

# OpenCV 2.0 C++ Reference

---

The OpenCV Wiki is here: <http://opencv.willowgarage.com/>

Contents:

- [Introduction](#)
  - [Namespace `cv` \(\) and Function Naming](#)
  - [Memory Management](#)
  - [Memory Management Part II. Automatic Data Allocation](#)
  - [Algebraic Operations](#)
  - [Fast Element Access](#)
  - [Saturation Arithmetics](#)
  - [Error handling](#)
  - [Threading and Reenterability](#)
- [cxcore. The Core Functionality](#)
  - [Basic Structures](#)
  - [Operations on Arrays](#)
  - [Dynamic Structures](#)
  - [Drawing Functions](#)
  - [XML/YAML Persistence](#)
  - [Clustering and Search in Multi-Dimensional Spaces](#)
  - [Utility and System Functions and Macros](#)
- [cv. Image Processing and Computer Vision](#)
  - [Image Filtering](#)
  - [Geometric Image Transformations](#)
  - [Miscellaneous Image Transformations](#)
  - [Histograms](#)
  - [Feature Detection](#)
  - [Motion Analysis and Object Tracking](#)
  - [Structural Analysis and Shape Descriptors](#)
  - [Planar Subdivisions](#)
  - [Object Detection](#)
  - [Camera Calibration and 3D Reconstruction](#)
- [highgui. High-level GUI and Media IO](#)
  - [User Interface](#)
  - [Reading and Writing Images and Video](#)
- [ml. Machine Learning](#)
  - [Statistical Models](#)
  - [Normal Bayes Classifier](#)
  - [K Nearest Neighbors](#)
  - [Support Vector Machines](#)
  - [Decision Trees](#)
  - [Boosting](#)
  - [Random Trees](#)

- [Expectation-Maximization](#)
- [Neural Networks](#)

## Indices and tables

---

- [Index](#)
- [Search Page](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Introduction

---

- [Namespace `cv` \(\) and Function Naming](#)
- [Memory Management](#)
- [Memory Management Part II. Automatic Data Allocation](#)
- [Algebraic Operations](#)
- [Fast Element Access](#)
- [Saturation Arithmetics](#)
- [Error handling](#)
- [Threading and Reenterability](#)

Starting from OpenCV 2.0 the new modern C++ interface has been introduced. It is crisp (less typing is needed to code the same thing), type-safe (no more **CvArr\*** a.k.a. `void*`) and, in general, more convenient to use. Here is a short example of what it looks like:

```
//
// Simple retro-style photo effect done by adding noise to
// the luminance channel and reducing intensity of the chroma
//

// include standard OpenCV headers, same as before
#include "cv.h"
#include "highgui.h"

// all the new API is put into "cv" namespace. Export its con
using namespace cv;

// enable/disable use of mixed API in the code below.
#define DEMO_MIXED_API_USE 1

int main( int argc, char** argv )
{
    const char* imagename = argc > 1 ? argv[1] : "lena.jpg";
    #if DEMO_MIXED_API_USE
        // Ptr<T> is safe ref-counting pointer class
        Ptr<IplImage> iplimg = cvLoadImage(imagename);

        // cv::Mat replaces the CvMat and IplImage, but it's easy
        // between the old and the new data structures
        // (by default, only the header is converted and the data
        Mat img(iplimg);
    #else
        // the newer cvLoadImage alternative with MATLAB-style na
        Mat img = imread(imagename);
    #endif

    if( !img.data ) // check if the image has been loaded pro
```

```

        return -1;

    Mat img_yuv;
    // convert image to YUV color space.
    // The output image will be allocated automatically
    cvtColor(img, img_yuv, CV_BGR2YCrCb);

    // split the image into separate color planes
    vector<Mat> planes;
    split(img_yuv, planes);

    // another Mat constructor; allocates a matrix of the specified size
    Mat noise(img.size(), CV_8U);

    // fills the matrix with normally distributed random values;
    // there is also randu() for uniformly distributed random numbers
    // Scalar replaces CvScalar, Scalar::all() replaces cvScalarAll()
    randn(noise, Scalar::all(128), Scalar::all(20));

    // blur the noise a bit, kernel size is 3x3 and both sigma's are 0.5
    GaussianBlur(noise, noise, Size(3, 3), 0.5, 0.5);

    const double brightness_gain = 0;
    const double contrast_gain = 1.7;
#if DEMO_MIXED_API_USE
    // it's easy to pass the new matrices to the functions that
    // only work with IplImage or CvMat:
    // step 1) - convert the headers, data will not be copied
    IplImage cv_planes_0 = planes[0], cv_noise = noise;
    // step 2) call the function; do not forget unary "&" to form pointer
    cvAddWeighted(&cv_planes_0, contrast_gain, &cv_noise, 1,
                  -128 + brightness_gain, &cv_planes_0);
#else
    addWeighted(planes[0], contrast_gain, noise, 1,
                -128 + brightness_gain, planes[0]);
#endif
    const double color_scale = 0.5;
    // Mat::convertTo() replaces cvConvertScale.
    // One must explicitly specify the output matrix type
    // (we keep it intact, i.e. pass planes[1].type())
    planes[1].convertTo(planes[1], planes[1].type(),
                        color_scale, 128*(1-color_scale));

    // alternative form of convertTo if we know the datatype
    // at compile time ("uchar" here).
    // This expression will not create any temporary arrays
    // and should be almost as fast as the above variant
    planes[2] = Mat_<uchar>(planes[2]*color_scale + 128*(1-color_scale));

    // Mat::mul replaces cvMul(). Again, no temporary arrays are
    // created in the case of simple expressions.

```

```
planes[0] = planes[0].mul(planes[0], 1./255);

// now merge the results back
merge(planes, img_yuv);
// and produce the output RGB image
cvtColor(img_yuv, img, CV_YCrCb2BGR);

// this is counterpart for cvNamedWindow
namedWindow("image with grain", CV_WINDOW_AUTOSIZE);
#if DEMO_MIXED_API_USE
// this is to demonstrate that img and iplimg really share the data
// the result of the above processing is stored to img and thus .
cvShowImage("image with grain", iplimg);
#else
imshow("image with grain", img);
#endif
waitKey();

return 0;
// all the memory will automatically be released
// by vector<>, Mat and Ptr<> destructors.
}
```

---

In the rest of the introduction we discuss the key features of the new interface in more details.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

## Namespace `cv()` and Function Naming

---

All the newly introduced classes and functions are placed into `cv()` namespace. Therefore, to access this functionality from your code, use `cv::()` specifier or `"using namespace cv;"` directive:

---

```
#include "cv.h"

...
cv::Mat H = cv::findHomography(points1, points2, cv::RANSAC,
...

```

---

or

---

```
#include "cv.h"

using namespace cv;

...
Mat H = findHomography(points1, points2, RANSAC, 5 );
...

```

---

It is probable that some of the current or future OpenCV external names conflict with STL or other libraries, in this case use explicit namespace specifiers to resolve the name conflicts:

---

```
Mat a(100, 100, CV_32F);
randu(a, Scalar::all(1), Scalar::all(std::rand()%256+1));
cv::log(a, a);
a /= std::log(2.);

```

---

For the most of the C functions and structures from OpenCV 1.x you may find the direct counterparts in the new C++ interface. The name is usually formed by omitting `cv()` or `Cv` prefix and turning the first letter to the low case (unless it's a own name, like Canny, Sobel etc). In case when there is no the new-style counterpart, it's possible to use the old functions with the new structures, as shown the first sample in the chapter.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.

- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Memory Management

---

When using the new interface, the most of memory deallocation and even memory allocation operations are done automatically when needed.

First of all, *Mat*, *SparseMat* and other classes have destructors that deallocate memory buffers occupied by the structures when needed.

Secondly, this “when needed” means that the destructors do not always deallocate the buffers, they take into account possible data sharing. That is, in a destructor the reference counter associated with the underlying data is decremented and the data is deallocated if and only if the reference counter becomes zero, that is, when no other structures refer to the same buffer. When such a structure containing a reference counter is copied, usually just the header is duplicated, while the underlying data is not; instead, the reference counter is incremented to memorize that there is another owner of the same data. Also, some structures, such as `cvMat()`, can refer to the user-allocated data. In this case the reference counter is `NULL` pointer and then no reference counting is done - the data is not deallocated by the destructors and should be deallocated manually by the user. We saw this scheme in the first example in the chapter:

---

```
// allocates IplImages and wraps it into shared pointer class
Ptr<IplImage> iplimg = cvLoadImage(...);

// constructs Mat header for IplImage data;
// does not copy the data;
// the reference counter will be NULL
Mat img(iplimg);
...
// in the end of the block img destructor is called,
// which does not try to deallocate the data because
// of NULL pointer to the reference counter.
//
// Then Ptr<IplImage> destructor is called that decrements
// the reference counter and, as the counter becomes 0 in thi
// the destructor calls cvReleaseImage().
```

---

The copying semantics was mentioned in the above paragraph, but deserves a dedicated discussion. By default, the new OpenCV structures implement shallow, so called  $O(1)$  (i.e. constant-time) assignment operations. It gives user possibility to pass quite big data structures to functions (though, e.g. passing `const Mat` is still faster than passing `cvMat()`), return them (e.g. see the example with [findHomography](#) above), store them in OpenCV and STL containers etc. - and do all of this very efficiently. On the other hand, most of the new data structures provide `clone()` method that creates a full copy of an object. Here is the sample:

---

```

// create a big 8Mb matrix
Mat A(1000, 1000, CV_64F);

// create another header for the same matrix;
// this is instant operation, regardless of the matrix size.
Mat B = A;
// create another header for the 3-rd row of A; no data is copied
Mat C = B.row(3);
// now create a separate copy of the matrix
Mat D = B.clone();
// copy the 5-th row of B to C, that is, copy the 5-th row of
B.row(5).copyTo(C);
// now let A and D share the data; after that the modified version
// of A is still referenced by B and C.
A = D;
// now make B an empty matrix (which references no memory but
// but the modified version of A will still be referenced by
// despite that C is just a single row of the original A
B.release();

// finally, make a full copy of C. In result, the big modified
// matrix will be deallocated, since it's not referenced by a
C = C.clone();

```

---

Memory management of the new data structures is automatic and thus easy. If, however, your code uses *IplImage*, *CvMat* or other C data structures a lot, memory management can still be automated without immediate migration to *Mat* by using the already mentioned template class *Ptr*, similar to *shared\_ptr* from Boost and C++ TR1. It wraps a pointer to an arbitrary object, provides transparent access to all the object fields and associates a reference counter with it. Instance of the class can be passed to any function that expects the original pointer. For correct deallocation of the object, you should specialize `Ptr<T>::delete_obj()` method. Such specialized methods already exist for the classical OpenCV structures, e.g.:

---

```

// cxoperations.hpp:
...
template<> inline Ptr<IplImage>::delete_obj() {
    cvReleaseImage(&obj);
}
...

```

---

See *Ptr* description for more details and other usage scenarios.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Memory Management Part II. Automatic Data Allocation

---

With the new interface not only explicit memory deallocation is not needed anymore, but the memory allocation is often done automatically too. That was demonstrated in the example in the beginning of the chapter when `cvtColor()` was called, and here are some more details.

*Mat* and other array classes provide method `create` that allocates a new buffer for array data if and only if the currently allocated array is not of the required size and type. If a new buffer is needed, the previously allocated buffer is released (by engaging all the reference counting mechanism described in the previous section). Now, since it is very quick to check whether the needed memory buffer is already allocated, most new OpenCV functions that have arrays as output parameters call the `create` method and this way the automatic data allocation concept is implemented. Here is the example:

---

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

using namespace cv;

int main(int, char**)
{
    VideoCapture cap(0);
    if(!cap.isOpened()) return -1;

    Mat edges;
    namedWindow("edges",1);
    for(;;)
    {
        Mat frame;
        cap >> frame;
        cvtColor(frame, edges, CV_BGR2GRAY);
        GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
        Canny(edges, edges, 0, 30, 3);
        imshow("edges", edges);
        if(waitKey(30) >= 0) break;
    }
    return 0;
}
```

---

The matrix `edges` is allocated during the first frame processing and unless the resolution will suddenly change, the same buffer will be reused for every next frame's edge map.

In many cases the output array type and size can be inferred from the input arrays' respective characteristics, but not always. In these rare cases the corresponding functions take separate input parameters that specify the data type and/or size of the output arrays, like [resize](#). Anyway, a vast majority of the new-style array processing functions call `create` for each of the output array, with just a few exceptions like `mixChannels`, `RNG::fill` and some others.

Note that this output array allocation semantic is only implemented in the new functions. If you want to pass the new structures to some old OpenCV function, you should first allocate the output arrays using `create` method, then make **CvMat** or **IplImage** headers and after that call the function.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Algebraic Operations

---

Just like in v1.x, OpenCV 2.x provides some basic functions operating on matrices, like `add`, `subtract`, `gemm` etc. In addition, it introduces overloaded operators that give the user a convenient algebraic notation, which is nearly as fast as using the functions directly. For example, here is how the least squares problem  $Ax = b$  can be solved using normal equations:

```
Mat x = (A.t()*A).inv()*A.t()*b;
```

The complete list of overloaded operators can be found in *Matrix Expressions*.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Fast Element Access

---

Historically, OpenCV provided many different ways to access image and matrix elements, and none of them was both fast and convenient. With the new data structures, OpenCV 2.x introduces a few more alternatives, hopefully more convenient than before. For detailed description of the operations, please, check *Mat* and `[MatT]bgroup({Mat_})` description. Here is part of the retro-photo-styling example rewritten (in simplified form) using the element access operations:

---

```
...
// split the image into separate color planes
vector<Mat> planes;
split(img_yuv, planes);

// method 1. process Y plane using an iterator
MatIterator_<uchar> it = planes[0].begin<uchar>(),
                    it_end = planes[0].end<uchar>();
for(; it != it_end; ++it)
{
    double v = *it*1.7 + rand()%21-10;
    *it = saturate_cast<uchar>(v*v/255.);
}

// method 2. process the first chroma plane using pre-stored
// method 3. process the second chroma plane using
            individual element access operations
for( int y = 0; y < img_yuv.rows; y++ )
{
    uchar* Uptr = planes[1].ptr<uchar>(y);
    for( int x = 0; x < img_yuv.cols; x++ )
    {
        Uptr[x] = saturate_cast<uchar>((Uptr[x]-128)/2 + 128)
        uchar& Vxy = planes[2].at<uchar>(y, x);
        Vxy = saturate_cast<uchar>((Vxy-128)/2 + 128);
    }
}

