is, to be added to the connected component, a pixel?s
```

```
color/brightness should be close enough to:

-   color/brightness of one of its neighbors that are already
    referred to the connected component in case of floating range
-   color/brightness of the seed point in case of fixed range.
```

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)*)

Finds contours in binary image

**Parameters**
- *image* – The source 8-bit single channel image. Non-zero pixels are treated as 1?s, zero pixels remain 0?s - that is image treated as `binary`. To get such a binary image from grayscale, one may use `cvThreshold`, `cvAdaptiveThreshold` or `cvCanny`. The function modifies the source image content.
- *storage* – Container of the retrieved contours.
- *first_contour* – Output parameter, will contain the pointer to the first outer contour.
- *header_size* – Size of the sequence header, >=sizeof(`CvChain`) if `method=CV_CHAIN_CODE`, and >=sizeof(`CvContour`) otherwise.
- *mode* – Retrieval mode.
  – `CV_RETR_EXTERNAL`- retrieve only the extreme outer contours
  – `CV_RETR_LIST`- retrieve all the contours and puts them in the list
  – `CV_RETR_CCOMP`- retrieve all the contours and organizes them into two-level hierarchy: top level are external boundaries of the components, second level are boundaries of the holes
  – `CV_RETR_TREE`- retrieve all the contours and reconstructs the full hierarchy of nested contours
- *method* – Approximation method (for all the modes, except `CV_RETR_RUNS`, which uses built-in approximation).
  – `CV_CHAIN_CODE`- output contours in the Freeman chain code. All other methods output polygons (sequences of vertices).
  – `CV_CHAIN_APPROX_NONE`- translate all the points from the chain code into points;
  – `CV_CHAIN_APPROX_SIMPLE`- compress horizontal, vertical, and diagonal segments, that is, the function leaves only their ending points;
  – `CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS`- apply one of the flavors of Teh-Chin chain approximation algorithm.
  – `CV_LINK_RUNS`- use completely different contour retrieval algorithm via linking of horizontal segments of 1?s. Only `CV_RETR_LIST` retrieval mode can be used with this method.
- *offset* – Offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context.

The function `cvFindContours` retrieves contours from the binary image and returns the number of retrieved contours. The pointer `first_contour` is filled by the function. It will contain pointer to the first most outer contour or NULL if no contours is detected (if the image is completely black). Other contours may be reached from `first_contour` using `h_next` and `v_next` links. The sample in `cvDrawContours` discussion shows how to use contours for connected component detection. Contours can be also used for shape analysis and object recognition - see `squares.c` in OpenCV sample directory.

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)*)

Initializes contour scanning process

**Parameters**
- *image* – The source 8-bit single channel binary image.
- *storage* – Container of the retrieved contours.
- *header_size* – Size of the sequence header, >=sizeof(`CvChain`) if `method=CV_CHAIN_CODE`, and >=sizeof(`CvContour`) otherwise.
- *mode* – Retrieval mode; see `cvFindContours`.
- *method* – Approximation method. It has the same meaning as in `cvFindContours`, but CV_LINK_RUNS can not be used here.
- *offset* – ROI offset; see `cvFindContours`.

The function `cvStartFindContours` initializes and returns pointer to the contour scanner. The scanner is used further in `cvFindNextContour` to retrieve the rest of contours.

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

Finds next contour in the image

**Parameter** *scanner* – Contour scanner initialized by The function `cvStartFindContours` .

The function `cvFindNextContour` locates and retrieves the next contour in the image and returns pointer to it. The function returns NULL, if there is no more contours.

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

Replaces retrieved contour

**Parameters**
- *scanner* – Contour scanner initialized by the function cvStartFindContours .
- *new_contour* – Substituting contour.

The function `cvSubstituteContour` replaces the retrieved contour, that was returned from the preceding call of The function `cvFindNextContour` and stored inside the contour scanner state, with the user-specified contour. The contour is inserted into the resulting structure, list, two-level hierarchy, or tree, depending on the retrieval mode. If the parameter `new_contour=NULL`, the retrieved contour is not included into the resulting structure, nor all of its children that might be added to this structure later.

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

Finishes scanning process

**Parameter** *scanner* – Pointer to the contour scanner.

The function `cvEndFindContours` finishes the scanning process and returns the pointer to the first contour on the highest level.

`void` **`cvPyrSegmentation`**(*IplImage* src, IplImage* dst, CvMemStorage* storage, CvSeq** comp, int level, double threshold1, double threshold2*)

Does image segmentation by pyramids

**Parameters**
- *src* – The source image.
- *dst* – The destination image.
- *storage* – Storage; stores the resulting sequence of connected components.
- *comp* – Pointer to the output sequence of the segmented components.
- *level* – Maximum level of the pyramid for the segmentation.
- *threshold1* – Error threshold for establishing the links.
- *threshold2* – Error threshold for the segments clustering.

The function `cvPyrSegmentation` implements image segmentation by pyramids. The pyramid builds up to the level `level`. The links between any pixel `a` on level `i` and its candidate father pixel `b` on the adjacent level are established if

`<p(c(a),c(b))threshold1`. After the connected components are defined, they are joined into several clusters. Any two segments A and B belong to the same cluster, if

<p(c(A),c(B))threshold2. The input image has only one channel, then

????`` p(c,c)=|c-c|``. If the input image has three channels (red, green and blue), then

??*??*??*??``p(c,c)=0,3(cr-cr)+0,59(cg-cg)+0,11(cb-cb) `` . There may be more than one connected component per a cluster.

The images `src` and `dst` should be 8-bit single-channel or 3-channel images or equal size

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

Does MeanShift image segmentation

**Parameters**
- *src* – The source 8-bit 3-channel image.
- *dst* – The destination image of the same format and the same size as the source.
- *sp* – The spatial window radius.
- *sr* – The color window radius.
- *max_level* – Maximum level of the pyramid for the segmentation.
- *termcrit* – Termination criteria: when to stop meanshift iterations.

The function `cvPyrMeanShiftFiltering` implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel `(X,Y)` of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel `(X,Y)` neighborhood in the joint space- color hyper-space is considered: ::{(x,y): X-sp?x?X+sp && Y-sp?y?Y+sp && ||(R,G,B)-(r,g,b)|| ? sr}, where `(R,G,B)` and `(r,g,b)` are the vectors of color components at `(X,Y)` and `(x,y)`, respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value `(X',Y')` and average color vector `(R',G',B')` are found and they act as the neighborhood center on the next iteration: ::(X,Y)~(X',Y'), (R,G,B)~(R',G',B'). After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration): ::I(X,Y) <- (R*,G*,B*).

Then `max_level`>0, the Gaussian pyramid of `max_level+1` levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much (>``sr``) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when `max_level==0`).

`void` **cvWatershed**(*const CvArr* image, CvArr* markers*)

Does watershed segmentation

**Parameters**
- *image* – The input 8-bit 3-channel image.
- *markers* – The input/output 32-bit single-channel image (map) of markers.

The function `cvWatershed` implements one of the variants of watershed, non-parametric marker-based segmentation algorithm, described in **'[Meyer92]'_** Before passing the image to the function, user has to outline roughly the desired regions in the image `markers` with positive (>0) indices, i.e. every region is represented as one or more connected components with the pixel values 1, 2, 3 etc. Those components will be "seeds" of the future image regions. All the other pixels in `markers`, which relation to the outlined regions is not known and should be defined by the algorithm, should be set to 0's. On the output of the function, each pixel in `markers` is set to one of values of the "seed" components, or to -1 at boundaries between the regions.

Note, that it is not necessary that every two neighbor connected components are separated by a watershed boundary (-1's pixels), for example, in case when such tangent components exist in the initial marker image. Visual demonstration and usage example of the function can be found in OpenCV samples directory; see `watershed.cpp` demo.

### Image and Contour moments

void **cvMoments**(*const CvArr* arr, CvMoments* moments, int binary=0*)
Calculates all moments up to third order of a polygon or rasterized shape

**Parameters**
- *arr* – Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points or a vector of points).
- *moments* – Pointer to returned moment state structure.
- *binary* – (For images only) If the flag is non-zero, all the zero pixel values are treated as zeroes, all the others are treated as 1?s.

The function `cvMoments` calculates spatial and central moments up to the third order and writes them to `moments`. The moments may be used then to calculate gravity center of the shape, its area, main axises and various shape characteristics including 7 Hu invariants.

double **cvGetSpatialMoment**(*CvMoments* moments, int x_order, int y_order*)
Retrieves spatial moment from moment state structure

**Parameters**
- *moments* – The moment state, calculated by `cvMoments`.
- *x_order* – x order of the retrieved moment, `x_order` >= 0.
- *y_order* – y order of the retrieved moment, `y_order` >= 0 and `x_order` + `y_order` <= 3.

The function `cvGetSpatialMoment` retrieves the spatial moment, which in case of image moments is defined as

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

where `I(x,y)` is the intensity of the pixel `(x, y)`.

double **cvGetCentralMoment**(*CvMoments* moments, int x_order, int y_order*)
Retrieves central moment from moment state structure

**Parameters**
- *moments* – Pointer to the moment state structure.
- *x_order* – x order of the retrieved moment, `x_order` >= 0.
- *y_order* – y order of the retrieved moment, `y_order` >= 0 and `x_order` + `y_order` <= 3.

The function `cvGetCentralMoment` retrieves the central moment, which in case of image moments is defined as

```
?x_order,y_order=sumx,y(I(x,y)?(x-xc)x_order?(y-yc)y_order),
```

where `xc=M10/M00, yc=M01/M00` - coordinates of the gravity center

double **cvGetNormalizedCentralMoment**(*CvMoments* moments, int x_order, int y_order*)
Retrieves normalized central moment from moment state structure

**Parameters**
- *moments* – Pointer to the moment state structure.
- *x_order* – x order of the retrieved moment, `x_order` >= 0.
- *y_order* – y order of the retrieved moment, `y_order` >= 0 and `x_order` + `y_order` <= 3.

The function `cvGetNormalizedCentralMoment` retrieves the normalized central moment

```
?x_order,y_order= ?x_order,y_order/M00((y_order+x_order)/2+1)
```

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

Calculates seven Hu invariants

**Parameters**
- *moments* – Pointer to the moment state structure.
- *hu_moments* – Pointer to Hu moments structure.

The function `cvGetHuMoments` calculates seven Hu invariants that are defined as:

```
h1=?20+?02

h2=(?20-?02)?+4?11?

h3=(?30-3?12)?+ (3?21-?03)?

h4=(?30+?12)?+ (?21+?03)?

h5=(?30-3?12)(?30+?12)[(?30+?12)?-3(?21+?03)?]+(3?21-?03)(?
21+?03)[3(?30+?12)?-(?21+?03)?]

h6=(?20-?02)[(?30+?12)?- (?21+?03)?]+4?11(?30+?12)(?21+?03)

h7=(3?21-?03)(?21+?03)[3(?30+?12)?-(?21+?03)?]-(?30-3?12)(?
21+?03)[3(?30+?12)?-(?21+?03)?]
```

where ?‘‘i,j‘‘ are normalized central moments of 2-nd and 3-rd orders. The computed values are proved to be invariant to the image scaling, rotation, and reflection except the seventh one, whose sign is changed by reflection.

### Special Image Transforms

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

Finds lines in binary image using Hough transform

**Parameters**
- *image* – The input 8-bit single-channel binary image. In case of probabilistic method the image is modified by the function.
- *line_storage* – The storage for the lines detected. It can be a memory storage (in this case a sequence of lines is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of a particular type (see below) to which the lines’ parameters are written. The matrix header is modified by the function so its `cols` or `rows` will contain a number of lines detected. If `line_storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (in case of standard hough transform the lines are sorted by the accumulator value).
- *method* – The Hough transform variant, one of:
  - `CV_HOUGH_STANDARD`- classical or standard Hough transform. Every line is represented by two floating-point numbers (?, ?), where ? is a distance between (0,0) point and the line, and ? is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of CV_32FC2 type.
  - `CV_HOUGH_PROBABILISTIC`- probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole lines. Every segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of CV_32SC4 type.
  - `CV_HOUGH_MULTI_SCALE`- multi-scale variant of classical Hough transform. The lines are encoded the same way as in CV_HOUGH_STANDARD.
- *rho* – Distance resolution in pixel-related units.

- *theta* – Angle resolution measured in radians.
- *threshold* – Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`.
- *param1* – The first method-dependent parameter:
  - For classical Hough transform it is not used (0).
  - For probabilistic Hough transform it is the minimum line length.
  - For multi-scale Hough transform it is divisor for distance resolution `rho`. (The coarse distance resolution will be `rho` and the accurate resolution will be (`rho` / `param1`)).
- *param2* – The second method-dependent parameter:
  - For classical Hough transform it is not used (0).
  - For probabilistic Hough transform it is the maximum gap between line segments lying on the same line to treat them as the single line segment (i.e. to join them).
  - For multi-scale Hough transform it is divisor for angle resolution `theta`. (The coarse angle resolution will be `theta` and the accurate resolution will be (`theta` / `param2`)).

The function `cvHoughLines2` implements a few variants of Hough transform for line detection.

**Example:** Detecting lines with Hough transform

```
/* This is a stand-alone 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, 100, 0, 0 );

        for( i = 0; i < MIN(lines->total,100); 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);
            double x0 = a*rho, y0 = b*rho;
            pt1.x = cvRound(x0 + 1000*(-b));
```

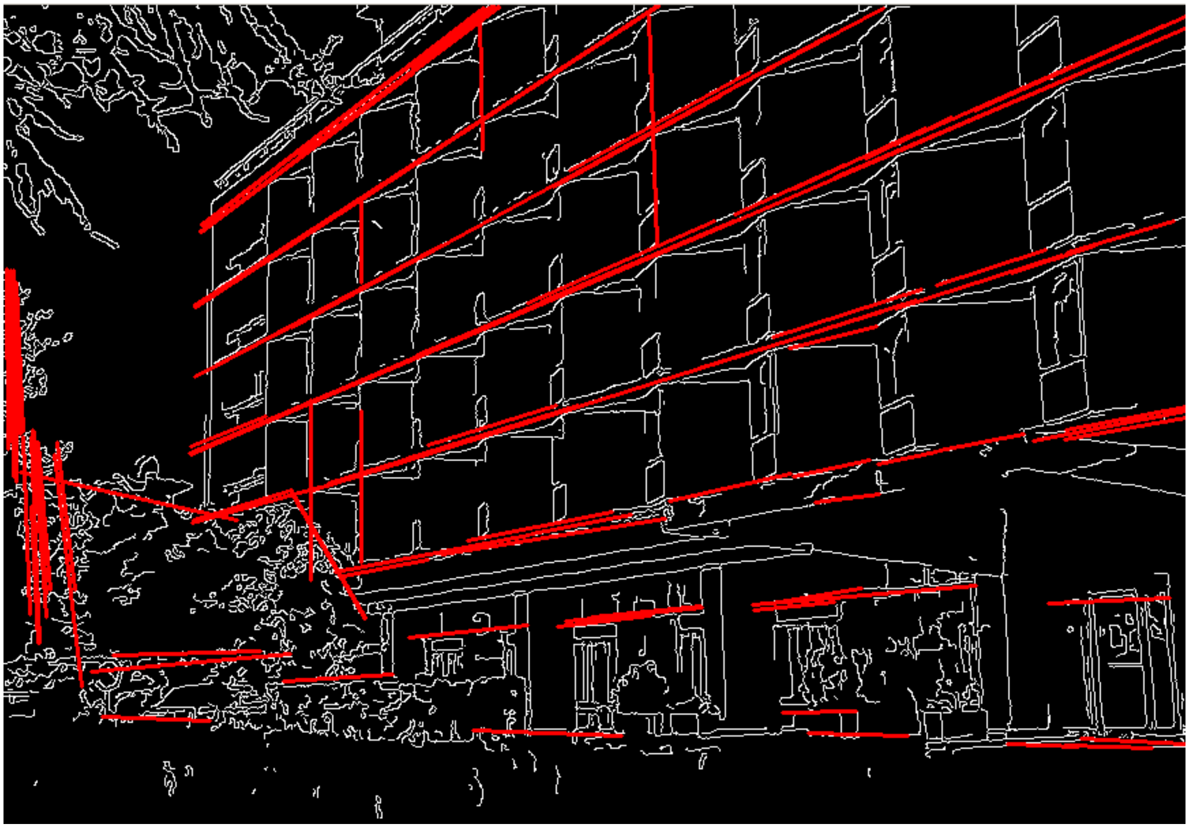
```
            pt1.y = cvRound(y0 + 1000*(a));
            pt2.x = cvRound(x0 - 1000*(-b));
            pt2.y = cvRound(y0 - 1000*(a));
            cvLine( color_dst, pt1, pt2,
            CV_RGB(255,0,0), 3, 8 );
        }
#else
        lines = cvHoughLines2( dst, storage,
        CV_HOUGH_PROBABILISTIC, 1, CV_PI/180, 50, 50, 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);
    }
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in case of probabilistic Hough transform ("#if 0" case):



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

Finds circles in grayscale image using Hough transform

**Parameters**

- *image* – The input 8-bit single-channel grayscale image.
- *circle_storage* – The storage for the circles detected. It can be a memory storage (in this case a sequence of circles is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of type CV_32FC3, to which the circles' parameters are written. The matrix header is modified by the function so its `cols` or `rows` will contain a number of lines detected. If `circle_storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of circles is returned. Every circle is encoded as 3 floating-point numbers: center coordinates (x,y) and the radius.
- *method* – Currently, the only implemented method is `CV_HOUGH_GRADIENT`, which is basically 21HT, described in **'**[Yuen03]**'_**.
- *dp* – Resolution of the accumulator used to detect centers of the circles. For example, if it is 1, the accumulator will have the same resolution as the input image, if it is 2 - accumulator will have twice smaller width and height, etc.
- *min_dist* – Minimum distance between centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed.
- *param1* – The first method-specific parameter. In case of `CV_HOUGH_GRADIENT` it is the higher threshold of the two passed to Canny edge detector (the lower one will be twice smaller).
- *param2* – The second method-specific parameter. In case of `CV_HOUGH_GRADIENT` it is accumulator threshold at the center detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first.

- *min_radius* – Minimal radius of the circles to search for.
- *max_radius* – Maximal radius of the circles to search for. By default the maximal radius is set to `max(image_width, image_height)`.

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

**Example:** Detecting circles with Hough transform

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

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

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

Calculates distance to closest zero pixel for all non-zero pixels of source image

**Parameters**

- *src* – Source 8-bit single-channel (binary) image.
- *dst* – Output image with calculated distances. In most cases it should be 32-bit floating-point, single-channel array of the same size as the input image. When `distance_type` == `CV_DIST_L1`, 8-bit, single-channel destination array may be also used (in-place operation is also supported in this case).
- *distance_type* – Type of distance; can be `CV_DIST_L1, CV_DIST_L2, CV_DIST_C` or `CV_DIST_USER`.
- *mask_size* – Size of distance transform mask; can be 3, 5 or 0. In case of `CV_DIST_L1` or `CV_DIST_C` the parameter is forced to 3, because 3?3 mask gives the same result as 5?5

yet it is faster. When `mask_size==0`, a different non-approximate algorithm is used to calculate distances.

- *mask* – User-defined mask in case of user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in case of 3?3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight?s move cost) in case of 5?5 mask.
- *labels* – The optional output 2d array of labels of integer type and the same size as `src` and `dst`, can now be used only with `mask_size==3` or 5.

The function `cvDistTransform` calculates the approximated or exact distance from every binary image pixel to the nearest zero pixel. When `mask_size==0`, the function uses the accurate algorithm **'[Felzenszwalb04]'_**. When `mask_size==3` or 5, the function uses the approximate algorithm **'[Borgefors86]'_**.

