The OpenCV 1.x C Reference Manual

Release 2.3

June 20, 2011

CONTENTS

1 core. The Core Functionality **1**
1.1 Basic Structures . . . 1
1.2 Operations on Arrays . . . 10
1.3 Dynamic Structures . . . 66
1.4 Drawing Functions . . . 96
1.5 XML/YAML Persistence . . . 104
1.6 Clustering . . . 121
1.7 Utility and System Functions and Macros . . . 125

2 imgproc. Image Processing **133**
2.1 Histograms . . . 133
2.2 Image Filtering . . . 142
2.3 Geometric Image Transformations . . . 148
2.4 Miscellaneous Image Transformations . . . 156
2.5 Structural Analysis and Shape Descriptors . . . 169
2.6 Planar Subdivisions . . . 187
2.7 Motion Analysis and Object Tracking . . . 196
2.8 Feature Detection . . . 197
2.9 Object Detection . . . 205

3 features2d. Feature Detection and Descriptor Extraction **207**
3.1 Feature detection and description . . . 207

4 objdetect. Object Detection **211**
4.1 Cascade Classification . . . 211

5 video. Video Analysis **219**
5.1 Motion Analysis and Object Tracking . . . 219

6 highgui. High-level GUI and Media I/O **233**
6.1 User Interface . . . 233
6.2 Reading and Writing Images and Video . . . 239
6.3 Qt new functions . . . 245

7 calib3d. Camera Calibration, Pose Estimation and Stereo **253**
7.1 Camera Calibration and 3d Reconstruction . . . 253

CHAPTER

ONE

CORE. THE CORE FUNCTIONALITY

1.1 Basic Structures

CvPoint

CvPoint

2D point with integer coordinates (usually zero-based).

```
typedef struct CvPoint
{
    int x;
    int y;
}
CvPoint;
```

x
    x-coordinate

y
    y-coordinate

```
/* Constructor */
inline CvPoint cvPoint( int x, int y );

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

CvPoint2D32f

CvPoint2D32f

2D point with floating-point coordinates

```
typedef struct CvPoint2D32f
{
    float x;
    float y;
}
CvPoint2D32f;
```

x
    x-coordinate

y
y-coordinate

```
/* Constructor */
inline CvPoint2D32f cvPoint2D32f( double x, double y );

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

CvPoint3D32f

CvPoint3D32f

3D point with floating-point coordinates

```
typedef struct CvPoint3D32f
{
    float x;
    float y;
    float z;
}
CvPoint3D32f;
```

x
x-coordinate

y
y-coordinate

z
z-coordinate

```
/* Constructor */
inline CvPoint3D32f cvPoint3D32f( double x, double y, double z );
```

CvPoint2D64f

CvPoint2D64f

2D point with double precision floating-point coordinates

```
typedef struct CvPoint2D64f
{
    double x;
    double y;
}
CvPoint2D64f;
```

x
x-coordinate

y
y-coordinate

```
/* Constructor */
inline CvPoint2D64f cvPoint2D64f( double x, double y );
```

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

CvPoint3D64f

`CvPoint3D64f`

3D point with double precision floating-point coordinates

```
typedef struct CvPoint3D64f
{
    double x;
    double y;
    double z;
}
CvPoint3D64f;
```

x
: x-coordinate

y
: y-coordinate

z
: z-coordinate

```
/* Constructor */
inline CvPoint3D64f cvPoint3D64f( double x, double y, double z );
```

CvSize

`CvSize`

Pixel-accurate size of a rectangle.

```
typedef struct CvSize
{
    int width;
    int height;
}
CvSize;
```

width
: Width of the rectangle

height
: Height of the rectangle

```
/* Constructor */
inline CvSize cvSize( int width, int height );
```

CvSize2D32f

`CvSize2D32f`

Sub-pixel accurate size of a rectangle.

```
typedef struct CvSize2D32f
{
    float width;
    float height;
}
CvSize2D32f;
```

width
Width of the rectangle

height
Height of the rectangle

```
/* Constructor */
inline CvSize2D32f cvSize2D32f( double width, double height );
```

CvRect

CvRect

Offset (usually the top-left corner) and size of a rectangle.

```
typedef struct CvRect
{
    int x;
    int y;
    int width;
    int height;
}
CvRect;
```

x
x-coordinate of the top-left corner

y
y-coordinate of the top-left corner (bottom-left for Windows bitmaps)

width
Width of the rectangle

height
Height of the rectangle

```
/* Constructor */
inline CvRect cvRect( int x, int y, int width, int height );
```

CvScalar

CvScalar

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

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

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

/* Constructor:
initializes val[0] with val0, and all of val[1]...val[3] with zeros
*/
inline CvScalar cvRealScalar( double val0 );
```

CvTermCriteria

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;
    int    max_iter;
    double epsilon;
}
CvTermCriteria;
```

type
A combination of CV _ TERMCRIT _ ITER and CV _ TERMCRIT _ EPS

max_iter
Maximum number of iterations

epsilon
Required accuracy

```
/* Constructor */
inline CvTermCriteria cvTermCriteria( int type, int max_iter, double epsilon );

/* Check and transform a CvTermCriteria 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

CvMat

A multi-channel matrix.

```
typedef struct CvMat
{
    int type;
    int step;

    int* refcount;

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

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

    union
    {
        int cols;
        int width;
    };
#else
    int rows;
    int cols;
#endif

} CvMat;
```

type
A CvMat signature (CV_MAT_MAGIC_VAL) containing the type of elements and flags

step
Full row length in bytes

refcount
Underlying data reference counter

data
Pointers to the actual matrix data

rows
Number of rows

cols
Number of columns

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

CvMatND

CvMatND

Multi-dimensional dense multi-channel array.

```
typedef struct CvMatND
{
    int type;
    int dims;

    int* refcount;

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

    struct
    {
        int size;
        int step;
    }
    dim[CV_MAX_DIM];

} CvMatND;
```

type
A CvMatND signature (CV _ MATND _ MAGIC _ VAL), combining the type of elements and flags

dims
The number of array dimensions

refcount
Underlying data reference counter

data
Pointers to the actual matrix data

dim
For each dimension, the pair (number of elements, distance between elements in bytes)

CvSparseMat

CvSparseMat

Multi-dimensional sparse multi-channel array.

```
typedef struct CvSparseMat
{
    int type;
    int dims;
    int* refcount;
    struct CvSet* heap;
    void** hashtable;
    int hashsize;
    int valoffset;
    int idxoffset;
    int size[CV_MAX_DIM];
```

```
} CvSparseMat;
```

type
: A CvSparseMat signature (CV _ SPARSE _ MAT _ MAGIC _ VAL), combining the type of elements and flags.

dims
: Number of dimensions

refcount
: Underlying reference counter. Not used.

heap
: A pool of hash table nodes

hashtable
: The hash table. Each entry is a list of nodes.

hashsize
: Size of the hash table

valoffset
: The value offset of the array nodes, in bytes

idxoffset
: The index offset of the array nodes, in bytes

size
: Array of dimension sizes

IplImage

IplImage

IPL image header

```
typedef struct _IplImage
{
    int  nSize;
    int  ID;
    int  nChannels;
    int  alphaChannel;
    int  depth;
    char colorModel[4];
    char channelSeq[4];
    int  dataOrder;
    int  origin;
    int  align;
    int  width;
    int  height;
    struct _IplROI *roi;
    struct _IplImage *maskROI;
    void  *imageId;
    struct _IplTileInfo *tileInfo;
    int  imageSize;
    char *imageData;
    int  widthStep;
    int  BorderMode[4];
    int  BorderConst[4];
```

```
    char *imageDataOrigin;
}
IplImage;
```

nSize
`sizeof(IplImage)`

ID
Version, always equals 0

nChannels
Number of channels. Most OpenCV functions support 1-4 channels.

alphaChannel
Ignored by OpenCV

depth
Channel depth in bits + the optional sign bit (`IPL_DEPTH_SIGN`). The supported depths are:

IPL_DEPTH_8U
Unsigned 8-bit integer

IPL_DEPTH_8S
Signed 8-bit integer

IPL_DEPTH_16U
Unsigned 16-bit integer

IPL_DEPTH_16S
Signed 16-bit integer

IPL_DEPTH_32S
Signed 32-bit integer

IPL_DEPTH_32F
Single-precision floating point

IPL_DEPTH_64F
Double-precision floating point

colorModel
Ignored by OpenCV. The OpenCV function *CvtColor* requires the source and destination color spaces as parameters.

channelSeq
Ignored by OpenCV

dataOrder
0 = `IPL_DATA_ORDER_PIXEL` - interleaved color channels, 1 - separate color channels. *CreateImage* only creates images with interleaved channels. For example, the usual layout of a color image is: $b_{00}g_{00}r_{00}b_{10}g_{10}r_{10}...$

origin
0 - top-left origin, 1 - bottom-left origin (Windows bitmap style)

align
Alignment of image rows (4 or 8). OpenCV ignores this and uses widthStep instead.

width
Image width in pixels

height
Image height in pixels

roi
Region Of Interest (ROI). If not NULL, only this image region will be processed.

maskROI
Must be NULL in OpenCV

imageId
Must be NULL in OpenCV

tileInfo
Must be NULL in OpenCV

imageSize
Image data size in bytes. For interleaved data, this equals `image->height` · `image->widthStep`

imageData
A pointer to the aligned image data

widthStep
The size of an aligned image row, in bytes

BorderMode
Border completion mode, ignored by OpenCV

BorderConst
Border completion mode, ignored by OpenCV

imageDataOrigin
A pointer to the origin of the image data (not necessarily aligned). This is used for image deallocation.

The *IplImage* structure was inherited from the Intel Image Processing Library, in which the format is native. OpenCV only supports a subset of possible *IplImage* formats, as outlined in the parameter list above.

In addition to the above restrictions, OpenCV handles ROIs differently. OpenCV functions require that the image size or ROI size of all source and destination images match exactly. On the other hand, the Intel Image Processing Library processes the area of intersection between the source and destination images (or ROIs), allowing them to vary independently.

CvArr

CvArr

Arbitrary array

```
typedef void CvArr;
```

The metatype `CvArr` is used *only* as a function parameter to specify that the function accepts arrays of multiple types, such as IplImage*, CvMat* or even CvSeq* sometimes. The particular array type is determined at runtime by analyzing the first 4 bytes of the header.

1.2 Operations on Arrays

AbsDiff

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 calculates absolute difference between two arrays.

$$\mathtt{dst}(i)_c = |\mathtt{src1}(I)_c - \mathtt{src2}(I)_c|$$

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

AbsDiffS

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

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

param src The source array

param dst The destination array

param value The scalar

The function calculates absolute difference between an array and a scalar.

$$\mathtt{dst}(i)_c = |\mathtt{src}(I)_c - \mathtt{value}_c|$$

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

Add

void **cvAdd**(const CvArr* *src1*, const CvArr* *src2*, CvArr* *dst*, const CvArr* *mask=NULL*)
Computes the 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 the destination array to be changed

The function adds one array to another:

```
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). For types that have limited range this operation is saturating.

AddS

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

Computes the sum of an array and a scalar.

Parameters

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

The function adds a scalar `value` to every element in the source array `src1` and stores the result in `dst` . For types that have limited range this operation is saturating.

```
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).

AddWeighted

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

Computes the weighted sum of two arrays.

Parameters

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

The function calculates the weighted sum of two arrays as follows:

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

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

And

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 the destination array to be changed

The function 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.

AndS

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

Calculates per-element bit-wise conjunction of an array and a 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 the destination array to be changed

The function calculates per-element bit-wise conjunction of an array and a scalar:

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

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). 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 the 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, absMask = 0x7fffffff;
cvAndS(&A, cvRealScalar(*(float*)&absMask), &A, 0);
for(i = 0; i < 9; i++ )
    printf("
```

The code should print:

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

Avg

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 calculates the average value `M` of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \ne 0)$$
$$M_c = \frac{\sum_{I,\,\texttt{mask}(I) \ne 0} \texttt{arr}(I)_c}{N}$$

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 S_0 .

AvgSdv

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

Calculates average (mean) of array elements.

Parameters

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

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

$$N = \sum_I (\texttt{mask}(I) \ne 0)$$
$$mean_c = \frac{1}{N} \sum_{I,\,\texttt{mask}(I) \ne 0} \texttt{arr}(I)_c$$
$$stdDev_c = \sqrt{\frac{1}{N} \sum_{I,\,\texttt{mask}(I) \ne 0} (\texttt{arr}(I)_c - mean_c)^2}$$

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 the output scalars ($mean_0$ and $stdDev_0$).

CalcCovarMatrix

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

Calculates covariance matrix of a set of vectors.

Parameters

- **vects** – The input vectors, all of which must have the same type and the same size. The vectors do not have to be 1D, they can be 2D (e.g., images) and so forth
- **count** – The number of input vectors
- **covMat** – The output covariance 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 covariance matrix is calculated as:

    $$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]$$

    , that is, the covariance matrix is $\texttt{count} \times \texttt{count}$. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces technique for face recognition). Eigenvalues of this “scrambled” matrix will match the eigenvalues of the true covariance matrix and the “true” eigenvectors can be easily calculated from the eigenvectors of the “scrambled” covariance matrix.

– **CV_COVAR_NORMAL** The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...] \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T$$

, that is, `covMat` will be a covariance matrix with the same linear size as the total number of elements in each 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 covariance 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 covariance matrix is scaled. In the "normal" mode `scale` is '1./count'; in the "scrambled" mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled ('scale=1').

– **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 calculates the covariance matrix and, optionally, the mean vector of the set of input vectors. The function can be used for PCA, for comparing vectors using Mahalanobis distance and so forth.

CartToPolar

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

Calculates the 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 $(0$ to $2\pi)$ or in degrees (0 to 360 degrees).
- **angleInDegrees** – The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the 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 degree accuracy. For the (0,0) point, the angle is set to 0.

Cbrt

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

Parameters

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

The function calculates the cubic root of the argument, and normally it is faster than `pow(value,1./3)` . In addition, negative arguments are handled properly. Special values ($\pm\infty$, NaN) are not handled.

ClearND

void **cvClearND** (CvArr* *arr*, int* *idx*)
Clears a specific array element.

Parameters

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

The function *ClearND* clears (sets to zero) a specific element of a dense array or deletes the element of a sparse array. If the sparse array element does not exists, the function does nothing.

CloneImage

IplImage* **cvCloneImage** (const IplImage* *image*)
Makes a full copy of an image, including the header, data, and ROI.

Parameters

- **image** – The original image

The returned `IplImage*` points to the image copy.

CloneMat

CvMat* **cvCloneMat** (const CvMat* *mat*)
Creates a full matrix copy.

Parameters

- **mat** – Matrix to be copied

Creates a full copy of a matrix and returns a pointer to the copy.

CloneMatND

CvMatND* **cvCloneMatND** (const CvMatND* *mat*)
Creates full copy of a multi-dimensional array and returns a pointer to the copy.

Parameters

- **mat** – Input array

CloneSparseMat

CvSparseMat* **cvCloneSparseMat** (const CvSparseMat* *mat*)

Creates full copy of sparse array.

Parameters

- **mat** – Input array

The function creates a copy of the input array and returns pointer to the copy.

Cmp

void **cvCmp** (const CvArr* *src1*, const CvArr* *src2*, CvArr* *dst*, int *cmpOp*)

Performs per-element comparison of two arrays.

Parameters

- **src1** – The first source array
- **src2** – The second source array. Both source arrays must have a single channel.
- **dst** – The destination array, must have 8u or 8s type
- **cmpOp** – 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 compares the corresponding elements of two arrays and fills the destination mask array:

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

`dst(I)` is set to 0xff (all `1` -bits) if the specific 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)

CmpS

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

Performs per-element comparison of an array and a 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
- **cmpOp** – 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 compares the corresponding elements of an array and a scalar and fills the destination mask array:

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

where `op` is $=, >, \geq, <, \leq$ *or* $\neq$.

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

ConvertScale

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 )
```

param src Source array

param dst Destination array

param scale Scale factor

param shift Value added to the scaled source array elements

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

$$\texttt{dst}(I) = \texttt{scale} \texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)$$

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

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

In the case of `scale=1, shift=0` no prescaling is done. This is a specially optimized case and it has the appropriate *Convert* name. 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 is caled *Scale* .

ConvertScaleAbs

void **cvConvertScaleAbs** (const CvArr* *src*, CvArr* *dst*, double *scale=1*, double *shift=0*)
Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

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 is similar to *ConvertScale* , but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scale} \texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of *ConvertScale* and *Abs* functions.

CvtScaleAbs

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

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

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 is similar to *ConvertScale* , but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scale} \texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of *ConvertScale* and *Abs* functions.

Copy

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 the destination array to be changed

The function copies selected elements from an input array to an output array:

$$\texttt{dst}(I) = \texttt{src}(I) \quad \text{if} \quad \texttt{mask}(I) \neq 0.$$

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).

CountNonZero

int **cvCountNonZero**(const CvArr* *arr*)

Counts non-zero array elements.

Parameters

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

The function returns the number of non-zero elements in arr:

$$\sum_I (\texttt{arr}(I) \neq 0)$$

In the case of `IplImage` both ROI and COI are supported.

CreateData

void **cvCreateData**(CvArr* *arr*)

Allocates array data

Parameters

- **arr** – Array header

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

CreateImage

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

Creates an image header and allocates the image data.

Parameters

- **size** – Image width and height
- **depth** – Bit depth of image elements. See *IplImage* for valid depths.
- **channels** – Number of channels per pixel. See *IplImage* for details. This function only creates images with interleaved channels.

This call is a shortened form of

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

CreateImageHeader

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

Creates an image header but does not allocate the image data.

Parameters

- **size** – Image width and height
- **depth** – Image depth (see *CreateImage*)
- **channels** – Number of channels (see *CreateImage*)

This call is an analogue of

```
hdr=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);
```

but it does not use IPL functions by default (see the `CV_TURN_ON_IPL_COMPATIBILITY` macro).

CreateMat

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

Creates a matrix header and allocates the matrix data.

Parameters

- **rows** – Number of rows in the matrix
- **cols** – Number of columns in the matrix
- **type** – The type of the matrix elements in the form `CV_<bit depth><S|U|F>C<number of channels>` , where S=signed, U=unsigned, F=float. For example, CV _ 8UC1 means the elements are 8-bit unsigned and the there is 1 channel, and CV _ 32SC2 means the elements are 32-bit signed and there are 2 channels.

This is the concise form for:

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

CreateMatHeader

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

Creates a matrix header but does not allocate the matrix data.

Parameters

- **rows** – Number of rows in the matrix
- **cols** – Number of columns in the matrix
- **type** – Type of the matrix elements, see *CreateMat*

The function allocates a new matrix header and returns a pointer to it. The matrix data can then be allocated using *CreateData* or set explicitly to user-allocated data via *SetData* .

CreateMatND

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

Creates the header and allocates the data for a multi-dimensional dense array.

Parameters

- **dims** – Number of array dimensions. This must not exceed CV _ MAX _ DIM (32 by default, but can be changed at build time).
- **sizes** – Array of dimension sizes.

- **type** – Type of array elements, see *CreateMat* .

This is a short form for:

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

CreateMatNDHeader

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

Creates a new matrix header but does not allocate the matrix data.

Parameters

- **dims** – Number of array dimensions
- **sizes** – Array of dimension sizes
- **type** – Type of array elements, see *CreateMat*

The function allocates a header for a multi-dimensional dense array. The array data can further be allocated using *CreateData* or set explicitly to user-allocated data via *SetData* .

CreateSparseMat

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

Creates sparse array.

Parameters

- **dims** – Number of array dimensions. In contrast to the dense matrix, the number of dimensions is practically unlimited (up to 2^{16}).
- **sizes** – Array of dimension sizes
- **type** – Type of array elements. The same as for CvMat

The function allocates a multi-dimensional sparse array. Initially the array contain no elements, that is *Get* or *GetReal* returns zero for every index.

CrossProduct

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

Calculates the cross product of two 3D vectors.

Parameters

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

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

$$\texttt{dst} = \texttt{src1} \times \texttt{src2}$$

or:

$$\begin{array}{l} \texttt{dst}_1 = \texttt{src1}_2 \texttt{src2}_3 - \texttt{src1}_3 \texttt{src2}_2 \\ \texttt{dst}_2 = \texttt{src1}_3 \texttt{src2}_1 - \texttt{src1}_1 \texttt{src2}_3 \\ \texttt{dst}_3 = \texttt{src1}_1 \texttt{src2}_2 - \texttt{src1}_2 \texttt{src2}_1 \end{array}$$

CvtPixToPlane

Synonym for *Split* .

DCT

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

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

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 a forward 1D or 2D transform.
  - **CV_DXT_INVERSE** do an inverse 1D or 2D transform.
  - **CV_DXT_ROWS** do a 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 and so forth.

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

Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C^{(N)}_{jk} = \sqrt{\alpha_j / N} \cos \left(\frac{\pi (2k+1) j}{2N} \right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.

Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)} \right)^{-1} \cdot Y = \left(C^{(N)} \right)^T \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)} \right)^T = I$)

Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)} \right)^T$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)} \right)^T \cdot X \cdot C^{(N)}$$

DFT

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

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

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 a forward 1D or 2D transform. The result is not scaled.
  - **CV_DXT_INVERSE** do an 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 a forward or inverse transform of every individual row of the input matrix. This flag allows the 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 and so forth.
  - **CV_DXT_INVERSE_SCALE** same as `CV_DXT_INVERSE + CV_DXT_SCALE`
- **nonzeroRows** – Number of nonzero rows in the source array (in the case of a forward 2d transform), or a number of rows of interest in the destination array (in the case of an 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 via DFT. See the example below.

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

Forward Fourier transform of 1D vector of N elements:

$$y = F^{(N)} \cdot x, where F^{(N)}_{jk} = exp(-i \cdot 2\pi \cdot j \cdot k/N)$$

,

$$i = sqrt(-1)$$

Inverse Fourier transform of 1D vector of N elements:

$$x' = (F^{(N)})^{-1} \cdot y = conj(F^(N)) \cdot yx = (1/N) \cdot x$$

Forward Fourier transform of 2D vector of M × N elements:

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

Inverse Fourier transform of 2D vector of M × N elements:

$$X' = conj(F^{(M)}) \cdot Y \cdot conj(F^{(N)}) X = (1/(M \cdot N)) \cdot X'$$

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

$$\begin{bmatrix} ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\ ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\ ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\ \multicolumn{9}{c}{\dotfill} \\ ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \multicolumn{3}{c}{\dotfill} & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\ ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \multicolumn{3}{c}{\dotfill} & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\ ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \multicolumn{3}{c}{\dotfill} & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2} \end{bmatrix}$$

Note: the last column is present if `N` is even, the last row is present if `M` is even. In the case of 1D real transform the result looks like the first row of the above matrix.

Here is the example of how to compute 2D convolution using DFT.

```
CvMat* A = cvCreateMat(M1, N1, CVg32F);
CvMat* B = cvCreateMat(M2, N2, A->type);

// it is also possible to have only abs(M2-M1)+1 times 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 dftgM = cvGetOptimalDFTSize(A->rows + B->rows - 1);
int dftgN = cvGetOptimalDFTSize(A->cols + B->cols - 1);

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

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

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

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

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

cvMulSpectrums(dftgA, dft_B, dft_A, 0 /* or CV_DXT_MUL_CONJ to get
                correlation rather than convolution */);

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

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

cvCopy(&tmp, conv);
```

DecRefData

void **cvDecRefData**(CvArr* *arr*)

Decrements an array data reference counter.

Parameters

- **arr** – Pointer to an array header

The function decrements the data reference counter in a *CvMat* or *CvMatND* if the reference counter pointer is not NULL. If the counter reaches zero, the data is deallocated. In the current implementation the reference counter is not NULL only if the data was allocated using the *CreateData* function. The counter will be NULL in other cases such as: external data was assigned to the header using *SetData* , the matrix header is part of a larger matrix or image, or the header was converted from an image or n-dimensional matrix header.

Det

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

Returns the determinant of a matrix.

Parameters

- **mat** – The source matrix

The function returns the 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 $U = V = 0$ and then calculate the determinant as a product of the diagonal elements of W .

Div

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 divides one array by another:

$$\texttt{dst}(I) = \begin{cases} \texttt{scale} \cdot \texttt{src1}(I)/\texttt{src2}(I) & \text{if } \texttt{src1} \text{ is not } \texttt{NULL} \\ \texttt{scale}/\texttt{src2}(I) & \text{otherwise} \end{cases}$$

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

DotProduct

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

Calculates the dot product of two arrays in Euclidian metrics.

Parameters

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

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

$$src1 \bullet src2 = \sum_I (\texttt{src1}(I)\texttt{src2}(I))$$

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

EigenVV

void **cvEigenVV**(CvArr* *mat*, CvArr* *evects*, CvArr* *evals*, double *eps=0*, int lowindex = *-1*, int highindex = *-1*)

Computes eigenvalues and eigenvectors of a symmetric matrix.

Parameters

- **mat** – The input symmetric square matrix, modified during the processing
- **evects** – The output matrix of eigenvectors, stored as subsequent rows
- **evals** – The output vector of eigenvalues, stored in the descending order (order of eigenvalues and eigenvectors is syncronized, of course)
- **eps** – Accuracy of diagonalization. Typically, `DBL_EPSILON` (about 10^{-15}) works well. THIS PARAMETER IS CURRENTLY IGNORED.
- **lowindex** – Optional index of largest eigenvalue/-vector to calculate. (See below.)
- **highindex** – Optional index of smallest eigenvalue/-vector to calculate. (See below.)

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

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

If either low- or highindex is supplied the other is required, too. Indexing is 0-based. Example: To calculate the largest eigenvector/-value set `lowindex=highindex=0` . To calculate all the eigenvalues, leave `lowindex=highindex=-1` . For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

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

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

Exp

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

Calculates the 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 calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}(I)}$$

The maximum relative error is about 7×10^{-6} . Currently, the function converts denormalized values to zeros on output.

FastArctan

float **cvFastArctan** (float *y*, float *x*)

Calculates the angle of a 2D vector.

Parameters

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

The function calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0 degrees to 360 degrees. The accuracy is about 0.1 degrees.

Flip

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

Flip a 2D array around vertical, horizontal or both axes.

Parameters

- **src** – Source array
- **dst** – Destination array. If $\texttt{dst} = \texttt{NULL}$ the flipping is done in place.
- **flipMode** – Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas:

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

$$dst(i,j) = \begin{cases} \texttt{src}(rows(\texttt{src}) - i - 1, j) & \text{if } \texttt{flipMode} = 0 \\ \texttt{src}(i, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} > 0 \\ \texttt{src}(rows(\texttt{src}) - i - 1, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (flipMode = 0) to switch between top-left and bottom-left image origin, which is a 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 (flipMode > 0)

- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flipMode < 0)
- reversing the order of 1d point arrays (flipMode > 0)

GEMM

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

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

Performs generalized matrix multiplication.

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 a 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

$$\texttt{alpha} \, \texttt{src1} ^T \, \texttt{src2} + \texttt{beta} \, \texttt{src3} ^T$$

The function performs generalized matrix multiplication:

$$\texttt{dst} = \texttt{alpha} \, op(\texttt{src1}) \, op(\texttt{src2}) + \texttt{beta} \, op(\texttt{src3}) \quad \text{where $op(X)$ is X or X^T}$$

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

Get?D

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

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

GetCol(s)

CvMat* **cvGetCol** (const CvArr* *arr*, CvMat* *submat*, int *col*)
Returns array column or column span.

CvMat* **cvGetCols** (const CvArr* *arr*, CvMat* *submat*, int *startCol*, int *endCol*)

Parameters

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

The functions `GetCol` and `GetCols` return the header, corresponding to a specified column span of the input array. `GetCol` is a shortcut for *GetCols* :

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

GetDiag

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 , 1 corresponds to the diagonal below the main, and so forth.

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

cvGetDims, cvGetDimSize

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

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

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

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 the array dimensionality and the array of dimension sizes. In the 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);
```

GetElemType

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

Returns type of array elements.

Parameters

- **arr** – Input array

The function returns type of the array elements as described in *CreateMat* discussion: `CV_8UC1` ... `CV_64FC4` .

GetImage

IplImage* **cvGetImage**(const CvArr* *arr*, IplImage* *imageHeader*)

Returns image header for arbitrary array.

Parameters

- **arr** – Input array
- **imageHeader** – Pointer to `IplImage` structure used as a temporary buffer

The function returns the image header for the input array that can be a matrix - *CvMat* , or an image - `IplImage*` . In the case of an image the function simply returns the input pointer. In the case of *CvMat* it initializes an `imageHeader` structure with the 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.

GetImageCOI

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

Returns the index of the channel of interest.

Parameters

- **image** – A pointer to the image header

Returns the channel of interest of in an IplImage. Returned values correspond to the `coi` in *SetImageCOI* .

GetImageROI

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

Returns the image ROI.

Parameters

- **image** – A pointer to the image header

If there is no ROI set, `cvRect(0,0,image->width,image->height)` is returned.

GetMat

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

Returns matrix header for arbitrary array.

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 CvMatND has two dimensions) or 1D matrix (when CvMatND has 1 dimension or more than 2 dimensions). The array must be continuous.

The function returns a matrix header for the input array that can be a matrix -

CvMat , an image - `IplImage` or a 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 the `header` structure with parameters of the current image ROI and returns the 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 arrays - `IplImage` and *CvMat* - using the same code. Reverse transform from *CvMat* to `IplImage` can be done using the *GetImage* function.

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 the pointer to the selected plane and COI = 0. It enables per-plane processing of multi-channel images with planar data layout using OpenCV functions.

GetNextSparseNode

CvSparseNode* **cvGetNextSparseNode** (CvSparseMatIterator* *matIterator*)

Returns the next sparse matrix element

Parameters

- **matIterator** – Sparse array iterator

The function 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 the hash table. The sample below demonstrates how to iterate through the sparse matrix:

Using *InitSparseMatIterator* and *GetNextSparseNode* 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 ))
{
```

```
    /* get pointer to the element indices */
    int* idx = CV_NODE_IDX(array, node);
    /* get value of the element (assume that the type is CV_32FC1) */
    float val = *(float*)CV_NODE_VAL(array, node);
    printf("(");
    for(i = 0; i < dims; i++ )
        printf("
    printf("

    sum += val;
}

printf("nTotal sum =
```

GetOptimalDFTSize

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

Returns optimal DFT size for a given vector size.

Parameters

- **size0** – Vector size

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

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

GetRawData

void **cvGetRawData**(const CvArr* *arr*, uchar** *data*, int* *step=NULL*, CvSize* *roiSize=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
- **roiSize** – Output ROI size

The function 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, the parameters of ROI are returned.

The following example shows how to get access to array elements. GetRawData calculates the absolute value of the elements in 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]);
```

GetReal1D

double **cvGetReal1D**(const CvArr* *arr*, int *idx0*)

Return a specific element of single-channel 1D array.

Parameters

- **arr** – Input array. Must have a single channel.
- **idx0** – The first zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that *Get* function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

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

GetReal2D

double **cvGetReal2D**(const CvArr* *arr*, int *idx0*, int *idx1*)

Return a specific element of single-channel 2D 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

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that *Get* function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

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

GetReal3D

double **cvGetReal3D**(const CvArr* *arr*, int *idx0*, int *idx1*, int *idx2*)

Return a specific 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

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that *Get* function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

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

GetRealND

double cvGetRealND(const CvArr* arr, int* idx)->float

Return a specific element of single-channel array.

Parameters

- **arr** – Input array. Must have a single channel.
- **idx** – Array of the element indices

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that *Get* function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

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

GetRow(s)

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

Returns array row or row span.

CvMat* **cvGetRows**(const CvArr* *arr*, CvMat* *submat*, int *startRow*, int *endRow*, int *deltaRow=1*)

Parameters

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

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

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

GetSize

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

Returns size of matrix or image ROI.

Parameters

- **arr** – array header

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

GetSubRect

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 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.

InRange

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

Checks that array elements lie between the 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 does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0 \land \texttt{lower}(I)_1 <= \texttt{src}(I)_1 < \texttt{upper}(I)_1$$

For two-channel arrays and so forth,

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).

InRangeS

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 does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0 \land \texttt{lower}_1 <= \texttt{src}(I)_1 < \texttt{upper}_1$$

For two-channel arrays nd so forth,

'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).

IncRefData

int **cvIncRefData**(CvArr* *arr*)

Increments array data reference counter.

Parameters

- **arr** – Array header

The function 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.

InitImageHeader

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

Initializes an image header that was previously allocated.

Parameters

- **image** – Image header to initialize
- **size** – Image width and height
- **depth** – Image depth (see *CreateImage*)
- **channels** – Number of channels (see *CreateImage*)
- **origin** – Top-left `IPL_ORIGIN_TL` or bottom-left `IPL_ORIGIN_BL`
- **align** – Alignment for image rows, typically 4 or 8 bytes

The returned `IplImage*` points to the initialized header.

InitMatHeader

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

Initializes a pre-allocated matrix header.

Parameters

- **mat** – A pointer to the matrix header to be initialized
- **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
- **step** – Optional: full row width in bytes of the assigned data. By default, the minimal possible step is used which assumes there are no gaps between subsequent rows of the matrix.

This function is often used to process raw data with OpenCV matrix functions. For example, the following code computes the matrix product of two matrices, stored as ordinary arrays:

```
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);
// the c array now contains the product of a (3x4) and b (4x3)
```

InitMatNDHeader

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

Initializes a pre-allocated multi-dimensional array header.

Parameters

- **mat** – A pointer to the array header to be initialized
- **dims** – The number of array dimensions
- **sizes** – An array of dimension sizes
- **type** – Type of array elements, see *CreateMat*
- **data** – Optional data pointer assigned to the matrix header

InitSparseMatIterator

CvSparseNode* **cvInitSparseMatIterator**(const CvSparseMat* *mat*, CvSparseMatIterator* *matIterator*)

Initializes sparse array elements iterator.

Parameters

- **mat** – Input array
- **matIterator** – Initialized iterator

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

InvSqrt

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

Calculates the inverse square root.

Parameters

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

The function calculates the 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 ($\pm\infty$, NaN) are not handled.

Inv

Invert

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

Finds the inverse or pseudo-inverse of a matrix.

Parameters

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

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

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

In the case of `SVD` methods, the function returns the inversed condition 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.

IsInf

int **cvIsInf** (double *value*)

Determines if the argument is Infinity.

Parameters

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

The function returns 1 if the argument is $\pm\infty$ (as defined by IEEE754 standard), 0 otherwise.

IsNaN

int **cvIsNaN** (double *value*)

Determines if the argument is Not A Number.

Parameters

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

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

LUT

void **cvLUT**(const CvArr* *src*, CvArr* *dst*, const CvArr* *lut*)
Performs a look-up table transform of an array.

Parameters

- **src** – Source array of 8-bit elements
- **dst** – Destination array of a given 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 the 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 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 follows:

$$\texttt{dst}_i \leftarrow \texttt{lut}_{\texttt{src}_i + d}$$

where

$$d = \begin{cases} 0 & \text{if } \texttt{src} \text{ has depth } \texttt{CV\_8U} \\ 128 & \text{if } \texttt{src} \text{ has depth } \texttt{CV\_8S} \end{cases}$$

Log

void **cvLog**(const CvArr* *src*, CvArr* *dst*)
Calculates the natural logarithm of every array element's 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 calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}[I] = \begin{cases} \log|\texttt{src}(I) & \text{if } \texttt{src}[I] \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where `C` is a large negative number (about -700 in the current implementation).

Mahalanobis

double **cvMahalanobis**(const CvArr* *vec1*, const CvArr* *vec2*, CvArr* *mat*)
Calculates the Mahalanobis distance between two vectors.

Parameters

- **vec1** – The first 1D source vector
- **vec2** – The second 1D source vector

- **mat** – The inverse covariance matrix

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

$$d(\texttt{vec1}, \texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the *CalcCovarMatrix* function and further inverted using the *Invert* function (CV _ SVD method is the prefered one because the matrix might be singular).

Mat

CvMat **cvMat** (int *rows*, int *cols*, int *type*, void* *data=NULL*)
Initializes matrix header (lightweight 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

Initializes a matrix header and assigns data to it. The matrix is filled *row* -wise (the first `cols` elements of data form the first row of the matrix, etc.)

This function is a fast inline substitution for *InitMatHeader* . Namely, it is equivalent to:

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

Max

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 calculates per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

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

MaxS

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 calculates per-element maximum of array and scalar:

$$\texttt{dst}(I) = \max(\texttt{src}(I), \texttt{value})$$

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

Merge

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

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

```
#define cvCvtPlaneToPix cvMerge
```

param src0 Input channel 0

param src1 Input channel 1

param src2 Input channel 2

param src3 Input channel 3

param dst Destination array

The function is the opposite to *Split* . If the destination array has N channels then if the first N input channels are not NULL, they all are copied to the destination array; 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. The rest of the source channels (beyond the first N) must always be NULL. For IplImage *Copy* with COI set can be also used to insert a single channel into the image.

Min

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 calculates per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

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

MinMaxLoc

void **cvMinMaxLoc** (const CvArr* *arr*, double* *minVal*, double* *maxVal*, CvPoint* *minLoc=NULL*, CvPoint* *maxLoc=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
- **minVal** – Pointer to returned minimum value
- **maxVal** – Pointer to returned maximum value
- **minLoc** – Pointer to returned minimum location
- **maxLoc** – Pointer to returned maximum location
- **mask** – The optional mask used to select a subarray

The function finds minimum and maximum element values and their positions. The extremums are searched across the whole array, selected `ROI` (in the 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 the case of multi-dimensional arrays, `minLoc->x` and `maxLoc->x` will contain raw (linear) positions of the extremums.

MinS

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

Finds per-element minimum of an array and a scalar.

Parameters

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

The function calculates minimum of an array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src}(I), \texttt{value})$$

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

Mirror

Synonym for *Flip* .

MixChannels

void **cvMixChannels**(const CvArr** *src*, int *srcCount*, CvArr** *dst*, int *dstCount*, const int* *fromTo*, int *pairCount*)

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

Parameters

- **src** – Input arrays
- **srcCount** – The number of input arrays.
- **dst** – Destination arrays
- **dstCount** – The number of output arrays.
- **fromTo** – The array of pairs of indices of the planes copied. `fromTo[k*2]` is the 0-based index of the input channel in `src` and `fromTo[k*2+1]` is the index of the output channel in `dst` . Here the continuous channel numbering is used, that is, the first input image channels are indexed from `0` to `channels(src[0])-1` , the second input image channels are indexed from `channels(src[0])` to `channels(src[0]) +`

channels(src[1])-1 etc., and the same scheme is used for the output image channels. As a special case, when fromTo[k*2] is negative, the corresponding output channel is filled with zero.

The function is a generalized form of *cvSplit* and *Merge* and some forms of *CvtColor* . 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.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B swapped) and separate alpha channel image:

```
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);
cvSet(rgba, cvScalar(1,2,3,4));

CvArr* out[] = { bgr, alpha };
int from_to[] = { 0,2,  1,1,  2,0,  3,3 };
cvMixChannels(&bgra, 1, out, 2, from_to, 4);
```

MulAddS

Synonym for *ScaleAdd* .

Mul

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

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I)$$

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

MulSpectrums

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 as the source arrays
- **flags** – A combination of the following values;

- **CV_DXT_ROWS** treats each row of the arrays as a separate spectrum (see *DFT* parameters description).
- **CV_DXT_MUL_CONJ** conjugate the second source array before the multiplication.

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

The function, together with *DFT* , may be used to calculate convolution of two arrays rapidly.

MulTransposed

void **cvMulTransposed**(const CvArr* *src*, CvArr* *dst*, int *order*, const CvArr* *delta=NULL*, double *scale=1.0*)

Calculates the product of an array and a transposed array.

