The OpenCV 2.x Python Reference Manual

Release 2.3

June 21, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

Starting with release 2.0, OpenCV has a new Python interface. This replaces the previous SWIG-based Python interface .

Some highlights of the new bindings:

- single import of all of OpenCV using `import cv`
- OpenCV functions no longer have the "cv" prefix
- simple types like CvRect and CvScalar use Python tuples
- sharing of Image storage, so image transport between OpenCV and other systems (e.g. numpy and ROS) is very efficient
- complete documentation for the Python functions

This cookbook section contains a few illustrative examples of OpenCV Python code.

1.1 Cookbook

Here is a collection of code fragments demonstrating some features of the OpenCV Python bindings.

Convert an image

Resize an image

To resize an image in OpenCV, create a destination image of the appropriate size, then call *Resize* .

Compute the Laplacian

Using GoodFeaturesToTrack

To find the 10 strongest corner features in an image, use *GoodFeaturesToTrack* like this:

Using GetSubRect

GetSubRect returns a rectangular part of another image. It does this without copying any data.

Using CreateMat, and accessing an element

ROS image message to OpenCV

See this tutorial: Using CvBridge to convert between ROS images And OpenCV images .

PIL Image to OpenCV

(For details on PIL see the PIL handbook .)

OpenCV to PIL Image

NumPy and OpenCV

Using the array interface , to use an OpenCV CvMat in NumPy:

and to use a NumPy array in OpenCV:

also, most OpenCV functions can work on NumPy arrays directly, for example:

Given a 2D array, the *fromarray* function (or the implicit version shown above) returns a single-channel *CvMat* of the same size. For a 3D array of size $j \times k \times l$, it returns a *CvMat* sized $j \times k$ with l channels.

Alternatively, use *fromarray* with the `allowND` option to always return a *CvMatND* .

OpenCV to pygame

To convert an OpenCV image to a pygame surface:

OpenCV and OpenEXR

Using OpenEXR's Python bindings you can make a simple image viewer:

```
import OpenEXR, Imath, cv
filename = "GoldenGate.exr"
exrimage = OpenEXR.InputFile(filename)

dw = exrimage.header()['dataWindow']
(width, height) = (dw.max.x - dw.min.x + 1, dw.max.y - dw.min.y + 1)

def fromstr(s):
    mat = cv.CreateMat(height, width, cv.CV_32FC1)
    cv.SetData(mat, s)
    return mat

pt = Imath.PixelType(Imath.PixelType.FLOAT)
(r, g, b) = [fromstr(s) for s in exrimage.channels("RGB", pt)]

bgr = cv.CreateMat(height, width, cv.CV_32FC3)
cv.Merge(b, g, r, None, bgr)

cv.ShowImage(filename, bgr)
cv.WaitKey()
```

CHAPTER

TWO

CORE. THE CORE FUNCTIONALITY

2.1 Basic Structures

CvPoint

class `CvPoint`

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

2D point, represented as a tuple `(x, y)`, where x and y are integers.

CvPoint2D32f

class `CvPoint2D32f`

2D point with floating-point coordinates

2D point, represented as a tuple `(x, y)`, where x and y are floats.

CvPoint3D32f

class `CvPoint3D32f`

3D point with floating-point coordinates

3D point, represented as a tuple `(x, y, z)`, where x, y and z are floats.

CvPoint2D64f

class `CvPoint2D64f`

2D point with double precision floating-point coordinates

2D point, represented as a tuple `(x, y)`, where x and y are floats.

CvPoint3D64f

class `CvPoint3D64f`

3D point with double precision floating-point coordinates

3D point, represented as a tuple `(x, y, z)`, where x, y and z are floats.

CvSize

class CvSize

Pixel-accurate size of a rectangle.

Size of a rectangle, represented as a tuple `(width, height)` , where width and height are integers.

CvSize2D32f

class CvSize2D32f

Sub-pixel accurate size of a rectangle.

Size of a rectangle, represented as a tuple `(width, height)` , where width and height are floats.

CvRect

class CvRect

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

Rectangle, represented as a tuple `(x, y, width, height)` , where all are integers.

CvScalar

class CvScalar

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

CvScalar is always represented as a 4-tuple.

CvTermCriteria

class CvTermCriteria

Termination criteria for iterative algorithms.

Represented by a tuple `(type, max_iter, epsilon)` .

type
: `CV_TERMCRIT_ITER` , `CV_TERMCRIT_EPS` or `CV_TERMCRIT_ITER | CV_TERMCRIT_EPS`

max_iter
: Maximum number of iterations

epsilon
: Required accuracy

```
(cv.CV_TERMCRIT_ITER, 10, 0)                      # terminate after 10 iterations
(cv.CV_TERMCRIT_EPS, 0, 0.01)                     # terminate when epsilon reaches 0.01
(cv.CV_TERMCRIT_ITER | cv.CV_TERMCRIT_EPS, 10, 0.01) # terminate as soon as either condition is met
```

CvMat

class CvMat

A multi-channel 2D matrix. Created by *CreateMat* , *LoadImageM* , *CreateMatHeader* , *fromarray* .

type
A CvMat signature containing the type of elements and flags, int

step
Full row length in bytes, int

rows
Number of rows, int

cols
Number of columns, int

tostring() → str
Returns the contents of the CvMat as a single string.

CvMatND

class CvMatND

Multi-dimensional dense multi-channel array.

type
A CvMatND signature combining the type of elements and flags, int

tostring() → str
Returns the contents of the CvMatND as a single string.

IplImage

class IplImage

The *IplImage* object was inherited from the Intel Image Processing Library, in which the format is native. OpenCV only supports a subset of possible *IplImage* formats.

nChannels
Number of channels, int.

width
Image width in pixels

height
Image height in pixels

depth
Pixel depth in bits. 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

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

tostring() → str
Returns the contents of the CvMatND as a single string.

CvArr

class CvArr

Arbitrary array

`CvArr` is used *only* as a function parameter to specify that the parameter can be:

- an *IplImage*
- a *CvMat*
- any other type that exports the array interface

2.2 Operations on Arrays

AbsDiff

AbsDiff(*src1*, *src2*, *dst*) → None
Calculates absolute difference between two arrays.

Parameters

- **src1** (`CvArr`) – The first source array
- **src2** (`CvArr`) – The second source array
- **dst** (`CvArr`) – The destination array

The function calculates absolute difference between two arrays.

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

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

AbsDiffS

AbsDiffS(*src*, *dst*, *value*) → None
Calculates absolute difference between an array and a scalar.

Parameters

- **src** (CvArr) – The source array
- **dst** (CvArr) – The destination array
- **value** (CvScalar) – The scalar

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

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

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

Add

Add(*src1*, *src2*, *dst*, *mask=NULL*) → None

Computes the per-element sum of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

AddS(*src*, *value*, *dst*, *mask=NULL*) → None

Computes the sum of an array and a scalar.

Parameters

- **src** (CvArr) – The source array
- **value** (CvScalar) – Added scalar
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

AddWeighted(*src1*, *alpha*, *src2*, *beta*, *gamma*, *dst*) → None

Computes the weighted sum of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **alpha** (*float*) – Weight for the first array elements
- **src2** (CvArr) – The second source array
- **beta** (*float*) – Weight for the second array elements
- **dst** (CvArr) – The destination array
- **gamma** (*float*) – 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

And(*src1*, *src2*, *dst*, *mask=NULL*) → None

Calculates per-element bit-wise conjunction of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

AndS(*src*, *value*, *dst*, *mask=NULL*) → None

Calculates per-element bit-wise conjunction of an array and a scalar.

Parameters

- **src** (CvArr) – The source array
- **value** (CvScalar) – Scalar to use in the operation
- **dst** (CvArr) – The destination array

- **mask** (CvArr) – 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.

Avg

Avg(*arr*, *mask=NULL*) → CvScalar

Calculates average (mean) of array elements.

Parameters

- **arr** (CvArr) – The array
- **mask** (CvArr) – The optional operation mask

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

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

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

AvgSdv(*arr*, *mask=NULL*)-> *(mean*, *stdDev*)

Calculates average (mean) of array elements.

Parameters

- **arr** (CvArr) – The array
- **mask** (CvArr) – The optional operation mask
- **mean** (CvScalar) – Mean value, a CvScalar
- **stdDev** (CvScalar) – Standard deviation, a CvScalar

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

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

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

CalcCovarMatrix(*vects*, *covMat*, *avg*, *flags*) → None

Calculates covariance matrix of a set of vectors.

Parameters

- **vects** (cvarr_count) – 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
- **covMat** (CvArr) – The output covariance matrix that should be floating-point and square
- **avg** (CvArr) – The input or output (depending on the flags) array - the mean (average) vector of the input vectors
- **flags** (*int*) – 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 count × 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

CartToPolar(*x*, *y*, *magnitude*, *angle=NULL*, *angleInDegrees=0*) → None

Calculates the magnitude and/or angle of 2d vectors.

Parameters

- **x** (CvArr) – The array of x-coordinates
- **y** (CvArr) – The array of y-coordinates
- **magnitude** (CvArr) – The destination array of magnitudes, may be set to NULL if it is not needed
- **angle** (CvArr) – The destination array of angles, may be set to NULL if it is not needed. The angles are measured in radians (0 to 2π) or in degrees (0 to 360 degrees).
- **angleInDegrees** (*int*) – 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

Cbrt(*value*) → float

Calculates the cubic root

Parameters **value** (*float*) – 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

ClearND(*arr*, *idx*) → None

Clears a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx** (*sequence of int*) – 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

CloneImage(*image*) → copy

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

Parameters **image** (IplImage) – The original image

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

CloneMat

CloneMat(*mat*) → copy
Creates a full matrix copy.

Parameters **mat** (CvMat) – Matrix to be copied

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

CloneMatND

CloneMatND(*mat*) → copy
Creates full copy of a multi-dimensional array and returns a pointer to the copy.

Parameters **mat** (CvMatND) – Input array

Cmp

Cmp(*src1*, *src2*, *dst*, *cmpOp*) → None
Performs per-element comparison of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array. Both source arrays must have a single channel.
- **dst** (CvArr) – The destination array, must have 8u or 8s type
- **cmpOp** (*int*) – 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

CmpS(*src*, *value*, *dst*, *cmpOp*) → None
Performs per-element comparison of an array and a scalar.

Parameters

- **src** (CvArr) – The source array, must have a single channel
- **value** (*float*) – The scalar value to compare each array element with

- **dst** (CvArr) – The destination array, must have 8u or 8s type
- **cmpOp** (*int*) – 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).

Convert

Convert(*src*, *dst*) → None

Converts one array to another.

Parameters

- **src** (CvArr) – Source array
- **dst** (CvArr) – Destination array

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.

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

ConvertScale

ConvertScale(*src*, *dst*, *scale=1.0*, *shift=0.0*) → None

Converts one array to another with optional linear transformation.

Parameters

- **src** (CvArr) – Source array
- **dst** (CvArr) – Destination array
- **scale** (*float*) – Scale factor
- **shift** (*float*) – 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

ConvertScaleAbs(*src*, *dst*, *scale=1.0*, *shift=0.0*) → None

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

Parameters

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

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

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

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

CvtScaleAbs(*src*, *dst*, *scale=1.0*, *shift=0.0*) → None

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:

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

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

Copy(*src*, *dst*, *mask=NULL*) → None

Copies one array to another.

Parameters

- **src** (CvArr) – The source array

- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

CountNonZero(*arr*) → int

Counts non-zero array elements.

Parameters **arr** (CvArr) – 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

CreateData(*arr*) → None

Allocates array data

Parameters **arr** (CvArr) – 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

CreateImage(*size*, *depth*, *channels*) → image

Creates an image header and allocates the image data.

Parameters

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

CreateImageHeader

CreateImageHeader(*size*, *depth*, *channels*) → image

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

Parameters

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

CreateMat

CreateMat(*rows*, *cols*, *type*) → mat

Creates a matrix header and allocates the matrix data.

Parameters

- **rows** (*int*) – Number of rows in the matrix
- **cols** (*int*) – Number of columns in the matrix
- **type** (*int*) – 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.

CreateMatHeader

CreateMatHeader(*rows*, *cols*, *type*) → mat

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

Parameters

- **rows** (*int*) – Number of rows in the matrix
- **cols** (*int*) – Number of columns in the matrix
- **type** (*int*) – 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

CreateMatND(*dims*, *type*) → None

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

Parameters

- **dims** (*sequence of int*) – List or tuple of array dimensions, up to 32 in length.
- **type** (*int*) – Type of array elements, see *CreateMat* .

This is a short form for:

CreateMatNDHeader

CreateMatNDHeader(*dims*, *type*) → None

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

Parameters

- **dims** (*sequence of int*) – List or tuple of array dimensions, up to 32 in length.
- **type** (*int*) – 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* .

CrossProduct

CrossProduct(*src1*, *src2*, *dst*) → None

Calculates the cross product of two 3D vectors.

Parameters

- **src1** (CvArr) – The first source vector
- **src2** (CvArr) – The second source vector
- **dst** (CvArr) – 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

DCT(*src*, *dst*, *flags*) → None

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

Parameters

- **src** (CvArr) – Source array, real 1D or 2D array
- **dst** (CvArr) – Destination array of the same size and same type as the source
- **flags** (*int*) – 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

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

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

DFT(*src*, *dst*, *flags*, *nonzeroRows=0*) → None

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

Parameters

- **src** (CvArr) – Source array, real or complex
- **dst** (CvArr) – Destination array of the same size and same type as the source
- **flags** (*int*) – 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** (*int*) – 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 $\times$ N elements:

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

Inverse Fourier transform of 2D vector of M $\times$ 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:

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

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.

Det

Det(*mat*) $\rightarrow$ double

Returns the determinant of a matrix.

Parameters **mat** (`CvArr`) – 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

Div(*src1*, *src2*, *dst*, *scale*) → None

Performs per-element division of two arrays.

Parameters

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

The function divides one array by another:

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

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

DotProduct

DotProduct(*src1*, *src2*) → double

Calculates the dot product of two arrays in Euclidian metrics.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – 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

EigenVV(*mat*, *evects*, *evals*, *eps*, *lowindex*, *highindex*) → None

Computes eigenvalues and eigenvectors of a symmetric matrix.

Parameters

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

- **highindex** (*int*) – 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

Exp(*src*, *dst*) → None

Calculates the exponent of every array element.

Parameters

- **src** (`CvArr`) – The source array
- **dst** (`CvArr`) – 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

FastArctan(*y*, *x*) → float

Calculates the angle of a 2D vector.

Parameters

- **x** (*float*) – x-coordinate of 2D vector
- **y** (*float*) – 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

Flip(*src*, *dst=NULL*, *flipMode=0*) → None

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

Parameters

- **src** (`CvArr`) – Source array

- **dst** (CvArr) – Destination array. If dst = NULL the flipping is done in place.
- **flipMode** (*int*) – 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)

fromarray

fromarray(*object*, *allowND = False*) → CvMat

Create a CvMat from an object that supports the array interface.

Parameters

- **object** – Any object that supports the array interface
- **allowND** – If true, will return a CvMatND

If the object supports the array interface , return a *CvMat* (allowND = False) or *CvMatND* (allowND = True).

If allowND = False , then the object's array must be either 2D or 3D. If it is 2D, then the returned CvMat has a single channel. If it is 3D, then the returned CvMat will have N channels, where N is the last dimension of the array. In this case, N cannot be greater than OpenCV's channel limit, CV_CN_MAX .

If allowND = True , then fromarray returns a single-channel *CvMatND* with the same shape as the original array.

For example, NumPy arrays support the array interface, so can be converted to OpenCV objects:

GEMM

GEMM(*src1*, *src2*, *alphs*, *src3*, *beta*, *dst*, *tABC=0*) → None

Performs generalized matrix multiplication.

Parameters

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

- **tABC** (*int*) – 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:

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

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

Get1D

Get1D(*arr*, *idx*) → scalar

Return a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx** (*int*) – Zero-based element index

Return a specific array element. Array must have dimension 3.