Here is how the approximate algorithm works. For zero pixels the function sets the zero distance. For others it finds the shortest path to a zero pixel, consisting of basic shifts: horizontal, vertical, diagonal or knight?s move (the latest is available for 5?5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all the horizontal and vertical shifts must have the same cost (that is denoted as `a`), all the diagonal shifts must have the same cost (denoted `b`), and all knight?s moves must have the same cost (denoted `c`). For `CV_DIST_C` and `CV_DIST_L1` types the distance is calculated precisely, whereas for `CV_DIST_L2` (Euclidean distance) the distance can be calculated only with some relative error (5?5 mask gives more accurate results), OpenCV uses the values suggested in **'[Borgefors86]'_**

```
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
```

And below are samples of distance field (black (0) pixel is in the middle of white square) in case of user-defined distance:

User-defined 3?3 mask (a=1, b=1.5)

```
4.543.533.544.5
432.522.534
3.52.51.511.52.53.5
3210123
3.52.51.511.52.53.5
432.522.534
4.543.533.544.5
```

User-defined 5?5 mask (a=1, b=1.5, c=2)

```
4.53.53333.54.5
3.5322233.5
321.511.523
3210123
321.511.523
3.5322233.5
43.53333.54
```

Typically, for fast coarse distance estimation CV_DIST_L2, 3?3 mask is used, and for more accurate distance estimation CV_DIST_L2, 5?5 mask is used.

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

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

void **cvInpaint**(*const CvArr* src, const CvArr* mask, CvArr* dst, int flags, double inpaintRadius*)
Inpaints the selected region in the image

**Parameters**
- *src* – The input 8-bit 1-channel or 3-channel image.
- *mask* – The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.
- *dst* – The output image of the same format and the same size as input.
- *flags* – The inpainting method, one of the following:

`CV_INPAINT_NS`- Navier-Stokes based method. `CV_INPAINT_TELEA`- The method by Alexandru Telea **‘[Telea04]‘_**

**param inpaintRadius** The radius of circular neighborhood of each point inpainted that is considered by the algorithm.

The function `cvInpaint` reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video.

## Histograms

### CvHistogram

Multi-dimensional histogram

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

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

**Parameters**
- *dims* – Number of histogram dimensions.
- *sizes* – Array of histogram dimension sizes.
- *type* – Histogram representation format: `CV_HIST_ARRAY` means that histogram data is represented as an multi-dimensional dense array `CvMatND`; `CV_HIST_SPARSE` means that histogram data is represented as a multi-dimensional sparse array `CvSparseMat`.
- *ranges* – Array of ranges for histogram bins. Its meaning depends on the `uniform` parameter value. The ranges are used for when histogram is calculated or back-projected to determine, which histogram bin corresponds to which value/tuple of values from the input image[s].
- *uniform* – Uniformity flag; if not 0, the histogram has evenly spaced bins and for every `<<0=icDims````ranges[i]` is array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is split

then into `dims[i]` equal parts to determine `i-th` input tuple value ranges for every histogram bin. And if `uniform=0`, then `i-th` element of `ranges` array contains `dims[i]+1` elements: `lower0, upper0, lower1, upper1 == lower2, ..., upperdims[i]-1`, where `lowerj` and `upperj` are lower and upper boundaries of `i-th` input tuple value for `j-th` bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin, are not counted by `cvCalcHist` and filled with 0 by `cvCalcBackProject`.

The function `cvCreateHist` creates a histogram of the specified size and returns the pointer to the created histogram. If the array `ranges` is 0, the histogram bin ranges must be specified later via The function `cvSetHistBinRanges`, though `cvCalcHist` and `cvCalcBackProject` may process 8-bit images without setting bin ranges, they assume equally spaced in 0..255 bins.

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

Sets bounds of histogram bins

**Parameters**
- *hist* – Histogram.
- *ranges* – Array of bin ranges arrays, see `cvCreateHist`.
- *uniform* – Uniformity flag, see `cvCreateHist`.

The function `cvSetHistBinRanges` is a stand-alone function for setting bin ranges in the histogram. For more detailed description of the parameters `ranges` and `uniform` see `cvCalcHist` function, that can initialize the ranges as well. Ranges for histogram bins must be set before the histogram is calculated or back projection of the histogram is calculated.

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

Releases histogram

**Parameter** *hist* – Double pointer to the released histogram.

The function `cvReleaseHist` releases the histogram (header and the data). The pointer to histogram is cleared by the function. If `*hist` pointer is already `NULL`, the function does nothing.

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

Clears histogram

**Parameter** *hist* – Histogram.

The function `cvClearHist` sets all histogram bins to 0 in case of dense histogram and removes all histogram bins in case of sparse array.

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

Makes a histogram out of array

**Parameters**
- *dims* – Number of histogram dimensions.
- *sizes* – Array of histogram dimension sizes.
- *hist* – The histogram header initialized by the function.
- *data* – Array that will be used to store histogram bins.
- *ranges* – Histogram bin ranges, see `cvCreateHist`.
- *uniform* – Uniformity flag, see `cvCreateHist`.

The function `cvMakeHistHeaderForArray` initializes the histogram, which header and bins are allocated by user. No `cvReleaseHist` need to be called afterwards. Only dense histograms can be initialized this way. The function returns `hist`.

**`cvQueryHistValue_1D`**

**`cvQueryHistValue_2D`**

**cvQueryHistValue_3D**

**cvQueryHistValue_nD**

Queries value of histogram bin

**Parameters**
- *hist* – Histogram.
- *idx0, idx1, idx2, idx3* – Indices of the bin.
- *idx* – Array of indices

The macros **'cvQueryHistValue_*D'_** return the value of the specified bin of 1D, 2D, 3D or N-D histogram. In case of sparse histogram the function returns 0, if the bin is not present in the histogram, and no new bin is created.

The macros are defined as

```
#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) )
```

**cvGetHistValue_1D**

**cvGetHistValue_2D**

**cvGetHistValue_3D**

**cvGetHistValue_nD**

Returns pointer to histogram bin

**Parameters**
- *hist* – Histogram.
- *idx0, idx1, idx2, idx3* – Indices of the bin.
- *idx* – Array of indices

The macros **'cvGetHistValue_*D'_** return pointer to the specified bin of 1D, 2D, 3D or N-D histogram. In case of sparse histogram the function creates a new bin and sets it to 0, unless it exists already.

The macros are defined as

```
#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 ))
```

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

Finds minimum and maximum histogram bins

**Parameters**
- *hist* – Histogram.
- *min_value* – Pointer to the minimum value of the histogram
- *max_value* – Pointer to the maximum value of the histogram
- *min_idx* – Pointer to the array of coordinates for minimum
- *max_idx* – Pointer to the array of coordinates for maximum

The function `cvGetMinMaxHistValue` finds the minimum and maximum histogram bins and their positions. Any of output arguments is optional. Among several extremums with the same value the ones with minimum index (in lexicographical order) In case of several maximums or minimums the earliest in lexicographical order extrema locations are returned.

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

Normalizes histogram

**Parameters**
- *hist* – Pointer to the histogram.
- *factor* – Normalization factor.

The function `cvNormalizeHist` normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor`.

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

Thresholds histogram

**Parameters**
- *hist* – Pointer to the histogram.
- *threshold* – Threshold level.

The function `cvThreshHist` clears histogram bins that are below the specified threshold.

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

Compares two dense histograms

**Parameters**
- *hist1* – The first dense histogram.
- *hist2* – The second dense histogram.
- *method* – Comparison method, one of:
  - CV_COMP_CORREL
  - CV_COMP_CHISQR
  - CV_COMP_INTERSECT
  - CV_COMP_BHATTACHARYYA

The function `cvCompareHist` compares two dense histograms using the specified method as following (`H1` denotes the first histogram, `H2` - the second)

```
Correlation (method=CV_COMP_CORREL):
d(H1,H2)=sumI(H'1(I)?H'2(I))/sqrt(sumI[H'1(I)2]?sumI[H'2(I)2])
where
H'k(I)=Hk(I)-1/N?sumJHk(J) (N=number of histogram bins)

Chi-Square (method=CV_COMP_CHISQR):
d(H1,H2)=sumI[(H1(I)-H2(I))/(H1(I)+H2(I))]

Intersection (method=CV_COMP_INTERSECT):
d(H1,H2)=sumImin(H1(I),H2(I))

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA):
d(H1,H2)=sqrt(1-sumI(sqrt(H1(I)?H2(I))))
```

The function returns `d(H1,H2)` value.

Note: the method `CV_COMP_BHATTACHARYYA` only works with normalized histograms.

To compare sparse histogram or more general sparse configurations of weighted points, consider using `cvCalcEMD2` function.

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

Copies histogram

**Parameters**
- *src* – Source histogram.
- *dst* – Pointer to destination histogram.

The function `cvCopyHist` makes a copy of the histogram. If the second histogram pointer `*dst` is NULL, a new histogram of the same size as `src` is created. Otherwise, both histograms must have equal types and sizes. Then the function copies the source histogram bins values to destination histogram and sets the same bin values ranges as in `src`.

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

Calculates histogram of image(s)

**Parameters**
- *image* – Source images (though, you may pass CvMat** as well), all are of the same size and type
- *hist* – Pointer to the histogram.
- *accumulate* – Accumulation flag. If it is set, the histogram is not cleared in the beginning. This feature allows user to compute a single histogram from several images, or to update the histogram online.
- *mask* – The operation mask, determines what pixels of the source images are counted.

The function `cvCalcHist` calculates the histogram of one or more single-channel images. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input images.

**Example:** Calculating and displaying 2D Hue-Saturation histogram of a color image

```
#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), /* draw a grayscale histogram.  i
                                                               you have idea how to do it nicer let us k
                        CV_FILLED );
            }
        }

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

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

        cvWaitKey(0);
    }
}
```

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

Calculates back projection

**Parameters**
- *image* – Source images (though you may pass CvMat** as well), all are of the same size and type
- *back_project* – Destination back projection image of the same type as the source images.
- *hist* – Histogram.

The function `cvCalcBackProject` calculates the back project of the histogram. For each tuple of pixels at the same position of all input single-channel images the function puts the value of the histogram bin, corresponding to the tuple, to the destination image. In terms of statistics, the value of each output image pixel is probability of the observed tuple given the distribution (histogram). For example, to find a red object in the picture, one may do the following:

1. Calculate a hue histogram for the red object assuming the image contains only this object. The histogram is likely to have a strong maximum, corresponding to red color.
2. Calculate back projection of a hue plane of input image where the object is searched, using the histogram. Threshold the image.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

That is the approximate algorithm of CamShift color object tracker, except for the 3rd step, instead of which CAMSHIFT algorithm is used to locate the object on the back projection given the previous object position.

void **cvCalcBackProjectPatch**(*IplImage** images, CvArr* dst, CvSize patch_size, CvHistogram* hist, int method, float factor*)

Locates a template within image by histogram comparison

**Parameters**

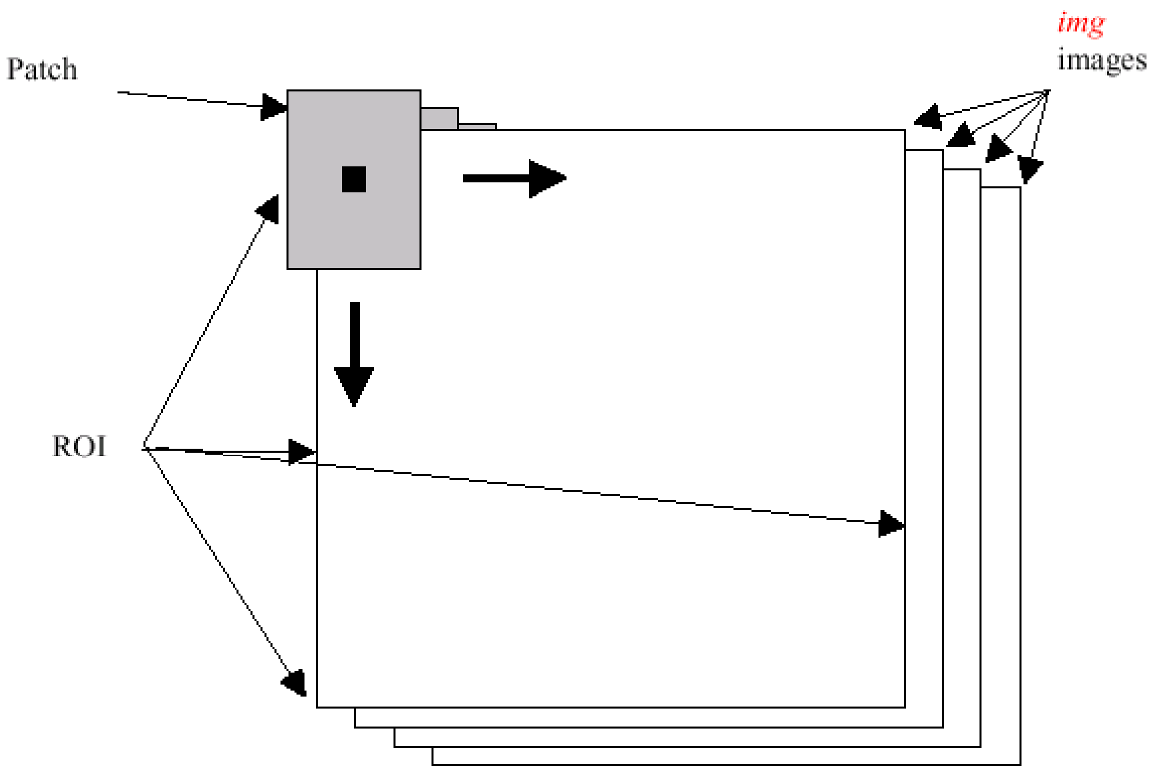
- *images* – Source images (though, you may pass CvMat** as well), all of the same size
- *dst* – Destination image.
- *patch_size* – Size of patch slid though the source images.
- *hist* – Histogram
- *method* – Comparison method, passed to `cvCompareHist` (see description of that function).
- *factor* – Normalization factor for histograms, will affect normalization scale of destination image, pass 1. if unsure.

The function `cvCalcBackProjectPatch` compares histogram, computed over each possible rectangular patch of the specified size in the input `images`, and stores the results to the output map `dst`.

In pseudo-code the operation may be written as:

```
for (x,y) in images (until
(x+patch_size.width-1,y+patch_size.height-1) is inside the images) do
    compute histogram over the ROI
    (x,y,x+patch_size.width,y+patch_size.height) in images
       (see cvCalcHist)
    normalize the histogram using the factor
       (see cvNormalizeHist)
    compare the normalized histogram with input histogram
    hist using the specified method
       (see cvCompareHist)
    store the result to dst(x,y)
end for
 See also a similar function :cfunc:`cvMatchTemplate`.
```

**Back Project Calculation by Patches**




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

Divides one histogram by another

**Parameters**
- *hist1* – first histogram (the divisor).
- *hist2* – second histogram.
- *dst_hist* – destination histogram.
- *scale* – scale factor for the destination histogram.

The function `cvCalcProbDensity` calculates the object probability density from the two histograms as

```
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)
```

So the destination histogram bins are within less than scale.

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

Equalizes histogram of grayscale image

**Parameters**
- *src* – The input 8-bit single-channel image.
- *dst* – The output image of the same size and the same data type as `src`.

The function `cvEqualizeHist` equalizes histogram of the input image using the following algorithm:

1. calculate histogram H for src.
2. normalize histogram, so that the sum of histogram bins is 255.
3. compute integral of the histogram: H?(i) = sum0?j?iH(j)

4.transform the image using H? as a look-up table: dst(x,y)=H?(src(x,y))

The algorithm normalizes brightness and increases contrast of the image.

## Matching

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

Compares template against overlapped image regions

**Parameters**
- *image* – Image where the search is running. It should be 8-bit or 32-bit floating-point.
- *templ* – Searched template; must be not greater than the source image and the same data type as the image.
- *result* – A map of comparison results; single-channel 32-bit floating-point. If `image` is `W`?‘‘H‘‘ and `templ` is `w`?‘‘h‘‘ then `result` must be `W-w+1`?‘‘H-h+1‘‘.
- *method* – Specifies the way the template must be compared with image regions (see below).

The function `cvMatchTemplate` is similar to `cvCalcBackProjectPatch`. It slides through `image`, compares overlapped patches of size `w`?‘‘h‘‘ with `templ` using the specified method and stores the comparison results to `result`. Here are the formulae for the different comparison methods one may use (`I` denotes image, `T` - template, `R` - result. The summation is done over template and/or the image patch: `x'=0..w-1`, `y'=0..h-1`)

```
  method=CV_TM_SQDIFF:
  R(x,y)=sumx',y'[T(x',y')-I(x+x',y+y')]2

  method=CV_TM_SQDIFF_NORMED:
  R(x,y)=sumx',y'[T(x',y')-I(x+x',y+y')]2/sqrt[sumx',y'T(x',y')
  2?sumx',y'I(x+x',y+y')2]

  method=CV_TM_CCORR:
  R(x,y)=sumx',y'[T(x',y')?I(x+x',y+y')]

  method=CV_TM_CCORR_NORMED:
  R(x,y)=sumx',y'[T(x',y')?I(x+x',y+y')]/sqrt[sumx',y'T(x',y')2
  ?sumx',y'I(x+x',y+y')2]

  method=CV_TM_CCOEFF:
  R(x,y)=sumx',y'[T'(x',y')?I'(x+x',y+y')],

  where T'(x',y')=T(x',y') - 1/(w?h)?sumx",y"T(x",y")
        I'(x+x',y+y')=I(x+x',y+y') -
        1/(w?h)?sumx",y"I(x+x",y+y")

  method=CV_TM_CCOEFF_NORMED:
  R(x,y)=sumx',y'[T'(x',y')?I'(x+x',y+y')]/sqrt[sumx',y'T'(x',y
  ')2?sumx',y'I'(x+x',y+y')2]

After the function finishes comparison, the best matches
can be found as global minimums (CV_TM_SQDIFF*) or maximums
(CV_TM_CCORR* and CV_TM_CCOEFF*) using :cfunc:`cvMinMaxLoc` function. In
case of color image and template summation in both numerator and
each sum in denominator is done over all the channels (and separate
mean values are used for each channel).
```

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

Compares two shapes

**Parameters**
- *object1* – First contour or grayscale image
- *object2* – Second contour or grayscale image
- *method* – Comparison method, one of CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3.
- *parameter* – Method-specific parameter (is not used now).

The function `cvMatchShapes` compares two shapes. The 3 implemented methods all use Hu moments (see `cvGetHuMoments`) (A ~ `object1`, B - `object2`)

```
method=CV_CONTOUR_MATCH_I1:
I1(A,B)=sumi=1..7abs(1/mAi - 1/mBi)

method=CV_CONTOUR_MATCH_I2:
I2(A,B)=sumi=1..7abs(mAi - mBi)

method=CV_CONTOUR_MATCH_I3:
I3(A,B)=sumi=1..7abs(mAi - mBi)/abs(mAi)

where
mAi=sign(hAi)?log(hAi),
mBi=sign(hBi)?log(hBi),
hAi, hBi - Hu moments of A and B, respectively.
```

`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*)

Computes "minimal work" distance between two weighted point configurations

```
typedef float (*CvDistanceFunction)(const float* f1, const float* f2, void* userdata)
```

**Parameters**
- *signature1* – First signature, `size1`?‘‘dims+1‘‘ floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used.
- *signature2* – Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different, in this case an extra "dummy" point is added to either `signature1` or `signature2`.
- *distance_type* – Metrics used; `CV_DIST_L1, CV_DIST_L2`, and `CV_DIST_C` stand for one of the standard metrics; `CV_DIST_USER` means that a user-defined function `distance_func` or pre-calculated `cost_matrix` is used.
- *distance_func* – The user-defined distance function. It takes coordinates of two points and returns the distance between the points.
- *cost_matrix* – The user-defined `size1`?‘‘size2‘‘ cost matrix. At least one of `cost_matrix` and `distance_func` must be NULL. Also, if a cost matrix is used, lower boundary (see below) can not be calculated, because it needs a metric function.
- *flow* – The resultant `size1`?‘‘size2‘‘ flow matrix: `flowij` is a flow from i-th point of `signature1` to j-th point of `signature2`
- *lower_bound* – Optional input/output parameter: lower boundary of distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or there is the signatures consist of weights only (i.e. the signature matrices have a single column). User *must* initialize `*lower_bound`. If the calculated distance between mass centers is greater or equal to `*lower_bound` (it means that the signatures are far enough) the function does not calculate EMD. In any case `*lower_bound` is set to the calculated distance between mass centers on return. Thus, if user wants to calculate both distance between mass centers and EMD, `*lower_bound` should be set to 0.