merge(planes, img_yuv);
...
```

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.

- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Saturation Arithmetics

---

In the above sample you may have noticed `[saturatecast]bgroup({saturate_ cast})` operator, and that's how all the pixel processing is done in OpenCV. When a result of image operation is 8-bit image with pixel values ranging from 0 to 255, each output pixel value is clipped to this available range:

$$I(x, y) = \min(\max(\text{value}, 0), 255)$$

and the similar rules are applied to 8-bit signed and 16-bit signed and unsigned types. This “saturation” semantics (different from usual C language “wrapping” semantics, where lowest bits are taken, is implemented in every image processing function, from the simple `cv::add()` to `cv::cvtColor()`, `cv::resize()`, `cv::filter2D()` etc. It is not a new feature of OpenCV v2.x, it was there from very beginning. In the new version this special `[saturatecast]bgroup({saturate_ cast})` template operator is introduced to simplify implementation of this semantic in your own functions.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Error handling

---

The modern error handling mechanism in OpenCV uses exceptions, as opposite to the manual stack unrolling used in previous versions. When OpenCV is built in DEBUG configuration, the error handler provokes memory access violation, so that the full call stack and context can be analyzed with debugger.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Threading and Reenterability

---

OpenCV uses OpenMP to run some time-consuming operations in parallel. Threading can be explicitly controlled by `setNumThreads` function. Also, functions and “const” methods of the classes are generally re-enterable, that is, they can be called from different threads asynchronously.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# cxcore. The Core Functionality

---

- **Basic Structures**
  - [DataType](#)
  - [Point](#)
  - [Point3](#)
  - [Size](#)
  - [Rect](#)
  - [RotatedRect](#)
  - [TermCriteria](#)
  - [Vec](#)
  - [Scalar](#)
  - [Range](#)
  - [Ptr](#)
  - [Mat](#)
  - [Matrix Expressions](#)
  - [Mat\\_](#)
  - [MatND](#)
  - [MatND\\_](#)
  - [SparseMat](#)
  - [SparseMat\\_](#)
- **Operations on Arrays**
  - [abs](#)
  - [absdiff](#)
  - [add](#)
  - [addWeighted](#)
  - [cv::bitwise\\_and](#)
  - [cv::bitwise\\_not](#)
  - [cv::bitwise\\_or](#)
  - [cv::bitwise\\_xor](#)
  - [calcCovarMatrix](#)
  - [cartToPolar](#)
  - [checkRange](#)
  - [compare](#)
  - [completeSymm](#)
  - [convertScaleAbs](#)
  - [countNonZero](#)
  - [cubeRoot](#)
  - [cvarrToMat](#)
  - [dct](#)
  - [dft](#)
  - [divide](#)
  - [determinant](#)
  - [eigen](#)
  - [exp](#)

- [extractImageCOI](#)
- [fastAtan2](#)
- [flip](#)
- [gemm](#)
- [getConvertElem](#)
- [getOptimalDFTSize](#)
- [idct](#)
- [idft](#)
- [inRange](#)
- [invert](#)
- [log](#)
- [LUT](#)
- [magnitude](#)
- [Mahalanobis](#)
- [max](#)
- [mean](#)
- [meanStdDev](#)
- [merge](#)
- [min](#)
- [minMaxLoc](#)
- [mixChannels](#)
- [mulSpectrums](#)
- [multiply](#)
- [mulTransposed](#)
- [norm](#)
- [normalize](#)
- [PCA](#)
- [perspectiveTransform](#)
- [phase](#)
- [polarToCart](#)
- [pow](#)
- [randu](#)
- [randn](#)
- [randShuffle](#)
- [reduce](#)
- [repeat](#)
- [saturate\\_cast](#)
- [scaleAdd](#)
- [setIdentity](#)
- [solve](#)
- [solveCubic](#)
- [solvePoly](#)
- [sort](#)
- [sortIdx](#)
- [split](#)
- [sqrt](#)

- [subtract](#)
- [SVD](#)
- [sum](#)
- [theRNG](#)
- [trace](#)
- [transform](#)
- [transpose](#)
- [Dynamic Structures](#)
- [Drawing Functions](#)
  - [circle](#)
  - [clipLine](#)
  - [ellipse](#)
  - [ellipse2Poly](#)
  - [fillConvexPoly](#)
  - [fillPoly](#)
  - [getTextSize](#)
  - [line](#)
  - [LineIterator](#)
  - [rectangle](#)
  - [polyLines](#)
  - [putText](#)
- [XML/YAML Persistence](#)
  - [FileStorage](#)
  - [FileNode](#)
  - [FileNodeIterator](#)
- [Clustering and Search in Multi-Dimensional Spaces](#)
  - [kmeans](#)
  - [partition](#)
  - [Fast Approximate Nearest Neighbor Search](#)
  - [flann::Index](#)
  - [flann::Index::Index](#)
  - [flann::Index::knnSearch](#)
  - [flann::Index::knnSearch](#)
  - [flann::Index::radiusSearch](#)
  - [flann::Index::radiusSearch](#)
  - [flann::Index::save](#)
  - [flann::hierarchicalClustering](#)
- [Utility and System Functions and Macros](#)
  - [alignPtr](#)
  - [alignSize](#)
  - [allocate](#)
  - [deallocate](#)
  - [CV\\_Assert](#)
  - [error](#)
  - [Exception](#)
  - [fastMalloc](#)

- [fastFree](#)
- [format](#)
- [getNumThreads](#)
- [getThreadNum](#)
- [getTickCount](#)
- [getTickFrequency](#)
- [setNumThreads](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Basic Structures

---

## DataType

---

Template “traits” class for other OpenCV primitive data types

```

template<typename _Tp> class DataType
{
    // value_type is always a synonym for _Tp.
    typedef _Tp value_type;

    // intermediate type used for operations on _Tp.
    // it is int for uchar, signed char, unsigned short, sign
    // float for float, double for double, ...
    typedef <...> work_type;
    // in the case of multi-channel data it is the data type
    typedef <...> channel_type;
    enum
    {
        // CV_8U ... CV_64F
        depth = DataDepth<channel_type>::value,
        // 1 ...
        channels = <...>,
        // '1u', '4i', '3f', '2d' etc.
        fmt=<...>,
        // CV_8UC3, CV_32FC2 ...
        type = CV_MAKETYPE(depth, channels)
    };
};

```

---

The template class `DataType` is descriptive class for OpenCV primitive data types and other types that comply with the following definition. A primitive OpenCV data type is one of unsigned char, bool ( $\sim$  unsigned char), signed char, unsigned short, signed short, int, float, double or a tuple of values of one of these types, where all the values in the tuple have the same type. If you are familiar with OpenCV `CvMat`'s type notation, CV\_8U ... CV\_32FC3, CV\_64FC2 etc., then a primitive type can be defined as a type for which you can give a unique identifier in a form CV\_<bit-depth>{U|S|F}C<number\_of\_channels>. A universal OpenCV structure able to store a single instance of such primitive data type is `Vec`. Multiple instances of such a type can be stored to a `std::vector`, `cvMat()`, `Mat_`, `MatND`, `MatND_`, `SparseMat`, `SparseMat_` or any other container that is able to store `Vec` instances.

The class `DataType` is basically used to provide some description of such primitive data types without adding any fields or methods to the corresponding classes (and it

is actually impossible to add anything to primitive C/C++ data types). This technique is known in C++ as class traits. It's not `DataType` itself that is used, but its specialized versions, such as:

---

```
template<> class DataType<uchar>
{
    typedef uchar value_type;
    typedef int work_type;
    typedef uchar channel_type;
    enum { channel_type = CV_8U, channels = 1, fmt='u', type
};
...
template<typename _Tp> DataType<std::complex<_Tp> >
{
    typedef std::complex<_Tp> value_type;
    typedef std::complex<_Tp> work_type;
    typedef _Tp channel_type;
    // DataDepth is another helper trait class
    enum { depth = DataDepth<_Tp>::value, channels=2,
          fmt=(channels-1)*256+DataDepth<_Tp>::fmt,
          type=CV_MAKETYPE(depth, channels) };
};
...
```

---

The main purpose of the classes is to convert compile-time type information to OpenCV-compatible data type identifier, for example:

---

```
// allocates 30x40 floating-point matrix
Mat A(30, 40, DataType<float>::type);

Mat B = Mat_<std::complex<double> >(3, 3);
// the statement below will print 6, 2 /* i.e. depth == CV_64
cout << B.depth() << ", " << B.channels() << endl;
```

---

that is, such traits are used to tell OpenCV which data type you are working with, even if such a type is not native to OpenCV (the matrix B initialization above compiles because OpenCV defines the proper specialized template class `DataType<complex<_Tp> >`). Also, this mechanism is useful (and used in OpenCV this way) for generic algorithms implementations.

## Point

---

Template class for 2D points

---

```
template<typename _Tp> class Point_
{
public:
```

```

typedef _Tp value_type;

Point_();
Point_(_Tp _x, _Tp _y);
Point_(const Point_& pt);
Point_(const CvPoint& pt);
Point_(const CvPoint2D32f& pt);
Point_(const Size_<_Tp>& sz);
Point_(const Vec<_Tp, 2>& v);
Point_& operator = (const Point_& pt);
template<typename _Tp2> operator Point_<_Tp2>() const;
operator CvPoint() const;
operator CvPoint2D32f() const;
operator Vec<_Tp, 2>() const;

// computes dot-product (this->x*pt.x + this->y*pt.y)
_Tp dot(const Point_& pt) const;
// computes dot-product using double-precision arithmetics
double ddot(const Point_& pt) const;
// returns true if the point is inside the rectangle "r".
bool inside(const Rect_<_Tp>& r) const;

    _Tp x, y;
};

```

The class represents a 2D point, specified by its coordinates  $x$  and  $y$ . Instance of the class is interchangeable with C structures **CvPoint** and **CvPoint2D32f**. There is also cast operator to convert point coordinates to the specified type. The conversion from floating-point coordinates to integer coordinates is done by rounding; in general case the conversion uses [saturatecast]bgroup({saturate\_cast}) operation on each of the coordinates. Besides the class members listed in the declaration above, the following operations on points are implemented:

- $pt1 = pt2 \pm pt3$
- $pt1 = \alpha * pt2$ ,  $pt1 = \alpha * pt2$
- $pt1 += pt2$ ,  $pt1 -= pt2$ ,  $pt1 *= \alpha$
- $double\ value = norm(pt)$ ; //  $L_2$ -norm
- $pt1 == pt2$ ,  $pt1 != pt2$

For user convenience, the following type aliases are defined:

```

typedef Point_<int> Point2i;
typedef Point2i Point;
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;

```

Here is a short example:

---

```
Point2f a(0.3f, 0.f), b(0.f, 0.4f);
Point pt = (a + b)*10.f;
cout << pt.x << ", " << pt.y << endl;
```

---

## Point3

---

### Template class for 3D points

---

```
template<typename _Tp> class Point3_
{
public:
    typedef _Tp value_type;

    Point3_();
    Point3_(_Tp _x, _Tp _y, _Tp _z);
    Point3_(const Point3_& pt);
    explicit Point3_(const Point_<_Tp>& pt);
    Point3_(const CvPoint3D32f& pt);
    Point3_(const Vec<_Tp, 3>& v);
    Point3_& operator = (const Point3_& pt);
    template<typename _Tp2> operator Point3_<_Tp2>() const;
    operator CvPoint3D32f() const;
    operator Vec<_Tp, 3>() const;

    _Tp dot(const Point3_& pt) const;
    double ddot(const Point3_& pt) const;

    _Tp x, y, z;
};
```

---

The class represents a 3D point, specified by its coordinates  $x$ ,  $y$  and  $z$ . Instance of the class is interchangeable with C structure **CvPoint2D32f**. Similarly to **Point\_**, the 3D points' coordinates can be converted to another type, and the vector arithmetic and comparison operations are also supported.

The following type aliases are available:

---

```
typedef Point3_<int> Point3i;
typedef Point3_<float> Point3f;
typedef Point3_<double> Point3d;
```

---

## Size

---

### Template class for specifying image or rectangle size.

---

```
template<typename _Tp> class Size_
```

```

{
public:
    typedef _Tp value_type;

    Size_();
    Size_(_Tp _width, _Tp _height);
    Size_(const Size_& sz);
    Size_(const CvSize& sz);
    Size_(const CvSize2D32f& sz);
    Size_(const Point_<_Tp>& pt);
    Size_& operator = (const Size_& sz);
    _Tp area() const;

    operator Size_<int>() const;
    operator Size_<float>() const;
    operator Size_<double>() const;
    operator CvSize() const;
    operator CvSize2D32f() const;

    _Tp width, height;
};

```

---

The class `Size_` is similar to `Point_`, except that the two members are called `width` and `height` instead of `x` and `y`. The structure can be converted to and from the old OpenCV structures `CvSize` and `CvSize2D32f`. The same set of arithmetic and comparison operations as for `Point_` is available.

OpenCV defines the following type aliases:

```

typedef Size_<int> Size2i;
typedef Size2i Size;
typedef Size_<float> Size2f;

```

---

## Rect

---

Template class for 2D rectangles

```

template<typename _Tp> class Rect_
{
public:
    typedef _Tp value_type;

    Rect_();
    Rect_(_Tp _x, _Tp _y, _Tp _width, _Tp _height);
    Rect_(const Rect_& r);
    Rect_(const CvRect& r);
    // (x, y) <- org, (width, height) <- sz
    Rect_(const Point_<_Tp>& org, const Size_<_Tp>& sz);

```

```

// (x, y) <- min(pt1, pt2), (width, height) <- max(pt1, pt2)
Rect_(const Point_<_Tp>& pt1, const Point_<_Tp>& pt2);
Rect_& operator = ( const Rect_& r );
// returns Point_<_Tp>(x, y)
Point_<_Tp> tl() const;
// returns Point_<_Tp>(x+width, y+height)
Point_<_Tp> br() const;

// returns Size_<_Tp>(width, height)
Size_<_Tp> size() const;
// returns width*height
_Tp area() const;

operator Rect_<int>() const;
operator Rect_<float>() const;
operator Rect_<double>() const;
operator CvRect() const;

// x <= pt.x && pt.x < x + width &&
// y <= pt.y && pt.y < y + height ? true : false
bool contains(const Point_<_Tp>& pt) const;

_Tp x, y, width, height;
};

```

---

The rectangle is described by the coordinates of the top-left corner (which is the default interpretation of `Rect_::x` and `Rect_::y` in OpenCV; though, in your algorithms you may count `x` and `y` from the bottom-left corner), the rectangle width and height.

Another assumption OpenCV usually makes is that the top and left boundary of the rectangle are inclusive, while the right and bottom boundaries are not, for example, the method `Rect_::contains` returns true if

`x pt.x < x+width,`

`y pt.y < y+height`

And virtually every loop over an image *ROI* in OpenCV (where ROI is specified by `Rect_<int>`) is implemented as:

---

```

for(int y = roi.y; y < roi.y + rect.height; y++)
    for(int x = roi.x; x < roi.x + rect.width; x++)
    {
        // ...
    }

```

---

In addition to the class members, the following operations on rectangles are implemented:

- `rect = rect ± point` (shifting rectangle by a certain offset)
- `rect = rect ± size` (expanding or shrinking rectangle by a certain amount)
- `rect += point`, `rect -= point`, `rect += size`, `rect -= size` (augmenting operations)
- `rect = rect1 & rect2` (rectangle intersection)
- `rect = rect1 | rect2` (minimum area rectangle containing `rect1` and `rect2`)
- `rect = rect1`, `rect |= rect1` (and the corresponding augmenting operations)
- `rect == rect1`, `rect != rect1` (rectangle comparison)

Example. Here is how the partial ordering on rectangles can be established (`rect1 ⊆ rect2`):

---

```
template<typename _Tp> inline bool
operator <= (const Rect_<_Tp>& r1, const Rect_<_Tp>& r2)
{
    return (r1 & r2) == r1;
}
```

---

For user convenience, the following type alias is available:

---

```
typedef Rect_<int> Rect;
```

---

## RotatedRect

---

Possibly rotated rectangle

---

```
class RotatedRect
{
public:
    // constructors
    RotatedRect();
    RotatedRect(const Point2f& _center, const Size2f& _size,
                RotatedRect(const CvBox2D& box);

    // returns minimal up-right rectangle that contains the r
    Rect boundingRect() const;
    // backward conversion to CvBox2D
    operator CvBox2D() const;

    // mass center of the rectangle
    Point2f center;
    // size
    Size2f size;
    // rotation angle in degrees
    float angle;
```

---

```
};
```

The class `RotatedRect` replaces the old `CvBox2D` and fully compatible with it.

## TermCriteria

---

Termination criteria for iterative algorithms

---

```
class TermCriteria
{
public:
    enum { COUNT=1, MAX_ITER=COUNT, EPS=2 };

    // constructors
    TermCriteria();
    // type can be MAX_ITER, EPS or MAX_ITER+EPS.
    // type = MAX_ITER means that only the number of iteratic
    // type = EPS means that only the required precision (eps
    // (though, most algorithms put some limit on the numb
    // type = MAX_ITER + EPS means that algorithm stops when
    // either the specified number of iterations is made,
    // or when the specified accuracy is achieved - whatever
    TermCriteria(int _type, int _maxCount, double _epsilon);
    TermCriteria(const CvTermCriteria& criteria);
    operator CvTermCriteria() const;

    int type;
    int maxCount;
    double epsilon;
};
```

---

The class `TermCriteria` replaces the old `CvTermCriteria` and fully compatible with it.

## Vec

---

Template class for short numerical vectors

---

```
template<typename _Tp, int cn> class Vec
{
public:
    typedef _Tp value_type;
    enum { depth = DataDepth<_Tp>::value, channels = cn,
          type = CV_MAKETYPE(depth, channels) };

    // default constructor: all elements are set to 0
    Vec();
```

```

// constructors taking up to 10 first elements as parameters
Vec(_Tp v0);
Vec(_Tp v0, _Tp v1);
Vec(_Tp v0, _Tp v1, _Tp v2);
...
Vec(_Tp v0, _Tp v1, _Tp v2, _Tp v3, _Tp v4,
    _Tp v5, _Tp v6, _Tp v7, _Tp v8, _Tp v9);
Vec(const Vec<_Tp, cn>& v);
// constructs vector with all the components set to alpha.
static Vec all(_Tp alpha);

// two variants of dot-product
_Tp dot(const Vec& v) const;
double ddot(const Vec& v) const;

// cross-product; valid only when cn == 3.
Vec cross(const Vec& v) const;

// element type conversion
template<typename T2> operator Vec<T2, cn>() const;

// conversion to/from CvScalar (valid only when cn==4)
operator CvScalar() const;

// element access
_Tp operator [](int i) const;
_Tp& operator [](int i);

_Tp val[cn];
};

```

The class is the most universal representation of short numerical vectors or tuples. It is possible to convert `Vec<T, 2>` to/from `Point_`, `Vec<T, 3>` to/from `Point3_`, and `Vec<T, 4>` to `CvScalar`. The elements of `Vec` are accessed using `operator[]`. All the expected vector operations are implemented too:

- $v1 = v2 \pm v3$ ,  $v1 = v2 * \alpha$ ,  $v1 = \alpha * v2$  (plus the corresponding augmenting operations; note that these operations apply `saturate_cast` to the each computed vector component)
- $v1 == v2$ ,  $v1 != v2$
- `double n = norm(v1);` //  $L_2$ -norm

For user convenience, the following type aliases are introduced:

```

typedef Vec<uchar, 2> Vec2b;
typedef Vec<uchar, 3> Vec3b;
typedef Vec<uchar, 4> Vec4b;

```