Get2D

Get2D(*arr*, *idx0*, *idx1*) → scalar

Return a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx0** (*int*) – Zero-based element row index
- **idx1** (*int*) – Zero-based element column index

Return a specific array element. Array must have dimension 2.

Get3D

Get3D(*arr*, *idx0*, *idx1*, *idx2*) → scalar

Return a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx0** (*int*) – Zero-based element index
- **idx1** (*int*) – Zero-based element index
- **idx2** (*int*) – Zero-based element index

Return a specific array element. Array must have dimension 3.

GetND

GetND(*arr*, *indices*) → scalar

Return a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **indices** (*sequence of int*) – List of zero-based element indices

Return a specific array element. The length of array indices must be the same as the dimension of the array.

GetCol

GetCol(*arr*, *col*) → submat

Returns array column.

Parameters

- **arr** (CvArr) – Input array
- **col** (*int*) – Zero-based index of the selected column
- **submat** (CvMat) – resulting single-column array

The function GetCol returns a single column from the input array.

GetCols

GetCols(*arr*, *startCol*, *endCol*) → submat

Returns array column span.

Parameters

- **arr** (CvArr) – Input array
- **startCol** (*int*) – Zero-based index of the starting column (inclusive) of the span
- **endCol** (*int*) – Zero-based index of the ending column (exclusive) of the span
- **submat** (CvMat) – resulting multi-column array

The function GetCols returns a column span from the input array.

GetDiag

GetDiag(*arr*, *diag=0*) → submat

Returns one of array diagonals.

Parameters

- **arr** (CvArr) – Input array
- **submat** (CvMat) – Pointer to the resulting sub-array header
- **diag** (*int*) – 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.

GetDims

GetDims(*arr*) → list
Returns list of array dimensions

Parameters **arr** (CvArr) – Input array

The function returns a list of array dimensions. In the case of IplImage or *CvMat* it always returns a list of length 2.

GetElemType

GetElemType(*arr*) → int
Returns type of array elements.

Parameters **arr** (CvArr) – Input array

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

GetImage

GetImage(*arr*) → iplimage
Returns image header for arbitrary array.

Parameters **arr** (CvMat) – Input array

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

GetImageCOI(*image*) → channel
Returns the index of the channel of interest.

Parameters **image** (IplImage) – A pointer to the image header

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

GetImageROI

GetImageROI(*image*) → CvRect
Returns the image ROI.

Parameters **image** (IplImage) – A pointer to the image header

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

GetMat

GetMat(*arr*, *allowND=0*) → cvmat
Returns matrix header for arbitrary array.

Parameters

- **arr** (IplImage) – Input array
- **allowND** (*int*) – 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.

GetOptimalDFTSize

GetOptimalDFTSize(*size0*) → int

Returns optimal DFT size for a given vector size.

Parameters **size0** (*int*) – 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)

GetReal1D

GetReal1D(*arr*, *idx0*) → float

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

Parameters

- **arr** (CvArr) – Input array. Must have a single channel.
- **idx0** (*int*) – 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

GetReal2D(*arr*, *idx0*, *idx1*) → float

Return a specific element of single-channel 2D array.

Parameters

- **arr** (CvArr) – Input array. Must have a single channel.

- **idx0** (*int*) – The first zero-based component of the element index
- **idx1** (*int*) – 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

GetReal3D(*arr*, *idx0*, *idx1*, *idx2*) → float

Return a specific element of single-channel array.

Parameters

- **arr** (CvArr) – Input array. Must have a single channel.
- **idx0** (*int*) – The first zero-based component of the element index
- **idx1** (*int*) – The second zero-based component of the element index
- **idx2** (*int*) – 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

GetRealND(*arr*, *idx*) → float

Return a specific element of single-channel array.

Parameters

- **arr** (CvArr) – Input array. Must have a single channel.
- **idx** (*sequence of int*) – 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

GetRow(*arr*, *row*) → submat

Returns array row.

Parameters

- **arr** (CvArr) – Input array
- **row** (*int*) – Zero-based index of the selected row
- **submat** (CvMat) – resulting single-row array

The function GetRow returns a single row from the input array.

GetRows

GetRows(*arr*, *startRow*, *endRow*, *deltaRow=1*) → submat

Returns array row span.

Parameters

- **arr** (CvArr) – Input array
- **startRow** (*int*) – Zero-based index of the starting row (inclusive) of the span
- **endRow** (*int*) – Zero-based index of the ending row (exclusive) of the span
- **deltaRow** (*int*) – Index step in the row span.
- **submat** (CvMat) – resulting multi-row array

The function GetRows returns a row span from the input array.

GetSize

GetSize(*arr*) → CvSize

Returns size of matrix or image ROI.

Parameters **arr** (CvArr) – 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

GetSubRect(*arr*, *rect*) → cvmat

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

Parameters

- **arr** (CvArr) – Input array
- **rect** (CvRect) – 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

InRange(*src*, *lower*, *upper*, *dst*) → None

Checks that array elements lie between the elements of two other arrays.

Parameters

- **src** (CvArr) – The first source array
- **lower** (CvArr) – The inclusive lower boundary array
- **upper** (CvArr) – The exclusive upper boundary array

- **dst** (CvArr) – 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,

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

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

InRangeS(*src*, *lower*, *upper*, *dst*) → None

Checks that array elements lie between two scalars.

Parameters

- **src** (CvArr) – The first source array
- **lower** (CvScalar) – The inclusive lower boundary
- **upper** (CvScalar) – The exclusive upper boundary
- **dst** (CvArr) – 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,

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

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).

InvSqrt

InvSqrt(*value*) → float

Calculates the inverse square root.

Parameters **value** (*float*) – 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

Invert(*src*, *dst*, *method=CV_LU*) → double

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

IsInf(*value*) → int

Determines if the argument is Infinity.

Parameters **value** (*float*) – The input floating-point value

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

IsNaN

IsNaN(*value*) → int

Determines if the argument is Not A Number.

Parameters **value** (*float*) – The input floating-point value

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

LUT

LUT(*src*, *dst*, *lut*) → None

Performs a look-up table transform of an array.

Parameters

- **src** (`CvArr`) – Source array of 8-bit elements
- **dst** (`CvArr`) – Destination array of a given depth and of the same number of channels as the source array
- **lut** (`CvArr`) – 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

Log(*src*, *dst*) → None

Calculates the natural logarithm of every array element's absolute value.

Parameters

- **src** (CvArr) – The source array
- **dst** (CvArr) – 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

Mahalonobis(*vec1*, *vec2*, *mat*) → None

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).

Max

Max(*src1*, *src2*, *dst*) → None

Finds per-element maximum of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – 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

MaxS(*src*, *value*, *dst*) → None

Finds per-element maximum of array and scalar.

Parameters

- **src** (CvArr) – The first source array
- **value** (*float*) – The scalar value
- **dst** (CvArr) – 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

Merge(*src0*, *src1*, *src2*, *src3*, *dst*) → None

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

Parameters

- **src0** (CvArr) – Input channel 0
- **src1** (CvArr) – Input channel 1
- **src2** (CvArr) – Input channel 2
- **src3** (CvArr) – Input channel 3
- **dst** (CvArr) – 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

Min(*src1*, *src2*, *dst*) → None

Finds per-element minimum of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – 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

MinMaxLoc(*arr*, *mask=NULL)-> (minVal*, *maxVal*, *minLoc*, *maxLoc*)

Finds global minimum and maximum in array or subarray.

Parameters

- **arr** (CvArr) – The source array, single-channel or multi-channel with COI set
- **minVal** (*float*) – Pointer to returned minimum value
- **maxVal** (*float*) – Pointer to returned maximum value
- **minLoc** (CvPoint) – Pointer to returned minimum location
- **maxLoc** (CvPoint) – Pointer to returned maximum location
- **mask** (CvArr) – 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

MinS(*src*, *value*, *dst*) → None

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

Parameters

- **src** (CvArr) – The first source array
- **value** (*float*) – The scalar value
- **dst** (CvArr) – 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

MixChannels(*src*, *dst*, *fromTo*) → None

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

Parameters

- **src** (cvarr_count) – Input arrays
- **dst** (cvarr_count) – Destination arrays
- **fromTo** (intpair) – The array of pairs of indices of the planes copied. Each pair fromTo[k]=(i,j) means that i-th plane from src is copied to the j-th plane in dst , where continuous plane numbering is used both in the input array list and the output array list. As a special case, when the fromTo[k][0] is negative, the corresponding output plane j

  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:

```
rgba = cv.CreateMat(100, 100, cv.CV_8UC4)
bgr =  cv.CreateMat(100, 100, cv.CV_8UC3)
alpha = cv.CreateMat(100, 100, cv.CV_8UC1)
cv.Set(rgba, (1,2,3,4))
cv.MixChannels([rgba], [bgr, alpha], [
    (0, 2),     # rgba[0] -> bgr[2]
    (1, 1),     # rgba[1] -> bgr[1]
    (2, 0),     # rgba[2] -> bgr[0]
    (3, 3)      # rgba[3] -> alpha[0]
])
```

MulAddS

Synonym for *ScaleAdd* .

Mul

Mul(*src1*, *src2*, *dst*, *scale*) → None

Calculates the per-element product of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array
- **scale** (*float*) – 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

MulSpectrums(*src1*, *src2*, *dst*, *flags*) → None

Performs per-element multiplication of two Fourier spectrums.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array of the same type and the same size as the source arrays
- **flags** (*int*) – 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

MulTransposed(*src*, *dst*, *order*, *delta=NULL*, *scale*) → None

Calculates the product of an array and a transposed array.

Parameters

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

Norm(*arr1*, *arr2*, *normType=CV_L2*, *mask=NULL*) → double

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

Parameters

- **arr1** (CvArr) – The first source image
- **arr2** (CvArr) – 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** (*int*) – Type of norm, see the discussion
- **mask** (CvArr) – 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

Not(*src*, *dst*) → None

Performs per-element bit-wise inversion of array elements.

Parameters

- **src** (CvArr) – The source array
- **dst** (CvArr) – The destination array

The function Not inverses every bit of every array element:

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

Or

Or(*src1*, *src2*, *dst*, *mask=NULL*) → None

Calculates per-element bit-wise disjunction of two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

OrS(*src*, *value*, *dst*, *mask=NULL*) → None

Calculates a per-element bit-wise disjunction of an array and a scalar.

Parameters

- **src** (CvArr) – The source array
- **value** (CvScalar) – Scalar to use in the operation
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

PerspectiveTransform(*src*, *dst*, *mat*) → None

Performs perspective matrix transformation of a vector array.

Parameters

- **src** (CvArr) – The source three-channel floating-point array
- **dst** (CvArr) – The destination three-channel floating-point array
- **mat** (CvMat) – 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

PolarToCart(*magnitude*, *angle*, *x*, *y*, *angleInDegrees=0*) → None

Calculates Cartesian coordinates of 2d vectors represented in polar form.

Parameters

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

Pow(*src*, *dst*, *power*) → None

Raises every array element to a power.

Parameters

- **src** (CvArr) – The source array
- **dst** (CvArr) – The destination array, should be the same type as the source
- **power** (*float*) – 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:

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

RNG

RNG(*seed=-1LL*) → CvRNG

Initializes a random number generator state.

Parameters **seed** (`int64`) – 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

RandArr(*rng*, *arr*, *distType*, *param1*, *param2*) → None

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

Parameters

- **rng** (CvRNG) – RNG state initialized by *RNG*
- **arr** (CvArr) – The destination array
- **distType** (*int*) – Distribution type
  - **CV_RAND_UNI** uniform distribution
  - **CV_RAND_NORMAL** normal or Gaussian distribution
- **param1** (CvScalar) – 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** (CvScalar) – 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.

RandInt

RandInt(*rng*) → unsigned

Returns a 32-bit unsigned integer and updates RNG.

Parameters **rng** (CvRNG) – 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.

RandReal

RandReal(*rng*) → double

Returns a floating-point random number and updates RNG.

Parameters **rng** (CvRNG) – RNG state initialized by *RNG*

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

Reduce

Reduce(*src*, *dst*, *dim=-1*, *op=CV_REDUCE_SUM*) → None

Reduces a matrix to a vector.

Parameters

- **src** (CvArr) – The input matrix.
- **dst** (CvArr) – The output single-row/single-column vector that accumulates somehow all the matrix rows/columns.
- **dim** (*int*) – 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** (*int*) – 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.

Repeat

Repeat(*src*, *dst*) → None

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

Parameters

- **src** (CvArr) – Source array, image or matrix
- **dst** (CvArr) – 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

ResetImageROI(*image*) → None

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

Parameters **image** (IplImage) – A pointer to the image header

This produces a similar result to the following

```
cv.SetImageROI(image, (0, 0, image.width, image.height))
cv.SetImageCOI(image, 0)
```

Reshape

Reshape(*arr*, *newCn*, *newRows=0*) → cvmat

Changes shape of matrix/image without copying data.

Parameters

- **arr** (CvArr) – Input array
- **newCn** (*int*) – New number of channels. 'newCn = 0' means that the number of channels remains unchanged.
- **newRows** (*int*) – 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.

ReshapeMatND

ReshapeMatND(*arr*, *newCn*, *newDims*) → cvmat

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

Parameters

- **arr** (CvMat) – Input array
- **newCn** (*int*) – New number of channels. `newCn` $= 0$ means that the number of channels remains unchanged.
- **newDims** (*sequence of int*) – List of new dimensions.

Returns a new *CvMatND* that shares the same data as `arr` but has different dimensions or number of channels. The only requirement is that the total length of the data is unchanged.

Round

Round(*value*) → int

Converts a floating-point number to the nearest integer value.

Parameters **value** (*float*) – The input floating-point value

On some architectures this function is much faster than the standard cast operations. 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.

Floor

Floor(*value*) → int

Converts a floating-point number to the nearest integer value that is not larger than the argument.

Parameters **value** (*float*) – The input floating-point value

On some architectures this function is much faster than the standard cast operations. 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.

Ceil

Ceil(*value*) → int

Converts a floating-point number to the nearest integer value that is not smaller than the argument.

Parameters **value** (*float*) – The input floating-point value

On some architectures this function is much faster than the standard cast operations. 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

ScaleAdd(*src1*, *scale*, *src2*, *dst*) → None

Calculates the sum of a scaled array and another array.

Parameters

- **src1** (CvArr) – The first source array
- **scale** (CvScalar) – Scale factor for the first array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – 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

Set(*arr*, *value*, *mask=NULL*) → None

Sets every element of an array to a given value.

Parameters

- **arr** (CvArr) – The destination array
- **value** (CvScalar) – Fill value
- **mask** (CvArr) – 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.

Set1D

Set1D(*arr*, *idx*, *value*) → None

Set a specific array element.

Parameters

- **arr** (CvArr) – Input array

- **idx** (*int*) – Zero-based element index
- **value** (CvScalar) – The value to assign to the element

Sets a specific array element. Array must have dimension 1.

Set2D

Set2D(*arr*, *idx0*, *idx1*, *value*) → None
Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx0** (*int*) – Zero-based element row index
- **idx1** (*int*) – Zero-based element column index
- **value** (CvScalar) – The value to assign to the element

Sets a specific array element. Array must have dimension 2.

Set3D

Set3D(*arr*, *idx0*, *idx1*, *idx2*, *value*) → None
Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx0** (*int*) – Zero-based element index
- **idx1** (*int*) – Zero-based element index
- **idx2** (*int*) – Zero-based element index
- **value** (CvScalar) – The value to assign to the element

Sets a specific array element. Array must have dimension 3.

SetND

SetND(*arr*, *indices*, *value*) → None
Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **indices** (*sequence of int*) – List of zero-based element indices
- **value** (CvScalar) – The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

SetData

SetData(*arr*, *data*, *step*) → None

Assigns user data to the array header.

Parameters

- **arr** (CvArr) – Array header
- **data** (*object*) – User data
- **step** (*int*) – 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

SetIdentity(*mat*, *value=1*) → None

Initializes a scaled identity matrix.

Parameters

- **mat** (CvArr) – The matrix to initialize (not necesserily square)
- **value** (CvScalar) – 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

SetImageCOI(*image*, *coi*) → None

Sets the channel of interest in an IplImage.