- *userdata* – Pointer to optional data that is passed into the user-defined distance function.

The function cvCalcEMD2 computes earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the application described in **'[RubnerSept98]'_** is multi-dimensional histogram comparison for image retrieval. EMD is a transportation problem that is solved using some modification of simplex algorithm, thus the complexity is exponential in the worst case, though, it is much faster in average. In case of a real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

## 1.2.2 Structural Analysis

### Contour Processing Functions

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

Approximates Freeman chain(s) with polygonal curve

**Parameters**
- *src_seq* – Pointer to the chain that can refer to other chains.
- *storage* – Storage location for the resulting polylines.
- *method* – Approximation method (see the description of the function cvFindContours).
- *parameter* – Method parameter (not used now).
- *minimal_perimeter* – Approximates only those contours whose perimeters are not less than `minimal_perimeter`. Other chains are removed from the resulting structure.
- *recursive* – If not 0, the function approximates all chains that access can be obtained to from `src_seq` by `h_next` or `v_next links`. If 0, the single chain is approximated.

This is a stand-alone approximation routine. The function cvApproxChains works exactly in the same way as cvFindContours with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via `v_next` or `h_next` fields of the returned structure.

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

Initializes chain reader

chain Pointer to chain. reader Chain reader state.

The function cvStartReadChainPoints initializes a special reader (see **'Dynamic Data Structures'_** for more information on sets and sequences).

CvPoint **cvReadChainPoint**(*CvChainPtReader* reader*)

Gets next chain point

**Parameter** *reader* – Chain reader state.

The function cvReadChainPoint returns the current chain point and updates the reader position.

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

Approximates polygonal curve(s) with desired precision

**Parameters**
- *src_seq* – Sequence of array of points.
- *header_size* – Header size of approximated curve[s].
- *storage* – Container for approximated contours. If it is NULL, the input sequences' storage is used.

- *method* – Approximation method; only `CV_POLY_APPROX_DP` is supported, that corresponds to Douglas- Peucker algorithm.
- *parameter* – Method-specific parameter; in case of `CV_POLY_APPROX_DP` it is a desired approximation accuracy.
- *parameter2* – If case if `src_seq` is sequence it means whether the single sequence should be approximated or all sequences on the same level or below `src_seq` (see `cvFindContours` for description of hierarchical contour structures). And if `src_seq` is array (`CvMat*`) of points, the parameter specifies whether the curve is closed (`parameter2!=0`) or not (`parameter2=0`).

The function `cvApproxPoly` approximates one or more curves and returns the approximation result[s]. In case of multiple curves approximation the resultant tree will have the same structure as the input one (1:1 correspondence).

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

Calculates up-right bounding rectangle of point set

**Parameters**
- *points* – Either a 2D point set, represented as a sequence (`CvSeq*`, `CvContour*`) or vector (`CvMat*`) of points, or 8-bit single-channel mask image (`CvMat*`, `IplImage*`), in which non-zero pixels are considered.
- *update* – The update flag. Here is list of possible combination of the flag values and type of `contour`:
  - `points``is CvContour*, ``update``=0: the bounding rectangle is not calculated, but it is read from ``rect` field of the contour header.
  - `points``is CvContour*, ``update``=1: the bounding rectangle is calculated and written to ``rect` field of the contour header. For example, this mode is used by `cvFindContours`.
  - `points``is CvSeq* or CvMat*: ``update` is ignored, the bounding rectangle is calculated and returned.

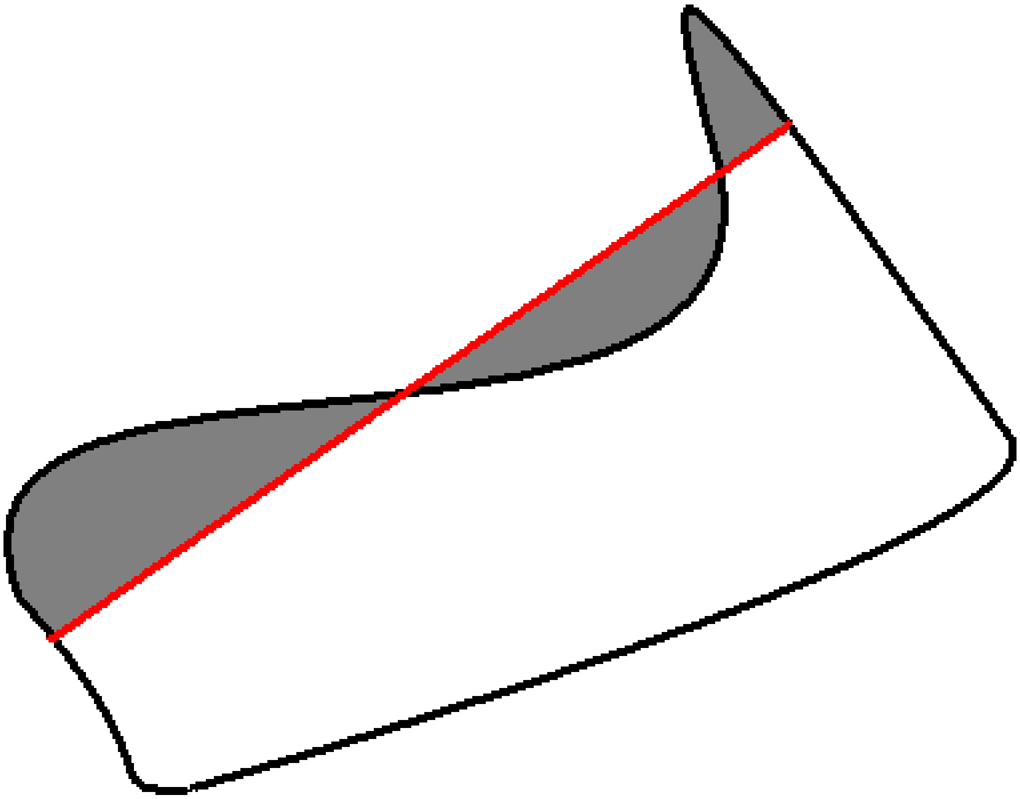
The function `cvBoundingRect` returns the up-right bounding rectangle for 2d point set.

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

Calculates area of the whole contour or contour section

**Parameters**
- *contour* – Contour (sequence or array of vertices).
- *slice* – Starting and ending points of the contour section of interest, by default area of the whole contour is calculated.

The function `cvContourArea` calculates area of the whole contour or contour section. In the latter case the total area bounded by the contour arc and the chord connecting the 2 selected points is calculated as shown on the picture below:



**Note:** Orientation of the contour affects the area sign, thus the function may return `negative` result. Use `fabs()` function from C runtime to get the absolute value of area.

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

Calculates contour perimeter or curve length

**Parameters**
- *curve* – Sequence or array of the curve points.
- *slice* – Starting and ending points of the curve, by default the whole curve length is calculated.
- *is_closed* – Indicates whether the curve is closed or not. There are 3 cases:
  - is_closed=0 - the curve is assumed to be unclosed.
  - is_closed>0 - the curve is assumed to be closed.
  - is_closed<0 - if curve is sequence, the flag CV_SEQ_FLAG_CLOSED of ((CvSeq*)curve)->flags is checked to determine if the curve is closed or not, otherwise (curve is represented by array (CvMat*) of points) it is assumed to be unclosed.

The function `cvArcLength` calculates length or curve as sum of lengths of segments between subsequent points

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

Creates hierarchical representation of contour

**Parameters**
- *contour* – Input contour.
- *storage* – Container for output tree.
- *threshold* – Approximation accuracy.

The function `cvCreateContourTree` creates binary tree representation for the input `contour` and returns the pointer to its root. If the parameter `threshold` is less than or equal to 0, the function creates full binary tree representation. If the threshold is greater than 0, the function creates representation with the precision threshold: if the vertices with the interceptive area of its base line are less than threshold, the tree should not be built any further. The function returns the created tree.

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

Restores contour from tree

**Parameters**
- *tree* – Contour tree.
- *storage* – Container for the reconstructed contour.
- *criteria* – Criteria, where to stop reconstruction.

The function cvContourFromContourTree restores the contour from its binary tree representation. The parameter criteria determines the accuracy and/or the number of tree levels used for reconstruction, so it is possible to build approximated contour. The function returns reconstructed contour.

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

Compares two contours using their tree representations

**Parameters**
- *tree1* – First contour tree.
- *tree2* – Second contour tree.
- *method* – Similarity measure, only CV_CONTOUR_TREES_MATCH_I1 is supported.
- *threshold* – Similarity threshold.

The function cvMatchContourTrees calculates the value of the matching measure for two contour trees. The similarity measure is calculated level by level from the binary tree roots. If at the certain level difference between contours becomes less than threshold, the reconstruction process is interrupted and the current difference is returned.

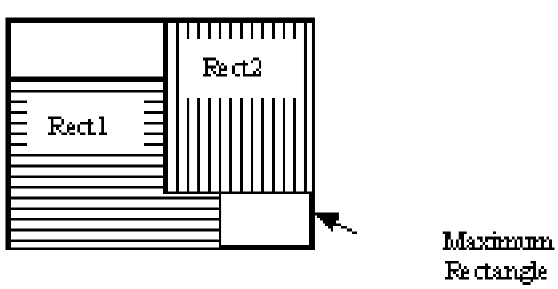
### Computational Geometry

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

Finds bounding rectangle for two given rectangles

**Parameters**
- *rect1* – First rectangle
- *rect2* – Second rectangle

The function cvMaxRect finds minimum area rectangle that contains both input rectangles inside:




**CvBox2D**

Rotated 2D box

```
typedef struct CvBox2D
{
    CvPoint2D32f center;  /* center of the box */
    CvSize2D32f  size;    /* box width and length */
    float angle;          /* angle between the horizontal axis
                             and the first side
                             (i.e. length) in degrees */
}
CvBox2D;
```

CvSeq* **cvPointSeqFromMat**(*int seq_kind, const CvArr* mat, CvContour* contour_header, CvSeqBlock* block*)

Initializes point sequence header from a point vector

**Parameters**

- *seq_kind* – Type of the point sequence: point set (0), a curve (CV_SEQ_KIND_CURVE), closed curve (CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED) etc.
- *mat* – Input matrix. It should be continuous 1-dimensional vector of points, that is, it should have type CV_32SC2 or CV_32FC2.
- *contour_header* – Contour header, initialized by the function.
- *block* – Sequence block header, initialized by the function.

The function cvPointSeqFromMat initializes sequence header to create a "virtual" sequence which elements reside in the specified matrix. No data is copied. The initialized sequence header may be passed to any function that takes a point sequence on input. No extra elements could be added to the sequence, but some may be removed. The function is a specialized variant of cvMakeSeqHeaderForArray and uses the latter internally. It returns pointer to the initialized contour header. Note that the bounding rectangle (field rect of CvContour structure is not initialized by the function. If you need one, use cvBoundingRect.

Here is the simple usage example

```
CvContour header;
CvSeqBlock block;
CvMat* vector = cvCreateMat( 1, 3, CV_32SC2 );

CV_MAT_ELEM( *vector, CvPoint, 0, 0 ) = cvPoint(100,100);
CV_MAT_ELEM( *vector, CvPoint, 0, 1 ) = cvPoint(100,200);
CV_MAT_ELEM( *vector, CvPoint, 0, 2 ) = cvPoint(200,100);

IplImage* img = cvCreateImage( cvSize(300,300), 8, 3 );
cvZero(img);

cvDrawContours( img,
cvPointSeqFromMat(CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED,
   vector, &header, &block), CV_RGB(255,0,0),
   CV_RGB(255,0,0), 0, 3, 8, cvPoint(0,0));
```

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

Finds box vertices

**Parameters**

- *box* – Box
- *pt* – Array of vertices

The function cvBoxPoints calculates vertices of the input 2d box. Here is the function code

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    double angle = box.angle*CV_PI/180.
    float a = (float)cos(angle)*0.5f;
    float b = (float)sin(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;
}
```

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

Fits ellipse to set of 2D points

**Parameter** *points* – Sequence or array of points.

The function `cvFitEllipse` calculates ellipse that fits best (in least-squares sense) to a set of 2D points. The meaning of the returned structure fields is similar to those in `cvEllipse` except that `size` stores the full lengths of the ellipse axises, not half-lengths

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

Fits line to 2D or 3D point set

**Parameters**
- *points* – Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates.
- *dist_type* – The distance used for fitting (see the discussion).
- *param* – Numerical parameter (`C`) for some types of distances, if 0 then some optimal value is chosen.
- *reps, aeps* – Sufficient accuracy for radius (distance between the coordinate origin and the line) and angle, respectively, 0.01 would be a good defaults for both.
- *line* – The output line parameters. In case of 2d fitting it is array of 4 floats `(vx, vy, x0, y0)` where `(vx, vy)` is a normalized vector collinear to the line and `(x0, y0)` is some point on the line. In case of 3D fitting it is array of 6 floats `(vx, vy, vz, x0, y0, z0)` where `(vx, vy, vz)` is a normalized vector collinear to the line and `(x0, y0, z0)` is some point on the line.

The function `cvFitLine` fits line to 2D or 3D point set by minimizing sumi?(ri), where ri is distance between i-th point and the line and ?(r) is a distance function, one of

```
dist_type=CV_DIST_L2 (L2):
?(r)=r2/2 (the simplest and the fastest least-squares method)

dist_type=CV_DIST_L1 (L1):
?(r)=r

dist_type=CV_DIST_L12 (L1-L2):
?(r)=2?[sqrt(1+r2/2) - 1]
```

```
dist_type=CV_DIST_FAIR (Fair):
?(r)=C2?[r/C - log(1 + r/C)],  C=1.3998

dist_type=CV_DIST_WELSCH (Welsch):
?(r)=C2/2?[1 - exp(-(r/C)2)],  C=2.9846

dist_type=CV_DIST_HUBER (Huber):
?(r)= r2/2,   if r < C
      C?(r-C/2),   otherwise;   C=1.345
```

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

Finds convex hull of point set

**Parameters**
- *points* – Sequence or array of 2D points with 32-bit integer or floating-point coordinates.
- *hull_storage* – The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified: the number of columns/rows is truncated down to the hull size.
- *orientation* – Desired orientation of convex hull: `CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`.
- *return_points* – If non-zero, the points themselves will be stored in the hull instead of indices if `hull_storage` is array, or pointers if `hull_storage` is memory storage.

The function `cvConvexHull2` finds convex hull of 2D point set using Sklansky's algorithm. If `hull_storage` is memory storage, the function creates a sequence containing the hull points or pointers to them, depending on `return_points` value and returns the sequence on output.

**Example:** Building convex hull for a sequence or array of points

```
#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
    }
}
```

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

Tests contour convex

**Parameter** *contour* – Tested contour (sequence or array of points).

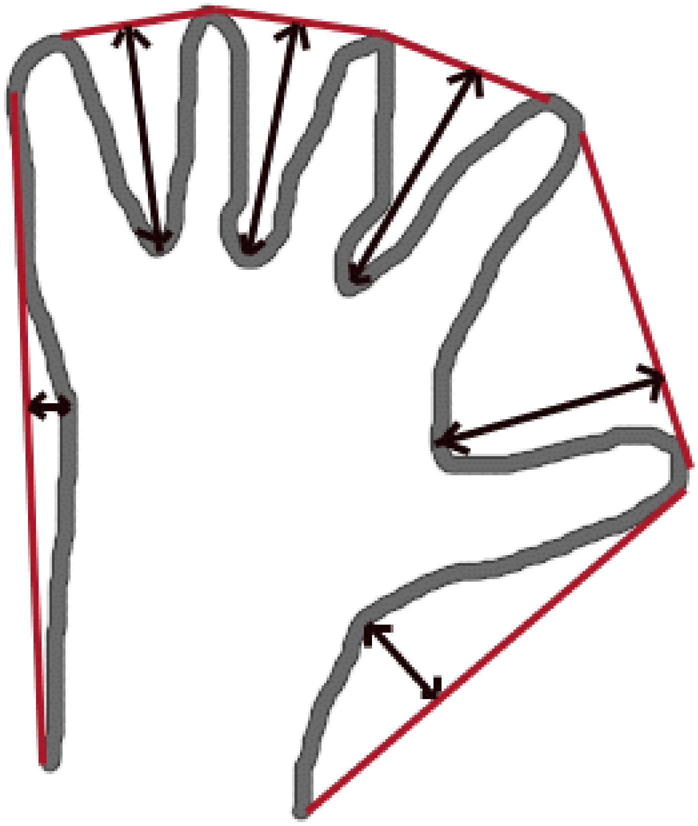
The function cvCheckContourConvexity tests whether the input contour is convex or not. The contour must be simple, i.e. without self- intersections.

**CvConvexityDefect**

Structure describing a single contour convexity detect