```

typedef Vec<short, 2> Vec2s;
typedef Vec<short, 3> Vec3s;
typedef Vec<short, 4> Vec4s;

typedef Vec<int, 2> Vec2i;
typedef Vec<int, 3> Vec3i;
typedef Vec<int, 4> Vec4i;

typedef Vec<float, 2> Vec2f;
typedef Vec<float, 3> Vec3f;
typedef Vec<float, 4> Vec4f;
typedef Vec<float, 6> Vec6f;

typedef Vec<double, 2> Vec2d;
typedef Vec<double, 3> Vec3d;
typedef Vec<double, 4> Vec4d;
typedef Vec<double, 6> Vec6d;

```

---

The class `Vec` can be used for declaring various numerical objects, e.g. `Vec<double, 9>` can be used to store a 3x3 double-precision matrix. It is also very useful for declaring and processing multi-channel arrays, see `Mat_` description.

## Scalar

---

### 4-element vector

```

template<typename _Tp> class Scalar_ : public Vec<_Tp, 4>
{
public:
    Scalar_();
    Scalar_(_Tp v0, _Tp v1, _Tp v2=0, _Tp v3=0);
    Scalar_(const CvScalar& s);
    Scalar_(_Tp v0);
    static Scalar_<_Tp> all(_Tp v0);
    operator CvScalar() const;

    template<typename T2> operator Scalar_<T2>() const;

    Scalar_<_Tp> mul(const Scalar_<_Tp>& t, double scale=1 )
    template<typename T2> void convertTo(T2* buf, int channel
};

typedef Scalar_<double> Scalar;

```

---

The template class `Scalar_` and its double-precision instantiation `Scalar` represent 4-element vector. Being derived from `Vec<_Tp, 4>`, they can be used as typical 4-element vectors, but in addition they can be converted to/from

**CvScalar**. The type `Scalar` is widely used in OpenCV for passing pixel values and it is a drop-in replacement for `CvScalar` that was used for the same purpose in the earlier versions of OpenCV.

## Range

---

Specifies a continuous subsequence (a.k.a. slice) of a sequence.

---

```
class Range
{
public:
    Range();
    Range(int _start, int _end);
    Range(const CvSlice& slice);
    int size() const;
    bool empty() const;
    static Range all();
    operator CvSlice() const;

    int start, end;
};
```

---

The class is used to specify a row or column span in a matrix (*Mat*), and for many other purposes. `Range(a, b)` is basically the same as `a:b` in Matlab or `a..b` in Python. As in Python, `start` is inclusive left boundary of the range, and `end` is exclusive right boundary of the range. Such a half-opened interval is usually denoted as  $[start, end)$ .

The static method `Range::all()` returns some special variable that means “the whole sequence” or “the whole range”, just like “:” in Matlab or “...” in Python. All the methods and functions in OpenCV that take `Range` support this special `Range::all()` value, but of course, in the case of your own custom processing you will probably have to check and handle it explicitly:

---

```
void my_function(..., const Range& r, ....)
{
    if(r == Range::all()) {
        // process all the data
    }
    else {
        // process [r.start, r.end)
    }
}
```

---

## Ptr

---

## A template class for smart reference-counting pointers

---

```

template<typename _Tp> class Ptr
{
public:
    // default constructor
    Ptr();
    // constructor that wraps the object pointer
    Ptr(_Tp* _obj);
    // destructor: calls release()
    ~Ptr();
    // copy constructor; increments ptr's reference counter
    Ptr(const Ptr& ptr);
    // assignment operator; decrements own reference counter
    // (with release()) and increments ptr's reference counte
    Ptr& operator = (const Ptr& ptr);
    // increments reference counter
    void addref();
    // decrements reference counter; when it becomes 0,
    // delete_obj() is called
    void release();
    // user-specified custom object deletion operation.
    // by default, "delete obj;" is called
    void delete_obj();
    // returns true if obj == 0;
    bool empty() const;

    // provide access to the object fields and methods
    _Tp* operator -> ();
    const _Tp* operator -> () const;

    // return the underlying object pointer;
    // thanks to the methods, the Ptr<_Tp> can be
    // used instead of _Tp*
    operator _Tp* ();
    operator const _Tp* () const;
protected:
    // the encapsulated object pointer
    _Tp* obj;
    // the associated reference counter
    int* refcount;
};

```

---

The class `Ptr<_Tp>` is a template class that wraps pointers of the corresponding type. It is similar to `shared_ptr` that is a part of Boost library ([http://www.boost.org/doc/libs/1\\_40\\_0/libs/smart\\_ptr/shared\\_ptr.htm](http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/shared_ptr.htm)) and also a part of the `bgroup` (<http://en.wikipedia.org/wiki/C>

By using this class you can get the following capabilities:

begin{itemize} item default constructor, copy constructor and assignment operator for an arbitrary C++ class or a C structure. For some objects, like files, windows, mutexes, sockets etc, copy constructor or assignment operator are difficult to define. For some other objects, like complex classifiers in OpenCV, copy constructors are absent and not easy to implement. Finally, some of complex OpenCV and your own data structures may have been written in C. However, copy constructors and default constructors can simplify programming a lot; besides, they are often required (e.g. by STL containers). By wrapping a pointer to such a complex object `texttt{TObj}` to `texttt{Ptr<TObj>}` you will automatically get all of the necessary constructors and the assignment operator.

item all the above-mentioned operations running very fast, regardless of the data size, i.e. as “O(1)” operations. Indeed, while some structures, like `texttt{std::vector}` provide a copy constructor and an assignment operator, the operations may take considerable time if the data structures are big. But if the structures are put into `texttt{Ptr<>}`, the overhead becomes small and independent of the data size.

item automatic destruction, even for C structures. See the example below with `texttt{FILE*}`.

item heterogeneous collections of objects. The standard STL and most other C++ and OpenCV containers can only store objects of the same type and the same size. The classical solution to store objects of different types in the same container is to store pointers to the base class `texttt{base_ class_ t*}` instead, but when you loose the automatic memory management. Again, by using `texttt{Ptr<base_ class_ t>()}` instead of the raw pointers, you can solve the problem.

end{itemize}

The class `texttt{Ptr}` treats the wrapped object as a black box, the reference counter is allocated and managed separately. The only thing the pointer class needs to know about the object is how to deallocate it. This knowledge is encapsulated in `texttt{Ptr::delete_ obj()}` method, which is called when the reference counter becomes 0. If the object is a C++ class instance, no additional coding is needed, because the default implementation of this method calls `texttt{delete obj;}`. However, if the object is deallocated in a different way, then the specialized method should be created. For example, if you want to wrap `texttt{FILE}`, the `texttt{delete_ obj}` may be implemented as following:

```
begin{lstlisting} template<> inline void Ptr<FILE>::delete_obj() {
    fclose(obj); // no need to clear the pointer afterwards,
                // it is done externally.
}
```

```
// now use it: Ptr<FILE> f(fopen("myfile.txt", "r")); if(f.empty())
```

```
    throw ...;
```

```
fprintf(f, ...); ... // the file will be closed automatically by the Ptr<FILE> destructor. end
{!stlisting}
```

textbf{Note}: The reference increment/decrement operations are implemented as atomic operations, and therefore it is normally safe to use the classes in multi-threaded applications. The same is true for cross{Mat} and other C++ OpenCV classes that operate on the reference counters.

```
)
```

## Mat

---

OpenCV C++ matrix class.

```
class Mat
{
public:
    // constructors
    Mat();
    // constructs matrix of the specified size and type
    // (_type is CV_8UC1, CV_64FC3, CV_32SC(12) etc.)
    Mat(int _rows, int _cols, int _type);
    // constucts matrix and fills it with the specified value
    Mat(int _rows, int _cols, int _type, const Scalar& _s);
    Mat(Size _size, int _type);
    // copy constructor
    Mat(const Mat& m);
    // constructor for matrix headers pointing to user-allocat
    Mat(int _rows, int _cols, int _type, void* _data, size_t
    Mat(Size _size, int _type, void* _data, size_t _step=AUTO
    // creates a matrix header for a part of the bigger matri
    Mat(const Mat& m, const Range& rowRange, const Range& col
    Mat(const Mat& m, const Rect& roi);
    // converts old-style CvMat to the new matrix; the data i
    Mat(const CvMat* m, bool copyData=false);
    // converts old-style IplImage to the new matrix; the dat
    Mat(const IplImage* img, bool copyData=false);
    // builds matrix from std::vector with or without copying
    template<typename _Tp> Mat(const vector<_Tp>& vec, bool c
    // helper constructor to compile matrix expressions
    Mat(const MatExpr_Base& expr);
    // destructor - calls release()
    ~Mat();
    // assignment operators
    Mat& operator = (const Mat& m);
```

```

Mat& operator = (const MatExpr_Base& expr);

...
// returns a new matrix header for the specified row
Mat row(int y) const;
// returns a new matrix header for the specified column
Mat col(int x) const;
// ... for the specified row span
Mat rowRange(int startrow, int endrow) const;
Mat rowRange(const Range& r) const;
// ... for the specified column span
Mat colRange(int startcol, int endcol) const;
Mat colRange(const Range& r) const;
// ... for the specified diagonal
// (d=0 - the main diagonal,
// >0 - a diagonal from the lower half,
// <0 - a diagonal from the upper half)
Mat diag(int d=0) const;
// constructs a square diagonal matrix which main diagonal is vector d
static Mat diag(const Mat& d);

// returns deep copy of the matrix, i.e. the data is copied
Mat clone() const;
// copies the matrix content to "m".
// It calls m.create(this->size(), this->type()).
void copyTo( Mat& m ) const;
// copies those matrix elements to "m" that are marked with non-zero elements in mask
void copyTo( Mat& m, const Mat& mask ) const;
// converts matrix to another datatype with optional scaling. See cvRound
void convertTo( Mat& m, int rtype, double alpha=1, double beta=0) const;

...
// sets every matrix element to s
Mat& operator = (const Scalar& s);
// sets some of the matrix elements to s, according to the mask
Mat& setTo(const Scalar& s, const Mat& mask=Mat());
// creates alternative matrix header for the same data, with different
// number of channels and/or different number of rows. see cvReshape
Mat reshape(int _cn, int _rows=0) const;

// matrix transposition by means of matrix expressions
MatExpr_<...> t() const;
// matrix inversion by means of matrix expressions
MatExpr_<...> inv(int method=DECOMP_LU) const;
// per-element matrix multiplication by means of matrix expressions
MatExpr_<...> mul(const Mat& m, double scale=1) const;
MatExpr_<...> mul(const MatExpr_<...>& m, double scale=1) const;

// computes cross-product of 2 3D vectors
Mat cross(const Mat& m) const;
// computes dot-product

```

```

double dot(const Mat& m) const;

// Matlab-style matrix initialization. see the description
static MatExpr_Initializer zeros(int rows, int cols, int
static MatExpr_Initializer zeros(Size size, int type);
static MatExpr_Initializer ones(int rows, int cols, int type
static MatExpr_Initializer ones(Size size, int type);
static MatExpr_Initializer eye(int rows, int cols, int type
static MatExpr_Initializer eye(Size size, int type);

// allocates new matrix data unless the matrix already has speci
// previous data is unreferenced if needed.
void create(int _rows, int _cols, int _type);
void create(Size _size, int _type);
// increases the reference counter; use with care to avoid memle
void addref();
// decreases reference counter;
// deallocate the data when reference counter reaches 0.
void release();

// locates matrix header within a parent matrix. See below
void locateROI( Size& wholeSize, Point& ofs ) const;
// moves/resizes the current matrix ROI inside the parent matrix
Mat& adjustROI( int dtop, int dbottom, int dleft, int dright
// extracts a rectangular sub-matrix
// (this is a generalized form of row, rowRange etc.)
Mat operator()( Range rowRange, Range colRange ) const;
Mat operator()( const Rect& roi ) const;

// converts header to CvMat; no data is copied
operator CvMat() const;
// converts header to IplImage; no data is copied
operator IplImage() const;

// returns true iff the matrix data is continuous
// (i.e. when there are no gaps between successive rows).
// similar to CV_IS_MAT_CONT(cvmat->type)
bool isContinuous() const;
// returns element size in bytes,
// similar to CV_ELEM_SIZE(cvmat->type)
size_t elemSize() const;
// returns the size of element channel in bytes.
size_t elemSize1() const;
// returns element type, similar to CV_MAT_TYPE(cvmat->type)
int type() const;
// returns element type, similar to CV_MAT_DEPTH(cvmat->type)
int depth() const;
// returns element type, similar to CV_MAT_CN(cvmat->type)
int channels() const;
// returns step/elemSize1()
size_t step1() const;

```

```

// returns matrix size:
// width == number of columns, height == number of rows
Size size() const;
// returns true if matrix data is NULL
bool empty() const;

// returns pointer to y-th row
uchar* ptr(int y=0);
const uchar* ptr(int y=0) const;

// template version of the above method
template<typename _Tp> _Tp* ptr(int y=0);
template<typename _Tp> const _Tp* ptr(int y=0) const;

// template methods for read-write or read-only element access.
// note that _Tp must match the actual matrix type -
// the functions do not do any on-fly type conversion
template<typename _Tp> _Tp& at(int y, int x);
template<typename _Tp> _Tp& at(Point pt);
template<typename _Tp> const _Tp& at(int y, int x) const;
template<typename _Tp> const _Tp& at(Point pt) const;

// template methods for iteration over matrix elements.
// the iterators take care of skipping gaps in the end of rows (
template<typename _Tp> MatIterator_<_Tp> begin();
template<typename _Tp> MatIterator_<_Tp> end();
template<typename _Tp> MatConstIterator_<_Tp> begin() const
template<typename _Tp> MatConstIterator_<_Tp> end() const

enum { MAGIC_VAL=0x42FF0000, AUTO_STEP=0, CONTINUOUS_FLAG

// includes several bit-fields:
// * the magic signature
// * continuity flag
// * depth
// * number of channels
int flags;
// the number of rows and columns
int rows, cols;
// a distance between successive rows in bytes; includes the gap
size_t step;
// pointer to the data
uchar* data;

// pointer to the reference counter;
// when matrix points to user-allocated data, the pointer is NULL.
int* refcount;

// helper fields used in locateROI and adjustROI
uchar* datastart;
uchar* dataend;

```

```
};
```

---

The class `cvMat()` represents a 2D numerical array that can act as a matrix (and further it's referred to as a matrix), image, optical flow map etc. It is very similar to `CvMat` type from earlier versions of OpenCV, and similarly to `CvMat`, the matrix can be multi-channel, but it also fully supports *ROI* mechanism, just like *IplImage*.

There are many different ways to create `cvMat()` object. Here are the some popular ones:

- using `create(nrows, ncols, type)` method or the similar constructor `Mat(nrows, ncols, type[, fill_value])` constructor. A new matrix of the specified size and specified type will be allocated. `type` has the same meaning as in `cvCreateMat` method, e.g. `CV_8UC1` means 8-bit single-channel matrix, `CV_32FC2` means 2-channel (i.e. complex) floating-point matrix etc:

---

```
// make 7x7 complex matrix filled with 1+3j.
cv::Mat M(7,7,CV_32FC2,Scalar(1,3));
// and now turn M to 100x60 15-channel 8-bit matrix.
// The old content will be deallocated
M.create(100,60,CV_8UC(15));
```

---

As noted in the introduction of this chapter, `create()` will only allocate a new matrix when the current matrix dimensionality or type are different from the specified.

- by using a copy constructor or assignment operator, where on the right side it can be a matrix or expression, see below. Again, as noted in the introduction, matrix assignment is  $O(1)$  operation because it only copies the header and increases the reference counter. `Mat::clone()` method can be used to get a full (a.k.a. deep) copy of the matrix when you need it.
- by constructing a header for a part of another matrix. It can be a single row, single column, several rows, several columns, rectangular region in the matrix (called a minor in algebra) or a diagonal. Such operations are also  $O(1)$ , because the new header will reference the same data. You can actually modify a part of the matrix using this feature, e.g.

---

```
// add 5-th row, multiplied by 3 to the 3rd row
M.row(3) = M.row(3) + M.row(5)*3;

// now copy 7-th column to the 1-st column
// M.col(1) = M.col(7); // this will not work
Mat M1 = M.col(1);
M.col(7).copyTo(M1);

// create new 320x240 image
```

```

cv::Mat img(Size(320,240),CV_8UC3);
// select a roi
cv::Mat roi(img, Rect(10,10,100,100));
// fill the ROI with (0,255,0) (which is green in RGB space);
// the original 320x240 image will be modified
roi = Scalar(0,255,0);

```

---

Thanks to the additional `datastart` and `dataend` members, it is possible to compute the relative sub-matrix position in the main “container” matrix using `locateROI()`:

```

Mat A = Mat::eye(10, 10, CV_32S);
// extracts A columns, 1 (inclusive) to 3 (exclusive).
Mat B = A(Range::all(), Range(1, 3));
// extracts B rows, 5 (inclusive) to 9 (exclusive).
// that is, C ~ A(Range(5, 9), Range(1, 3))
Mat C = B(Range(5, 9), Range::all());
Size size; Point ofs;
C.locateROI(size, ofs);
// size will be (width=10,height=10) and the ofs will be (x=1

```

---

As in the case of whole matrices, if you need a deep copy, use `clone()` method of the extracted sub-matrices.

- by making a header for user-allocated-data. It can be useful for
- processing “foreign” data using OpenCV (e.g. when you implement a `DirectShow` filter or a processing module for `gstreamer` etc.), e.g.