Parameters

- **src** – The source matrix
- **dst** – The destination matrix. Must be `CV_32F` or `CV_64F` .
- **order** – Order of multipliers
- **delta** – An optional array, subtracted from `src` before multiplication
- **scale** – An optional scaling

The function calculates the product of src and its transposition:

$$\texttt{dst} = \texttt{scale} (\texttt{src} - \texttt{delta}) (\texttt{src} - \texttt{delta})^T$$

if $\texttt{order} = 0$, and

$$\texttt{dst} = \texttt{scale} (\texttt{src} - \texttt{delta})^T (\texttt{src} - \texttt{delta})$$

otherwise.

Norm

double **cvNorm**(const CvArr* *arr1*, const CvArr* *arr2=NULL*, int *normType=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 the absolute or relative norm of `arr1` - `arr2` is calculated.
- **normType** – Type of norm, see the discussion
- **mask** – The optional operation mask

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

$$norm = \begin{cases} ||\texttt{arr1}||_C = \max_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV\_C} \\ ||\texttt{arr1}||_{L1} = \sum_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV\_L1} \\ ||\texttt{arr1}||_{L2} = \sqrt{\sum_I \texttt{arr1}(I)^2} & \text{if } \texttt{normType} = \texttt{CV\_L2} \end{cases}$$

or the absolute difference norm if `arr2` is not NULL:

$$norm = \begin{cases} ||\texttt{arr1} - \texttt{arr2}||_C = \max_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV\_C} \\ ||\texttt{arr1} - \texttt{arr2}||_{L1} = \sum_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV\_L1} \\ ||\texttt{arr1} - \texttt{arr2}||_{L2} = \sqrt{\sum_I (\texttt{arr1}(I) - \texttt{arr2}(I))^2} & \text{if } \texttt{normType} = \texttt{CV\_L2} \end{cases}$$

or the relative difference norm if `arr2` is not NULL and `(normType & CV_RELATIVE) != 0`:

$$norm = \begin{cases} \frac{||\texttt{arr1} - \texttt{arr2}||_C}{||\texttt{arr2}||_C} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_C} \\ \frac{||\texttt{arr1} - \texttt{arr2}||_{L1}}{||\texttt{arr2}||_{L1}} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_L1} \\ \frac{||\texttt{arr1} - \texttt{arr2}||_{L2}}{||\texttt{arr2}||_{L2}} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_L2} \end{cases}$$

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

Not

void **cvNot**(const CvArr* *src*, CvArr* *dst*)
Performs per-element bit-wise inversion of array elements.

Parameters

- **src** – The source array
- **dst** – The destination array

The function Not inverses every bit of every array element:

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

Or

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 the destination array to be changed

The function 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.

OrS

void **cvOrS**(const CvArr* *src*, CvScalar *value*, CvArr* *dst*, const CvArr* *mask=NULL*)
Calculates a per-element bit-wise disjunction of an array and a 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 the destination array to be changed

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

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

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). 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.

PerspectiveTransform

void **cvPerspectiveTransform**(const CvArr* *src*, CvArr* *dst*, const CvMat* *mat*)
Performs perspective matrix transformation of a vector array.

Parameters

- **src** – The source three-channel floating-point array
- **dst** – The destination three-channel floating-point array
- **mat** – 3×3 or 4×4 transformation matrix

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

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

PolarToCart

void **cvPolarToCart**(const CvArr* *magnitude*, const CvArr* *angle*, CvArr* *x*, CvArr* *y*, int *angleInDegrees=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 to be 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, mau be set to NULL if it is not needed

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

The function calculates either the x-coodinate, 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))
```

Pow

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

Raises every array element to a 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 raises every element of the input array to `p` :

$$\texttt{dst}[I] = \begin{cases} \texttt{src}(I)^p & \text{if } \texttt{p} \text{ is integer} \\ |\texttt{src}(I)^p| & \text{otherwise} \end{cases}$$

That is, for a 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 example, computing the 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.

Ptr?D

uchar* **cvPtr1D**(const CvArr* *arr*, int *idx0*, int* *type=NULL*)

uchar* **cvPtr2D**(const CvArr* *arr*, int *idx0*, int *idx1*, int* *type=NULL*)

uchar* **cvPtr3D**(const CvArr* *arr*, int *idx0*, int *idx1*, int *idx2*, int* *type=NULL*)

uchar* **cvPtrND**(const CvArr* *arr*, int* *idx*, int* *type=NULL*, int *createNode=1*, unsigned* *precalcHashval=NULL*)

Return pointer to a 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
- **createNode** – 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.
- **precalcHashval** – 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 return a pointer to a specific 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 (*Get* , *GetReal* , *Set* , *SetReal*) raise an error in case if the element index is out of range.

RNG

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

Initializes a random number generator state.

Parameters

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

The function initializes a random number generator and returns the state. The pointer to the state can be then passed to the *RandInt* , *RandReal* and *RandArr* functions. In the current implementation a multiply-with-carry generator is used.

RandArr

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

Fills an array with random numbers and updates the RNG state.

Parameters

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

The function fills the destination array with uniformly or normally distributed random numbers.

In the example 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);

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

/* generate values */
cvRandArr(&rng_state, values, CV_RAND_NORMAL,
           cvRealScalar(100), // average intensity
           cvRealScalar(30) // deviation of the intensity
          );

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

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

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

RandInt

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

Returns a 32-bit unsigned integer and updates RNG.

Parameters

- **rng** – RNG state initialized by `RandInit` and, optionally, customized by `RandSetRange` (though, the latter function does not affect the discussed function outcome)

The function returns a uniformly-distributed random 32-bit unsigned integer and updates the RNG state. It is similar to the rand() function from the C runtime library, but it always generates a 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 a modulo operation and floating-point numbers can be generated by scaling from 0 to 1 or any other specific range.

Here is the example from the previous function discussion rewritten using *RandInt* :

```
/* the input and the task is the same as in the previous sample. */
CvRNG rnggstate = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* ... - no arrays are allocated here */
CvSize size = cvGetSize(noisygscreen);
/* make a buffer for normally distributed numbers to reduce call overhead */
#define bufferSize 16
float normalValueBuffer[bufferSize];
CvMat normalValueMat = cvMat(bufferSize, 1, CVg32F, normalValueBuffer);
int valuesLeft = 0;

for(i = 0; i < pointCount; i++ )
{
    CvPoint pt;
    /* generate random point */
    pt.x = cvRandInt(&rnggstate )
    pt.y = cvRandInt(&rnggstate )

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

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

RandReal

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

Returns a floating-point random number and updates RNG.

Parameters

- **rng** – RNG state initialized by *RNG*

The function returns a uniformly-distributed random floating-point number between 0 and 1 (1 is not included).

Reduce

void **cvReduce** (const CvArr* *src*, CvArr* *dst*, int dim = *-1*, int *op=CV_REDUCE_SUM*)

Reduces a 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 reduced. 0 means that the matrix is reduced to a single row, 1 means that the matrix is reduced to a single column and -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 of the matrix's rows/columns.
  - **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
  - **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/columns.
  - **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/columns.

The function 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 the 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.

ReleaseData

void **cvReleaseData**(CvArr* *arr*)

Releases array data.

Parameters

- **arr** – Array header

The function releases the array data. In the case of *CvMat* or *CvMatND* it simply calls cvDecRefData(), that is the function can not deallocate external data. See also the note to *CreateData* .

ReleaseImage

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

Deallocates the image header and the image data.

Parameters

- **image** – Double pointer to the image header

This call is a shortened form of

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

ReleaseImageHeader

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

Deallocates an image header.

Parameters

- **image** – Double pointer to the image header

This call is an analogue of

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

but it does not use IPL functions by default (see the `CV_TURN_ON_IPL_COMPATIBILITY` macro).

ReleaseMat

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

Deallocates a matrix.

Parameters

- **mat** – Double pointer to the matrix

The function decrements the matrix data reference counter and deallocates matrix header. If the data reference counter is 0, it also deallocates the data.

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

ReleaseMatND

void **cvReleaseMatND** (CvMatND** *mat*)

Deallocates a multi-dimensional array.

Parameters

- **mat** – Double pointer to the array

The function decrements the array data reference counter and releases the array header. If the reference counter reaches 0, it also deallocates the data.

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

ReleaseSparseMat

void **cvReleaseSparseMat** (CvSparseMat** *mat*)

Deallocates sparse array.

Parameters

- **mat** – Double pointer to the array

The function releases the sparse array and clears the array pointer upon exit.

Repeat

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

Fill the destination array with repeated copies of the source array.

Parameters

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

The function fills the destination array with repeated copies of the source array:

```
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.

ResetImageROI

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

Resets the image ROI to include the entire image and releases the ROI structure.

Parameters

- **image** – A pointer to the image header

This produces a similar result to the following , but in addition it releases the ROI structure.

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

Reshape

CvMat* **cvReshape**(const CvArr* *arr*, CvMat* *header*, int *newCn*, int *newRows=0*)

Changes shape of matrix/image without copying data.

Parameters

- **arr** – Input array
- **header** – Output header to be filled
- **newCn** – New number of channels. 'newCn = 0' means that the number of channels remains unchanged.
- **newRows** – New number of rows. 'newRows = 0' means that the number of rows remains unchanged unless it needs to be changed according to `newCn` value.

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

The following example code creates one image buffer and two image headers, the first is for a 320x240x3 image and the second is for a 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 a 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);
```

ReshapeMatND

CvArr* **cvReshapeMatND**(const CvArr* *arr*, int *sizeofHeader*, CvArr* *header*, int *newCn*, int *newDims*, int* *newSizes*)

Changes the shape of a multi-dimensional array without copying the data.

```
#define cvReshapeND(arr, header, newCn, newDims, newSizes )   \
      cvReshapeMatND((arr), sizeof(*(header)), (header),         \
                      (newCn), (newDims), (newSizes))
```

param arr Input array

param sizeofHeader Size of output header to distinguish between IplImage, CvMat and CvMatND output headers

param header Output header to be filled

param newCn New number of channels. $\texttt{newCn} = 0$ means that the number of channels remains unchanged.

param newDims New number of dimensions. $\texttt{newDims} = 0$ means that the number of dimensions remains the same.

param newSizes Array of new dimension sizes. Only $\texttt{newDims} - 1$ values are used, because the total number of elements must remain the same. Thus, if $\texttt{newDims} = 1$, `newSizes` array is not used.

The function is an advanced version of *Reshape* 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 *Reshape* description rewritten using *ReshapeMatND* :

```
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 = (CvMat*)cvReshapeND(mat, &row_header, 0, 1, 0);
```

cvRound, cvFloor, cvCeil

int **cvRound**(double value) int cvFloor(double value) int cvCeil(double *value*)

Converts a floating-point number to an integer.

Parameters

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

The functions convert the input floating-point number to an integer using one of the rounding modes. `Round` returns the nearest integer value to the argument. `Floor` returns the maximum integer value that is not larger than the argument. `Ceil` 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 the absolute value of the argument is greater than 2^{31} , the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

ScaleAdd

void **cvScaleAdd**(const CvArr* *src1*, CvScalar *scale*, const CvArr* *src2*, CvArr* *dst*)
Calculates the sum of a scaled array and another array.

Parameters

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

The function calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale} \, \texttt{src1}(I) + \texttt{src2}(I)$$

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

Set

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

Parameters

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

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

$$\texttt{arr}(I) = \texttt{value} \quad \text{if} \quad \texttt{mask}(I) \neq 0$$

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

Set?D

void **cvSet1D**(CvArr* *arr*, int *idx0*, CvScalar *value*)

void **cvSet2D**(CvArr* *arr*, int *idx0*, int *idx1*, CvScalar *value*)

void **cvSet3D**(CvArr* *arr*, int *idx0*, int *idx1*, int *idx2*, CvScalar *value*)

void **cvSetND**(CvArr* *arr*, int* *idx*, CvScalar *value*)
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 assign the new value to a particular array element. In the case of a sparse array the functions create the node if it does not exist yet.

SetData

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

SetIdentity

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

Parameters

- **mat** – The matrix to initialize (not necesserily square)
- **value** – The value to assign to the diagonal elements

The function initializes a scaled identity matrix:

$$\texttt{arr}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

SetImageCOI

void **cvSetImageCOI** (IplImage* *image*, int *coi*)
Sets the channel of interest in an IplImage.

Parameters

- **image** – A pointer to the image header
- **coi** – The channel of interest. 0 - all channels are selected, 1 - first channel is selected, etc. Note that the channel indices become 1-based.

If the ROI is set to `NULL` and the coi is *not* 0, the ROI is allocated. Most OpenCV functions do *not* support the COI setting, so to process an individual image/matrix channel one may copy (via *Copy* or *Split*) the channel to a separate image/matrix, process it and then copy the result back (via *Copy* or *Merge*) if needed.

SetImageROI

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

Sets an image Region Of Interest (ROI) for a given rectangle.

Parameters

- **image** – A pointer to the image header
- **rect** – The ROI rectangle

If the original image ROI was `NULL` and the `rect` is not the whole image, the ROI structure is allocated.

Most OpenCV functions support the use of ROI and treat the image rectangle as a separate image. For example, all of the pixel coordinates are counted from the top-left (or bottom-left) corner of the ROI, not the original image.

SetReal?D

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

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

void **cvSetReal3D** (CvArr* *arr*, int *idx0*, int *idx1*, int *idx2*, double *value*)

void **cvSetRealND** (CvArr* *arr*, int* *idx*, double *value*)

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

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

SetZero

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

Clears the array.

```
#define cvZero cvSetZero
```

param arr Array to be cleared

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

Solve

int **cvSolve**(const CvArr* *src1*, const CvArr* *src2*, CvArr* *dst*, int *method=CV_LU*)
Solves a 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 chosen
  - **CV_SVD** Singular value decomposition (SVD) method
  - **CV_SVD_SYM** SVD method for a symmetric positively-defined matrix.

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

$$\texttt{dst} = argmin_X ||\texttt{src1}\,\texttt{X} - \texttt{src2}||$$

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.

SolveCubic

void **cvSolveCubic**(const CvArr* *coeffs*, CvArr* *roots*)
Finds the real roots of a cubic equation.

Parameters

- **coeffs** – The equation coefficients, an array of 3 or 4 elements
- **roots** – The output array of real roots which should have 3 elements

The function finds the real roots of a cubic equation:

If coeffs is a 4-element vector:

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or if coeffs is 3-element vector:

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

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.

Split

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.

Parameters

- **src** – Source array
- **dst0** – Destination channel 0

- **dst1** – Destination channel 1
- **dst2** – Destination channel 2
- **dst3** – Destination channel 3

The function 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, they all are extracted from the source array; if only a single destination channel of the first N is not NULL, this particular channel is extracted; otherwise an error is raised. The rest of the destination channels (beyond the first N) must always be NULL. For IplImage *Copy* with COI set can be also used to extract a single channel from the image.

Sqrt

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

Calculates the square root.

Parameters

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

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

Sub

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

Computes the 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 the destination array to be changed

The function 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). For types that have limited range this operation is saturating.

SubRS

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

Computes the difference between a scalar and an 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 the destination array to be changed

The function 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). For types that have limited range this operation is saturating.

SubS

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

Computes the difference between an array and a scalar.

Parameters

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

The function 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). For types that have limited range this operation is saturating.

Sum

CvScalar **cvSum** (const CvArr* *arr*)

Adds up array elements.

Parameters

- **arr** – The array

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

$$\sum_I \texttt{arr}(I)_c$$

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.

SVBkSb

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 (tranposed, perhaps)

- **V** – Right orthogonal matrix (tranposed, perhaps)
- **B** – The matrix to multiply the pseudo-inverse of the original matrix `A` by. This is an optional parameter. If it is omitted then it is assumed to be an identity matrix of an appropriate size (so that `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 *SVD*

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

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

where

$$W^{-1}_{(i,i)} = \begin{cases} 1/W_{(i,i)} & \text{if } W_{(i,i)} > \epsilon \sum_i W_{(i,i)} \\ 0 & \text{otherwise} \end{cases}$$

and ϵ is a small number that depends on the matrix data type.

This function together with *SVD* is used inside *Invert* and *Solve* , and the possible reason to use these (svd and bksb) "low-level" function, is to avoid allocation of temporary matrices inside the high-level counterparts (inv and solve).

SVD

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

Performs singular value decomposition of a real floating-point matrix.

Parameters

- **A** – Source $\texttt{M} \times \texttt{N}$ matrix
- **W** – Resulting singular value diagonal matrix ($\texttt{M} \times \texttt{N}$ or $\min(\texttt{M}, \texttt{N}) \times \min(\texttt{M}, \texttt{N})$) or $\min(\texttt{M},\texttt{N}) \times 1$ vector of the singular values
- **U** – Optional left orthogonal matrix, $\texttt{M} \times \min(\texttt{M}, \texttt{N})$ (when `CV_SVD_U_T` is not set), or $\min(\texttt{M},\texttt{N}) \times \texttt{M}$ (when `CV_SVD_U_T` is set), or $\texttt{M} \times \texttt{M}$ (regardless of `CV_SVD_U_T` flag).
- **V** – Optional right orthogonal matrix, $\texttt{N} \times \min(\texttt{M}, \texttt{N})$ (when `CV_SVD_V_T` is not set), or $\min(\texttt{M},\texttt{N}) \times \texttt{N}$ (when `CV_SVD_V_T` is set), or $\texttt{N} \times \texttt{N}$ (regardless of `CV_SVD_V_T` flag).
- **flags** – Operation flags; can be 0 or a combination of the following values:
  - **CV_SVD_MODIFY_A** enables modification of matrix `A` 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 decomposes matrix `A` into the product of a diagonal matrix and two

orthogonal matrices:

$$A = U \, W \, V^T$$

where W is a 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 V columns) in descending order.

An SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix `A` is a square, symmetric, and positively defined matrix, for example, when

  it is a covariance matrix.

  W in this case will be a vector/matrix

  of the eigenvalues, and

  $U = V$ will be a matrix of the eigenvectors.

- accurate solution of a poor-conditioned linear system.
- least-squares solution of an overdetermined linear system. This and the preceeding is done by using the *Solve* function with the `CV_SVD` method.
- accurate calculation of different matrix characteristics such as the matrix rank (the number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), and determinant (absolute value of the determinant is equal to the product of singular values).

Trace

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

Returns the trace of a matrix.

Parameters

- **mat** – The source matrix

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

$$tr(\texttt{mat}) = \sum_i \texttt{mat}(i,i)$$

Transform

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

Performs matrix transformation of every array element.

Parameters

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

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

$$dst(I) = transmat \cdot src(I) + shiftvec$$

That is, every element of an `N` -channel array `src` is considered as an `N` -element vector which is transformed using a $\texttt{M} \times \texttt{N}$ matrix `transmat` and shift vector `shiftvec` into an element of `M` -channel array `dst` . There is an option to embedd `shiftvec` into `transmat` . In this case `transmat` should be a $\texttt{M} \times (N+1)$ matrix and the rightmost 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 n dimensional point set, arbitrary linear color space transformation, shuffling the channels and so forth.

Transpose

void **cvTranspose**(const CvArr* *src*, CvArr* *dst*)
Transposes a matrix.

Parameters

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

The function transposes matrix `src1` :

$$\texttt{dst}(i,j) = \texttt{src}(j,i)$$

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

Xor

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 the destination array to be changed

The function 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.

XorS

void **cvXorS**(const CvArr* *src*, CvScalar *value*, CvArr* *dst*, const CvArr* *mask=NULL*)
Performs per-element bit-wise "exclusive or" operation on an array and a 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 the destination array to be changed

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

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

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). 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, negMask = 0x80000000;
cvXorS(&A, cvScalar(0, *(float*)&negMask, 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)
```

mGet

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

Returns 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 is a fast replacement for *GetReal2D* in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

mSet

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

Returns a specific element of a single-channel floating-point matrix.

Parameters

- **mat** – The matrix
- **row** – The zero-based index of row
- **col** – The zero-based index of column
- **value** – The new value of the matrix element

The function is a fast replacement for *SetReal2D* in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

1.3 Dynamic Structures

CvMemStorage

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 dynamicly 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; all blocks between `top` and the last block, not including `top` , are considered free and `top` itself is partly ocupied - `free_space` contains the number of free bytes left in the end of `top` .

A new memory buffer that may be allocated explicitly by *MemStorageAlloc* function or implicitly by higher-level functions, such as *SeqPush* , *GraphAddEdge* , 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 portion 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 are no more free blocks, a new block is allocated (or borrowed from the parent, see *CreateChildMemStorage*) 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 *SaveMemStoragePos* , restored via *RestoreMemStoragePos* , or reset via *ClearStorage* .

CvMemBlock

CvMemBlock

Memory storage block.

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

The structure *CvMemBlock* represents a single block of memory storage. The actual data in 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, there is normally no need to access the storage structure fields directly.

CvMemStoragePos

CvMemStoragePos

Memory storage position.

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

The structure described above stores the position of the stack top that can be saved via *SaveMemStoragePos* and restored via *RestoreMemStoragePos* .

CvSeq

CvSeq

Growable sequence of elements.

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

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

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. The base type for dense sequences is *CvSeq* and such sequences are used to represent growable 1d arrays - vectors, stacks, queues, and 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* as a base class and they are discussed later in more detail. They are sequences of nodes; each 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 and so forth.

The field `header_size` contains the actual size of the sequence header and should be greater than 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 sequences in the vertical direction, that is, the 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 a sparse sequence.

The field `flags` contains 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 rather element size stored in `elem_size` . If a sequence contains the numeric data for one of the *CvMat* type then the element type matches to the corresponding *CvMat* element type, e.g., `CV_32SC2` may be used for a sequence of 2D points, `CV_32FC1` for sequences of floating-point values, etc. A `CV_SEQ_ELTYPE(seq_header_ptr)` macro retrieves the type of sequence elements. Processing functions that work with numerical sequences check that `elem_size` is equal to that calculated from the type element size. Besides *CvMat* compatible types, there are few extra element types defined in the `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 sequence:

Standard Kinds of Sequences

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

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

/* sparse sequence (or set) subtypes */
#define CV_SEQ_KIND_GRAPH       (3 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_SUBDIV2D    (4 << CV_SEQ_ELTYPE_BITS)
```

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. The definition of the building blocks of sequences can be found below.

CvSeqBlock

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 `startIndex` 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 `startIndex = 2` , then pairs `(startIndex, count)` for the sequence blocks are (2,3), (5, 5), and (10, 2) correspondingly. The parameter `startIndex` of the first block is usually `0` unless some elements have been inserted at the beginning of the sequence.

CvSlice

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 a `CvSlice slice` parameter that is often set to the whole sequence (CV _ WHOLE _ SEQ) by default. Either of the `startIndex` and `endIndex` may be negative or exceed the sequence length, `startIndex` is inclusive, and `endIndex` is an 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 at the beginning of the sequence. For example, `cvSlice(-2, 3)` in the case of a 10-element sequence will select a 5-element slice, containing the 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, *SliceLength* is called to determine the length of the slice, then, `startIndex` of the slice is normalized similarly to the argument of *GetSeqElem* (i.e., negative indices are allowed). The actual slice to process starts at the normalized `startIndex` and lasts *SliceLength* elements (again, assuming the sequence is a circular structure).

If a function does not accept a slice argument, but you want to process only a part of the sequence, the sub-sequence may be extracted using the *SeqSlice* function, or stored into a continuous buffer with *CvtSeqToArray* (optionally, followed by *MakeSeqHeaderForArray*).

CvSet

CvSet

Collection of nodes.

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

#define CV_SET_FIELDS()     \
    CV_SEQUENCE_FIELDS()    /* inherits from [#CvSeq 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 the `free_elems` field, which is a list of free nodes, to it. Every set node, whether free or not, is an element of the underlying sequence. While there are no restrictions on elements of dense sequences, the set (and derived structures) elements must start with an integer field and be able to fit CvSetElem structure, because these two fields (an integer followed by a pointer) are required for the organization of a node set with the list of free nodes. If a node is free, the `flags` field is negative (the most-significant bit, or MSB, of the field is set), and the `next_free` points to the next free node (the first free node is referenced by the `free_elems` field of *CvSet*). And if a node is occupied, the `flags` field is positive and contains the node index that may be retrieved using the (`set_elem->flags & CV_SET_ELEM_IDX_MASK`) expressions, 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 then updated. 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, the `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*), and planar subdivisions *CvSubdiv2D* .

CvGraph

CvGraph

Oriented or unoriented 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.

The graph structure inherits from *CvSet* - which describes common graph properties and the graph vertices, and contains another set as a member - which 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, which simplify extension and customization of the structures. While the vertex and edge structures do not inherit from *CvSetElem* explicitly, they satisfy both conditions of the set elements: having an integer field in the beginning and fitting within the 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 *CreateGraphScanner* et al.), so it is better not to use them directly.

The graph is represented as a set of edges each of which has a list of incident edges. The incidence lists for different vertices are interleaved to avoid information duplication as much as posssible.

The graph may be oriented or unoriented. In the latter case there is no distiction between the 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 $A \rightarrow B$ and $B \rightarrow A$ edges.

CvGraphScanner

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.

cvmacro Helper macro for a tree node type declaration.

The macro `CV_TREE_NODE_FIELDS()` is used to declare structures that can be organized into hierarchical strucutures (trees), such as *CvSeq* - the basic type for all dynamic structures. The trees created with nodes declared using this macro can be processed using the functions described below in this section.

CvTreeNodeIterator

CvTreeNodeIterator

Opens existing or creates new file storage.

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

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

The structure *CvTreeNodeIterator* is used to traverse trees. Each tree node should start with the certain fields which are defined by `CV_TREE_NODE_FIELDS(...)` macro. In C++ terms, each tree node should be a structure “derived” from

```
struct _BaseTreeNode
{
    CV_TREE_NODE_FIELDS(_BaseTreeNode);
}
```

`CvSeq` , `CvSet` , `CvGraph` and other dynamic structures derived from `CvSeq` comply with the requirement.

ClearGraph

void **cvClearGraph**(CvGraph* *graph*)

Clears a graph.

Parameters

- **graph** – Graph

The function removes all vertices and edges from a graph. The function has O(1) time complexity.

ClearMemStorage

void **cvClearMemStorage** (CvMemStorage* *storage*)

Clears memory storage.

Parameters

- **storage** – Memory storage

The function 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.

ClearSeq

void **cvClearSeq** (CvSeq* *seq*)

Clears a sequence.

Parameters

- **seq** – Sequence

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

ClearSet

void **cvClearSet** (CvSet* *setHeader*)

Clears a set.

Parameters

- **setHeader** – Cleared set

The function removes all elements from set. It has O(1) time complexity.

CloneGraph

CvGraph* **cvCloneGraph** (const CvGraph* *graph*, CvMemStorage* *storage*)

Clones a graph.

Parameters

- **graph** – The graph to copy
- **storage** – Container for the copy

The function creates a full copy of the specified graph. If the graph vertices or edges have pointers to some external data, it can 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.

CloneSeq

CvSeq* **cvCloneSeq**(const CvSeq* *seq*, CvMemStorage* *storage=NULL*)
Creates a copy of a sequence.

Parameters

- **seq** – Sequence
- **storage** – The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

The function makes a complete copy of the input sequence and returns it.

The call

```
cvCloneSeq( seq, storage )
```

is equivalent to

```
cvSeqSlice( seq, CV_WHOLE_SEQ, storage, 1 )
```

CreateChildMemStorage

CvMemStorage* **cvCreateChildMemStorage**(CvMemStorage* *parent*)
Creates child memory storage.

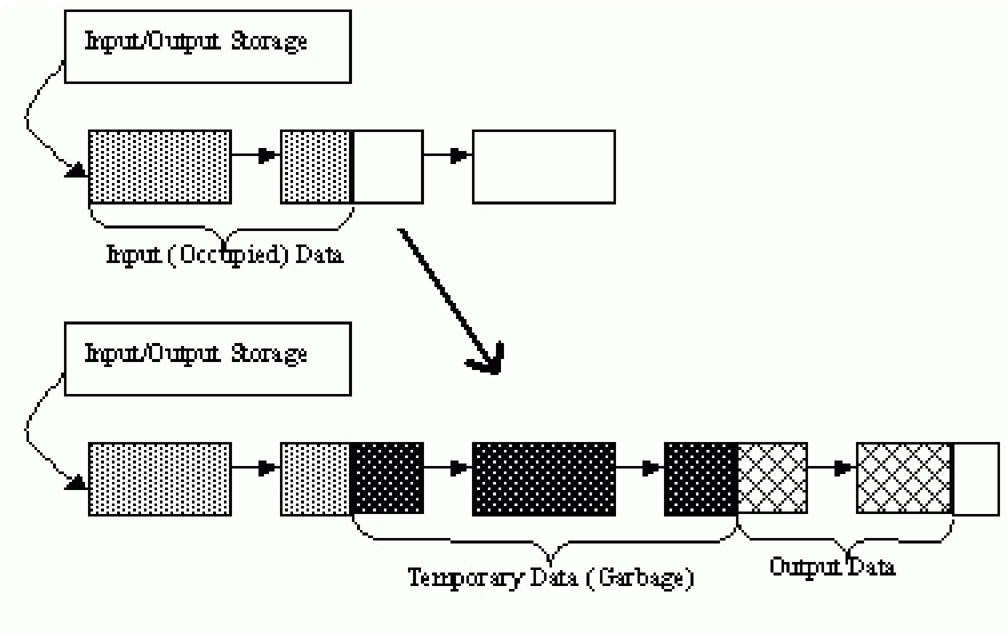
Parameters

- **parent** – Parent memory storage

The function 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, child storage is the same as simple storage.

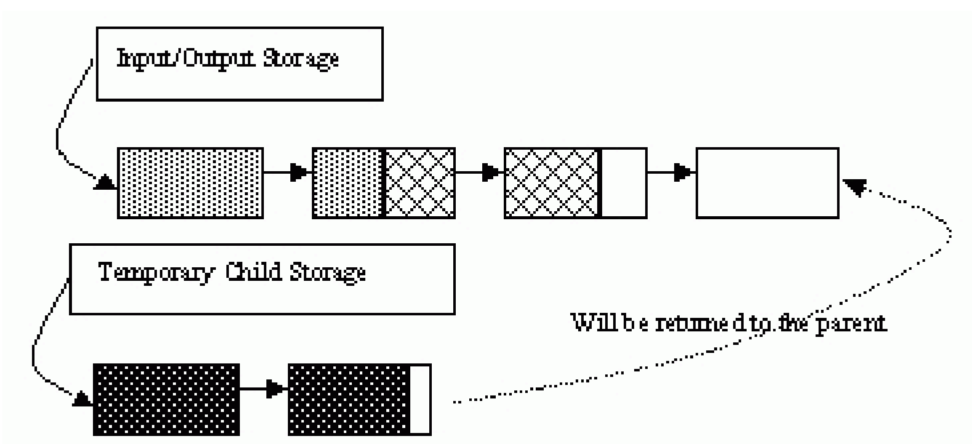
Child storage is useful in the following situation. Imagine that the user needs to process dynamic data residing in a given storage area and put the result back to that same storage area. With the simplest approach, when temporary data is resided in the same storage area as the input and output data, the storage area will look as follows 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 at the beginning of processing, writes temporary data there, and releases the child storage at the end, no garbage will appear in the source/destination storage:

Dynamic data processing using a child storage



CreateGraph

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

Creates an empty graph.

Parameters

- **graph_flags** – Type of the created graph. Usually, it is either `CV_SEQ_KIND_GRAPH` for generic unoriented 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_size** – Graph vertex size; the custom vertex structure must start with *CvGraphVtx* (use `CV_GRAPH_VERTEX_FIELDS()`)
- **edge_size** – Graph edge size; the custom edge structure must start with *CvGraphEdge* (use `CV_GRAPH_EDGE_FIELDS()`)
- **storage** – The graph container

The function creates an empty graph and returns a pointer to it.

CreateGraphScanner

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 of interest to the user (where *NextGraphItem* function returns control to the user) It can be `CV_GRAPH_ALL_ITEMS` (all events are of interest) or a 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 conecting 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 graph traversal)
  – **CV_GRAPH_ANY_EDGE** stop at 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 a `CV_GRAPH_NEW_TREE` event. For unoriented 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 creates a structure for depth-first graph traversal/search. The initialized structure is used in the *NextGraphItem* function - the incremental traversal procedure.

CreateMemStorage

CvMemStorage* **cvCreateMemStorage** (int *blockSize=0*)
Creates memory storage.

Parameters

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

The function creates an empty memory storage. See *CvMemStorage* description.

CreateSeq

CvSeq* **cvCreateSeq** (int *seqFlags*, int *headerSize*, int *elemSize*, CvMemStorage* *storage*)
Creates a sequence.

Parameters

- **seqFlags** – 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.
- **headerSize** – Size of the sequence header; must be greater than or equal to `sizeof(CvSeq)` . If a specific type or its extension is indicated, this type must fit the base type header.
- **elemSize** – 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 `elemSize` must be equal to `sizeof(CvPoint)` .
- **storage** – Sequence location

The function 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` , `elemSize` , `headerSize` , and `storage` to passed values, sets `delta_elems` to the default value (that may be reassigned using the *SetSeqBlockSize* function), and clears other header fields, including the space following the first `sizeof(CvSeq)` bytes.

CreateSet

CvSet* **cvCreateSet** (int *set_flags*, int *header_size*, int *elem_size*, CvMemStorage* *storage*)
Creates an 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 creates an empty set with a specified header size and element size, and returns the pointer to the set. This function is just a thin layer on top of *CreateSeq* .

CvtSeqToArray

void* **cvCvtSeqToArray**(const CvSeq* *seq*, void* *elements*, CvSlice *slice=CV_WHOLE_SEQ*)
Copies a 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 portion to copy to the array

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

EndWriteSeq

CvSeq* **cvEndWriteSeq**(CvSeqWriter* *writer*)
Finishes the process of writing a sequence.

Parameters

- **writer** – Writer state

The function 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 memory storage. After that, the sequence can be read and modified safely. See *cvStartWriteSeq* and *cvStartAppendToSeq*

FindGraphEdge

CvGraphEdge* **cvFindGraphEdge**(const CvGraph* *graph*, int *start_idx*, int *end_idx*)
Finds an edge in a graph.

```
#define cvGraphFindEdge cvFindGraphEdge
```

param graph Graph

param start_idx Index of the starting vertex of the edge

param end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function finds the graph edge connecting two specified vertices and returns a pointer to it or NULL if the edge does not exist.