```
typedef struct CvConvexityDefect
{
    CvPoint* start; /* point of the contour where the defect begins */
    CvPoint* end; /* point of the contour where the defect ends */
    CvPoint* depth_point; /* the farthest from the convex hull point
    within the defect */
    float depth; /* distance between the farthest point and the convex
    hull */
} CvConvexityDefect;
```

Picture. Convexity defects of hand contour.



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

Finds convexity defects of contour

**Parameters**
- *contour* – Input contour.
- *convexhull* – Convex hull obtained using `cvConvexHull2` that should contain pointers or indices to the contour points, not the hull points themselves, i.e. `return_points` parameter in `cvConvexHull2` should be 0.
- *storage* – Container for output sequence of convexity defects. If it is NULL, contour or hull (in that order) storage is used.

The function `cvConvexityDefects` finds all convexity defects of the input contour and returns a sequence of the `CvConvexityDefect` structures.

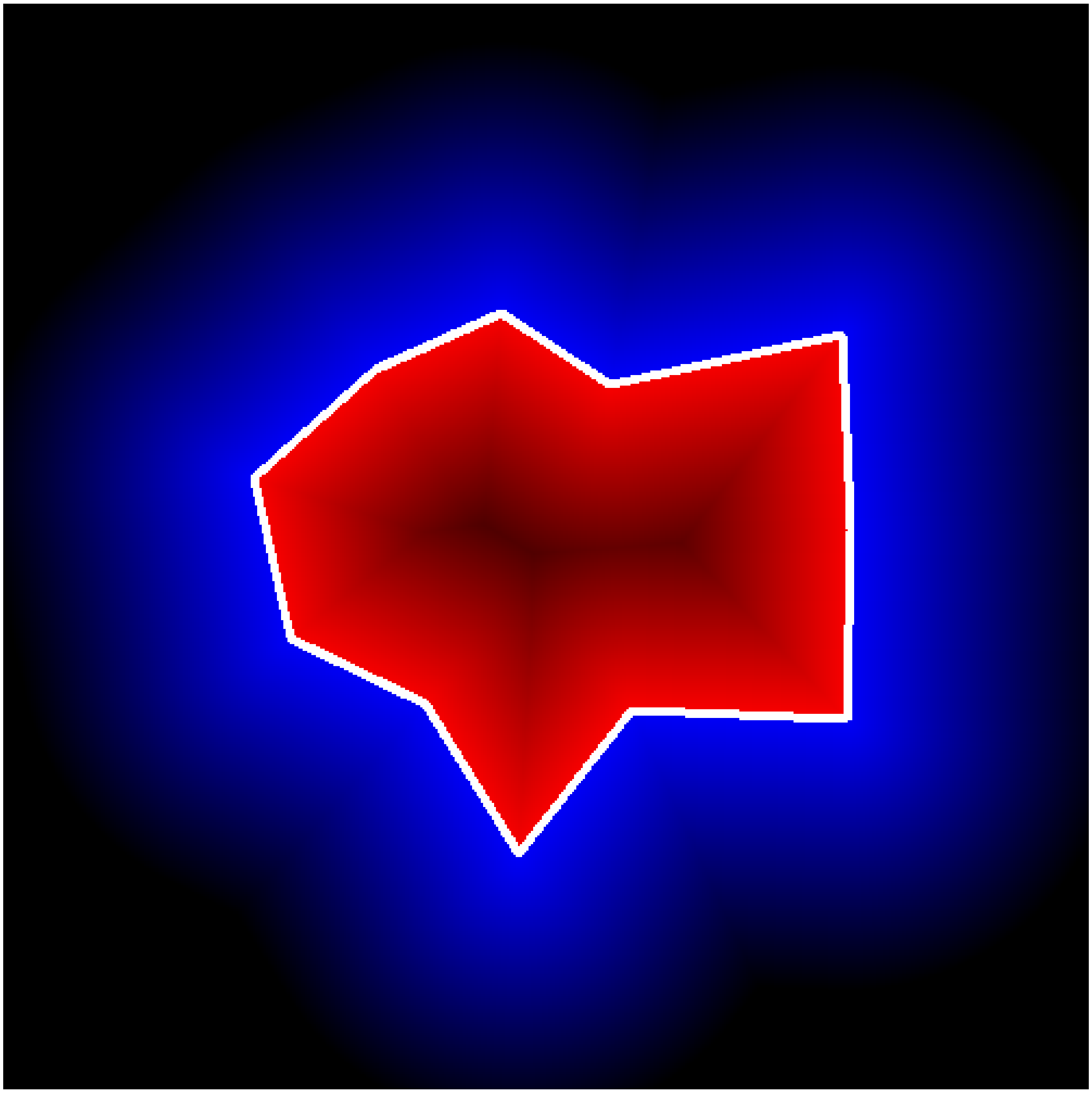
double **cvPointPolygonTest**(*const CvArr* contour, CvPoint2D32f pt, int measure_dist*)

Point in contour test

**Parameters**
- *contour* – Input contour.
- *pt* – The point tested against the contour.
- *measure_dist* – If it is non-zero, the function estimates distance from the point to the nearest contour edge.

The function `cvPointPolygonTest` determines whether the point is inside contour, outside, or lies on an edge (or coincides with a vertex). It returns positive, negative or zero value, correspondingly. When `measure_dist=0`, the return value is +1, -1 and 0, respectively. When ?‘‘measure_dist0‘‘, it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



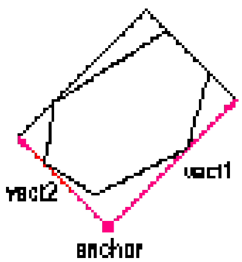
CvBox2D **cvMinAreaRect2** (*const CvArr* points, CvMemStorage* storage=NULL*)

Finds circumscribed rectangle of minimal area for given 2D point set

**Parameters**
- *points* – Sequence or array of points.
- *storage* – Optional temporary memory storage.

The function `cvMinAreaRect2` finds a circumscribed rectangle of the minimal area for 2D point set by building convex hull for the set and applying rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour




int **cvMinEnclosingCircle**(*const CvArr* points, CvPoint2D32f* center, float* radius*)
Finds circumscribed circle of minimal area for given 2D point set

**Parameters**
- *points* – Sequence or array of 2D points.
- *center* – Output parameter. The center of the enclosing circle.
- *radius* – Output parameter. The radius of the enclosing circle.

The function `cvMinEnclosingCircle` finds the minimal circumscribed circle for 2D point set using iterative algorithm. It returns nonzero if the resultant circle contains all the input points and zero otherwise (i.e. algorithm failed).

void **cvCalcPGH**(*const CvSeq* contour, CvHistogram* hist*)
Calculates pair-wise geometrical histogram for contour

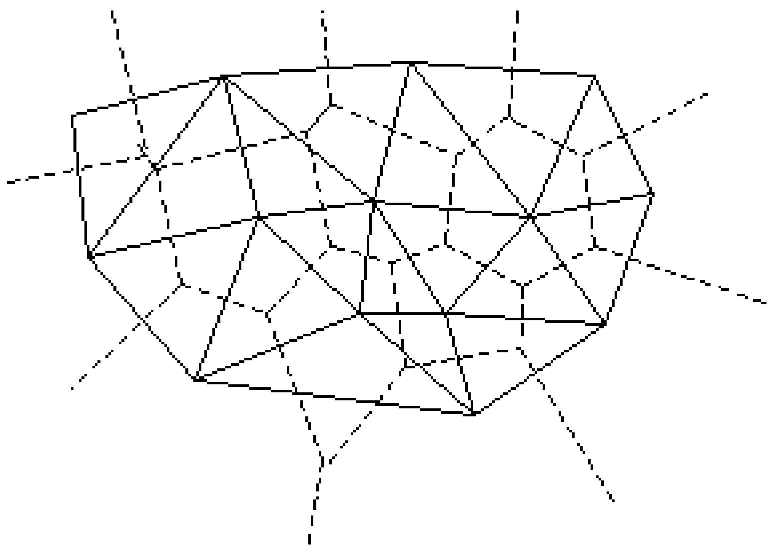
**Parameters**
- *contour* – Input contour. Currently, only integer point coordinates are allowed.
- *hist* – Calculated histogram; must be two-dimensional.

The function `cvCalcPGH` calculates 2D pair-wise geometrical histogram (PGH), described in **'[Iivarinen97]'_**, for the contour. The algorithm considers every pair of the contour edges. The angle between the edges and the minimum/maximum distances are determined for every pair. To do this each of the edges in turn is taken as the base, while the function loops through all the other edges. When the base edge and any other edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented (that is, the histogram is transposed relatively to [Iivarninen97] definition). The histogram can be used for contour matching.

### Planar Subdivisions

Planar subdivision is a subdivision of a plane into a set of non- overlapped regions (facets) that cover the whole plane. The structure `CvSubdiv2D` describes a subdivision built on 2d point set, where the points are linked together and form a planar graph, which, together with a few edges connecting exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists dual subdivision there facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called virtual point below) of dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dot lines



OpenCV subdivides plane into triangles using Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case the dual subdivision is Voronoi diagram of input 2d point set. The subdivisions can be used for 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) etc.

**CvSubdiv2D**

Planar subdivision structure

```
#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;
```

**CvQuadEdge2D**

Quad-edge of planar subdivision

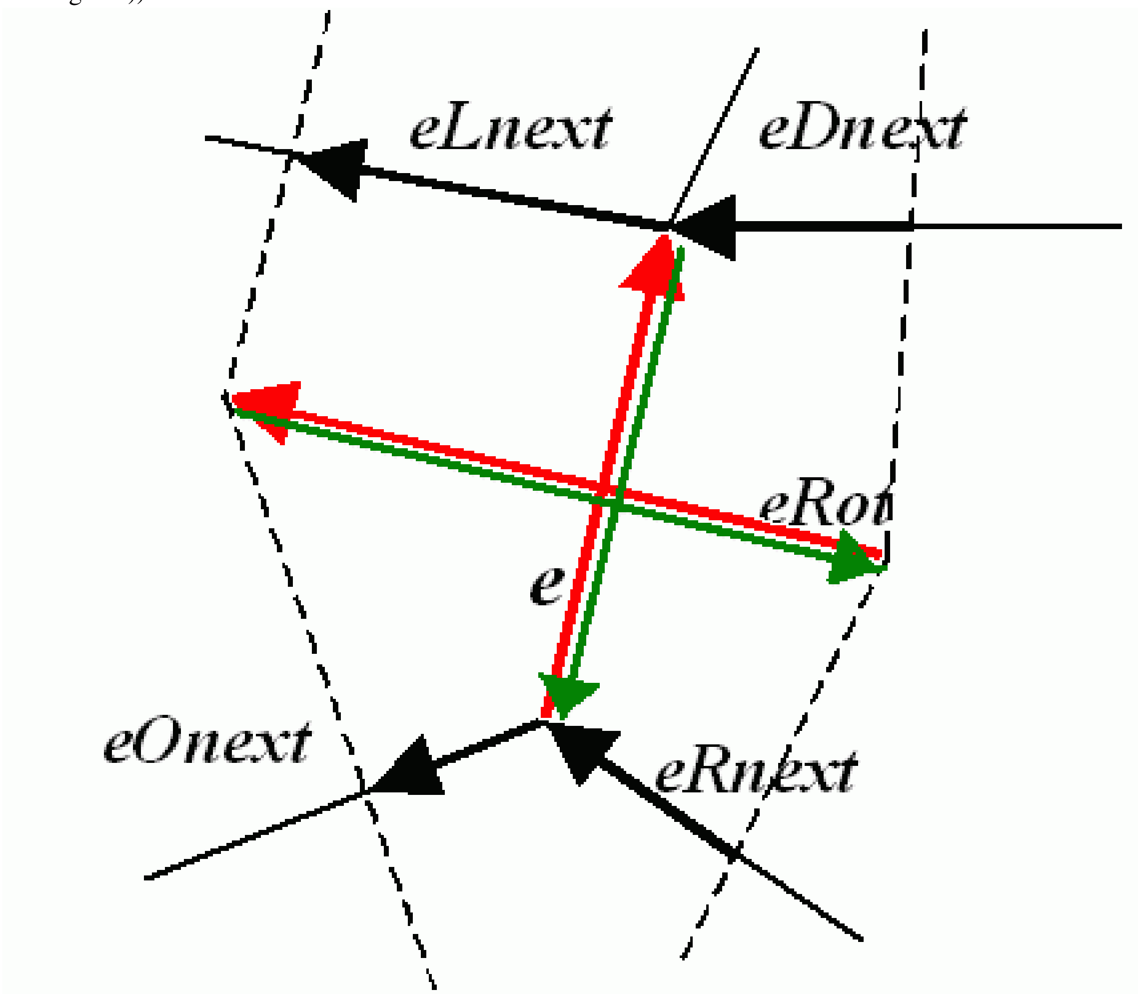
```
/* one of edges within quad-edge, lower 2 bits is index (0..3)
   and upper bits are quad-edge pointer */
typedef long CvSubdiv2DEdge;

/* quad-edge structure fields */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
```

```
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```

Quad-edge is a basic element of subdivision, it contains four edges (e, eRot (in red) and reversed e & eRot (in green)):




**`CvSubdiv2DPoint`**

Point of original or dual subdivision

```
#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;
```

`CvSubdiv2DEdge` **`cvSubdiv2DGetEdge`** (*CvSubdiv2DEdge edge, CvNextEdgeType type*)

Returns one of edges related to given

**Parameters**
- *edge* – Subdivision edge (not a quad-edge)
- *type* – Specifies, which of related edges to return, one of:
  - CV_NEXT_AROUND_ORG - next around the edge origin (`eOnext` on the picture above if `e` is the input edge)
  - CV_NEXT_AROUND_DST - next around the edge vertex (`eDnext`)
  - CV_PREV_AROUND_ORG - previous around the edge origin (reversed `eRnext`)
  - CV_PREV_AROUND_DST - previous around the edge destination (reversed `eLnext`)
  - CV_NEXT_AROUND_LEFT - next around the left facet (`eLnext`)
  - CV_NEXT_AROUND_RIGHT - next around the right facet (`eRnext`)
  - CV_PREV_AROUND_LEFT - previous around the left facet (reversed `eOnext`)
  - CV_PREV_AROUND_RIGHT - previous around the right facet (reversed `eDnext`)

The function `cvSubdiv2DGetEdge` returns one the edges related to the input edge.

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

Returns another edge of the same quad-edge

**Parameters**
- *edge* – Subdivision edge (not a quad-edge)
- *type* – Specifies, which of edges of the same quad-edge as the input one to return, one of:
  - 0 - the input edge (`e` on the picture above if `e` is the input edge)
  - 1 - the rotated edge (`eRot`)
  - 2 - the reversed edge (reversed `e` (in green))
  - 3 - the reversed rotated edge (reversed `eRot` (in green))

The function `cvSubdiv2DRotateEdge` returns one the edges of the same quad-edge as the input edge.

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

Returns edge origin

**Parameter** *edge* – Subdivision edge (not a quad-edge)

The function `cvSubdiv2DEdgeOrg` returns the edge origin. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using function `cvCalcSubdivVoronoi2D`.

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

Returns edge destination

**Parameter** *edge* – Subdivision edge (not a quad-edge)

The function `cvSubdiv2DEdgeDst` returns the edge destination. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using function `cvCalcSubdivVoronoi2D`.

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

Creates empty Delaunay triangulation

**Parameters**
- *rect* – Rectangle that includes all the 2d points that are to be added to subdivision.
- *storage* – Container for subdivision.

The function `cvCreateSubdivDelaunay2D` creates an empty Delaunay subdivision, where 2d points can be added further using function `cvSubdivDelaunay2DInsert`. All the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

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

Inserts a single point to Delaunay triangulation

**Parameters**
- *subdiv* – Delaunay subdivision created by function `cvCreateSubdivDelaunay2D`.
- *pt* – Inserted point.

The function `cvSubdivDelaunay2DInsert` inserts a single point to subdivision and modifies the subdivision topology appropriately. If a points with same coordinates exists already, no new points is added. The function returns pointer to the allocated point. No virtual points coordinates is calculated at this stage.

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

Inserts a single point to Delaunay triangulation

**Parameters**
- *subdiv* – Delaunay or another subdivision.
- *pt* – The point to locate.
- *edge* – The output edge the point falls onto or right to.
- *vertex* – Optional output vertex double pointer the input point coincides with.

The function `cvSubdiv2DLocate` locates input point within subdivision. There are 5 cases:

- point falls into some facet. The function returns CV_PTLOC_INSIDE and `*edge` will contain one of edges of the facet.
- point falls onto the edge. The function returns CV_PTLOC_ON_EDGE and `*edge` will contain this edge.
- point coincides with one of subdivision vertices. The function returns CV_PTLOC_VERTEX and `*vertex` will contain pointer to the vertex.
- point is outside the subdivision reference rectangle. The function returns CV_PTLOC_OUTSIDE_RECT and no pointers is filled.
- one of input arguments is invalid. Runtime error is raised or, if silent or "parent" error processing mode is selected, CV_PTLOC_ERROR is returned.

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

Finds the closest subdivision vertex to given point

**Parameters**
- *subdiv* – Delaunay or another subdivision.
- *pt* – Input point.

The function `cvFindNearestPoint2D` is another function that locates input point within subdivision. It finds subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using `cvSubdiv2DLocate`) is used as a starting point. The function returns pointer to the found subdivision vertex.

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

Calculates coordinates of Voronoi diagram cells

**Parameter** *subdiv* – Delaunay subdivision, where all the points are added already.

The function `cvCalcSubdivVoronoi2D` calculates coordinates of virtual points. All virtual points corresponding to some vertex of original subdivision form (when connected together) a boundary of Voronoi cell of that point.

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

Removes all virtual points

**Parameter** *subdiv* – Delaunay subdivision.

The function `cvClearSubdivVoronoi2D` removes all virtual points. It is called internally in `cvCalcSubdivVoronoi2D` if the subdivision was modified after previous call to the function.

**Note:** There are a few other lower-level functions that work with planar subdivisions, see cv.h and the sources. Demo script delaunay.c that builds Delaunay triangulation and Voronoi diagram of random 2d point set can be found at opencv/samples/c.

### 1.2.3 Motion Analysis and Object Tracking

**Contents**



#### Accumulation of Background Statistics

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

Adds frame to accumulator

**Parameters**
- *image* – Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently).
- *sum* – Accumulator of the same number of channels as input image, 32-bit floating-point.
- *mask* – Optional operation mask.

The function `cvAcc` adds the whole image `image` or its selected region to accumulator `sum`

$$sum(x,y) = sum(x,y) + image(x,y) \quad \text{if} \quad mask(x,y)! = 0$$

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

Adds the square of source image to accumulator

**Parameters**
- *image* – Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently).
- *sqsum* – Accumulator of the same number of channels as input image, 32-bit or 64-bit floating-point.
- *mask* – Optional operation mask.

The function `cvSquareAcc` adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum`

```
sqsum(x,y) = sqsum(x,y) + image(x,y)2 if mask(x,y) != 0
```

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

Adds product of two input images to accumulator

**Parameters**
- *image1* – First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently).
- *image2* – Second input image, the same format as the first one.
- *acc* – Accumulator of the same number of channels as input images, 32-bit or 64-bit floating-point.
- *mask* – Optional operation mask.

The function `cvMultiplyAcc` adds product of 2 images or their selected regions to accumulator `acc`

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

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

Updates running average

**Parameters**
- *image* – Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently).
- *acc* – Accumulator of the same number of channels as input image, 32-bit or 64-bit floating-point.
- *alpha* – Weight of input image.
- *mask* – Optional operation mask.

The function `cvRunningAvg` calculates weighted sum of input image `image` and the accumulator `acc` so that `acc` becomes a running average of frame sequence

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

where ? (alpha) regulates update speed (how fast accumulator forgets about previous frames).

### Motion Templates

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

Updates motion history image by moving silhouette

**Parameters**
- *silhouette* – Silhouette mask that has non-zero pixels where the motion occurs.
- *mhi* – Motion history image, that is updated by the function (single-channel, 32-bit floating-point)
- *timestamp* – Current time in milliseconds or other units.
- *duration* – Maximal duration of motion track in the same units as `timestamp`.

The function `cvUpdateMotionHistory` updates the motion history image as following

```
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
```

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

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

Calculates gradient orientation of motion history image

**Parameters**
- *mhi* – Motion history image.
- *mask* – Mask image; marks pixels where motion gradient data is correct. Output parameter.
- *orientation* – Motion gradient orientation image; contains angles from 0 to ~360?.
- *delta1, delta2* – The function finds minimum (m(x,y)) and maximum (M(x,y)) mhi values over each pixel (x,y) neighborhood and assumes the gradient is valid only if ::min(delta1,delta2) <= M(x,y)-m(x,y) <= max(delta1,delta2).
- *aperture_size* – Aperture size of derivative operators used by the function: CV_SCHARR, 1, 3, 5 or 7 (see `cvSobel`).

The function `cvCalcMotionGradient` calculates the derivatives `Dx` and `Dy` of `mhi` and then calculates gradient orientation as

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

where both `Dx(x,y)` ' and `Dy(x,y)` ' signs are taken into account (as in `cvCartToPolar` function). After that `mask` is filled to indicate where the orientation is valid (see `delta1` and `delta2` description).

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

Calculates global motion orientation of some selected region

**Parameters**
- *orientation* – Motion gradient orientation image; calculated by the function `cvCalcMotionGradient`.
- *mask* – Mask image. It may be a conjunction of valid gradient mask, obtained with `cvCalcMotionGradient` and mask of the region, whose direction needs to be calculated.
- *mhi* – Motion history image.
- *timestamp* – Current time in milliseconds or other units, it is better to store time passed to `cvUpdateMotionHistory` before and reuse it here, because running `cvUpdateMotionHistory` and `cvCalcMotionGradient` on large images may take some time.
- *duration* – Maximal duration of motion track in milliseconds, the same as in `cvUpdateMotionHistory`.