```

void process_video_frame(const unsigned char* pixels,
                        int width, int height, int step)
{
    cv::Mat img(height, width, CV_8UC3, pixels, step);
    cv::GaussianBlur(img, img, cv::Size(7,7), 1.5, 1.5);
}

```

---

- for quick initialization of small matrices and/or super-fast element access

```

double m[3][3] = {{a, b, c}, {d, e, f}, {g, h, i}};
cv::Mat M = cv::Mat(3, 3, CV_64F, m).inv();

```

---

partial yet very common cases of this “user-allocated data” case are conversions from `CvMat` and `IplImage` to `cvMat()`. For this purpose there are special constructors taking pointers to `CvMat` or `IplImage` and the optional flag indicating whether to copy the data or not.

Backward conversion from `cvMat()` to `CvMat` or `IplImage` is provided via

cast operators `Mat::operator CvMat()` const and `Mat::operator IplImage()`. The operators do not copy the data.

---

```
IplImage* img = cvLoadImage("greatwave.jpg", 1);
Mat mtx(img); // convert IplImage* -> cv::Mat
CvMat oldmat = mtx; // convert cv::Mat -> CvMat
CV_Assert(oldmat.cols == img->width && oldmat.rows == img->he
          oldmat.data.ptr == (uchar*)img->imageData && oldmat.step
```

---

- by using MATLAB-style matrix initializers, `zeros()`, `ones()`, `eye()`, e.g.:

---

```
// create a double-precision identity matrix and add it to M.
M += Mat::eye(M.rows, M.cols, CV_64F);
```

---

- by using comma-separated initializer:

---

```
// create 3x3 double-precision identity matrix
Mat M = (Mat_<double>(3,3) << 1, 0, 0, 0, 1, 0, 0, 0, 1);
```

---

here we first call constructor of `Mat_` class (that we describe further) with the proper matrix, and then we just put `<<` operator followed by comma-separated values that can be constants, variables, expressions etc. Also, note the extra parentheses that are needed to avoid compiler errors.

Once matrix is created, it will be automatically managed by using reference-counting mechanism (unless the matrix header is built on top of user-allocated data, in which case you should handle the data by yourself). The matrix data will be deallocated when no one points to it; if you want to release the data pointed by a matrix header before the matrix destructor is called, use `Mat::release()`.

The next important thing to learn about the matrix class is element access. Here is how the matrix is stored. The elements are stored in row-major order (row by row). The `Mat::data` member points to the first element of the first row, `Mat::rows` contains the number of matrix rows and `Mat::cols` the number of matrix columns. There is yet another member, called `Mat::step` that is used to actually compute address of a matrix element. The `Mat::step` is needed because the matrix can be a part of another matrix or because there can some padding space in the end of each row for a proper alignment.

Given these parameters, address of the matrix element  $M_{ij}$  is computed as following:

$$\text{addr}(\text{:math:}`\$M_{ij}``) = \text{M.data} + \text{M.step} * i + j * \text{M.elemSize}()$$

if you know the matrix element type, e.g. it is `float`, then you can use `at<>()`

method:

```
addr(:math:`$M_{ij}$`) = M.at<float>(i, j)
```

(where `is` is used to convert the reference returned by `at` to a pointer). if you need to process a whole row of matrix, the most efficient way is to get the pointer to the row first, and then just use plain C operator `[]`:

---

```
// compute sum of positive matrix elements
// (assuming that M is double-precision matrix)
double sum=0;
for(int i = 0; i < M.rows; i++)
{
    const double* Mi = M.ptr<double>(i);
    for(int j = 0; j < M.cols; j++)
        sum += std::max(Mi[j], 0.);
}

```

---

Some operations, like the above one, do not actually depend on the matrix shape, they just process elements of a matrix one by one (or elements from multiple matrices that are sitting in the same place, e.g. matrix addition). Such operations are called element-wise and it makes sense to check whether all the input/output matrices are continuous, i.e. have no gaps in the end of each row, and if yes, process them as a single long row:

---

```
// compute sum of positive matrix elements, optimized variant
double sum=0;
int cols = M.cols, rows = M.rows;
if(M.isContinuous())
{
    cols *= rows;
    rows = 1;
}
for(int i = 0; i < rows; i++)
{
    const double* Mi = M.ptr<double>(i);
    for(int j = 0; j < cols; j++)
        sum += std::max(Mi[j], 0.);
}

```

---

in the case of continuous matrix the outer loop body will be executed just once, so the overhead will be smaller, which will be especially noticeable in the case of small matrices.

Finally, there are STL-style iterators that are smart enough to skip gaps between successive rows:

---

```
// compute sum of positive matrix elements, iterator-based va
```

---

```
double sum=0;
MatConstIterator_<double> it = M.begin<double>(), it_end = M.
for(; it != it_end; ++it)
    sum += std::max(*it, 0.);
```

---

The matrix iterators are random-access iterators, so they can be passed to any STL algorithm, including `std::sort()`.

## Matrix Expressions

---

This is a list of implemented matrix operations that can be combined in arbitrary complex expressions (here  $A$ ,  $B$  stand for matrices (`cvMat()`),  $s$  for a scalar (Scalar),  $\alpha$  for a real-valued scalar (double)):

- addition, subtraction, negation:  $A \pm B$ ,  $A \pm s$ ,  $s \pm A$ ,  $-A$
- scaling:  $A * \alpha$ ,  $A / \alpha$
- per-element multiplication and division:  $A.mul(B)$ ,  $A/B$ ,  $A / \alpha$
- matrix multiplication:  $A * B$
- transposition:  $A.t()$   $\sim A^t$
- matrix inversion and pseudo-inversion, solving linear systems and least-squares problems:  $A.inv([method])$   $\sim A^{-1}$ ,  $A.inv([method]) * B$   $\sim X$ ,  $AX=B$
- comparison:  $A \geq B$ ,  $A \neq B$ ,  $A \geq \alpha$ ,  $A \neq \alpha$ . The result of comparison is 8-bit single channel mask, which elements are set to 255 (if the particular element or pair of elements satisfy the condition) and 0 otherwise.
- bitwise logical operations:  $A \& B$ ,  $A \& s$ ,  $A | B$ ,  $A | s$ ,  $A \wedge B$ ,  $A \wedge s$ ,  $\sim A$
- element-wise minimum and maximum:  $\min(A, B)$ ,  $\min(A, \alpha)$ ,  $\max(A, B)$ ,  $\max(A, \alpha)$
- element-wise absolute value:  $abs(A)$
- cross-product, dot-product:  $A.cross(B)$ ,  $A.dot(B)$
- any function of matrix or matrices and scalars that returns a matrix or a scalar, such as [norm](#), [mean](#), [sum](#), [countNonZero](#), [trace](#), [determinant](#), [repeat](#) etc.
- matrix initializers (`eye()`, `zeros()`, `ones()`), matrix comma-separated initializers, matrix constructors and operators that extract sub-matrices (see *Mat* description).
- `Mat_<destination_type>()` constructors to cast the result to the proper type.

Note, however, that comma-separated initializers and probably some other operations may require additional explicit `Mat()` or `Mat_<T>()` constructor calls to resolve possible ambiguity.

## Mat\_

---

## Template matrix class derived from *Mat*

---

```

template<typename _Tp> class Mat_ : public Mat
{
public:
    typedef _Tp value_type;
    typedef typename DataType<_Tp>::channel_type channel_type
    typedef MatIterator_<_Tp> iterator;
    typedef MatConstIterator_<_Tp> const_iterator;

    Mat_();
    // equivalent to Mat(_rows, _cols, DataType<_Tp>::type)
    Mat_(int _rows, int _cols);
    // other forms of the above constructor
    Mat_(int _rows, int _cols, const _Tp& value);
    explicit Mat_(Size _size);
    Mat_(Size _size, const _Tp& value);
    // copy/conversion constructor. If m is of different type,
    Mat_(const Mat& m);
    // copy constructor
    Mat_(const Mat_& m);
    // construct a matrix on top of user-allocated data.
    // step is in bytes(!!!), regardless of the type
    Mat_(int _rows, int _cols, _Tp* _data, size_t _step=AUTO_
    // minor selection
    Mat_(const Mat_& m, const Range& rowRange, const Range& c
    Mat_(const Mat_& m, const Rect& roi);
    // to support complex matrix expressions
    Mat_(const MatExpr_Base& expr);
    // makes a matrix out of Vec or std::vector. The matrix w
    template<int n> explicit Mat_(const Vec<_Tp, n>& vec);
    Mat_(const vector<_Tp>& vec, bool copyData=false);

    Mat_& operator = (const Mat& m);
    Mat_& operator = (const Mat_& m);
    // set all the elements to s.
    Mat_& operator = (const _Tp& s);

    // iterators; they are smart enough to skip gaps in the e
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    // equivalent to Mat::create(_rows, _cols, DataType<_Tp>:
    void create(int _rows, int _cols);
    void create(Size _size);
    // cross-product
    Mat_ cross(const Mat_& m) const;
    // to support complex matrix expressions
    Mat_& operator = (const MatExpr_Base& expr);

```

```

// data type conversion
template<typename T2> operator Mat_<T2>() const;
// overridden forms of Mat::row() etc.
Mat_ row(int y) const;
Mat_ col(int x) const;
Mat_ diag(int d=0) const;
Mat_ clone() const;

// transposition, inversion, per-element multiplication
MatExpr_<...> t() const;
MatExpr_<...> inv(int method=DECOMP_LU) const;

MatExpr_<...> mul(const Mat_& m, double scale=1) const;
MatExpr_<...> mul(const MatExpr_<...>& m, double scale=1)

// overridden forms of Mat::elemSize() etc.
size_t elemSize() const;
size_t elemSize1() const;
int type() const;
int depth() const;
int channels() const;
size_t step1() const;
// returns step()/sizeof(_Tp)
size_t stepT() const;

// overridden forms of Mat::zeros() etc. Data type is omitted, o
static MatExpr_Initializer zeros(int rows, int cols);
static MatExpr_Initializer zeros(Size size);
static MatExpr_Initializer ones(int rows, int cols);
static MatExpr_Initializer ones(Size size);
static MatExpr_Initializer eye(int rows, int cols);
static MatExpr_Initializer eye(Size size);

// some more overridden methods
Mat_ reshape(int _rows) const;
Mat_ & adjustROI( int dtop, int dbottom, int dleft, int dright
Mat_ operator()( const Range& rowRange, const Range& colRange
Mat_ operator()( const Rect& roi ) const;

// more convenient forms of row and element access operators
_Tp* operator [] (int y);
const _Tp* operator [] (int y) const;

_Tp& operator () (int row, int col);
const _Tp& operator () (int row, int col) const;
_Tp& operator () (Point pt);
const _Tp& operator () (Point pt) const;

// to support matrix expressions
operator MatExpr_<Mat_, Mat_>() const;

```

---

```

    // conversion to vector.
    operator vector<_Tp>() const;
};

```

---

The class `Mat_<_Tp>` is a “thin” template wrapper on top of `cvMat()` class. It does not have any extra data fields, nor it or `cvMat()` have any virtual methods and thus references or pointers to these two classes can be freely converted one to another. But do it with care, e.g.:

---

```

// create 100x100 8-bit matrix
Mat M(100,100,CV_8U);
// this will compile fine. no any data conversion will be don
Mat_<float>& M1 = (Mat_<float>&)M;
// the program will likely crash at the statement below
M1(99,99) = 1.f;

```

---

While `cvMat()` is sufficient in most cases, `Mat_` can be more convenient if you use a lot of element access operations and if you know matrix type at compile time. Note that `Mat::at<_Tp>(int y, int x)` and `Mat_<_Tp>::operator()(int y, int x)` do absolutely the same and run at the same speed, but the latter is certainly shorter:

---

```

Mat_<double> M(20,20);
for(int i = 0; i < M.rows; i++)
    for(int j = 0; j < M.cols; j++)
        M(i,j) = 1./(i+j+1);
Mat E, V;
eigen(M,E,V);
cout << E.at<double>(0,0)/E.at<double>(M.rows-1,0);

```

---

How to use `Mat_` for multi-channel images/matrices?

This is simple - just pass `Vec` as `Mat_` parameter:

---

```

// allocate 320x240 color image and fill it with green (in RG
Mat_<Vec3b> img(240, 320, Vec3b(0,255,0));
// now draw a diagonal white line
for(int i = 0; i < 100; i++)
    img(i,i)=Vec3b(255,255,255);
// and now scramble the 2nd (red) channel of each pixel
for(int i = 0; i < img.rows; i++)
    for(int j = 0; j < img.cols; j++)
        img(i,j)[2] ^= (uchar)(i ^ j);

```

---

## MatND

---

## n-dimensional dense array

---

```

class MatND
{
public:
    // default constructor
    MatND();
    // constructs array with specific size and data type
    MatND(int _ndims, const int* _sizes, int _type);
    // constructs array and fills it with the specified value
    MatND(int _ndims, const int* _sizes, int _type, const Sca
    // copy constructor. only the header is copied.
    MatND(const MatND& m);
    // sub-array selection. only the header is copied
    MatND(const MatND& m, const Range* ranges);
    // converts old-style nd array to MatND; optionally, copi
    MatND(const CvMatND* m, bool copyData=false);
    ~MatND();
    MatND& operator = (const MatND& m);

    // creates a complete copy of the matrix (all the data is
    MatND clone() const;
    // sub-array selection; only the header is copied
    MatND operator()(const Range* ranges) const;

    // copies the data to another matrix.
    // Calls m.create(this->size(), this->type()) prior to
    // copying the data
    void copyTo( MatND& m ) const;
    // copies only the selected elements to another matrix.
    void copyTo( MatND& m, const MatND& mask ) const;
    // converts data to the specified data type.
    // calls m.create(this->size(), rtype) prior to the conve
    void convertTo( MatND& m, int rtype, double alpha=1, doub

    // assigns "s" to each array element.
    MatND& operator = (const Scalar& s);
    // assigns "s" to the selected elements of array
    // (or to all the elements if mask==MatND())
    MatND& setTo(const Scalar& s, const MatND& mask=MatND());
    // modifies geometry of array without copying the data
    MatND reshape(int _newcn, int _newndims=0, const int* _ne

    // allocates a new buffer for the data unless the current
    // has the specified size and type.
    void create(int _ndims, const int* _sizes, int _type);
    // manually increment reference counter (use with care !!
    void addref();
    // decrements the reference counter. Deallocates the data
    // the reference counter reaches zero.
    void release();

```

```

// converts the matrix to 2D Mat or to the old-style CvMatND.
// In either case the data is not copied.
operator Mat() const;
operator CvMatND() const;
// returns true if the array data is stored continuously
bool isContinuous() const;
// returns size of each element in bytes
size_t elemSize() const;
// returns size of each element channel in bytes
size_t elemSize1() const;
// returns OpenCV data type id (CV_8UC1, ... CV_64FC4,...)
int type() const;
// returns depth (CV_8U ... CV_64F)
int depth() const;
// returns the number of channels
int channels() const;
// step1() ~ step()/elemSize1()
size_t step1(int i) const;

// return pointer to the element (versions for 1D, 2D, 3D and general)
uchar* ptr(int i0);
const uchar* ptr(int i0) const;
uchar* ptr(int i0, int i1);
const uchar* ptr(int i0, int i1) const;
uchar* ptr(int i0, int i1, int i2);
const uchar* ptr(int i0, int i1, int i2) const;
uchar* ptr(const int* idx);
const uchar* ptr(const int* idx) const;

// convenient template methods for element access.
// note that _Tp must match the actual matrix type -
// the functions do not do any on-fly type conversion
template<typename _Tp> _Tp& at(int i0);
template<typename _Tp> const _Tp& at(int i0) const;
template<typename _Tp> _Tp& at(int i0, int i1);
template<typename _Tp> const _Tp& at(int i0, int i1) const;
template<typename _Tp> _Tp& at(int i0, int i1, int i2);
template<typename _Tp> const _Tp& at(int i0, int i1, int i2) const;
template<typename _Tp> _Tp& at(const int* idx);
template<typename _Tp> const _Tp& at(const int* idx) const;

enum { MAGIC_VAL=0x42FE0000, AUTO_STEP=-1,
       CONTINUOUS_FLAG=CV_MAT_CONT_FLAG, MAX_DIM=CV_MAX_DIM };

// combines data type, continuity flag, signature (magic value)
int flags;
// the array dimensionality
int dims;

// data reference counter

```

```

    int* refcount;
    // pointer to the data
    uchar* data;
    // and its actual beginning and end
    uchar* datastart;
    uchar* dataend;