FindGraphEdgeByPtr

CvGraphEdge* **cvFindGraphEdgeByPtr**(const CvGraph* *graph*, const CvGraphVtx* *startVtx*, const CvGraphVtx* *endVtx*)
Finds an edge in a graph by using its pointer.

```
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

param graph Graph

param startVtx Pointer to the starting vertex of the edge

param endVtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

FlushSeqWriter

void **cvFlushSeqWriter**(CvSeqWriter* *writer*)

Updates sequence headers from the writer.

Parameters

- **writer** – Writer state

The function 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 at any time. If an algorithm requires frequent flushes, consider using *SeqPush* instead.

GetGraphVtx

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

Finds a graph vertex by using its index.

Parameters

- **graph** – Graph
- **vtx_idx** – Index of the vertex

The function finds the graph vertex by using its index and returns the pointer to it or NULL if the vertex does not belong to the graph.

GetSeqElem

char* **cvGetSeqElem**(const CvSeq* *seq*, int *index*)

Returns a pointer to a sequence element according to its index.

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

param seq Sequence

param index Index of element

The function 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 *GetSeqElem* call. The function has O(1) time complexity assuming that the number of blocks is much smaller than the number of elements.

GetSeqReaderPos

int **cvGetSeqReaderPos**(CvSeqReader* *reader*)
Returns the current reader position.

Parameters

- **reader** – Reader state

The function returns the current reader position (within 0 ... `reader->seq->total` - 1).

GetSetElem

CvSetElem* **cvGetSetElem**(const CvSet* *setHeader*, int *index*)
Finds a set element by its index.

Parameters

- **setHeader** – Set
- **index** – Index of the set element within a sequence

The function finds a set element by its 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 *GetSeqElem* to locate the node.

GraphAddEdge

int **cvGraphAddEdge**(CvGraph* *graph*, int *start_idx*, int *end_idx*, const CvGraphEdge* *edge=NULL*, CvGraphEdge** *inserted_edge=NULL*)
Adds an edge to a 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 an unoriented 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 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.

GraphAddEdgeByPtr

int **cvGraphAddEdgeByPtr**(CvGraph* *graph*, CvGraphVtx* *start_vtx*, CvGraphVtx* *end_vtx*, const CvGraphEdge* *edge=NULL*, CvGraphEdge** *inserted_edge=NULL*)
Adds an edge to a graph by using its pointer.

Parameters

- **graph** – Graph
- **start_vtx** – Pointer to the starting vertex of the edge

- **end_vtx** – Pointer to the ending vertex of the edge. For an unoriented 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 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.

GraphAddVtx

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

Adds a vertex to a 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 here.

The function adds a vertex to the graph and returns the vertex index.

GraphEdgeIdx

int **cvGraphEdgeIdx**(CvGraph* *graph*, CvGraphEdge* *edge*)

Returns the index of a graph edge.

Parameters

- **graph** – Graph
- **edge** – Pointer to the graph edge

The function returns the index of a graph edge.

GraphRemoveEdge

void **cvGraphRemoveEdge**(CvGraph* *graph*, int *start_idx*, int *end_idx*)

Removes an edge from a 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 an unoriented graph, the order of the vertex parameters does not matter.

The function removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

GraphRemoveEdgeByPtr

void **cvGraphRemoveEdgeByPtr** (CvGraph* *graph*, CvGraphVtx* *start_vtx*, CvGraphVtx* *end_vtx*)

Removes an edge from a graph by using its pointer.

Parameters

- **graph** – Graph
- **start_vtx** – Pointer to the starting vertex of the edge
- **end_vtx** – Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

GraphRemoveVtx

int **cvGraphRemoveVtx** (CvGraph* *graph*, int *index*)

Removes a vertex from a graph.

Parameters

- **graph** – Graph
- **vtx_idx** – Index of the removed vertex

The function removes a vertex from a 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 the number of edges deleted, or -1 if the vertex does not belong to the graph.

GraphRemoveVtxByPtr

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

Removes a vertex from a graph by using its pointer.

Parameters

- **graph** – Graph
- **vtx** – Pointer to the removed vertex

The function removes a vertex from the graph by using its pointer 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 the number of edges deleted, or -1 if the vertex does not belong to the graph.

GraphVtxDegree

int **cvGraphVtxDegree** (const CvGraph* *graph*, int *vtxIdx*)

Counts the number of edges indicent to the vertex.

Parameters

- **graph** – Graph
- **vtxIdx** – Index of the graph vertex

The function 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` .

GraphVtxDegreeByPtr

int **cvGraphVtxDegreeByPtr** (const CvGraph* *graph*, const CvGraphVtx* *vtx*)

Finds an edge in a graph.

Parameters

- **graph** – Graph
- **vtx** – Pointer to the graph vertex

The function returns the number of edges incident to the specified vertex, both incoming and outcoming.

GraphVtxIdx

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

Returns the index of a graph vertex.

Parameters

- **graph** – Graph
- **vtx** – Pointer to the graph vertex

The function returns the index of a graph vertex.

InitTreeNodeIterator

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

Initializes the 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, and so forth.

The function initializes the tree iterator. The tree is traversed in depth-first order.

InsertNodeIntoTree

void **cvInsertNodeIntoTree**(void* *node*, void* *parent*, void* *frame*)

Adds a new node to a 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, the `v_prev` field of `node` is set to NULL rather than `parent` .

The function adds another node into tree. The function does not allocate any memory, it can only modify links of the tree nodes.

MakeSeqHeaderForArray

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

Constructs a sequence header for an 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
- **elem_size** – Size of the sequence elements
- **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 initializes a sequence header for an 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.

MemStorageAlloc

void* **cvMemStorageAlloc**(CvMemStorage* *storage*, size_t *size*)

Allocates a memory buffer in a storage block.

Parameters

- **storage** – Memory storage
- **size** – Buffer size

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

MemStorageAllocString

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

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

param storage Memory storage

param ptr The string

param len Length of the string (not counting the ending `NUL`) . If the parameter is negative, the function computes the length.

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

NextGraphItem

int **cvNextGraphItem**(CvGraphScanner* *scanner*)
Executes one or more steps of the graph traversal procedure.

Parameters

- **scanner** – Graph traversal state. It is updated by this function.

The function traverses through the graph until an event of interest to the user (that is, an event, specified in the `mask` in the *CreateGraphScanner* call) is met or the traversal is completed. In the first case, it returns one of the events listed in the description of the `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` , `CV_GRAPH_BACKTRACKING` , or `CV_GRAPH_NEW_TREE` , the currently observed vertex is stored in `scanner-:math:`>`vtx` . And if the event is edge-related, the edge itself is stored at `scanner-:math:`>`edge` , the previously visited vertex - at `scanner-:math:`>`vtx` and the other ending vertex of the edge - at `scanner-:math:`>`dst` .

NextTreeNode

void* **cvNextTreeNode**(CvTreeNodeIterator* *tree_iterator*)
Returns the currently observed node and moves the iterator toward the next node.

Parameters

- **tree_iterator** – Tree iterator initialized by the function

The function returns the currently observed node and then updates the iterator - moving it toward the next node. In other words, the function behavior is similar to the `*p++` expression on a typical C pointer or C++ collection iterator. The function returns NULL if there are no more nodes.

PrevTreeNode

void* **cvPrevTreeNode**(CvTreeNodeIterator* *tree_iterator*)
Returns the currently observed node and moves the iterator toward the previous node.

Parameters

- **tree_iterator** – Tree iterator initialized by the function

The function returns the currently observed node and then updates the iterator - moving it toward the previous node. In other words, the function behavior is similar to the `*p--` expression on a typical C pointer or C++ collection iterator. The function returns NULL if there are no more nodes.

ReleaseGraphScanner

void **cvReleaseGraphScanner**(CvGraphScanner** *scanner*)
Completes the graph traversal procedure.

Parameters

- **scanner** – Double pointer to graph traverser

The function completes the graph traversal procedure and releases the traverser state.

ReleaseMemStorage

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

Parameters

- **storage** – Pointer to the released storage

The function 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 child storage associated with a given parent storage block must be released before the parent storage block is released.

RestoreMemStoragePos

void **cvRestoreMemStoragePos**(CvMemStorage* *storage*, CvMemStoragePos* *pos*)
Restores memory storage position.

Parameters

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

The function 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 an occupied portion of a storage block.

SaveMemStoragePos

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

SeqElemIdx

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

Returns the index of a specific 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 returns the index of a sequence element or a negative number if the element is not found.

SeqInsert

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

Inserts an element in the middle of a sequence.

Parameters

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

The function 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 a pointer to the inserted element.

SeqInsertSlice

void **cvSeqInsertSlice**(CvSeq* *seq*, int *beforeIndex*, const CvArr* *fromArr*)

Inserts an array in the middle of a sequence.

Parameters

- **seq** – Sequence
- **beforeIndex** – Index before which the array is inserted
- **fromArr** – The array to take elements from

The function inserts all `fromArr` array elements at the specified position of the sequence. The array `fromArr` can be a matrix or another sequence.

SeqInvert

void **cvSeqInvert**(CvSeq* *seq*)

Reverses the order of sequence elements.

Parameters

- **seq** – Sequence

The function reverses the sequence in-place - the first element becomes the last one, the last element becomes the first one and so forth.

SeqPop

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

Removes an element from the end of a sequence.

Parameters

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

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

SeqPopFront

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

Removes an element from the beginning of a sequence.

Parameters

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

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

SeqPopMulti

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

Removes several elements from either end of a sequence.

Parameters

- **seq** – Sequence
- **elements** – Removed elements
- **count** – Number of elements to pop
- **in_front** – The flags specifying which end of the modified sequence.
  - **CV_BACK** the elements are added to the end of the sequence
  - **CV_FRONT** the elements are added to the beginning of the sequence

The function 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.

SeqPush

char* **cvSeqPush** (CvSeq* *seq*, void* *element=NULL*)
Adds an element to the end of a sequence.

Parameters

- **seq** – Sequence
- **element** – Added element

The function adds an element to the end of a sequence and returns a 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( "
}

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

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

SeqPushFront

char* **cvSeqPushFront** (CvSeq* *seq*, void* *element=NULL*)
Adds an element to the beginning of a sequence.

Parameters

- **seq** – Sequence
- **element** – Added element

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

SeqPushMulti

void **cvSeqPushMulti** (CvSeq* *seq*, void* *elements*, int *count*, int *in_front=0*)
Pushes several elements to either end of a sequence.

Parameters

- **seq** – Sequence
- **elements** – Added elements

- **count** – Number of elements to push
- **in_front** – The flags specifying which end of the modified sequence.
  - **CV_BACK** the elements are added to the end of the sequence
  - **CV_FRONT** the elements are added to the beginning of the sequence

The function adds several elements to either end of a 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.

SeqRemove

void **cvSeqRemove** (CvSeq* *seq*, int *index*)

Removes an element from the middle of a sequence.

Parameters

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

The function 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 special 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.

SeqRemoveSlice

void **cvSeqRemoveSlice** (CvSeq* *seq*, CvSlice *slice*)

Removes a sequence slice.

Parameters

- **seq** – Sequence
- **slice** – The part of the sequence to remove

The function removes a slice from the sequence.

SeqSearch

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

Searches for an element in a sequence.

Parameters

- **seq** – The sequence
- **elem** – The element to look for
- **func** – The comparison function that returns negative, zero or positive value depending on the relationships among the elements (see also *SeqSort*)
- **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 compasion function; helps to avoid global variables in some cases

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

The function searches for the element in the sequence. If the sequence is sorted, a binary O(log(N)) search is used; otherwise, a simple linear search is used. If the element is not found, the function returns a NULL pointer and the index is set to the number of sequence elements if a linear search is used, or to the smallest index `i, seq(i)>elem` .

SeqSlice

CvSeq* **cvSeqSlice**(const CvSeq* *seq*, CvSlice *slice*, CvMemStorage* *storage=NULL*, int *copy_data=0*)
Makes a separate header for a sequence slice.

Parameters

- **seq** – Sequence
- **slice** – The part of the sequence to be extracted
- **storage** – The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block 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 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 its 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.

SeqSort

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);
```

param seq The sequence to sort

param func The comparison function that returns a negative, zero, or positive value depending on the relationships among the elements (see the above declaration and the example below) - a similar function is used by `qsort` from C runline except that in the latter, `userdata` is not used

param userdata The user parameter passed to the compasion function; helps to avoid global variables in some cases

The function sorts the sequence in-place using the specified criteria. Below is an example of using this function:

```
/* 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()
    pt.y = rand()
    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( "(
}

cvReleaseMemStorage( &storage );
```

SetAdd

int **cvSetAdd**(CvSet* *setHeader*, CvSetElem* *elem=NULL*, CvSetElem** *inserted_elem=NULL*)

Occupies a node in the set.

Parameters

- **setHeader** – Set
- **elem** – Optional input argument, an 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 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 the `flags` field of the node. The function has O(1) complexity; however, there exists a faster function for allocating set nodes (see *SetNew*).

SetNew

CvSetElem* **cvSetNew**(CvSet* *setHeader*)

Adds an element to a set (fast variant).

Parameters

- **setHeader** – Set

The function is an inline lightweight variant of *SetAdd* . It occupies a new node and returns a pointer to it rather than an index.

SetRemove

void **cvSetRemove** (CvSet* *setHeader*, int *index*)

Removes an element from a set.

Parameters

- **setHeader** – Set
- **index** – Index of the removed element

The function 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, *SetRemoveByPtr* provides a quicker way to remove a set element if it is located already.

SetRemoveByPtr

void **cvSetRemoveByPtr** (CvSet* *setHeader*, void* *elem*)

Removes a set element based on its pointer.

Parameters

- **setHeader** – Set
- **elem** – Removed element

The function is an inline lightweight variant of *SetRemove* that requires an element pointer. The function does not check whether the node is occupied or not - the user should take care of that.

SetSeqBlockSize

void **cvSetSeqBlockSize** (CvSeq* *seq*, int *deltaElems*)

Sets up sequence block size.

Parameters

- **seq** – Sequence
- **deltaElems** – Desirable sequence block size for elements

The function affects memory allocation granularity. When the free space in the sequence buffers has run out, the function allocates the space for `deltaElems` 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 block is wasted. When the sequence is created, the parameter `deltaElems` is set to the default value of about 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 memory storage constraints.

SetSeqReaderPos

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

Moves the reader to the 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 moves the read position to an absolute position or relative to the current position.

StartAppendToSeq

void **cvStartAppendToSeq**(CvSeq* *seq*, CvSeqWriter* *writer*)
Initializes the process of writing data to a sequence.

Parameters

- **seq** – Pointer to the sequence
- **writer** – Writer state; initialized by the function

The function initializes the process of writing data to a sequence. Written elements are added to the end of the sequence by using the `CV_WRITE_SEQ_ELEM( written_elem, writer )` macro. Note that during the writing process, other operations on the sequence may yield an incorrect result or even corrupt the sequence (see description of *FlushSeqWriter* , which helps to avoid some of these problems).

StartReadSeq

void **cvStartReadSeq**(const CvSeq* *seq*, CvSeqReader* *reader*, int *reverse=0*)
Initializes the process of sequential reading from a 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 initializes the reader state. After that, all the sequence elements from the first one down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM( read_elem, reader )` in the case of forward reading and by using `CV_REV_READ_SEQ_ELEM( read_elem, reader )` in the case of reverse 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 the 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()
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("
}
cvEndWriteSeq( &writer );
```

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

cvReleaseStorage( &storage );
```

StartWriteSeq

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

Creates a new sequence and initializes a 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 within the base type header.
- **elem_size** – Size of the sequence elements in bytes; must be consistent with the sequence type. For example, if a 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 is a combination of *CreateSeq* and *StartAppendToSeq* . The pointer to the created sequence is stored at `writer->seq` and is also returned by the *EndWriteSeq* function that should be called at the end.

TreeToNodeSeq

CvSeq* **cvTreeToNodeSeq**(const void* *first*, int *header_size*, CvMemStorage* *storage*)

Gathers all node pointers to a single sequence.

Parameters

- **first** – The initial tree node
- **header_size** – Header size of the created sequence (sizeof(CvSeq) is the most frequently used value)
- **storage** – Container for the sequence

The function puts pointers of all nodes reacheable from `first` into a single sequence. The pointers are written sequentially in the depth-first order.

1.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with `CV_RGB` macro or the `cvScalar()` function) for color images and brightness for grayscale images. For color images the order channel is normally *Blue, Green, Red* , this is what `imshow()` , `imread()` and `imwrite()` expect , so if you form a color using `cvScalar()` , it should look like:

$$\texttt{cvScalar}(blue\_component, green\_component, red\_component[, alpha\_component])$$

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using `cvtColor()` .

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}(x * 2^{-shift}, y * 2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

Circle

void **cvCircle**(CvArr* *img*, CvPoint *center*, int *radius*, CvScalar *color*, int *thickness=1*, int *lineType=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 this indicates that a filled circle is to be drawn
- **lineType** – Type of the circle boundary, see *Line* description
- **shift** – Number of fractional bits in the center coordinates and radius value

The function draws a simple or filled circle with a given center and radius.

ClipLine

int **cvClipLine**(CvSize *imgSize*, CvPoint* *pt1*, CvPoint* *pt2*)

Clips the line against the image rectangle.

Parameters

- **imgSize** – 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 calculates a part of the line segment which is entirely within the image. It returns 0 if the line segment is completely outside the image and 1 otherwise.

DrawContours

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

Draws contour outlines or interiors in an image.

Parameters

- **img** – Image where the contours are to be drawn. As with 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 following it on the same level are drawn. If 2, all contours following and all contours one level below the contours are drawn, and so forth. If the value is negative, the function does not draw the contours following after `contour` but draws the child contours of `contour` up to the $|\texttt{max\_level}| - 1$ level.
- **thickness** – Thickness of lines the contours are drawn with. If it is negative (For example, =CV _ FILLED), the contour interiors are drawn.
- **lineType** – Type of the contour segments, see *Line* description

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$.

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);
    }
}
```

Ellipse

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

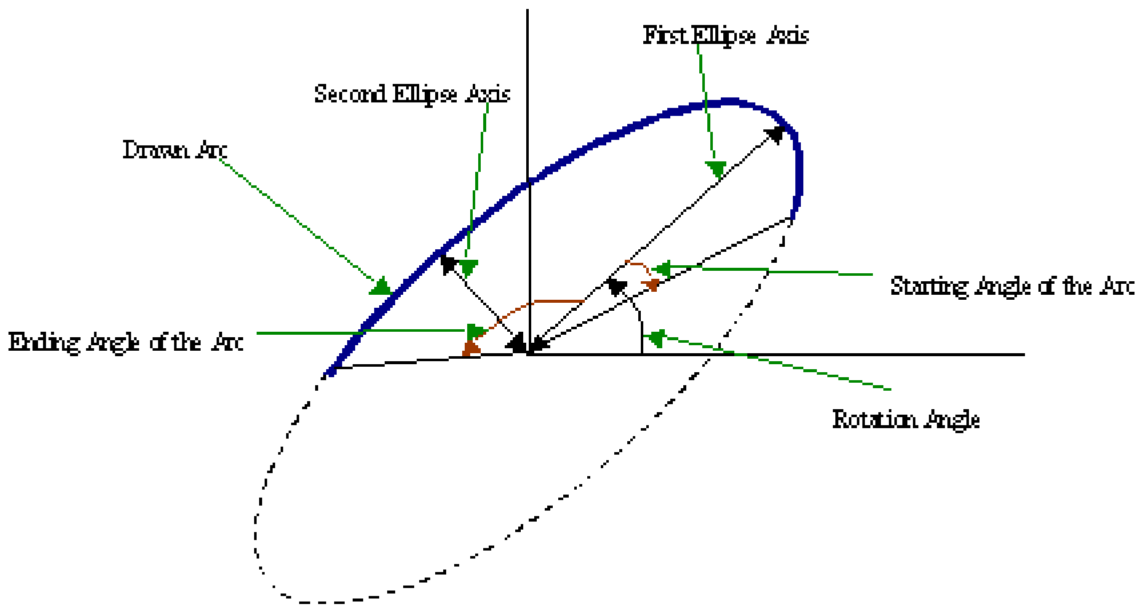
Draws a simple or thick elliptic arc or an fills ellipse sector.

Parameters

- **img** – The 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 outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn
- **lineType** – Type of the ellipse boundary, see *Line* description
- **shift** – Number of fractional bits in the center coordinates and axes' values

The function draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by the 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



EllipseBox

void **cvEllipseBox** (CvArr* *img*, CvBox2D *box*, CvScalar *color*, int *thickness=1*, int *lineType=8*, int *shift=0*)

Draws a simple or thick elliptic arc or fills an ellipse sector.

Parameters

- **img** – Image
- **box** – The enclosing box of the ellipse drawn
- **thickness** – Thickness of the ellipse boundary
- **lineType** – Type of the ellipse boundary, see *Line* description
- **shift** – Number of fractional bits in the box vertex coordinates

The function 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 *CamShift* and *FitEllipse* do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

FillConvexPoly

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

Fills a convex polygon.

Parameters

- **img** – Image
- **pts** – Array of pointers to a single polygon
- **npts** – Polygon vertex counter
- **color** – Polygon color

- **lineType** – Type of the polygon boundaries, see *Line* description
- **shift** – Number of fractional bits in the vertex coordinates

The function fills a convex polygon's interior. This function is much faster than the function `cvFillPoly` and can fill not only convex polygons but any monotonic polygon, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most.

FillPoly

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

Fills a polygon's 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
- **lineType** – Type of the polygon boundaries, see *Line* description
- **shift** – Number of fractional bits in the vertex coordinates

The function fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, and so forth.

GetTextSize

void **cvGetTextSize** (const char* *textString*, const CvFont* *font*, CvSize* *textSize*, int* *baseline*)

Retrieves the width and height of a text string.

Parameters

- **font** – Pointer to the font structure
- **textString** – Input string
- **textSize** – 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 relative to the bottom-most text point

The function calculates the dimensions of a rectangle to enclose a text string when a specified font is used.

InitFont

void **cvInitFont** (CvFont* *font*, int *fontFace*, double *hscale*, double *vscale*, double *shear=0*, int *thickness=1*, int *lineType=8*)

Initializes font structure.

Parameters

- **font** – Pointer to the font structure initialized by the function

- **fontFace** – 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 composited from one of the values above and an optional `CV_FONT_ITALIC` flag, which indicates 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. A zero value means a non-italic font, `1.0f` means about a 45 degree slope, etc.
- **thickness** – Thickness of the text strokes
- **lineType** – Type of the strokes, see *Line* description

The function initializes the font structure that can be passed to text rendering functions.

InitLineIterator

int **cvInitLineIterator** (const CvArr* *image*, CvPoint *pt1*, CvPoint *pt2*, CvLineIterator* *line_iterator*, int *connectivity=8*, int *left_to_right=0*)

Initializes the 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** – If ($\texttt{left\_to\_right} = 0$) then the line is scanned in the specified order, from `pt1` to `pt2` . If ($\texttt{left\_to\_right} \ne 0$) the line is scanned from left-most point to right-most.

The function initializes the line iterator and returns the number of pixels between the 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 a 4-connected or 8-connected Bresenham algorithm.

Example: Using line iterator to calculate the 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("(
        }
    }
    return cvScalar( blue_sum, green_sum, red_sum );
}
```

Line

void **cvLine** (CvArr* *img*, CvPoint *pt1*, CvPoint *pt2*, CvScalar *color*, int *thickness=1*, int *lineType=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
- **lineType** – Type of the line:
  - **8** (or omitted) 8-connected line.
  - **4** 4-connected line.
  - **CV_AA** antialiased line.
- **shift** – Number of fractional bits in the point coordinates

The function 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 )`.

PolyLine

void **cvPolyLine**(CvArr* *img*, CvPoint** *pts*, int* *npts*, int *contours*, int *is_closed*, CvScalar *color*, int *thickness=1*, int *lineType=8*, int *shift=0*)

Draws simple or thick polygons.

Parameters

- **pts** – Array of pointers to polygons
- **npts** – Array of polygon vertex counters
- **contours** – Number of contours that bind the filled region
- **img** – Image
- **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
- **lineType** – Type of the line segments, see *Line* description
- **shift** – Number of fractional bits in the vertex coordinates

The function draws single or multiple polygonal curves.

PutText

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

Draws a 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 renders the text in the image with the specified font and color. The printed text is clipped by the ROI rectangle. Symbols that do not belong to the specified font are replaced with the symbol for a rectangle.

Rectangle

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

Draws a simple, thick, or filled rectangle.

Parameters

- **img** – Image
- **pt1** – One of the rectangle's 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, cause the function to draw a filled rectangle.
- **lineType** – Type of the line, see *Line* description
- **shift** – Number of fractional bits in the point coordinates

The function draws a rectangle with two opposite corners `pt1` and `pt2` .

CV_RGB

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

Constructs a color value.

Parameters

- **red** – Red component
- **grn** – Green component
- **blu** – Blue component

1.5 XML/YAML Persistence

CvFileStorage

CvFileStorage

File Storage.

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

The structure *CvFileStorage* is a "black box" representation of the file storage associated with a file on disk. Several functions that are described below take `CvFileStorage` as inputs and allow theuser 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 an example of 3×3 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:

```
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 (similar to the 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

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 readed 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 the file). When data is written to a file, it is done sequentially, with minimal buffering. No data is stored in the file storage.

In opposite, when data is read from a file, the whole file is parsed and represented in memory as a tree. Every node of the tree is represented by *CvFileNode* . The type of file node `N` can be retrieved as `CV_NODE_TYPE(N->tag)` . Some file nodes (leaves) are scalars: text strings, integers, 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 (*GetSeqElem*), while elements of maps are accessed by name (*GetFileNodeByName*). The table below describes the different types of file nodes:

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` (see below)

There is no need to access the `map` field directly (by the way, `CvMap` is a hidden structure). The elements of the map can be retrieved with the *GetFileNodeByName* function that takes a pointer to the "map" file node.

A user (custom) object is an instance of either one of the standard CxCore types, such as *CvMat* , *CvSeq* etc., or any type registered with *RegisterTypeInfo* . Such an object is initially represented in a file as a map (as shown in XML and YAML example files above) after the file storage has been opened and parsed. Then the object can be decoded (coverted to native representation) by request - when a user calls the *Read* or *ReadByName* functions.

CvAttrList

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 *Write*). XML attributes inside tags are not supported, aside from the object type specification (`type_id` attribute).

CvTypeInfo

CvTypeInfo

Type information.

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

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 the standard or user-defined types. Instances of the type may or may not contain a pointer to the corresponding *CvTypeInfo* structure. In any case, there is a way to find the type info structure for a given object using the *TypeOf* function. Aternatively, type info can be found by type name using *FindType* , which is used when an object is read from file storage. The user can register a new type with *RegisterType* 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.

Clone

void* **cvClone**(const void* *structPtr*)

Makes a clone of an object.

Parameters

- **structPtr** – The object to clone

The function finds the type of a given object and calls `clone` with the passed object.

EndWriteStruct

void **cvEndWriteStruct**(CvFileStorage* *fs*)

Ends the writing of a structure.

Parameters

- **fs** – File storage

The function finishes the currently written structure.

FindType

CvTypeInfo* **cvFindType**(const char* *typeName*)

Finds a type by its name.

Parameters

- **typeName** – Type name

The function finds a registered type by its name. It returns NULL if there is no type with the specified name.

FirstType

CvTypeInfo* **cvFirstType**(void)

Returns the beginning of a type list.

The function returns the first type in the list of registered types. Navigation through the list can be done via the `prev` and `next` fields of the *CvTypeInfo* structure.

GetFileNode

CvFileNode* **cvGetFileNode**(CvFileStorage* *fs*, CvFileNode* *map*, const CvStringHashNode* *key*, int *createMissing=0*)

Finds a node in a 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 *GetHashedKey*
- **createMissing** – Flag that specifies whether an absent node should be added to the map

The function finds a file node. It is a faster version of *GetFileNodeByName* (see *GetHashedKey* discussion). Also, the function can insert a new node, if it is not in the map yet.

GetFileNodeByName

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

Finds a node in a 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 with the first one.
- **name** – The file node name

The function finds a file node by `name` . The node is searched either in `map` or, if the pointer is NULL, among the top-level file storage nodes. Using this function for maps and *GetSeqElem* (or sequence reader) for sequences, it is possible to nagivate through the file storage. To speed up multiple queries for a certain key (e.g., in the case of an array of structures) one may use a combination of *GetHashedKey* and *GetFileNode* .

GetFileNodeName

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

Returns the name of a file node.

Parameters

- **node** – File node

The function returns the name of a file node or NULL, if the file node does not have a name or if `node` is `NULL` .

GetHashedKey

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

Returns a unique pointer for a given name.

Parameters

- **fs** – File storage
- **name** – Literal node name
- **len** – Length of the name (if it is known apriori), or -1 if it needs to be calculated
- **createMissing** – Flag that specifies, whether an absent key should be added into the hash table

The function returns a unique pointer for each particular file node name. This pointer can be then passed to the *GetFileNode* function that is faster than *GetFileNodeByName* because it compares text strings by comparing pointers rather than the strings' content.

Consider the following example where an array of points is encoded as a sequence of 2-entry maps:

```
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.

```
#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("
        }
    }
    cvReleaseFileStorage( &fs );
    return 0;
}
```

Please note that whatever method of accessing a map you are using, it is still much slower than using plain sequences; for example, in the above example, it is more efficient to encode the points as pairs of integers in a single numeric sequence.

GetRootFileNode

CvFileNode* **cvGetRootFileNode**(const CvFileStorage* *fs*, int *stream_index=0*)
Retrieves one of the top-level nodes of the file storage.

Parameters

- **fs** – File storage
- **stream_index** – Zero-based index of the stream. See *StartNextStream* . In most cases, there is only one stream in the file; however, there can be several.

The function returns one of the 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 a NULL pointer, so all the top-level nodes may be iterated by subsequent calls to the function with `stream_index=0,1,...` , until the NULL pointer is returned. This function may be used as a base for recursive traversal of the file storage.

Load

void* **cvLoad**(const char* *filename*, CvMemStorage* *storage=NULL*, const char* *name=NULL*, const char** *realName=NULL*)
Loads an object from a file.

Parameters

- **filename** – File name
- **storage** – 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.
- **realName** – Optional output parameter that will contain the name of the loaded object (useful if `name=NULL`)

The function loads an object from a file. It provides a simple interface to *Read* . After the object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as a sequence, contour, or graph, one should pass a valid memory storage destination to the function.

OpenFileStorage

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 opens file storage for reading or writing data. In the latter case, a new file is created or an existing file is rewritten. The type of the read or written file is determined by the filename extension: `.xml` for `XML` and `.yml` or `.yaml` for `YAML` . The function returns a pointer to the *CvFileStorage* structure.

Read

void* **cvRead**(CvFileStorage* *fs*, CvFileNode* *node*, CvAttrList* *attributes=NULL*)
Decodes an object and returns a pointer to it.

Parameters

- **fs** – File storage
- **node** – The root object node
- **attributes** – Unused parameter

The function decodes a user object (creates an object in a native representation from the file storage subtree) and returns it. The object to be decoded must be an instance of a registered type that supports the `read` method (see *CvTypeInfo*). The type of the object is determined by the type name that is encoded in the file. If the object is a dynamic structure, it is created either in memory storage and passed to *OpenFileStorage* or, if a NULL pointer was passed, in temporary memory storage, which is released when *ReleaseFileStorage* is called. Otherwise, if the object is not a dynamic structure, it is created in a heap and should be released with a specialized function or by using the generic *Release* .

ReadByName

void* **cvReadByName**(CvFileStorage* *fs*, const CvFileNode* *map*, const char* *name*, CvAttrList* *attributes=NULL*)
Finds an object by name 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 is a simple superposition of *GetFileNodeByName* and *Read* .

ReadInt

int **cvReadInt**(const CvFileNode* *node*, int *defaultValue=0*)
Retrieves an integer value from a file node.

Parameters

- **node** – File node
- **defaultValue** – The value that is returned if `node` is NULL

The function returns an integer that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after *GetFileNode* without checking for a NULL pointer). If the file node has type `CV_NODE_INT` , then `node->data.i` is returned. If the file node has type `CV_NODE_REAL` , then `node->data.f` is converted to an integer and returned. Otherwise the result is not determined.

ReadIntByName

int **cvReadIntByName**(const CvFileStorage* *fs*, const CvFileNode* *map*, const char* *name*, int *defaultValue=0*)

Finds a 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
- **defaultValue** – The value that is returned if the file node is not found

The function is a simple superposition of *GetFileNodeByName* and *ReadInt* .

ReadRawData

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 *WriteRawData* .

The function reads elements from a file node that represents a sequence of scalars.

ReadRawDataSlice

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 *StartReadRawData* .
- **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 *WriteRawData* .

The function reads one or more elements from the file node, representing a sequence, to a 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` $\times$ 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 *SetSeqReaderPos* .

ReadReal

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

Retrieves a floating-point value from a file node.

Parameters

- **node** – File node
- **defaultValue** – The value that is returned if `node` is NULL

The function returns a floating-point value that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after *GetFileNode* without checking for a NULL pointer). If the file node has type `CV_NODE_REAL` , then `node->data.f` is returned. If the file node has type `CV_NODE_INT` , then `node-:math:`>`data.f` is converted to floating-point and returned. Otherwise the result is not determined.

ReadRealByName

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

Finds a 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
- **defaultValue** – The value that is returned if the file node is not found

The function is a simple superposition of *GetFileNodeByName* and *ReadReal* .

ReadString

const char* **cvReadString**(const CvFileNode* *node*, const char* *defaultValue=NULL*)

Retrieves a text string from a file node.

Parameters

- **node** – File node
- **defaultValue** – The value that is returned if `node` is NULL

The function returns a text string that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after *GetFileNode* without checking for a NULL pointer). If the file node has type `CV_NODE_STR` , then `node-:math:`>`data.str.ptr` is returned. Otherwise the result is not determined.

ReadStringByName

const char* **cvReadStringByName**(const CvFileStorage* *fs*, const CvFileNode* *map*, const char* *name*, const char* *defaultValue=NULL*)

Finds a file node by its name 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
- **defaultValue** – The value that is returned if the file node is not found

The function is a simple superposition of *GetFileNodeByName* and *ReadString* .

RegisterType

void **cvRegisterType** (const CvTypeInfo* *info*)
Registers a new type.

Parameters

- **info** – Type info structure

The function registers a new type, which is described by `info` . The function creates a copy of the structure, so the user should delete it after calling the function.

Release

void **cvRelease** (void** *structPtr*)
Releases an object.

Parameters

- **structPtr** – Double pointer to the object

The function finds the type of a given object and calls `release` with the double pointer.

ReleaseFileStorage

void **cvReleaseFileStorage** (CvFileStorage** *fs*)
Releases file storage.

Parameters

- **fs** – Double pointer to the released file storage

The function 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.

Save

void **cvSave** (const char* *filename*, const void* *structPtr*, const char* *name=NULL*, const char* *comment=NULL*, CvAttrList *attributes=cvAttrList()*)
Saves an object to a file.

Parameters

- **filename** – File name
- **structPtr** – 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 *Write*

The function saves an object to a file. It provides a simple interface to *Write* .

StartNextStream

void **cvStartNextStream**(CvFileStorage* *fs*)
Starts the next stream.

Parameters

- **fs** – File storage

The function starts the next stream in file storage. Both YAML and XML support multiple "streams." This is useful for concatenating files or for resuming the writing process.

StartReadRawData

void **cvStartReadRawData**(const CvFileStorage* *fs*, const CvFileNode* *src*, CvSeqReader* *reader*)
Initializes the 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 initializes the sequence reader to read data from a file node. The initialized reader can be then passed to *ReadRawDataSlice* .

StartWriteStruct

void **cvStartWriteStruct**(CvFileStorage* *fs*, const char* *name*, int *struct_flags*, const char* *typeName=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_flags** – A 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 makes 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.
- **typeName** – Optional parameter - the object type name. In case of XML it is written as a `type_id` attribute of the structure opening tag. In the case of YAML it is written after a colon following the structure name (see the example in *CvFileStorage* description). Mainly

it is used with user objects. When the storage is read, the encoded type name is used to determine the object type (see *CvTypeInfo* and *FindTypeInfo*).

- **attributes** – This parameter is not used in the current implementation

The function 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, *EndWriteStruct* should be called. The function can be used to group some objects or to implement the `write` function for a some user object (see *CvTypeInfo*).

TypeOf

CvTypeInfo* **cvTypeOf** (const void* *structPtr*)

Returns the type of an object.

Parameters

- **structPtr** – The object pointer

The function finds the type of a given object. It iterates through the list of registered types and calls the `is_instance` function/method for every type info structure with that object until one of them returns non-zero or until the whole list has been traversed. In the latter case, the function returns NULL.

UnregisterType

void **cvUnregisterType** (const char* *typeName*)

Unregisters the type.

Parameters

- **typeName** – Name of an unregistered type

The function unregisters a type with a specified name. If the name is unknown, it is possible to locate the type info by an instance of the type using *TypeOf* or by iterating the type list, starting from *FirstType* , and then calling `cvUnregisterType(info->typeName)` .

Write

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

Writes a 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 dicsussion below).

The function writes an object to file storage. First, the appropriate type info is found using *TypeOf* . Then, the `write` method associated with 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 *WriteRawData*):

1. CvSeq

- **header_dt** description of user fields of the sequence header that follow CvSeq, or CvChain (if the sequence is a 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.

2. Cvgraph
   - **header_dt** description of user fields of the graph header that follows CvGraph;
   - **vertex_dt** description of user fields of graph vertices
   - **edge_dt** description of user fields of graph edges (note that the 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 the `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;
}
```

WriteComment

void **cvWriteComment**(CvFileStorage* *fs*, const char* *comment*, int *eolComment*)

Writes a comment.

Parameters

- **fs** – File storage
- **comment** – The written comment, single-line or multi-line
- **eolComment** – If non-zero, the function tries to put the comment at the end of current line. If the flag is zero, if the comment is multi-line, or if it does not fit at the end of the current line, the comment starts a new line.

The function writes a comment into file storage. The comments are skipped when the storage is read, so they may be used only for debugging or descriptive purposes.

WriteFileNode

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

Writes a file node to another file storage.

Parameters

- **fs** – Destination file storage

- **new_file_node** – New name of the file node in the destination file storage. To keep the existing name, use *cvGetFileNodeName*
- **node** – The written node
- **embed** – If the written node is a collection and this parameter is not zero, no extra level of hiararchy is created. Instead, all the elements of `node` are written into the currently written structure. Of course, map elements may be written only to a map, and sequence elements may be written only to a sequence.

The function writes a copy of a file node to file storage. Possible applications of the function are merging several file storages into one and conversion between XML and YAML formats.

WriteInt

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 writes a single integer value (with or without a name) to the file storage.

WriteRawData

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 which are written as a signed integer. The type can be used to store structures w**
  example, `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` ‘ and so forth. Other examples: `u` means that the array consists of bytes, and `2d` means the array consists of pairs of doubles.

The function writes an array, whose elements consist of single or multiple numbers. The function call can be replaced with a loop containing a few *WriteInt* and *WriteReal* 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.

WriteReal

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 writes a single floating-point value (with or without a name) to file storage. Special values are encoded as follows: NaN (Not A Number) as .NaN, $\pm\infty$ 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 )
        cvWriteInteger( fs, "max_iterations", termcrit->max_iter );
    if( termcrit->type & CV_TERMCRIT_EPS )
        cvWriteReal( fs, "accuracy", termcrit->epsilon );
    cvEndWriteStruct( fs );
}
```

WriteString

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. 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 writes a text string to file storage.

1.6 Clustering

KMeans2

int **cvKMeans2** (const CvArr* *samples*, int *nclusters*, CvArr* *labels*, CvTermCriteria *termcrit*, int *attempts=1*, CvRNG* *rng=0*, int *flags=0*, CvArr* *centers=0*, double* *compactness=0*)

Splits set of vectors by a given number of clusters.

Parameters

- **samples** – Floating-point matrix of input samples, one row per sample
- **nclusters** – Number of clusters to split the set by
- **labels** – Output integer vector storing cluster indices for every sample
- **termcrit** – Specifies maximum number of iterations and/or accuracy (distance the centers can move by between 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 $\sum_i ||\texttt{samples}_i - \texttt{centers}_{\texttt{labels}_i}||^2$ after every attempt; the best (minimum) value is chosen and the corresponding labels are returned by the function. Basically, the 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=CV_KMEANS_USE_INITIAL_LABELS`) and, based on the output compactness or any other criteria, choose the best clustering.

The function `cvKMeans2` implements a k-means algorithm that finds the centers of `nclusters` clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a cluster index for samples stored in the i-th row of the `samples` matrix.

```
#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)
    int i, sample_count = cvRandInt(&rng)
    CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
    CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

    /* generate random sample from multigaussian distribution */
    for( k = 0; k < cluster_count; k++ )
    {
        CvPoint center;
        CvMat point_chunk;
        center.x = cvRandInt(&rng)
        center.y = cvRandInt(&rng)
        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)
        CvPoint2D32f* pt2 =
            (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)
        CvPoint2D32f temp;
        CV_SWAP( *pt1, *pt2, temp );
    }

    cvKMeans2( points, cluster_count, clusters,
               cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

    cvZero( img );

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

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

```
        cvShowImage( "clusters", img );

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

SeqPartition

int **cvSeqPartition**(const CvSeq* *seq*, CvMemStorage* *storage*, CvSeq** *labels*, CvCmpFunc *is_equal*, void* *userdata*)

Splits a sequence into equivalency classes.

Parameters

- **seq** – The sequence to partition
- **storage** – The storage block to store the sequence of equivalency classes. If it is NULL, the function uses `seq->storage` for output labels
- **labels** – Ouput 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 an equivalency criteria
- **userdata** – Pointer that is transparently passed to the `is_equal` function

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

The function `cvSeqPartition` implements a quadratic algorithm for splitting a set into one or more equivalancy classes. The function returns the number of equivalency classes.

```
#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) + (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("
    cvZero( canvas );

    for( i = 0; i < labels->total; i++ )
    {
        CvPoint pt = *(CvPoint*)cvGetSeqElem( point_seq, i );
        CvScalar color = colors[*(int*)cvGetSeqElem( labels, i )];
        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 )
        pt.y = cvRandInt( &rng )
        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;
}
```

1.7 Utility and System Functions and Macros

Error Handling

Error handling in OpenCV is similar to IPL (Image Processing Library). In the case of an error, functions do not return the error code. Instead, they raise an error using `CV_ERROR` macro that calls *Error* that, in its turn, sets the error status with *SetErrStatus* and calls a standard or user-defined error handler (that can display a message box, write to log, etc., see *RedirectError*). There is a global variable, one per each program thread, that contains current error status (an integer value). The status can be retrieved with the *GetErrStatus* function.

There are three modes of error handling (see *SetErrMode* and *GetErrMode*):

- **Leaf** . The program is terminated after the 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 unwound (it is done w/o using a C++ exception mechanism). The user may check error code after calling the `CxCore` function with *GetErrStatus* and react.
- **Silent** . Similar to `Parent` mode, but no error handler is called.

Actually, the semantics of the `Leaf` and `Parent` modes are implemented by error handlers and the above description is true for them. *GuiBoxReport* behaves slightly differently, and some custom error handlers may implement quite different semantics.

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 )                                       \


/* Checks status after calling CXCORE function */
#define CV_CHECK()                                                  \


/* Provies shorthand for CXCORE function call and CV_CHECK() */
#define CV_CALL( Statement )                                        \


/* Checks some condition in both debug and release configurations */
#define CV_ASSERT( Condition )                                          \


/* 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, below is a documented example of a typical CXCORE function and an example of the function use.