Parameters

- **image** (IplImage) – A pointer to the image header
- **coi** (*int*) – 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

SetImageROI(*image*, *rect*) → None

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

Parameters

- **image** (IplImage) – A pointer to the image header
- **rect** (CvRect) – 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.

SetReal1D

SetReal1D(*arr*, *idx*, *value*) → None

Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx** (*int*) – Zero-based element index
- **value** (*float*) – The value to assign to the element

Sets a specific array element. Array must have dimension 1.

SetReal2D

SetReal2D(*arr*, *idx0*, *idx1*, *value*) → None

Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx0** (*int*) – Zero-based element row index
- **idx1** (*int*) – Zero-based element column index
- **value** (*float*) – The value to assign to the element

Sets a specific array element. Array must have dimension 2.

SetReal3D

SetReal3D(*arr*, *idx0*, *idx1*, *idx2*, *value*) → None

Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **idx0** (*int*) – Zero-based element index
- **idx1** (*int*) – Zero-based element index
- **idx2** (*int*) – Zero-based element index
- **value** (*float*) – The value to assign to the element

Sets a specific array element. Array must have dimension 3.

SetRealND

SetRealND(*arr*, *indices*, *value*) → None
Set a specific array element.

Parameters

- **arr** (CvArr) – Input array
- **indices** (*sequence of int*) – List of zero-based element indices
- **value** (*float*) – The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

SetZero

SetZero(*arr*) → None
Clears the array.

Parameters **arr** (CvArr) – 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

Solve(*A*, *B*, *X*, *method=CV_LU*) → None
Solves a linear system or least-squares problem.

Parameters

- **A** (CvArr) – The source matrix
- **B** (CvArr) – The right-hand part of the linear system
- **X** (CvArr) – The output solution
- **method** (*int*) – 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

SolveCubic(*coeffs*, *roots*) → None
Finds the real roots of a cubic equation.

Parameters

- **coeffs** (CvMat) – The equation coefficients, an array of 3 or 4 elements

- **roots** (CvMat) – 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

Split(*src*, *dst0*, *dst1*, *dst2*, *dst3*) → None

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

Parameters

- **src** (CvArr) – Source array
- **dst0** (CvArr) – Destination channel 0
- **dst1** (CvArr) – Destination channel 1
- **dst2** (CvArr) – Destination channel 2
- **dst3** (CvArr) – 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

Sqrt(*value*) → float

Calculates the square root.

Parameters **value** (*float*) – 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

Sub(*src1*, *src2*, *dst*, *mask=NULL*) → None

Computes the per-element difference between two arrays.

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array

- **mask** (CvArr) – 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

SubRS(*src*, *value*, *dst*, *mask=NULL*) → None

Computes the difference between a scalar and an array.

Parameters

- **src** (CvArr) – The first source array
- **value** (CvScalar) – Scalar to subtract from
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

SubS(*src*, *value*, *dst*, *mask=NULL*) → None

Computes the difference between an array and a scalar.

Parameters

- **src** (CvArr) – The source array
- **value** (CvScalar) – Subtracted scalar
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

Sum(*arr*) → CvScalar

Adds up array elements.

Parameters **arr** (CvArr) – 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

SVBkSb(*W*, *U*, *V*, *B*, *X*, *flags*) → None

Performs singular value back substitution.

Parameters

- **W** (CvArr) – Matrix or vector of singular values
- **U** (CvArr) – Left orthogonal matrix (tranposed, perhaps)
- **V** (CvArr) – Right orthogonal matrix (tranposed, perhaps)
- **B** (CvArr) – 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** (CvArr) – The destination matrix: result of back substitution
- **flags** (*int*) – 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

SVD(*A*, *W*, *U = None*, *V = None*, *flags=0*) → None

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

Parameters

- **A** (CvArr) – Source $\texttt{M} \times \texttt{N}$ matrix
- **W** (CvArr) – 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** (CvArr) – 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** (CvArr) – 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** (*int*) – 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

Trace(*mat*) → CvScalar

Returns the trace of a matrix.

Parameters **mat** (CvArr) – 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

Transform(*src*, *dst*, *transmat*, *shiftvec=NULL*) → None

Performs matrix transformation of every array element.

Parameters

- **src** (CvArr) – The first source array
- **dst** (CvArr) – The destination array
- **transmat** (CvMat) – Transformation matrix
- **shiftvec** (CvMat) – 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 `M` × `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 `M` × $(N + 1)$ 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

Transpose(*src*, *dst*) → None

Transposes a matrix.

Parameters

- **src** (CvArr) – The source matrix
- **dst** (CvArr) – 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

Xor(*src1*, *src2*, *dst*, *mask=NULL*) → None

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

Parameters

- **src1** (CvArr) – The first source array
- **src2** (CvArr) – The second source array
- **dst** (CvArr) – The destination array

- **mask** (CvArr) – 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

XorS(*src*, *value*, *dst*, *mask=NULL*) → None

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

Parameters

- **src** (CvArr) – The source array
- **value** (CvScalar) – Scalar to use in the operation
- **dst** (CvArr) – The destination array
- **mask** (CvArr) – 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

mGet

mGet(*mat*, *row*, *col*) → double

Returns the particular element of single-channel floating-point matrix.

Parameters

- **mat** (CvMat) – Input matrix
- **row** (*int*) – The zero-based index of row
- **col** (*int*) – 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

mSet(*mat*, *row*, *col*, *value*) → None

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

Parameters

- **mat** (CvMat) – The matrix

- **row** (*int*) – The zero-based index of row
- **col** (*int*) – The zero-based index of column
- **value** (*float*) – 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.

2.3 Dynamic Structures

CvMemStorage

class `CvMemStorage`

Growing memory storage.

Many OpenCV functions use a given storage area for their results and working storage. These storage areas can be created using *CreateMemStorage* . OpenCV Python tracks the objects occupying a CvMemStorage, and automatically releases the CvMemStorage when there are no objects referring to it. For this reason, there is explicit function to release a CvMemStorage.

CvSeq

class `CvSeq`

Growable sequence of elements.

Many OpenCV functions return a CvSeq object. The CvSeq obect is a sequence, so these are all legal:

```
seq = cv.FindContours(scribble, storage, cv.CV_RETR_CCOMP, cv.CV_CHAIN_APPROX_SIMPLE)
# seq is a sequence of point pairs
print len(seq)
# FindContours returns a sequence of (x,y) points, so to print them out:
for (x,y) in seq:
   print (x,y)
print seq[10]             # tenth entry in the seqeuence
print seq[::-1]           # reversed sequence
print sorted(list(seq))   # sorted sequence
```

Also, a CvSeq object has methods `h_next()` , `h_prev()` , `v_next()` and `v_prev()` . Some OpenCV functions (for example *FindContours*) can return multiple CvSeq objects, connected by these relations. In this case the methods return the other sequences. If no relation between sequences exists, then the methods return `None` .

CvSet

class `CvSet`

Collection of nodes.

Some OpenCV functions return a CvSet object. The CvSet obect is iterable, for example:

```
for i in s:
  print i
print set(s)
print list(s)
```

CloneSeq

CloneSeq(*seq*, *storage*) → None
Creates a copy of a sequence.

Parameters

- **seq** (CvSeq) – Sequence
- **storage** (CvMemStorage) – 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.

CreateMemStorage

CreateMemStorage(*blockSize = 0*) → memstorage
Creates memory storage.

Parameters **blockSize** (*int*) – 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.

SeqInvert

SeqInvert(*seq*) → None
Reverses the order of sequence elements.

Parameters **seq** (CvSeq) – Sequence

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

SeqRemove

SeqRemove(*seq*, *index*) → None
Removes an element from the middle of a sequence.

Parameters

- **seq** (CvSeq) – Sequence
- **index** (*int*) – 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

SeqRemoveSlice(*seq*, *slice*) → None
Removes a sequence slice.

Parameters

- **seq** (CvSeq) – Sequence
- **slice** (CvSlice) – The part of the sequence to remove

The function removes a slice from the sequence.

2.4 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) 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 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

Circle(*img*, *center*, *radius*, *color*, *thickness=1*, *lineType=8*, *shift=0*) → None

Draws a circle.

Parameters

- **img** (CvArr) – Image where the circle is drawn
- **center** (CvPoint) – Center of the circle
- **radius** (*int*) – Radius of the circle
- **color** (CvScalar) – Circle color
- **thickness** (*int*) – Thickness of the circle outline if positive, otherwise this indicates that a filled circle is to be drawn
- **lineType** (*int*) – Type of the circle boundary, see *Line* description
- **shift** (*int*) – 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

ClipLine(*imgSize*, *pt1*, *pt2) -> (clipped_pt1*, *clipped_pt2*)

Clips the line against the image rectangle.

Parameters

- **imgSize** (CvSize) – Size of the image

- **pt1** (CvPoint) – First ending point of the line segment.
- **pt2** (CvPoint) – Second ending point of the line segment.

The function calculates a part of the line segment which is entirely within the image. If the line segment is outside the image, it returns None. If the line segment is inside the image it returns a new pair of points.

DrawContours

DrawContours(*img*, *contour*, *external_color*, *hole_color*, *max_level*, *thickness=1*, *lineType=8*, *offset=(0, 0)*) → None

Draws contour outlines or interiors in an image.

Parameters

- **img** (CvArr) – Image where the contours are to be drawn. As with any other drawing function, the contours are clipped with the ROI.
- **contour** (CvSeq) – Pointer to the first contour
- **external_color** (CvScalar) – Color of the external contours
- **hole_color** (CvScalar) – Color of internal contours (holes)
- **max_level** (*int*) – 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** (*int*) – Thickness of lines the contours are drawn with. If it is negative (For example, =CV _ FILLED), the contour interiors are drawn.
- **lineType** (*int*) – 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 $\texttt{thickness} < 0$.

Ellipse

Ellipse(*img*, *center*, *axes*, *angle*, *start_angle*, *end_angle*, *color*, *thickness=1*, *lineType=8*, *shift=0*) → None

Draws a simple or thick elliptic arc or an fills ellipse sector.

Parameters

- **img** (CvArr) – The image
- **center** (CvPoint) – Center of the ellipse
- **axes** (CvSize) – Length of the ellipse axes
- **angle** (*float*) – Rotation angle
- **start_angle** (*float*) – Starting angle of the elliptic arc
- **end_angle** (*float*) – Ending angle of the elliptic arc.
- **color** (CvScalar) – Ellipse color
- **thickness** (*int*) – Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn
- **lineType** (*int*) – Type of the ellipse boundary, see *Line* description

- **shift** (*int*) – 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

EllipseBox(*img*, *box*, *color*, *thickness=1*, *lineType=8*, *shift=0*) → None

Draws a simple or thick elliptic arc or fills an ellipse sector.

Parameters

- **img** (CvArr) – Image
- **box** (CvBox2D) – The enclosing box of the ellipse drawn
- **thickness** (*int*) – Thickness of the ellipse boundary
- **lineType** (*int*) – Type of the ellipse boundary, see *Line* description
- **shift** (*int*) – 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

FillConvexPoly(*img*, *pn*, *color*, *lineType=8*, *shift=0*) → None

Fills a convex polygon.

Parameters

- **img** (CvArr) – Image
- **pn** (CvPoints) – List of coordinate pairs
- **color** (CvScalar) – Polygon color
- **lineType** (*int*) – Type of the polygon boundaries, see *Line* description
- **shift** (*int*) – 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

FillPoly(*img*, *polys*, *color*, *lineType=8*, *shift=0*) → None

Fills a polygon's interior.

Parameters

- **img** (CvArr) – Image
- **polys** (*list of lists of (x,y) pairs*) – List of lists of (x,y) pairs. Each list of points is a polygon.

- **color** (CvScalar) – Polygon color
- **lineType** (*int*) – Type of the polygon boundaries, see *Line* description
- **shift** (*int*) – 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

GetTextSize(*textString*, *font*)-> (*textSize*, *baseline*)

Retrieves the width and height of a text string.

Parameters

- **font** (CvFont) – Pointer to the font structure
- **textString** (*str*) – Input string
- **textSize** (CvSize) – Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.
- **baseline** (*int*) – 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

InitFont(*fontFace*, *hscale*, *vscale*, *shear=0*, *thickness=1*, *lineType=8*) → font

Initializes font structure.

Parameters

- **font** (CvFont) – Pointer to the font structure initialized by the function
- **fontFace** (*int*) – 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** (*float*) – 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** (*float*) – 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** (*float*) – 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** (*int*) – Thickness of the text strokes
- **lineType** (*int*) – Type of the strokes, see *Line* description

The function initializes the font structure that can be passed to text rendering functions.

InitLineIterator

InitLineIterator(*image*, *pt1*, *pt2*, *connectivity=8*, *left_to_right=0*) → line_iterator

Initializes the line iterator.

Parameters

- **image** (`CvArr`) – Image to sample the line from
- **pt1** (`CvPoint`) – First ending point of the line segment
- **pt2** (`CvPoint`) – Second ending point of the line segment
- **connectivity** (*int*) – The scanned line connectivity, 4 or 8.
- **left_to_right** (*int*) – If (`left_to_right` $= 0$) then the line is scanned in the specified order, from `pt1` to `pt2` . If (`left_to_right` $\neq 0$) the line is scanned from left-most point to right-most.
- **line_iterator** (`iter`) – Iterator over the pixels of the line

The function returns an iterator over the pixels connecting the two points. 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 a color line

or more concisely using zip :

Line

Line(*img*, *pt1*, *pt2*, *color*, *thickness=1*, *lineType=8*, *shift=0*) → None

Draws a line segment connecting two points.

Parameters

- **img** (`CvArr`) – The image
- **pt1** (`CvPoint`) – First point of the line segment
- **pt2** (`CvPoint`) – Second point of the line segment
- **color** (`CvScalar`) – Line color
- **thickness** (*int*) – Line thickness
- **lineType** (*int*) – Type of the line:
  - **8** (or omitted) 8-connected line.

  - **4** 4-connected line.
  - **CV_AA** antialiased line.
- **shift** (*int*) – 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

PolyLine(*img*, *polys*, *is_closed*, *color*, *thickness=1*, *lineType=8*, *shift=0*) → None

Draws simple or thick polygons.

Parameters

- **polys** (*list of lists of (x,y) pairs*) – List of lists of (x,y) pairs. Each list of points is a polygon.
- **img** (`CvArr`) – Image
- **is_closed** (*int*) – 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** (`CvScalar`) – Polyline color
- **thickness** (*int*) – Thickness of the polyline edges
- **lineType** (*int*) – Type of the line segments, see *Line* description
- **shift** (*int*) – Number of fractional bits in the vertex coordinates

The function draws single or multiple polygonal curves.

PutText

PutText(*img*, *text*, *org*, *font*, *color*) → None

Draws a text string.

Parameters

- **img** (`CvArr`) – Input image
- **text** (*str*) – String to print
- **org** (`CvPoint`) – Coordinates of the bottom-left corner of the first letter
- **font** (`CvFont`) – Pointer to the font structure
- **color** (`CvScalar`) – 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

Rectangle(*img*, *pt1*, *pt2*, *color*, *thickness=1*, *lineType=8*, *shift=0*) → None

Draws a simple, thick, or filled rectangle.

Parameters

- **img** (CvArr) – Image
- **pt1** (CvPoint) – One of the rectangle's vertices
- **pt2** (CvPoint) – Opposite rectangle vertex
- **color** (CvScalar) – Line color (RGB) or brightness (grayscale image)
- **thickness** (*int*) – Thickness of lines that make up the rectangle. Negative values, e.g., CV _ FILLED, cause the function to draw a filled rectangle.
- **lineType** (*int*) – Type of the line, see *Line* description
- **shift** (*int*) – Number of fractional bits in the point coordinates

The function draws a rectangle with two opposite corners `pt1` and `pt2` .

CV_RGB

CV_RGB(*red*, *grn*, *blu*) → CvScalar

Constructs a color value.

Parameters

- **red** (*float*) – Red component
- **grn** (*float*) – Green component
- **blu** (*float*) – Blue component

2.5 XML/YAML Persistence

Load

Load(*filename*, *storage=NULL*, *name=NULL*) → generic

Loads an object from a file.