The function `cvCalcGlobalOrientation` calculates the general motion direction in the selected region and returns the angle between 0? and 360?. At first the function builds the orientation histogram and finds the basic orientation as a coordinate of the histogram maximum. After that the function calculates the shift relative to the basic orientation as a weighted sum of all orientation vectors: the more recent is the motion, the greater is the weight. The resultant angle is a circular sum of the basic orientation and the shift.

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

Segments whole motion into separate moving parts

**Parameters**
- *mhi* – Motion history image.
- *seg_mask* – Image where the mask found should be stored, single-channel, 32-bit floating-point.
- *storage* – Memory storage that will contain a sequence of motion connected components.
- *timestamp* – Current time in milliseconds or other units.
- *seg_thresh* – Segmentation threshold; recommended to be equal to the interval between motion history "steps" or greater.

The function `cvSegmentMotion` finds all the motion segments and marks them in `seg_mask` with individual values each (1,2,...). It also returns a sequence of `CvConnectedComp` structures, one per each motion components. After than the motion direction for every component can be calculated with `cvCalcGlobalOrientation` using extracted mask of the particular component (using `cvCmp`)

### Object Tracking

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

Finds object center on back projection

**Parameters**
- *prob_image* – Back projection of object histogram (see `cvCalcBackProject`).
- *window* – Initial search window.

- *criteria* – Criteria applied to determine when the window search should be finished.
- *comp* – Resultant structure that contains converged search window coordinates (`comp->rect` field) and sum of all pixels inside the window (`comp->area` field).

The function `cvMeanShift` iterates to find the object center given its back projection and initial position of search window. The iterations are made until the search window center moves by less than the given value and/or until the function has done the maximum number of iterations. The function returns the number of iterations made.

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

Finds object center, size, and orientation

**Parameters**
- *prob_image* – Back projection of object histogram (see `cvCalcBackProject`).
- *window* – Initial search window.
- *criteria* – Criteria applied to determine when the window search should be finished.
- *comp* – Resultant structure that contains converged search window coordinates (`comp->rect` field) and sum of all pixels inside the window (`comp->area` field).
- *box* – Circumscribed box for the object. If not `NULL`, contains object size and orientation.

The function `cvCamShift` implements CAMSHIFT object tracking algorithm (**'[Bradski98]'_**). First, it finds an object center using `cvMeanShift` and, after that, calculates the object size and orientation. The function returns number of iterations made within `cvMeanShift`.

`CvCamShiftTracker` class declared in cv.hpp implements color object tracker that uses the function.

`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*)

Changes contour position to minimize its energy

**Parameters**
- *image* – The source image or external energy field.
- *points* – Contour points (snake).
- *length* – Number of points in the contour.
- *alpha* – Weight[s] of continuity energy, single float or array of `length` floats, one per each contour point.
- *beta* – Weight[s] of curvature energy, similar to `alpha`.
- *gamma* – Weight[s] of image energy, similar to `alpha`.
- *coeff_usage* – Variant of usage of the previous three parameters:
  - `CV_VALUE`'indicates that each of ``alpha, beta, gamma`` is a pointer to a single value to be used for all points;
  - `CV_ARRAY`'indicates that each of ``alpha, beta, gamma`` is a pointer to an array of coefficients different for all the points of the snake. All the arrays must have the size equal to the contour size.
- *win* – Size of neighborhood of every point used to search the minimum, both `win.width` and `win.height` must be odd.
- *criteria* – Termination criteria.
- *calc_gradient* – Gradient flag. If not 0, the function calculates gradient magnitude for every image pixel and considers it as the energy field, otherwise the input image itself is considered.

The function `cvSnakeImage` updates snake in order to minimize its total energy that is a sum of internal energy that depends on contour shape (the smoother contour is, the smaller internal energy is) and external energy that depends on the energy field and reaches minimum at the local energy extremums that correspond to the image edges in case of image gradient.

The parameter `criteria.epsilon` is used to define the minimal number of points that must be moved during any iteration to keep the iteration process running.

If at some iteration the number of moved points is less than `criteria.epsilon` or the function performed `criteria.max_iter` iterations, the function terminates.

### Optical Flow

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

Calculates optical flow for two images

**Parameters**
- *prev* – First image, 8-bit, single-channel.
- *curr* – Second image, 8-bit, single-channel.
- *use_previous* – Uses previous (input) velocity field.
- *velx* – Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.
- *vely* – Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.
- *lambda* – Lagrangian multiplier.
- *criteria* – Criteria of termination of velocity computing.

The function `cvCalcOpticalFlowHS` computes flow for every pixel of the first input image using Horn & Schunck algorithm **'[Horn81]'_**.

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

Calculates optical flow for two images

**Parameters**
- *prev* – First image, 8-bit, single-channel.
- *curr* – Second image, 8-bit, single-channel.
- *win_size* – Size of the averaging window used for grouping pixels.
- *velx* – Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.
- *vely* – Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.

The function `cvCalcOpticalFlowLK` computes flow for every pixel of the first input image using Lucas & Kanade algorithm **'[Lucas81]'_**.

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

Calculates optical flow for two images by block matching method

**Parameters**
- *prev* – First image, 8-bit, single-channel.
- *curr* – Second image, 8-bit, single-channel.
- *block_size* – Size of basic blocks that are compared.
- *shift_size* – Block coordinate increments.
- *max_range* – Size of the scanned neighborhood in pixels around block.
- *use_previous* – Uses previous (input) velocity field.
- *velx* – Horizontal component of the optical flow of floor((prev->width - block_size.width)/shiftSize.width) ? floor((prev->height - block_size.height)/shiftSize.height) size, 32-bit floating-point, single- channel.
- *vely* – Vertical component of the optical flow of the same size `velx`, 32-bit floating-point, single-channel.

The function `cvCalcOpticalFlowBM` calculates optical flow for overlapped blocks ?‘‘block_size.widthblock_size.height‘‘ pixels each, thus the velocity fields are smaller than the original images. For every block in `prev` the functions tries to find a similar block in `curr` in some neighborhood of the original block or shifted by (velx(x0,y0),vely(x0,y0)) block as has been calculated by previous function call (if `use_previous=1`)

`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*)

Calculates optical flow for a sparse feature set using iterative Lucas-Kanade method in pyramids

**Parameters**
- *prev* – First frame, at time `t`.
- *curr* – Second frame, at time `t + dt` .
- *prev_pyr* – Buffer for the pyramid for the first frame. If the pointer is not `NULL` , the buffer must have a sufficient size to store the pyramid from level `1` to level #‘‘level‘‘ ; the total size of `(image_width+8)*image_height/3` bytes is sufficient.
- *curr_pyr* – Similar to `prev_pyr`, used for the second frame.
- *prev_features* – Array of points for which the flow needs to be found.
- *curr_features* – Array of 2D points containing calculated new positions of input features in the second image.
- *count* – Number of feature points.
- *win_size* – Size of the search window of each pyramid level.
- *level* – Maximal pyramid level number. If `0` , pyramids are not used (single level), if `1` , two levels are used, etc.
- *status* – Array. Every element of the array is set to `1` if the flow for the corresponding feature has been found, `0` otherwise.
- *track_error* – Array of double numbers containing difference between patches around the original and moved points. Optional parameter; can be `NULL`.
- *criteria* – Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped.
- *flags* – Miscellaneous flags:
  - `CV_LKFLOW_PYR_A_READY`, pyramid for the first frame is pre-calculated before the call;
  - `CV_LKFLOW_PYR_B_READY`, pyramid for the second frame is pre-calculated before the call;
  - `CV_LKFLOW_INITIAL_GUESSES`, array B contains initial coordinates of features before the function call.

The function `cvCalcOpticalFlowPyrLK` implements sparse iterative version of Lucas-Kanade optical flow in pyramids (**‘[Bouguet00]‘_**). It calculates coordinates of the feature points on the current video frame given their coordinates on the previous frame. The function finds the coordinates with sub-pixel accuracy.

Both parameters `prev_pyr` and `curr_pyr` comply with the following rules: if the image pointer is 0, the function allocates the buffer internally, calculates the pyramid, and releases the buffer after processing. Otherwise, the function calculates the pyramid and stores it in the buffer unless the flag `CV_LKFLOW_PYR_A[B]_READY` is set. The image should be large enough to fit the Gaussian pyramid data. After the function call both pyramids are calculated and the readiness flag for the corresponding image can be set in the next call (i.e., typically, for all the image pairs except the very first one `CV_LKFLOW_PYR_A_READY` is set).

### Feature Matching

`CvFeatureTree*` **cvCreateFeatureTree**(*CvMat* desc*)

Constructs a tree of feature vectors

**Parameter** *desc* – n x d matrix of n d-dimensional feature vectors (CV_32FC1 or CV_64FC1).

**The function `cvCreateFeatureTree` constructs a balanced kd-tree** index of the given feature vectors. The lifetime of the desc matrix

must exceed that of the returned tree. I.e., no copy is made of the vectors.

`void` **`cvReleaseFeatureTree`**`(`*CvFeatureTree* tr*`)`

Destroys a tree of feature vectors

**Parameter** *tr* – pointer to tree being destroyed.

The function `cvReleaseFeatureTree` deallocates the given kd-tree.

`void` **`cvFindFeatures`**`(`*CvFeatureTree* tr, CvMat* desc, CvMat* results, CvMat* dist, int k=2, int emax=20*`)`

Finds approximate k nearest neighbors of given vectors using best- bin-first search

**Parameters**
- *tr* – pointer to kd-tree index of reference vectors.
- *desc* – m x d matrix of (row-)vectors to find the nearest neighbors of.
- *results* – m x k set of row indices of matching vectors (referring to matrix passed to cvCreateFeatureTree). Contains -1 in some columns if fewer than k neighbors found.
- *dist* – m x k matrix of distances to k nearest neighbors.
- *k* – The number of neighbors to find.
- *emax* – The maximum number of leaves to visit.

The function `cvFindFeatures` finds (with high probability) the `k` nearest neighbors in `tr` for each of the given (row-)vectors in `desc`, using best-bin-first searching (**'[Beis97]'_**). The complexity of the entire operation is at most `O(m*emax*log2(n))`, where `n` is the number of vectors in the tree.

`int` **`cvFindFeaturesBoxed`**`(`*CvFeatureTree* tr, CvMat* bounds_min, CvMat* bounds_max, CvMat* results*`)`

Orthogonal range search

**Parameters**
- *tr* – pointer to kd-tree index of reference vectors.
- *bounds_min* – 1 x d or d x 1 vector (CV_32FC1 or CV_64FC1) giving minimum value for each dimension.
- *bounds_max* – 1 x d or d x 1 vector (CV_32FC1 or CV_64FC1) giving maximum value for each dimension.
- *results* – 1 x m or m x 1 vector (CV_32SC1) to contain output row indices (referring to matrix passed to cvCreateFeatureTree).

The function `cvFindFeaturesBoxed` performs orthogonal range seaching on the given kd-tree. That is, it returns the set of vectors `v` in `tr` that satisfy `bounds_min[i] <= v[i] <= bounds_max[i], 0 <= i < d`, where `d` is the dimension of vectors in the tree. The function returns the number of such vectors found.

### Estimators

**`CvKalman`**

Kalman filter state

```
typedef struct CvKalman
{
    int MP;                     /* number of measurement vector
    dimensions */
    int DP;                     /* number of state vector dimensions */
    int CP;                     /* number of control vector dimensions */
```

```
    /* backward compatibility fields */
#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;           /* predicted state (x'(k)):
x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* corrected state (x(k)):
x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* state transition matrix (A) */
    CvMat* control_matrix;      /* control matrix (B)
                                   (it is
                                   not used if there is no control)*/
    CvMat* measurement_matrix;  /* measurement matrix (H) */
    CvMat* process_noise_cov;   /* process noise covariance matrix (Q) */
    CvMat* measurement_noise_cov; /* measurement noise covariance matrix
    (R) */
    CvMat* error_cov_pre;       /* priori error estimate covariance
    matrix (P'(k)):
P'(k)=A*P(k-1)*At + Q)*/
    CvMat* gain;                /* Kalman gain matrix (K(k)):
K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* posteriori error estimate covariance
    matrix (P(k)):
P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

The structure `CvKalman` is used to keep Kalman filter state. It is created by `cvCreateKalman` function, updated by `cvKalmanPredict` and `cvKalmanCorrect` functions and released by `cvReleaseKalman` functions. Normally, the structure is used for standard Kalman filter (notation and the formulae below are borrowed from the excellent Kalman tutorial **'[Welch95]'_**)

```
xk=A?xk-1+B?uk+wk
zk=H?xk+vk,
```

where

```
xk (xk-1) - state of the system at the moment k (k-1)
zk - measurement of the system state at the moment k
uk - external control applied at the moment k

wk and vk are normally-distributed process and measurement
```

```
noise, respectively:
p(w) ~ N(0,Q)
p(v) ~ N(0,R),

that is,
Q - process noise covariance matrix, constant or variable,
R - measurement noise covariance matrix, constant or variable
```

In case of standard Kalman filter, all the matrices: A, B, H, Q and R are initialized once after `CvKalman` structure is allocated via `cvCreateKalman`. However, the same structure and the same functions may be used to simulate extended Kalman filter by linearizing extended Kalman filter equation in the current system state neighborhood, in this case A, B, H (and, probably, Q and R) should be updated on every step.

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

Allocates Kalman filter structure

**Parameters**
- *dynam_params* – dimensionality of the state vector
- *measure_params* – dimensionality of the measurement vector
- *control_params* – dimensionality of the control vector

The function `cvCreateKalman` allocates `CvKalman` and all its matrices and initializes them somehow.

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

Deallocates Kalman filter structure

**Parameter** *kalman* – double pointer to the Kalman filter structure.

The function `cvReleaseKalman` releases the structure `CvKalman` and all underlying matrices.

`const CvMat*` **cvKalmanPredict** (*CvKalman* kalman, const CvMat* control=NULL*)

Estimates subsequent model state

```
#define cvKalmanUpdateByTime cvKalmanPredict
```

**Parameters**
- *kalman* – Kalman filter state.
- *control* – Control vector (uk), should be NULL iff there is no external control (`control_params=0`).

The function `cvKalmanPredict` estimates the subsequent stochastic model state by its current state and stores it at `kalman->state_pre`

```
x'k=A?xk+B?uk
P'k=A?Pk-1*AT + Q,
```

where

```
x'k is predicted state (kalman->state_pre),
xk-1 is corrected state on the previous step
(kalman->state_post)
                (should be initialized
                somehow in the beginning, zero vector by default),
uk is external control (control parameter),
P'k is priori error covariance matrix (kalman->error_cov_pre)
Pk-1 is posteriori error covariance matrix on the previous
step (kalman->error_cov_post)
                (should be initialized
                somehow in the beginning, identity matrix by
```

```
                default),
The function returns the estimated state.
```

const CvMat* **cvKalmanCorrect** (*CvKalman* kalman, const CvMat* measurement*)

Adjusts model state

```
#define cvKalmanUpdateByMeasurement cvKalmanCorrect
```

**Parameters**
- *kalman* – Pointer to the structure to be updated.
- *measurement* – Pointer to the structure CvMat containing the measurement vector.

The function `cvKalmanCorrect` adjusts stochastic model state on the basis of the given measurement of the model state

```
Kk=P'k?HT?(H?P'k?HT+R)-1
xk=x'k+Kk?(zk-H?x'k)
Pk=(I-Kk?H)?P'k
where
zk - given measurement (mesurement parameter)
Kk - Kalman "gain" matrix.
```

The function stores adjusted state at `kalman->state_post` and returns it on output.

**Example:** Using Kalman filter to track a rotating point

```
#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**

ConDenstation state

```
typedef struct CvConDensation
{
    int MP;      //Dimension of measurement vector
    int DP;      // Dimension of state vector
    float* DynamMatr;        // Matrix of the linear Dynamics
    system
    float* State;            // Vector of State
    int SamplesNum;          // Number of the Samples
    float** flSamples;       // array of the Sample Vectors
    float** flNewSamples;    // temporary array of the Sample
    Vectors
    float* flConfidence;     // Confidence for each Sample
    float* flCumulative;     // Cumulative confidence
    float* Temp;             // Temporary vector
    float* RandomSample;     // RandomVector to update sample set
    CvRandState* RandS;      // Array of structures to generate
    random vectors
} CvConDensation;
```

The structure `CvConDensation` stores CONditional DENSity propagATION tracker state. The information about the algorithm can be found at **‘http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/ISARD1/condensation.html‘_**

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

Allocates ConDensation filter structure

**Parameters**
- *dynam_params* – Dimension of the state vector.
- *measure_params* – Dimension of the measurement vector.
- *sample_count* – Number of samples.

The function `cvCreateConDensation` creates `CvConDensation` structure and returns pointer to the structure.

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

Deallocates ConDensation filter structure

**Parameter** *condens* – Pointer to the pointer to the structure to be released.

The function `cvReleaseConDensation` releases the structure `CvConDensation` (see `cvConDensation`) and frees all memory previously allocated for the structure.

void **cvConDensInitSampleSet**(*CvConDensation* condens, CvMat* lower_bound, CvMat* upper_bound*)
Initializes sample set for ConDensation algorithm

**Parameters**
- *condens* – Pointer to a structure to be initialized.
- *lower_bound* – Vector of the lower boundary for each dimension.
- *upper_bound* – Vector of the upper boundary for each dimension.

The function `cvConDensInitSampleSet` fills the samples arrays in the structure `CvConDensation` with values within specified ranges.

void **cvConDensUpdateByTime**(*CvConDensation* condens*)
Estimates subsequent model state

**Parameter** *condens* – Pointer to the structure to be updated.

The function `cvConDensUpdateByTime` estimates the subsequent stochastic model state from its current state.

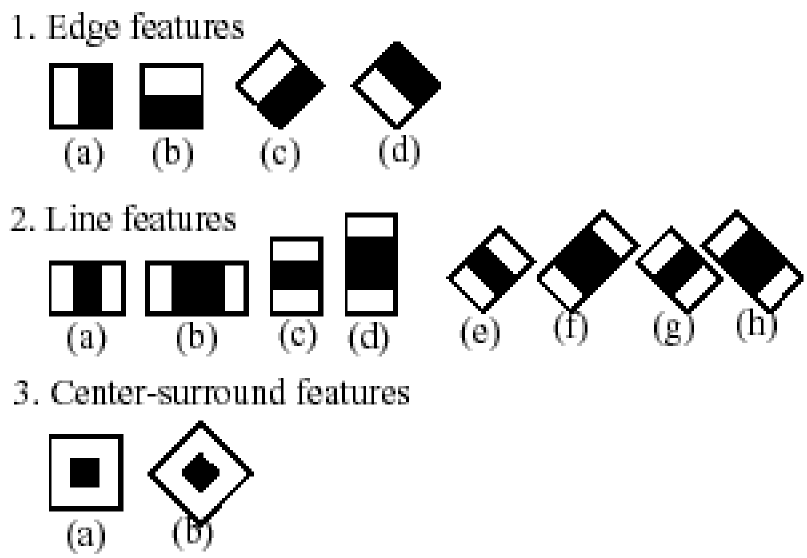
## 1.2.4 Pattern Recognition

### Object Detection

The object detector described below has been initially proposed by Paul Viola **'[Viola01]'_** and improved by Rainer Lienhart **'[Lienhart02]'_**. First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundreds of sample views of a particular object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.

After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a "1" if the region is likely to show the object (i.e., face/car), and "0" otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word "cascade" in the classifier name means that the resultant classifier consists of several simpler classifiers (`stages`) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word "boosted" means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different `boosting` techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifiers, and are calculated as described below. The current algorithm uses the following Haar-like features:




The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and ‘ cvIntegral‘_ description).

To see the object detector at work, have a look at HaarFaceDetect demo.

The following reference is for the detection part only. There is a separate application called `haartraining` that can train a cascade of boosted classifiers from a set of samples. See `opencv/apps/haartraining` for details.