    // step and size for each dimension, MAX_DIM at max
    int size[MAX_DIM];
    size_t step[MAX_DIM];
};

```

---

The class `MatND` describes n-dimensional dense numerical single-channel or multi-channel array. This is a convenient representation for multi-dimensional histograms (when they are not very sparse, otherwise `SparseMat` will do better), voxel volumes, stacked motion fields etc. The data layout of matrix  $M$  is defined by the array of `M.step[]`, so that the address of element  $(i_0, \dots, i_{M.dims-1})$ , where  $0 \leq i_k < M.size[k]$  is computed as:

$$addr(M_{i_0, \dots, i_{M.dims-1}}) = M.data + M.step[0] * i_0 + M.step[1] * i_1 + \dots + M.step[M$$

which is more general form of the respective formula for `Mat`, wherein `size[0] ~ rows`, `size[1] ~ cols`, `step[0]` was simply called `step`, and `step[1]` was not stored at all but computed as `Mat::elemSize()`.

In other aspects `MatND` is also very similar to `cvMat()`, with the following limitations and differences:

- much less operations are implemented for `MatND`
- currently, algebraic expressions with `MatND`'s are not supported
- the `MatND` iterator is completely different from `cvMat()` and `Mat_` iterators. The latter are per-element iterators, while the former is per-slice iterator, see below.

Here is how you can use `MatND` to compute NxNxN histogram of color 8bpp image (i.e. each channel value ranges from 0..255 and we quantize it to 0..N-1):

---

```

void computeColorHist(const Mat& image, MatND& hist, int N)
{
    const int histSize[] = {N, N, N};

    // make sure that the histogram has proper size and type
    hist.create(3, histSize, CV_32F);

    // and clear it
    hist = Scalar(0);

    // the loop below assumes that the image
    // is 8-bit 3-channel, so let's check it.

```

```

CV_Assert(image.type() == CV_8UC3);
MatConstIterator_<Vec3b> it = image.begin<Vec3b>(),
                           it_end = image.end<Vec3b>();
for( ; it != it_end; ++it )
{
    const Vec3b& pix = *it;

    // we could have incremented the cells by 1.f/(image.rows*image.cols)
    // instead of 1.f to make the histogram normalized.
    hist.at<float>(pix[0]*N/256, pix[1]*N/256, pix[2]*N/256) += 1.f;
}
}

```

---

And here is how you can iterate through MatND elements:

```

void normalizeColorHist(MatND& hist)
{
    #if 1
        // intialize iterator (the style is different from STL).
        // after initialization the iterator will contain
        // the number of slices or planes
        // the iterator will go through
        MatNDIterator it(hist);
        double s = 0;
        // iterate through the matrix. on each iteration
        // it.planes[*] (of type Mat) will be set to the current
        for(int p = 0; p < it.nplanes; p++, ++it)
            s += sum(it.planes[0])[0];
        it = MatNDIterator(hist);
        s = 1./s;
        for(int p = 0; p < it.nplanes; p++, ++it)
            it.planes[0] *= s;
    #elif 1
        // this is a shorter implementation of the above
        // using built-in operations on MatND
        double s = sum(hist)[0];
        hist.convertTo(hist, hist.type(), 1./s, 0);
    #else
        // and this is even shorter one
        // (assuming that the histogram elements are non-negative)
        normalize(hist, hist, 1, 0, NORM_L1);
    #endif
}

```

---

You can iterate though several matrices simultaneously as long as they have the same geometry (dimensionality and all the dimension sizes are the same), which is useful for binary and n-ary operations on such matrices. Just pass those matrices to MatNDIterator. Then, during the iteration `it.planes[0]`, `it.planes[1]`, ... will be the slices of the corresponding matrices.

## MatND\_

Template class for n-dimensional dense array derived from *MatND*.

```

template<typename _Tp> class MatND_ : public MatND
{
public:
    typedef _Tp value_type;
    typedef typename DataType<_Tp>::channel_type channel_type

    // constructors, the same as in MatND, only the type is c
    MatND_();
    MatND_(int dims, const int* _sizes);
    MatND_(int dims, const int* _sizes, const _Tp& _s);
    MatND_(const MatND& m);
    MatND_(const MatND_& m);
    MatND_(const MatND_& m, const Range* ranges);
    MatND_(const CvMatND* m, bool copyData=false);
    MatND_& operator = (const MatND& m);
    MatND_& operator = (const MatND_& m);
    // different initialization function
    // where we take _Tp instead of Scalar
    MatND_& operator = (const _Tp& s);

    // no special destructor is needed; use the one from MatN

    void create(int dims, const int* _sizes);
    template<typename T2> operator MatND_<T2>() const;
    MatND_ clone() const;
    MatND_ operator ()(const Range* ranges) const;

    size_t elemSize() const;
    size_t elemSize1() const;
    int type() const;
    int depth() const;
    int channels() const;
    // step[i]/elemSize()
    size_t stepT(int i) const;
    size_t step1(int i) const;

    // shorter alternatives for MatND::at<_Tp>.
    _Tp& operator ()(const int* idx);
    const _Tp& operator ()(const int* idx) const;
    _Tp& operator ()(int idx0);
    const _Tp& operator ()(int idx0) const;
    _Tp& operator ()(int idx0, int idx1);
    const _Tp& operator ()(int idx0, int idx1) const;
    _Tp& operator ()(int idx0, int idx1, int idx2);
    const _Tp& operator ()(int idx0, int idx1, int idx2) cons

```

```

    _Tp& operator ()(int idx0, int idx1, int idx2);
    const _Tp& operator ()(int idx0, int idx1, int idx2) const
};

```

---

MatND\_ relates to MatND almost like Mat\_ to **cvMat()** - it provides a bit more convenient element access operations and adds no extra members of virtual methods to the base class, thus references/pointers to MatND\_ and MatND can be easily converted one to another, e.g.

```

// alternative variant of the above histogram accumulation lc
...
CV_Assert(hist.type() == CV_32FC1);
MatND_<float>& _hist = (MatND_<float>&)hist;
for( ; it != it_end; ++it )
{
    const Vec3b& pix = *it;
    _hist(pix[0]*N/256, pix[1]*N/256, pix[2]*N/256) += 1.f;
}
...

```

---

## SparseMat

---

Sparse n-dimensional array.

```

class SparseMat
{
public:
    typedef SparseMatIterator iterator;
    typedef SparseMatConstIterator const_iterator;

    // internal structure - sparse matrix header
    struct Hdr
    {
        ...
    };

    // sparse matrix node - element of a hash table
    struct Node
    {
        size_t hashval;
        size_t next;
        int idx[CV_MAX_DIM];
    };

    /////////////// constructors and destructor ///////////////
    // default constructor
    SparseMat();
    // creates matrix of the specified size and type

```

```

SparseMat(int dims, const int* _sizes, int _type);
// copy constructor
SparseMat(const SparseMat& m);
// converts dense 2d matrix to the sparse form,
// if try1d is true and matrix is a single-column matrix (Nx1),
// then the sparse matrix will be 1-dimensional.
SparseMat(const Mat& m, bool try1d=false);
// converts dense n-d matrix to the sparse form
SparseMat(const MatND& m);
// converts old-style sparse matrix to the new-style.
// all the data is copied, so that "m" can be safely
// deleted after the conversion
SparseMat(const CvSparseMat* m);
// destructor
~SparseMat();

////////// assignment operations //////////

// this is O(1) operation; no data is copied
SparseMat& operator = (const SparseMat& m);
// (equivalent to the corresponding constructor with try1d=false)
SparseMat& operator = (const Mat& m);
SparseMat& operator = (const MatND& m);

// creates full copy of the matrix
SparseMat clone() const;

// copy all the data to the destination matrix.
// the destination will be reallocated if needed.
void copyTo( SparseMat& m ) const;
// converts 1D or 2D sparse matrix to dense 2D matrix.
// If the sparse matrix is 1D, then the result will
// be a single-column matrix.
void copyTo( Mat& m ) const;
// converts arbitrary sparse matrix to dense matrix.
// watch out the memory!
void copyTo( MatND& m ) const;
// multiplies all the matrix elements by the specified scalar
void convertTo( SparseMat& m, int rtype, double alpha=1 )
// converts sparse matrix to dense matrix with optional type conversion
// When rtype=-1, the destination element type will be the same
// as the sparse matrix element type.
// Otherwise rtype will specify the depth and
// the number of channels will remain the same as in the sparse matrix
void convertTo( Mat& m, int rtype, double alpha=1, double beta=0 ) const;
void convertTo( MatND& m, int rtype, double alpha=1, double beta=0 ) const;

// not used now
void assignTo( SparseMat& m, int type=-1 ) const;

// reallocates sparse matrix. If it was already of the proper size,

```

```

// it is simply cleared with clear(), otherwise,
// the old matrix is released (using release()) and the new one .
void create(int dims, const int* _sizes, int _type);
// sets all the matrix elements to 0, which means clearing the ha
void clear();
// manually increases reference counter to the header.
void addref();
// decreases the header reference counter, when it reaches 0,
// the header and all the underlying data are deallocated.
void release();

// converts sparse matrix to the old-style representation.
// all the elements are copied.
operator CvSparseMat*() const;
// size of each element in bytes
// (the matrix nodes will be bigger because of
// element indices and other SparseMat::Node elements).
size_t elemSize() const;
// elemSize()/channels()
size_t elemSize1() const;

// the same is in Mat and MatND
int type() const;
int depth() const;
int channels() const;

// returns the array of sizes and 0 if the matrix is not allocated
const int* size() const;
// returns i-th size (or 0)
int size(int i) const;
// returns the matrix dimensionality
int dims() const;
// returns the number of non-zero elements
size_t nzcount() const;

// compute element hash value from the element indices:
// 1D case
size_t hash(int i0) const;
// 2D case
size_t hash(int i0, int i1) const;
// 3D case
size_t hash(int i0, int i1, int i2) const;
// n-D case
size_t hash(const int* idx) const;

// low-level element-access functions,
// special variants for 1D, 2D, 3D cases and the generic one for
//
// return pointer to the matrix element.
// if the element is there (it's non-zero), the pointer to it is
// if it's not there and createMissing=false, NULL pointer is re

```

```

// if it's not there and createMissing=true, then the new element
// is created and initialized with 0. Pointer to it is returned.
// If the optional hashval pointer is not NULL, the element hashval
// not computed, but *hashval is taken instead.
uchar* ptr(int i0, bool createMissing, size_t* hashval=0);
uchar* ptr(int i0, int i1, bool createMissing, size_t* hashval);
uchar* ptr(int i0, int i1, int i2, bool createMissing, size_t* hashval);
uchar* ptr(const int* idx, bool createMissing, size_t* hashval);

// higher-level element access functions:
// ref<_Tp>(i0,...[,hashval]) - equivalent to *(_Tp*)ptr(i0,...,t)
// always return valid reference to the element.
// If it's did not exist, it is created.
// find<_Tp>(i0,...[,hashval]) - equivalent to (_const Tp*)ptr(i0,...,t)
// return pointer to the element or NULL pointer if the element is not there.
// value<_Tp>(i0,...[,hashval]) - equivalent to
// { const _Tp* p = find<_Tp>(i0,...[,hashval]); return p ? *p : 0; }
// that is, 0 is returned when the element is not there.
// note that _Tp must match the actual matrix type -
// the functions do not do any on-fly type conversion

// 1D case
template<typename _Tp> _Tp& ref(int i0, size_t* hashval=0)
template<typename _Tp> _Tp value(int i0, size_t* hashval=0)
template<typename _Tp> const _Tp* find(int i0, size_t* hashval=0)

// 2D case
template<typename _Tp> _Tp& ref(int i0, int i1, size_t* hashval=0)
template<typename _Tp> _Tp value(int i0, int i1, size_t* hashval=0)
template<typename _Tp> const _Tp* find(int i0, int i1, size_t* hashval=0)

// 3D case
template<typename _Tp> _Tp& ref(int i0, int i1, int i2, size_t* hashval=0)
template<typename _Tp> _Tp value(int i0, int i1, int i2, size_t* hashval=0)
template<typename _Tp> const _Tp* find(int i0, int i1, int i2, size_t* hashval=0)

// n-D case
template<typename _Tp> _Tp& ref(const int* idx, size_t* hashval=0)
template<typename _Tp> _Tp value(const int* idx, size_t* hashval=0)
template<typename _Tp> const _Tp* find(const int* idx, size_t* hashval=0)

// erase the specified matrix element.
// When there is no such element, the methods do nothing
void erase(int i0, int i1, size_t* hashval=0);
void erase(int i0, int i1, int i2, size_t* hashval=0);
void erase(const int* idx, size_t* hashval=0);

// return the matrix iterators,
// pointing to the first sparse matrix element,
SparseMatIterator begin();
SparseMatConstIterator begin() const;

```

```

// ... or to the point after the last sparse matrix element
SparseMatIterator end();
SparseMatConstIterator end() const;

// and the template forms of the above methods.
// _Tp must match the actual matrix type.
template<typename _Tp> SparseMatIterator<_Tp> begin();
template<typename _Tp> SparseMatConstIterator<_Tp> begin
template<typename _Tp> SparseMatIterator<_Tp> end();
template<typename _Tp> SparseMatConstIterator<_Tp> end()

// return value stored in the sparse matrix node
template<typename _Tp> _Tp& value(Node* n);
template<typename _Tp> const _Tp& value(const Node* n) const

////////// some internal-use methods //////////
...

// pointer to the sparse matrix header
Hdr* hdr;
};

```

---

The class `SparseMat` represents multi-dimensional sparse numerical arrays. Such a sparse array can store elements of any type that `Mat` and `MatND` can store. “Sparse” means that only non-zero elements are stored (though, as a result of operations on a sparse matrix, some of its stored elements can actually become 0. It’s up to the user to detect such elements and delete them using `SparseMat::erase`). The non-zero elements are stored in a hash table that grows when it’s filled enough, so that the search time is  $O(1)$  in average (regardless of whether element is there or not). Elements can be accessed using the following methods:

- query operations (`SparseMat::ptr` and the higher-level `SparseMat::ref`, `SparseMat::value` and `SparseMat::find`), e.g.:

```

const int dims = 5;
int size[] = {10, 10, 10, 10, 10};
SparseMat sparse_mat(dims, size, CV_32F);
for(int i = 0; i < 1000; i++)
{
    int idx[dims];
    for(int k = 0; k < dims; k++)
        idx[k] = rand()%sparse_mat.size(k);
    sparse_mat.ref<float>(idx) += 1.f;
}

```

---

- sparse matrix iterators. Like `Mat` iterators and unlike `MatND` iterators, the sparse matrix iterators are STL-style, that is, the iteration loop is familiar to C++ users:

---

```

// prints elements of a sparse floating-point matrix
// and the sum of elements.
SparseMatConstIterator_<float>
    it = sparse_mat.begin<float>(),
    it_end = sparse_mat.end<float>();
double s = 0;
int dims = sparse_mat.dims();
for(; it != it_end; ++it)
{
    // print element indices and the element value
    const Node* n = it.node();
    printf("(")
    for(int i = 0; i < dims; i++)
        printf("%3d%c", n->idx[i], i < dims-1 ? ',' : ')');
    printf(": %f\n", *it);
    s += *it;
}
printf("Element sum is %g\n", s);

```

---

If you run this loop, you will notice that elements are enumerated in no any logical order (lexicographical etc.), they come in the same order as they stored in the hash table, i.e. semi-randomly. You may collect pointers to the nodes and sort them to get the proper ordering. Note, however, that pointers to the nodes may become invalid when you add more elements to the matrix; this is because of possible buffer reallocation.

- a combination of the above 2 methods when you need to process 2 or more sparse matrices simultaneously, e.g. this is how you can compute unnormalized cross-correlation of the 2 floating-point sparse matrices:

---

```

double cross_corr(const SparseMat& a, const SparseMat& b)
{
    const SparseMat *_a = &a, *_b = &b;
    // if b contains less elements than a,
    // it's faster to iterate through b
    if(_a->nzcount() > _b->nzcount())
        std::swap(_a, _b);
    SparseMatConstIterator_<float> it = _a->begin<float>(),
                                    it_end = _a->end<float>();
    double ccorr = 0;
    for(; it != it_end; ++it)
    {
        // take the next element from the first matrix
        float avalue = *it;
        const Node* anode = it.node();
        // and try to find element with the same index in the
        // since the hash value depends only on the element i
        // we reuse hashvalue stored in the node
        float bvalue = _b->value<float>(anode->idx, &anode->ha

```

```

        ccorr += avalue*bvalue;
    }
    return ccorr;
}

```

## SparseMat\_

Template sparse n-dimensional array class derived from *SparseMat*

```

template<typename _Tp> class SparseMat_ : public SparseMat
{
public:
    typedef SparseMatIterator<_Tp> iterator;
    typedef SparseMatConstIterator<_Tp> const_iterator;

    // constructors;
    // the created matrix will have data type = DataType<_Tp>
    SparseMat_();
    SparseMat_(int dims, const int* _sizes);
    SparseMat_(const SparseMat& m);
    SparseMat_(const SparseMat_& m);
    SparseMat_(const Mat& m);
    SparseMat_(const MatND& m);
    SparseMat_(const CvSparseMat* m);
    // assignment operators; data type conversion is done whe
    SparseMat_& operator = (const SparseMat& m);
    SparseMat_& operator = (const SparseMat_& m);
    SparseMat_& operator = (const Mat& m);
    SparseMat_& operator = (const MatND& m);

    // equivalent to the corresponding parent class methods
    SparseMat_ clone() const;
    void create(int dims, const int* _sizes);
    operator CvSparseMat*() const;

    // overridden methods that do extra checks for the data ty
    int type() const;
    int depth() const;
    int channels() const;

    // more convenient element access operations.
    // ref() is retained (but <_Tp> specification is not need
    // operator () is equivalent to SparseMat::value<_Tp>
    _Tp& ref(int i0, size_t* hashval=0);
    _Tp operator()(int i0, size_t* hashval=0) const;
    _Tp& ref(int i0, int i1, size_t* hashval=0);
    _Tp operator()(int i0, int i1, size_t* hashval=0) const;
    _Tp& ref(int i0, int i1, int i2, size_t* hashval=0);
    _Tp operator()(int i0, int i1, int i2, size_t* hashval=0)

```

```

_Tp& ref(const int* idx, size_t* hashval=0);
_Tp operator()(const int* idx, size_t* hashval=0) const;

// iterators
SparseMatIterator_<_Tp> begin();
SparseMatConstIterator_<_Tp> begin() const;
SparseMatIterator_<_Tp> end();
SparseMatConstIterator_<_Tp> end() const;
};

```

---

`SparseMat_` is a thin wrapper on top of *SparseMat*, made in the same way as `Mat_` and `MatND_`. It simplifies notation of some operations, and that's it.