Parameters

- **filename** (*str*) – File name
- **storage** (CvMemStorage) – Memory storage for dynamic structures, such as *CvSeq* or *CvGraph* . It is not used for matrices or images.
- **name** (*str*) – Optional object name. If it is NULL, the first top-level object in the storage will be loaded.

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.

Save

Save(*filename*, *structPtr*, *name=NULL*, *comment=NULL*) → None

Saves an object to a file.

Parameters

- **filename** (*str*) – File name

- **structPtr** (`generic`) – Object to save
- **name** (*str*) – Optional object name. If it is NULL, the name will be formed from `filename` .
- **comment** (*str*) – Optional comment to put in the beginning of the file

The function saves an object to a file. It provides a simple interface to *Write* .

2.6 Clustering

KMeans2

KMeans2(*samples*, *nclusters*, *labels*, *termcrit*) → None

Splits set of vectors by a given number of clusters.

Parameters

- **samples** (`CvArr`) – Floating-point matrix of input samples, one row per sample
- **nclusters** (*int*) – Number of clusters to split the set by
- **labels** (`CvArr`) – Output integer vector storing cluster indices for every sample
- **termcrit** (`CvTermCriteria`) – Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

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.

2.7 Utility and System Functions and Macros

Error Handling

Errors in argument type cause a `TypeError` exception. OpenCV errors cause an `cv.error` exception.

For example a function argument that is the wrong type produces a `TypeError` :

A function with the

GetTickCount

GetTickCount() → long

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

GetTickFrequency() → long

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.

CHAPTER

THREE

IMGPROC. IMAGE PROCESSING

3.1 Histograms

CvHistogram

class CvHistogram

Multi-dimensional histogram.

A CvHistogram is a multi-dimensional histogram, created by function *CreateHist* . It has an attribute `bins` a *CvMatND* containing the histogram counts.

CalcBackProject

CalcBackProject(*image*, *back_project*, *hist*) → None

Calculates the back projection.

Parameters

- **image** (sequence of `IplImage`) – Source images (though you may pass CvMat** as well)
- **back_project** (`CvArr`) – Destination back projection image of the same type as the source images
- **hist** (`CvHistogram`) – 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

CalcBackProjectPatch(*images*, *dst*, *patch_size*, *hist*, *method*, *factor*) → None

Locates a template within an image by using a histogram comparison.

Parameters

- **images** (sequence of IplImage) – Source images (though, you may pass CvMat** as well)
- **dst** (CvArr) – Destination image
- **patch_size** (CvSize) – Size of the patch slid though the source image
- **hist** (CvHistogram) – Histogram
- **method** (*int*) – Comparison method, passed to *CompareHist* (see description of that function)
- **factor** (*float*) – 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

CalcHist(*image*, *hist*, *accumulate=0*, *mask=NULL*) → None

Calculates the histogram of image(s).

Parameters

- **image** (sequence of IplImage) – Source images (though you may pass CvMat** as well)
- **hist** (CvHistogram) – Pointer to the histogram
- **accumulate** (*int*) – 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** (CvArr) – 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.

```
# Calculating and displaying 2D Hue-Saturation histogram of a color image

import sys
import cv

def hs_histogram(src):
    # Convert to HSV
    hsv = cv.CreateImage(cv.GetSize(src), 8, 3)
    cv.CvtColor(src, hsv, cv.CV_BGR2HSV)

    # Extract the H and S planes
    h_plane = cv.CreateMat(src.rows, src.cols, cv.CV_8UC1)
    s_plane = cv.CreateMat(src.rows, src.cols, cv.CV_8UC1)
    cv.Split(hsv, h_plane, s_plane, None, None)
    planes = [h_plane, s_plane]

    h_bins = 30
    s_bins = 32
    hist_size = [h_bins, s_bins]
    # hue varies from 0 (~0 deg red) to 180 (~360 deg red again */
    h_ranges = [0, 180]
    # saturation varies from 0 (black-gray-white) to
    # 255 (pure spectrum color)
    s_ranges = [0, 255]
    ranges = [h_ranges, s_ranges]
    scale = 10
    hist = cv.CreateHist([h_bins, s_bins], cv.CV_HIST_ARRAY, ranges, 1)
    cv.CalcHist([cv.GetImage(i) for i in planes], hist)
    (_, max_value, _, _) = cv.GetMinMaxHistValue(hist)

    hist_img = cv.CreateImage((h_bins*scale, s_bins*scale), 8, 3)

    for h in range(h_bins):
        for s in range(s_bins):
            bin_val = cv.QueryHistValue_2D(hist, h, s)
            intensity = cv.Round(bin_val * 255 / max_value)
            cv.Rectangle(hist_img,
                         (h*scale, s*scale),
                         ((h+1)*scale - 1, (s+1)*scale - 1),
                         cv.RGB(intensity, intensity, intensity),
                         cv.CV_FILLED)
    return hist_img

if __name__ == '__main__':
    src = cv.LoadImageM(sys.argv[1])
    cv.NamedWindow("Source", 1)
    cv.ShowImage("Source", src)

    cv.NamedWindow("H-S Histogram", 1)
    cv.ShowImage("H-S Histogram", hs_histogram(src))

    cv.WaitKey(0)
```

CalcProbDensity

CalcProbDensity(*hist1*, *hist2*, *dst_hist*, *scale=255*) → None

Divides one histogram by another.

Parameters

- **hist1** (`CvHistogram`) – first histogram (the divisor)
- **hist2** (`CvHistogram`) – second histogram
- **dst_hist** (`CvHistogram`) – destination histogram
- **scale** (*float*) – scale factor for the destination histogram

The function calculates the object probability density from the two histograms as:

$$\texttt{dist\_hist}(I) = \begin{cases} 0 & \text{if } \texttt{hist1}(I) = 0 \\ \texttt{scale} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) > \texttt{hist1}(I) \\ \frac{\texttt{hist2}(I) \cdot \texttt{scale}}{\texttt{hist1}(I)} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) \leq \texttt{hist1}(I) \end{cases}$$

So the destination histogram bins are within less than `scale` .

ClearHist

ClearHist(*hist*) → None

Clears the histogram.

Parameters **hist** (`CvHistogram`) – 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

CompareHist(*hist1*, *hist2*, *method*) → float

Compares two dense histograms.

Parameters

- **hist1** (`CvHistogram`) – The first dense histogram
- **hist2** (`CvHistogram`) – The second dense histogram
- **method** (*int*) – 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.

CreateHist

CreateHist(*dims*, *type*, *ranges*, *uniform = 1*) → hist

Creates a histogram.

Parameters

- **dims** (*sequence of int*) – for an N-dimensional histogram, list of length N giving the size of each dimension
- **type** (*int*) – 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** (*list of tuples of ints*) – 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** (*int*) – 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.

GetMinMaxHistValue

GetMinMaxHistValue(*hist*)-> *(min_value*, *max_value*, *min_idx*, *max_idx*)

Finds the minimum and maximum histogram bins.

Parameters

- **hist** (`CvHistogram`) – Histogram
- **min_value** (`CvScalar`) – Minimum value of the histogram
- **max_value** (`CvScalar`) – Maximum value of the histogram
- **min_idx** (*sequence of int*) – Coordinates of the minimum
- **max_idx** (*sequence of int*) – Coordinates of 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.

NormalizeHist

NormalizeHist(*hist*, *factor*) → None

Normalizes the histogram.

Parameters

- **hist** (`CvHistogram`) – Pointer to the histogram
- **factor** (*float*) – Normalization factor

The function normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor` .

QueryHistValue_1D

QueryHistValue_1D(*hist*, *idx0*) → float

Returns the value from a 1D histogram bin.

Parameters

- **hist** (`CvHistogram`) – Histogram
- **idx0** (*int*) – bin index 0

QueryHistValue_2D

QueryHistValue_2D(*hist*, *idx0*, *idx1*) → float

Returns the value from a 2D histogram bin.

Parameters

- **hist** (`CvHistogram`) – Histogram
- **idx0** (*int*) – bin index 0
- **idx1** (*int*) – bin index 1

QueryHistValue_3D

QueryHistValue_3D(*hist*, *idx0*, *idx1*, *idx2*) → float

Returns the value from a 3D histogram bin.

Parameters

- **hist** (`CvHistogram`) – Histogram
- **idx0** (*int*) – bin index 0
- **idx1** (*int*) – bin index 1
- **idx2** (*int*) – bin index 2

QueryHistValue_nD

QueryHistValue_nD(*hist*, *idx*) → float

Returns the value from a 1D histogram bin.

Parameters

- **hist** (`CvHistogram`) – Histogram
- **idx** (*sequence of int*) – list of indices, of same length as the dimension of the histogram's bin.

ThreshHist

ThreshHist(*hist*, *threshold*) → None

Thresholds the histogram.

Parameters

- **hist** (`CvHistogram`) – Pointer to the histogram
- **threshold** (*float*) – Threshold level

The function clears histogram bins that are below the specified threshold.

3.2 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

class `IplConvKernel`

An IplConvKernel is a rectangular convolution kernel, created by function *CreateStructuringElementEx* .

CopyMakeBorder

`CopyMakeBorder`(*src*, *dst*, *offset*, *bordertype*, *value=(0, 0, 0, 0)*) → None

Copies an image and makes a border around it.

Parameters

- **src** (`CvArr`) – The source image
- **dst** (`CvArr`) – The destination image
- **offset** (`CvPoint`) – 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** (*int*) – 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** (`CvScalar`) – 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

`CreateStructuringElementEx`(*cols*, *rows*, *anchorX*, *anchorY*, *shape*, *values=None*) → kernel

Creates a structuring element.

Parameters

- **cols** (*int*) – Number of columns in the structuring element

- **rows** (*int*) – Number of rows in the structuring element
- **anchorX** (*int*) – Relative horizontal offset of the anchor point
- **anchorY** (*int*) – Relative vertical offset of the anchor point
- **shape** (*int*) – 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** (*sequence of int*) – 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

Dilate(*src*, *dst*, *element=None*, *iterations=1*) → None

Dilates an image by using a specific structuring element.

Parameters

- **src** (`CvArr`) – Source image
- **dst** (`CvArr`) – Destination image
- **element** (`IplConvKernel`) – Structuring element used for dilation. If it is `None` , a `3 x 3` rectangular structuring element is used
- **iterations** (*int*) – 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

Erode(*src*, *dst*, *element=None*, *iterations=1*) → None

Erodes an image by using a specific structuring element.

Parameters

- **src** (`CvArr`) – Source image
- **dst** (`CvArr`) – Destination image

- **element** (IplConvKernel) – Structuring element used for erosion. If it is None , a 3 x 3 rectangular structuring element is used
- **iterations** (*int*) – 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

Filter2D(*src*, *dst*, *kernel*, *anchor=(-1, -1)*) → None

Convolves an image with the kernel.

Parameters

- **src** (CvArr) – The source image
- **dst** (CvArr) – The destination image
- **kernel** (CvMat) – 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** (CvPoint) – 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

Laplace(*src*, *dst*, *apertureSize=3*) → None

Calculates the Laplacian of an image.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Destination image
- **apertureSize** (*int*) – 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

MorphologyEx(*src*, *dst*, *temp*, *element*, *operation*, *iterations=1*) → None

Performs advanced morphological transformations.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Destination image
- **temp** (CvArr) – Temporary image, required in some cases
- **element** (IplConvKernel) – Structuring element
- **operation** (*int*) – 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** (*int*) – 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

PyrDown(*src*, *dst*, *filter=CV_GAUSSIAN_5X5*) → None

Downsamples an image.

Parameters

- **src** (CvArr) – The source image

- **dst** (CvArr) – The destination image, should have a half as large width and height than the source
- **filter** (*int*) – 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.

Smooth

Smooth(*src*, *dst*, *smoothtype=CV_GAUSSIAN*, *param1=3*, *param2=0*, *param3=0*, *param4=0*) → None

Smooths the image in one of several ways.

Parameters

- **src** (CvArr) – The source image
- **dst** (CvArr) – The destination image
- **smoothtype** (*int*) – Type of the smoothing:
  - **CV_BLUR_NO_SCALE** linear convolution with param1 × 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 param1 × param2 box kernel (all 1's) with subsequent scaling by $1/(\texttt{param1} \cdot \texttt{param2})$
  - **CV_GAUSSIAN** linear convolution with a param1 × param2 Gaussian kernel
  - **CV_MEDIAN** median filter with a param1 × param1 square aperture
  - **CV_BILATERAL** bilateral filter with a param1 × 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** (*int*) – The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)
- **param2** (*int*) – 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** (*float*) – 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

Sobel(*src*, *dst*, *xorder*, *yorder*, *apertureSize = 3*) → None

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator.

Parameters

- **src** (CvArr) – Source image of type CvArr*
- **dst** (CvArr) – Destination image
- **xorder** (*int*) – Order of the derivative x
- **yorder** (*int*) – Order of the derivative y
- **apertureSize** (*int*) – Size of the extended Sobel kernel, must be 1, 3, 5 or 7

In all cases except 1, an `apertureSize`×`apertureSize` separable kernel will be used to calculate the derivative. For `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.

3.3 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

GetRotationMatrix2D (*center*, *angle*, *scale*, *mapMatrix*) → None

Calculates the affine matrix of 2d rotation.

Parameters

- **center** (`CvPoint2D32f`) – Center of the rotation in the source image
- **angle** (*float*) – The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)
- **scale** (*float*) – Isotropic scale factor
- **mapMatrix** (`CvMat`) – 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

GetAffineTransform(*src*, *dst*, *mapMatrix*) → None

Calculates the affine transform from 3 corresponding points.

Parameters

- **src** (CvPoint2D32f) – Coordinates of 3 triangle vertices in the source image
- **dst** (CvPoint2D32f) – Coordinates of the 3 corresponding triangle vertices in the destination image
- **mapMatrix** (CvMat) – 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

GetPerspectiveTransform(*src*, *dst*, *mapMatrix*) → None

Calculates the perspective transform from 4 corresponding points.

Parameters

- **src** (CvPoint2D32f) – Coordinates of 4 quadrangle vertices in the source image
- **dst** (CvPoint2D32f) – Coordinates of the 4 corresponding quadrangle vertices in the destination image
- **mapMatrix** (CvMat) – 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

GetQuadrangleSubPix(*src*, *dst*, *mapMatrix*) → None

Retrieves the pixel quadrangle from an image with sub-pixel accuracy.

Parameters

- **src** (CvArr) – Source image

- **dst** (CvArr) – Extracted quadrangle
- **mapMatrix** (CvMat) – 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

GetRectSubPix(*src*, *dst*, *center*) → None

Retrieves the pixel rectangle from an image with sub-pixel accuracy.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Extracted rectangle
- **center** (CvPoint2D32f) – 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

LogPolar(*src*, *dst*, *center*, *M*, *flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS*) → None

Remaps an image to log-polar space.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Destination image
- **center** (CvPoint2D32f) – The transformation center; where the output precision is maximal
- **M** (*float*) – Magnitude scale parameter. See below

- **flags** (*int*) – 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.

Remap

Remap(*src*, *dst*, *mapx*, *mapy*, *flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS*, *fillval=(0, 0, 0, 0)*) → None

Applies a generic geometrical transformation to the image.

Parameters

- **src** (`CvArr`) – Source image
- **dst** (`CvArr`) – Destination image
- **mapx** (`CvArr`) – The map of x-coordinates (CV _ 32FC1 image)
- **mapy** (`CvArr`) – The map of y-coordinates (CV _ 32FC1 image)
- **flags** (*int*) – 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** (`CvScalar`) – 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

Resize(*src*, *dst*, *interpolation=CV_INTER_LINEAR*) → None

Resizes an image.