Example: 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 could not be accessed from the epilogue.

    if( !CV_IS_MAT(input_array) || !CV_IS_MAT(output_array) )
        // use CV_ERROR() to raise an error
        CV_ERROR( CV_StsBadArg,
        "input_array or output_array are not valid matrices" );

    // some restrictions that are going to be removed later, may be checked
    // with CV_ASSERT()
    CV_ASSERT( input_array->rows == 1 && output_array->rows == 1 );

    // use CV_CALL for safe function call
    CV_CALL( temp_array = cvCreateMat( input_array->rows,
                                       MAX(input_array->cols,
                                       output_array->cols),
                                       input_array->type ));

    if( output_array->cols > input_array->cols )
        CV_CALL( cvZero( temp_array ));

    temp_array->cols = input_array->cols;
    CV_CALL( cvDCT( input_array, temp_array, CV_DXT_FORWARD ));
    temp_array->cols = output_array->cols;
    CV_CALL( cvDCT( temp_array, output_array, CV_DXT_INVERSE ));
    CV_CALL( cvScale( output_array,
                      output_array,
                      1./sqrt((double)input_array->cols*output_array->cols), 0 ));

    __END__; // finish processing. Epilogue follows after the macro.

    // release temp_array. If temp_array has not been allocated
    // before an error occured, cvReleaseMat
    // takes care of it and does nothing in this case.
    cvReleaseMat( &temp_array );
}

int main( int argc, char** argv )
{
    CvMat* src = cvCreateMat( 1, 512, CV_32F );
```

```
#if 1 /* no errors */
    CvMat* dst = cvCreateMat( 1, 256, CV_32F );
#else
    CvMat* dst = 0; /* test error processing mechanism */
#endif
    cvSet( src, cvRealScalar(1.), 0 );
#if 0 /* change 0 to 1 to suppress error handler invocation */
    cvSetErrMode( CV_ErrModeSilent );
#endif
    cvResizeDCT( src, dst ); // if some error occurs, the message
                             // box will popup, or a message will be
                             // written to log, or some user-defined
                             // processing will be done
    if( cvGetErrStatus() < 0 )
        printf("Some error occured" );
    else
        printf("Everything is OK" );
    return 0;
}
```

GetErrStatus

int **cvGetErrStatus** (void)
Returns the current error status.

The function returns the current error status - the value set with the last *SetErrStatus* call. Note that in `Leaf` mode, the program terminates immediately after an error occurs, so to always gain control after the function call, one should call *SetErrMode* and set the `Parent` or `Silent` error mode.

SetErrStatus

void **cvSetErrStatus** (int *status*)
Sets the error status.

Parameters

- **status** – The error status

The function 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 an error. In other cases it is more natural to call *Error* or `CV_ERROR` .

GetErrMode

int **cvGetErrMode** (void)
Returns the current error mode.

The function returns the current error mode - the value set with the last *SetErrMode* call.

SetErrMode

..

int **cvSetErrMode** (int *mode*)
Sets the error mode.

#define CV_ErrModeLeaf 0 #define CV_ErrModeParent 1 #define CV_ErrModeSilent 2

param mode The error mode

The function sets the specified error mode. For descriptions of different error modes, see the beginning of the error section.

Error

int **cvError** (int *status*, const char* *func_name*, const char* *err_msg*, const char* *filename*, int *line*)

Raises an error.

Parameters

- **status** – The error status
- **func_name** – Name of the function where the error occured
- **err_msg** – Additional information/diagnostics about the error
- **filename** – Name of the file where the error occured
- **line** – Line number, where the error occured

The function sets the error status to the specified value (via *SetErrStatus*) and, if the error mode is not `Silent` , calls the error handler.

ErrorStr

const char* **cvErrorStr** (int *status*)

Returns textual description of an error status code.

Parameters

- **status** – The error status

The function returns the textual description for the specified error status code. In the case of unknown status, the function returns a NULL pointer.

RedirectError

CvErrorCallback **cvRedirectError** (CvErrorCallback *error_handler*, void* *userdata=NULL*, void** *prevUserdata=NULL*)

Sets a new error handler.

Parameters

- **error_handler** – The new error _ handler
- **userdata** – Arbitrary pointer that is transparently passed to the error handler
- **prevUserdata** – Pointer to the previously assigned user data pointer

```
typedef int (CV_CDECL *CvErrorCallback)( int status, const char* func_name,
                    const char* err_msg, const char* file_name, int line );
```

The function sets a new error handler that can be one of the standard handlers or a custom handler that has a specific interface. The handler takes the same parameters as the *Error* function. If the handler returns a non-zero value, the program is terminated; otherwise, it continues. The error handler may check the current error mode with *GetErrMode* to make a decision.

cvNulDevReport cvStdErrReport cvGuiBoxReport

int **cvNulDevReport** (int *status*, const char* *func_name*, const char* *err_msg*, const char* *file_name*, int *line*, void* *userdata*)

int **cvStdErrReport** (int *status*, const char* *func_name*, const char* *err_msg*, const char* *file_name*, int *line*, void* *userdata*)

int **cvGuiBoxReport** (int *status*, const char* *func_name*, const char* *err_msg*, const char* *file_name*, int *line*, void* *userdata*)

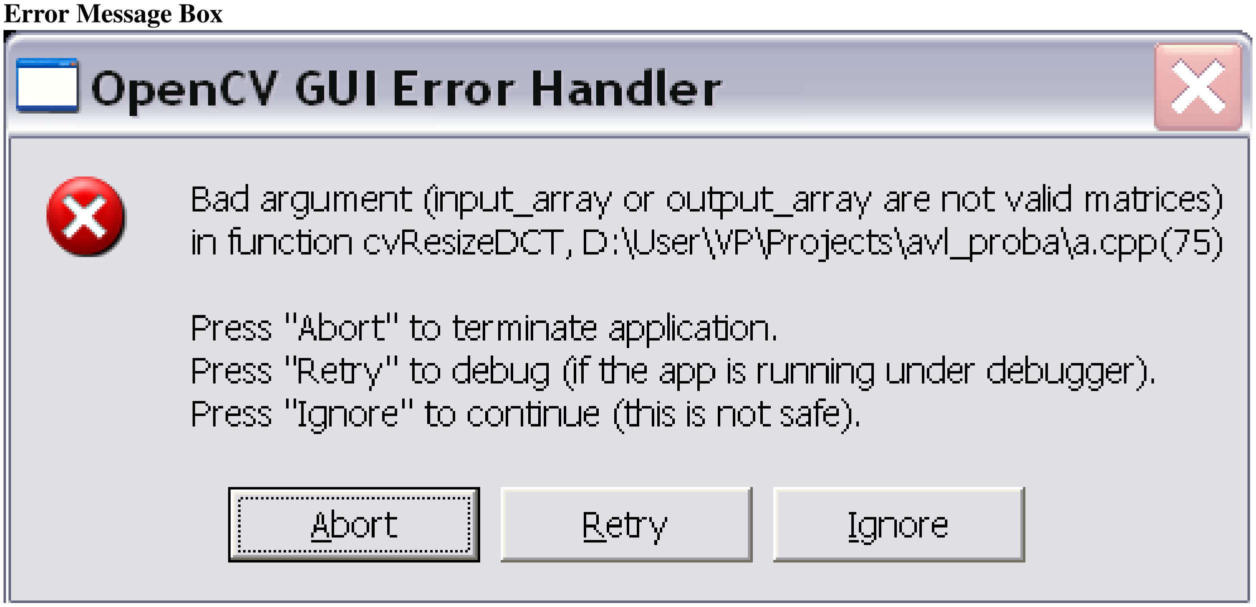
Provide standard error handling.

Parameters

- **status** – The error status
- **func_name** – Name of the function where the error occured
- **err_msg** – Additional information/diagnostics about the error
- **filename** – Name of the file where the error occured
- **line** – Line number, where the error occured
- **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` is the default on other systems. `cvGuiBoxReport` pops up a message box with the error description and suggest a few options. Below is an example message box that may be recieved with the sample code above, if one introduces an error as described in the sample.

Error Message Box



If the error handler is set to `cvStdErrReport` , the above message will be printed to standard error output and the 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:UserVPProjectsavl_probaa.cpp(75)
Terminating the application...
```

Alloc

void* **cvAlloc** (size_t *size*)
Allocates a memory buffer.

Parameters

- **size** – Buffer size in bytes

The function allocates `size` bytes and returns a pointer to the allocated buffer. In the case of an error the function reports an error and returns a NULL pointer. By default, `cvAlloc` calls `icvAlloc` which itself calls `malloc` . However it is possible to assign user-defined memory allocation/deallocation functions using the *SetMemoryManager* function.

Free

void **cvFree** (void** *ptr*)
Deallocates a memory buffer.

Parameters

- **ptr** – Double pointer to released buffer

The function deallocates a memory buffer allocated by *Alloc* . It clears the pointer to buffer upon exit, which is why the double pointer is used. If the `*buffer` is already NULL, the function does nothing.

GetTickCount

int64 **cvGetTickCount** (void)
Returns the number of ticks.

The function returns number of the ticks 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 ticks to time units, use *GetTickFrequency* .

GetTickFrequency

double **cvGetTickFrequency** (void)
Returns the number of ticks per microsecond.

The function returns the number of ticks per microsecond. Thus, the quotient of *GetTickCount* and *GetTickFrequency* will give the number of microseconds starting from the platform-dependent event.

RegisterModule

..

int **cvRegisterModule** (const CvModuleInfo* *moduleInfo*)
Registers another 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;

param moduleInfo Information about the module

The function adds a module to the list of registered modules. After the module is registered, information about it can be retrieved using the *GetModuleInfo* function. Also, the registered module makes full use of optimized plugins (IPP, MKL, ...), supported by CXCORE. CXCORE itself, CV (computer vision), CVAUX (auxilary computer vision), and HIGHGUI (visualization and image/video acquisition) are examples of modules. Registration is usually done when the shared library is loaded. See `cxcore/src/cxswitcher.cpp` and `cv/src/cvswitcher.cpp` for details about 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.

GetModuleInfo

void **cvGetModuleInfo**(const char* *moduleName*, const char** *version*, const char** *loadedAddonPlugins*)

Retrieves information about registered module(s) and plugins.

Parameters

- **moduleName** – Name of the module of interest, or NULL, which means all the modules
- **version** – The output parameter. Information about the module(s), including version
- **loadedAddonPlugins** – The list of names and versions of the optimized plugins that CXCORE was able to find and load

The function returns information about one or all of the registered modules. The returned information is stored inside the libraries, so the user should not deallocate or modify the returned text strings.

UseOptimized

int **cvUseOptimized**(int *onoff*)

Switches between optimized/non-optimized modes.

Parameters

- **onoff** – Use optimized ($\neq 0$) or not ($= 0$)

The function 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 and MKL upgrading 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).

SetMemoryManager

..

void **cvSetMemoryManager**(CvAllocFunc *allocFunc=NULL*, CvFreeFunc *freeFunc=NULL*, void* *userdata=NULL*)

Accesses custom/default memory managing functions.

typedef void* (CV_CDECL *CvAllocFunc)(size_t size, void* userdata); typedef int (CV_CDECL *CvFreeFunc)(void* pptr, void* userdata);

param allocFunc Allocation function; the interface is similar to `malloc` , except that `userdata` may be used to determine the context

param freeFunc Deallocation function; the interface is similar to `free`

param userdata User data that is transparently passed to the custom functions

The function sets user-defined memory managment functions (substitutes 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 `cvAlloc` and *Free* from the custom allocation/deallocation functions.

If the `alloc_func` and `free_func` pointers are `NULL` , the default memory managing functions are restored.

SetIPLAllocators

\
\

void **cvSetIPLAllocators** (Cv_iplCreateImageHeader *create_header*, Cv_iplAllocateImageData *allocate_data*, Cv_iplDeallocate *deallocate*, Cv_iplCreateROI *create_roi*, Cv_iplCloneImage *clone_image*)

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)

param create_header Pointer to iplCreateImageHeader

param allocate_data Pointer to iplAllocateImage

param deallocate Pointer to iplDeallocate

param create_roi Pointer to iplCreateROI

param clone_image Pointer to iplCloneImage

The function causes 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.

CHAPTER

TWO

IMGPROC. IMAGE PROCESSING

2.1 Histograms

CvHistogram

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;
```

CalcBackProject

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

Calculates the back projection.

Parameters

- **image** – Source images (though you may pass CvMat** as well)
- **back_project** – Destination back projection image of the same type as the source images
- **hist** – Histogram

The function 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 in the destination image. In terms of statistics, the value of each output image pixel is the 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.

CalcBackProjectPatch

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

Locates a template within an image by using a histogram comparison.

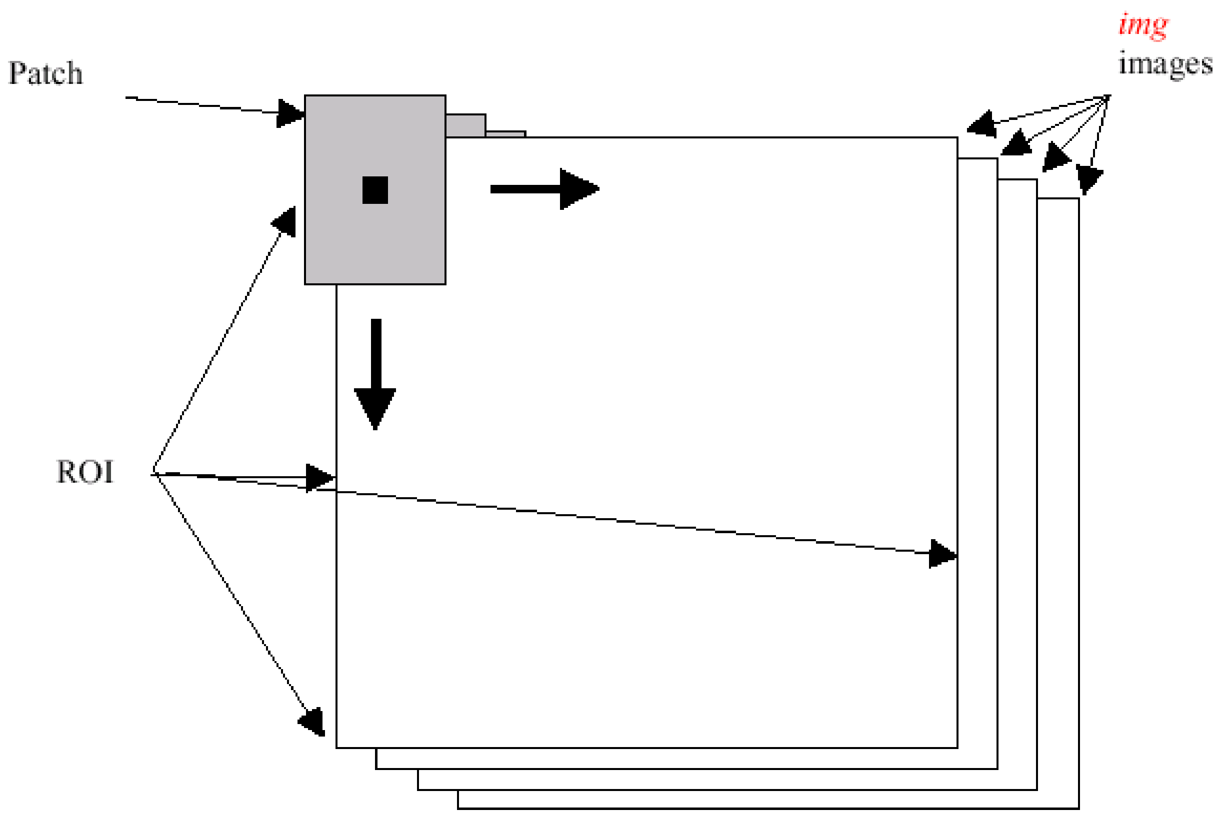
Parameters

- **images** – Source images (though, you may pass CvMat** as well)
- **dst** – Destination image
- **patch_size** – Size of the patch slid though the source image
- **hist** – Histogram
- **method** – Comparison method, passed to *CompareHist* (see description of that function)
- **factor** – Normalization factor for histograms, will affect the normalization scale of the destination image, pass 1 if unsure

The function calculates the back projection by comparing histograms of the source image patches with the given histogram. Taking measurement results from some image at each location over ROI creates an array `image` . These results might be one or more of hue, `x` derivative, `y` derivative, Laplacian filter, oriented Gabor filter, etc. Each measurement output is collected into its own separate image. The `image` image array is a collection of these measurement images. A multi-dimensional histogram `hist` is constructed by sampling from the `image` image array. The final histogram is normalized. The `hist` histogram has as many dimensions as the number of elements in `image` array.

Each new image is measured and then converted into an `image` image array over a chosen ROI. Histograms are taken from this `image` image in an area covered by a "patch" with an anchor at center as shown in the picture below. The histogram is normalized using the parameter `norm_factor` so that it may be compared with `hist` . The calculated histogram is compared to the model histogram; `hist` uses The function `cvCompareHist` with the comparison method= `method`). The resulting output is placed at the location corresponding to the patch anchor in the probability image `dst` . This process is repeated as the patch is slid over the ROI. Iterative histogram update by subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can save a lot of operations, though it is not implemented yet.

Back Project Calculation by Patches



CalcHist

void **cvCalcHist** (IplImage** *image*, CvHistogram* *hist*, int *accumulate=0*, const CvArr* *mask=NULL*)
Calculates the histogram of image(s).

Parameters

- **image** – Source images (though you may pass CvMat** as well)
- **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 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.

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

int main( int argc, char** argv )
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
```

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

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

        for( h = 0; h < h_bins; h++ )
        {
            for( s = 0; s < s_bins; s++ )
            {
                float bin_val = cvQueryHistValue_2D( hist, h, s );
                int intensity = cvRound(bin_val*255/max_value);
                cvRectangle( hist_img, cvPoint( h*scale, s*scale ),
                             cvPoint( (h+1)*scale - 1, (s+1)*scale - 1),
                             CV_RGB(intensity,intensity,intensity),
                             CV_FILLED );
            }
        }

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

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

        cvWaitKey(0);
    }
}
```

CalcProbDensity

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 calculates the object probability density from the two histograms as:

$$\texttt{dist\_hist}(I) = \begin{cases} 0 & \text{if } \texttt{hist1}(I) = 0 \\ \texttt{scale} & \text{if } \texttt{hist1}(I) \ne 0 \text{ and } \texttt{hist2}(I) > \texttt{hist1}(I) \\ \frac{\texttt{hist2}(I) \cdot \texttt{scale}}{\texttt{hist1}(I)} & \text{if } \texttt{hist1}(I) \ne 0 \text{ and } \texttt{hist2}(I) \le \texttt{hist1}(I) \end{cases}$$

So the destination histogram bins are within less than `scale` .

ClearHist

void **cvClearHist** (CvHistogram* *hist*)

Clears the histogram.

Parameters

- **hist** – Histogram

The function sets all of the histogram bins to 0 in the case of a dense histogram and removes all histogram bins in the case of a sparse array.

CompareHist

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 the following:
  - **CV_COMP_CORREL** Correlation
  - **CV_COMP_CHISQR** Chi-Square
  - **CV_COMP_INTERSECT** Intersection
  - **CV_COMP_BHATTACHARYYA** Bhattacharyya distance

The function compares two dense histograms using the specified method (H_1 denotes the first histogram, H_2 the second):

- Correlation (method=CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H'_1(I) \cdot H'_2(I))}{\sqrt{\sum_I (H'_1(I)^2) \cdot \sum_I (H'_2(I)^2)}}$$

where

$$H'_k(I) = \frac{H_k(I) - 1}{N \cdot \sum_J H_k(J)}$$

where N is the number of histogram bins.

- Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

- Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

- Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \sum_I \frac{\sqrt{H_1(I) \cdot H_2(I)}}{\sqrt{\sum_I H_1(I) \cdot \sum_I H_2(I)}}}$$

The function returns $d(H_1, H_2)$.

Note: the method `CV_COMP_BHATTACHARYYA` only works with normalized histograms.

To compare a sparse histogram or more general sparse configurations of weighted points, consider using the *CalcEMD2* function.

CopyHist

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

Copies a histogram.

Parameters

- **src** – Source histogram
- **dst** – Pointer to destination histogram

The function 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's bin values to the destination histogram and sets the same bin value ranges as in `src` .

CreateHist

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

Creates a histogram.

Parameters

- **dims** – Number of histogram dimensions
- **sizes** – Array of the histogram dimension sizes
- **type** – Histogram representation format: `CV_HIST_ARRAY` means that the histogram data is represented as a 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 the histogram bins. Its meaning depends on the `uniform` parameter value. The ranges are used for when the histogram is calculated or backprojected 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 $0 <= i < cDims$ `ranges[i]` is an array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is then split into `dims[i]` equal parts to determine the `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: $\texttt{lower}_0, \texttt{upper}_0, \texttt{lower}_1, \texttt{upper}_1 = \texttt{lower}_2, ...\texttt{upper}_{dims[i]-1}$ where $\texttt{lower}_j$ and $\texttt{upper}_j$ 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 *CalcHist* and filled with 0 by *CalcBackProject*

The function creates a histogram of the specified size and returns a pointer to the created histogram. If the array `ranges` is 0, the histogram bin ranges must be specified later via the function *SetHistBinRanges* . Though *CalcHist* and *CalcBackProject* may process 8-bit images without setting bin ranges, they assume thy are equally spaced in 0 to 255 bins.

GetHistValue*D

float **cvGetHistValue_1D**(hist, idx0)

float **cvGetHistValue_2D**(hist, idx0, idx1)

float **cvGetHistValue_3D**(hist, idx0, idx1, idx2)

float **cvGetHistValue_nD**(hist, idx)

Returns a pointer to the histogram bin.

Parameters

- **hist** – Histogram
- **idx1, idx2, idx3** (*idx0,*) – Indices of the bin
- **idx** – Array of indices

```
#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 )))
```

The macros `GetHistValue` return a pointer to the specified bin of the 1D, 2D, 3D or N-D histogram. In the case of a sparse histogram the function creates a new bin and sets it to 0, unless it exists already.

GetMinMaxHistValue

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

Finds the 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 the minimum
- **max_idx** – Pointer to the array of coordinates for the maximum

The function finds the minimum and maximum histogram bins and their positions. All of output arguments are optional. Among several extremas with the same value the ones with the minimum index (in lexicographical order) are returned. In the case of several maximums or minimums, the earliest in lexicographical order (extrema locations) is returned.

MakeHistHeaderForArray

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

Makes a histogram out of an array.

Parameters

- **dims** – Number of histogram dimensions
- **sizes** – Array of the 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 *CreateHist*
- **uniform** – Uniformity flag, see *CreateHist*

The function initializes the histogram, whose header and bins are allocated by th user. *ReleaseHist* does not need to be called afterwards. Only dense histograms can be initialized this way. The function returns `hist` .

NormalizeHist

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

Normalizes the histogram.

Parameters

- **hist** – Pointer to the histogram
- **factor** – Normalization factor

The function normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor` .

QueryHistValue*D

float **QueryHistValue_1D**(CvHistogram *hist*, int *idx0*)

Queries the value of the histogram bin.

Parameters

- **hist** – Histogram
- **idx1, idx2, idx3** (*idx0,*) – Indices of the bin

- **idx** – Array of indices

```
#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) )
```

The macros return the value of the specified bin of the 1D, 2D, 3D or N-D histogram. In the case of a sparse histogram the function returns 0, if the bin is not present in the histogram no new bin is created.

ReleaseHist

void **cvReleaseHist** (CvHistogram** *hist*)

Releases the histogram.

Parameters

- **hist** – Double pointer to the released histogram

The function releases the histogram (header and the data). The pointer to the histogram is cleared by the function. If `*hist` pointer is already `NULL` , the function does nothing.

SetHistBinRanges

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

Sets the bounds of the histogram bins.

Parameters

- **hist** – Histogram
- **ranges** – Array of bin ranges arrays, see *CreateHist*
- **uniform** – Uniformity flag, see *CreateHist*

The function is a stand-alone function for setting bin ranges in the histogram. For a more detailed description of the parameters `ranges` and `uniform` see the *CalcHist* function, that can initialize the ranges as well. Ranges for the histogram bins must be set before the histogram is calculated or the backproject of the histogram is calculated.

ThreshHist

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

Thresholds the histogram.

Parameters

- **hist** – Pointer to the histogram
- **threshold** – Threshold level

The function clears histogram bins that are below the specified threshold.

2.2 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as `Mat()` 's), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y) . It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc.

IplConvKernel

`IplConvKernel`

An IplConvKernel is a rectangular convolution kernel, created by function *CreateStructuringElementEx* .

CopyMakeBorder

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

Copies an image and makes a border around it.

Parameters

- **src** – The source image
- **dst** – The destination image
- **offset** – Coordinates of the top-left corner (or bottom-left in the case of images with bottom-left origin) of the destination image rectangle where the source image (or its ROI) is copied. Size of the rectanlge matches the source image size/ROI size
- **bordertype** – Type of the border to create around the copied source image rectangle; types include:
  - **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` is `IPL_BORDER_CONSTANT`

The function copies the source 2D array into the interior of the 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 a zero border or a border, filled with 1's or 255's.

CreateStructuringElementEx

IplConvKernel* **cvCreateStructuringElementEx**(int *cols*, int *rows*, int *anchorX*, int *anchorY*, int *shape*, int* *values=NULL*)

Creates a structuring element.

Parameters

- **cols** – Number of columns in the structuring element
- **rows** – Number of rows in the structuring element
- **anchorX** – Relative horizontal offset of the anchor point
- **anchorY** – 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 CreateStructuringElementEx allocates and fills the structure `IplConvKernel` , which can be used as a structuring element in the morphological operations.

Dilate

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

Dilates an image by using a specific structuring element.

Parameters

- **src** – Source image
- **dst** – Destination image
- **element** – Structuring element used for dilation. If it is `NULL`, a `3 x 3` rectangular structuring element is used
- **iterations** – Number of times dilation is applied

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\max_{(x',y')\, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. For color images, each channel is processed independently.

Erode

void **cvErode** (const CvArr* *src*, CvArr* *dst*, IplConvKernel* *element=NULL*, int *iterations=1*)
Erodes an image by using a specific structuring element.

Parameters

- **src** – Source image
- **dst** – Destination image
- **element** – Structuring element used for erosion. If it is `NULL` , a `3 x 3` rectangular structuring element is used
- **iterations** – Number of times erosion is applied

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\min_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. For color images, each channel is processed independently.

Filter2D

void **cvFilter2D** (const CvArr* *src*, CvArr* *dst*, const CvMat* *kernel*, CvPoint *anchor=cvPoint(-1*, -1))
Convolves an image with the kernel.

Parameters

- **src** – The source image
- **dst** – The destination image
- **kernel** – Convolution kernel, a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using *Split* and process them individually
- **anchor** – The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor shoud lie within the kernel. The special default value (-1,-1) means that it is at the kernel center

The function applies an arbitrary 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 that are inside the image.

Laplace

void **cvLaplace** (const CvArr* *src*, CvArr* *dst*, int *apertureSize=3*)
Calculates the Laplacian of an image.

Parameters

- **src** – Source image
- **dst** – Destination image
- **apertureSize** – Aperture size (it has the same meaning as *Sobel*)

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

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

Setting `apertureSize` = 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 the *Sobel* function, no scaling is done and the same combinations of input and output formats are supported.

MorphologyEx

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 the following:
  - **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 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 a morphological gradient and, in the case of in-place operation, for “top hat” and “black hat”.

PyrDown

void **cvPyrDown** (const CvArr* *src*, CvArr* *dst*, int *filter=CV_GAUSSIAN_5x5*)
Downsamples an image.

Parameters

- **src** – The source image
- **dst** – The destination image, should have a half as large width and height than the source
- **filter** – Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported

The function performs the downsampling step of the Gaussian pyramid decomposition. First it convolves the source image with the specified filter and then downsamples the image by rejecting even rows and columns.

ReleaseStructuringElement

void **cvReleaseStructuringElement** (IplConvKernel** *element*)
Deletes a structuring element.

Parameters

- **element** – Pointer to the deleted structuring element

The function releases the structure `IplConvKernel` that is no longer needed. If `*element` is `NULL` , the function has no effect.

Smooth

void **cvSmooth** (const CvArr* *src*, CvArr* *dst*, int *smoothtype=CV_GAUSSIAN*, int *param1=3*, int *param2=0*, double *param3=0*, double *param4=0*)
Smooths the image in one of several ways.

Parameters

- **src** – The source image
- **dst** – The destination image
- **smoothtype** – Type of the smoothing:
  - **CV_BLUR_NO_SCALE** linear convolution with $\texttt{param1} \times \texttt{param2}$ box kernel (all 1's). If you want to smooth different pixels with different-size box kernels, you can use the integral image that is computed using *Integral*
  - **CV_BLUR** linear convolution with $\texttt{param1} \times \texttt{param2}$ box kernel (all 1's) with subsequent scaling by $1/(\texttt{param1} \cdot \texttt{param2})$
  - **CV_GAUSSIAN** linear convolution with a $\texttt{param1} \times \texttt{param2}$ Gaussian kernel
  - **CV_MEDIAN** median filter with a $\texttt{param1} \times \texttt{param1}$ square aperture
  - **CV_BILATERAL** bilateral filter with a $\texttt{param1} \times \texttt{param1}$ square aperture, color sigma= `param3` and spatial sigma= `param4` . If `param1=0` , the aperture square side is set to `cvRound(param4*1.5)*2+1` . Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

- **param1** – The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)
- **param2** – The second parameter of the smoothing operation, the aperture height. Ignored by `CV_MEDIAN` and `CV_BILATERAL` methods. In the case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1` . Otherwise it must be a positive odd number.
- **param3** – In the case of a Gaussian parameter this parameter may specify Gaussian σ (standard deviation). If it is zero, it is calculated from the kernel size:

$$\sigma = 0.3(n/2 - 1) + 0.8 \quad \text{where} \quad n = \begin{array}{l} \texttt{param1} \text{ for horizontal kernel} \\ \texttt{param2} \text{ for vertical kernel} \end{array}$$

  Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function smooths an image using one of several 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 8-bit to 16-bit format (similar to *Sobel* and *Laplace*) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

Sobel

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

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator.

Parameters

- **src** – Source image of type CvArr*
- **dst** – Destination image
- **xorder** – Order of the derivative x
- **yorder** – Order of the derivative y
- **apertureSize** – Size of the extended Sobel kernel, must be 1, 3, 5 or 7

In all cases except 1, an $\texttt{apertureSize} \times \texttt{apertureSize}$ separable kernel will be used to calculate the derivative. For $\texttt{apertureSize} = 1$ a 3×1 or 1×3 a kernel is used (Gaussian smoothing is not done). There is also the special value `CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. Scharr aperture is

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

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

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

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `apertureSize` = 3) or (`xorder` = 0, `yorder` = 1, `apertureSize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

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

and the second one corresponds to a kernel of:

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

or a kernel of:

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

depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger numbers (in absolute values) than the source image does. To avoid overflow, the function requires a 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using the *ConvertScale* or the *ConvertScaleAbs* function. Besides 8-bit images the function can process 32-bit floating-point images. Both the source and the destination must be single-channel images of equal size or equal ROI size.

2.3 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x,y) = \texttt{src}(f_x(x,y), f_y(x,y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic *Remap* and to the simplest and the fastest *Resize* , need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x, y) one of $f_x(x, y)$ or $f_y(x, y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method `BORDER_TRANSPARENT` , which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x, y)$ and $f_y(x, y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods , where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x, y), f_y(x, y))$ and then the value of the polynomial at $(f_x(x, y), f_y(x, y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see *Resize* .

GetRotationMatrix2D

CvMat* **cv2DRotationMatrix**(CvPoint2D32f *center*, double *angle*, double *scale*, CvMat* *mapMatrix*)

Calculates the 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 to be the top-left corner)
- **scale** – Isotropic scale factor
- **mapMatrix** – Pointer to the destination 2×3 matrix

The function `cv2DRotationMatrix` calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} - (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot cos(\texttt{angle}), \beta = \texttt{scale} \cdot sin(\texttt{angle})$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

GetAffineTransform

CvMat* **cvGetAffineTransform**(const CvPoint2D32f* *src*, const CvPoint2D32f* *dst*, CvMat* *mapMatrix*)

Calculates the 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
- **mapMatrix** – Pointer to the destination 2×3 matrix

The function cvGetAffineTransform calculates the matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

GetPerspectiveTransform

CvMat* **cvGetPerspectiveTransform**(const CvPoint2D32f* *src*, const CvPoint2D32f* *dst*, CvMat* *mapMatrix*)

Calculates the 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
- **mapMatrix** – Pointer to the destination 3×3 matrix

The function `cvGetPerspectiveTransform` calculates a matrix of perspective transforms such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

GetQuadrangleSubPix

void **cvGetQuadrangleSubPix** (const CvArr* *src*, CvArr* *dst*, const CvMat* *mapMatrix*)

Retrieves the pixel quadrangle from an image with sub-pixel accuracy.

Parameters

- **src** – Source image
- **dst** – Extracted quadrangle
- **mapMatrix** – 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:

$$dst(x, y) = src(A_{11}x' + A_{12}y' + b_1, A_{21}x' + A_{22}y' + b_2)$$

where

$$x' = x - \frac{(width(dst) - 1)}{2}, y' = y - \frac{(height(dst) - 1)}{2}$$

and

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

The values of pixels at non-integer coordinates 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.

GetRectSubPix

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

Retrieves the pixel rectangle from an 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` :

$$dst(x,y) = src(x + \texttt{center.x} - (width(\texttt{dst}) - 1) * 0.5, y + \texttt{center.y} - (height(\texttt{dst}) - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

LogPolar

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

Remaps an 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 methods and the following optional flags:
  - **CV_WARP_FILL_OUTLIERS** fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero
  - **CV_WARP_INVERSE_MAP** See below

The function `cvLogPolar` transforms the 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

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

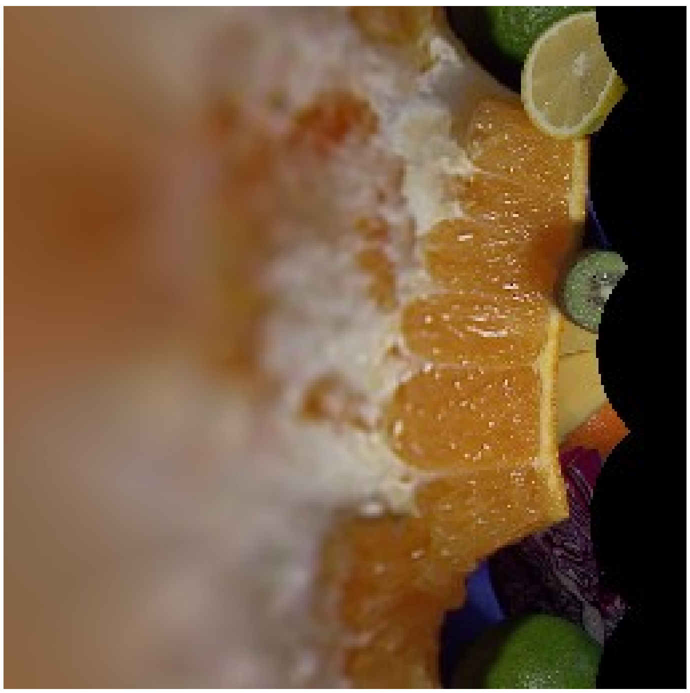
```
#include <cv.h>
#include <highgui.h>

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



Remap

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 a generic geometrical transformation to the image.

Parameters

- **src** – Source image
- **dst** – Destination image
- **mapx** – The map of x-coordinates (CV _ 32FC1 image)
- **mapy** – The map of y-coordinates (CV _ 32FC1 image)
- **flags** – A combination of interpolation method and the following optional flag(s):

- **CV_WARP_FILL_OUTLIERS** fills all of 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 the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(\texttt{mapx}(x,y), \texttt{mapy}(x,y))$$

Similar to other geometrical transformations, some interpolation method (specified by user) is used to extract pixels with non-integer coordinates. Note that the function can not operate in-place.

Resize

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

Resizes an image.

Parameters

- **src** – Source image
- **dst** – Destination image
- **interpolation** – Interpolation method:
  - **CV_INTER_NN** nearest-neigbor 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 terms of zooming it is similar to the `CV_INTER_NN` method
  - **CV_INTER_CUBIC** bicubic interpolation

The function `cvResize` resizes an image `src` so that it fits exactly into `dst` . If ROI is set, the function considers the ROI as supported.

WarpAffine

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

Applies an affine transformation to an image.

Parameters

- **src** – Source image
- **dst** – Destination image
- **mapMatrix** – 2×3 transformation matrix
- **flags** – A combination of interpolation methods and the following optional flags:
  - **CV_WARP_FILL_OUTLIERS** fills all of 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 inversely** transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

- **fillval** – A value used to fill outliers

The function `cvWarpAffine` transforms the source image using the specified matrix:

$$dst(x', y') = src(x, y)$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV\_WARP\_INVERSE\_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

The function is similar to *GetQuadrangleSubPix* but they are not exactly the same. *WarpAffine* 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 *GetQuadrangleSubPix* may extract quadrangles from 8-bit images into floating-point buffer, has smaller overhead and always changes the whole destination image content. Note that the function can not operate in-place.

To transform a sparse set of points, use the *Transform* function from cxcore.

WarpPerspective

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

Applies a perspective transformation to an image.

Parameters

- **src** – Source image
- **dst** – Destination image
- **mapMatrix** – 3×3 transformation matrix
- **flags** – A combination of interpolation methods and the following optional flags:
  - **CV_WARP_FILL_OUTLIERS** fills all of 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 inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`
- **fillval** – A value used to fill outliers

The function `cvWarpPerspective` transforms the source image using the specified matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV\_WARP\_INVERSE\_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

Note that the function can not operate in-place. For a sparse set of points use the *PerspectiveTransform* function from CxCore.

2.4 Miscellaneous Image Transformations

AdaptiveThreshold

void **cvAdaptiveThreshold**(const CvArr* *src*, CvArr* *dst*, double *maxValue*, int *adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C*, int *thresholdType=CV_THRESH_BINARY*, int *blockSize=3*, double *param1=5*)

Applies an adaptive threshold to an array.

Parameters

- **src** – Source image
- **dst** – Destination image
- **maxValue** – 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)
- **thresholdType** – Thresholding type; must be one of
  - **CV_THRESH_BINARY** xxx
  - **CV_THRESH_BINARY_INV** xxx
- **blockSize** – The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on
- **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 the mean or weighted mean (see the discussion), though it may be negative

The function transforms a grayscale image to a binary image according to the formulas:

- **CV_THRESH_BINARY**

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

- **CV_THRESH_BINARY_INV**

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

For the method `CV_ADAPTIVE_THRESH_MEAN_C` it is the mean of a $\texttt{blockSize} \times \texttt{blockSize}$ pixel neighborhood, minus `param1` .

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is the weighted sum (gaussian) of a $\texttt{blockSize} \times \texttt{blockSize}$ pixel neighborhood, minus `param1` .

CvtColor

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

Converts an 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. The number of channels may be different
- **code** – Color conversion operation that can be specifed using `CV_*src_color_space* 2 *dst_color_space*` constants (see below)

The function converts the input image from one color space to another. The function ignores the `colorModel` and `channelSeq` fields of the `IplImage` header, so the source image color space should be specified correctly (including order of the channels in the case of RGB space. For example, BGR means 24-bit format with $B_0, G_0, R_0, B_1, G_1, R_1, ...$ layout whereas RGB means 24-format with $R_0, G_0, B_0, R_1, G_1, B_1, ...$ layout).

The conventional range for R,G,B channel values is:

- 0 to 255 for 8-bit images
- 0 to 65535 for 16-bit images and
- 0 to 1 for floating-point images.

Of course, in the case of linear transformations the range can be specific, but in order to get correct results in the case of non-linear transformations, the input image should be scaled.

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

  ```
  cvCvtColor(src ,bwsrc, CV_RGB2GRAY)
  ```

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB`):

  $$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

  $$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X , Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB $\leftrightarrow$ YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB`)

  $$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

Y, Cr and Cb cover the whole value range.

- RGB ↔ HSV (`CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V = R \\ 120 + 60(B - R)/S & \text{if } V = G \\ 240 + 60(R - G)/S & \text{if } V = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$ On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

  - 8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

  - 16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

  - **32-bit images** H, S, V are left as is

- RGB ↔ HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R, G, B)$$

$$V_{min} \leftarrow min(R, G, B)$$

$$L \leftarrow \frac{V_{max} + V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max}-V_{min}}{V_{max}+V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max}-V_{min}}{2-(V_{max}+V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V_{max} = R \\ 120 + 60(B-R)/S & \text{if } V_{max} = G \\ 240 + 60(R-G)/S & \text{if } V_{max} = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$ On output $0 \leq L \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

– 8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

– 16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

– **32-bit images** H, S, V are left as is

- RGB $\leftrightarrow$ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t <= 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \leq L \leq 100$, $-127 \leq a \leq 127$, $-127 \leq b \leq 127$ The values are then converted to the destination data type:

– 8-bit images

$$L \leftarrow L * 255/100, a \leftarrow a + 128, b \leftarrow b + 128$$

– **16-bit images** currently not supported

– **32-bit images** L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116 Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3 Y & \text{for } Y <= 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

– 8-bit images

$$L \leftarrow 255/100L, u \leftarrow 255/354(u + 134), v \leftarrow 255/256(v + 140)$$

– **16-bit images** currently not supported

– **32-bit images** L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Ford98 at the Charles Poynton site.