```

int sz[] = {10, 20, 30};
SparseMat_<double> M(3, sz);
...
M.ref(1, 2, 3) = M(4, 5, 6) + M(7, 8, 9);

```

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Operations on Arrays

---

## abs

---

MatExpr<...> **abs**(const Mat& src)

MatExpr<...> **abs**(const MatExpr<...>& src)

Computes absolute value of each matrix element

**Parameter:** src – matrix or matrix expression

abs is a meta-function that is expanded to one of *absdiff* forms:

- $C = \text{abs}(A-B)$  is equivalent to `absdiff(A, B, C)` and
- $C = \text{abs}(A)$  is equivalent to `absdiff(A, Scalar::all(0), C)`.
- $C = \text{Mat\_}<\text{Vec}<\text{uchar}, n> >(\text{abs}(A * \text{math:}`\alpha` $` + \text{math:}`\beta` $`))$  is equivalent to `convertScaleAbs(A, C, alpha, beta)`

The output matrix will have the same size and the same type as the input one (except for the last case, where C will be `depth=CV_8U`).

See also: *Matrix Expressions*, *absdiff*, `[cppfunc.saturatecast]bgroup({saturate_cast})`

## absdiff

---

void **absdiff**(const Mat& src1, const Mat& src2, Mat& dst)

void **absdiff**(const Mat& src1, const Scalar& sc, Mat& dst)

void **absdiff**(const MatND& src1, const MatND& src2, MatND& dst)

void **absdiff**(const MatND& src1, const Scalar& sc, MatND& dst)

Computes per-element absolute difference between 2 arrays or between array and a scalar.

- Parameters:**
- *src1* – The first input array
  - *src2* – The second input array; Must be the same size and same type as *src1*
  - *sc* – Scalar; the second input parameter
  - *dst* – The destination array; it will have the same size and same type as *src1*; see `Mat::create`

The functions `absdiff` compute:

- absolute difference between two arrays

$$\text{dst}(I) = \text{saturate}(|\text{src1}(I) - \text{src2}(I)|)$$

- or absolute difference between array and a scalar:

$$\text{dst}(I) = \text{saturate}(|\text{src1}(I) - \text{sc}|)$$

where *I* is multi-dimensional index of array elements. in the case of multi-channel arrays each channel is processed independently.

See also: [abs](#), `[cppfunc.saturatecast]bgroup({saturate_ cast})`

## add

---

void **add**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*)

void **add**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, const Mat& *mask*)

void **add**(const Mat& *src1*, const Scalar& *sc*, Mat& *dst*, const Mat& *mask=Mat()*)

void **add**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*)

void **add**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*, const MatND& *mask*)

void **add**(const MatND& *src1*, const Scalar& *sc*, MatND& *dst*, const MatND& *mask=MatND()*)

Computes the per-element sum of two arrays or an array and a scalar.

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array. It must have the same size

and same type as `src1`

- `sc` – Scalar; the second input parameter
- `dst` – The destination array; it will have the same size and same type as `src1`; see `Mat::create`
- `mask` – The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `add` compute:

- the sum of two arrays:

$$\text{dst}(I) = \text{saturate}(\text{src1}(I) + \text{src2}(I)) \quad \text{if } \text{mask}(I) \neq 0$$

- or the sum of array and a scalar:

$$\text{dst}(I) = \text{saturate}(\text{src1}(I) + \text{sc}) \quad \text{if } \text{mask}(I) \neq 0$$

where `I` is multi-dimensional index of array elements.

The first function in the above list can be replaced with matrix expressions:

---

```
dst = src1 + src2;
dst += src1; // equivalent to add(dst, src1, dst);
```

---

in the case of multi-channel arrays each channel is processed independently.

See also: [subtract](#), [addWeighted](#), [scaleAdd](#), [convertScale](#), [Matrix Expressions](#), `[cppfunc.saturatecast]bgroup({saturate_cast})`.

## addWeighted

---

void **addWeighted**(const Mat& *src1*, double *alpha*, const Mat& *src2*, double *beta*, double *gamma*, Mat& *dst*)

void **addWeighted**(const MatND& *src1*, double *alpha*, const MatND& *src2*, double *beta*, double *gamma*, MatND& *dst*)

Computes the weighted sum of two arrays.

- Parameters:**
- `src1` – The first source array
  - `alpha` – Weight for the first array elements
  - `src2` – The second source array; must have the same size and same type as `src1`
  - `beta` – Weight for the second array elements
  - `dst` – The destination array; it will have the same size and

- same type as `src1`
- *gamma* – Scalar, added to each sum

The functions `addWeighted` calculate the weighted sum of two arrays as follows:

$$\text{dst}(I) = \text{saturate}(\text{src1}(I) * \text{alpha} + \text{src2}(I) * \text{beta} + \text{gamma})$$

where  $I$  is multi-dimensional index of array elements.

The first function can be replaced with a matrix expression:

---

```
dst = src1*alpha + src2*beta + gamma;
```

---

In the case of multi-channel arrays each channel is processed independently.

See also: [add](#), [subtract](#), [scaleAdd](#), [convertScale](#), [Matrix Expressions](#), `[cppfunc.saturatecast]bgroup({saturate_ cast})`.

## cv::bitwise\_and

---

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

```
void bitwise_and(const Mat& src1, const Mat& src2, Mat& dst, const Mat& mask=Mat())
```

```
void bitwise_and(const Mat& src1, const Scalar& sc, Mat& dst, const Mat& mask=Mat())
```

```
void bitwise_and(const MatND& src1, const MatND& src2, MatND& dst, const MatND& mask=MatND())
```

```
void bitwise_and(const MatND& src1, const Scalar& sc, MatND& dst, const MatND& mask=MatND())
```

- **src1** - The first source array
- **src2** - The second source array. It must have the same size and same type as `src1`
- **sc** - Scalar; the second input parameter
- **dst** - The destination array; it will have the same size and same type as `src1`; see `Mat::create`

- **mask** - The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_and` compute per-element bit-wise logical conjunction:

- of two arrays

$$\text{dst}(I) = \text{src1}(I) \wedge \text{src2}(I) \quad \text{if } \text{mask}(I) \neq 0$$

- or array and a scalar:

$$\text{dst}(I) = \text{src1}(I) \wedge \text{sc} \quad \text{if } \text{mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation, and in the case of multi-channel arrays each channel is processed independently.

See also: `[cppfunc.bitwise.and]bgroup({bitwise_and})`, `[cppfunc.bitwise.not]bgroup({bitwise_not})`, `[cppfunc.bitwise.xor]bgroup({bitwise_xor})`

## cv::bitwise\_not

---

Inverts every bit of array

void **bitwise\_not**(const Mat& *src*, Mat& *dst*)

void **bitwise\_not**(const MatND& *src*, MatND& *dst*)

- **src1** - The source array
- **dst** - The destination array; it is reallocated to be of the same size and the same type as `src`; see `Mat::create`
- **mask** - The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_not` compute per-element bit-wise inversion of the source array:

$$\text{dst}(I) = \neg \text{src}(I)$$

In the case of floating-point source array its machine-specific bit representation (usually IEEE754-compliant) is used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: `[cppfunc.bitwise.and]bgroup({bitwise_and})`, `[cppfunc.bitwise.or]bgroup({bitwise_or})`, `[cppfunc.bitwise.xor]bgroup({bitwise_xor})`

## cv::bitwise\_or

---

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

```
void bitwise_or(const Mat& src1, const Mat& src2, Mat& dst, const Mat&
mask=Mat())
```

```
void bitwise_or(const Mat& src1, const Scalar& sc, Mat& dst, const Mat&
mask=Mat())
```

```
void bitwise_or(const MatND& src1, const MatND& src2, MatND& dst,
const MatND& mask=MatND())
```

```
void bitwise_or(const MatND& src1, const Scalar& sc, MatND& dst, const
MatND& mask=MatND())
```

- **src1** - The first source array
- **src2** - The second source array. It must have the same size and same type as *src1*
- **sc** - Scalar; the second input parameter
- **dst** - The destination array; it is reallocated to be of the same size and the same type as *src1*; see `Mat::create`
- **mask** - The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_or` compute per-element bit-wise logical disjunction

- of two arrays

$$\text{dst}(I) = \text{src1}(I) \vee \text{src2}(I) \quad \text{if } \text{mask}(I) \neq 0$$

- or array and a scalar:

$$\text{dst}(I) = \text{src1}(I) \vee \text{sc} \quad \text{if } \text{mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: `[cppfunc.bitwise.and]bgroup({bitwise_and})`, `[cppfunc.bitwise.not]bgroup({bitwise_not})`, `[cppfunc.bitwise.or]bgroup({bitwise_or})`

## cv::bitwise\_xor

---

Calculates per-element bit-wise “exclusive or” operation on two arrays and an array and a scalar.

```
void bitwise_xor(const Mat& src1, const Mat& src2, Mat& dst, const Mat& mask=Mat())
```

```
void bitwise_xor(const Mat& src1, const Scalar& sc, Mat& dst, const Mat& mask=Mat())
```

```
void bitwise_xor(const MatND& src1, const MatND& src2, MatND& dst, const MatND& mask=MatND())
```

```
void bitwise_xor(const MatND& src1, const Scalar& sc, MatND& dst, const MatND& mask=MatND())
```

- **src1** - The first source array
- **src2** - The second source array. It must have the same size and same type as *src1*
- **sc** - Scalar; the second input parameter
- **dst** - The destination array; it is reallocated to be of the same size and the same type as *src1*; see `Mat::create`
- **mask** - The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_xor` compute per-element bit-wise logical “exclusive or” operation

- on two arrays

$$\text{dst}(I) = \text{src1}(I) \oplus \text{src2}(I) \quad \text{if } \text{mask}(I) \neq 0$$

- or array and a scalar:

$$\text{dst}(I) = \text{src1}(I) \oplus \text{sc} \quad \text{if } \text{mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: `[cppfunc.bitwise.and]bgroup({bitwise_and})`, `[cppfunc.bitwise.not]bgroup({bitwise_not})`, `[cppfunc.bitwise.or]bgroup({bitwise_or})`

## calcCovarMatrix

void **calcCovarMatrix**(const Mat\* *samples*, int *nsamples*, Mat& *covar*, Mat& *mean*, int *flags*, int *ctype*=CV\_64F)

void **calcCovarMatrix**(const Mat& *samples*, Mat& *covar*, Mat& *mean*, int *flags*, int *ctype*=CV\_64F)

Calculates covariation matrix of a set of vectors

- Parameters:**
- *samples* – The samples, stored as separate matrices, or as rows of
  - *nsamples* – The number of samples when they are stored separate
  - *covar* – The output covariance matrix; it will have type="`ctype`" and
  - *mean* – The input or output (depending on the flags) array - the input vectors
  - *flags* – The operation flags, a combination of the following values
    - **CV\_COVAR\_SCRAMBLED** - The output covariance matrix is calculated as:
 
$$\text{scale} \cdot [\text{vects}[0] - \text{mean}, \text{vects}[1] - \text{mean}, \dots]^T \cdot [\text{vects}[0] -$$

, that is, the covariance matrix will be `nsamples × nsamples`. 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:

$$\text{scale} \cdot [\text{vects}[0] - \text{mean}, \text{vects}[1] - \text{mean}, \dots] \cdot [\text{vects}[0] - \text{mean}, \text{vects}[1] - \text{mean}, \dots]^T$$

, that is, `covar` will be a square matrix of the same 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 `mean` from the input vectors, but, instead, uses the passed `mean` vector. This is useful if `mean` has been pre-computed or known a-priori, or if the covariance matrix is calculated by parts - in this case, `mean` is not a mean vector of the input subset 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./nsamples`; 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 (i.e. `scale=1`).
- **CV\_COVAR\_ROWS** - [Only useful in the second variant of the function] The flag means that all the input vectors are stored as rows of the `samples` matrix. `mean` should be a single-row vector in this case.
- **CV\_COVAR\_COLS** - [Only useful in the second variant of the function] The flag means that all the input vectors are stored as columns of the `samples` matrix. `mean` should be a single-column vector in this case.

The functions `calcCovarMatrix` calculate the covariance matrix and, optionally, the mean vector of the set of input vectors.

See also: [PCA](#), [mulTransposed](#), [Mahalanobis](#)

## cartToPolar

---

void **cartToPolar**(const Mat& *x*, const Mat& *y*, Mat& *magnitude*, Mat& *angle*, bool *angleInDegrees=false*)

Calculates the magnitude and angle of 2d vectors.

- Parameters:**
- *x* – The array of x-coordinates; must be single-precision or double-precision floating-point array
  - *y* – The array of y-coordinates; it must have the same size and same type as *x*
  - *magnitude* – The destination array of magnitudes of the same size and same type as *x*
  - *angle* – The destination array of angles of the same size and same type as *x*. The angles are measured in radians (0 to  $2\pi$ ) or in degrees (0 to 360 degrees).
  - *angleInDegrees* – The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

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

$$\begin{aligned} \text{magnitude}(I) &= \sqrt{x(I)^2 + y(I)^2}, \\ \text{angle}(I) &= \text{atan2}(y(I), x(I)) \cdot [180/\pi] \end{aligned}$$

The angles are calculated with  $\sim 0.3^\circ$  accuracy. For the (0,0) point, the angle is

set to 0.

## checkRange

---

bool **checkRange**(const Mat& *src*, bool *quiet=true*, Point\* *pos=0*, double *minVal=-DBL\_MAX*, double *maxVal=DBL\_MAX*)

bool **checkRange**(const MatND& *src*, bool *quiet=true*, int\* *pos=0*, double *minVal=-DBL\_MAX*, double *maxVal=DBL\_MAX*)

Checks every element of an input array for invalid values.

- Parameters:**
- *src* – The array to check
  - *quiet* – The flag indicating whether the functions quietly return false when the array elements are out of range, or they throw an exception.
  - *pos* – The optional output parameter, where the position of the first outlier is stored. In the second function *pos*, when not NULL, must be a pointer to array of *src.dims* elements
  - *minVal* – The inclusive lower boundary of valid values range
  - *maxVal* – The exclusive upper boundary of valid values range

The functions `checkRange` check that every array element is neither NaN nor  $\pm\infty$ . When *minVal* < -DBL\_MAX and *maxVal* < DBL\_MAX, then the functions also check that each value is between *minVal* and *maxVal*. in the case of multi-channel arrays each channel is processed independently. If some values are out of range, position of the first outlier is stored in *pos* (when *pos*  $\neq$  0), and then the functions either return false (when *quiet=true*) or throw an exception.

## compare

---

void **compare**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, int *cmpop*)

void **compare**(const Mat& *src1*, double *value*, Mat& *dst*, int *cmpop*)

void **compare**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*, int *cmpop*)

void **compare**(const MatND& *src1*, double *value*, MatND& *dst*, int *cmpop*)

Performs per-element comparison of two arrays or an array and scalar value.

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array; must have the same size and same type as *src1*
  - *value* – The scalar value to compare each array element with
  - *dst* – The destination array; will have the same size as *src1* and type=`CV_8UC1`
  - *cmpop* –  
The flag specifying the relation between the elements to be checked
    - **CMP\_EQ** -  $\text{src1}(I) = \text{src2}(I)$  or  $\text{src1}(I) = \text{value}$
    - **CMP\_GT** -  $\text{src1}(I) > \text{src2}(I)$  or  $\text{src1}(I) > \text{value}$
    - **CMP\_GE** -  $\text{src1}(I) \geq \text{src2}(I)$  or  $\text{src1}(I) \geq \text{value}$
    - **CMP\_LT** -  $\text{src1}(I) < \text{src2}(I)$  or  $\text{src1}(I) < \text{value}$
    - **CMP\_LE** -  $\text{src1}(I) \leq \text{src2}(I)$  or  $\text{src1}(I) \leq \text{value}$
    - **CMP\_NE** -  $\text{src1}(I) \neq \text{src2}(I)$  or  $\text{src1}(I) \neq \text{value}$

The functions `compare` compare each element of *src1* with the corresponding element of *src2* or with real scalar value. When the comparison result is true, the corresponding element of destination array is set to 255, otherwise it is set to 0:

- $\text{dst}(I) = \text{src1}(I) \text{ cmpop } \text{src2}(I) ? 255 : 0$
- $\text{dst}(I) = \text{src1}(I) \text{ cmpop } \text{value} ? 255 : 0$

The comparison operations can be replaced with the equivalent matrix expressions:

---