**`CvHaarClassifierCascade`**

Boosted Haar classifier structures

```
#define CV_HAAR_FEATURE_MAX  3

/* 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 sub-rectangles of
   the window) and gives out
   a value depending on the response */
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 response is =
       threshold, left branch
                      is chosen, otherwise right branch
                      is chosen.
       alpha[i] - output value corresponding 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' responses
   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 straight 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
                                             cvSetImagesForHaarClassifierC
                                             ascade */
}
CvHaarClassifierCascade;
```

All the structures are used for representing a cascaded of boosted Haar classifiers. The cascade has the following hierarchical structure

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

The whole hierarchy can be constructed manually or loaded from a file using functions `cvLoadHaarClassifierCascade` or `cvLoad`.

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

Loads a trained cascade classifier from file or the classifier database embedded in OpenCV

**Parameters**
- *directory* – Name of directory containing the description of a trained cascade classifier.
- *orig_window_size* – Original size of objects the cascade has been trained on. Note that it is not stored in the cascade and therefore must be specified separately.

The function `cvLoadHaarClassifierCascade` loads a trained cascade of haar classifiers from a file or the classifier database embedded in OpenCV. The base can be trained using `haartraining` application (see opencv/apps/haartraining for details).

*The function is obsolete*. Nowadays object detection classifiers are stored in XML or YAML files, rather than in directories. To load cascade from a file, use `cvLoad` function.

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

Releases haar classifier cascade

**Parameter** *cascade* – Double pointer to the released cascade. The pointer is cleared by the function.

The function `cvReleaseHaarClassifierCascade` deallocates the cascade that has been created manually or loaded using `cvLoadHaarClassifierCascade` or `cvLoad`.

`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)*)

Detects objects in the image

```
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;
```

**Parameters**
- *image* – Image to detect objects in.
- *cascade* – Haar classifier cascade in internal representation.
- *storage* – Memory storage to store the resultant sequence of the object candidate rectangles.
- *scale_factor* – The factor by which the search window is scaled between the subsequent scans, for example, 1.1 means increasing window by 10%.
- *min_neighbors* – Minimum number (minus 1) of neighbor rectangles that makes up an object. All the groups of a smaller number of rectangles than `min_neighbors`-1 are rejected. If `min_neighbors` is 0, the function does not any grouping at all and returns all the detected candidate rectangles, which may be useful if the user wants to apply a customized grouping procedure.
- *flags* – Mode of operation. It can be a combination of zero or more of the following values:
  - `CV_HAAR_SCALE_IMAGE`- for each scale factor used the function will downscale the image rather than "zoom" the feature coordinates in the classifier cascade. Currently, the option can only be used alone, i.e. the flag can not be set together with the others.
  - `CV_HAAR_DO_CANNY_PRUNING`- If it is set, the function uses Canny edge detector to reject some image regions that contain too few or too much edges and thus can not contain the searched object. The particular threshold values are tuned for face detection and in this case the pruning speeds up the processing.
  - `CV_HAAR_FIND_BIGGEST_OBJECT`- If it is set, the function finds the largest object (if any) in the image. That is, the output sequence will contain one (or zero) element(s).
  - `CV_HAAR_DO_ROUGH_SEARCH`- It should be used only when
  - `CV_HAAR_FIND_BIGGEST_OBJECT` is set and `min_neighbors` > 0. If the flag is set, the function does not look for candidates of a smaller size as soon as it has found the object (with enough neighbor candidates) at the current scale. Typically, when `min_neighbors` is fixed, the mode yields less accurate (a bit larger) object rectangle than the regular single-object mode `(flags``=:const:`CV_HAAR_FIND_BIGGEST_OBJECT`), but it is much faster, up to an order of magnitude. A greater value of ``min_neighbors` may be specified to improve the accuracy.

  Note, that in single-object mode `CV_HAAR_DO_CANNY_PRUNING` does not improve performance much and can even slow down the processing.
- *min_size* – Minimum window size. By default, it is set to the size of samples the classifier has been trained on (~20?20 for face detection).

The function `cvHaarDetectObjects` finds rectangular regions in the given image that are likely to contain objects the cascade has been trained for and returns those regions as a sequence of rectangles. The function scans the image several times at different scales (see ' cvSetImagesForHaarClassifierCascade'_). Each time it considers overlapping regions in the image and applies the classifiers to the regions using `cvRunHaarClassifierCascade`. It may also apply some heuristics to reduce number of analyzed regions, such as Canny pruning. After it has proceeded and collected the candidate rectangles (regions that passed the classifier cascade), it groups them and returns a sequence of average rectangles for each large enough group. The default parameters (`scale_factor=1.1`, `min_neighbors=3`, `flags=0`) are tuned for accurate yet slow object detection. For a faster operation on real video images the more preferable settings are: scale_factor=1.2, min_neighbors=2, flags=CV_HAAR_DO_CANNY_PRUNING, min_size=<minimum possible face size> (for example, ~1/4 to 1/16 of the image area in case of video conferencing).

**Example:** Using cascade of Haar classifiers to find objects (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 input
    image 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 rectangles 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;
}
```

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

Assigns images to the hidden cascade

**Parameters**

- *cascade* – Hidden Haar classifier cascade, created by ‘ cvCreateHidHaarClassifierCascade‘_.
- *sum* – Integral (sum) single-channel image of 32-bit integer format. This image as well as the two subsequent images are used for fast feature evaluation and brightness/contrast normalization. They all can be retrieved from input 8-bit or floating point single- channel image using The function `cvIntegral`.
- *sqsum* – Square sum single-channel image of 64-bit floating- point format.
- *tilted_sum* – Tilted sum single-channel image of 32-bit integer format.
- *scale* – Window scale for the cascade. If `scale=1`, original window size is used (objects of that size are searched) - the same size as specified in `cvLoadHaarClassifierCascade` (24x24 in case of “<default_face_cascade>”), if `scale=2`, a two times larger window is used (48x48 in case of default face cascade). While this will speed-up search about four times, faces smaller than 48x48 cannot be detected.

The function `cvSetImagesForHaarClassifierCascade` assigns images and/or window scale to the hidden classifier cascade. If image pointers are NULL, the previously set images are used further (i.e. NULLs mean “do not change images”). Scale parameter has no such a “protection” value, but the previous value can be retrieved by ‘ cvGetHaarClassifierCascadeScale‘_ function and reused again. The function is used to prepare cascade for detecting object of the particular size in the particular image. The function is called internally by ‘ cvHaarDetectObjects‘_, but it can be called by user if there is a need in using lower-level function `cvRunHaarClassifierCascade`.

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

Runs cascade of boosted classifier at given image location

**Parameters**

- *cascade* – Haar classifier cascade.
- *pt* – Top-left corner of the analyzed region. Size of the region is a original window size scaled by the currently set scale. The current window size may be retrieved using ‘ cvGetHaarClassifierCascadeWindowSize‘_ function.
- *start_stage* – Initial zero-based index of the cascade stage to start from. The function assumes that all the previous stages are passed. This feature is used internally by ‘ cvHaarDetectObjects‘_ for better processor cache utilization.

The function `cvRunHaarHaarClassifierCascade` runs Haar classifier cascade at a single image location. Before using this function the integral images and the appropriate scale (=> window size) should be set using `cvSetImagesForHaarClassifierCascade`. The function returns positive value if the analyzed rectangle passed all the classifier stages (it is a candidate) and zero or negative value otherwise.

### 1.2.5 Camera Calibration and 3D Reconstruction

#### Pinhole Camera Model, Distortion

The functions in this section use so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using perspective transformation.

```
s*m' = A*[R|t]*M', or

 [u]    [fx 0 cx] [r11 r12 r13 t1] [X]
s[v] = [0 fy cy]*[r21 r22 r23 t2]*[Y]
 [1]    [0  0  1] [r31 r32 r33 t2] [Z]
                                   [1]
```

Where `(X, Y, Z)` are coordinates of a 3D point in the world coordinate space, `(u, v)` are coordinates of point projection in pixels. `A` is called a camera matrix, or matrix of intrinsic parameters. `(cx, cy)` is a principal point (that is usually at the image center), and `fx, fy` are focal lengths expressed in pixel-related units. Thus, if an image from camera is up-sampled/down-sampled by some factor, all these parameters (`fx, fy, cx` and `cy`) should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re- used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix `[R|t]` is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, `[R|t]` translates coordinates of a point `(X, Y, Z)` to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when z?0):

```
[x]      [X]
[y] = R*[Y] + t
[z]      [Z]

x' = x/z
y' = y/z

u = fx*x' + cx
v = fy*y' + cy
```

Real lens usually have some distortion, which is mainly a radial distortion and slight tangential distortion. So, the above model is extended as:

```
[x]      [X]
[y] = R*[Y] + t
[z]      [Z]

x' = x/z
y' = y/z

x" = x'*(1 + k1r2 + k2r4 + k3r6) + 2*p1x'*y' + p2(r2+2*x'2)
y" = y'*(1 + k1r2 + k2r4 + k3r6) + p1(r2+2*y'2) + 2*p2*x'*y'
```

where

```
r2 = x'2+y'2
u = fx*x" + cx
v = fy*y" + cy
```

k1, k2, k3 are radial distortion coefficients, p1, p2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. The distortion coefficients also do not depend on the scene viewed, thus they are intrinsic camera parameters. **And they remain the same regardless of the captured image resolution.** That is, if, for example, a camera has been calibrated on images of 320x240 resolution, absolutely the same distortion coefficients can be used for images of 640x480 resolution from the same camera (while fx, fy, cx and cy need to be scaled appropriately).

Another note.Many of calibration related functions take the vector of distortion coefficients. It can be 4x1, 1x4, 5x1 or 1x5 floating-point vector (CvMat*). The ordering of the distortion coefficients is the following

```
(k1, k2, p1, p2[, k3]).
```

That is, the first 2 radial distortion coefficients are followed by 2 tangential distortion coefficients and then, optionally, by the third radial distortion coefficients. Such ordering is used to keep backward compatibility with previous versions of OpenCV.

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspondences).

The functionality described in this section is largely based on the camera calibration toolbox **'[Bouguet04]'_**.

### Single and Stereo Camera Calibration

void **cvProjectPoints2**(*const CvMat* object_points, const CvMat* rotation_vector, const CvMat* translation_vector, const CvMat* intrinsic_matrix, const CvMat* distortion_coeffs, CvMat* image_points, CvMat* dpdrot=NULL, CvMat* dpdt=NULL, CvMat* dpdf=NULL, CvMat* dpdc=NULL, CvMat* dpddist=NULL, double aspect_ratio=0*)

Projects 3D points to image plane

**Parameters**
- *object_points* – The array of object points, 3xN or Nx3, where N is the number of points in the view.
- *rotation_vector* – The rotation vector, 1x3 or 3x1.
- *translation_vector* – The translation vector, 1x3 or 3x1.
- *intrinsic_matrix* – The camera matrix (A) [fx 0 cx; 0 fy cy; 0 0 1].
- *distortion_coeffs* – The vector of distortion coefficients, **'4x1, 1x4, 5x1 or 1x5'_**. If the vector is NULL, the function assumes that all the distortion coefficients are 0's.
- *image_points* – The output array of image points, 2xN or Nx2, where N is the total number of points in the view.
- *dpdrot* – Optional Nx3 matrix of derivatives of image points with respect to components of the rotation vector.
- *dpdt* – Optional Nx3 matrix of derivatives of image points w.r.t. components of the translation vector.
- *dpdf* – Optional Nx2 matrix of derivatives of image points w.r.t. fx and fy.
- *dpdc* – Optional Nx2 matrix of derivatives of image points w.r.t. cx and cy.

- *dpddist* – Optional Nx4 matrix of derivatives of image points w.r.t. distortion coefficients.
- *aspect_ratio* – Optional aspect ratio parameter used to correct the output `dpdf`. (When `cvCalibrateCamera2` or `cvStereoCalibrate` are called with the flag `CV_CALIB_FIX_ASPECT_RATIO`, only `fy` is estimated as independent parameter, and `fx` is computed as `fy*apect_ratio`; this affects `dpdf` too). If the parameter is 0, it means that the aspect ratio is not fixed.

The function `cvProjectPoints2` computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes Jacobians - matrices of partial derivatives of image points as functions of all the input parameters w.r.t. the particular parameters, intrinsic and/or extrinsic. The Jacobians are used during the global optimization in `cvCalibrateCamera2` and `cvFindExtrinsicCameraParams2`. The function itself is also used to compute reprojection error for with current intrinsic and extrinsic parameters.

Note, that with intrinsic and/or extrinsic parameters set to special values, the function can be used to compute just extrinsic transformation or just intrinsic transformation (i.e. distortion of a sparse set of points).

void **cvFindHomography**(*const CvMat* src_points, const CvMat* dst_points, CvMat* homography, int method=0, double ransacReprojThreshold=0, CvMat* mask=NULL*)

Finds perspective transformation between two planes

**Parameters**
- *src_points* – Point coordinates in the original plane, 2xN, Nx2, 3xN or Nx3 array (the latter two are for representation in homogeneous coordinates), where N is the number of points.
- *dst_points* – Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 array (the latter two are for representation in homogeneous coordinates)
- *homography* – Output 3x3 homography matrix.
- *method* – The method used to computed homography matrix. One of:
  - `0`- regular method using all the point pairs
  - `CV_RANSAC`- RANSAC-based robust method
  - `CV_LMEDS`- Least-Median robust method
- *ransacReprojThreshold* – The maximum allowed reprojection error to treat a point pair as an inlier. The parameter is only used in RANSAC-based homography estimation. E.g. if `dst_points` coordinates are measured in pixels with pixel-accurate precision, it makes sense to set this parameter somewhere in the range ~1..3.
- *mask* – The optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`).

The function `cvFindHomography` finds perspective transformation `H=||hij||` between the source and the destination planes:

```
  [x'i]    [xi]
si[y'i]~H*[yi]
  [1  ]    [ 1]
```

So that the reprojection error is minimized:

```
sum_i((x'i-(h11*xi + h12*yi + h13)/(h31*xi + h32*yi + h33))2+
      (y'i-(h21*xi + h22*yi + h23)/(h31*xi + h32*yi +
      h33))2) -> min
```

If the parameter `method` is set to the default value 0, the function uses all the point pairs and estimates the best suitable homography matrix. However, if there can not all the points pairs (`src_points` i, `dst_points` i) fit the rigid perspective transformation (i.e. there can be outliers), it is still possible to estimate the correct transformation using one of the robust methods available. Both methods, `CV_RANSAC` and `CV_LMEDS`, try many different random subsets of the corresponding point pairs (of 5 pairs each), estimate homography matrix using this subset using simple least-square algorithm and then compute quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median reprojection error for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in case of a robust method) with Levenberg-Marquardt method in order to reduce the reprojection error even more.

The method `CV_RANSAC` can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method `CV_LMEDS` does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features and there can be only some small noise, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized to make h33=1.

`void` **cvCalibrateCamera2**(*const CvMat* object_points, const CvMat* image_points, const CvMat* point_counts, CvSize image_size, CvMat* intrinsic_matrix, CvMat* distortion_coeffs, CvMat* rotation_vectors=NULL, CvMat* translation_vectors=NULL, int flags=0*)

Finds intrinsic and extrinsic camera parameters using calibration pattern

**Parameters**

- *object_points* – The joint matrix of object points, 3xN or Nx3, where N is the total number of points in all views.
- *image_points* – The joint matrix of corresponding image points, 2xN or Nx2, where N is the total number of points in all views.
- *point_counts* – Vector containing numbers of points in each particular view, 1xM or Mx1, where M is the number of a scene views.
- *image_size* – Size of the image, used only to initialize intrinsic camera matrix.
- *intrinsic_matrix* – The output camera matrix (A) [fx 0 cx; 0 fy cy; 0 0 1]. If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized.
- *distortion_coeffs* – The output vector of distortion coefficients, **‘4x1, 1x4, 5x1 or 1x5‘_**.
- *rotation_vectors* – The output 3xM or Mx3 array of rotation vectors (compact representation of rotation matrices, see `cvRodrigues2`).
- *translation_vectors* – The output 3xM or Mx3 array of translation vectors.
- *flags* – Different flags, may be 0 or combination of the following values:
  - `CV_CALIB_USE_INTRINSIC_GUESS`- `intrinsic_matrix` contains valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`image_size` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function. Use `cvFindExtrinsicCameraParams2` instead.
  - `CV_CALIB_FIX_PRINCIPAL_POINT`- The principal point is not changed during the global optimization, it stays at the center and at the other location specified (when `CV_CALIB_FIX_FOCAL_LENGTH` - Both fx and fy are fixed.
  - `CV_CALIB_USE_INTRINSIC_GUESS` is set as well).
  - `CV_CALIB_FIX_ASPECT_RATIO`- The optimization procedure consider only one of `fx` and `fy` as independent variable and keeps the aspect ratio `fx/fy` the same as it was set initially in `intrinsic_matrix`. In this case the actual initial values of `(fx, fy)` are either taken from the matrix (when `CV_CALIB_USE_INTRINSIC_GUESS` is set) or estimated somehow (in the latter case `fx, fy` may be set to arbitrary values, only their ratio is used).
  - `CV_CALIB_ZERO_TANGENT_DIST`- Tangential distortion coefficients are set to zeros and do not change during the optimization.
  - `CV_CALIB_FIX_K1`- The 0-th distortion coefficient (k1) is fixed (to 0 or to the initial passed value if `CV_CALIB_USE_INTRINSIC_GUESS` is passed)

- CV_CALIB_FIX_K2- The 1-st distortion coefficient (k2) is fixed (see above)
- CV_CALIB_FIX_K3- The 4-th distortion coefficient (k3) is fixed (see above)

The function cvCalibrateCamera2 estimates intrinsic camera parameters and, optionally, the extrinsic parameters for each view of the calibration pattern. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chess board as a calibration rig (see cvFindChessboardCorners). Currently, initialization of intrinsic parameters (when CV_CALIB_USE_INTRINSIC_GUESS is not set) is only implemented for planar calibration rigs (z-coordinates of object points must be all 0's). 3D rigs can still be used as long as the initial intrinsic_matrix is provided. After the initial values of intrinsic and extrinsic parameters are obtained by the function, they are optimized further to minimize the total reprojection error - the sum of squared differences between the actual coordinates of image points and the ones computed using cvProjectPoints2 with current intrinsic and extrinsic parameters.

void **cvCalibrationMatrixValues**(*const CvMat* calibMatr, int imgWidth, int imgHeight, double apertureWidth=0, double apertureHeight=0, double* fovx=NULL, double* fovy=NULL, double* focalLength=NULL, CvPoint2D64f* principalPoint=NULL, double* pixelAspectRatio=NULL*)

Finds intrinsic and extrinsic camera parameters using calibration pattern

**Parameters**
- *calibMatr* – The matrix of intrinsic parameters, e.g. computed by cvCalibrateCamera2
- *imgWidth* – Image width in pixels
- *imgHeight* – Image height in pixels
- *apertureWidth* – Aperture width in realworld units (optional input parameter)
- *apertureHeight* – Aperture width in realworld units (optional input parameter)
- *fovx* – Field of view angle in x direction in degrees (optional output parameter)
- *fovx* – Field of view angle in y direction in degrees (optional output parameter)
- *focalLength* – Focal length in realworld units (optional output parameter)
- *principalPoint* – The principal point in realworld units (optional output parameter)
- *pixelAspectRatio* – The pixel aspect ratio ~ fy/fx (optional output parameter)

The function cvCalibrationMatrixValues computes various useful camera (sensor/lens) characteristics using the computed camera calibration matrix, image frame resolution in pixels and the physical aperture size.

void **cvFindExtrinsicCameraParams2**(*const CvMat* object_points, const CvMat* image_points, const CvMat* intrinsic_matrix, const CvMat* distortion_coeffs, CvMat* rotation_vector, CvMat* translation_vector*)

Finds extrinsic camera parameters for particular view

**Parameters**
- *object_points* – The array of object points, 3xN or Nx3, where N is the number of points in the view.
- *image_points* – The array of corresponding image points, 2xN or Nx2, where N is the number of points in the view.
- *intrinsic_matrix* – The camera matrix (A) [fx 0 cx; 0 fy cy; 0 0 1].
- *distortion_coeffs* – The vector of distortion coefficients, **'4x1, 1x4, 5x1 or 1x5'_**. If it is NULL, the function assumes that all the distortion coefficients are 0's.
- *rotation_vector* – The output 3x1 or 1x3 rotation vector (compact representation of a rotation matrix, see cvRodrigues2).
- *translation_vector* – The output 3x1 or 1x3 translation vector.

The function cvFindExtrinsicCameraParams2 estimates the object pose using the intrinsic camera parameters and a few (>=4) 2D<->3D point correspondences.

void **cvStereoCalibrate** (*const CvMat* object_points, const CvMat* image_points1, const CvMat* image_points2, const CvMat* point_counts, CvMat* camera_matrix1, CvMat* dist_coeffs1, CvMat* camera_matrix2, CvMat* dist_coeffs2, CvSize image_size, CvMat* R, CvMat* T, CvMat* E=0, CvMat* F=0, CvTermCriteria term_crit=cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 30, 1e-6), int flags=CV_CALIB_FIX_INTRINSIC*)

Calibrates stereo camera

**Parameters**

- *object_points* – The joint matrix of object points, 3xN or Nx3, where N is the total number of points in all views.
- *image_points1* – The joint matrix of corresponding image points in the views from the 1st camera, 2xN or Nx2, where N is the total number of points in all views.
- *image_points2* – The joint matrix of corresponding image points in the views from the 2nd camera, 2xN or Nx2, where N is the total number of points in all views.
- *point_counts* – Vector containing numbers of points in each view, 1xM or Mx1, where M is the number of views.
- *camera_matrix1, camera_matrix2* – The input/output camera matrices [fxk 0 cxk; 0 fyk cyk; 0 0 1]. If `CV_CALIB_USE_INTRINSIC_GUESS` or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of the elements of the matrices must be initialized.
- *dist_coeffs1, dist_coeffs2* – The input/output vectors of distortion coefficients for each camera, **'4x1, 1x4, 5x1 or 1x5'_**.
- *image_size* – Size of the image, used only to initialize intrinsic camera matrix.
- *R* – The rotation matrix between the 1st and the 2nd cameras' coordinate systems
- *T* – The translation vector between the cameras' coordinate systems.
- *E* – The optional output essential matrix
- *F* – The optional output fundamental matrix
- *term_crit* – Termination criteria for the iterative optimiziation algorithm.
- *flags* – Different flags, may be 0 or combination of the following values:
  - `CV_CALIB_FIX_INTRINSIC`- If it is set, `camera_matrix1,2`, as well as `dist_coeffs1,2` are fixed, so that only extrinsic parameters are optimized.
  - `CV_CALIB_USE_INTRINSIC_GUESS`- The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user
  - `CV_CALIB_FIX_PRINCIPAL_POINT`- The principal points are fixed during the optimization.
  - `CV_CALIB_FIX_FOCAL_LENGTH`- fxk and fyk are fixed
  - `CV_CALIB_FIX_ASPECT_RATIO`- fyk is optimized, but the ratio fxk/fyk is fixed.
  - `CV_CALIB_SAME_FOCAL_LENGTH`- Enforces fx0=fx1 and fy0=fy1.
  - **`CV_CALIB_ZERO_TANGENT_DIST` - Tangential distortion coefficients for** each camera are set to zeros and fixed there.
  - `CV_CALIB_FIX_K1`- The 0-th distortion coefficients (k1) are fixed
  - `CV_CALIB_FIX_K2`- The 1-st distortion coefficients (k2) are fixed
  - `CV_CALIB_FIX_K3`- The 4-th distortion coefficients (k3) are fixed

The function `cvStereoCalibrate` estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientatation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with `cvFindExtrinsicCameraParams2`), obviously, those poses will relate to each other, i.e. given (R1, T1) it should be possible to compute (R2, T2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that