- Bayer $\rightarrow$ RGB (`CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerGB2RGB, CV_BayerRG2RGB, CV_BayerGR2RGB`) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{array}{ccccc} R & G & R & G & R \\ G & \boxed{\mathrm{B}} & \boxed{\mathrm{G}} & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{array}$$

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 C_1 and C_2 in the conversion constants `CV_Bayer` C_1C_2 `2BGR` and `CV_Bayer` C_1C_2 `2RGB` 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.

DistTransform

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 the distance to the closest zero pixel for all non-zero pixels of the source image.

Parameters

- **src** – 8-bit, single-channel (binary) source image
- **dst** – Output image with calculated distances (32-bit floating-point, single-channel)
- **distance_type** – Type of distance; can be `CV_DIST_L1, CV_DIST_L2, CV_DIST_C` or `CV_DIST_USER`
- **mask_size** – Size of the distance transform mask; can be 3 or 5. in the case of `CV_DIST_L1` or `CV_DIST_C` the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 yet it is faster
- **mask** – User-defined mask in the case of a user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in the case ofa 3×3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight's move cost) in the case of a 5×5 mask
- **labels** – The optional output 2d array of integer type labels, the same size as `src` and `dst`

The function calculates the approximated distance from every binary image pixel to the nearest zero pixel. For zero pixels the function sets the zero distance, for others it finds the shortest path consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all of 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` (Euclidian distance) the distance can be calculated only with some relative error (a 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 the distance field (black (0) pixel is in the middle of white square) in the case of a user-defined distance:

User-defined 3×3 mask (a=1, b=1.5)

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

User-defined 5×5 mask (a=1, b=1.5, c=2)

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

Typically, for a fast, coarse distance estimation `CV_DIST_L2` , a 3×3 mask is used, and for a more accurate distance estimation `CV_DIST_L2` , a 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.

CvConnectedComp

CvConnectedComp

```
typedef struct CvConnectedComp
{
    double area;    /* area of the segmented component */
    CvScalar value; /* average color of the connected component */
    CvRect rect;    /* ROI of the segmented component */
    CvSeq* contour; /* optional component boundary
                      (the contour might have child contours corresponding to the holes) */
} CvConnectedComp;
```

FloodFill

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 the given color.

Parameters

- **image** – Input 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the `CV_FLOODFILL_MASK_ONLY` flag is set (see below)
- **seed_point** – The starting point
- **new_val** – New value of the repainted domain pixels
- **lo_diff** – Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value
- **up_diff** – Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value
- **comp** – Pointer to the structure that the function fills with the information about the repainted domain. Note that the function does not fill `comp->contour` field. The boundary of the filled component can be retrieved from the output mask image using *FindContours*

- **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 a combination of the following flags:
  - **CV_FLOODFILL_FIXED_RANGE** if set, the difference between the current pixel and seed pixel is considered, otherwise the 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 fills the mask (that must be non-NULL in this case)
- **mask** – Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller than `image` . If not NULL, the function uses and updates the mask, so the user takes responsibility of initializing the `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. 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 the mask is larger than the filled image, a pixel in `mask` that corresponds to (x, y) pixel in `image` will have coordinates $(x + 1, y + 1)$

The function 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:

- grayscale image, floating range

$$src(x', y') - \texttt{lo\_diff} <= src(x, y) <= src(x', y') + \texttt{up\_diff}$$

- grayscale image, fixed range

$$src(seed.x, seed.y) - \texttt{lo\_diff} <= src(x, y) <= src(seed.x, seed.y) + \texttt{up\_diff}$$

- color image, floating range

$$src(x', y')_r - \texttt{lo\_diff}_r <= src(x, y)_r <= src(x', y')_r + \texttt{up\_diff}_r$$

$$src(x', y')_g - \texttt{lo\_diff}_g <= src(x, y)_g <= src(x', y')_g + \texttt{up\_diff}_g$$

$$src(x', y')_b - \texttt{lo\_diff}_b <= src(x, y)_b <= src(x', y')_b + \texttt{up\_diff}_b$$

- color image, fixed range

$$src(seed.x, seed.y)_r - \texttt{lo\_diff}_r <= src(x, y)_r <= src(seed.x, seed.y)_r + \texttt{up\_diff}_r$$

$$src(seed.x, seed.y)_g - \texttt{lo\_diff}_g <= src(x, y)_g <= src(seed.x, seed.y)_g + \texttt{up\_diff}_g$$

$$src(seed.x, seed.y)_b - \texttt{lo\_diff}_b <= src(x, y)_b <= src(seed.x, seed.y)_b + \texttt{up\_diff}_b$$

where $src(x', y')$ is the 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 the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

Inpaint

void **cvInpaint** (const CvArr* *src*, const CvArr* *mask*, CvArr* *dst*, double *inpaintRadius*, int *flags*)
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.
- **inpaintRadius** – The radius of circular neighborhood of each point inpainted that is considered by the algorithm.
- **flags** – The inpainting method, one of the following:
  - **CV_INPAINT_NS** Navier-Stokes based method.
  - **CV_INPAINT_TELEA** The method by Alexandru Telea Telea04

The function 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.

Integral

void **cvIntegral** (const CvArr* *image*, CvArr* *sum*, CvArr* *sqsum=NULL*, CvArr* *tiltedSum=NULL*)
Calculates the integral of an image.

Parameters

- **image** – The source image, $W \times H$, 8-bit or floating-point (32f or 64f)
- **sum** – The integral image, $(W+1) \times (H+1)$, 32-bit integer or double precision floating-point (64f)
- **sqsum** – The integral image for squared pixel values, $(W+1) \times (H+1)$, double precision floating-point (64f)
- **tiltedSum** – The integral for the image rotated by 45 degrees, $(W+1) \times (H+1)$, the same data type as `sum`

The function calculates one or more integral images for the source image as following:

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

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

$$\texttt{tiltedSum} (X,Y) = \sum _{y<Y,abs(x-X+1) \leq Y-y-1} \texttt{image} (x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum _{x_1<=x<x_2, \, y_1<=y<y_2} = \texttt{sum} (x_2,y_2)- \texttt{sum} (x_1,y_2)- \texttt{sum} (x_2,y_1)+ \texttt{sum} (x_1,y_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

PyrMeanShiftFiltering

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

$$(x, y) : X - \texttt{sp} \le x \le X + \texttt{sp}, Y - \texttt{sp} \le y \le Y + \texttt{sp}, ||(R, G, B) - (r, g, b)|| \le \texttt{sr}$$

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 $\texttt{max\_level} > 0$, the gaussian pyramid of $\texttt{max\_level} + 1$ 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 ($> \texttt{sr}$) 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 $\texttt{max\_level} == 0$).

PyrSegmentation

void **cvPyrSegmentation**(IplImage* *src*, IplImage* *dst*, CvMemStorage* *storage*, CvSeq** *comp*, int *level*, double *threshold1*, double *threshold2*)

Implements 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 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 $p(c(A), c(B)) < threshold2$. If the input image has only one channel, then $p(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then

$$p(c^1, c^2) = 0.30(c_r^1 - c_r^2) + 0.59(c_g^1 - c_g^2) + 0.11(c_b^1 - c_b^2).$$

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.

Threshold

double **cvThreshold**(const CvArr* *src*, CvArr* *dst*, double *threshold*, double *maxValue*, int *thresholdType*)

Applies a fixed-level threshold to array elements.

Parameters

- **src** – Source array (single-channel, 8-bit or 32-bit floating point)
- **dst** – Destination array; must be either the same type as `src` or 8-bit
- **threshold** – Threshold value
- **maxValue** – Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types
- **thresholdType** – Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (*CmpS* 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 that the function supports that are determined by `thresholdType` :

- **CV_THRESH_BINARY**

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxValue} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

- **CV_THRESH_BINARY_INV**

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

- **CV_THRESH_TRUNC**

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

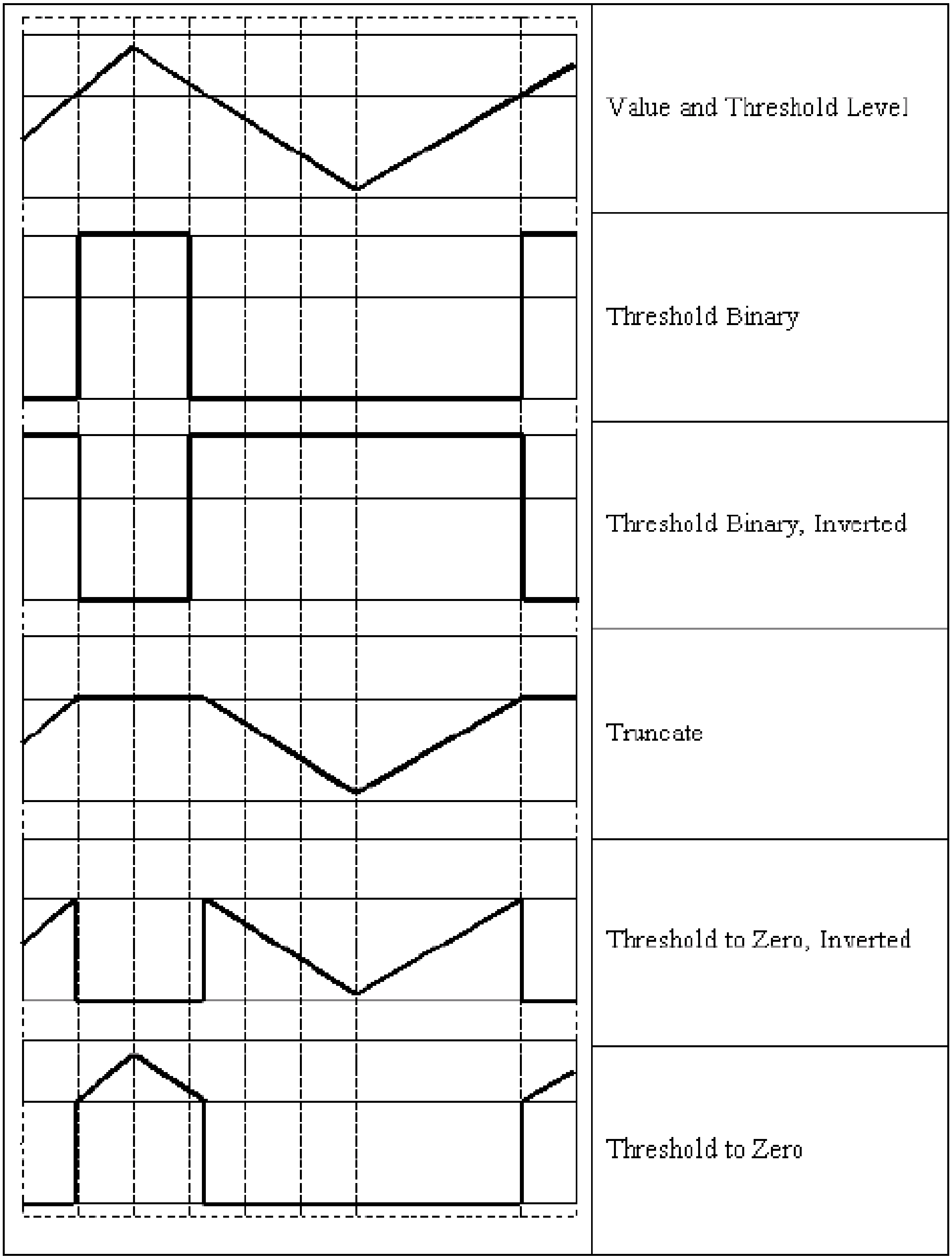
- **CV_THRESH_TOZERO**

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

- **CV_THRESH_TOZERO_INV**

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

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's algorithm and uses it instead of the specified `thresh` . The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.


Value and Threshold Level
Threshold Binary
Threshold Binary, Inverted
Truncate
Threshold to Zero, Inverted
Threshold to Zero

2.5 Structural Analysis and Shape Descriptors

ApproxChains

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 a 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 *FindContours*)
- **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 using the `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 *FindContours* with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via the `v_next` or `h_next` fields of the returned structure.

ApproxPoly

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

Approximates polygonal curve(s) with the specified precision.

Parameters

- **src_seq** – Sequence of an array of points
- **header_size** – Header size of the approximated curve[s]
- **storage** – Container for the 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 the Douglas-Peucker algorithm
- **parameter** – Method-specific parameter; in the case of `CV_POLY_APPROX_DP` it is a desired approximation accuracy
- **parameter2** – If case if `src_seq` is a sequence, the parameter determines whether the single sequence should be approximated or all sequences on the same level or below `src_seq` (see *FindContours* for description of hierarchical contour structures). If `src_seq` is an array CvMat* of points, the parameter specifies whether the curve is closed (`parameter2` !=0) or not (`parameter2` =0)

The function approximates one or more curves and returns the approximation result[s]. In the case of multiple curves, the resultant tree will have the same structure as the input one (1:1 correspondence).

ArcLength

double **cvArcLength** (const void* *curve*, CvSlice *slice=CV_WHOLE_SEQ*, int *isClosed=-1*)
Calculates the contour perimeter or the 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
- **isClosed** – Indicates whether the curve is closed or not. There are 3 cases:
  - `isClosed` $= 0$ the curve is assumed to be unclosed.
  - `isClosed` > 0 the curve is assumed to be closed.
  - `isClosed` < 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 calculates the length or curve as the sum of lengths of segments between subsequent points

BoundingRect

CvRect **cvBoundingRect** (CvArr* *points*, int *update=0*)
Calculates the up-right bounding rectangle of a point set.

Parameters

- **points** – 2D point set, either a sequence or vector (`CvMat`) of points
- **update** – The update flag. See below.

The function returns the up-right bounding rectangle for a 2d point set. Here is the list of possible combination of the flag values and type of `points` :

update	points	action
0	`CvContour*`	the bounding rectangle is not calculated, but it is taken from `rect` field of the contour header.
1	`CvContour*`	the bounding rectangle is calculated and written to `rect` field of the contour header.
0	`CvSeq*` or `CvMat*`	the bounding rectangle is calculated and returned.
1	`CvSeq*` or `CvMat*`	runtime error is raised.

BoxPoints

void **cvBoxPoints** (CvBox2D *box*, CvPoint2D32f *pt[4]*)
Finds the box vertices.

Parameters

- **box** – Box
- **points** – Array of vertices

The function calculates the vertices of the input 2d box.

Here is the function code:

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

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

CalcPGH

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

Calculates a pair-wise geometrical histogram for a contour.

Parameters

- **contour** – Input contour. Currently, only integer point coordinates are allowed
- **hist** – Calculated histogram; must be two-dimensional

The function calculates a 2D pair-wise geometrical histogram (PGH), described in *Iivarinen97* for the contour. The algorithm considers every pair of 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 the *Iivarninen97* definition). The histogram can be used for contour matching.

CalcEMD2

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 the "minimal work" distance between two weighted point configurations.

Parameters

- **signature1** – First signature, a $\texttt{size1} \times \texttt{dims} + 1$ 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-supplied distance function. It takes coordinates of two points and returns the distance between the points `` typedef float (*CvDistanceFunction)(const float* f1, const float* f2, void* userdata);``
- **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: $\texttt{flow}_{i,j}$ 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 if the signatures consist of weights only (i.e. the signature matrices have a single column). The 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 computes the earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the applications described in *RubnerSept98* is multi-dimensional histogram comparison for image retrieval. EMD is a a transportation problem that is solved using some modification of a simplex algorithm, thus the complexity is exponential in the worst case, though, on average it is much faster. In the 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.

CheckContourConvexity

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

Tests contour convexity.

Parameters

- **contour** – Tested contour (sequence or array of points)

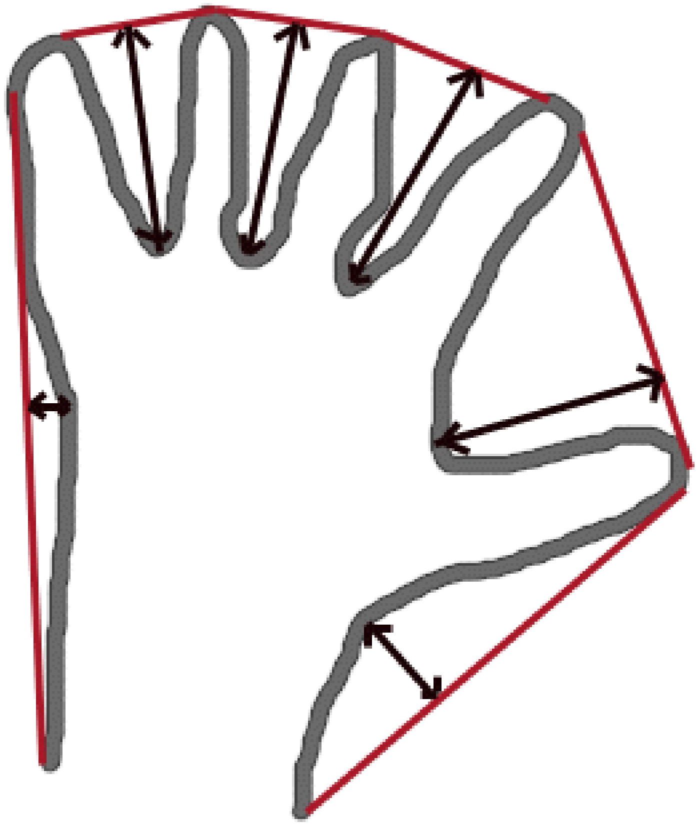
The function tests whether the input contour is convex or not. The contour must be simple, without self-intersections.

CvConvexityDefect

CvConvexityDefect

Structure describing a single contour convexity defect.

```
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;
```



ContourArea

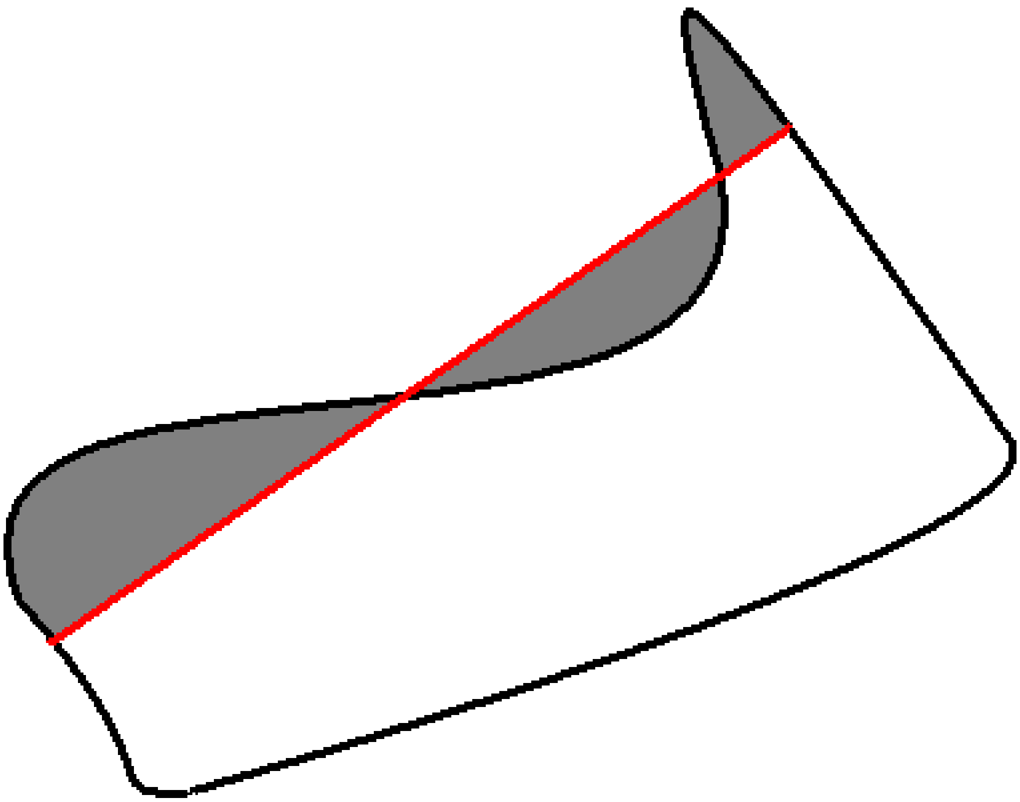
double **cvContourArea** (const CvArr* *contour*, CvSlice *slice=CV_WHOLE_SEQ*)

Calculates the area of a whole contour or a contour section.

Parameters

- **contour** – Contour (sequence or array of vertices)
- **slice** – Starting and ending points of the contour section of interest, by default, the area of the whole contour is calculated

The function calculates the area of a whole contour or a 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:



Orientation of the contour affects the area sign, thus the function may return a *negative* result. Use the `fabs()` function from C runtime to get the absolute value of the area.

ContourFromContourTree

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

Restores a contour from the tree.

Parameters

- **tree** – Contour tree
- **storage** – Container for the reconstructed contour
- **criteria** – Criteria, where to stop reconstruction

The function 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 an approximated contour. The function returns the reconstructed contour.

ConvexHull2

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

Finds the convex hull of a point set.

Parameters

- **points** – Sequence or array of 2D points with 32-bit integer or floating-point coordinates

- **storage** – The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is an array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified as to truncate the array down to the hull size. If `storage` is NULL then the convex hull will be stored in the same storage as the input sequence
- **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 `storage` is an array, or pointers if `storage` is memory storage

The function finds the convex hull of a 2D point set using Sklansky's algorithm. If `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. If `storage` is a CvMat, the function returns NULL.

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()
        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()
            pt0.y = rand()
            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()
            pt0.y = rand()
            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
    }
}
```

ConvexityDefects

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

Finds the convexity defects of a contour.

Parameters

- **contour** – Input contour
- **convexhull** – Convex hull obtained using *ConvexHull2* that should contain pointers or indices to the contour points, not the hull points themselves (the `return_points` parameter in *ConvexHull2* should be 0)
- **storage** – Container for the output sequence of convexity defects. If it is NULL, the contour or hull (in that order) storage is used

The function finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect structures.

CreateContourTree

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

Creates a hierarchical representation of a contour.

Parameters

- **contour** – Input contour
- **storage** – Container for output tree
- **threshold** – Approximation accuracy

The function creates a 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 a full binary tree representation. If the threshold is greater than 0, the function creates a 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.

EndFindContours

CvSeq* **cvEndFindContours** (CvContourScanner* *scanner*)

Finishes the scanning process.

Parameters

- **scanner** – Pointer to the contour scanner

The function finishes the scanning process and returns a pointer to the first contour on the highest level.

FindContours

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 the contours in a binary image.

Parameters

- **image** – The source, an 8-bit single channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary` . To get such a binary image from grayscale, one may use *Threshold* , *AdaptiveThreshold* or *Canny* . The function modifies the source image's 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, $\geq$ `sizeof(CvChain)` if `method` $=$ `CV_CHAIN_CODE` , and $\geq$ `sizeof(CvContour)` otherwise
- **mode** – Retrieval mode
  - **CV_RETR_EXTERNAL** retrives only the extreme outer contours
  - **CV_RETR_LIST** retrieves all of the contours and puts them in the list
  - **CV_RETR_CCOMP** retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes
  - **CV_RETR_TREE** retrieves all of the contours and reconstructs the full hierarchy of nested contours
- **method** – Approximation method (for all the modes, except `CV_LINK_RUNS` , which uses built-in approximation)
  - **CV_CHAIN_CODE** outputs contours in the Freeman chain code. All other methods output polygons (sequences of vertices)
  - **CV_CHAIN_APPROX_NONE** translates all of the points from the chain code into points
  - **CV_CHAIN_APPROX_SIMPLE** compresses horizontal, vertical, and diagonal segments and leaves only their end points
  - **CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS** applies one of the flavors of the Teh-Chin chain approximation algorithm.
  - **CV_LINK_RUNS** uses a completely different contour retrieval algorithm by linking horizontal segments of 1's. Only the `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 retrieves contours from the binary image using the algorithm Suzuki85 . The contours are a useful tool for shape analysis and object detection and recognition.

The function 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 a pointer to the first outermost contour or `NULL` if no contours are detected (if the image is completely black). Other contours may be reached from `first_contour` using the `h_next` and `v_next` links. The sample in the *DrawContours* 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 the OpenCV sample directory.

Note: the source `image` is modified by this function.

FindNextContour

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

Finds the next contour in the image.

Parameters

- **scanner** – Contour scanner initialized by *StartFindContours*

The function locates and retrieves the next contour in the image and returns a pointer to it. The function returns NULL if there are no more contours.

FitEllipse2

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

Fits an ellipse around a set of 2D points.

Parameters

- **points** – Sequence or array of points

The function calculates the ellipse that fits best (in least-squares sense) around a set of 2D points. The meaning of the returned structure fields is similar to those in *Ellipse* except that `size` stores the full lengths of the ellipse axises, not half-lengths.

FitLine

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

Fits a line to a 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** – Sufficient accuracy for the radius (distance between the coordinate origin and the line). 0.01 is a good default value.
- **aeps** – Sufficient accuracy for the angle. 0.01 is a good default value.
- **line** – The output line parameters. In the case of a 2d fitting, it is an 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 the case of a 3D fitting it is an 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 fits a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i th point and the line and $\rho(r)$ is a distance function, one of:

- dist_type=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

- dist_type=CV_DIST_L1

$$\rho(r) = r$$

- dist_type=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

- dist_type=CV_DIST_FAIR

$$\rho(r) = C^2 \cdot \left(\frac{r}{C} - \log \left(1 + \frac{r}{C} \right) \right) \quad \text{where} \quad C = 1.3998$$

- dist_type=CV_DIST_WELSCH

$$\rho(r) = \frac{C^2}{2} \cdot \left(1 - \exp \left(- \left(\frac{r}{C} \right)^2 \right) \right) \quad \text{where} \quad C = 2.9846$$

- dist_type=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

GetCentralMoment

double **cvGetCentralMoment** (CvMoments* *moments*, int *x_order*, int *y_order*)

Retrieves the central moment from the 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 retrieves the central moment, which in the case of image moments is defined as:

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

where x_c, y_c are the coordinates of the gravity center:

$$x_c = \frac{M_{10}}{M_{00}}, y_c = \frac{M_{01}}{M_{00}}$$

GetHuMoments

void **cvGetHuMoments** (const CvMoments* *moments*, CvHuMoments* *hu*)

Calculates the seven Hu invariants.

Parameters

- **moments** – The input moments, computed with *Moments*
- **hu** – The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment , that are defined as:

$$\begin{array}{l}
hu_1 = \eta_{20} + \eta_{02} \\
hu_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
hu_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
hu_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
hu_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
hu_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
hu_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{array}$$

where η_{ji} denote the normalized central moments.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

GetNormalizedCentralMoment

double **cvGetNormalizedCentralMoment** (CvMoments* *moments*, int *x_order*, int *y_order*)

Retrieves the normalized central moment from the moment state structure.

Parameters

- **moments** – Pointer to the moment state structure
- **x_order** – x order of the retrieved moment, $\texttt{x\_order} >= 0$
- **y_order** – y order of the retrieved moment, $\texttt{y\_order} >= 0$ and $\texttt{x\_order} + \texttt{y\_order} <= 3$

The function retrieves the normalized central moment:

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

GetSpatialMoment

double **cvGetSpatialMoment** (CvMoments* *moments*, int *x_order*, int *y_order*)

Retrieves the spatial moment from the moment state structure.

Parameters

- **moments** – The moment state, calculated by *Moments*
- **x_order** – x order of the retrieved moment, $\texttt{x\_order} >= 0$
- **y_order** – y order of the retrieved moment, $\texttt{y\_order} >= 0$ and $\texttt{x\_order} + \texttt{y\_order} <= 3$

The function retrieves the spatial moment, which in the case of image moments is defined as:

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

where $I(x,y)$ is the intensity of the pixel (x,y) .

MatchContourTrees

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 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 a certain level the difference between contours becomes less than `threshold` , the reconstruction process is interrupted and the current difference is returned.

MatchShapes

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; `CV_CONTOURS_MATCH_I1` , `CV_CONTOURS_MATCH_I2`

  or `CV_CONTOURS_MATCH_I3`
- **parameter** – Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu moments (see *GetHuMoments*) (A is `object1` , B is `object2`):

- method=CV_CONTOURS_MATCH_I1

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

- method=CV_CONTOURS_MATCH_I2

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

- method=CV_CONTOURS_MATCH_I3

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

where

$$m_i^A = sign(h_i^A) \cdot \log h_i^A m_i^B = sign(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

MinAreaRect2

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

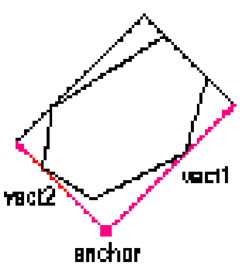
Finds the circumscribed rectangle of minimal area for a given 2D point set.

Parameters

- **points** – Sequence or array of points
- **storage** – Optional temporary memory storage

The function finds a circumscribed rectangle of the minimal area for a 2D point set by building a convex hull for the set and applying the rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour



MinEnclosingCircle

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

Finds the circumscribed circle of minimal area for a 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 finds the minimal circumscribed circle for a 2D point set using an iterative algorithm. It returns nonzero if the resultant circle contains all the input points and zero otherwise (i.e. the algorithm failed).

Moments

void **cvMoments** (const CvArr* *arr*, CvMoments* *moments*, int *binary=0*)

Calculates all of the moments up to the 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's state structure
- **binary** – (For images only) If the flag is non-zero, all of the zero pixel values are treated as zeroes, and all of the others are treated as 1's

The function calculates spatial and central moments up to the third order and writes them to `moments` . The moments may then be used then to calculate the gravity center of the shape, its area, main axises and various shape characteristics including 7 Hu invariants.

PointPolygonTest

double **cvPointPolygonTest**(const CvArr* *contour*, CvPoint2D32f *pt*, int *measure_dist*)

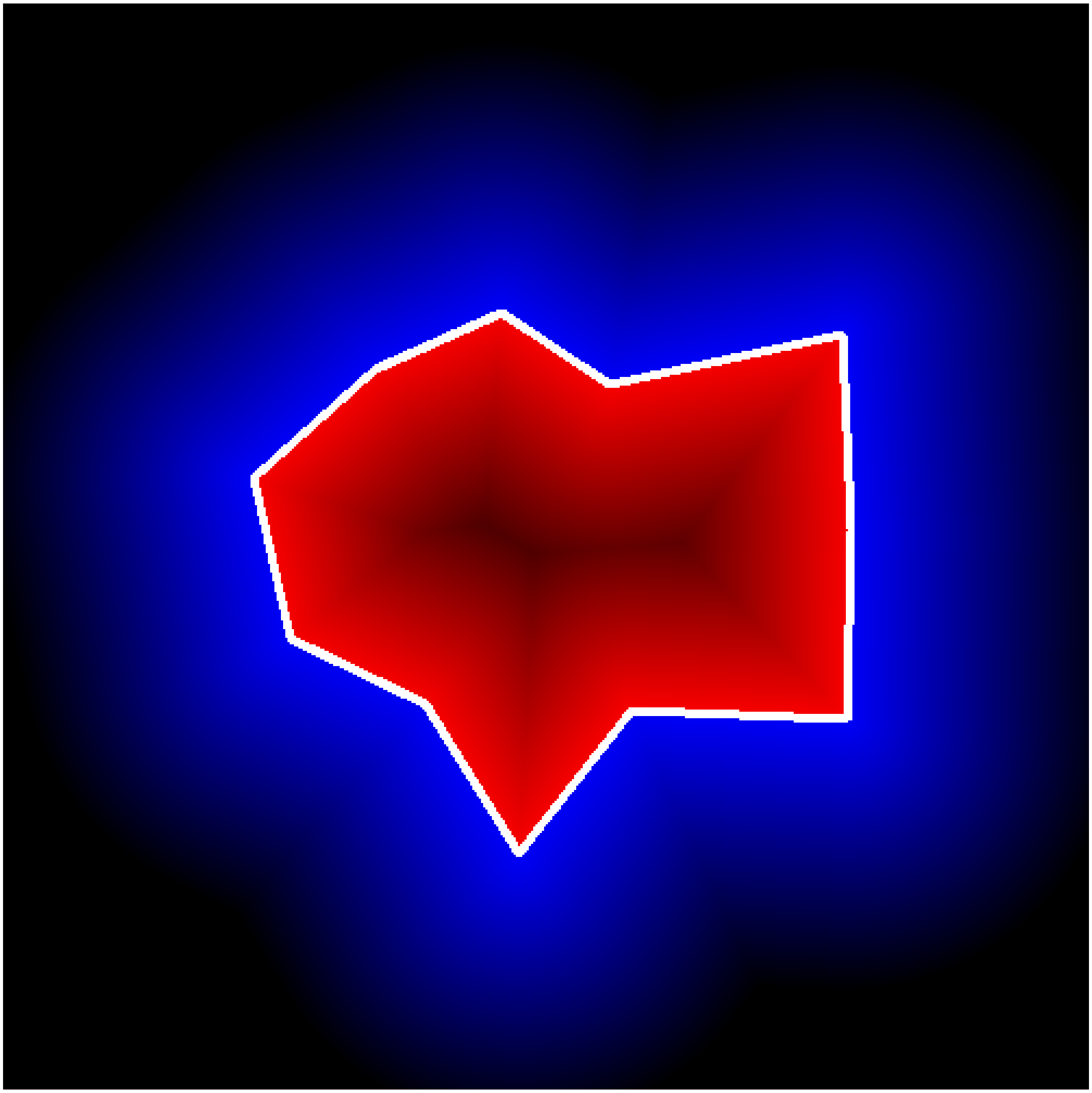
Point in contour test.

Parameters

- **contour** – Input contour
- **pt** – The point tested against the contour
- **measure_dist** – If it is non-zero, the function estimates the distance from the point to the nearest contour edge

The function determines whether the point is inside a contour, outside, or lies on an edge (or coinsides with a vertex). It returns positive, negative or zero value, correspondingly. When $\texttt{measure\_dist} = 0$, the return value is +1, -1 and 0, respectively. When $\texttt{measure\_dist} \neq 0$, 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.



PointSeqFromMat

CvSeq* **cvPointSeqFromMat** (int *seq_kind*, const CvArr* *mat*, CvContour* *contour_header*, CvSeqBlock* *block*)

Initializes a 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 a 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 initializes a sequence header to create a "virtual" sequence in 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 can be added to the sequence, but some may be removed. The function is a specialized variant of *MakeSeqHeaderForArray* and uses the latter internally. It returns a pointer to the initialized contour header. Note that the bounding rectangle (field `rect` of `CvContour` strucuture) is not initialized by the function. If you need one, use *BoundingRect* .

Here is a 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));
```

ReadChainPoint

CvPoint **cvReadChainPoint** (CvChainPtReader* *reader*)

Gets the next chain point.

Parameters

- **reader** – Chain reader state

The function returns the current chain point and updates the reader position.

StartFindContours

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 the contour scanning process.

Parameters

- **image** – The 8-bit, single channel, binary source 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 *FindContours*
- **method** – Approximation method. It has the same meaning in *FindContours* , but `CV_LINK_RUNS` can not be used here
- **offset** – ROI offset; see *FindContours*

The function initializes and returns a pointer to the contour scanner. The scanner is used in *FindNextContour* to retrieve the rest of the contours.

StartReadChainPoints

void **cvStartReadChainPoints**(CvChain* *chain*, CvChainPtReader* *reader*)
Initializes the chain reader.

The function initializes a special reader.

SubstituteContour

void **cvSubstituteContour**(CvContourScanner *scanner*, CvSeq* *new_contour*)
Replaces a retrieved contour.

Parameters

- **scanner** – Contour scanner initialized by *StartFindContours*
- **new_contour** – Substituting contour

The function replaces the retrieved contour, that was returned from the preceding call of *FindNextContour* 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` is `NULL` , the retrieved contour is not included in the resulting structure, nor are any of its children that might be added to this structure later.

2.6 Planar Subdivisions

CvSubdiv2D

CvSubdiv2D

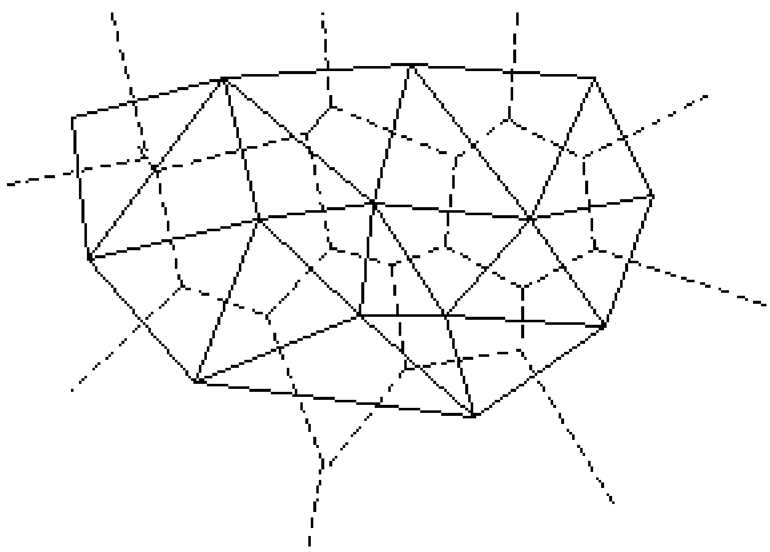
Planar subdivision.

```
#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;
```

Planar subdivision is the subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on a 2d point set, where the points are linked together and form a planar graph, which, together with a few edges connecting the exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists a dual subdivision in which facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called a virtual point below) of the dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dotted lines.



OpenCV subdivides a 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 a Voronoi diagram of the input 2d point set. The subdivisions can be used for the 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) and so forth.