```
Mat dst1 = src1 >= src2;
Mat dst2 = src1 < 8;
...
```

---

See also: [checkRange](#), [min](#), [max](#), [threshold](#), [Matrix Expressions](#)

## completeSymm

---

void **completeSymm**(Mat& *mtx*, bool *lowerToUpper*=false)

Copies the lower or the upper half of a square matrix to another half.

- Parameters:**
- *mtx* – Input-output floating-point square matrix
  - *lowerToUpper* – If true, the lower half is copied to the upper half, otherwise the upper half is copied to the lower half

The function `completeSymm` copies the lower half of a square matrix to its

another half; the matrix diagonal remains unchanged:

- $mtx_{ij} = mtx_{ji}$  for  $i > j$  if `lowerToUpper=false`
- $mtx_{ij} = mtx_{ji}$  for  $i < j$  if `lowerToUpper=true`

See also: [flip](#), [transpose](#)

## convertScaleAbs

---

void **convertScaleAbs**(const Mat& *src*, Mat& *dst*, double *alpha*=1, double *beta*=0)

Scales, computes absolute values and converts the result to 8-bit.

- Parameters:**
- *src* – The source array
  - *dst* – The destination array
  - *alpha* – The optional scale factor
  - *beta* – The optional delta added to the scaled values

On each element of the input array the function `convertScaleAbs` performs 3 operations sequentially: scaling, taking absolute value, conversion to unsigned 8-bit type:

$$dst(I) = \text{saturate\_cast}\langle\text{uchar}\rangle(|src(I) * \alpha + \beta|)$$

in the case of multi-channel arrays the function processes each channel independently. When the output is not 8-bit, the operation can be emulated by calling `Mat::convertTo` method (or by using matrix expressions) and then by computing absolute value of the result, for example:

---

```
Mat_<float> A(30,30);
randu(A, Scalar(-100), Scalar(100));
Mat_<float> B = A*5 + 3;
B = abs(B);
// Mat_<float> B = abs(A*5+3) will also do the job,
// but it will allocate a temporary matrix
```

---

See also: `Mat::convertTo`, [abs](#)

## countNonZero

---

int **countNonZero**(const Mat& *mtx*)

int **countNonZero**(const MatND& *mtx*)

Counts non-zero array elements.

**Parameter:** *mtx* – Single-channel array

The function **cvCountNonZero()** returns the number of non-zero elements in *mtx*:

$$\sum_{I: \text{mtx}(I) \neq 0} 1$$

See also: [mean](#), [meanStdDev](#), [norm](#), [minMaxLoc](#), [calcCovarMatrix](#)

## cubeRoot

---

float **cubeRoot**(float *val*)

Computes cube root of the argument

**Parameter:** *val* – The function argument

The function `cubeRoot` computes  $\sqrt[3]{\text{val}}$ . Negative arguments are handled correctly, NaN and  $\pm\infty$  are not handled. The accuracy approaches the maximum possible accuracy for single-precision data.

## cvarrToMat

---

Mat **cvarrToMat**(const CvArr\* *src*, bool *copyData=false*, bool *allowND=true*, int *coiMode=0*)

Converts CvMat, IplImage or CvMatND to cv::Mat.

- Parameters:**
- *src* – The source **CvMat**, **IplImage** or **CvMatND**
  - *copyData* – When it is false (default value), no data is copied, only the new header is created. In this case the original array should not be deallocated while the new matrix header is used. The the parameter is true, all the data is copied, then user may deallocate the original array right after the conversion
  - *allowND* – When it is true (default value), then **CvMatND** is converted to **cvMat()** if it's possible (e.g. then the data is contiguous). If it's not possible, or when the parameter is false, the function will report an error
  - *coiMode* –

The parameter specifies how the `IplImage` COI (when set) is handled.

- If `coiMode=0`, the function will report an error if COI is set.
- If `coiMode=1`, the function will never report an error; instead it returns the header to the whole original image and user will have to check and process COI manually, see [extractImageCOI](#).

The function `cvarrToMat()` converts `CvMat`, `IplImage` or `CvMatND` header to `Mat` header, and optionally duplicates the underlying data. The constructed header is returned by the function.

When `copyData=false`, the conversion is done really fast (in  $O(1)$  time) and the newly created matrix header will have `refcount=0`, which means that no reference counting is done for the matrix data, and user has to preserve the data until the new header is destructed. Otherwise, when `copyData=true`, the new buffer will be allocated and managed as if you created a new matrix from scratch and copy the data there. That is, `cvarrToMat(src, true) :math: \sim \$() \text{ cvarrToMat}(src, false).clone()` (assuming that COI is not set). The function provides uniform way of supporting `CvArr` paradigm in the code that is migrated to use new-style data structures internally. The reverse transformation, from `Mat` to `CvMat` or `IplImage` can be done by simple assignment:

---

```
CvMat* A = cvCreateMat(10, 10, CV_32F);
cvSetIdentity(A);
IplImage A1; cvGetImage(A, &A1);
Mat B = cvarrToMat(A);
Mat B1 = cvarrToMat(&A1);
IplImage C = B;
CvMat C1 = B1;
// now A, A1, B, B1, C and C1 are different headers
// for the same 10x10 floating-point array.
// note, that you will need to use "&"
// to pass C & C1 to OpenCV functions, e.g:
printf("%g", cvDet(&C1));
```

---

Normally, the function is used to convert an old-style 2D array (`CvMat` or `IplImage`) to `cvMat()`, however, the function can also take `CvMatND` on input and create `Mat` for it, if it's possible. And for `CvMatND A` it is possible if and only if `A.dim[i].size*A.dim.step[i] == A.dim.step[i-1]` for all or for all but one `i`,  $0 < i < A.dims$ . That is, the matrix data should be continuous or it should be representable as a sequence of continuous matrices. By using this function in this way, you can process `CvMatND` using arbitrary element-wise function. But for more complex operations, such as filtering functions, it will not work, and you need to convert `CvMatND` to `MatND` using the

corresponding constructor of the latter.

The last parameter, `coiMode`, specifies how to react on an image with COI set: by default it's 0, and then the function reports an error when an image with COI comes in. And `coiMode=1` means that no error is signaled - user has to check COI presence and handle it manually. The modern structures, such as *Mat* and *MatND* do not support COI natively. To process individual channel of an new-style array, you will need either to organize loop over the array (e.g. using matrix iterators) where the channel of interest will be processed, or extract the COI using [mixChannels](#) (for new-style arrays) or [extractImageCOI](#) (for old-style arrays), process this individual channel and insert it back to the destination array if need (using *mixChannel* or *insertImageCOI*, respectively).

See also: [cvGetImage](#), [cvGetMat](#), [cvGetMatND](#), [extractImageCOI](#), [insertImageCOI](#), [mixChannels](#)

## dct

---

void **dct**(const Mat& *src*, Mat& *dst*, int *flags*=0)

Performs a forward or inverse discrete cosine transform of 1D or 2D array

- Parameters:**
- *src* – The source floating-point array
  - *dst* – The destination array; will have the same size and same type as `src`
  - *flags* –
    - **None** - `Nonebgroup({flags})bgroup({Transformation flags, a combination of the following values begin{description} cvarg{DCT_ INVERSE}{do an inverse 1D or 2D transform instead of the default forward transform.} cvarg{DCT_ 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.} end{description}})`

The function `dct` performs a forward or inverse discrete cosine transform (DCT) 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:

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

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

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

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

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

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

The function chooses the mode of operation by looking at the flags and size of the input array:

- if `(flags & DCT_INVERSE) == 0`, the function does forward 1D or 2D transform, otherwise it is inverse 1D or 2D transform.
- if `(flags & DCT_ROWS) != 0`, the function performs 1D transform of each row.
- otherwise, if the array is a single column or a single row, the function performs 1D transform
- otherwise it performs 2D transform.

Important note: currently `cv::dct` supports even-size arrays (2, 4, 6 ...). For data analysis and approximation you can pad the array when necessary.

Also, the function's performance depends very much, and not monotonically, on the array size, see [getOptimalDFTSize](#). In the current implementation DCT of a vector of size  $N$  is computed via DFT of a vector of size  $N/2$ , thus the optimal DCT size  $N^* \geq N$  can be computed as:

---

```
size_t getOptimalDCTSize(size_t N) { return 2*getOptimalDFTSize(N); }
```

---

See also: [dft](#), [getOptimalDFTSize](#), [idct](#)

## dft

---

```
void dft(const Mat& src, Mat& dst, int flags=0, int nonzeroRows=0)
```

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

- Parameters:**
- *src* – The source array, real or complex
  - *dst* – The destination array, which size and type depends on the `flags`
  - *flags* – Transformation flags, a combination of the following values
    - **DFT\_INVERSE** - do an inverse 1D or 2D transform instead of the default forward transform.
    - **DFT\_SCALE** - scale the result: divide it by the number of array elements. Normally, it is combined with `DFT_INVERSE`

- **DFT\_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.
- **DFT\_COMPLEX\_OUTPUT** - then the function performs forward transformation of 1D or 2D real array, the result, though being a complex array, has complex-conjugate symmetry (CCS), see the description below. Such an array can be packed into real array of the same size as input, which is the fastest option and which is what the function does by default. However, you may wish to get the full complex array (for simpler spectrum analysis etc.). Pass the flag to tell the function to produce full-size complex output array.
- **DFT\_REAL\_OUTPUT** - then the function performs inverse transformation of 1D or 2D complex array, the result is normally a complex array of the same size. However, if the source array has conjugate-complex symmetry (for example, it is a result of forward transformation with `DFT_COMPLEX_OUTPUT` flag), then the output is real array. While the function itself does not check whether the input is symmetrical or not, you can pass the flag and then the function will assume the symmetry and produce the real output array. Note that when the input is packed real array and inverse transformation is executed, the function treats the input as packed complex-conjugate symmetrical array, so the output will also be real array

**param nonzeroRows:**

When the parameter  $\neq 0$ , the function assumes that only the first

`nonzeroRows` rows of the input array (`DFT_INVERSE` is not set) or only the first `nonzeroRows` of the output array (`DFT_INVERSE` is set) contain non-zeros, thus the function can handle the rest of the rows more efficiently and thus save some time. This technique is very useful for computing array cross-correlation or convolution using DFT

Forward Fourier transform of 1D vector of  $N$  elements:

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

where  $F_{jk}^{(N)} = \exp(-2\pi ijk/N)$  and  $i = \sqrt{-1}$

Inverse Fourier transform of 1D vector of  $N$  elements:

$$\begin{aligned} X' &= (F^{(N)})^{-1} \cdot Y = (F^{(N)})^* \cdot y \\ X &= (1/N) \cdot X', \end{aligned}$$

where  $F^* = (\text{Re}(F^{(N)}) - \text{Im}(F^{(N)}))^{T}$

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:

$$\begin{aligned} X' &= (F^{(M)})^* \cdot Y \cdot (F^{(N)})^* \\ X &= \frac{1}{M \cdot N} \cdot X' \end{aligned}$$

In the case of real (single-channel) data, the packed format called CCS (complex-conjugate-symmetrical) that was borrowed from IPL and used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix} \text{Re}Y_{0,0} & \text{Re}Y_{0,1} & \text{Im}Y_{0,1} & \text{Re}Y_{0,2} & \text{Im}Y_{0,2} & \cdots & \text{Re}Y_{0,N/2-1} \\ \text{Re}Y_{1,0} & \text{Re}Y_{1,1} & \text{Im}Y_{1,1} & \text{Re}Y_{1,2} & \text{Im}Y_{1,2} & \cdots & \text{Re}Y_{1,N/2-1} \\ \text{Im}Y_{1,0} & \text{Re}Y_{2,1} & \text{Im}Y_{2,1} & \text{Re}Y_{2,2} & \text{Im}Y_{2,2} & \cdots & \text{Re}Y_{2,N/2-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \text{Re}Y_{M/2-1,0} & \text{Re}Y_{M-3,1} & \text{Im}Y_{M-3,1} & \dots & \dots & \dots & \text{Re}Y_{M-3,N/2-1} & I_1 \\ \text{Im}Y_{M/2-1,0} & \text{Re}Y_{M-2,1} & \text{Im}Y_{M-2,1} & \dots & \dots & \dots & \text{Re}Y_{M-2,N/2-1} & I_1 \\ \text{Re}Y_{M/2,0} & \text{Re}Y_{M-1,1} & \text{Im}Y_{M-1,1} & \dots & \dots & \dots & \text{Re}Y_{M-1,N/2-1} & I_1 \end{bmatrix}$$

in the case of 1D transform of real vector, the output will look as the first row of the above matrix.

So, the function chooses the operation mode depending on the flags and size of the input array:

- if `DFT_ROWS` is set or the input array has single row or single column then the function performs 1D forward or inverse transform (of each row of a matrix when `DFT_ROWS` is set, otherwise it will be 2D transform).

- if input array is real and `DFT_INVERSE` is not set, the function does forward 1D or 2D transform:
- when `DFT_COMPLEX_OUTPUT` is set then the output will be complex matrix of the same size as input.
- otherwise the output will be a real matrix of the same size as input. in the case of 2D transform it will use the packed format as shown above; in the case of single 1D transform it will look as the first row of the above matrix; in the case of multiple 1D transforms (when using `DFT_ROWS` flag) each row of the output matrix will look like the first row of the above matrix.
- otherwise, if the input array is complex and either `DFT_INVERSE` or `DFT_REAL_OUTPUT` are not set then the output will be a complex array of the same size as input and the function will perform the forward or inverse 1D or 2D transform of the whole input array or each row of the input array independently, depending on the flags `DFT_INVERSE` and `DFT_ROWS`.
- otherwise, i.e. when `DFT_INVERSE` is set, the input array is real, or it is complex but `DFT_REAL_OUTPUT` is set, the output will be a real array of the same size as input, and the function will perform 1D or 2D inverse transformation of the whole input array or each individual row, depending on the flags `DFT_INVERSE` and `DFT_ROWS`.

The scaling is done after the transformation if `DFT_SCALE` is set.

Unlike `dct`, the function supports arrays of arbitrary size, but only those arrays are processed efficiently, which sizes can be factorized in a product of small prime numbers (2, 3 and 5 in the current implementation). Such an efficient DFT size can be computed using `getOptimalDFTSize` method.

Here is the sample on how to compute DFT-based convolution of two 2D real arrays:

---

```
void convolveDFT(const Mat& A, const Mat& B, Mat& C)
{
    // reallocate the output array if needed
    C.create(abs(A.rows - B.rows)+1, abs(A.cols - B.cols)+
        Size dftSize;
    // compute the size of DFT transform
    dftSize.width = getOptimalDFTSize(A.cols + B.cols - 1)
    dftSize.height = getOptimalDFTSize(A.rows + B.rows - 1)

    // allocate temporary buffers and initialize them with
    Mat tempA(dftSize, A.type(), Scalar::all(0));
    Mat tempB(dftSize, B.type(), Scalar::all(0));

    // copy A and B to the top-left corners of tempA and t
    Mat roiA(tempA, Rect(0,0,A.cols,A.rows));
    A.copyTo(roiA);
    Mat roiB(tempB, Rect(0,0,B.cols,B.rows));
```

```

B.copyTo(roiB);

// now transform the padded A & B in-place;
// use "nonzeroRows" hint for faster processing
dft(tempA, tempA, 0, A.rows);
dft(tempB, tempB, 0, B.rows);

// multiply the spectrums;
// the function handles packed spectrum representations well
mulSpectrums(tempA, tempB, tempA);

// transform the product back from the frequency domain.
// Even though all the result rows will be non-zero,
// we need only the first C.rows of them, and thus we
// pass nonzeroRows == C.rows
dft(tempA, tempA, DFT_INVERSE + DFT_SCALE, C.rows);

// now copy the result back to C.
tempA(Rect(0, 0, C.cols, C.rows)).copyTo(C);