```
   R2=R*R1
   T2=R*T1 + T,
    Optionally, it computes the essential matrix E: ::
       [0 -T2 T1]
   E = [T2 0 -T0]*R,
       [-T1 T0 0]
    where Ti are components of the translation vector T: T=[T0,
    T1, T2]T. And also the function can compute the fundamental matrix
    F: ::
   F = inv(camera_matrix2)T*E*inv(camera_matrix1),

Besides the stereo-related information, the function can
also perform full calibration of each of the 2 cameras. However,
because of the high dimensionality of the parameter space and noise
in the input data the function can diverge from the correct
solution. Thus, if intrinsic parameters can be estimated with high
accuracy for each of the cameras individually (e.g. using
:cfunc:`cvCalibrateCamera2`), it is recommended to do so and then pass
:const:`CV_CALIB_FIX_INTRINSIC` flag to the function along with the
computed intrinsic parameters. Otherwise, if all the parameters are
estimated at once, it makes sense to restrict some parameters, e.g.
pass :const:`CV_CALIB_SAME_FOCAL_LENGTH` and :const:`CV_CALIB_ZERO_TANGENT_DIST` flags, which ar
reasonable assumptions.
```

void **cvStereoRectify**(*const CvMat* camera_matrix1, const CvMat* camera_matrix2, const CvMat* dist_coeffs1, const CvMat* dist_coeffs2, CvSize image_size, const CvMat* R, const CvMat* T, CvMat* R1, CvMat* R2, CvMat* P1, CvMat* P2, CvMat* Q=0, int flags=CV_CALIB_ZERO_DISPARITY*)

Computes rectification transform for stereo camera

**Parameters**
- *camera_matrix1, camera_matrix2* – The camera matrices [fxk 0 cxk; 0 fyk cyk; 0 0 1].
- *dist_coeffs1, dist_coeffs2* – The vectors of distortion coefficients for each camera, **‘4x1, 1x4, 5x1 or 1x5‘_**.
- *image_size* – Size of the image used for stereo calibration.
- *R* – The rotation matrix between the 1st and the 2nd cameras’ coordinate systems
- *T* – The translation vector between the cameras’ coordinate systems.
- *R1, R2* – 3x3 Rectification transforms (rotation matrices) for the first and the second cameras, respectively
- *P1, P2* – 3x4 Projection matrices in the new (rectified) coordinate systems
- *Q* – The optional output disparity-to-depth mapping matrix, 4x4, see `cvReprojectImageTo3D`.
- *flags* – The operation flags; may be 0 or `CV_CALIB_ZERO_DISPARITY`. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function can shift one of the image in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximise the useful image area.

The function `cvStereoRectify` computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by `cvStereoCalibrate` and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The function is normally called after `cvStereoCalibrate` that computes both camera matrices, the distortion coefficients, `R` and `T`. The 2 cases are distinguished by the function:

1. horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

```
   [f 0 cx1 0]
P1=[0 f cy  0]
   [0 0  1  0]

   [f 0 cx2 Tx*f]
P2=[0 f cy   0 ],
   [0 0  1   0 ]
```

where Tx is horizontal shift between the cameras and cx1=cx2 if `CV_CALIB_ZERO_DISPARITY` is set.

2. vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

```
   [f 0 cx  0]
P1=[0 f cy1 0]
   [0 0  1  0]

   [f 0 cx   0  ]
P2=[0 f cy2 Ty*f],
   [0 0  1   0  ]
```

where Ty is vertical shift between the cameras and cy1=cy2 if `CV_CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of P1 and P2 will effectively be the new "rectified" camera matrices.

`void` **cvStereoRectifyUncalibrated**(*const CvMat* points1, const CvMat* points2, const CvMat* F, CvSize image_size, CvMat* H1, CvMat* H2, double threshold=5*)

Computes rectification transform for uncalibrated stereo camera

**Parameters**
- *points1, points2* – The 2 arrays of corresponding 2D points.
- *F* – Fundamental matrix. It can be computed using the same set of point pairs `points1` and `points2` using `cvFindFundamentalMat`
- *image_size* – Size of the image.
- *H1, H2* – The rectification homography matrices for the first and for the second images.
- *threshold* – Optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which `fabs(points2[i]T*F*points1[i])>threshold`) are rejected prior to computing the homographies.

The function `cvStereoRectifyUncalibrated` computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from `cvStereoRectify` is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the following algorithm **'[Hartley99]'_**.

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using `cvCalibrateCamera2` and then the images can be corrected using `cvUndistort2`.

`int` **cvRodrigues2**(*const CvMat* src, CvMat* dst, CvMat* jacobian=0*)

Converts rotation matrix to rotation vector or vice versa

**Parameters**
- *src* – The input rotation vector (3x1 or 1x3) or rotation matrix (3x3).
- *dst* – The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively.

- *jacobian* – Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components w.r.t the input array components.

The function cvRodrigues2 converts a rotation vector to rotation matrix or vice versa. Rotation vector is a compact representation of rotation matrix. Direction of the rotation vector is the rotation axis and the length of the vector is the rotation angle around the axis. The rotation matrix `R`, corresponding to the rotation vector `r`, is computed as following:

```
theta <- norm(r)
r <- r/theta
                                                    [0 -rz ry]
R = cos(theta)*I + (1-cos(theta))*rrT + sin(theta)*[rz 0 -rx]
                                                    [-ry rx 0]
```

Inverse transformation can also be done easily as

```
[0 -rz ry]
sin(theta)*[rz 0 -rx] = (R - RT)/2
[-ry rx 0]
```

Rotation vector is a convenient representation of a rotation matrix as a matrix with only 3 degrees of freedom. The representation is used in the global optimization procedures inside cvFindExtrinsicCameraParams2 and cvCalibrateCamera2.

void **cvUndistort2**(*const CvArr* src, CvArr* dst, const CvMat* intrinsic_matrix, const CvMat* distortion_coeffs*)

Transforms image to compensate lens distortion

**Parameters**
- *src* – The input (distorted) image.
- *dst* – The output (corrected) image.
- *intrinsic_matrix* – The camera matrix (A) [fx 0 cx; 0 fy cy; 0 0 1].
- *distortion_coeffs* – The vector of distortion coefficients, **'4x1, 1x4, 5x1 or 1x5'_**.

The function cvUndistort2 transforms the image to compensate radial and tangential lens distortion. The camera matrix and distortion parameters can be determined using cvCalibrateCamera2. For every pixel in the output image the function computes coordinates of the corresponding location in the input image using the formulae in the section beginning. Then, the pixel value is computed using bilinear interpolation. If the resolution of images is different from what was used at the calibration stage, `fx, fy, cx` and `cy` need to be adjusted appropriately, while the distortion coefficients remain the same.

In the undistorted image the principal point will be at the image center.

void **cvInitUndistortMap**(*const CvMat* camera_matrix, const CvMat* distortion_coeffs, CvArr* mapx, CvArr* mapy*)

Computes undistortion map

**Parameters**
- *camera_matrix* – The camera matrix (A) [fx 0 cx; 0 fy cy; 0 0 1].
- *distortion_coeffs* – The vector of distortion coefficients, **'4x1, 1x4, 5x1 or 1x5'_**.
- *mapx* – The output array of x-coordinates of the map.
- *mapy* – The output array of y-coordinates of the map.

The function cvInitUndistortMap pre-computes the undistortion map - coordinates of the corresponding pixel in the distorted image for every pixel in the corrected image. Then, the map (together with input and output images) can be passed to cvRemap function.

In the undistorted image the principal point will be at the image center.

void **cvInitUndistortRectifyMap**(*const CvMat* camera_matrix, const CvMat* dist_coeffs, const CvMat* R, const CvMat* new_camera_matrix, CvArr* mapx, CvArr* mapy*)

Computes undistortion+rectification transformation map a head of stereo camera

**Parameters**
- *camera_matrix* – The camera matrix A=[fx 0 cx; 0 fy cy; 0 0 1].
- *distortion_coeffs* – The vector of distortion coefficients, **‘4x1, 1x4, 5x1 or 1x5‘_**.
- *R* – The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by `cvStereoRectify` can be passed here. If the parameter is NULL, the identity matrix is used.
- *new_camera_matrix* – The new camera matrix A’=[fx’ 0 cx’; 0 fy’ cy’; 0 0 1].
- *mapx* – The output array of x-coordinates of the map.
- *mapy* – The output array of y-coordinates of the map.

The function `cvInitUndistortRectifyMap` is an extended version of `cvInitUndistortMap`. That is, in addition to the correction of lens distortion, the function can also apply arbitrary perspective transformation R and finally it can scale and shift the image according to the new camera matrix. That is, in pseudo code the transformation can be represented as

```
// (u,v) is the input point,
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// new_camera_matrix=[fx' 0 cx'; 0 fy' cy'; 0 0 1]
x = (u - cx')/fx'
y = (v - cy')/fy'
[X,Y,W]T = R-1*[x y 1]T
x' = X/W, y' = Y/W
x" = x'*(1 + k1r2 + k2r4 + k3r6) + 2*p1x'*y' + p2(r2+2*x'2)
y" = y'*(1 + k1r2 + k2r4 + k3r6) + p1(r2+2*y'2) + 2*p2*x'*y'
mapx(u,v) = x"*fx + cx
mapy(u,v) = y"*fy + cy
```

Note that the code above does the reverse transformation from the target image (i.e. the ideal one, after undistortion and rectification) to the original “raw” image straight from the camera. That’s for bilinear interpolation purposes and in order to fill the whole destination image w/o gaps using `cvRemap`.

Normally, this function is called [twice, once for each head of stereo camera] after `cvStereoRectify`. But it is also possible to compute the rectification transformations directly from the fundamental matrix, e.g. by using `cvStereoRectifyUncalibrated`. Such functions work with pixels and produce homographies as rectification transformations, not rotation matrices `R` in 3D space. In this case, the `R` can be computed from the homography matrix `H` as

```
R = inv(camera_matrix)*H*camera_matrix
```

void **cvUndistortPoints**(*const CvMat* src, CvMat* dst, const CvMat* camera_matrix, const CvMat* dist_coeffs, const CvMat* R=NULL, const CvMat* P=NULL*)

Computes the ideal point coordinates from the observed point coordinates

**Parameter** *src* – The observed point coordinates.

:param dst :param The ideal point coordinates, after undistortion and

reverse perspective transformation.

**Parameters**
- *camera_matrix* – The camera matrix A=[fx 0 cx; 0 fy cy; 0 0 1].
- *distortion_coeffs* – The vector of distortion coefficients, **‘4x1, 1x4, 5x1 or 1x5‘_**.
- *R* – The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by `cvStereoRectify` can be passed here. If the parameter is NULL, the identity matrix is used.
- *P* – The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2`, computed by `cvStereoRectify` can be passed here. If the parameter is NULL, the identity matrix is used.

The function cvUndistortPoints is similar to cvInitUndistortRectifyMap and is opposite to it at the same time. The functions are similar in that they both are used to correct lens distortion and to perform the optional perspective (rectification) transformation. They are opposite because the function cvInitUndistortRectifyMap does actually perform the reverse transformation in order to initialize the maps properly, while this function does the forward transformation. That is, in pseudo-code it can be expressed as

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used as for stereo cameras, as well as for individual cameras when R=NULL.

int **cvFindChessboardCorners**(*const void* image, CvSize pattern_size, CvPoint2D32f* corners, int* corner_count=NULL, int flags=CV_CALIB_CB_ADAPTIVE_THRESH*)

Finds positions of internal corners of the chessboard

**Parameters**
- *image* – Source chessboard view; it must be 8-bit grayscale or color image.
- *pattern_size* – The number of inner corners per chessboard row and column.
- *corners* – The output array of corners detected.
- *corner_count* – The output corner counter. If it is not NULL, the function stores there the number of corners found.
- *flags* – Various operation flags, can be 0 or a combination of the following values:
  - CV_CALIB_CB_ADAPTIVE_THRESH- use adaptive thresholding to convert the image to black-n-white, rather than a fixed threshold level (computed from the average image brightness).
  - CV_CALIB_CB_NORMALIZE_IMAGE- normalize the image using cvNormalizeHist before applying fixed or adaptive thresholding.
  - CV_CALIB_CB_FILTER_QUADS- use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

The function cvFindChessboardCorners attempts to determine whether the input image is a view of the chessboard pattern and locate internal chessboard corners. The function returns non-zero value if all the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cvFindCornerSubPix.

void **cvDrawChessboardCorners**(*CvArr* image, CvSize pattern_size, CvPoint2D32f* corners, int count, int pattern_was_found*)

Renders the detected chessboard corners

**Parameters**
- *image* – The destination image; it must be 8-bit color image.
- *pattern_size* – The number of inner corners per chessboard row and column.
- *corners* – The array of corners detected.

- *count* – The number of corners.
- *pattern_was_found* – Indicates whether the complete board was found (?0) or not (=0). One may just pass the return value `cvFindChessboardCorners` here.

The function `cvDrawChessboardCorners` draws the individual chessboard corners detected (as red circles) in case if the board was not found (`pattern_was_found=0`) or the colored corners connected with lines when the board was found (`pattern_was_found`?0).

### Pose Estimation

CvPOSITObject* **cvCreatePOSITObject** (*CvPoint3D32f* points, int point_count*)
Initializes structure containing object information

**Parameters**
- *points* – Pointer to the points of the 3D object model.
- *point_count* – Number of object points.

The function `cvCreatePOSITObject` allocates memory for the object structure and computes the object inverse matrix.

The pre-processed object data is stored in the structure `CvPOSITObject`, internal for OpenCV, which means that the user cannot directly access the structure data. The user may only create this structure and pass its pointer to the function.

Object is defined as a set of points given in a coordinate system. The function `cvPOSIT` computes a vector that begins at a camera-related coordinate system center and ends at the `points[0]` of the object.

Once the work with a given object is finished, the function `cvReleasePOSITObject` must be called to free memory.

void **cvPOSIT** (*CvPOSITObject* posit_object, CvPoint2D32f* image_points, double focal_length, CvTermCriteria criteria, CvMatr32f rotation_matrix, CvVect32f translation_vector*)
Implements POSIT algorithm

**Parameters**
- *posit_object* – Pointer to the object structure.
- *image_points* – Pointer to the object points projections on the 2D image plane.
- *focal_length* – Focal length of the camera used.
- *criteria* – Termination criteria of the iterative POSIT algorithm.
- *rotation_matrix* – Matrix of rotations.
- *translation_vector* – Translation vector.

The function `cvPOSIT` implements POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using camera calibration functions. At every iteration of the algorithm new perspective projection of estimated pose is computed.

Difference norm between two projections is the maximal distance between corresponding points. The parameter `criteria.epsilon` serves to stop the algorithm if the difference is small.

void **cvReleasePOSITObject** (*CvPOSITObject** posit_object*)
Deallocates 3D object structure

**Parameter** *posit_object* – Double pointer to `CvPOSIT` structure.

The function `cvReleasePOSITObject` releases memory previously allocated by the function `cvCreatePOSITObject`.

void **cvCalcImageHomography** (*float* line, CvPoint3D32f* center, float* intrinsic, float* homography*)
Calculates homography matrix for oblong planar object (e.g. arm)

**Parameters**
- *line* – the main object axis direction (vector (dx,dy,dz)).

- *center* – object center ((cx,cy,cz)).
- *intrinsic* – intrinsic camera parameters (3x3 matrix).
- *homography* – output homography matrix (3x3).

The function `cvCalcImageHomography` calculates the homography matrix for the initial image transformation from image plane to the plane, defined by 3D oblong object line (See *Figure 6-10* in OpenCV Guide 3D Reconstruction Chapter).

### Epipolar Geometry, Stereo Correspondence

`int` **`cvFindFundamentalMat`** (*const CvMat* points1, const CvMat* points2, CvMat* fundamental_matrix, int method=CV_FM_RANSAC, double param1=3., double param2=0.99, CvMat* status=NULL*)

Calculates fundamental matrix from corresponding points in two images

**Parameters**
- *points1* – Array of the first image points of `2xN, Nx2, 3xN` or `Nx3` size (where `N` is number of points). Multi- channel `1xN` or `Nx1` array is also acceptable. The point coordinates should be floating-point (single or double precision)
- *points2* – Array of the second image points of the same size and format as `points1`

**:param fundamental_matrix** [The output fundamental matrix or] matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices).

**Parameters**
- *method* – Method for computing the fundamental matrix CV_FM_7POINT - for 7-point algorithm. N == 7 CV_FM_8POINT - for 8-point algorithm. N >= 8 CV_FM_RANSAC - for RANSAC algorithm. N > 8 CV_FM_LMEDS - for LMedS algorithm. N > 8
- *param1* – The parameter is used for RANSAC method only. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. Usually it is set somewhere from 1 to 3.

**:param param2** [The parameter is used for RANSAC or LMedS methods] only. It denotes the desirable level of confidence of the fundamental matrix estimate.

**Parameter** *status* – The optional output array of N elements, every element of which is set to 0 for outliers and to 1 for the "inliers", i.e. points that comply well with the estimated epipolar geometry. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1?s.

The epipolar geometry is described by the following equation