Parameters

- **src** (`CvArr`) – Source image

- **dst** (CvArr) – Destination image
- **interpolation** (*int*) – 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

WarpAffine(*src*, *dst*, *mapMatrix*, *flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS*, *fillval=(0, 0, 0, 0)*) → None

Applies an affine transformation to an image.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Destination image
- **mapMatrix** (CvMat) – 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

  type flags int
- **fillval** (CvScalar) – 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

WarpPerspective(*src*, *dst*, *mapMatrix*, *flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS*, *fillval=(0, 0, 0, 0)*) → None

Applies a perspective transformation to an image.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Destination image
- **mapMatrix** (CvMat) – 3×3 transformation matrix
- **flags** (*int*) – 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** (CvScalar) – 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.

3.4 Miscellaneous Image Transformations

AdaptiveThreshold

AdaptiveThreshold(*src*, *dst*, *maxValue*, *adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C*, *thresholdType=CV_THRESH_BINARY*, *blockSize=3*, *param1=5*) → None

Applies an adaptive threshold to an array.

Parameters

- **src** (CvArr) – Source image
- **dst** (CvArr) – Destination image
- **maxValue** (*float*) – Maximum value that is used with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV`
- **adaptive_method** (*int*) – Adaptive thresholding algorithm to use: `CV_ADAPTIVE_THRESH_MEAN_C` or `CV_ADAPTIVE_THRESH_GAUSSIAN_C` (see the discussion)

- **thresholdType** (*int*) – Thresholding type; must be one of
  - **CV_THRESH_BINARY** xxx
  - **CV_THRESH_BINARY_INV** xxx
- **blockSize** (*int*) – The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on
- **param1** (*float*) – 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 `blockSize` × `blockSize` pixel neighborhood, minus `param1` .

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is the weighted sum (gaussian) of a `blockSize` × `blockSize` pixel neighborhood, minus `param1` .

CvtColor

CvtColor(*src*, *dst*, *code*) → None

Converts an image from one color space to another.

Parameters

- **src** (`CvArr`) – The source 8-bit (8u), 16-bit (16u) or single-precision floating-point (32f) image
- **dst** (`CvArr`) – The destination image of the same data type as the source. The number of channels may be different
- **code** (*int*) – 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 ↔ 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

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \le 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

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 $0 \le L \le 100$, $-127 \le a \le 127$, $-127 \le b \le 127$ 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 ↔ 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

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y <= 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

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 → 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:

$$\begin{matrix} R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

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

`DistTransform`(*src*, *dst*, *distance_type=CV_DIST_L2*, *mask_size=3*, *mask=None*, *labels=NULL*) → None

Calculates the distance to the closest zero pixel for all non-zero pixels of the source image.

Parameters

- **src** (`CvArr`) – 8-bit, single-channel (binary) source image
- **dst** (`CvArr`) – Output image with calculated distances (32-bit floating-point, single-channel)

- **distance_type** (*int*) – Type of distance; can be `CV_DIST_L1`, `CV_DIST_L2`, `CV_DIST_C` or `CV_DIST_USER`
- **mask_size** (*int*) – 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** (*sequence of float*) – 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** (`CvArr`) – 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

class CvConnectedComp

Connected component, represented as a tuple (area, value, rect), where area is the area of the component as a float, value is the average color as a *CvScalar* , and rect is the ROI of the component, as a *CvRect* .

FloodFill

FloodFill(*image*, *seed_point*, *new_val*, *lo_diff=(0, 0, 0, 0)*, *up_diff=(0, 0, 0, 0)*, *flags=4*, *mask=NULL*) → comp

Fills a connected component with the given color.

Parameters

- **image** (CvArr) – 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** (CvPoint) – The starting point
- **new_val** (CvScalar) – New value of the repainted domain pixels
- **lo_diff** (CvScalar) – 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** (CvScalar) – 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** (CvConnectedComp) – Returned connected component for 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** (*int*) – 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** (CvArr) – 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

Inpaint(*src*, *mask*, *dst*, *inpaintRadius*, *flags*) → None

Inpaints the selected region in the image.

Parameters

- **src** (CvArr) – The input 8-bit 1-channel or 3-channel image.
- **mask** (CvArr) – The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.
- **dst** (CvArr) – The output image of the same format and the same size as input.
- **inpaintRadius** (*float*) – The radius of circlular neighborhood of each point inpainted that is considered by the algorithm.

- **flags** (*int*) – 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

Integral(*image*, *sum*, *sqsum=NULL*, *tiltedSum=NULL*) → None

Calculates the integral of an image.

Parameters

- **image** (CvArr) – The source image, $W \times H$, 8-bit or floating-point (32f or 64f)
- **sum** (CvArr) – The integral image, $(W+1) \times (H+1)$, 32-bit integer or double precision floating-point (64f)
- **sqsum** (CvArr) – The integral image for squared pixel values, $(W+1) \times (H+1)$, double precision floating-point (64f)
- **tiltedSum** (CvArr) – 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:

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

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

PyrMeanShiftFiltering(*src*, *dst*, *sp*, *sr*, *max_level=1*, *termcrit=(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 5, 1)*) → None

Does meanshift image segmentation

Parameters

- **src** (CvArr) – The source 8-bit, 3-channel image.
- **dst** (CvArr) – The destination image of the same format and the same size as the source.

- **sp** (*float*) – The spatial window radius.
- **sr** (*float*) – The color window radius.
- **max_level** (*int*) – Maximum level of the pyramid for the segmentation.
- **termcrit** (CvTermCriteria) – 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:

$$(x, y) : X - \texttt{sp} \leq x \leq X + \texttt{sp}, Y - \texttt{sp} \leq y \leq Y + \texttt{sp}, ||(R, G, B) - (r, g, b)|| \leq \texttt{sr}$$

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 `max_level` > 0, the gaussian pyramid of `max_level` $+ 1$ levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much ($>$ `sr`) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when `max_level` $== 0$).

PyrSegmentation

PyrSegmentation(*src*, *dst*, *storage*, *level*, *threshold1*, *threshold2*) $\rightarrow$ comp

Implements image segmentation by pyramids.

Parameters

- **src** (IplImage) – The source image
- **dst** (IplImage) – The destination image
- **storage** (CvMemStorage) – Storage; stores the resulting sequence of connected components
- **comp** (CvSeq) – Pointer to the output sequence of the segmented components
- **level** (*int*) – Maximum level of the pyramid for the segmentation
- **threshold1** (*float*) – Error threshold for establishing the links
- **threshold2** (*float*) – 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

$$p(c^1, c^2) = 0.30(c^1_r - c^2_r) + 0.59(c^1_g - c^2_g) + 0.11(c^1_b - c^2_b).$$

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

Threshold(*src*, *dst*, *threshold*, *maxValue*, *thresholdType*) → None

Applies a fixed-level threshold to array elements.

Parameters

- **src** (CvArr) – Source array (single-channel, 8-bit or 32-bit floating point)
- **dst** (CvArr) – Destination array; must be either the same type as src or 8-bit
- **threshold** (*float*) – Threshold value
- **maxValue** (*float*) – Maximum value to use with CV_THRESH_BINARY and CV_THRESH_BINARY_INV thresholding types
- **thresholdType** (*int*) – 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.

3.5 Structural Analysis and Shape Descriptors

ApproxChains

ApproxChains(*src_seq*, *storage*, *method=CV_CHAIN_APPROX_SIMPLE*, *parameter=0*, *minimal_perimeter=0*, *recursive=0*) → chains

Approximates Freeman chain(s) with a polygonal curve.

Parameters

- **src_seq** (CvSeq) – Pointer to the chain that can refer to other chains
- **storage** (CvMemStorage) – Storage location for the resulting polylines
- **method** (*int*) – Approximation method (see the description of the function *FindContours*)
- **parameter** (*float*) – Method parameter (not used now)
- **minimal_perimeter** (*int*) – Approximates only those contours whose perimeters are not less than `minimal_perimeter` . Other chains are removed from the resulting structure
- **recursive** (*int*) – 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

ApproxPoly(src_seq, storage, method, parameter=0, parameter2=0) -> sequence

Approximates polygonal curve(s) with the specified precision.

param src_seq Sequence of an array of points

type src_seq CvArr or CvSeq

param storage Container for the approximated contours. If it is NULL, the input sequences' storage is used

type storage CvMemStorage

param method Approximation method; only `CV_POLY_APPROX_DP` is supported, that corresponds to the Douglas-Peucker algorithm

type method int

param parameter Method-specific parameter; in the case of `CV_POLY_APPROX_DP` it is a desired approximation accuracy

type parameter float

param 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)

type parameter2 int

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

ArcLength(*curve*, *slice=CV_WHOLE_SEQ*, *isClosed=-1*) → double

Calculates the contour perimeter or the curve length.

Parameters

- **curve** (CvArr or CvSeq) – Sequence or array of the curve points
- **slice** (CvSlice) – Starting and ending points of the curve, by default, the whole curve length is calculated
- **isClosed** (*int*) – 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

BoundingRect(*points*, *update=0*) → CvRect

Calculates the up-right bounding rectangle of a point set.

Parameters

- **points** (CvArr or CvSeq) – 2D point set, either a sequence or vector (CvMat) of points
- **update** (*int*) – 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

BoxPoints(*box*) → points

Finds the box vertices.

Parameters

- **box** (CvBox2D) – Box
- **points** (CvPoint2D32f_4) – Array of vertices

The function calculates the vertices of the input 2d box.

CalcPGH

CalcPGH(*contour*, *hist*) → None

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

CalcEMD2(*signature1*, *signature2*, *distance_type*, *distance_func = None*, *cost_matrix=None*, *flow=None*, *lower_bound=None*, *userdata = None*) → float

Computes the "minimal work" distance between two weighted point configurations.

Parameters

- **signature1** (CvArr) – First signature, a size1 × dims + 1 floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used
- **signature2** (CvArr) – 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** (*int*) – 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** (PyCallableObject) – The user-supplied distance function. It takes coordinates of two points pt0 and pt1 , and returns the distance between the points, with sigature '' func(pt0, pt1, userdata) -> float''
- **cost_matrix** (CvArr) – 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** (CvArr) – 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** (*float*) – 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** (*object*) – 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

CheckContourConvexity(*contour*) → int

Tests contour convexity.

Parameters **contour** (`CvArr` or `CvSeq`) – 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

class CvConvexityDefect

A single contour convexity defect, represented by a tuple `(start, end, depthpoint, depth)`.

start
(x, y) point of the contour where the defect begins

end
(x, y) point of the contour where the defect ends

depthpoint
(x, y) point farthest from the convex hull point within the defect

depth
distance between the farthest point and the convex hull

ContourArea

ContourArea(*contour*, *slice=CV_WHOLE_SEQ*) → double

Calculates the area of a whole contour or a contour section.

Parameters

- **contour** (`CvArr` or `CvSeq`) – Contour (sequence or array of vertices)
- **slice** (`CvSlice`) – 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

ContourFromContourTree(*tree*, *storage*, *criteria*) → contour

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

ConvexHull2(*points*, *storage*, *orientation=CV_CLOCKWISE*, *return_points=0*) → convex_hull

Finds the convex hull of a point set.

Parameters

- **points** (`CvArr` or `CvSeq`) – Sequence or array of 2D points with 32-bit integer or floating-point coordinates
- **storage** (`CvMemStorage`) – 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** (*int*) – Desired orientation of convex hull: `CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`
- **return_points** (*int*) – 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.

ConvexityDefects

ConvexityDefects(*contour*, *convexhull*, *storage*) → convexity_defects

Finds the convexity defects of a contour.

Parameters

- **contour** (`CvArr` or `CvSeq`) – Input contour

- **convexhull** (CvSeq) – 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** (CvMemStorage) – 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

CreateContourTree(*contour*, *storage*, *threshold*) → contour_tree

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.

FindContours

FindContours(*image*, *storage*, *mode=CV_RETR_LIST*, *method=CV_CHAIN_APPROX_SIMPLE*, *offset=(0, 0)*) → cvseq

Finds the contours in a binary image.

Parameters

- **image** (CvArr) – 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** (CvMemStorage) – Container of the retrieved contours
- **mode** (*int*) – 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** (*int*) – 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** (`CvPoint`) – 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.py` in the OpenCV sample directory.

Note: the source `image` is modified by this function.

FitEllipse2

FitEllipse2(*points*) → Box2D

Fits an ellipse around a set of 2D points.

Parameters **points** (`CvArr`) – 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

FitLine(*points*, *dist_type*, *param*, *reps*, *aeps*) → line

Fits a line to a 2D or 3D point set.

Parameters

- **points** (`CvArr`) – Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates
- **dist_type** (*int*) – The distance used for fitting (see the discussion)
- **param** (*float*) – Numerical parameter (`C`) for some types of distances, if 0 then some optimal value is chosen
- **reps** (*float*) – Sufficient accuracy for the radius (distance between the coordinate origin and the line). 0.01 is a good default value.

- **aeps** (*float*) – Sufficient accuracy for the angle. 0.01 is a good default value.
- **line** (*object*) – The output line parameters. In the case of a 2d fitting, it is a tuple 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 a tuple 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

$$\rho\left(r\right) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right) \quad \text{where} \quad C = 1.3998$$

- dist_type=CV_DIST_WELSCH

$$\rho\left(r\right) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

- 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

GetCentralMoment(*moments*, *x_order*, *y_order*) → double

Retrieves the central moment from the moment state structure.

Parameters

- **moments** (`CvMoments`) – Pointer to the moment state structure

- **x_order** (*int*) – x order of the retrieved moment, `x_order` $>= 0$
- **y_order** (*int*) – 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

GetHuMoments(*moments*) → hu

Calculates the seven Hu invariants.

Parameters

- **moments** (`CvMoments`) – The input moments, computed with *Moments*
- **hu** (*object*) – The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment , that are defined as:

$$\begin{aligned}
hu_1 &= \eta_{20} + \eta_{02} \\
hu_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
hu_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
hu_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
hu_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
hu_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
hu_7 &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}$$

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

GetNormalizedCentralMoment(*moments*, *x_order*, *y_order*) → double

Retrieves the normalized central moment from the moment state structure.

Parameters

- **moments** (`CvMoments`) – Pointer to the moment state structure
- **x_order** (*int*) – x order of the retrieved moment, `x_order` $>= 0$
- **y_order** (*int*) – y order of the retrieved moment, `y_order` $>= 0$ and `x_order` + `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

GetSpatialMoment(*moments*, *x_order*, *y_order*) → double

Retrieves the spatial moment from the moment state structure.

Parameters

- **moments** (CvMoments) – The moment state, calculated by *Moments*
- **x_order** (*int*) – x order of the retrieved moment, x_order >= 0
- **y_order** (*int*) – y order of the retrieved moment, y_order >= 0 and x_order + 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

MatchContourTrees(*tree1*, *tree2*, *method*, *threshold*) → double

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

MatchShapes(*object1*, *object2*, *method*, *parameter=0*) → None

Compares two shapes.

Parameters

- **object1** (CvSeq) – First contour or grayscale image
- **object2** (CvSeq) – Second contour or grayscale image
- **method** (*int*) –

  Comparison method; CV_CONTOUR_MATCH_I1 , CV_CONTOURS_MATCH_I2

  or CV_CONTOURS_MATCH_I3
- **parameter** (*float*) – 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_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1\ldots7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

- method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1\ldots7} \left| m_i^A - m_i^B \right|$$

- method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1\ldots7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

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

MinAreaRect2(*points*, *storage=NULL*) → CvBox2D

Finds the circumscribed rectangle of minimal area for a given 2D point set.

Parameters

- **points** (CvArr or CvSeq) – Sequence or array of points
- **storage** (CvMemStorage) – 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

MinEnclosingCircle(*points*)-> (*int*, *center*, *radius*)

Finds the circumscribed circle of minimal area for a given 2D point set.

Parameters

- **points** (CvArr or CvSeq) – Sequence or array of 2D points
- **center** (CvPoint2D32f) – Output parameter; the center of the enclosing circle
- **radius** (*float*) – 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

Moments(*arr*, *binary = 0*) → moments

Calculates all of the moments up to the third order of a polygon or rasterized shape.

Parameters

- **arr** (CvArr or CvSeq) – Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points or a vector of points)
- **moments** (CvMoments) – Pointer to returned moment's state structure
- **binary** (*int*) – (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 characeteristics including 7 Hu invariants.