CvQuadEdge2D

CvQuadEdge2D

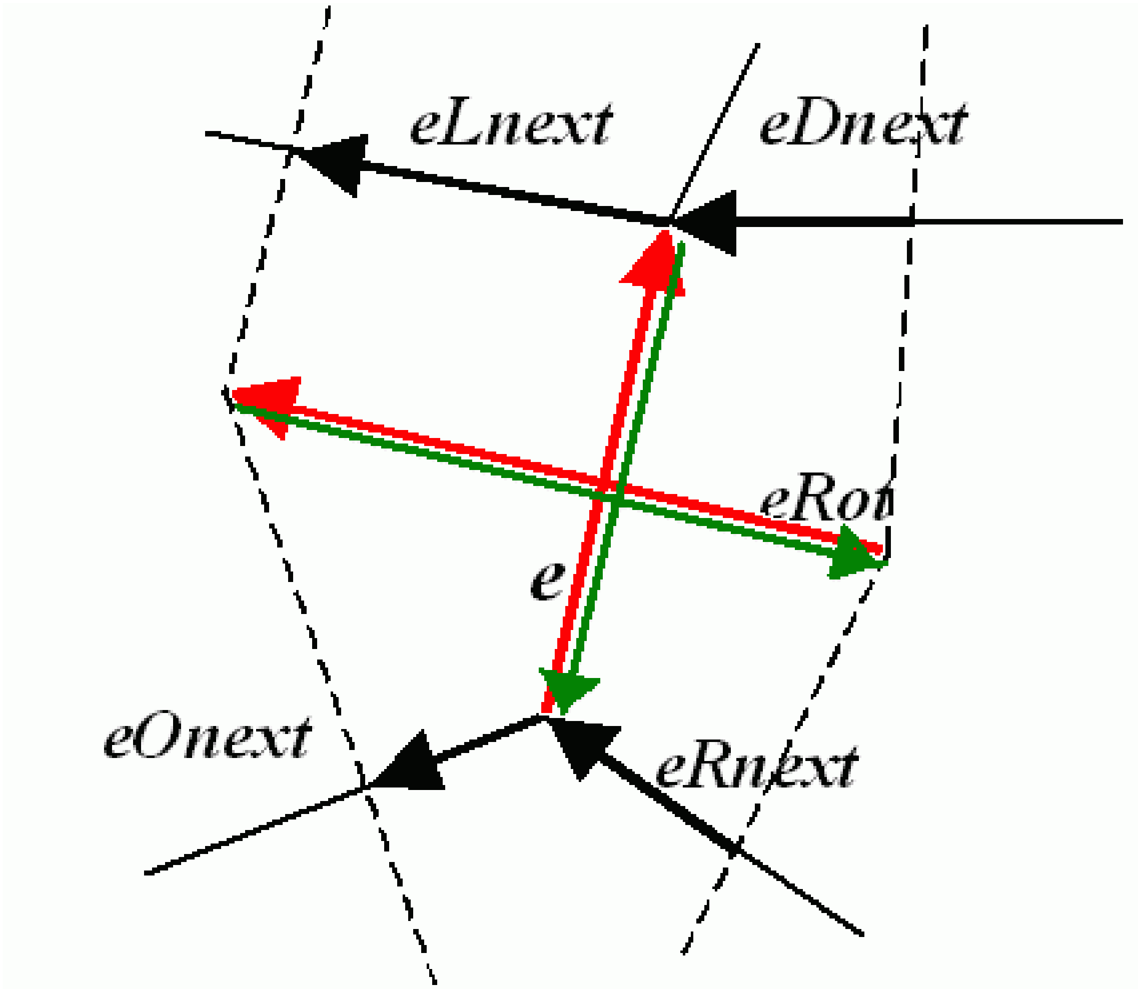
Quad-edge of planar subdivision.

```
/* 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 containing four edges (e, eRot, reversed e and reversed eRot):



CvSubdiv2DPoint

CvSubdiv2DPoint

Point of original or dual subdivision.

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

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

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

- **id** This integer can be used to index auxillary data associated with each vertex of the planar subdivision

CalcSubdivVoronoi2D

void **cvCalcSubdivVoronoi2D**(CvSubdiv2D* *subdiv*)

Calculates the coordinates of Voronoi diagram cells.

Parameters

- **subdiv** – Delaunay subdivision, in which all the points are already added

The function calculates the coordinates of virtual points. All virtual points corresponding to some vertex of the original subdivision form (when connected together) a boundary of the Voronoi cell at that point.

ClearSubdivVoronoi2D

void **cvClearSubdivVoronoi2D**(CvSubdiv2D* *subdiv*)

Removes all virtual points.

Parameters

- **subdiv** – Delaunay subdivision

The function removes all of the virtual points. It is called internally in *CalcSubdivVoronoi2D* if the subdivision was modified after previous call to the function.

CreateSubdivDelaunay2D

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

Creates an empty Delaunay triangulation.

Parameters

- **rect** – Rectangle that includes all of the 2d points that are to be added to the subdivision
- **storage** – Container for subdivision

The function creates an empty Delaunay subdivision, where 2d points can be added using the function *SubdivDelaunay2DInsert* . All of the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

Note that the triangulation is a single large triangle that covers the given rectangle. Hence the three vertices of this triangle are outside the rectangle `rect` .

FindNearestPoint2D

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

Finds the closest subdivision vertex to the given point.

Parameters

- **subdiv** – Delaunay or another subdivision
- **pt** – Input point

The function is another function that locates the input point within the subdivision. It finds the 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 *Subdiv2DLocate*) is used as a starting point. The function returns a pointer to the found subdivision vertex.

Subdiv2DEdgeDst

CvSubdiv2DPoint* **cvSubdiv2DEdgeDst** (CvSubdiv2DEdge *edge*)

Returns the edge destination.

Parameters

- **edge** – Subdivision edge (not a quad-edge)

The function 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 the function *CalcSubdivVoronoi2D* .

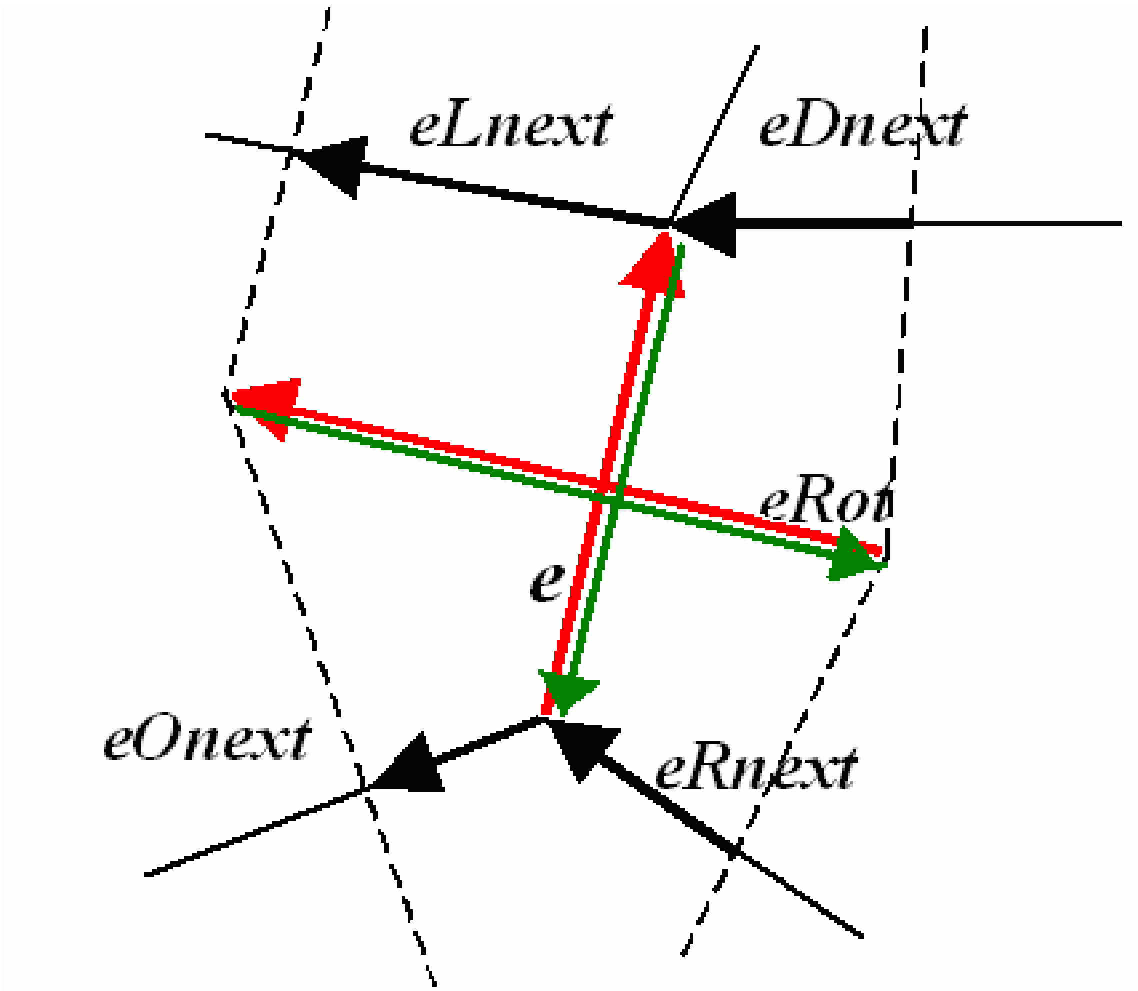
Subdiv2DGetEdge

CvSubdiv2DEdge **cvSubdiv2DGetEdge** (CvSubdiv2DEdge *edge*, CvNextEdgeType *type*)

Returns one of the edges related to the given edge.

Parameters

- **edge** – Subdivision edge (not a quad-edge)
- **type** – Specifies which of the related edges to return, one of the following:
  - **CV_NEXT_AROUND_ORG** next around the edge origin (`eOnext` on the picture below 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 returns one of the edges related to the input edge.

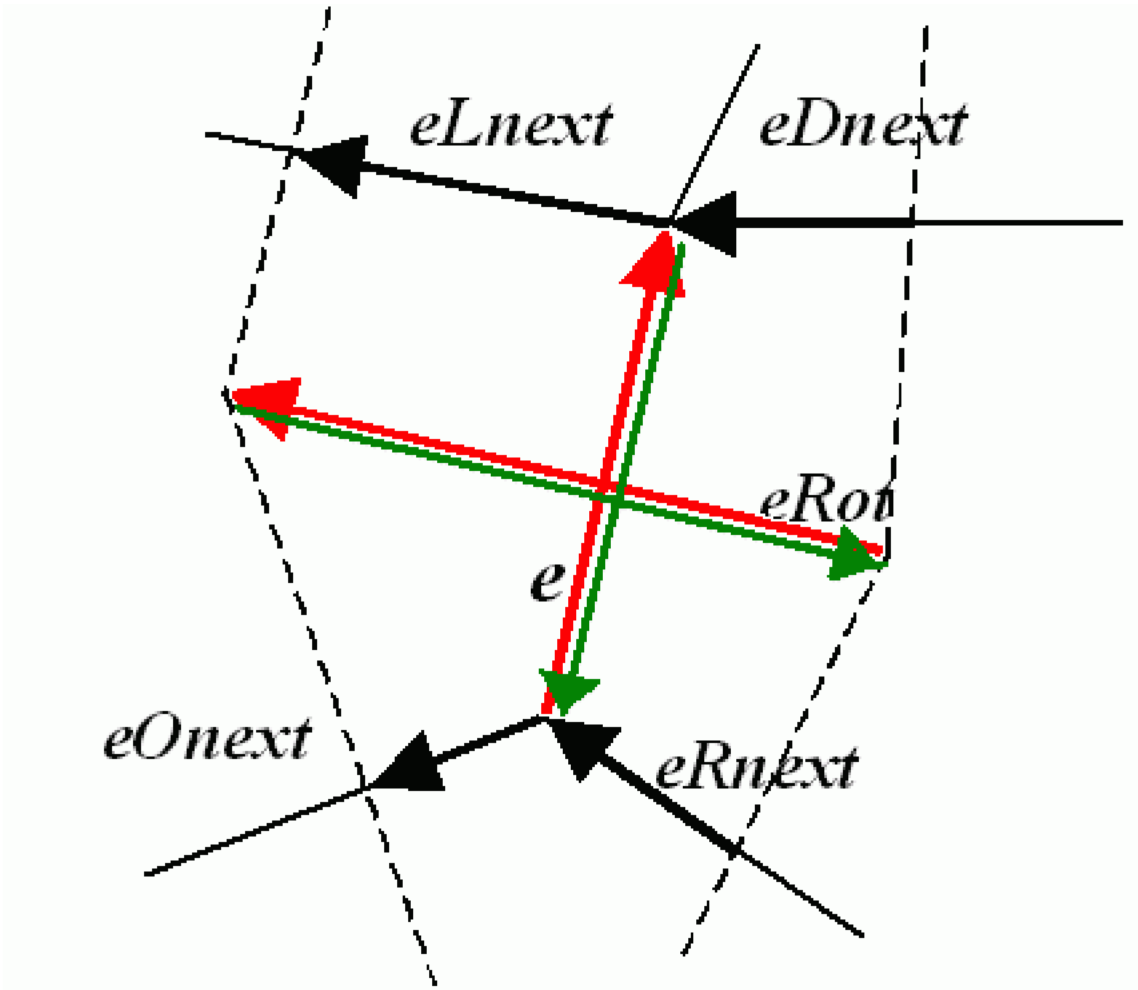
Subdiv2DNextEdge

CvSubdiv2DEdge **cvSubdiv2DNextEdge** (CvSubdiv2DEdge *edge*)

Returns next edge around the edge origin

Parameters

- **edge** – Subdivision edge (not a quad-edge)



The function returns the next edge around the edge origin: `eOnext` on the picture above if `e` is the input edge)

Subdiv2DLocate

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

Returns the location of a point within a 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 coinsides with

The function locates the input point within the subdivision. There are 5 cases:

- The point falls into some facet. The function returns `CV_PTLOC_INSIDE` and `*edge` will contain one of edges of the facet.
- The point falls onto the edge. The function returns `CV_PTLOC_ON_EDGE` and `*edge` will contain this edge.

- The point coincides with one of the subdivision vertices. The function returns `CV_PTLOC_VERTEX` and `*vertex` will contain a pointer to the vertex.
- The point is outside the subdivsion reference rectangle. The function returns `CV_PTLOC_OUTSIDE_RECT` and no pointers are filled.
- One of input arguments is invalid. A runtime error is raised or, if silent or "parent" error processing mode is selected, `CV_PTLOC_ERROR` is returnd.

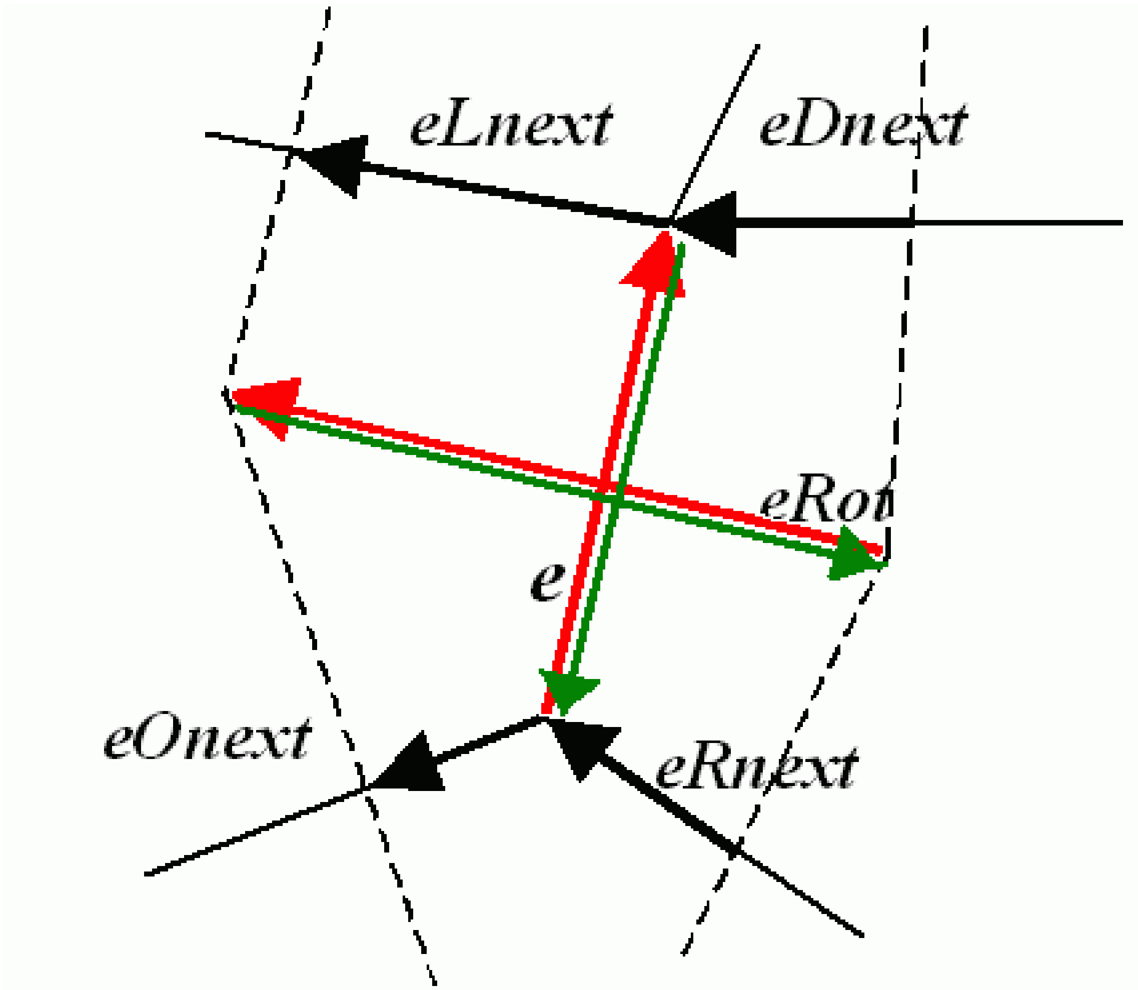
Subdiv2DRotateEdge

CvSubdiv2DEdge **cvSubdiv2DRotateEdge**(CvSubdiv2DEdge *edge*, int *rotate*)

Returns another edge of the same quad-edge.

Parameters

- **edge** – Subdivision edge (not a quad-edge)
- **rotate** – Specifies which of the edges of the same quad-edge as the input one to return, one of the following:
  - **0** the input edge (`e` on the picture below 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 returns one of the edges of the same quad-edge as the input edge.

SubdivDelaunay2DInsert

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

Inserts a single point into a Delaunay triangulation.

Parameters

- **subdiv** – Delaunay subdivision created by the function *CreateSubdivDelaunay2D*
- **pt** – Inserted point

The function inserts a single point into a subdivision and modifies the subdivision topology appropriately. If a point with the same coordinates exists already, no new point is added. The function returns a pointer to the allocated point. No virtual point coordinates are calculated at this stage.

2.7 Motion Analysis and Object Tracking

Acc

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

Adds a frame to an 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 with the same number of channels as input image, 32-bit or 64-bit floating-point
- **mask** – Optional operation mask

The function adds the whole image `image` or its selected region to the accumulator `sum` :

$$\texttt{sum}(x,y) \leftarrow \texttt{sum}(x,y) + \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

MultiplyAcc

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

Adds the product of two input images to the 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 with the same number of channels as input images, 32-bit or 64-bit floating-point
- **mask** – Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `acc` :

$$\texttt{acc}(x,y) \leftarrow \texttt{acc}(x,y) + \texttt{image1}(x,y) \cdot \texttt{image2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

RunningAvg

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

Updates the 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 with 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 calculates the weighted sum of the input image `image` and the accumulator `acc` so that `acc` becomes a running average of frame sequence:

$$\texttt{acc}(x,y) \leftarrow (1-\alpha) \cdot \texttt{acc}(x,y) + \alpha \cdot \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

where α regulates the update speed (how fast the accumulator forgets about previous frames).

SquareAcc

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

Adds the square of the source image to the 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 with the same number of channels as input image, 32-bit or 64-bit floating-point
- **mask** – Optional operation mask

The function adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum` :

$$\texttt{sqsum}(x,y) \leftarrow \texttt{sqsum}(x,y) + \texttt{image}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

2.8 Feature Detection

Canny

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

Implements the Canny algorithm for edge detection.

Parameters

- **image** – Single-channel input image
- **edges** – Single-channel image to store the edges found by the function
- **threshold1** – The first threshold
- **threshold2** – The second threshold
- **aperture_size** – Aperture parameter for the Sobel operator (see *Sobel*)

The function finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges.

CornerEigenValsAndVecs

void **cvCornerEigenValsAndVecs**(const CvArr* *image*, CvArr* *eigenvv*, int *blockSize*, 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
- **blockSize** – Neighborhood size (see discussion)
- **aperture_size** – Aperture parameter for the Sobel operator (see *Sobel*)

For every pixel, the function `cvCornerEigenValsAndVecs` considers a `blockSize` $\times$ `blockSize` neigborhood S(p). It calcualtes the covariation matrix of derivatives over the neigborhood as:

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

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

- λ_1, λ_2 are the eigenvalues of M ; not sorted
- x_1, y_1 are the eigenvectors corresponding to λ_1
- x_2, y_2 are the eigenvectors corresponding to λ_2

CornerHarris

void **cvCornerHarris** (const CvArr* *image*, CvArr* *harris_dst*, int *blockSize*, 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`
- **blockSize** – Neighborhood size (see the discussion of *CornerEigenValsAndVecs*)
- **aperture_size** – Aperture parameter for the Sobel operator (see *Sobel*).
- **k** – Harris detector free parameter. See the formula below

The function runs the Harris edge detector on the image. Similarly to *CornerMinEigenVal* and *CornerEigenValsAndVecs* , for each pixel it calculates a 2×2 gradient covariation matrix M over a `blockSize` $\times$ `blockSize` neighborhood. Then, it stores

$$det(M) - k \, trace(M)^2$$

to the destination image. Corners in the image can be found as the local maxima of the destination image.

CornerMinEigenVal

void **cvCornerMinEigenVal** (const CvArr* *image*, CvArr* *eigenval*, int *blockSize*, int *aperture_size=3*)

Calculates the 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`

- **blockSize** – Neighborhood size (see the discussion of *CornerEigenValsAndVecs*)
- **aperture_size** – Aperture parameter for the Sobel operator (see *Sobel*).

The function is similar to *CornerEigenValsAndVecs* but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. $min(\lambda_1, \lambda_2)$ in terms of the previous function.

FindCornerSubPix

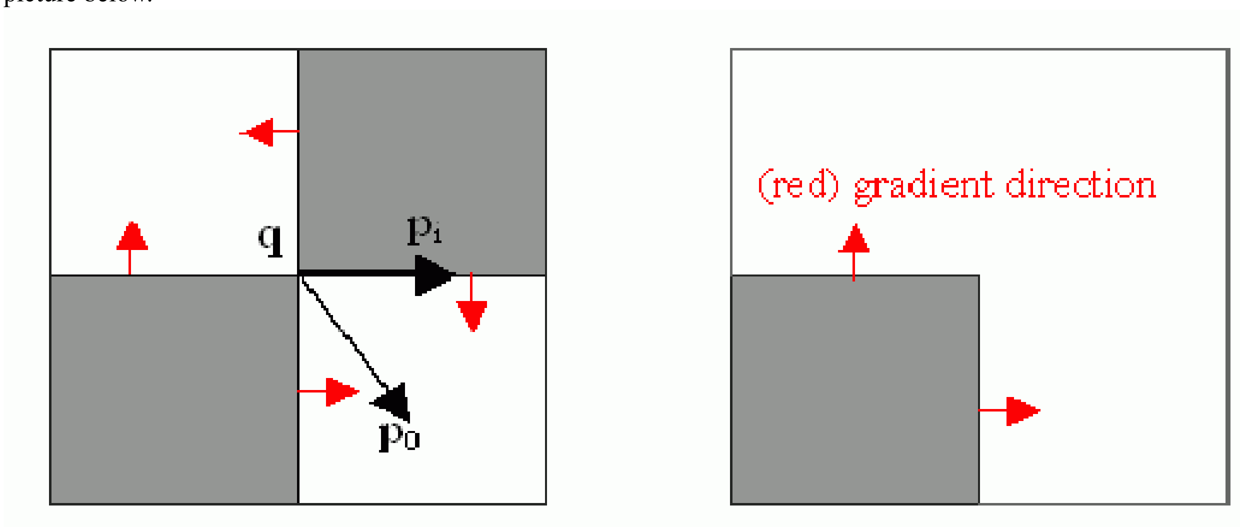
void **cvFindCornerSubPix** (const CvArr* *image*, CvPoint2D32f* *corners*, int *count*, CvSize *win*, CvSize *zero_zone*, CvTermCriteria *criteria*)

Refines the corner locations.

Parameters

- **image** – Input image
- **corners** – Initial coordinates of the input corners; refined coordinates on output
- **count** – Number of corners
- **win** – Half of the side length of the search window. For example, if `win` =(5,5), then a $5 * 2 + 1 \times 5 * 2 + 1 = 11 \times 11$ search window would be used
- **zero_zone** – Half of the size of the dead region in the middle of the search zone over which the summation in the formula 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 a certain number of iterations 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 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:

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

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q . The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

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

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:

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

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.

GoodFeaturesToTrack

void **cvGoodFeaturesToTrack**(const CvArr* image CvArr* *eigImage*, CvArr* tempImage CvPoint2D32f* corners int* cornerCount double qualityLevel double minDistance const CvArr* mask=NULL int blockSize=3 int useHarris=0 double *k=0.04*)

Determines strong corners on an image.

Parameters

- **image** – The source 8-bit or floating-point 32-bit, single-channel image
- **eigImage** – Temporary floating-point 32-bit image, the same size as `image`
- **tempImage** – Another temporary image, the same size and format as `eigImage`
- **corners** – Output parameter; detected corners
- **cornerCount** – Output parameter; number of detected corners
- **qualityLevel** – Multiplier for the max/min eigenvalue; specifies the minimal accepted quality of image corners
- **minDistance** – Limit, specifying the minimum possible distance between the returned corners; Euclidian 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
- **blockSize** – Size of the averaging block, passed to the underlying *CornerMinEigenVal* or *CornerHarris* used by the function
- **useHarris** – If nonzero, Harris operator (*CornerHarris*) is used instead of default *CornerMinEigenVal*
- **k** – Free parameter of Harris detector; used only if ($\texttt{useHarris} != 0$)

The function finds the corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using the *CornerMinEigenVal* function and stores them in `eigImage` . Then it performs non-maxima suppression (only the local maxima in 3×3 neighborhood are retained). The next step rejects the corners with the minimal eigenvalue less than $\texttt{qualityLevel} \cdot max(\texttt{eigImage}(x, y))$. Finally, the function ensures that the distance between any two corners is not smaller than `minDistance` . The weaker corners (with a smaller min eigenvalue) that are too close to the stronger corners are rejected.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel` , and `A` > {B}, the array of returned corners with `qualityLevel=A` will be the prefix of the output corners array with `qualityLevel=B` .

HoughLines2

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

Finds lines in a binary image using a Hough transform.

Parameters

- **image** – The 8-bit, single-channel, binary source image. In the case of a probabilistic method, the image is modified by the function
- **storage** – The storage for the lines that are 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 the number of lines detected. If `storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (in the case of standard hough transform the lines are sorted by the accumulator value)
- **method** – The Hough transform variant, one of the following:
  - **CV_HOUGH_STANDARD** classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ) , where ρ is a distance between (0,0) point and the line, and θ 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 line. Each 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 the classical Hough transform. The lines are encoded the same way as `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 the classical Hough transform it is not used (0).
  - For the probabilistic Hough transform it is the minimum line length.
  - For the multi-scale Hough transform it is the divisor for the distance resolution ρ . (The coarse distance resolution will be ρ and the accurate resolution will be $(\rho/\texttt{param1})$).
- **param2** – The second method-dependent parameter:
  - For the classical Hough transform it is not used (0).
  - For the probabilistic Hough transform it is the maximum gap between line segments lying on the same line to treat them as a single line segment (i.e. to join them).
  - For the multi-scale Hough transform it is the divisor for the angle resolution θ . (The coarse angle resolution will be θ and the accurate resolution will be $(\theta/\texttt{param2})$).

The function implements a few variants of the Hough transform for line detection.

Example. Detecting lines with Hough transform.

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

int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* lines = 0;
        int i;
        cvCanny( src, dst, 50, 200, 3 );
        cvCvtColor( dst, color_dst, CV_GRAY2BGR );
#if 1
        lines = cvHoughLines2( dst,
                               storage,
                               CV_HOUGH_STANDARD,
                               1,
                               CV_PI/180,
                               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,
                               80,
                               30,
                               10 );
        for( i = 0; i < lines->total; i++ )
        {
            CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
            cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3, 8 );
        }
#endif
```

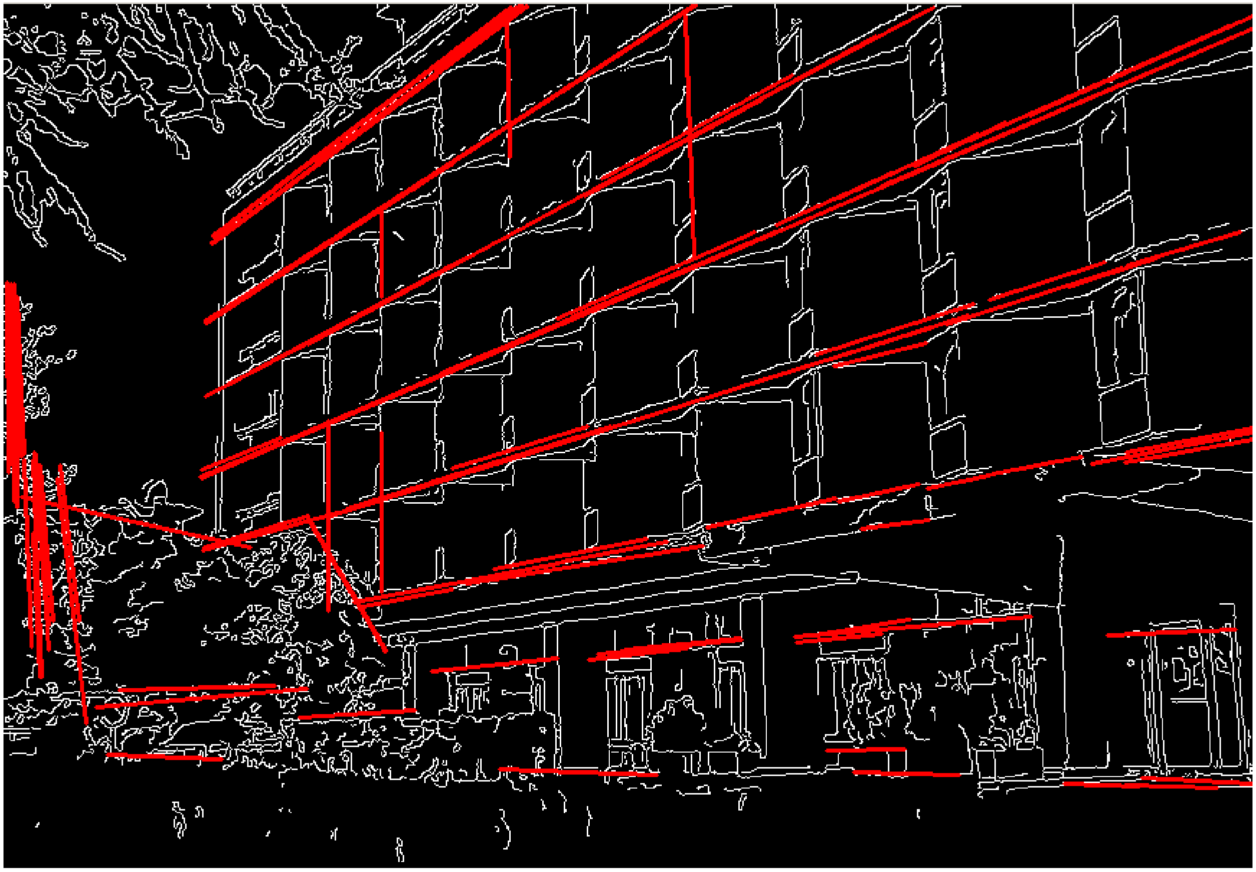
```
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

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

        cvWaitKey(0);
    }
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in the case of probabilistic Hough transform (`#if 0` case):



PreCornerDetect

void **cvPreCornerDetect** (const CvArr* *image*, CvArr* *corners*, int *apertureSize=3*)
Calculates the feature map for corner detection.

Parameters

- **image** – Input image
- **corners** – Image to store the corner candidates
- **apertureSize** – Aperture parameter for the Sobel operator (see *Sobel*)

The function calculates the function

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

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 below:

```
// 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 );
```

SampleLine

int **cvSampleLine**(const CvArr* image CvPoint pt1 CvPoint pt2 void* buffer int *connectivity=8*)
Reads the raster line to the buffer.

Parameters

- **image** – Image to sample the line from
- **pt1** – Starting line point
- **pt2** – Ending line point
- **buffer** – Buffer to store the line points; must have enough size to store $max(|\texttt{pt2.x} - \texttt{pt1.x}|+1, |\texttt{pt2.y}-\texttt{pt1.y}|+1)$ points in the case of an 8-connected line and $(|\texttt{pt2.x}-\texttt{pt1.x}| + |\texttt{pt2.y} - \texttt{pt1.y}| + 1)$ in the case of a 4-connected line
- **connectivity** – The line connectivity, 4 or 8

The function implements a particular application of line iterators. The function reads all of the image points lying on the line between `pt1` and `pt2` , including the end points, and stores them into the buffer.

2.9 Object Detection

MatchTemplate

void **cvMatchTemplate**(const CvArr* *image*, const CvArr* *templ*, CvArr* *result*, int *method*)
Compares a template against overlapped image regions.

Parameters

- **image** – Image where the search is running; 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 $W \times H$ and `templ` is $w \times h$ then `result` must be $(W - w + 1) \times (H - h + 1)$
- **method** – Specifies the way the template must be compared with the image regions (see below)

The function is similar to *CalcBackProjectPatch* . It slides through `image` , compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result` . Here are the formulas 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

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

- method=CV_TM_SQDIFF_NORMED

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

- method=CV_TM_CCORR

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

- method=CV_TM_CCORR_NORMED

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

- method=CV_TM_CCOEFF

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

  where

$$\begin{array}{l} T'(x',y')=T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'') \\ I'(x+x',y+y')=I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'') \end{array}$$

- method=CV_TM_CCOEFF_NORMED

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

After the function finishes the comparison, the best matches can be found as global minimums (`CV_TM_SQDIFF`) or maximums (`CV_TM_CCORR` and `CV_TM_CCOEFF`) using the *MinMaxLoc* function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel).

CHAPTER

THREE

FEATURES2D. FEATURE DETECTION AND DESCRIPTOR EXTRACTION

3.1 Feature detection and description

ExtractSURF

void **cvExtractSURF**(const CvArr* *image*, const CvArr* *mask*, CvSeq** *keypoints*, CvSeq** *descriptors*, CvMemStorage* *storage*, CvSURFParams *params*)

Extracts Speeded Up Robust Features from an 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. The sequence of CvSURFPoint structures is as follows:

```
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 match)
   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 feature strengths;
                    // see also params.hessianThreshold)
}
CvSURFPoint;
```

Parameters

- **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 a 64-element or a 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 are extracted.
                 // 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 values 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 default parameters
```

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 the `find_obj.cpp` demo in OpenCV samples directory.

GetStarKeypoints

CvSeq* **cvGetStarKeypoints**(const CvArr* *image*, CvMemStorage* *storage*, CvStarDetectorParams *params=cvStarDetectorParams()*)

Retrieves keypoints using the StarDetector algorithm.

Parameters

- **image** – The input 8-bit grayscale image
- **storage** – Memory storage where the keypoints will be stored
- **params** – Various algorithm parameters given 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 eliminate edges
   int suppressNonmaxSize; // linear size of a pixel neighborhood
                // for non-maxima suppression
}
CvStarDetectorParams;
```

The function GetStarKeypoints 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 of 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 a square, hexagon or octagon it uses an 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();
}
```

CHAPTER

FOUR

OBJDETECT. OBJECT DETECTION

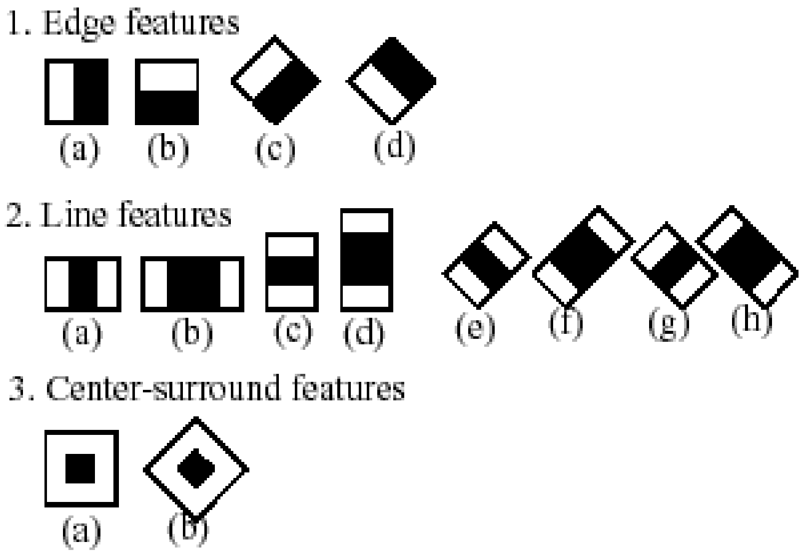
4.1 Cascade Classification

Haar Feature-based Cascade Classifier for 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 hundred 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 classifers, 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 the 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 the *Integral* description).

To see the object detector at work, have a look at the 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.

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, **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 subrectangles 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 responce is <= threshold,
                    left branch is chosen, otherwise right branch is chosen.
       alpha[i] - output value correponding to the leaf. */
    CvHaarFeature* haar_feature;
    float* threshold;
    int* left;
    int* right;
    float* alpha;
}
CvHaarClassifier;

/* a boosted battery of classifiers(=stage classifier):
   the stage classifier returns 1
   if the sum of the classifiers 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 stright cascades */
    int next;
    int child;
    int parent;
}
CvHaarStageClassifier;

typedef struct CvHidHaarClassifierCascade CvHidHaarClassifierCascade;

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

    /* these two parameters are set by cvSetImagesForHaarClassifierCascade */
    CvSize real_window_size; /* current object size */
    double scale; /* current scale */
    CvHaarStageClassifier* stage_classifier; /* array of stage classifiers */
```

```
    CvHidHaarClassifierCascade* hid_cascade; /* hidden optimized
                        representation of the
                        cascade, created by
                cvSetImagesForHaarClassifierCascade */
}
CvHaarClassifierCascade;
```

All the structures are used for representing a cascaded of boosted Haar classifiers. The cascade has the following hierarchical structure:

```
Cascade:
        Stage,,1,,:
            Classifier,,11,,:
                Feature,,11,,
            Classifier,,12,,:
                Feature,,12,,
            ...
        Stage,,2,,:
            Classifier,,21,,:
                Feature,,21,,
            ...
        ...
```

The whole hierarchy can be constructed manually or loaded from a file or an embedded base using the function *LoadHaarClassifierCascade* .

LoadHaarClassifierCascade

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

Loads a trained cascade classifier from a file or the classifier database embedded in OpenCV.

Parameters

- **directory** – Name of the directory containing the description of a trained cascade classifier
- **orig_window_size** – Original size of the objects the cascade has been trained on. Note that it is not stored in the cascade and therefore must be specified separately

The function loads a trained cascade of haar classifiers from a file or the classifier database embedded in OpenCV. The base can be trained using the `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 a cascade from a file, use the *Load* function.

HaarDetectObjects

..

CvSeq* **cvHaarDetectObjects**(const CvArr* *image*, CvHaarClassifierCascade* *cascade*, CvMemStorage* *storage*, double *scaleFactor=1.1*, int *minNeighbors=3*, int *flags=0*, CvSize *minSize=cvSize(0*, 0), CvSize *maxSize=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;

param image Image to detect objects in

param cascade Haar classifier cascade in internal representation

param storage Memory storage to store the resultant sequence of the object candidate rectangles

param scaleFactor The factor by which the search window is scaled between the subsequent scans, 1.1 means increasing window by 10 %

param minNeighbors 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 `minNeighbors` 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

param flags Mode of operation. Currently the only flag that may be specified is `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

param minSize Minimum window size. By default, it is set to the size of samples the classifier has been trained on ($\sim 20 \times 20$ for face detection)

param maxSize Maximum window size to use. By default, it is set to the size of the image.

The function 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 *SetImagesForHaarClassifierCascade*). Each time it considers overlapping regions in the image and applies the classifiers to the regions using *RunHaarClassifierCascade* . It may also apply some heuristics to reduce number of analyzed regions, such as Canny prunning. 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 settings are: `scale_factor` =1.2, `min_neighbors` =2, `flags` = `CV_HAAR_DO_CANNY_PRUNING` , `min_size` = *minimum possible face size* (for example, $\sim$ 1/4 to 1/16 of the image area in the case of video conferencing).