```
p2T*F*p1=0,
```

where `F` is fundamental matrix, `p1` and `p2` are corresponding points in the first and the second images, respectively.

The function `cvFindFundamentalMat` calculates fundamental matrix using one of four methods listed above and returns the number of fundamental matrices found (1 or 3) and 0, if no matrix is found.

The calculated fundamental matrix may be passed further to `cvComputeCorrespondEpilines` that finds epipolar lines corresponding to the specified points.

**Example:** Estimation of fundamental matrix using RANSAC algorithm

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

points1 = cvCreateMat(1,point_count,CV_32FC2);
points2 = cvCreateMat(1,point_count,CV_32FC2);
status = cvCreateMat(1,point_count,CV_8UC1);

/* Fill the points here ... */
for( i = 0; i < point_count; i++ )
{
    points1->data.db[i*2] = <x1,i>;
    points1->data.db[i*2+1] = <y1,i>;
    points2->data.db[i*2] = <x2,i>;
    points2->data.db[i*2+1] = <y2,i>;
}

fundamental_matrix = cvCreateMat(3,3,CV_32FC1);
int fm_count = cvFindFundamentalMat( points1,points2,fundamental_matrix, CV_FM_RANSAC,3,0.99,sta
```

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

For points in one image of stereo pair computes the corresponding epilines in the other image

**Parameters**
- *points* – The input points. `2xN, Nx2, 3xN` or `Nx3` array (where `N` number of points). Multi-channel `1xN` or `Nx1` array is also acceptable.
- *which_image* – Index of the image (1 or 2) that contains the `points`

:param fundamental_matrix : Fundamental matrix :param correspondent_lines: Computed epilines, `3xN` or `Nx3`

array

For every point in one of the two images of stereo-pair the function `cvComputeCorrespondEpilines` finds equation of a line that contains the corresponding point (i.e. projection of the same 3D point) in the other image. Each line is encoded by a vector of 3 elements `l=[a,b,c]T`, so that:

```
lT*[x, y, 1]T=0, or
a*x + b*y + c = 0
```

From the fundamental matrix definition (see `cvFindFundamentalMatrix` discussion), line `l2` for a point `p1` in the first image (`which_image=1`) can be computed as

```
l2=F*p1
```

and the line `l1` for a point `p2` in the second image (`which_image=1`) can be computed as

```
l1=FT*p2
```

Line coefficients are defined up to a scale. They are normalized (`a2+b2=1`) are stored into `correspondent_lines`.

void **cvConvertPointsHomogeneous**(*const CvMat* src, CvMat* dst*)

Convert points to/from homogeneous coordinates

**Parameters**
- *src* – The input point array, `2xN, Nx2, 3xN, Nx3, 4xN or Nx4` (where `N` is the number of points). Multi-channel `1xN` or `Nx1` array is also acceptable.

- *dst* – The output point array, must contain the same number of points as the input; The dimensionality must be the same, 1 less or 1 more than the input, and also within 2..4.

The function `cvConvertPointsHomogeneous` converts 2D or 3D points from/to homogeneous coordinates, or simply copies or transposes the array. In case if the input array dimensionality is larger than the output, each point coordinates are divided by the last coordinate:

```
(x,y[,z],w) -> (x',y'[,z']):

x' = x/w
y' = y/w
z' = z/w (if output is 3D)
```

If the output array dimensionality is larger, an extra 1 is appended to each point.

```
(x,y[,z]) -> (x,y[,z],1)
```

Otherwise, the input array is simply copied (with optional transposition) to the output. **Note** that, because the function accepts a large variety of array layouts, it may report an error when input/output array dimensionality is ambiguous. It is always safe to use the function with number of points `N`>=5, or to use multi-channel `Nx1` or `1xN` arrays.

**CvStereoBMState**

The structure for block matching stereo correspondence algorithm

```
typedef struct CvStereoBMState
{
    //pre filters (normalize input images):
    int       preFilterType; // 0 for now
    int       preFilterSize; // ~5x5..21x21
    int       preFilterCap;  // up to ~31
    //correspondence using Sum of Absolute Difference (SAD):
    int       SADWindowSize; // Could be 5x5..21x21
    int       minDisparity;  // minimum disparity (=0)
    int       numberOfDisparities; // maximum disparity - minimum
    disparity
    //post filters (knock out bad matches):
    int       textureThreshold; // areas with no texture are ignored
    float     uniquenessRatio;// filter out pixels if there are other
    close matches
                              // with different
                              disparity
    int       speckleWindowSize;// Disparity variation window (not used)
    int       speckleRange; // Acceptable range of variation in window
    (not used)
    // internal buffers, do not modify (!)
    CvMat* preFilteredImg0;
    CvMat* preFilteredImg1;
    CvMat* slidingSumBuf;
}
CvStereoBMState;
```

The block matching stereo correspondence algorithm, by Kurt Konolige, is very fast one-pass stereo matching algorithm that uses sliding sums of absolute differences between pixels in the left image and the pixels in the right image, shifted by some varying amount of pixels (from `minDisparity` to `minDisparity+numberOfDisparities`). On a pair of images WxH the algorithm computes disparity in `O(W*H*numberOfDisparities)` time. In order to improve quality and reability of the disparity map, the algorithm includes pre-filtering and post-filtering procedures.

Note that the algorithm searches for the corresponding blocks in x direction only. It means that the supplied stereo pair should be rectified. Vertical stereo layout is not directly supported, but in such a case the images could be transposed by user.

CvStereoBMState* **cvCreateStereoBMState**(*int preset=CV_STEREO_BM_BASIC, int numberOfDisparities=0*)

Creates block matching stereo correspondence structure

```
#define CV_STEREO_BM_BASIC 0
#define CV_STEREO_BM_FISH_EYE 1
#define CV_STEREO_BM_NARROW 2
```

**Parameters**
- *preset* – ID of one of the pre-defined parameter sets. Any of the parameters can be overridden after creating the structure.
- *numberOfDisparities* – The number of disparities. If the parameter is 0, it is taken from the preset, otherwise the supplied value overrides the one from preset.

The function cvCreateStereoBMState creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cvFindStereoCorrespondenceBM.

void **cvReleaseStereoBMState**(*CvStereoBMState** state*)

Releases block matching stereo correspondence structure

**Parameter** *state* – Double pointer to the released structure

The function cvReleaseStereoBMState releases the stereo correspondence structure and all the associated internal buffers.

void **cvFindStereoCorrespondenceBM**(*const CvArr* left, const CvArr* right, CvArr* disparity, CvStereoBMState* state*)

Computes the disparity map using block matching algorithm

**Parameters**
- *left* – The left single-channel, 8-bit image.
- *right* – The right image of the same size and the same type.
- *disparity* – The output single-channel 16-bit signed disparity map of the same size as input images. Its elements will be the computed disparities, multiplied by 16 and rounded to integer's.
- *state* – Stereo correspondence structure.

The function cvFindStereoCorrespondenceBM computes disparity map for the input rectified stereo pair.

**CvStereoGCState**

The structure for graph cuts-based stereo correspondence algorithm

```
typedef struct CvStereoGCState
{
    int Ithreshold; // threshold for piece-wise linear data cost function
    (5 by default)
    int interactionRadius; // radius for smoothness cost function (1 by
    default; means Potts model)
    float K, lambda, lambda1, lambda2; // parameters for the cost
    function
 // (usually computed adaptively from
                                          the input data)
    int occlusionCost; // 10000 by default
    int minDisparity; // 0 by default; see CvStereoBMState
    int numberOfDisparities; // defined by user; see CvStereoBMState
    int maxIters; // number of iterations; defined by user.
```

```
    // internal buffers
    CvMat* left;
    CvMat* right;
    CvMat* dispLeft;
    CvMat* dispRight;
    CvMat* ptrLeft;
    CvMat* ptrRight;
    CvMat* vtxBuf;
    CvMat* edgeBuf;
}
CvStereoGCState;
```

The graph cuts stereo correspondence algorithm, described in **'[Kolmogorov03]'_** (as **KZ1**), is non-realtime stereo correpsondence algorithm that usually gives very accurate depth map with well-defined object boundaries. The algorithm represents stereo problem as a sequence of binary optimization problems, each of those is solved using maximum graph flow algorithm. The state structure above should not be allocated and initialized manually; instead, use `cvCreateStereoGCState` and then override necessary parameters if needed.

`CvStereoGCState*` **cvCreateStereoGCState**(*int numberOfDisparities, int maxIters*)

Creates the state of graph cut-based stereo correspondence algorithm

**Parameters**
- *numberOfDisparities* – The number of disparities. The disparity search range will be ?<`state->minDisparity disparity state->minDisparity + state->numberOfDisparities`
- *maxIters* – Maximum number of iterations. On each iteration all possible (or reasonable) alpha-expansions are tried. The algorithm may terminate earlier if it could not find an alpha-expansion that decreases the overall cost function value. See **'[Kolmogorov03]'_** for details.

The function cvCreateStereoGCState creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to `cvFindStereoCorrespondenceGC`.

`void` **cvReleaseStereoGCState**(*CvStereoGCState** state*)

Releases the state structure of the graph cut-based stereo correspondence algorithm

**Parameter** *state* – Double pointer to the released structure

The function cvReleaseStereoGCState releases the stereo correspondence structure and all the associated internal buffers.

`void` **cvFindStereoCorrespondenceGC**(*const CvArr* left, const CvArr* right, CvArr* dispLeft, CvArr* dispRight, CvStereoGCState* state, int useDisparityGuess CV_DEFAULT(0)*)

Computes the disparity map using graph cut-based algorithm

**Parameters**
- *left* – The left single-channel, 8-bit image.
- *right* – The right image of the same size and the same type.
- *dispLeft* – The optional output single-channel 16-bit signed left disparity map of the same size as input images.
- *dispRight* – The optional output single-channel 16-bit signed right disparity map of the same size as input images.
- *state* – Stereo correspondence structure.
- *useDisparityGuess* – If the parameter is not zero, the algorithm will start with pre-defined disparity maps. Both dispLeft and dispRight should be valid disparity maps. Otherwise, the function starts with blank disparity maps (all pixels are marked as occlusions).

The function cvFindStereoCorrespondenceGC computes disparity maps for the input rectified stereo pair. Note that the left disparity image will contain values in the following range:

```
-state->numberOfDisparities-state->minDisparity <
dispLeft(x,y) ? -state->minDisparity,
or
dispLeft(x,y) == CV_STEREO_GC_OCCLUSION,
 where as for the right disparity image the following will
 be true: ::
state->minDisparity ? dispRight(x,y) <
state->minDisparity+state->numberOfDisparities,
or
dispRight(x,y) == CV_STEREO_GC_OCCLUSION,
```

that is, the range for the left disparity image will be inversed, and the pixels for which no good match has been found, will be marked as occlusions.

Here is how the function can be called

```
// image_left and image_right are the input 8-bit single-
channel images
// from the left and the right cameras, respectively
CvSize size = cvGetSize(image_left);
CvMat* disparity_left = cvCreateMat( size.height, size.width,
CV_16S );
CvMat* disparity_right = cvCreateMat( size.height,
size.width, CV_16S );
CvStereoGCState* state = cvCreateStereoGCState( 16, 2 );
cvFindStereoCorrespondenceGC( image_left, image_right,
    disparity_left, disparity_right, state, 0 );
cvReleaseStereoGCState( &state );
// now process the computed disparity images as you want ...
 and this is the output left disparity image computed from
 the well-known Tsukuba stereo pair and multiplied by -16 (because
 the values in the left disparity images are usually negative): ::
CvMat* disparity_left_visual = cvCreateMat( size.height,
size.width, CV_8U );
cvConvertScale( disparity_left, disparity_left_visual, -16 );
cvSave( "disparity.png", disparity_left_visual );
```

void **cvReprojectImageTo3D** (*const CvArr* disparity, CvArr* _3dImage, const CvMat* Q*)
Reprojects disparity image to 3D space

**Parameters**
- *disparity* – Disparity map.
- *_3dImage* – 3-channel, 16-bit integer or 32-bit floating- point image - the output map of 3D points.
- *Q* – The reprojection 4x4 matrix.

The function cvReprojectImageTo3D transforms 1-channel disparity map to 3-channel image, a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

```
[X Y Z W]T = Q*[x y d 1]T
_3dImage(x,y) = (X/W, Y/W, Z/W)
```

The matrix Q can be arbitrary, e.g. the one, computed by `cvStereoRectify`. To reproject a sparse set of points {(x,y,d),...} to 3D space, use `cvPerspectiveTransform`.

## 1.3 Bibliography

This bibliography provides a list of publications that were might be useful to the OpenCV users. This list is not complete; it serves only as a starting point.


1. **[Agrawal08]** Motilal Agrawal, Kurt Konolige, and Morten Rufus Blas “CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching”. ECCV 2008, Part IV, LNCS 5305, pp. 102-115, 2008.
2. **[Bay06]** Herbert Bay, Tinne Tuytelaars and Luc Van Gool “SURF: Speeded Up Robust Features”, Proceedings of the 9th European Conference on Computer Vision, Springer LNCS volume 3951, part 1, pp 404–417, 2006.

3. **[Beis97]** J.S. Beis and D.G. Lowe, "Shape indexing using approximate nearest-neighbor search in highdimensional spaces". In Proc. IEEE Conf. Comp. Vision Patt. Recog., pages 1000–1006, 1997.
4. **[Borgefors86]** Gunilla Borgefors, "Distance Transformations in Digital Images". Computer Vision, Graphics and Image Processing 34, 344-371 (1986).
5. **[Bouguet00]** Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker.

**The paper is included into OpenCV distribution ('algo_tracking.pdf'_)** 1. **[Bouguet04]** Jean-Yves Bouguet. The Camera Calibration Toolbox for Matlab. **'http://www.vision.caltech.edu/bouguetj/calib_doc/'_**

2. **[Bradski98]** XXX Unicode decode Error ? XXX

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'_**)

1. **[Bradski00]** G. Bradski and J. Davis. Motion Segmentation and Pose Recognition with Motion History Gradients. IEEE WACV'00, 2000.
2. **[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.
3. **[Canny86]** J. Canny. A Computational Approach to Edge Detection, IEEE Trans. on Pattern Analysis and Machine Intelligence, 8(6), pp. 679-698 (1986).
4. **[Davis97]** J. Davis and Bobick. The Representation and Recognition of Action Using Temporal Templates. MIT Media Lab Technical Report 402, 1997.
5. **[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.
6. **[Felzenszwalb04]** Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance Transforms of Sampled Functions. Cornell Computing and Information Science TR2004-1963.
7. **[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.
8. **[Ford98]** Adrian Ford, Alan Roberts. Colour Space Conversions. **'http://www.poynton.com/PDFs/coloureq.pdf'_**
9. **[Hartley99]** XXX Unicode decode Error ? XXX
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'_** 1. **[Jahne97]** B. Jahne. Digital Image Processing. Springer, New York, 1997.

2. **[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.
3. **[Kass88]** M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models, International Journal of Computer Vision, pp. 321-331, 1988.
4. **[Kolmogorov03]**V. Kolmogorov. Graph Based Algorithms for Scene Reconstruction from Two or More Views. PhD thesis, Cornell University, September 2003.

5. **[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'_**

1. **[Matas98]** J.Matas, C.Galambos, J.Kittler. Progressive Probabilistic Hough Transform. British Machine Vision Conference, 1998.
2. **[Meyer92]** Meyer, F. (1992). Color image segmentation. In Proceedings of the International Conference on Image Processing and its Applications, pages 303–306.
3. **[Rosenfeld73]** A. Rosenfeld and E. Johnston. Angle Detection on Digital Curves. IEEE Trans. Computers, 22:875-878, 1973.
4. **[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.
5. **[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.
6. **[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'_
7. **[Serra82]** J. Serra. Image Analysis and Mathematical Morphology. Academic Press, 1982.
8. **[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.
9. **[Suzuki85]** S. Suzuki, K. Abe. Topological Structural Analysis of Digital Binary Images by Border Following. CVGIP, v.30, n.1. 1985, pp. 32-46.
10. **[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.
11. **[Telea04]** XXX Unicode decode Error ? XXX
12. **[Trucco98]** Emanuele Trucco, Alessandro Verri. Introductory Techniques for 3-D Computer Vision. Prentice Hall, Inc., 1998.
13. **[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/'_** 1. **[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/kalmanIntro.html'_

1. **[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.
2. **[Yuen03]** H.K. Yuen, J. Princen, J. Illingworth and J. Kittler. Comparative study of Hough Transform methods for circle finding.

‘ http://www.sciencedirect.com/science/article/B6V09-48TCV4N- 5Y/2/91f551d124777f7a4cf7b18325235673‘_

1. **[Yuille89]** A.Y.Yuille, D.S.Cohen, and P.W.Hallinan. Feature Extraction from Faces Using Deformable Templates in CVPR, pp. 104-109, 1989.
2. **[Zhang96]** Z. Zhang. Parameter Estimation Techniques: A Tutorial with Application to Conic Fitting, Image and Vision Computing Journal, 1996.
3. **[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.
4. **[Zhang00]** Z. Zhang. A Flexible New Technique for Camera Calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(11):1330-1334, 2000.

## 1.4 `cvaux` – Experimental and Obsolete Functionality

**Note:** to be written

## 1.5 `highgui` – Simple GUI and Utility I/O Functions

While OpenCV is intended and designed for being used in production level applications, HighGUI is just an addendum for quick software prototypes and experimentation setups. The general idea behind its design is to have a small set of directly useable functions to interface your computer vision code with the environment.

Usually, you will need to get source images into your program and resulting image out to disk. In addition, simple methods to display images on screen and to allow (limited) user input are provided.

**Note:** None of the methods implemented in HighGUI allow for building sleek user interfaces with production level error handling. If you intend to build end user applications, don’t use HighGUI for this. In the opposite, look at native libraries for your target system. For example: camera input methods in HighGUI are designed to be easily useable. However, there are no means to react on cameras being plugged in or out during run time, etc.

**Note:** to be written

## 1.6 `ml` – Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different set of features (like ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class `CvStatModel` that all the other ML classes are derived from.

**Note:** to be written

CHAPTER

TWO

# INDEX AND SEARCH

CHAPTER

THREE

# FURTHER INFORMATION

- Installation
- Frequently Asked Questions

# MODULE INDEX

# INDEX

## C

## H



## I



## M



## P