PointPolygonTest

PointPolygonTest(*contour*, *pt*, *measure_dist*) → double

Point in contour test.

Parameters

- **contour** (CvArr or CvSeq) – Input contour
- **pt** (CvPoint2D32f) – The point tested against the contour
- **measure_dist** (*int*) – 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 `measure_dist` $= 0$, the return value is +1, -1 and 0, respectively. When `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.

3.6 Planar Subdivisions

CvSubdiv2D

class CvSubdiv2D

Planar subdivision.

edges
A *CvSet* of *CvSubdiv2DEdge*

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.

CvSubdiv2DPoint

class CvSubdiv2DPoint

Point of original or dual subdivision.

first
A connected *CvSubdiv2DEdge*

pt
Position, as a *CvPoint2D32f*

CalcSubdivVoronoi2D

CalcSubdivVoronoi2D(*subdiv*) → None
Calculates the coordinates of Voronoi diagram cells.

Parameters **subdiv** (CvSubdiv2D) – 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

ClearSubdivVoronoi2D(*subdiv*) → None
Removes all virtual points.

Parameters **subdiv** (CvSubdiv2D) – 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

CreateSubdivDelaunay2D(*rect*, *storage*) → delaunay_triangulation
Creates an empty Delaunay triangulation.

Parameters

- **rect** (CvRect) – Rectangle that includes all of the 2d points that are to be added to the subdivision
- **storage** (CvMemStorage) – 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

FindNearestPoint2D(*subdiv*, *pt*) → point

Finds the closest subdivision vertex to the given point.

Parameters

- **subdiv** (`CvSubdiv2D`) – Delaunay or another subdivision
- **pt** (`CvPoint2D32f`) – 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

Subdiv2DEdgeDst(*edge*) → point

Returns the edge destination.

Parameters **edge** (`CvSubdiv2DEdge`) – 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

Subdiv2DGetEdge(*edge*, *type*) → CvSubdiv2DEdge

Returns one of the edges related to the given edge.

Parameters

- **edge** (`CvSubdiv2DEdge`) – Subdivision edge (not a quad-edge)
- **type** – Specifies which of the related edges to return, one of the following:

The function returns one of the edges related to the input edge.

Subdiv2DNextEdge

Subdiv2DNextEdge(*edge*) → CvSubdiv2DEdge

Returns next edge around the edge origin

Parameters **edge** (`CvSubdiv2DEdge`) – 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

Subdiv2DLocate(*subdiv*, *pt*) -> (*loc*, *where*)
Returns the location of a point within a Delaunay triangulation.

Parameters

- **subdiv** (CvSubdiv2D) – Delaunay or another subdivision
- **pt** (CvPoint2D32f) – The point to locate
- **loc** (*int*) – The location of the point within the triangulation
- **where** (CvSubdiv2DEdge, CvSubdiv2DPoint) – The edge or vertex. See below.

The function locates the input point within the subdivision. There are 5 cases:

- The point falls into some facet. loc is CV_PTLOC_INSIDE and where is one of edges of the facet.
- The point falls onto the edge. loc is CV_PTLOC_ON_EDGE and where is the edge.
- The point coincides with one of the subdivision vertices. loc is CV_PTLOC_VERTEX and where is the vertex.
- The point is outside the subdivsion reference rectangle. loc is CV_PTLOC_OUTSIDE_RECT and where is None.
- One of input arguments is invalid. The function raises an exception.

Subdiv2DRotateEdge

Subdiv2DRotateEdge(*edge*, *rotate*) → CvSubdiv2DEdge
Returns another edge of the same quad-edge.

Parameters

- **edge** (CvSubdiv2DEdge) – Subdivision edge (not a quad-edge)
- **rotate** (*int*) – 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

SubdivDelaunay2DInsert(*subdiv*, *pt*) → point
Inserts a single point into a Delaunay triangulation.

Parameters

- **subdiv** (CvSubdiv2D) – Delaunay subdivision created by the function *CreateSubdivDelaunay2D*
- **pt** (CvPoint2D32f) – 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.

3.7 Motion Analysis and Object Tracking

Acc

Acc(*image*, *sum*, *mask=NULL*) → None

Adds a frame to an accumulator.

Parameters

- **image** (CvArr) – Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently)
- **sum** (CvArr) – Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point
- **mask** (CvArr) – 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

MultiplyAcc(*image1*, *image2*, *acc*, *mask=NULL*) → None

Adds the product of two input images to the accumulator.

Parameters

- **image1** (CvArr) – First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)
- **image2** (CvArr) – Second input image, the same format as the first one
- **acc** (CvArr) – Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point
- **mask** (CvArr) – 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

RunningAvg(*image*, *acc*, *alpha*, *mask=NULL*) → None

Updates the running average.

Parameters

- **image** (CvArr) – Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)
- **acc** (CvArr) – Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point
- **alpha** (*float*) – Weight of input image
- **mask** (CvArr) – 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

SquareAcc(*image*, *sqsum*, *mask=NULL*) → None

Adds the square of the source image to the accumulator.

Parameters

- **image** (CvArr) – Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)
- **sqsum** (CvArr) – Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point
- **mask** (CvArr) – Optional operation mask

The function adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum`:

$$\texttt{sqsum}(x,y) \leftarrow \texttt{sqsum}(x,y) + \texttt{image}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

3.8 Feature Detection

Canny

Canny(*image*, *edges*, *threshold1*, *threshold2*, *aperture_size=3*) → None

Implements the Canny algorithm for edge detection.

Parameters

- **image** (CvArr) – Single-channel input image
- **edges** (CvArr) – Single-channel image to store the edges found by the function
- **threshold1** (*float*) – The first threshold
- **threshold2** (*float*) – The second threshold
- **aperture_size** (*int*) – 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

CornerEigenValsAndVecs(*image*, *eigenvv*, *blockSize*, *aperture_size=3*) → None

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

Parameters

- **image** (CvArr) – Input image
- **eigenvv** (CvArr) – Image to store the results. It must be 6 times wider than the input image
- **blockSize** (*int*) – Neighborhood size (see discussion)
- **aperture_size** (*int*) – Aperture parameter for the Sobel operator (see *Sobel*)

For every pixel, the function `cvCornerEigenValsAndVecs` considers a `blockSize` × `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

CornerHarris(*image*, *harris_dst*, *blockSize*, *aperture_size=3*, *k=0.04*) → None

Harris edge detector.

Parameters

- **image** (CvArr) – Input image
- **harris_dst** (CvArr) – Image to store the Harris detector responses. Should have the same size as `image`
- **blockSize** (*int*) – Neighborhood size (see the discussion of *CornerEigenValsAndVecs*)
- **aperture_size** (*int*) – Aperture parameter for the Sobel operator (see *Sobel*).
- **k** (*float*) – 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` × `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

CornerMinEigenVal(*image*, *eigenval*, *blockSize*, *aperture_size=3*) → None

Calculates the minimal eigenvalue of gradient matrices for corner detection.

Parameters

- **image** (CvArr) – Input image
- **eigenval** (CvArr) – Image to store the minimal eigenvalues. Should have the same size as `image`
- **blockSize** (*int*) – Neighborhood size (see the discussion of *CornerEigenValsAndVecs*)
- **aperture_size** (*int*) – 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

FindCornerSubPix(*image*, *corners*, *win*, *zero_zone*, *criteria*) → corners

Refines the corner locations.

Parameters

- **image** (CvArr) – Input image
- **corners** (*sequence of (float, float)*) – Initial coordinates of the input corners as a list of (x, y) pairs
- **win** (CvSize) – 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** (CvSize) – 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** (CvTermCriteria) – 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. It returns the refined coordinates as a list of (x, y) pairs.

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

GoodFeaturesToTrack(*image*, *eigImage*, *tempImage*, *cornerCount*, *qualityLevel*, *minDistance*, *mask=NULL*, *blockSize=3*, *useHarris=0*, *k=0.04*) → corners

Determines strong corners on an image.

Parameters

- **image** (CvArr) – The source 8-bit or floating-point 32-bit, single-channel image
- **eigImage** (CvArr) – Temporary floating-point 32-bit image, the same size as `image`
- **tempImage** (CvArr) – Another temporary image, the same size and format as `eigImage`
- **cornerCount** (*int*) – number of corners to detect
- **qualityLevel** (*float*) – Multiplier for the max/min eigenvalue; specifies the minimal accepted quality of image corners
- **minDistance** (*float*) – Limit, specifying the minimum possible distance between the returned corners; Euclidian distance is used
- **mask** (CvArr) – Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL
- **blockSize** (*int*) – Size of the averaging block, passed to the underlying *CornerMinEigenVal* or *CornerHarris* used by the function
- **useHarris** (*int*) – If nonzero, Harris operator (*CornerHarris*) is used instead of default *CornerMinEigenVal*
- **k** (*float*) – Free parameter of Harris detector; used only if (`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

HoughLines2(*image*, *storage*, *method*, *rho*, *theta*, *threshold*, *param1=0*, *param2=0*) → lines

Finds lines in a binary image using a Hough transform.

Parameters

- **image** (CvArr) – The 8-bit, single-channel, binary source image. In the case of a probabilistic method, the image is modified by the function
- **storage** (CvMemStorage) – 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** (*int*) – 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** (*float*) – Distance resolution in pixel-related units
- **theta** (*float*) – Angle resolution measured in radians
- **threshold** (*int*) – Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`
- **param1** (*float*) – 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** (*float*) – 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.

PreCornerDetect

PreCornerDetect(*image*, *corners*, *apertureSize=3*) → None

Calculates the feature map for corner detection.

Parameters

- **image** (`CvArr`) – Input image
- **corners** (`CvArr`) – Image to store the corner candidates
- **apertureSize** (*int*) – 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:

```
import cv

def precornerdetect(image):
    # assume that the image is floating-point
    corners = cv.CloneMat(image)
    cv.PreCornerDetect(image, corners, 3)

    dilated_corners = cv.CloneMat(image)
    cv.Dilate(corners, dilated_corners, None, 1)

    corner_mask = cv.CreateMat(image.rows, image.cols, cv.CV_8UC1)
    cv.Sub(corners, dilated_corners, corners)
    cv.CmpS(corners, 0, corner_mask, cv.CV_CMP_GE)
    return (corners, corner_mask)
```

3.9 Object Detection

MatchTemplate

MatchTemplate(*image*, *templ*, *result*, *method*) → None

Compares a template against overlapped image regions.

Parameters

- **image** (CvArr) – Image where the search is running; should be 8-bit or 32-bit floating-point
- **templ** (CvArr) – Searched template; must be not greater than the source image and the same data type as the image
- **result** (CvArr) – 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** (*int*) – 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

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

- 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

FOUR

FEATURES2D. FEATURE DETECTION AND DESCRIPTOR EXTRACTION

4.1 Feature detection and description

CvSURFPoint

class CvSURFPoint

A SURF keypoint, represented as a tuple `((x, y), laplacian, size, dir, hessian)`.

x
x-coordinate of the feature within the image

y
y-coordinate of the feature within the image

laplacian
-1, 0 or +1. sign of the laplacian at the point. Can be used to speedup feature comparison since features with laplacians of different signs can not match

size
size of the feature

dir
orientation of the feature: 0..360 degrees

hessian
value of the hessian (can be used to approximately estimate the feature strengths; see also params.hessianThreshold)

ExtractSURF

ExtractSURF(*image*, *mask*, *storage*, *params*)-> *(keypoints, descriptors)*
Extracts Speeded Up Robust Features from an image.

Parameters

- **image** (CvArr) – The input 8-bit grayscale image
- **mask** (CvArr) – 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** (CvSeq of CvSURFPoint) – sequence of keypoints.

- **descriptors** (CvSeq of list of float) – sequence of descriptors. Each SURF descriptor is a list of floats, of length 64 or 128.
- **storage** (CvMemStorage) – Memory storage where keypoints and descriptors will be stored
- **params** (CvSURFParams) – Various algorithm parameters in a tuple `(extended, hessianThreshold, nOctaves, nOctaveLayers)`:
  - **extended** 0 means basic descriptors (64 elements each), 1 means extended descriptors (128 elements each)
  - **hessianThreshold** only features with 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.
  - **nOctaves** the number of octaves to be used for extraction. With each next octave the feature size is doubled (3 by default)
  - **nOctaveLayers** The number of layers within each octave (4 by default)

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.

To extract strong SURF features from an image

GetStarKeypoints

GetStarKeypoints(*image*, *storage*, *params*) → keypoints

Retrieves keypoints using the StarDetector algorithm.

Parameters

- **image** (CvArr) – The input 8-bit grayscale image
- **storage** (CvMemStorage) – Memory storage where the keypoints will be stored
- **params** (CvStarDetectorParams) – Various algorithm parameters in a tuple `(maxSize, responseThreshold, lineThresholdProjected, lineThresholdBinarized, suppressNonmaxSize)`:
  - **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
  - **responseThreshold** threshold for the approximatd laplacian, used to eliminate weak features
  - **lineThresholdProjected** another threshold for laplacian to eliminate edges
  - **lineThresholdBinarized** another threshold for the feature scale to eliminate edges
  - **suppressNonmaxSize** linear size of a pixel neighborhood for non-maxima suppression

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 keypoint is represented by a tuple `((x, y), size, response)`:

- **x, y** Screen coordinates of the keypoint
- **size** feature size, up to `maxSize`
- **response** approximated laplacian value for the keypoint

CHAPTER

FIVE

OBJDETECT. OBJECT DETECTION

5.1 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).

A simple demonstration of face detection, which draws a rectangle around each detected face:

```
hc = cv.Load("haarcascade_frontalface_default.xml")
img = cv.LoadImage("faces.jpg", 0)
faces = cv.HaarDetectObjects(img, hc, cv.CreateMemStorage())
for (x,y,w,h),n in faces:
    cv.Rectangle(img, (x,y), (x+w,y+h), 255)
cv.SaveImage("faces_detected.jpg", img)
```

HaarDetectObjects

HaarDetectObjects(*image*, *cascade*, *storage*, *scaleFactor=1.1*, *minNeighbors=3*, *flags=0*, *minSize=(0, 0)*) → detected_objects

Detects objects in the image.

Parameters

- **image** (CvArr) – Image to detect objects in
- **cascade** (CvHaarClassifierCascade) – Haar classifier cascade in internal representation
- **storage** (CvMemStorage) – Memory storage to store the resultant sequence of the object candidate rectangles
- **scaleFactor** – The factor by which the search window is scaled between the subsequent scans, 1.1 means increasing window by 10 %
- **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
- **flags** (*int*) – 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
- **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)
- **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).

The function returns a list of tuples, `(rect, neighbors)` , where rect is a *CvRect* specifying the object's extents and neighbors is a number of neighbors.

CHAPTER

SIX

VIDEO. VIDEO ANALYSIS

6.1 Motion Analysis and Object Tracking

CalcGlobalOrientation

CalcGlobalOrientation(*orientation*, *mask*, *mhi*, *timestamp*, *duration*) → float

Calculates the global motion orientation of some selected region.

Parameters

- **orientation** (CvArr) – Motion gradient orientation image; calculated by the function *CalcMotionGradient*
- **mask** (CvArr) – 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** (CvArr) – Motion history image
- **timestamp** (*float*) – 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** (*float*) – 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

CalcMotionGradient(*mhi*, *mask*, *orientation*, *delta1*, *delta2*, *apertureSize=3*) → None

Calculates the gradient orientation of a motion history image.

Parameters

- **mhi** (CvArr) – Motion history image
- **mask** (CvArr) – Mask image; marks pixels where the motion gradient data is correct; output parameter

- **orientation** (CvArr) – Motion gradient orientation image; contains angles from 0 to ~360 degrees
- **delta1** (*float*) – See below
- **delta2** (*float*) – See below
- **apertureSize** (*int*) – 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

$$\min(\texttt{delta1}, \texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1}, \texttt{delta2}).$$

CalcOpticalFlowBM

CalcOpticalFlowBM(*prev*, *curr*, *blockSize*, *shiftSize*, *max_range*, *usePrevious*, *velx*, *vely*) → None

Calculates the optical flow for two images by using the block matching method.