```
#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 rectanlges only */
        CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i );
        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;
}
```

SetImagesForHaarClassifierCascade

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 *CreateHidHaarClassifierCascade*
- **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 *Integral*

- **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, the original window size is used (objects of that size are searched) - the same size as specified in *LoadHaarClassifierCascade* (24x24 in the case of `default_face_cascade`), if `scale` =2, a two times larger window is used (48x48 in the case of default face cascade). While this will speed-up search about four times, faces smaller than 48x48 cannot be detected

The function 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 the *GetHaarClassifierCascadeScale* 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 *HaarDetectObjects* , but it can be called by the user if they are using the lower-level function *RunHaarClassifierCascade* .

ReleaseHaarClassifierCascade

void **cvReleaseHaarClassifierCascade**(CvHaarClassifierCascade** *cascade*)

Releases the haar classifier cascade.

Parameters

- **cascade** – Double pointer to the released cascade. The pointer is cleared by the function

The function deallocates the cascade that has been created manually or loaded using *LoadHaarClassifierCascade* or *Load* .

RunHaarClassifierCascade

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

Runs a cascade of boosted classifiers at the 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 currenly set scale. The current window size may be retrieved using the *GetHaarClassifierCascadeWindowSize* 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 *HaarDetectObjects* for better processor cache utilization

The function runs the 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 *SetImagesForHaarClassifierCascade* . The function returns a positive value if the analyzed rectangle passed all the classifier stages (it is a candidate) and a zero or negative value otherwise.

CHAPTER

FIVE

VIDEO. VIDEO ANALYSIS

5.1 Motion Analysis and Object Tracking

CalcGlobalOrientation

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

Calculates the global motion orientation of some selected region.

Parameters

- **orientation** – Motion gradient orientation image; calculated by the function *CalcMotionGradient*
- **mask** – Mask image. It may be a conjunction of a valid gradient mask, obtained with *CalcMotionGradient* and the 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 *UpdateMotionHistory* before and reuse it here, because running *UpdateMotionHistory* and *CalcMotionGradient* on large images may take some time
- **duration** – Maximal duration of motion track in milliseconds, the same as *UpdateMotionHistory*

The function calculates the general motion direction in the selected region and returns the angle between 0 degrees and 360 degrees . 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 of the orientation vectors: the more recent the motion, the greater the weight. The resultant angle is a circular sum of the basic orientation and the shift.

CalcMotionGradient

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

Calculates the gradient orientation of a motion history image.

Parameters

- **mhi** – Motion history image
- **mask** – Mask image; marks pixels where the motion gradient data is correct; output parameter

- **orientation** – Motion gradient orientation image; contains angles from 0 to ~360 degrees
- **delta1** – See below
- **delta2** – See below
- **apertureSize** – Aperture size of derivative operators used by the function: CV _ SCHARR, 1, 3, 5 or 7 (see *Sobel*)

The function calculates the derivatives Dx and Dy of `mhi` and then calculates gradient orientation as:

$$\texttt{orientation}(x, y) = \arctan \frac{Dy(x, y)}{Dx(x, y)}$$

where both $Dx(x, y)$ and $Dy(x, y)$ signs are taken into account (as in the *CartToPolar* function). After that `mask` is filled to indicate where the orientation is valid (see the `delta1` and `delta2` description).

The function finds the 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

$$\min(\texttt{delta1}, \texttt{delta2}) \le M(x, y) - m(x, y) \le \max(\texttt{delta1}, \texttt{delta2}).$$

CalcOpticalFlowBM

void **cvCalcOpticalFlowBM**(const CvArr* *prev*, const CvArr* *curr*, CvSize *blockSize*, CvSize *shiftSize*, CvSize *max_range*, int *usePrevious*, CvArr* *velx*, CvArr* *vely*)

Calculates the optical flow for two images by using the block matching method.

Parameters

- **prev** – First image, 8-bit, single-channel
- **curr** – Second image, 8-bit, single-channel
- **blockSize** – Size of basic blocks that are compared
- **shiftSize** – Block coordinate increments
- **max_range** – Size of the scanned neighborhood in pixels around the block
- **usePrevious** – Uses the previous (input) velocity field
- **velx** – Horizontal component of the optical flow of

$$\left\lfloor \frac{\texttt{prev->width} - \texttt{blockSize.width}}{\texttt{shiftSize.width}} \right\rfloor \times \left\lfloor \frac{\texttt{prev->height} - \texttt{blockSize.height}}{\texttt{shiftSize.height}} \right\rfloor$$

  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 calculates the optical flow for overlapped blocks $\texttt{blockSize.width} \times \texttt{blockSize.height}$ 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 `usePrevious=1`)

CalcOpticalFlowHS

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

Calculates the optical flow for two images.

Parameters

- **prev** – First image, 8-bit, single-channel
- **curr** – Second image, 8-bit, single-channel
- **usePrevious** – Uses the 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 computes the flow for every pixel of the first input image using the Horn and Schunck algorithm Horn81 .

CalcOpticalFlowLK

void **cvCalcOpticalFlowLK**(const CvArr* *prev*, const CvArr* *curr*, CvSize *winSize*, CvArr* *velx*, CvArr* *vely*)

Calculates the optical flow for two images.

Parameters

- **prev** – First image, 8-bit, single-channel
- **curr** – Second image, 8-bit, single-channel
- **winSize** – 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 computes the flow for every pixel of the first input image using the Lucas and Kanade algorithm Lucas81 .

CalcOpticalFlowPyrLK

void **cvCalcOpticalFlowPyrLK**(const CvArr* *prev*, const CvArr* *curr*, CvArr* *prevPyr*, CvArr* *currPyr*, const CvPoint2D32f* *prevFeatures*, CvPoint2D32f* *currFeatures*, int *count*, CvSize *winSize*, int *level*, char* *status*, float* *track_error*, CvTermCriteria *criteria*, int *flags*)

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

Parameters

- **prev** – First frame, at time `t`

- **curr** – Second frame, at time `t + dt`
- **prevPyr** – 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
- **currPyr** – Similar to `prevPyr` , used for the second frame
- **prevFeatures** – Array of points for which the flow needs to be found
- **currFeatures** – Array of 2D points containing the calculated new positions of the input features in the second image
- **count** – Number of feature points
- **winSize** – 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 the 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_LKFLOWPyr_A_READY** pyramid for the first frame is precalculated before the call
  - **CV_LKFLOWPyr_B_READY** pyramid for the second frame is precalculated before the call
  - **CV_LKFLOW_INITIAL_GUESSES** array B contains initial coordinates of features before the function call

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids Bouguet00 . It calculates the 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 `prevPyr` and `currPyr` 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_LKFLOWPyr_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_LKFLOWPyr_A_READY` is set).

CamShift

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

Finds the object center, size, and orientation.

Parameters

- **prob_image** – Back projection of object histogram (see *CalcBackProject*)
- **window** – Initial search window

- **criteria** – Criteria applied to determine when the window search should be finished
- **comp** – Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)
- **box** – Circumscribed box for the object. If not `NULL` , it contains object size and orientation

The function implements the CAMSHIFT object tracking algrorithm Bradski98 . First, it finds an object center using *MeanShift* and, after that, calculates the object size and orientation. The function returns number of iterations made within *MeanShift* .

The `CamShiftTracker` class declared in cv.hpp implements the color object tracker that uses the function.

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 the 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 .

CreateConDensation

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

Allocates the 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 creates a `CvConDensation` structure and returns a pointer to the structure.

ConDensInitSampleSet

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

Initializes the sample set for the 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 fills the samples arrays in the structure `condens` with values within the specified ranges.

CvKalman

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 the Kalman filter state. It is created by the *CreateKalman* function, updated by the *KalmanPredict* and *KalmanCorrect* functions and released by the *ReleaseKalman* function . Normally, the structure is used for the standard Kalman filter (notation and the formulas below are borrowed from the excellent Kalman tutorial Welch95)

$$
\begin{array}{l}
x_k = A \cdot x_{k-1} + B \cdot u_k + w_k \\
z_k = H \cdot x_k + v_k
\end{array}
$$

where:

x_k (x_{k-1})	state of the system at the moment *k* (*k-1*)
z_k	measurement of the system state at the moment *k*
u_k	external control applied at the moment *k*

w_k and v_k are normally-distributed process and measurement noise, respectively:

$$
\begin{array}{l}
p(w) \sim N(0, Q) \\
p(v) \sim N(0, R)
\end{array}
$$

that is,

Q process noise covariance matrix, constant or variable,

R measurement noise covariance matrix, constant or variable

In the case of the standard Kalman filter, all of the matrices: A, B, H, Q and R are initialized once after the *CvKalman* structure is allocated via *CreateKalman* . However, the same structure and the same functions may be used to simulate the extended Kalman filter by linearizing the 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.

CreateKalman

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

Allocates the 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 allocates *CvKalman* and all its matrices and initializes them somehow.

KalmanCorrect

const CvMat* **cvKalmanCorrect** (CvKalman* *kalman*, const CvMat* *measurement*)

Adjusts the model state.

Parameters

- **kalman** – Pointer to the structure to be updated
- **measurement** – CvMat containing the measurement vector

The function adjusts the stochastic model state on the basis of the given measurement of the model state:

$$
\begin{array}{l}
K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1} \\
x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k) \\
P_k = (I - K_k \cdot H) \cdot P'_k
\end{array}
$$

where

z_k	**given measurement (`mesurement` parameter)**
K_k	Kalman "gain" matrix.

The function stores the 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;
}
```

KalmanPredict

const CvMat* **cvKalmanPredict** (CvKalman* *kalman*, const CvMat* *control=NULL*)

Estimates the subsequent model state.

Parameters

- **kalman** – Kalman filter state
- **control** – Control vector u_k , should be NULL iff there is no external control (`control_params` =0)

The function estimates the subsequent stochastic model state by its current state and stores it at `kalman->state_pre`:

$$\begin{array}{l} x'_k = Ax_{k-1} + Bu_k \\ P'_k = AP_{k-1}A^T + Q \end{array}$$

where

x'_k	**is predicted state `kalman->state_pre`,**
x_{k-1}	is corrected state on the previous step `kalman->state_post` (should be initialized somehow in the beginning, zero vector by default),
u_k	is external control (`control` parameter),
P'_k	is priori error covariance matrix `kalman->error_cov_pre`
P_{k-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.

KalmanUpdateByMeasurement

Synonym for *KalmanCorrect*

KalmanUpdateByTime

Synonym for *KalmanPredict*

MeanShift

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

Finds the object center on back projection.

Parameters

- **prob_image** – Back projection of the object histogram (see *CalcBackProject*)
- **window** – Initial search window

- **criteria** – Criteria applied to determine when the window search should be finished
- **comp** – Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

The function 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.

ReleaseConDensation

void **cvReleaseConDensation** (CvConDensation** *condens*)

Deallocates the ConDensation filter structure.

Parameters

- **condens** – Pointer to the pointer to the structure to be released

The function releases the structure `condens`) and frees all memory previously allocated for the structure.

ReleaseKalman

void **cvReleaseKalman** (CvKalman** *kalman*)

Deallocates the Kalman filter structure.

Parameters

- **kalman** – double pointer to the Kalman filter structure

The function releases the structure *CvKalman* and all of the underlying matrices.

SegmentMotion

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

Segments a 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 finds all of the motion segments and marks them in `seg_mask` with individual values (1,2,...). It also returns a sequence of *CvConnectedComp* structures, one for each motion component. After that the motion direction for every component can be calculated with *CalcGlobalOrientation* using the extracted mask of the particular component *Cmp* .

SnakeImage

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 the 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 for each contour point
- **beta** – Weight[s] of curvature energy, similar to `alpha`
- **gamma** – Weight[s] of image energy, similar to `alpha`
- **coeff_usage** – Different uses 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 the gradient magnitude for every image pixel and consideres it as the energy field, otherwise the input image itself is considered

The function updates the snake in order to minimize its total energy that is a sum of internal energy that depends on the 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 the case of using an 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.

UpdateMotionHistory

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

Updates the motion history image by a 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 the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \ne 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

CHAPTER

SIX

HIGHGUI. HIGH-LEVEL GUI AND MEDIA I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and "remember" their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

6.1 User Interface

ConvertImage

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

Converts one image to another with an optional vertical flip.

Parameters

- **src** – Source image.
- **dst** – Destination image. Must be single-channel or 3-channel 8-bit image.
- **flags** – The operation flags:
  - **CV_CVTIMG_FLIP** Flips the image vertically
  - **CV_CVTIMG_SWAP_RB** Swaps the red and blue channels. In OpenCV color images have `BGR` channel order, however on some systems the order needs to be reversed before displaying the image (*ShowImage* does this automatically).

The function `cvConvertImage` converts one image to another and flips the result vertically if desired. The function is used by *ShowImage* .

CreateTrackbar

int **cvCreateTrackbar**(const char* *trackbarName*, const char* *windowName*, int* *value*, int *count*, CvTrackbarCallback *onChange*)

Creates a trackbar and attaches it to the specified window

Parameters

- **trackbarName** – Name of the created trackbar.
- **windowName** – Name of the window which will be used as a parent for created trackbar.
- **value** – Pointer to an integer variable, whose value will reflect the position of the slider. Upon creation, the slider position is defined by this variable.
- **count** – Maximal position of the slider. Minimal position is always 0.
- **onChange** – Pointer to the function to be called every time the slider changes position. This function should be prototyped as `void Foo(int);` Can be NULL if callback is not required.

The function `cvCreateTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable to be syncronized with trackbar position and specifies a callback function to be called on trackbar position change. The created trackbar is displayed on the top of the given window. **[Qt Backend Only]** qt-specific details:

- **windowName** Name of the window which will be used as a parent for created trackbar. Can be NULL if the trackbar should be attached to the control panel.

The created trackbar is displayed at the bottom of the given window if *windowName* is correctly provided, or displayed on the control panel if *windowName* is NULL.

By clicking on the label of each trackbar, it is possible to edit the trackbar's value manually for a more accurate control of it.

```
CV_EXTERN_C_FUNCPTR( void (*CvTrackbarCallback)(int pos) );
```

DestroyAllWindows

void **cvDestroyAllWindows**(void)

Destroys all of the HighGUI windows.

The function `cvDestroyAllWindows` destroys all of the opened HighGUI windows.

DestroyWindow

void **cvDestroyWindow**(const char* *name*)

Destroys a window.

Parameters

- **name** – Name of the window to be destroyed.

The function `cvDestroyWindow` destroys the window with the given name.

GetTrackbarPos

int **cvGetTrackbarPos**(const char* *trackbarName*, const char* *windowName*)

Returns the trackbar position.

Parameters

- **trackbarName** – Name of the trackbar.
- **windowName** – Name of the window which is the parent of the trackbar.

The function `cvGetTrackbarPos` returns the current position of the specified trackbar. **[Qt Backend Only]** qt-specific details:

- **windowName** Name of the window which is the parent of the trackbar. Can be NULL if the trackbar is attached to the control panel.

GetWindowHandle

void* **cvGetWindowHandle**(const char* *name*)

Gets the window's handle by its name.

Parameters

- **name** – Name of the window

.

The function `cvGetWindowHandle` returns the native window handle (HWND in case of Win32 and GtkWidget in case of GTK+). **[Qt Backend Only]** qt-specific details: The function `cvGetWindowHandle` returns the native window handle inheriting from the Qt class QWidget.

GetWindowName

const char* **cvGetWindowName**(void* *windowHandle*)

Gets the window's name by its handle.

Parameters

- **windowHandle** – Handle of the window.

The function `cvGetWindowName` returns the name of the window given its native handle (HWND in case of Win32 and GtkWidget in case of GTK+). **[Qt Backend Only]** qt-specific details: The function `cvGetWindowName` returns the name of the window given its native handle (QWidget).

InitSystem

int **cvInitSystem**(int *argc*, char** *argv*)

Initializes HighGUI.

Parameters

- **argc** – Number of command line arguments
- **argv** – Array of command line arguments

The function `cvInitSystem` initializes HighGUI. If it wasn't called explicitly by the user before the first window was created, it is called implicitly then with `argc=0` , `argv=NULL` . Under Win32 there is no need to call it explicitly. Under X Window the arguments may be used to customize a look of HighGUI windows and controls. **[Qt Backend Only]** qt-specific details: The function `cvInitSystem` is automatically called at the first cvNamedWindow call.

MoveWindow

void **cvMoveWindow** (const char* *name*, int *x*, int *y*)

Sets the position of the window.

Parameters

- **name** – Name of the window to be moved.
- **x** – New x coordinate of the top-left corner
- **y** – New y coordinate of the top-left corner

The function `cvMoveWindow` changes the position of the window.

NamedWindow

int **cvNamedWindow** (const char* *name*, int *flags*)

Creates a window.

Parameters

- **name** – Name of the window in the window caption that may be used as a window identifier.
- **flags** – Flags of the window. Currently the only supported flag is `CV_WINDOW_AUTOSIZE` . If this is set, window size is automatically adjusted to fit the displayed image (see *ShowImage*), and the user can not change the window size manually.

The function `cvNamedWindow` creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing. **[Qt Backend Only]** qt-specific details:

- **flags** Flags of the window. Currently the supported flags are:
  - **CV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE:** `CV_WINDOW_NORMAL` let the user resize the window, whereas `CV_WINDOW_AUTOSIZE` adjusts automatically the window's size to fit the displayed image (see *ShowImage*), and the user can not change the window size manually.
  - **CV_WINDOW_FREERATIO or CV_WINDOW_KEEPRATIO:** `CV_WINDOW_FREERATIO` adjust the image without respect the its ration, whereas `CV_WINDOW_KEEPRATIO` keep the image's ratio.
  - **CV_GUI_NORMAL or CV_GUI_EXPANDED:** `CV_GUI_NORMAL` is the old way to draw the window without statusbar and toolbar, whereas `CV_GUI_EXPANDED` is the new enhance GUI.

  This parameter is optional. The default flags set for a new window are `CV_WINDOW_AUTOSIZE` , `CV_WINDOW_KEEPRATIO` , and `CV_GUI_EXPANDED` .

  However, if you want to modify the flags, you can combine them using OR operator, ie:

  ```
  cvNamedWindow( ``myWindow'', ``CV_WINDOW_NORMAL``   textbar   ``CV_GUI_NORMAL`` );
  ```

ResizeWindow

void **cvResizeWindow** (const char* *name*, int *width*, int *height*)

Sets the window size.

Parameters

- **name** – Name of the window to be resized.
- **width** – New width
- **height** – New height

The function `cvResizeWindow` changes the size of the window.

SetMouseCallback

void **cvSetMouseCallback**(const char* *windowName*, CvMouseCallback *onMouse*, void* *param=NULL*)

Assigns callback for mouse events.

Parameters

- **windowName** – Name of the window.
- **onMouse** – Pointer to the function to be called every time a mouse event occurs in the specified window. This function should be prototyped as `void Foo(int event, int x, int y, int flags, void* param);` where `event` is one of `CV_EVENT_*` , `x` and `y` are the coordinates of the mouse pointer in image coordinates (not window coordinates), `flags` is a combination of `CV_EVENT_FLAG_*` , and `param` is a user-defined parameter passed to the `cvSetMouseCallback` function call.
- **param** – User-defined parameter to be passed to the callback function.

The function `cvSetMouseCallback` sets the callback function for mouse events occuring within the specified window.

The `event` parameter is one of:

- **CV_EVENT_MOUSEMOVE** Mouse movement
- **CV_EVENT_LBUTTONDOWN** Left button down
- **CV_EVENT_RBUTTONDOWN** Right button down
- **CV_EVENT_MBUTTONDOWN** Middle button down
- **CV_EVENT_LBUTTONUP** Left button up
- **CV_EVENT_RBUTTONUP** Right button up
- **CV_EVENT_MBUTTONUP** Middle button up
- **CV_EVENT_LBUTTONDBLCLK** Left button double click
- **CV_EVENT_RBUTTONDBLCLK** Right button double click
- **CV_EVENT_MBUTTONDBLCLK** Middle button double click

The `flags` parameter is a combination of :

- **CV_EVENT_FLAG_LBUTTON** Left button pressed
- **CV_EVENT_FLAG_RBUTTON** Right button pressed
- **CV_EVENT_FLAG_MBUTTON** Middle button pressed
- **CV_EVENT_FLAG_CTRLKEY** Control key pressed
- **CV_EVENT_FLAG_SHIFTKEY** Shift key pressed
- **CV_EVENT_FLAG_ALTKEY** Alt key pressed

SetTrackbarPos

void **cvSetTrackbarPos** (const char* *trackbarName*, const char* *windowName*, int *pos*)
Sets the trackbar position.

Parameters

- **trackbarName** – Name of the trackbar.
- **windowName** – Name of the window which is the parent of trackbar.
- **pos** – New position.

The function `cvSetTrackbarPos` sets the position of the specified trackbar. **[Qt Backend Only]** qt-specific details:

- **windowName** Name of the window which is the parent of trackbar. Can be NULL if the trackbar is attached to the control panel.

ShowImage

void **cvShowImage** (const char* *name*, const CvArr* *image*)
Displays the image in the specified window

Parameters

- **name** – Name of the window.
- **image** – Image to be shown.

The function `cvShowImage` displays the image in the specified window. If the window was created with the `CV_WINDOW_AUTOSIZE` flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

WaitKey

int **cvWaitKey** (int *delay=0*)
Waits for a pressed key.

Parameters

- **delay** – Delay in milliseconds.

The function `cvWaitKey` waits for key event infinitely (`delay` $<=$ 0) or for `delay` milliseconds. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Notes:

- This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

[Qt Backend Only] qt-specific details: With this current Qt implementation, this is the only way to process event such as repaint for the windows, and so on ldots

- The function only works if there is at least one HighGUI window created and the window is active. If there are several HighGUI windows, any of them can be active.

6.2 Reading and Writing Images and Video

LoadImage

IplImage* **cvLoadImage**(const char* *filename*, int *iscolor=CV_LOAD_IMAGE_COLOR*)
Loads an image from a file as an IplImage.

Parameters

- **filename** – Name of file to be loaded.
- **iscolor** – Specific color type of the loaded image:
  - **CV_LOAD_IMAGE_COLOR** the loaded image is forced to be a 3-channel color image
  - **CV_LOAD_IMAGE_GRAYSCALE** the loaded image is forced to be grayscale
  - **CV_LOAD_IMAGE_UNCHANGED** the loaded image will be loaded as is.

The function `cvLoadImage` loads an image from the specified file and returns the pointer to the loaded image. Currently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

Note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB.

LoadImageM

CvMat* **cvLoadImageM**(const char* *filename*, int *iscolor=CV_LOAD_IMAGE_COLOR*)
Loads an image from a file as a CvMat.

Parameters

- **filename** – Name of file to be loaded.
- **iscolor** – Specific color type of the loaded image:
  - **CV_LOAD_IMAGE_COLOR** the loaded image is forced to be a 3-channel color image
  - **CV_LOAD_IMAGE_GRAYSCALE** the loaded image is forced to be grayscale
  - **CV_LOAD_IMAGE_UNCHANGED** the loaded image will be loaded as is.

The function `cvLoadImageM` loads an image from the specified file and returns the pointer to the loaded image. urrently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE

- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

Note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB.

SaveImage

int **cvSaveImage**(const char* *filename*, const CvArr* *image*)

Saves an image to a specified file.

Parameters

- **filename** – Name of the file.
- **image** – Image to be saved.

The function `cvSaveImage` saves the image to the specified file. The image format is chosen based on the `filename` extension, see *LoadImage* . Only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use `cvCvtScale` and `cvCvtColor` to convert it before saving, or use universal `cvSave` to save the image to XML or YAML format.

CvCapture

`CvCapture`

Video capturing structure.

typedef struct CvCapture **CvCapture**()

The structure `CvCapture` does not have a public interface and is used only as a parameter for video capturing functions.

CaptureFromCAM

CvCapture* **cvCaptureFromCAM**(int *index*)

Initializes capturing a video from a camera.

Parameters

- **index** – Index of the camera to be used. If there is only one camera or it does not matter what camera is used -1 may be passed.

The function `cvCaptureFromCAM` allocates and initializes the CvCapture structure for reading a video stream from the camera. Currently two camera interfaces can be used on Windows: Video for Windows (VFW) and Matrox Imaging Library (MIL); and two on Linux: V4L and FireWire (IEEE1394).

To release the structure, use *ReleaseCapture* .

CaptureFromFile

CvCapture* **cvCaptureFromFile**(const char* *filename*)

Initializes capturing a video from a file.

Parameters

- **filename** – Name of the video file.

The function `cvCaptureFromFile` allocates and initializes the CvCapture structure for reading the video stream from the specified file. Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect and how to prepare your video files.

After the allocated structure is not used any more it should be released by the *ReleaseCapture* function.

GetCaptureProperty

double **cvGetCaptureProperty**(CvCapture* *capture*, int *property_id*)

Gets video capturing properties.

Parameters

- **capture** – video capturing structure.
- **property_id** – Property identifier. Can be one of the following:
  - **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
  - **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
  - **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
  - **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
  - **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
  - **CV_CAP_PROP_FPS** Frame rate
  - **CV_CAP_PROP_FOURCC** 4-character code of codec
  - **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
  - **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
  - **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
  - **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
  - **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
  - **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
  - **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
  - **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
  - **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
  - **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB

 – **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
 – **CV_CAP_PROP_RECTIFICATION** TOWRITE (note: only supported by DC1394 v 2.x backend currently)

The function `cvGetCaptureProperty` retrieves the specified property of the camera or video file.

GrabFrame

int **cvGrabFrame**(CvCapture* *capture*)
Grabs the frame from a camera or file.

Parameters

- **capture** – video capturing structure.

The function `cvGrabFrame` grabs the frame from a camera or file. The grabbed frame is stored internally. The purpose of this function is to grab the frame *quickly* so that syncronization can occur if it has to read from several cameras simultaneously. The grabbed frames are not exposed because they may be stored in a compressed format (as defined by the camera/driver). To retrieve the grabbed frame, *RetrieveFrame* should be used.

QueryFrame

IplImage* **cvQueryFrame**(CvCapture* *capture*)
Grabs and returns a frame from a camera or file.

Parameters

- **capture** – video capturing structure.

The function `cvQueryFrame` grabs a frame from a camera or video file, decompresses it and returns it. This function is just a combination of *GrabFrame* and *RetrieveFrame* , but in one call. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

ReleaseCapture

void **cvReleaseCapture**(CvCapture** *capture*)
Releases the CvCapture structure.

Parameters

- **capture** – Pointer to video the capturing structure.

The function `cvReleaseCapture` releases the CvCapture structure allocated by *CaptureFromFile* or *CaptureFromCAM* .

RetrieveFrame

IplImage* **cvRetrieveFrame**(CvCapture* *capture*)
Gets the image grabbed with cvGrabFrame.

Parameters

- **capture** – video capturing structure.

The function `cvRetrieveFrame` returns the pointer to the image grabbed with the *GrabFrame* function. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

SetCaptureProperty

int **cvSetCaptureProperty** (CvCapture* *capture*, int *property_id*, double *value*)

Sets video capturing properties.

Parameters

- **capture** – video capturing structure.
- **property_id** – property identifier. Can be one of the following:
  - **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
  - **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
  - **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
  - **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
  - **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
  - **CV_CAP_PROP_FPS** Frame rate
  - **CV_CAP_PROP_FOURCC** 4-character code of codec
  - **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
  - **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
  - **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
  - **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
  - **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
  - **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
  - **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
  - **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
  - **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
  - **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB
  - **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
  - **CV_CAP_PROP_RECTIFICATION** TOWRITE (note: only supported by DC1394 v 2.x backend currently)
- **value** – value of the property.

The function `cvSetCaptureProperty` sets the specified property of video capturing. Currently the function supports only video files: `CV_CAP_PROP_POS_MSEC`, `CV_CAP_PROP_POS_FRAMES`, `CV_CAP_PROP_POS_AVI_RATIO`.

NB This function currently does nothing when using the latest CVS download on linux with FFMPEG (the function contents are hidden if 0 is used and returned).

CreateVideoWriter

typedef struct CvVideoWriter CvVideoWriter CvVideoWriter* **cvCreateVideoWriter**(const char* *filename*, int *fourcc*, double *fps*, CvSize *frame_size*, int *is_color=1*)

Creates the video file writer.

Parameters

- **filename** – Name of the output video file.
- **fourcc** – 4-character code of codec used to compress the frames. For example, `CV_FOURCC('P','I','M','1')` is a MPEG-1 codec, `CV_FOURCC('M','J','P','G')` is a motion-jpeg codec etc. Under Win32 it is possible to pass -1 in order to choose compression method and additional compression parameters from dialog. Under Win32 if 0 is passed while using an avi filename it will create a video writer that creates an uncompressed avi file.
- **fps** – Framerate of the created video stream.
- **frame_size** – Size of the video frames.
- **is_color** – If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only).

The function `cvCreateVideoWriter` creates the video writer structure.

Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect.

ReleaseVideoWriter

void **cvReleaseVideoWriter**(CvVideoWriter** *writer*)

Releases the AVI writer.

Parameters

- **writer** – Pointer to the video file writer structure.

The function `cvReleaseVideoWriter` finishes writing to the video file and releases the structure.

WriteFrame

int **cvWriteFrame**(CvVideoWriter* *writer*, const IplImage* *image*)

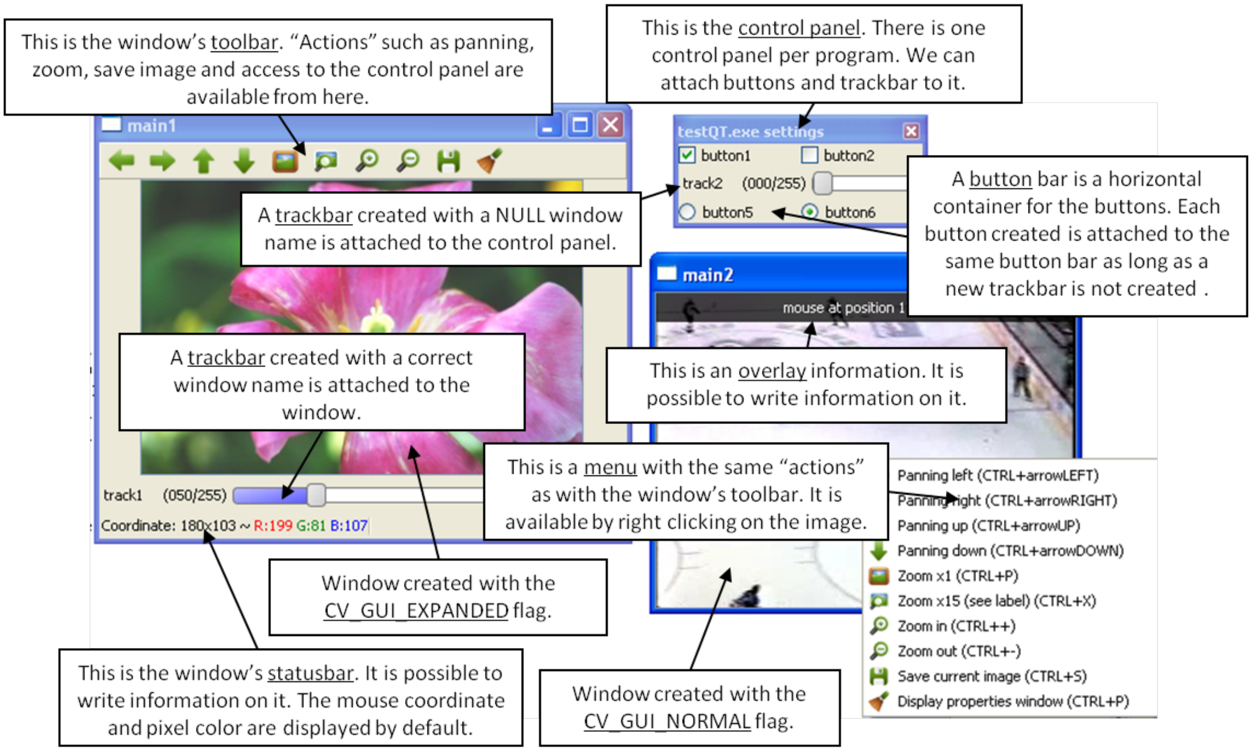
Writes a frame to a video file.

Parameters

- **writer** – Video writer structure
- **image** – The written frame

The function `cvWriteFrame` writes/appends one frame to a video file.

6.3 Qt new functions



This figure explains the new functionalities implemented with Qt GUI. As we can see, the new GUI provides a statusbar, a toolbar, and a control panel. The control panel can have trackbars and buttonbars attached to it.

- To attach a trackbar, the window _ name parameter must be NULL.
- To attach a buttonbar, a button must be created. If the last bar attached to the control panel is a buttonbar, the new button is added on the right of the last button. If the last bar attached to the control panel is a trackbar, or the control panel is empty, a new buttonbar is created. Then a new button is attached to it.

The following code is an example used to generate the figure.

```
int main(int argc, char *argv[])
    int value = 50;
    int value2 = 0;

    cvNamedWindow("main1",CV_WINDOW_NORMAL);
    cvNamedWindow("main2",CV_WINDOW_AUTOSIZE | CV_GUI_NORMAL);

    cvCreateTrackbar( "track1", "main1", &value, 255,  NULL);//OK tested
    char* nameb1 = "button1";
    char* nameb2 = "button2";
    cvCreateButton(nameb1,callbackButton,nameb1,CV_CHECKBOX,1);

    cvCreateButton(nameb2,callbackButton,nameb2,CV_CHECKBOX,0);
    cvCreateTrackbar( "track2", NULL, &value2, 255, NULL);
    cvCreateButton("button5",callbackButton1,NULL,CV_RADIOBOX,0);
    cvCreateButton("button6",callbackButton2,NULL,CV_RADIOBOX,1);

    cvSetMouseCallback( "main2",on_mouse,NULL );

    IplImage* img1 = cvLoadImage("files/flower.jpg");
```

```
    IplImage* img2 = cvCreateImage(cvGetSize(img1),8,3);
    CvCapture* video = cvCaptureFromFile("files/hockey.avi");
    IplImage* img3 = cvCreateImage(cvGetSize(cvQueryFrame(video)),8,3);

    while(cvWaitKey(33) != 27)
    {
        cvAddS(img1,cvScalarAll(value),img2);
        cvAddS(cvQueryFrame(video),cvScalarAll(value2),img3);
        cvShowImage("main1",img2);
        cvShowImage("main2",img3);
    }

    cvDestroyAllWindows();
    cvReleaseImage(&img1);
    cvReleaseImage(&img2);
    cvReleaseImage(&img3);
    cvReleaseCapture(&video);
    return 0;
}
```

SetWindowProperty

void **cvSetWindowProperty**(const char* *name*, int *prop_id*, double *prop_value*)

Change the parameters of the window dynamically.

Parameters

- **name** – Name of the window.
- **prop_id** – Window's property to edit. The operation flags:
  - **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (`CV_WINDOW_NORMAL` or `CV_WINDOW_FULLSCREEN`).
  - **CV_WND_PROP_AUTOSIZE** Change if the user can resize the window (texttt {CV_WINDOW_NORMAL} or `CV_WINDOW_AUTOSIZE`).
  - **CV_WND_PROP_ASPECTRATIO** Change if the image's aspect ratio is preserved (texttt {CV_WINDOW_FREERATIO} or `CV_WINDOW_KEEPRATIO`).
- **prop_value** – New value of the Window's property. The operation flags:
  - **CV_WINDOW_NORMAL** Change the window in normal size, or allows the user to resize the window.
  - **CV_WINDOW_AUTOSIZE** The user cannot resize the window, the size is constrainted by the image displayed.
  - **CV_WINDOW_FULLSCREEN** Change the window to fullscreen.
  - **CV_WINDOW_FREERATIO** The image expends as much as it can (no ratio constraint)
  - **CV_WINDOW_KEEPRATIO** The ration image is respected.

The function ‘‘ cvSetWindowProperty‘‘ allows to change the window's properties.

GetWindowProperty

void **cvGetWindowProperty**(const char* *name*, int *prop_id*)

Get the parameters of the window.

Parameters

- **name** – Name of the window.
- **prop_id** – Window's property to retrive. The operation flags:
  - **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (`CV_WINDOW_NORMAL` or `CV_WINDOW_FULLSCREEN`).
  - **CV_WND_PROP_AUTOSIZE** Change if the user can resize the window (texttt {CV_WINDOW_NORMAL} or `CV_WINDOW_AUTOSIZE`).
  - **CV_WND_PROP_ASPECTRATIO** Change if the image's aspect ratio is preserved (texttt {CV_WINDOW_FREERATIO} or `CV_WINDOW_KEEPRATIO`).

See *SetWindowProperty* to know the meaning of the returned values.

The function ‘‘ cvGetWindowProperty‘‘ return window's properties.

FontQt

AddText

CvFont **cvFontQt** (const char* *nameFont*, int pointSize = *-1*, CvScalar color = *cvScalarAll(0)*, int weight = *CV_FONT_NORMAL*, int style = *CV_STYLE_NORMAL*, int spacing = *0*)

Create the font to be used to draw text on an image (with).

Parameters

- **nameFont** – Name of the font. The name should match the name of a system font (such as ‘‘Times’‘). If the font is not found, a default one will be used.
- **pointSize** – Size of the font. If not specified, equal zero or negative, the point size of the font is set to a system-dependent default value. Generally, this is 12 points.
- **color** – Color of the font in BGRA – A = 255 is fully transparent. Use the macro CV _ RGB for simplicity.
- **weight** – The operation flags:
  - **CV_FONT_LIGHT** Weight of 25
  - **CV_FONT_NORMAL** Weight of 50
  - **CV_FONT_DEMIBOLD** Weight of 63
  - **CV_FONT_BOLD** Weight of 75
  - **CV_FONT_BLACK** Weight of 87

  You can also specify a positive integer for more control.
- **style** – The operation flags:
  - **CV_STYLE_NORMAL** Font is normal
  - **CV_STYLE_ITALIC** Font is in italic
  - **CV_STYLE_OBLIQUE** Font is oblique
- **spacing** – Spacing between characters. Can be negative or positive

The function `cvFontQt` creates a CvFont object to be used with *AddText* . This CvFont is not compatible with cvPutText.

A basic usage of this function is:

```
CvFont font = cvFontQt(''Times'');
cvAddText( img1, ``Hello World !'', cvPoint(50,50), font);
```

AddText

void **cvAddText**(const CvArr* *img*, const char* *text*, CvPoint *location*, CvFont **font*)

Create the font to be used to draw text on an image

Parameters

- **img** – Image where the text should be drawn
- **text** – Text to write on the image
- **location** – Point(x,y) where the text should start on the image
- **font** – Font to use to draw the text

The function `cvAddText` draw *text* on the image *img* using a specific font *font* (see example *FontQt*)

DisplayOverlay

void **cvDisplayOverlay**(const char* *name*, const char* *text*, int *delay*)

Display text on the window's image as an overlay for delay milliseconds. This is not editing the image's data. The text is display on the top of the image.

Parameters

- **name** – Name of the window
- **text** – Overlay text to write on the window's image
- **delay** – Delay to display the overlay text. If this function is called before the previous overlay text time out, the timer is restarted and the text updated. . If this value is zero, the text never disapers.

The function `cvDisplayOverlay` aims at displaying useful information/tips on the window for a certain amount of time *delay* . This information is display on the top of the window.

DisplayStatusBar

void **cvDisplayStatusBar**(const char* *name*, const char* *text*, int *delayms*)

Display text on the window's statusbar as for delay milliseconds.

Parameters

- **name** – Name of the window
- **text** – Text to write on the window's statusbar
- **delay** – Delay to display the text. If this function is called before the previous text time out, the timer is restarted and the text updated. If this value is zero, the text never disapers.

The function `cvDisplayOverlay` aims at displaying useful information/tips on the window for a certain amount of time *delay* . This information is displayed on the window's statubar (the window must be created with `CV_GUI_EXPANDED` flags).

CreateOpenGLCallback

–

void **cvCreateOpenGLCallback**(const char* *window_name*, CvOpenGLCallback *callbackOpenGL*, void* userdata *CV_DEFAULT(NULL)*, double angle *CV_DEFAULT(-1)*, double zmin *CV_DEFAULT(-1)*, double zmax *CV_DEFAULT(-1)*)

Create a callback function called to draw OpenGL on top the the image display by windowname.