Parameters

- **prev** (CvArr) – First image, 8-bit, single-channel
- **curr** (CvArr) – Second image, 8-bit, single-channel
- **blockSize** (CvSize) – Size of basic blocks that are compared
- **shiftSize** (CvSize) – Block coordinate increments
- **max_range** (CvSize) – Size of the scanned neighborhood in pixels around the block
- **usePrevious** (*int*) – Uses the previous (input) velocity field
- **velx** (CvArr) – 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** (CvArr) – 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 blockSize.width × 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

CalcOpticalFlowHS(*prev*, *curr*, *usePrevious*, *velx*, *vely*, *lambda*, *criteria*) → None

Calculates the optical flow for two images.

Parameters

- **prev** (CvArr) – First image, 8-bit, single-channel
- **curr** (CvArr) – Second image, 8-bit, single-channel
- **usePrevious** (*int*) – Uses the previous (input) velocity field
- **velx** (CvArr) – Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel
- **vely** (CvArr) – Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel
- **lambda** (*float*) – Lagrangian multiplier
- **criteria** (CvTermCriteria) – 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

CalcOpticalFlowLK(*prev*, *curr*, *winSize*, *velx*, *vely*) → None

Calculates the optical flow for two images.

Parameters

- **prev** (CvArr) – First image, 8-bit, single-channel
- **curr** (CvArr) – Second image, 8-bit, single-channel
- **winSize** (CvSize) – Size of the averaging window used for grouping pixels
- **velx** (CvArr) – Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel
- **vely** (CvArr) – 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

CalcOpticalFlowPyrLK(*prev*, *curr*, *prevPyr*, *currPyr*, *prevFeatures*, *winSize*, *level*, *criteria*, *flags*, *guesses = None*) -> (*currFeatures*, *status*, *track_error*)

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

Parameters

- **prev** (CvArr) – First frame, at time `t`
- **curr** (CvArr) – Second frame, at time `t + dt`

- **prevPyr** (CvArr) – 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** (CvArr) – Similar to prevPyr , used for the second frame
- **prevFeatures** (CvPoint2D32f) – Array of points for which the flow needs to be found
- **currFeatures** (CvPoint2D32f) – Array of 2D points containing the calculated new positions of the input features in the second image
- **winSize** (CvSize) – Size of the search window of each pyramid level
- **level** (*int*) – Maximal pyramid level number. If 0 , pyramids are not used (single level), if 1 , two levels are used, etc
- **status** (*str*) – Array. Every element of the array is set to 1 if the flow for the corresponding feature has been found, 0 otherwise
- **track_error** (*float*) – Array of double numbers containing the difference between patches around the original and moved points. Optional parameter; can be NULL
- **criteria** (CvTermCriteria) – Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped
- **flags** (*int*) – 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
- **guesses** (CvPoint2D32f) – optional array of estimated coordinates of features in second frame, with same length as prevFeatures

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

CamShift(*prob_image*, *window*, *criteria*)-> (*int*, *comp*, *box*)

Finds the object center, size, and orientation.

Parameters

- **prob_image** (CvArr) – Back projection of object histogram (see *CalcBackProject*)
- **window** (CvRect) – Initial search window
- **criteria** (CvTermCriteria) – Criteria applied to determine when the window search should be finished

- **comp** (`CvConnectedComp`) – 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** (`CvBox2D`) – Circumscribed box for the object.

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.

CvKalman

class `CvKalman`

Kalman filter state.

`MP`
number of measurement vector dimensions

`DP`
number of state vector dimensions

`CP`
number of control vector dimensions

`state_pre`
predicted state (x'(k)): x(k)=A*x(k-1)+B*u(k)

`state_post`
corrected state (x(k)): x(k)=x'(k)+K(k)*(z(k)-H*x'(k))

`transition_matrix`
state transition matrix (A)

`control_matrix`
control matrix (B) (it is not used if there is no control)

`measurement_matrix`
measurement matrix (H)

`process_noise_cov`
process noise covariance matrix (Q)

`measurement_noise_cov`
measurement noise covariance matrix (R)

`error_cov_pre`
priori error estimate covariance matrix (P'(k)): P'(k)=A*P(k-1)*At + Q

`gain`
Kalman gain matrix (K(k)): K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)

`error_cov_post`
posteriori error estimate covariance matrix (P(k)): P(k)=(I-K(k)*H)*P'(k)

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 . Normally, the structure is used for the standard Kalman filter

(notation and the formulas below are borrowed from the excellent Kalman tutorial Welch95)

$$x_k = A \cdot x_{k-1} + B \cdot u_k + w_k$$
$$z_k = H \cdot x_k + v_k$$

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:

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

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

CreateKalman(*dynam_params*, *measure_params*, *control_params=0*) → CvKalman

Allocates the Kalman filter structure.

Parameters

- **dynam_params** (*int*) – dimensionality of the state vector
- **measure_params** (*int*) – dimensionality of the measurement vector
- **control_params** (*int*) – dimensionality of the control vector

The function allocates *CvKalman* and all its matrices and initializes them somehow.

KalmanCorrect

KalmanCorrect(*kalman*, *measurement*) → cvmat

Adjusts the model state.

Parameters

- **kalman** (`CvKalman`) – Kalman filter object returned by *CreateKalman*
- **measurement** (`CvMat`) – CvMat containing the measurement vector

The function adjusts the stochastic model state on the basis of the given measurement of the model state:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

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.

KalmanPredict

KalmanPredict(*kalman*, *control=None*) → cvmat

Estimates the subsequent model state.

Parameters

- **kalman** (CvKalman) – Kalman filter object returned by *CreateKalman*
- **control** (CvMat) – 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`:

$$x'_k = Ax_{k-1} + Bu_k$$
$$P'_k = AP_{k-1}A^T + Q$$

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

MeanShift(*prob_image*, *window*, *criteria*) → comp

Finds the object center on back projection.

Parameters

- **prob_image** (CvArr) – Back projection of the object histogram (see *CalcBackProject*)
- **window** (CvRect) – Initial search window
- **criteria** (CvTermCriteria) – Criteria applied to determine when the window search should be finished
- **comp** (CvConnectedComp) – 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.

SegmentMotion

SegmentMotion(*mhi*, *seg_mask*, *storage*, *timestamp*, *seg_thresh*) → None

Segments a whole motion into separate moving parts.

Parameters

- **mhi** (CvArr) – Motion history image
- **seg_mask** (CvArr) – Image where the mask found should be stored, single-channel, 32-bit floating-point
- **storage** (CvMemStorage) – Memory storage that will contain a sequence of motion connected components
- **timestamp** (*float*) – Current time in milliseconds or other units
- **seg_thresh** (*float*) – 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

SnakeImage(*image*, *points*, *alpha*, *beta*, *gamma*, *win*, *criteria*, *calc_gradient=1*) → new_points

Changes the contour position to minimize its energy.

Parameters

- **image** (IplImage) – The source image or external energy field
- **points** (CvPoints) – Contour points (snake)
- **alpha** (*sequence of float*) – Weight[s] of continuity energy, single float or a list of floats, one for each contour point
- **beta** (*sequence of float*) – Weight[s] of curvature energy, similar to `alpha`
- **gamma** (*sequence of float*) – Weight[s] of image energy, similar to `alpha`
- **win** (CvSize) – Size of neighborhood of every point used to search the minimum, both `win.width` and `win.height` must be odd
- **criteria** (CvTermCriteria) – Termination criteria
- **calc_gradient** (*int*) – 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.

The function returns the updated list of points.

UpdateMotionHistory

UpdateMotionHistory(*silhouette*, *mhi*, *timestamp*, *duration*) → None

Updates the motion history image by a moving silhouette.

Parameters

- **silhouette** (CvArr) – Silhouette mask that has non-zero pixels where the motion occurs
- **mhi** (CvArr) – Motion history image, that is updated by the function (single-channel, 32-bit floating-point)
- **timestamp** (*float*) – Current time in milliseconds or other units
- **duration** (*float*) – Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

CHAPTER

SEVEN

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.

7.1 User Interface

CreateTrackbar

CreateTrackbar(*trackbarName*, *windowName*, *value*, *count*, *onChange*) → None

Creates a trackbar and attaches it to the specified window

Parameters

- **trackbarName** (*str*) – Name of the created trackbar.
- **windowName** (*str*) – Name of the window which will be used as a parent for created trackbar.
- **value** (*int*) – Initial value for the slider position, between 0 and `count` .
- **count** (*int*) – Maximal position of the slider. Minimal position is always 0.
- **onChange** (`PyCallableObject`) – OpenCV calls `onChange` every time the slider changes position. OpenCV will call it as `func(x)` where `x` is the new position of the slider.

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.

DestroyAllWindows

DestroyAllWindows() → None
Destroys all of the HighGUI windows.

The function `cvDestroyAllWindows` destroys all of the opened HighGUI windows.

DestroyWindow

DestroyWindow(*name*) → None
Destroys a window.

Parameters **name** (*str*) – Name of the window to be destroyed.

The function `cvDestroyWindow` destroys the window with the given name.

GetTrackbarPos

GetTrackbarPos(*trackbarName*, *windowName*) → None
Returns the trackbar position.

Parameters

- **trackbarName** (*str*) – Name of the trackbar.
- **windowName** (*str*) – 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.

MoveWindow

MoveWindow(*name*, *x*, *y*) → None
Sets the position of the window.

Parameters

- **name** (*str*) – Name of the window to be moved.
- **x** (*int*) – New x coordinate of the top-left corner
- **y** (*int*) – New y coordinate of the top-left corner

The function `cvMoveWindow` changes the position of the window.

NamedWindow

NamedWindow(*name*, *flags=CV_WINDOW_AUTOSIZE*) → None

Creates a window.

Parameters

- **name** (*str*) – Name of the window in the window caption that may be used as a window identifier.
- **flags** (*int*) – 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

ResizeWindow(*name*, *width*, *height*) → None

Sets the window size.

Parameters

- **name** (*str*) – Name of the window to be resized.
- **width** (*int*) – New width
- **height** (*int*) – New height

The function `cvResizeWindow` changes the size of the window.

SetMouseCallback

SetMouseCallback(*windowName*, *onMouse*, *param*) → None

Assigns callback for mouse events.

Parameters

- **windowName** (*str*) – Name of the window.
- **onMouse** (`PyCallableObject`) – Callable to be called every time a mouse event occurs in the specified window. This callable should have signature `` Foo(event, x, y, flags, param)-> None `` 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** (*object*) – 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

SetTrackbarPos(*trackbarName*, *windowName*, *pos*) → None

Sets the trackbar position.

Parameters

- **trackbarName** (*str*) – Name of the trackbar.
- **windowName** (*str*) – Name of the window which is the parent of trackbar.
- **pos** (*int*) – 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

ShowImage(*name*, *image*) → None

Displays the image in the specified window

Parameters

- **name** (*str*) – Name of the window.
- **image** (CvArr) – 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

WaitKey(*delay=0*) → int

Waits for a pressed key.

Parameters **delay** (*int*) – 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.

Note: 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

7.2 Reading and Writing Images and Video

LoadImage

LoadImage(*filename*, *iscolor=CV_LOAD_IMAGE_COLOR*) → None

Loads an image from a file as an IplImage.

Parameters

- **filename** (*str*) – Name of file to be loaded.
- **iscolor** (*int*) – 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

LoadImageM(*filename*, *iscolor=CV_LOAD_IMAGE_COLOR*) → None

Loads an image from a file as a CvMat.

Parameters

- **filename** (*str*) – Name of file to be loaded.
- **iscolor** (*int*) – 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

SaveImage(*filename*, *image*) → None

Saves an image to a specified file.

Parameters

- **filename** (*str*) – Name of the file.
- **image** (`CvArr`) – 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

class CvCapture

Video capturing structure.

The structure CvCapture does not have a public interface and is used only as a parameter for video capturing functions.

CaptureFromCAM

CaptureFromCAM(*index*) → CvCapture

Initializes capturing a video from a camera.

Parameters **index** (*int*) – 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

CaptureFromFile(*filename*) → CvCapture

Initializes capturing a video from a file.

Parameters **filename** (*str*) – 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

GetCaptureProperty(*capture*, *property_id*) → double

Gets video capturing properties.

Parameters

- **capture** (CvCapture) – video capturing structure.
- **property_id** – Property identifier. Can be one of the following:

The function cvGetCaptureProperty retrieves the specified property of the camera or video file.

GrabFrame

GrabFrame(*capture*) → int
Grabs the frame from a camera or file.

Parameters **capture** (CvCapture) – 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

QueryFrame(*capture*) → iplimage
Grabs and returns a frame from a camera or file.

Parameters **capture** (CvCapture) – 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.

RetrieveFrame

RetrieveFrame(*capture*) → iplimage
Gets the image grabbed with cvGrabFrame.

Parameters **capture** (CvCapture) – 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

SetCaptureProperty(*capture*, *property_id*, *value*) → None
Sets video capturing properties.

Parameters

- **capture** (CvCapture) – video capturing structure.
- **property_id** – property identifier. Can be one of the following:
- **value** (*float*) – 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

CreateVideoWriter(*filename*, *fourcc*, *fps*, *frame_size*, *is_color*) → CvVideoWriter

Creates the video file writer.

Parameters

- **filename** (*str*) – Name of the output video file.
- **fourcc** (*int*) – 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** (*float*) – Framerate of the created video stream.
- **frame_size** (`CvSize`) – Size of the video frames.
- **is_color** (*int*) – 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.

WriteFrame

WriteFrame(*writer*, *image*) → int

Writes a frame to a video file.

Parameters

- **writer** (`CvVideoWriter`) – Video writer structure
- **image** (`IplImage`) – The written frame

The function `cvWriteFrame` writes/appends one frame to a video file.

CHAPTER

EIGHT

CALIB3D. CAMERA CALIBRATION, POSE ESTIMATION AND STEREO

8.1 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 $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distortion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2+2x'^2)$$
$$y'' = y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2+2y'^2) + 2p_2 x' y'$$
$$\text{where} \quad r^2 = x'^2 + y'^2$$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

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.

CalibrateCamera2

CalibrateCamera2(*objectPoints*, *imagePoints*, *pointCounts*, *imageSize*, *cameraMatrix*, *distCoeffs*, *rvecs*, *tvecs*, *flags=0*) → None

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

Parameters

- **objectPoints** (CvMat) – 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** (CvMat) – 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** (CvMat) – 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** (CvSize) – Size of the image, used only to initialize the intrinsic camera matrix

- **cameraMatrix** (CvMat) – 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** (CvMat) – 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** (CvMat) – 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** (CvMat) – 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.

    type flags int
  - **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* .

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

ComputeCorrespondEpilines(*points*, *whichImage*, *F*, *lines*) → None

For points in one image of a stereo pair, computes the corresponding epilines in the other image.

Parameters

- **points** (`CvMat`) – The input points. `2xN, Nx2, 3xN` or `Nx3` array (where `N` number of points). Multi-channel `1xN` or `Nx1` array is also acceptable
- **whichImage** (*int*) – Index of the image (1 or 2) that contains the `points`
- **F** (`CvMat`) – The fundamental matrix that can be estimated using *FindFundamentalMat* or *StereoRectify* .
- **lines** (`CvMat`) – 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 $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2` , $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

ConvertPointsHomogeneous

ConvertPointsHomogeneous(*src*, *dst*) → None

Convert points to/from homogeneous coordinates.

Parameters

- **src** (CvMat) – The input array or vector of 2D, 3D or 4D points
- **dst** (CvMat) – The output vector of 2D or 2D points

The 2D or 3D points from/to homogeneous coordinates, or simply the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$$x' = x/w$$
$$y' = y/w$$
$$z' = z/w \quad \text{(if output is 3D)}$$

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.

CreatePOSITObject

CreatePOSITObject(*points*) → POSITObject

Initializes a structure containing object information.

Parameters **points** (CvPoint3D32fs) – List of 3D 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

CreateStereoBMState(*preset=CV_STEREO_BM_BASIC*, *numberOfDisparities=0*) → StereoBMState

Creates block matching stereo correspondence structure.