Parameters

- **window_name** – Name of the window
- **callbackOpenGL** – Pointer to the function to be called every frame. This function should be prototyped as `void Foo(*void);` .
- **userdata** – pointer passed to the callback function. *(Optional)*
- **angle** – Specifies the field of view angle, in degrees, in the y direction.. *(Optional - Default 45 degree)*
- **zmin** – Specifies the distance from the viewer to the near clipping plane (always positive). *(Optional - Default 0.01)*
- **zmax** – Specifies the distance from the viewer to the far clipping plane (always positive). *(Optional - Default 1000)*

The function `cvCreateOpenGLCallback` can be used to draw 3D data on the window. An example of callback could be:

```
void on_opengl(void* param)
{
    //draw scene here
    glLoadIdentity();

    glTranslated(0.0, 0.0, -1.0);

    glRotatef( 55, 1, 0, 0 );
    glRotatef( 45, 0, 1, 0 );
    glRotatef( 0, 0, 0, 1 );

    static const int coords[6][4][3] = {
        { { +1, -1, -1 }, { -1, -1, -1 }, { -1, +1, -1 }, { +1, +1, -1 } },
        { { +1, +1, -1 }, { -1, +1, -1 }, { -1, +1, +1 }, { +1, +1, +1 } },
        { { +1, -1, +1 }, { +1, -1, -1 }, { +1, +1, -1 }, { +1, +1, +1 } },
        { { -1, -1, -1 }, { -1, -1, +1 }, { -1, +1, +1 }, { -1, +1, -1 } },
        { { +1, -1, +1 }, { -1, -1, +1 }, { -1, -1, -1 }, { +1, -1, -1 } },
        { { -1, -1, +1 }, { +1, -1, +1 }, { +1, +1, +1 }, { -1, +1, +1 } }
    };

    for (int i = 0; i < 6; ++i) {
                glColor3ub( i*20, 100+i*10, i*42 );
                glBegin(GL_QUADS);
                for (int j = 0; j < 4; ++j) {
                        glVertex3d(0.2 * coords[i][j][0], 0.2 * coords[i][j][1], 0.2 * coords[i][j][2
                }
                glEnd();
    }
}
```

```
CV_EXTERN_C_FUNCPTR( *CvOpenGLCallback)(void* userdata));
```

SaveWindowParameters

–

void **cvSaveWindowParameters**(const char* *name*)

Save parameters of the window windowname.

Parameters

- **name** – Name of the window

The function `cvSaveWindowParameters` saves size, location, flags, trackbars' value, zoom and panning location of the window *window_name*

LoadWindowParameters

–

void **cvLoadWindowParameters**(const char* *name*)

Load parameters of the window windowname.

Parameters

- **name** – Name of the window

The function `cvLoadWindowParameters` load size, location, flags, trackbars' value, zoom and panning location of the window *window_name*

CreateButton

–

cvCreateButton(const char* button_name *CV_DEFAULT(NULL)*, CvButtonCallback on_change *CV_DEFAULT(NULL)*, void* userdata *CV_DEFAULT(NULL)*, int button_type *CV_DEFAULT(CV_PUSH_BUTTON)*, int initial_button_state *CV_DEFAULT(0)*)

Create a callback function called to draw OpenGL on top the the image display by windowname.

Parameters

- **button_name** – Name of the button *(if NULL, the name will be "button <number of boutton>")*
- **on_change** – Pointer to the function to be called every time the button changed its state. This function should be prototyped as `void Foo(int state,*void);` . *state* is the current state of the button. It could be -1 for a push button, 0 or 1 for a check/radio box button.
- **userdata** – pointer passed to the callback function. *(Optional)*

The `button_type` parameter can be : *(Optional – Will be a push button by default.)

- **CV_PUSH_BUTTON** The button will be a push button.
- **CV_CHECKBOX** The button will be a checkbox button.

- **CV_RADIOBOX** The button will be a radiobox button. The radiobox on the same buttonbar (same line) are exclusive; one on can be select at the time.
-
- **initial_button_state** Default state of the button. Use for checkbox and radiobox, its value could be 0 or 1. *(Optional)*

The function `cvCreateButton` attach button to the control panel. Each button is added to a buttonbar on the right of the last button. A new buttonbar is create if nothing was attached to the control panel before, or if the last element attached to the control panel was a trackbar.

Here are various example of `cvCreateButton` function call:

```
cvCreateButton(NULL,callbackButton);//create a push button "button 0", that will call callbackButton
cvCreateButton("button2",callbackButton,NULL,CV_CHECKBOX,0);
cvCreateButton("button3",callbackButton,&value);
cvCreateButton("button5",callbackButton1,NULL,CV_RADIOBOX);
cvCreateButton("button6",callbackButton2,NULL,CV_PUSH_BUTTON,1);
```

```
CV_EXTERN_C_FUNCPTR( *CvButtonCallback)(int state, void* userdata));
```

CHAPTER

SEVEN

CALIB3D. CAMERA CALIBRATION, POSE ESTIMATION AND STEREO

7.1 Camera Calibration and 3d Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters 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 $z \neq 0$):

$$\begin{array}{l} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\ x' = x/z \\ y' = y/z \\ u = f_x * x' + c_x \\ v = f_y * y' + c_y \end{array}$$

Real lenses usually have some distortion, mostly radial distortion and slight tangential distortion. So, the above model is extended as:

$$\begin{array}{l} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\ x' = x/z \\ y' = y/z \\ x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2 p_1 x' y' + p_2(r^2 + 2 x'^2) \\ y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y' \\ \text{where} \quad r^2 = x'^2 + y'^2 \\ u = f_x * x'' + c_x \\ v = f_y * y'' + c_y \end{array}$$

k_1 , k_2 , k_3 , k_4 , k_5 , k_6 are radial distortion coefficients, p_1 , p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the 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 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x , f_y , c_x and c_y need to be scaled appropriately).

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).
- Estimate the relative position and orientation of the stereo camera “heads” and compute the *rectification* transformation that makes the camera optical axes parallel.

CalcImageHomography

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

Calculates the homography matrix for an 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 calculates the homography matrix for the initial image transformation from image plane to the plane, defined by a 3D oblong object line (See _ _ Figure 6-10 _ _ in the OpenCV Guide 3D Reconstruction Chapter).

CalibrateCamera2

double **cvCalibrateCamera2** (const CvMat* *objectPoints*, const CvMat* *imagePoints*, const CvMat* *pointCounts*, CvSize *imageSize*, CvMat* *cameraMatrix*, CvMat* *distCoeffs*, CvMat* *rvecs=NULL*, CvMat* *tvecs=NULL*, int *flags=0*)

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

Parameters

- **objectPoints** – The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.
- **imagePoints** – The joint matrix of object points projections in the camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views
- **pointCounts** – Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints` (=N).
- **imageSize** – Size of the image, used only to initialize the intrinsic camera matrix
- **cameraMatrix** – The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. 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 before calling the function
- **distCoeffs** – The output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements
- **rvecs** – The output 3x *M* or *M* x3 1-channel, or 1x *M* or *M* x1 3-channel array of rotation vectors (see *Rodrigues2*), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0.. *M* -1)
- **tvecs** – The output 3x *M* or *M* x3 1-channel, or 1x *M* or *M* x1 3-channel array of translation vectors, estimated for each pattern view.
- **flags** – Different flags, may be 0 or combination of the following values:
  - **CV_CALIB_USE_INTRINSIC_GUESS** `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`imageSize` 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 just to estimate the extrinsic parameters. Use *FindExtrinsicCameraParams2* instead.
  - **CV_CALIB_FIX_PRINCIPAL_POINT** The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.
  - **CV_CALIB_FIX_ASPECT_RATIO** The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix` . When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

- **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.
- **CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6** Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.
- **CV_CALIB_RATIONAL_MODEL** Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute only 5 distortion coefficients.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. 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 chessboard as a calibration rig (see *FindChessboardCorners*). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).
2. The initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using *FindExtrinsicCameraParams2*
3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points `imagePoints` and the projected (using the current estimates for camera parameters and the poses) object points `objectPoints` ; see *ProjectPoints2* .

The function returns the final re-projection error. Note: if you're using a non-square (=non-NxN) grid and `findChessboardCorners()` for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2 - 0.5, h/2 - 0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probably used `patternSize=cvSize(rows,cols)` , but should use `patternSize=cvSize(cols,rows)` in *FindChessboardCorners* .

See also: *FindChessboardCorners* , *FindExtrinsicCameraParams2* , `initCameraMatrix2D()` , *StereoCalibrate* , *Undistort2*

ComputeCorrespondEpilines

void **cvComputeCorrespondEpilines**(const CvMat* *points*, int *whichImage*, const CvMat* *F*, CvMat* *lines*)

For points in one image of a 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
- **whichImage** – Index of the image (1 or 2) that contains the `points`
- **F** – The fundamental matrix that can be estimated using *FindFundamentalMat* or *StereoRectify* .

- **lines** – The output epilines, a `3xN` or `Nx3` array. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see *FindFundamentalMat*), line $l^{(2)}_i$ in the second image for the point $p^{(1)}_i$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l^{(2)}_i = Fp^{(1)}_i$$

and, vice versa, when `whichImage=2` , $l^{(1)}_i$ is computed from $p^{(2)}_i$ as:

$$l^{(1)}_i = F^T p^{(2)}_i$$

Line coefficients are defined up to a scale. They are normalized, such that $a^2_i + b^2_i = 1$.

ConvertPointsHomogeneous

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 to 4

The function converts 2D or 3D points from/to homogeneous coordinates, or simply copies or transposes the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$\begin{array}{l} (x,y[,z],w)->(x',y'[,z']) \\ \text{where} \\ x'=x/w \\ y'=y/w \\ z'=z/w \quad \text{(if output is 3D)} \end{array}$$

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

Note 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 $\texttt{N} \ge 5$, or to use multi-channel `Nx1` or `1xN` arrays.

CreatePOSITObject

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

Initializes a structure containing object information.

Parameters

- **points** – Pointer to the points of the 3D object model
- **point_count** – Number of object points

The function allocates memory for the object structure and computes the object inverse matrix.

The preprocessed 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.

An object is defined as a set of points given in a coordinate system. The function *POSIT* 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 *ReleasePOSITObject* must be called to free memory.

CreateStereoBMState

CvStereoBMState* **cvCreateStereoBMState**(int *preset=CV_STEREO_BM_BASIC*, int *numberOfDisparities=0*)

Creates block matching stereo correspondence structure.

Parameters

- **preset** – ID of one of the pre-defined parameter sets. Any of the parameters can be overridden after creating the structure. Values are
  - **CV_STEREO_BM_BASIC** Parameters suitable for general cameras
  - **CV_STEREO_BM_FISH_EYE** Parameters suitable for wide-angle cameras
  - **CV_STEREO_BM_NARROW** Parameters suitable for narrow-angle cameras
- **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 creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to *FindStereoCorrespondenceBM* .

CreateStereoGCState

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` $\leq$ *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 creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to *FindStereoCorrespondenceGC* .

CvStereoBMState

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
    int       uniquenessRatio;// invalidate disparity at pixels where there are other close matches
                              // with different disparity
    int       speckleWindowSize; // the maximum area of speckles to remove
                                 // (set to 0 to disable speckle filtering)
    int       speckleRange; // acceptable range of disparity variation in each connected component

    int trySmallerWindows; // not used
    CvRect roi1, roi2; // clipping ROIs

    int disp12MaxDiff; // maximum allowed disparity difference in the left-right check

    // internal data
    ...
}
CvStereoBMState;
```

preFilterType
type of the prefilter, `CV_STEREO_BM_NORMALIZED_RESPONSE` or the default and the recommended `CV_STEREO_BM_XSOBEL` , int

preFilterSize
~5x5..21x21, int

preFilterCap
up to ~31, int

SADWindowSize
Could be 5x5..21x21 or higher, but with 21x21 or smaller windows the processing speed is much higher, int

minDisparity
minimum disparity (=0), int

numberOfDisparities
maximum disparity - minimum disparity, int

textureThreshold
the textureness threshold. That is, if the sum of absolute values of x-derivatives computed over `SADWindowSize` by `SADWindowSize` pixel neighborhood is smaller than the parameter, no disparity is computed at the pixel, int

uniquenessRatio
the minimum margin in percents between the best (minimum) cost function value and the second best value to accept the computed disparity, int

speckleWindowSize
the maximum area of speckles to remove (set to 0 to disable speckle filtering), int

speckleRange
acceptable range of disparity variation in each connected component, int

trySmallerWindows
not used currently (0), int

roi1, roi2
These are the clipping ROIs for the left and the right images. The function *StereoRectify* returns the largest rectangles in the left and right images where after the rectification all the pixels are valid. If you copy those rectangles to the `CvStereoBMState` structure, the stereo correspondence function will automatically clear out the pixels outside of the "valid" disparity rectangle computed by *GetValidDisparityROI* . Thus you will get more "invalid disparity" pixels than usual, but the remaining pixels are more probable to be valid.

disp12MaxDiff
The maximum allowed difference between the explicitly computed left-to-right disparity map and the implicitly (by *ValidateDisparity*) computed right-to-left disparity. If for some pixel the difference is larger than the specified threshold, the disparity at the pixel is invalidated. By default this parameter is set to (-1), which means that the left-right check is not performed.

The block matching stereo correspondence algorithm, by Kurt Konolige, is very fast single-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 readability 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.

CvStereoGCState

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 correspondence 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 *CreateStereoGCState* and then override necessary parameters if needed.

DecomposeProjectionMatrix

void **cvDecomposeProjectionMatrix**(const CvMat **projMatrix*, CvMat **cameraMatrix*, CvMat **rotMatrix*, CvMat **transVect*, CvMat **rotMatrX=NULL*, CvMat **rotMatrY=NULL*, CvMat **rotMatrZ=NULL*, CvPoint3D64f **eulerAngles=NULL*)

Decomposes the projection matrix into a rotation matrix and a camera matrix.

Parameters

- **projMatrix** – The 3x4 input projection matrix P
- **cameraMatrix** – The output 3x3 camera matrix K
- **rotMatrix** – The output 3x3 external rotation matrix R
- **transVect** – The output 4x1 translation vector T
- **rotMatrX** – Optional 3x3 rotation matrix around x-axis
- **rotMatrY** – Optional 3x3 rotation matrix around y-axis
- **rotMatrZ** – Optional 3x3 rotation matrix around z-axis
- **eulerAngles** – Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on *RQDecomp3x3* .

DrawChessboardCorners

void **cvDrawChessboardCorners**(CvArr* *image*, CvSize *patternSize*, CvPoint2D32f* *corners*, int *count*, int *patternWasFound*)

Renders the detected chessboard corners.

Parameters

- **image** – The destination image; it must be an 8-bit color image
- **patternSize** – The number of inner corners per chessboard row and column. (patternSize = cv::Size(points _ per _ row,points _ per _ column) = cv::Size(rows,columns))
- **corners** – The array of corners detected, this should be the output from findChessboardCorners wrapped in a cv::Mat().
- **count** – The number of corners
- **patternWasFound** – Indicates whether the complete board was found $(\neq 0)$ or not $(= 0)$. One may just pass the return value *FindChessboardCorners* here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

FindChessboardCorners

int **cvFindChessboardCorners** (const void* *image*, CvSize *patternSize*, CvPoint2D32f* *corners*, int* *cornerCount=NULL*, int *flags=CV_CALIB_CB_ADAPTIVE_THRESH*)

Finds the positions of the internal corners of the chessboard.

Parameters

- **image** – Source chessboard view; it must be an 8-bit grayscale or color image
- **patternSize** – The number of inner corners per chessboard row and column (patternSize = cvSize(points _ per _ row,points _ per _ colum) = cvSize(columns,rows))
- **corners** – The output array of corners detected
- **cornerCount** – The output corner counter. If it is not NULL, it stores 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 and white, rather than a fixed threshold level (computed from the average image brightness).
  - **CV_CALIB_CB_NORMALIZE_IMAGE** normalize the image gamma with *EqualizeHist* 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.
  - **CALIB_CB_FAST_CHECK Runs a fast check on the image that looks for chessboard corners, and shortc**
    no chessboard is observed.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of 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 *FindCornerSubPix* .

Sample usage of detecting and drawing chessboard corners:

```
Size patternsize(8,6); //interior number of corners
Mat gray = ....; //source image
vector<Point2f> corners; //this will be filled by the detected corners

//CALIB_CB_FAST_CHECK saves a lot of time on images
//that don't contain any chessboard corners
bool patternfound = findChessboardCorners(gray, patternsize, corners,
        CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE
        + CALIB_CB_FAST_CHECK);

if(patternfound)
  cornerSubPix(gray, corners, Size(11, 11), Size(-1, -1),
    TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

drawChessboardCorners(img, patternsize, Mat(corners), patternfound);
```

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

FindExtrinsicCameraParams2

void **cvFindExtrinsicCameraParams2**(const CvMat* *objectPoints*, const CvMat* *imagePoints*, const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, CvMat* *rvec*, CvMat* *tvec*, int *useExtrinsicGuess=0*)

Finds the object pose from the 3D-2D point correspondences

Parameters

- **objectPoints** – The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points.
- **imagePoints** – The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points.
- **cameraMatrix** – The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **rvec** – The output rotation vector (see *Rodrigues2*) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system
- **tvec** – The output translation vector
- **useExtrinsicGuess** – If true (1), the function will use the provided `rvec` and `tvec` as the initial approximations of the rotation and translation vectors, respectively, and will further optimize them.

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using *ProjectPoints2*) `objectPoints` .

The function's counterpart in the C++ API is

FindFundamentalMat

int **cvFindFundamentalMat**(const CvMat* *points1*, const CvMat* *points2*, CvMat* *fundamentalMatrix*, int *method=CV_FM_RANSAC*, double *param1=1.*, double *param2=0.99*, CvMat* *status=NULL*)

Calculates the fundamental matrix from the corresponding points in two images.

Parameters

- **points1** – Array of `N` points from the first image. It can be `2xN, Nx2, 3xN` or `Nx3` 1-channel array or `1xN` or `Nx1` 2- or 3-channel array . 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`
- **fundamentalMatrix** – The output fundamental matrix or matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices)

- **method** – Method for computing the fundamental matrix
  - **CV_FM_7POINT** for a 7-point algorithm. $N = 7$
  - **CV_FM_8POINT** for an 8-point algorithm. $N \geq 8$
  - **CV_FM_RANSAC** for the RANSAC algorithm. $N \geq 8$
  - **CV_FM_LMEDS** for the LMedS algorithm. $N \geq 8$
- **param1** – The parameter is used for RANSAC. 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. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise
- **param2** – The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct
- **status** – The optional output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. 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:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the 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 . Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to *ComputeCorrespondEpilines* that finds the epipolar lines corresponding to the specified points. It can also be passed to *StereoRectifyUncalibrated* to compute the rectification transformation.

```
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.fl[i*2] = <x,,1,i,,>;
    points1->data.fl[i*2+1] = <y,,1,i,,>;
    points2->data.fl[i*2] = <x,,2,i,,>;
    points2->data.fl[i*2+1] = <y,,2,i,,>;
}

fundamental_matrix = cvCreateMat(3,3,CV_32FC1);
int fm_count = cvFindFundamentalMat( points1,points2,fundamental_matrix,
                                     CV_FM_RANSAC,1.0,0.99,status );
```

FindHomography

void **cvFindHomography**(const CvMat* *srcPoints*, const CvMat* *dstPoints*, CvMat* H int *method=0*, double *ransacReprojThreshold=3*, CvMat* *status=NULL*)

Finds the perspective transformation between two planes.

Parameters

- **srcPoints** – Coordinates of the points in the original plane, 2xN, Nx2, 3xN or Nx3 1-channel array (the latter two are for representation in homogeneous coordinates), where N is the number of points. 1xN or Nx1 2- or 3-channel array can also be passed.
- **dstPoints** – Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 1-channel, or 1xN or Nx1 2- or 3-channel array.
- **H** – The output 3x3 homography matrix
- **method** – The method used to computed homography matrix; one of the following:
  - **0** a regular method using all the points
  - **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 (used in the RANSAC method only). That is, if

  $$\|\texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

  then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.
- **status** – The optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). *Note that the input mask values are ignored.*

The function finds the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter `method` is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, `RANSAC` and `LMeDS` , try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection 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 the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method `RANSAC` can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method `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, where can be only some small noise present, 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 so that $h_{33} = 1$.

See also: *GetAffineTransform* , *GetPerspectiveTransform* , *EstimateRigidMotion* , *WarpPerspective* , *PerspectiveTransform*

FindStereoCorrespondenceBM

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, or 32-bit floating-point disparity map of the same size as input images. In the first case the computed disparities are represented as fixed-point numbers with 4 fractional bits (i.e. the computed disparity values are multiplied by 16 and rounded to integers).
- **state** – Stereo correspondence structure.

The function cvFindStereoCorrespondenceBM computes disparity map for the input rectified stereo pair. Invalid pixels (for which disparity can not be computed) are set to `state->minDisparity - 1` (or to `(state->minDisparity-1)*16` in the case of 16-bit fixed-point disparity map)

FindStereoCorrespondenceGC

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 computes disparity maps for the input rectified stereo pair. Note that the left disparity image will contain values in the following range:

$$-\texttt{state->numberOfDisparities} - \texttt{state->minDisparity} < dispLeft(x, y) \leq -\texttt{state->minDisparity},$$

or

$$dispLeft(x, y) == \texttt{CV\_STEREO\_GC\_OCCLUSION}$$

and for the right disparity image the following will be true:

$$\texttt{state->minDisparity} \le dispRight(x, y) < \texttt{state->minDisparity} + \texttt{state->numberOfDisparities}$$

or

$$dispRight(x, y) == \texttt{CV\_STEREO\_GC\_OCCLUSION}$$

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 used:

```
// 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.pgm", disparity_left_visual );
```

GetOptimalNewCameraMatrix

void **cvGetOptimalNewCameraMatrix**(const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, CvSize *imageSize*, double *alpha*, CvMat* *newCameraMatrix*, CvSize *newImageSize=cvSize(0*, 0), CvRect* *validPixROI=0*)

Returns the new camera matrix based on the free scaling parameter

Parameters

- **cameraMatrix** – The input camera matrix
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **imageSize** – The original image size
- **alpha** – The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see *StereoRectify*
- **newCameraMatrix** – The output new camera matrix.
- **newImageSize** – The image size after rectification. By default it will be set to `imageSize` .
- **validPixROI** – The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in *StereoRectify*

The function computes the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0` , keep all the original image pixels if there is valuable information in the corners `alpha=1` , or get something in between. When `alpha>0` , the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to *InitUndistortRectifyMap* to produce the maps for *Remap* .

InitIntrinsicParams2D

void **cvInitIntrinsicParams2D**(const CvMat* *objectPoints*, const CvMat* *imagePoints*, const CvMat* *npoints*, CvSize *imageSize*, CvMat* *cameraMatrix*, double *aspectRatio=1.*)

Finds the initial camera matrix from the 3D-2D point correspondences

Parameters

- **objectPoints** – The joint array of object points; see *CalibrateCamera2*
- **imagePoints** – The joint array of object point projections; see *CalibrateCamera2*
- **npoints** – The array of point counts; see *CalibrateCamera2*
- **imageSize** – The image size in pixels; used to initialize the principal point
- **cameraMatrix** – The output camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **aspectRatio** – If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

InitUndistortMap

void **cvInitUndistortMap**(const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, CvArr* *map1*, CvArr* *map2*)

Computes an undistortion map.

Parameters

- **cameraMatrix** – The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **map1** – The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient
- **map2** – The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function is a simplified variant of *InitUndistortRectifyMap* where the rectification transformation `R` is identity matrix and `newCameraMatrix=cameraMatrix` .

InitUndistortRectifyMap

void **cvInitUndistortRectifyMap**(const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, const CvMat* *R*, const CvMat* *newCameraMatrix*, CvArr* *map1*, CvArr* *map2*)

Computes the undistortion and rectification transformation map.

Parameters

- **cameraMatrix** – The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **R** – The optional rectification transformation in object space (3x3 matrix). `R1` or `R2` , computed by *StereoRectify* can be passed here. If the matrix is NULL , the identity transformation is assumed
- **newCameraMatrix** – The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$
- **map1** – The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient
- **map2** – The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for *Remap* . The undistorted image will look like the original, as if it was captured with a camera with camera matrix `=newCameraMatrix` and zero distortion. In the case of monocular camera `newCameraMatrix` is usually equal to `cameraMatrix` , or it can be computed by *GetOptimalNewCameraMatrix* for a better control over scaling. In the case of stereo camera `newCameraMatrix` is normally set to `P1` or `P2` computed by *StereoRectify* .

Also, this new camera will be oriented differently in the coordinate space, according to `R` . That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images become horizontal and have the same y-coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by *Remap* . That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$\begin{array}{l} x \leftarrow (u - {c'}_x)/{f'}_x \\ y \leftarrow (v - {c'}_y)/{f'}_y \\ {[X\,Y\,W]}^T \leftarrow R^{-1} * [x\,y\,1]^T \\ x' \leftarrow X/W \\ y' \leftarrow Y/W \\ x" \leftarrow x' (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2 x'^2) \\ y" \leftarrow y' (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y' \\ map_x(u,v) \leftarrow x" f_x + c_x \\ map_y(u,v) \leftarrow y" f_y + c_y \end{array}$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after *StereoRectify* , which in its turn is called after *StereoCalibrate* . But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using *StereoRectifyUncalibrated* . For each camera the function computes homography `H` as the rectification transformation in pixel domain, not a rotation matrix `R` in 3D space. The `R` can be computed from `H` as

$$\texttt{R} = \texttt{cameraMatrix} ^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

POSIT

void **cvPOSIT**(CvPOSITObject* *posit_object*, CvPoint2D32f* *imagePoints*, double *focal_length*, CvTermCriteria *criteria*, CvMatr32f *rotationMatrix*, CvVect32f *translation_vector*)

Implements the POSIT algorithm.

Parameters

- **posit_object** – Pointer to the object structure
- **imagePoints** – 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
- **rotationMatrix** – Matrix of rotations
- **translation_vector** – Translation vector

The function implements the POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using the camera calibration functions. At every iteration of the algorithm a new perspective projection of the 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.

ProjectPoints2

void **cvProjectPoints2**(const CvMat* *objectPoints*, const CvMat* *rvec*, const CvMat* *tvec*, const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, CvMat* *imagePoints*, CvMat* *dpdrot=NULL*, CvMat* *dpdt=NULL*, CvMat* *dpdf=NULL*, CvMat* *dpdc=NULL*, CvMat* *dpddist=NULL*)

Project 3D points on to an image plane.

Parameters

- **objectPoints** – The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel , where N is the number of points in the view
- **rvec** – The rotation vector, see *Rodrigues2*
- **tvec** – The translation vector
- **cameraMatrix** – The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **imagePoints** – The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel
- **dpdrot** – Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector
- **dpdt** – Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector
- **dpdf** – Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y
- **dpdc** – Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y
- **dpddist** – Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function 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 coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in *CalibrateCamera2* , *FindExtrinsicCameraParams2* and *StereoCalibrate* . The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)` , or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

ReprojectImageTo3D

void **cvReprojectImageTo3D**(const CvArr* *disparity*, CvArr* *_3dImage*, const CvMat* *Q*, int *handleMissingValues=0*)

Reprojects disparity image to 3D space.

Parameters

- **disparity** – The input single-channel 16-bit signed or 32-bit floating-point disparity image

- **_3dImage** – The output 3-channel floating-point image of the same size as `disparity` . Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)` , computed from the disparity map.
- **Q** – The 4×4 perspective transformation matrix that can be obtained with *StereoRectify*
- **handleMissingValues** – If true, when the pixels with the minimal disparity (that corresponds to the outliers; see *FindStereoCorrespondenceBM*) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \texttt{Q} * [x\ y\ \texttt{disparity}(x,y)\ 1]^T$$
$$\texttt{\_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by *StereoRectify* . To reproject a sparse set of points {(x,y,d),...} to 3D space, use *PerspectiveTransform* .

RQDecomp3x3

void **cvRQDecomp3x3**(const CvMat **M*, CvMat **R*, CvMat **Q*, CvMat **Qx=NULL*, CvMat **Qy=NULL*, CvMat **Qz=NULL*, CvPoint3D64f **eulerAngles=NULL*)

Computes the 'RQ' decomposition of 3x3 matrices.

Parameters

- **M** – The 3x3 input matrix
- **R** – The output 3x3 upper-triangular matrix
- **Q** – The output 3x3 orthogonal matrix
- **Qx** – Optional 3x3 rotation matrix around x-axis
- **Qy** – Optional 3x3 rotation matrix around y-axis
- **Qz** – Optional 3x3 rotation matrix around z-axis
- **eulerAngles** – Optional three Euler angles of rotation

The function computes a RQ decomposition using the given rotations. This function is used in *DecomposeProjectionMatrix* to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

ReleasePOSITObject

void **cvReleasePOSITObject**(CvPOSITObject** *posit_object*)

Deallocates a 3D object structure.

Parameters

- **posit_object** – Double pointer to `CvPOSIT` structure

The function releases memory previously allocated by the function *CreatePOSITObject* .

ReleaseStereoBMState

void **cvReleaseStereoBMState** (CvStereoBMState** *state*)
Releases block matching stereo correspondence structure.

Parameters

- **state** – Double pointer to the released structure.

The function releases the stereo correspondence structure and all the associated internal buffers.

ReleaseStereoGCState

void **cvReleaseStereoGCState** (CvStereoGCState** *state*)
Releases the state structure of the graph cut-based stereo correspondence algorithm.

Parameters

- **state** – Double pointer to the released structure.

The function releases the stereo correspondence structure and all the associated internal buffers.

Rodrigues2

int **cvRodrigues2** (const CvMat* *src*, CvMat* *dst*, CvMat* *jacobian=0*)
Converts a rotation matrix to a 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 with respect to the input array components

$$\begin{array}{l} \theta \leftarrow norm(r) \\ r \leftarrow r/\theta \\ R = \cos\theta I + (1-\cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} \end{array}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like *CalibrateCamera2* , *StereoCalibrate* or *FindExtrinsicCameraParams2* .

StereoCalibrate

double **cvStereoCalibrate** (const CvMat* *objectPoints*, const CvMat* *imagePoints1*, const CvMat* *imagePoints2*, const CvMat* *pointCounts*, CvMat* *cameraMatrix1*, CvMat* *distCoeffs1*, CvMat* *cameraMatrix2*, CvMat* *distCoeffs2*, CvSize *imageSize*, 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

- **objectPoints** – The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.
- **imagePoints1** – The joint matrix of object points projections in the first camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views
- **imagePoints2** – The joint matrix of object points projections in the second camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views
- **pointCounts** – Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints*` (=N).
- **cameraMatrix1** – The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of `CV_CALIB_USE_INTRINSIC_GUESS` , `CV_CALIB_FIX_ASPECT_RATIO` , `CV_CALIB_FIX_INTRINSIC` or `CV_CALIB_FIX_FOCAL_LENGTH` are specified, some or all of the matrices' components must be initialized; see the flags description
- **distCoeffs1** – The input/output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements.
- **cameraMatrix2** – The input/output second camera matrix, as cameraMatrix1.
- **distCoeffs2** – The input/output lens distortion coefficients for the second camera, as `distCoeffs1` .
- **imageSize** – Size of the image, used only to initialize intrinsic camera matrix.
- **R** – The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.
- **T** – The output translation vector between the cameras' coordinate systems.
- **E** – The optional output essential matrix.
- **F** – The optional output fundamental matrix.
- **term_crit** – The termination criteria for the iterative optimization algorithm.
- **flags** – Different flags, may be 0 or combination of the following values:
  - **CV_CALIB_FIX_INTRINSIC** If it is set, `cameraMatrix?` , as well as `distCoeffs?` are fixed, so that only `R, T, E` and `F` are estimated.
  - **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** $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.
- **CV_CALIB_FIX_ASPECT_RATIO** $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.
- **CV_CALIB_SAME_FOCAL_LENGTH** Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$
- **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients for each camera are set to zeros and fixed there.
- **CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6** Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.
- **CV_CALIB_RATIONAL_MODEL** Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute only 5 distortion coefficients.

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientation 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 *FindExtrinsicCameraParams2*), obviously, those poses will relate to each other, i.e. given (R_1 , T_1) it should be possible to compute (R_2 , T_2) - 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:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector $T : T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} EcameraMatrix1^{-1}$$

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 *CalibrateCamera2*), it is recommended to do so and then pass `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 `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which are usually reasonable assumptions.

Similarly to *CalibrateCamera2* , the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

StereoRectify

void **cvStereoRectify** (const CvMat* *cameraMatrix1*, const CvMat* *cameraMatrix2*, const CvMat* *distCoeffs1*, const CvMat* *distCoeffs2*, CvSize *imageSize*, const CvMat* *R*, const CvMat* *T*, CvMat* *R1*, CvMat* *R2*, CvMat* *P1*, CvMat* *P2*, CvMat* *Q=0*, int *flags=CV_CALIB_ZERO_DISPARITY*, double *alpha=-1*, CvSize *newImageSize=cvSize(0*, 0), CvRect* *roi1=0*, CvRect* *roi2=0*)

Computes rectification transforms for each head of a calibrated stereo camera.

Parameters

- **cameraMatrix1** – The first camera matrix.
- **cameraMatrix2** – The second camera matrix.
- **distCoeffs1** – The first camera distortion parameters.
- **distCoeffs2** – The second camera distortion parameters.
- **imageSize** – 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.
- **R2** (*R1,*) – The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.
- **P2** (*P1,*) – The output 3×4 projection matrices in the new (rectified) coordinate systems.
- **Q** – The output 4×4 disparity-to-depth mapping matrix, see `reprojectImageTo3D()` .
- **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 may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.
- **alpha** – The free scaling parameter. If it is -1 , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.
- **newImageSize** – The new image resolution after rectification. The same size should be passed to *InitUndistortRectifyMap* , see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize` . Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.
- **roi2** (*roi1,*) – The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0` , the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function 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 `stereoCalibrate()` and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

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:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ 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:

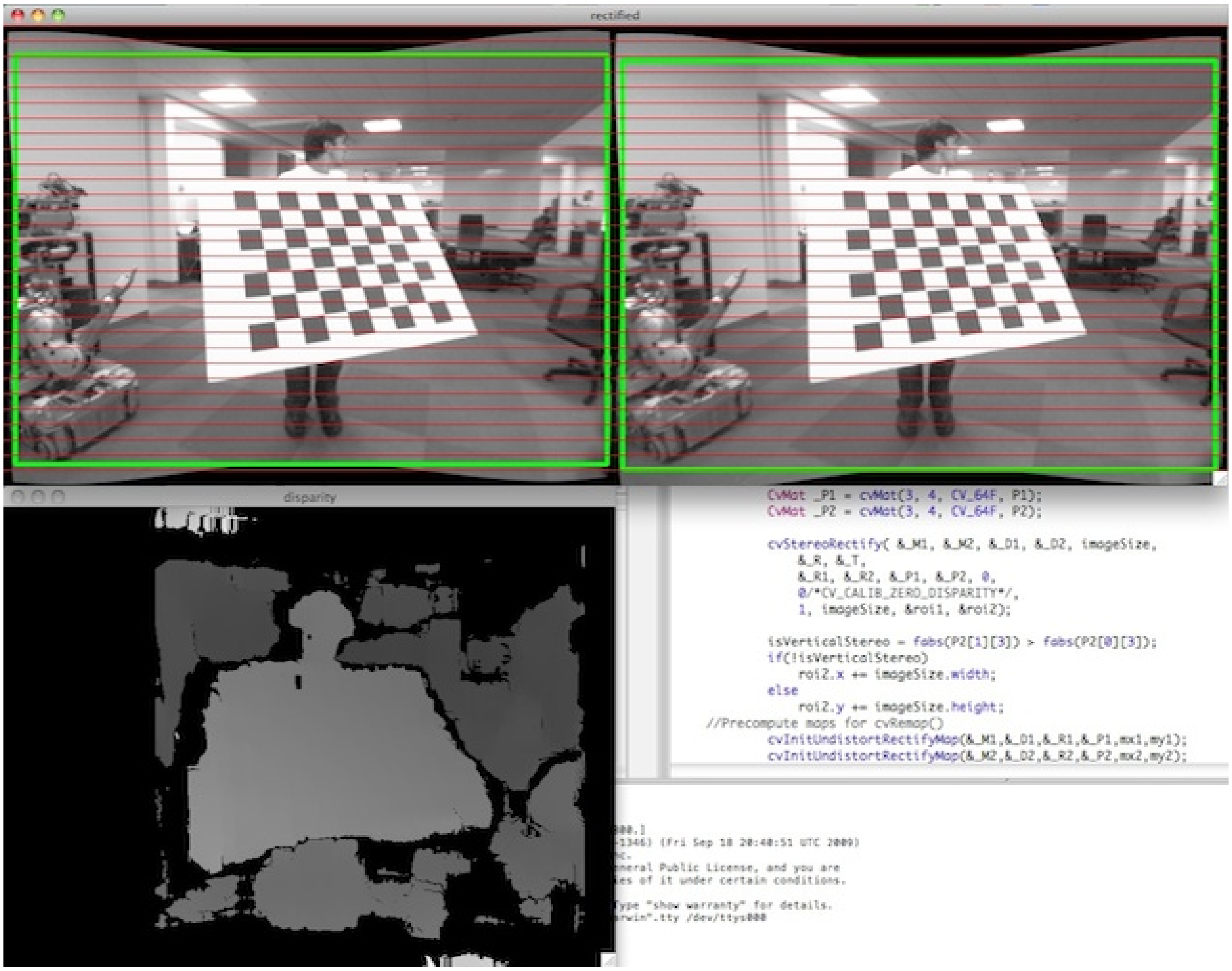
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `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. The matrices, together with `R1` and `R2` , can then be passed to *InitUndistortRectifyMap* to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



StereoRectifyUncalibrated

void **cvStereoRectifyUncalibrated**(const CvMat* *points1*, const CvMat* *points2*, const CvMat* *F*, CvSize *imageSize*, CvMat* *H1*, CvMat* *H2*, double *threshold=5*)

Computes rectification transform for uncalibrated stereo camera.

Parameters

- **points2** (*points1,*) – The 2 arrays of corresponding 2D points. The same formats as in *FindFundamentalMat* are supported
- **F** – The input fundamental matrix. It can be computed from the same set of point pairs using *FindFundamentalMat* .
- **imageSize** – Size of the image.
- **H2** (*H1,*) – The output rectification homography matrices for the first and for the second images.
- **threshold** – The 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 $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function 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 *StereoRectify* 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 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 *CalibrateCamera2* and then the images can be corrected using *Undistort2* , or just the point coordinates can be corrected with *UndistortPoints* .

Undistort2

void **cvUndistort2** (const CvArr* *src*, CvArr* *dst*, const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, const CvMat* *newCameraMatrix=0*)

Transforms an image to compensate for lens distortion.

Parameters

- **src** – The input (distorted) image
- **dst** – The output (corrected) image; will have the same size and the same type as `src`
- **cameraMatrix** – The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of *InitUndistortRectifyMap* (with unity `R`) and *Remap* (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix` . You can use *GetOptimalNewCameraMatrix* to compute the appropriate `newCameraMatrix` , depending on your requirements.

The camera matrix and the distortion parameters can be determined using *CalibrateCamera2* . If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

UndistortPoints

void **cvUndistortPoints** (const CvMat* *src*, CvMat* *dst*, const CvMat* *cameraMatrix*, const CvMat* *distCoeffs*, const CvMat* *R=NULL*, const CvMat* *P=NULL*)

Computes the ideal point coordinates from the observed point coordinates.

Parameters

- **src** – The observed point coordinates, 1xN or Nx1 2-channel (CV _ 32FC2 or CV _ 64FC2).
- **dst** – The output ideal point coordinates, after undistortion and reverse perspective transformation , same format as `src` .

- **cameraMatrix** – The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** – The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **R** – The rectification transformation in object space (3x3 matrix). `R1` or `R2` , computed by `StereoRectify()` can be passed here. If the matrix is empty, the identity transformation is used
- **P** – The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2` , computed by `StereoRectify()` can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to *Undistort2* and *InitUndistortRectifyMap* , but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to *ProjectPoints2* (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (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 both for a stereo camera head or for monocular camera (when R is NULL).