Parameters

- **preset** (*int*) – 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** (*int*) – 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

CreateStereoGCState(*numberOfDisparities*, *maxIters*) → StereoGCState

Creates the state of graph cut-based stereo correspondence algorithm.

Parameters

- **numberOfDisparities** (*int*) – The number of disparities. The disparity search range will be `state->minDisparity` $\leq$ *disparity* $<$ `state->minDisparity` + `state->numberOfDisparities`
- **maxIters** (*int*) – 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

class CvStereoBMState

The structure for block matching stereo correspondence algorithm.

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

class CvStereoGCState

The structure for graph cuts-based stereo correspondence algorithm

Ithreshold
threshold for piece-wise linear data cost function (5 by default)

interactionRadius
radius for smoothness cost function (1 by default; means Potts model)

K, lambda, lambda1, lambda2
parameters for the cost function (usually computed adaptively from the input data)

occlusionCost
10000 by default

minDisparity
0 by default; see *CvStereoBMState*

numberOfDisparities
defined by user; see *CvStereoBMState*

maxIters
number of iterations; defined by user.

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

DecomposeProjectionMatrix(*projMatrix*, *cameraMatrix*, *rotMatrix*, *transVect*, *rotMatrX* = *None*, *rotMatrY* = *None*, *rotMatrZ* = *None*) → eulerAngles

Decomposes the projection matrix into a rotation matrix and a camera matrix.

Parameters

- **projMatrix** (CvMat) – The 3x4 input projection matrix P
- **cameraMatrix** (CvMat) – The output 3x3 camera matrix K
- **rotMatrix** (CvMat) – The output 3x3 external rotation matrix R
- **transVect** (CvMat) – The output 4x1 translation vector T
- **rotMatrX** (CvMat) – Optional 3x3 rotation matrix around x-axis
- **rotMatrY** (CvMat) – Optional 3x3 rotation matrix around y-axis
- **rotMatrZ** (CvMat) – Optional 3x3 rotation matrix around z-axis
- **eulerAngles** (CvPoint3D64f) – 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

DrawChessboardCorners(*image*, *patternSize*, *corners*, *patternWasFound*) → None

Renders the detected chessboard corners.

Parameters

- **image** (CvArr) – The destination image; it must be an 8-bit color image
- **patternSize** (CvSize) – The number of inner corners per chessboard row and column. (patternSize = cv::Size(points _ per _ row,points _ per _ column) = cv::Size(rows,columns))
- **corners** (*sequence of (float, float)*) – The array of corners detected, this should be the output from findChessboardCorners wrapped in a cv::Mat().
- **patternWasFound** (*int*) – 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

FindChessboardCorners(*image*, *patternSize*, *flags=CV_CALIB_CB_ADAPTIVE_THRESH*) → corners

Finds the positions of the internal corners of the chessboard.

Parameters

- **image** (CvArr) – Source chessboard view; it must be an 8-bit grayscale or color image
- **patternSize** (CvSize) – The number of inner corners per chessboard row and column (patternSize = cvSize(points _ per _ row,points _ per _ colum) = cvSize(columns,rows))
- **corners** (*sequence of (float, float)*) – The output array of corners detected
- **flags** (*int*) – 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

FindExtrinsicCameraParams2(*objectPoints*, *imagePoints*, *cameraMatrix*, *distCoeffs*, *rvec*, *tvec*, *useExtrinsicGuess=0*) → None

Finds the object pose from the 3D-2D point correspondences

Parameters

- **objectPoints** (CvMat) – 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** (CvMat) – 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** (CvMat) – The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** (CvMat) – 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** (CvMat) – The output rotation vector (see *Rodrigues2*) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system
- **tvec** (CvMat) – The output translation vector
- **useExtrinsicGuess** (*int*) – 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

FindFundamentalMat(*points1*, *points2*, *fundamentalMatrix*, *method=CV_FM_RANSAC*, *param1=1.*, *param2=0.99*, *status = None*) → None

Calculates the fundamental matrix from the corresponding points in two images.

Parameters

- **points1** (CvMat) – 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** (CvMat) – Array of the second image points of the same size and format as `points1`
- **fundamentalMatrix** (CvMat) – The output fundamental matrix or matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices)
- **method** (*int*) – 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** (*float*) – 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** (*float*) – 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** (CvMat) – 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.

FindHomography

FindHomography(*srcPoints*, *dstPoints*, *H*, *method*, *ransacReprojThreshold=3.0*, *status=None*) → None

Finds the perspective transformation between two planes.

Parameters

- **srcPoints** (CvMat) – 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** (CvMat) – Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 1-channel, or 1xN or Nx1 2- or 3-channel array.
- **H** (CvMat) – The output 3x3 homography matrix
- **method** (*int*) – 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** (*float*) – 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** (CvMat) – 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

FindStereoCorrespondenceBM(*left*, *right*, *disparity*, *state*) → None

Computes the disparity map using block matching algorithm.

Parameters

- **left** (`CvArr`) – The left single-channel, 8-bit image.
- **right** (`CvArr`) – The right image of the same size and the same type.
- **disparity** (`CvArr`) – 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** (`CvStereoBMState`) – 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

FindStereoCorrespondenceGC(*left*, *right*, *dispLeft*, *dispRight*, *state*, *useDisparityGuess=(0)*) → None

Computes the disparity map using graph cut-based algorithm.

Parameters

- **left** (CvArr) – The left single-channel, 8-bit image.
- **right** (CvArr) – The right image of the same size and the same type.
- **dispLeft** (CvArr) – The optional output single-channel 16-bit signed left disparity map of the same size as input images.
- **dispRight** (CvArr) – The optional output single-channel 16-bit signed right disparity map of the same size as input images.
- **state** (CvStereoGCState) – Stereo correspondence structure.
- **useDisparityGuess** (*int*) – If the parameter is not zero, the algorithm will start with predefined 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:

$$\texttt{state->minDisparity} \leq dispRight(x,y) < \texttt{state->minDisparity} + \texttt{state->numberOfDisparities}$$

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:

```
import sys
import cv

def findstereocorrespondence(image_left, image_right):
    # image_left and image_right are the input 8-bit single-channel images
    # from the left and the right cameras, respectively
    (r, c) = (image_left.rows, image_left.cols)
    disparity_left = cv.CreateMat(r, c, cv.CV_16S)
    disparity_right = cv.CreateMat(r, c, cv.CV_16S)
    state = cv.CreateStereoGCState(16, 2)
    cv.FindStereoCorrespondenceGC(image_left, image_right, disparity_left, disparity_right, state, 0)
    return (disparity_left, disparity_right)


if __name__ == '__main__':

    (l, r) = [cv.LoadImageM(f, cv.CV_LOAD_IMAGE_GRAYSCALE) for f in sys.argv[1:]]
```

```
(disparity_left, disparity_right) = findstereocorrespondence(l, r)

disparity_left_visual = cv.CreateMat(l.rows, l.cols, cv.CV_8U)
cv.ConvertScale(disparity_left, disparity_left_visual, -16)
cv.SaveImage("disparity.pgm", disparity_left_visual)
```

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):

GetOptimalNewCameraMatrix

GetOptimalNewCameraMatrix(*cameraMatrix*, *distCoeffs*, *imageSize*, *alpha*, *newCameraMatrix*, *newImageSize=(0, 0)*, *validPixROI=0*) → None

Returns the new camera matrix based on the free scaling parameter

Parameters

- **cameraMatrix** (CvMat) – The input camera matrix
- **distCoeffs** (CvMat) – 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** (CvSize) – The original image size
- **alpha** (*float*) – 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** (CvMat) – The output new camera matrix.
- **newImageSize** (CvSize) – The image size after rectification. By default it will be set to `imageSize` .
- **validPixROI** (CvRect) – 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

InitIntrinsicParams2D(*objectPoints*, *imagePoints*, *npoints*, *imageSize*, *cameraMatrix*, *aspectRatio=1.*) → None

Finds the initial camera matrix from the 3D-2D point correspondences

Parameters

- **objectPoints** (CvMat) – The joint array of object points; see *CalibrateCamera2*
- **imagePoints** (CvMat) – The joint array of object point projections; see *CalibrateCamera2*
- **npoints** (CvMat) – The array of point counts; see *CalibrateCamera2*
- **imageSize** (CvSize) – The image size in pixels; used to initialize the principal point

- **cameraMatrix** (CvMat) – The output camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **aspectRatio** (*float*) – 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

InitUndistortMap(*cameraMatrix*, *distCoeffs*, *map1*, *map2*) → None

Computes an undistortion map.

Parameters

- **cameraMatrix** (CvMat) – The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** (CvMat) – 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** (CvArr) – The first output map of type CV_32FC1 or CV_16SC2 - the second variant is more efficient
- **map2** (CvArr) – 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

InitUndistortRectifyMap(*cameraMatrix*, *distCoeffs*, *R*, *newCameraMatrix*, *map1*, *map2*) → None

Computes the undistortion and rectification transformation map.

Parameters

- **cameraMatrix** (CvMat) – The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** (CvMat) – 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** (CvMat) – 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** (CvMat) – The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$
- **map1** (CvArr) – The first output map of type CV_32FC1 or CV_16SC2 - the second variant is more efficient

- **map2** (CvArr) – 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:

$$\begin{array}{l} x \leftarrow (u - {c'}_x)/{f'}_x \\ y \leftarrow (v - {c'}_y)/{f'}_y \\ [X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\ x' \leftarrow X/W \\ y' \leftarrow Y/W \\ x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\ map_x(u, v) \leftarrow x" f_x + c_x \\ map_y(u, v) \leftarrow y" f_y + c_y \end{array}$$

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

POSIT(*posit_object*, *imagePoints*, *focal_length*, *criteria*)-> (*rotationMatrix*, *translation_vector*)

Implements the POSIT algorithm.

Parameters

- **posit_object** (CvPOSITObject) – Pointer to the object structure
- **imagePoints** (CvPoint2D32f) – Pointer to the object points projections on the 2D image plane
- **focal_length** (*float*) – Focal length of the camera used
- **criteria** (CvTermCriteria) – Termination criteria of the iterative POSIT algorithm
- **rotationMatrix** (CvMatr32f_i) – Matrix of rotations
- **translation_vector** (CvVect32f_i) – 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

ProjectPoints2(*objectPoints*, *rvec*, *tvec*, *cameraMatrix*, *distCoeffs*, *imagePoints*, *dpdrot=NULL*, *dpdt=NULL*, *dpdf=NULL*, *dpdc=NULL*, *dpddist=NULL*) → None

Project 3D points on to an image plane.

Parameters

- **objectPoints** (CvMat) – 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** (CvMat) – The rotation vector, see *Rodrigues2*
- **tvec** (CvMat) – The translation vector
- **cameraMatrix** (CvMat) – The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** (CvMat) – 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** (CvMat) – The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel
- **dpdrot** (CvMat) – Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector
- **dpdt** (CvMat) – Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector
- **dpdf** (CvMat) – Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y
- **dpdc** (CvMat) – Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y
- **dpddist** (CvMat) – 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

ReprojectImageTo3D(*disparity*, *_3dImage*, *Q*, *handleMissingValues=0*) → None

Reprojects disparity image to 3D space.

Parameters

- **disparity** (CvArr) – The input single-channel 16-bit signed or 32-bit floating-point disparity image
- **_3dImage** (CvArr) – 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** (CvMat) – The 4×4 perspective transformation matrix that can be obtained with *StereoRectify*
- **handleMissingValues** (*int*) – 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

RQDecomp3x3(*M*, *R*, *Q*, *Qx = None*, *Qy = None*, *Qz = None*) → eulerAngles

Computes the 'RQ' decomposition of 3x3 matrices.

Parameters

- **M** (CvMat) – The 3x3 input matrix
- **R** (CvMat) – The output 3x3 upper-triangular matrix
- **Q** (CvMat) – The output 3x3 orthogonal matrix
- **Qx** (CvMat) – Optional 3x3 rotation matrix around x-axis
- **Qy** (CvMat) – Optional 3x3 rotation matrix around y-axis
- **Qz** (CvMat) – Optional 3x3 rotation matrix around z-axis
- **eulerAngles** (CvPoint3D64f) – 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.

Rodrigues2

Rodrigues2(*src*, *dst*, *jacobian=0*) → None

Converts a rotation matrix to a rotation vector or vice versa.

Parameters

- **src** (CvMat) – The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)
- **dst** (CvMat) – The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively
- **jacobian** (CvMat) – Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

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

StereoCalibrate(*objectPoints*, *imagePoints1*, *imagePoints2*, *pointCounts*, *cameraMatrix1*, *distCoeffs1*, *cameraMatrix2*, *distCoeffs2*, *imageSize*, *R*, *T*, *E=NULL*, *F=NULL*, *term_crit=(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS*, *30*, *1e-6)*, *flags=CV_CALIB_FIX_INTRINSIC*) → None

Calibrates stereo camera.

Parameters

- **objectPoints** (CvMat) – 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** (CvMat) – 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** (CvMat) – 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** (CvMat) – 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** (CvMat) – 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

- **distCoeffs** – 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** (CvMat) – The input/output second camera matrix, as cameraMatrix1.
- **distCoeffs2** (CvMat) – The input/output lens distortion coefficients for the second camera, as `distCoeffs1` .
- **imageSize** (CvSize) – Size of the image, used only to initialize intrinsic camera matrix.
- **R** (CvMat) – The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.
- **T** (CvMat) – The output translation vector between the cameras' coordinate systems.
- **E** (CvMat) – The optional output essential matrix.
- **F** (CvMat) – The optional output fundamental matrix.
- **term_crit** (CvTermCriteria) – The termination criteria for the iterative optimization algorithm.
- **flags** (*int*) – 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:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

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.

StereoRectify

StereoRectify(*cameraMatrix1*, *cameraMatrix2*, *distCoeffs1*, *distCoeffs2*, *imageSize*, *R*, *T*, *R1*, *R2*, *P1*, *P2*, *Q=NULL*, *flags=CV_CALIB_ZERO_DISPARITY*, *alpha=-1*, *newImageSize=(0, 0)*)-> (*roi1*, *roi2*)

Computes rectification transforms for each head of a calibrated stereo camera.

Parameters

- **cameraMatrix2** (*cameraMatrix1,*) – The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.
- **distCoeffs** – The input vectors 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 each. If the vectors are NULL/empty, the zero distortion coefficients are assumed.
- **imageSize** (`CvSize`) – Size of the image used for stereo calibration.
- **R** (`CvMat`) – The rotation matrix between the 1st and the 2nd cameras' coordinate systems.
- **T** (`CvMat`) – 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** (`CvMat`) – The output 4×4 disparity-to-depth mapping matrix, see `reprojectImageTo3D()` .
- **flags** (*int*) – 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** (*float*) – 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** (`CvSize`) – 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

StereoRectifyUncalibrated(*points1*, *points2*, *F*, *imageSize*, *H1*, *H2*, *threshold=5*) → None

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** (CvMat) – The input fundamental matrix. It can be computed from the same set of point pairs using *FindFundamentalMat* .
- **imageSize** (CvSize) – Size of the image.
- **H2** (*H1,*) – The output rectification homography matrices for the first and for the second images.
- **threshold** (*float*) – 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

Undistort2(*src*, *dst*, *cameraMatrix*, *distCoeffs*) → None

Transforms an image to compensate for lens distortion.

Parameters

- **src** (CvArr) – The input (distorted) image
- **dst** (CvArr) – The output (corrected) image; will have the same size and the same type as src
- **cameraMatrix** (CvMat) – The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** (CvMat) – 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

UndistortPoints(*src*, *dst*, *cameraMatrix*, *distCoeffs*, *R=NULL*, *P=NULL*) → None

Computes the ideal point coordinates from the observed point coordinates.

Parameters

- **src** (`CvMat`) – The observed point coordinates, 1xN or Nx1 2-channel (CV _ 32FC2 or CV _ 64FC2).
- **dst** (`CvMat`) – The output ideal point coordinates, after undistortion and reverse perspective transformation , same format as `src` .
- **cameraMatrix** (`CvMat`) – The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- **distCoeffs** (`CvMat`) – 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** (`CvMat`) – 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** (`CvMat`) – 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 None).