// all the temporary buffers will be deallocated automaticall
}

```

---

What can be optimized in the above sample?

- since we passed `nonzeroRows  $\neq$  0` to the forward transform calls and since we copied A/B to the top-left corners of `tempA/tempB`, respectively, it's not necessary to clear the whole `tempA` and `tempB`; it is only necessary to clear the `tempA.cols - A.cols` (`tempB.cols - B.cols`) rightmost columns of the matrices.
- this DFT-based convolution does not have to be applied to the whole big arrays, especially if B is significantly smaller than A or vice versa. Instead, we can compute convolution by parts. For that we need to split the destination array C into multiple tiles and for each tile estimate, which parts of A and B are required to compute convolution in this tile. If the tiles in C are too small, the speed will decrease a lot, because of repeated work - in the ultimate case, when each tile in C is a single pixel, the algorithm becomes equivalent to the naive convolution algorithm. If the tiles are too big, the temporary arrays `tempA` and `tempB` become too big and there is also slowdown because of bad cache locality. So there is optimal tile size somewhere in the middle.
- if the convolution is done by parts, since different tiles in C can be computed in parallel, the loop can be threaded.

All of the above improvements have been implemented in [matchTemplate](#) and [filter2D](#), therefore, by using them, you can get even better performance than with the above theoretically optimal implementation (though, those two functions actually compute cross-correlation, not convolution, so you will need to “flip” the kernel or the image around the center using [flip](#)).

See also: [dct](#), [getOptimalDFTSize](#), [mulSpectrums](#), [filter2D](#), [matchTemplate](#), [flip](#), [cartToPolar](#), [magnitude](#), [phase](#)

## divide

---

void **divide**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, double *scale*=1)

void **divide**(double *scale*, const Mat& *src2*, Mat& *dst*)

void **divide**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*, double *scale*=1)

void **divide**(double *scale*, const MatND& *src2*, MatND& *dst*)

Performs per-element division of two arrays or a scalar by an array.

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array; should have the same size and same type as *src1*
  - *scale* – Scale factor
  - *dst* – The destination array; will have the same size and same type as *src2*

The functions `divide` divide one array by another:

$$\text{dst}(I) = \text{saturate}(\text{src1}(I) * \text{scale} / \text{src2}(I))$$

or a scalar by array, when there is no *src1*:

$$\text{dst}(I) = \text{saturate}(\text{scale} / \text{src2}(I))$$

The result will have the same type as *src1*. When  $\text{src2}(I) = 0$ ,  $\text{dst}(I) = 0$  too.

See also: [multiply](#), [add](#), [subtract](#), [Matrix Expressions](#)

## determinant

---

double **determinant**(const Mat& *mtx*)

Returns determinant of a square floating-point matrix.

**Parameter:** *mtx* – The input matrix; must have **CV\_32FC1** or **CV\_64FC1**

### type and square size

The function `determinant` computes and returns determinant of the specified matrix. For small matrices (`mtx.cols=mtx.rows<=3`) the direct method is used; for larger matrices the function uses LU factorization.

For symmetric positive-determined matrices, it is also possible to compute [SVD](#):  $\mathbf{mtx} = \mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T$  and then calculate the determinant as a product of the diagonal elements of  $\mathbf{W}$ .

See also: [SVD](#), [trace](#), [invert](#), [solve](#), [Matrix Expressions](#)

## eigen

bool **eigen**(const Mat& *src*, Mat& *eigenvalues*, int *lowindex*=-1, int *highindex*=-1)

bool **eigen**(const Mat& *src*, Mat& *eigenvalues*, Mat& *eigenvectors*, int *lowindex*=-1, int *highindex*=-1)

Computes eigenvalues and eigenvectors of a symmetric matrix.

- Parameters:**
- *src* – The input matrix; must have **CV\_32FC1** or **CV\_64FC1** type, square size and be symmetric:  $\mathbf{src}^T = \mathbf{src}$
  - *eigenvalues* – The output vector of eigenvalues of the same type as *src*; The eigenvalues are stored in the descending order.
  - *eigenvectors* – The output matrix of eigenvectors; It will have the same size and the same type as *src*; The eigenvectors are stored as subsequent matrix rows, in the same order as the corresponding eigenvalues
  - *lowindex* – Optional index of largest eigenvalue/-vector to calculate. (See below.)
  - *highindex* – Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The functions `eigen` compute just eigenvalues, or eigenvalues and eigenvectors of symmetric matrix *src*:

---

```
src*eigenvectors(i,:) = eigenvalues(i)*eigenvectors(i,:)
```

---

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. 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.

See also: [SVD](#), [completeSymm](#), [PCA](#)

## exp

---

void **exp**(const Mat& *src*, Mat& *dst*)

void **exp**(const MatND& *src*, MatND& *dst*)

Calculates the exponent of every array element.

**Parameters:**

- *src* – The source array
- *dst* – The destination array; will have the same size and same type as `src`

The function `exp` calculates the exponent of every element of the input array:

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

The maximum relative error is about  $7 \times 10^{-6}$  for single-precision and less than  $10^{-10}$  for double-precision. Currently, the function converts denormalized values to zeros on output. Special values (NaN,  $\pm\infty$ ) are not handled.

See also: [log](#), [cartToPolar](#), [polarToCart](#), [phase](#), [pow](#), [sqrt](#), [magnitude](#)

## extractImageCOI

---

void **extractImageCOI**(const CvArr\* *src*, Mat& *dst*, int *coi*=-1)

Extract the selected image channel

**Parameters:**

- *src* – The source array. It should be a pointer to `CvMat` or `IplImage`
- *dst* – The destination array; will have single-channel, and the same size and the same depth as `src`
- *coi* – If the parameter is  $\geq 0$ , it specifies the channel to extract; If it is  $< 0$ , `src` must be a pointer to `IplImage` with valid COI set - then the selected COI is extracted.

The function `extractImageCOI` is used to extract image COI from an old-style array and put the result to the new-style C++ matrix. As usual, the destination matrix is reallocated using `Mat::create` if needed.

To extract a channel from a new-style matrix, use [mixChannels](#) or [split](#)

See also: [mixChannels](#), [split](#), [merge](#), [cvarrToMat](#), [cvSetImageCOI](#), [cvGetImageCOI](#)

## fastAtan2

---

float **fastAtan2**(float y, float x)

Calculates the angle of a 2D vector in degrees

**Parameters:**

- *x* – x-coordinate of the vector
- *y* – y-coordinate of the vector

The function `fastAtan2` calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from  $0^\circ$  to  $360^\circ$ . The accuracy is about  $0.3^\circ$ .

## flip

---

void **flip**(const Mat& src, Mat& dst, int flipCode)

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

**Parameters:**

- *src* – The source array
- *dst* – The destination array; will have the same size and same type as *src*
- *flipCode* – 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 `flip` flips the array in one of three different ways (row and column indices are 0-based):

$$\text{dst}_{ij} = \begin{cases} \text{src}_{\text{src.rows}-i-1,j} & \text{if flipCode} = 0 \\ \text{src}_{i,\text{src.cols}-j-1} & \text{if flipCode} \neq 0 \\ \text{src}_{\text{src.rows}-i-1,\text{src.cols}-j-1} & \text{if flipCode} \neq 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (`flipCode = 0`) to switch between top-left and bottom-left image origin, which is a typical operation in video processing in Windows.
- horizontal flipping of the image with subsequent horizontal shift and

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

See also: [transpose](#), [repeat](#), [completeSymm](#)

## gemm

---

void **gemm**(const Mat& *src1*, const Mat& *src2*, double *alpha*, const Mat& *src3*, double *beta*, Mat& *dst*, int *flags*=0)

Performs generalized matrix multiplication.

- Parameters:**
- src1* – The first multiplied input matrix; should have `CV_32FC1`, `CV_64FC1`, `CV_32FC2` or `CV_64FC2` type
  - src2* – The second multiplied input matrix; should have the same type as *src1*
  - alpha* – The weight of the matrix product
  - src3* – The third optional delta matrix added to the matrix product; should have the same type as *src1* and *src2*
  - beta* – The weight of *src3*
  - dst* – The destination matrix; It will have the proper size and the same type as input matrices
  - flags* –
    - Operation flags:
      - `GEMM_1_T` - transpose *src1*
      - `GEMM_2_T` - transpose *src2*
      - `GEMM_3_T` - transpose *src3*

The function performs generalized matrix multiplication and similar to the corresponding functions `*gemm` in BLAS level 3. For example, `gemm(src1, src2, alpha, src3, beta, dst, GEMM_1_T + GEMM_3_T)` corresponds to

$$dst = alpha \cdot src1^T \cdot src2 + beta \cdot src3^T$$

The function can be replaced with a matrix expression, e.g. the above call can be replaced with:

---

```
dst = alpha*src1.t()*src2 + beta*src3.t();
```

---

See also: [mulTransposed](#), [transform](#), [Matrix Expressions](#)

## getConvertElem

---

**ConvertData** **getConvertElem**(int *fromType*, int *toType*)

**ConvertScaleData** **getConvertScaleElem**(int *fromType*, int *toType*)

```
typedef void (*ConvertData) (const void* from, void* to, int cn)
```

```
typedef void (*ConvertScaleData) (const void* from, void* to, int
cn, double alpha, double beta)
```

Returns conversion function for a single pixel

- Parameters:**
- *fromType* – The source pixel type
  - *toType* – The destination pixel type
  - *from* – Callback parameter: pointer to the input pixel
  - *to* – Callback parameter: pointer to the output pixel
  - *cn* – Callback parameter: the number of channels; can be arbitrary, 1, 100, 100000, ...
  - *alpha* – ConvertScaleData callback optional parameter: the scale factor
  - *beta* – ConvertScaleData callback optional parameter: the delta or offset

The functions `getConvertElem` and `getConvertScaleElem` return pointers to the functions for converting individual pixels from one type to another. While the main function purpose is to convert single pixels (actually, for converting sparse matrices from one type to another), you can use them to convert the whole row of a dense matrix or the whole matrix at once, by setting `cn = matrix.cols*matrix.rows*matrix.channels()` if the matrix data is continuous.

See also: `Mat::convertTo`, `MatND::convertTo`, `SparseMat::convertTo`

## getOptimalDFTSize

---

int **getOptimalDFTSize**(int *vecsize*)

Returns optimal DFT size for a given vector size.

**Parameter:** *vecsize* – Vector size

DFT performance is not a monotonic function of a vector size, therefore, when you compute convolution of two arrays or do a spectral analysis of array, it usually makes sense to pad the input data with zeros to get a bit larger array that can be transformed much faster than the original one. Arrays, which size is a power-of-two (2, 4, 8, 16, 32, ...) are the fastest to process, though, the arrays, which size is a product of 2's, 3's and 5's (e.g.  $300 = 5 \cdot 5 \cdot 3 \cdot 2 \cdot 2$ ), are also processed quite efficiently.

The function `getOptimalDFTSize` returns the minimum number  $N$  that is greater than or equal to *vecsize*, such that the DFT of a vector of size  $N$  can be computed efficiently. 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 *vecsize* is too large (very close to `INT_MAX`).

While the function cannot be used directly to estimate the optimal vector size for DCT transform (since the current DCT implementation supports only even-size vectors), it can be easily computed as `getOptimalDFTSize((vecsize+1)/2)*2`.

See also: [dft](#), [dct](#), [idft](#), [idct](#), [mulSpectrums](#)

## idct

---

void **idct**(const Mat& *src*, Mat& *dst*, int *flags*=0)

Computes inverse Discrete Cosine Transform of a 1D or 2D array

**Parameters:**

- *src* – The source floating-point single-channel array
- *dst* – The destination array. Will have the same size and same type as *src*
- *flags* – The operation flags.

`idct(src, dst, flags)` is equivalent to `dct(src, dst, flags | DCT_INVERSE)`. See [dct](#) for details.

See also: [dct](#), [dft](#), [idft](#), [getOptimalDFTSize](#)

## idft

---

void **idft**(const Mat& *src*, Mat& *dst*, int *flags*=0, int *outputRows*=0)

Computes inverse Discrete Fourier Transform of a 1D or 2D array

- Parameters:**
- *src* – The source floating-point real or complex array
  - *dst* – The destination array, which size and type depends on the *flags*
  - *flags* – The operation flags. See [dft](#)
  - *nonzeroRows* – The number of *dst* rows to compute. The rest of the rows will have undefined content. See the convolution sample in [dft](#) description

`idft(src, dst, flags)` is equivalent to `dct(src, dst, flags | DFT_INVERSE)`. See [dft](#) for details. Note, that none of `dft` and `idft` scale the result by default. Thus, you should pass `DFT_SCALE` to one of `dft` or `idft` explicitly to make these transforms mutually inverse.

See also: [dft](#), [dct](#), [idct](#), [mulSpectrums](#), [getOptimalDFTSize](#)

## inRange

---

void **inRange**(const Mat& *src*, const Mat& *lowerb*, const Mat& *upperb*, Mat& *dst*)

void **inRange**(const Mat& *src*, const Scalar& *lowerb*, const Scalar& *upperb*, Mat& *dst*)

void **inRange**(const MatND& *src*, const MatND& *lowerb*, const MatND& *upperb*, MatND& *dst*)

void **inRange**(const MatND& *src*, const Scalar& *lowerb*, const Scalar& *upperb*, MatND& *dst*)

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

- Parameters:**
- *src* – The first source array
  - *lowerb* – The inclusive lower boundary array of the same size and type as *src*
  - *upperb* – The exclusive upper boundary array of the same size and type as *src*
  - *dst* – The destination array, will have the same size as *src* and `CV_8U` type

The functions `inRange` do the range check for every element of the input

array:

$$\text{dst}(I) = \text{lowerb}(I)_0 \leq \text{src}(I)_0 < \text{upperb}(I)_0$$

for single-channel arrays,

$$\text{dst}(I) = \text{lowerb}(I)_0 \leq \text{src}(I)_0 < \text{upperb}(I)_0 \wedge \text{lowerb}(I)_1 \leq \text{src}(I)_1 < \text{upp}$$

for two-channel arrays and so forth. `dst(I)` is set to 255 (all 1-bits) if `src(I)` is within the specified range and 0 otherwise.

## invert

---

double **invert**(const Mat& *src*, Mat& *dst*, int *method*=*DECOMP\_LU*)

Finds the inverse or pseudo-inverse of a matrix

- Parameters:**
- *src* – The source floating-point  $M \times N$  matrix
  - *dst* – The destination matrix; will have  $N \times M$  size and the same type as *src*
  - *flags* –  
The inversion method :
    - **DECOMP\_LU** - Gaussian elimination with optimal pivot element chosen
    - **DECOMP\_SVD** - Singular value decomposition (SVD) method
    - **DECOMP\_CHOLESKY** - Cholesky decomposition. The matrix must be symmetrical and positively defined

The function `invert` inverts matrix `src` and stores the result in `dst`. When the matrix `src` is singular or non-square, the function computes the pseudo-inverse matrix, i.e. the matrix `dst`, such that  $\|\text{src} \cdot \text{dst} - I\|$  is minimal.

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

In the case of `DECOMP_SVD` method, the function returns the inversed condition number of `src` (the ratio of the smallest singular value to the largest singular value) and 0 if `src` is singular. The SVD method calculates a pseudo-inverse matrix if `src` is singular.

Similarly to `DECOMP_LU`, the method `DECOMP_CHOLESKY` works only with non-singular square matrices. In this case the function stores the inverted matrix in `dst` and returns non-zero, otherwise it returns 0.

See also: [solve](#), [SVD](#)

## log

---

void **log**(const Mat& *src*, Mat& *dst*)

void **log**(const MatND& *src*, MatND& *dst*)

Calculates the natural logarithm of every array element.

**Parameters:**

- *src* – The source array
- *dst* – The destination array; will have the same size and same type as *src*

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

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

Where  $C$  is a large negative number (about  $-700$  in the current implementation). The maximum relative error is about  $7 \times 10^{-6}$  for single-precision input and less than  $10^{-10}$  for double-precision input. Special values (NaN,  $\pm\infty$ ) are not handled.

See also: [exp](#), [cartToPolar](#), [polarToCart](#), [phase](#), [pow](#), [sqrt](#), [magnitude](#)

## LUT

---

void **LUT**(const Mat& *src*, const Mat& *lut*, Mat& *dst*)

Performs a look-up table transform of an array.

**Parameters:**

- *src* – Source array of 8-bit elements
- *lut* – Look-up table of 256 elements. In the case of multi-channel source array, 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 in the source array
- *dst* – Destination array; will have the same size and the same number of channels as *src*, and the same depth as *lut*

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

$$\text{dst}(I) \leftarrow \text{lut}(\text{src}(I) + d)$$

where

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

See also: [convertScaleAbs](#), `Mat::convertTo`

## magnitude

---

void **magnitude**(const Mat& *x*, const Mat& *y*, Mat& *magnitude*)

Calculates magnitude of 2D vectors.

- Parameters:**
- *x* – The floating-point array of x-coordinates of the vectors
  - *y* – The floating-point array of y-coordinates of the vectors; must have the same size as *x*
  - *dst* – The destination array; will have the same size and same type as *x*

The function `magnitude` calculates magnitude of 2D vectors formed from the corresponding elements of *x* and *y* arrays:

$$\text{dst}(I) = \sqrt{x(I)^2 + y(I)^2}$$

See also: [cartToPolar](#), [polarToCart](#), [phase](#), [sqrt](#)

## Mahalanobis

---

double **Mahalanobis**(const Mat& *vec1*, const Mat& *vec2*, const Mat& *icovar*)

Calculates the Mahalanobis distance between two vectors.

- Parameters:**
- *vec1* – The first 1D source vector
  - *vec2* – The second 1D source vector
  - *icovar* – The inverse covariance matrix

The function `cvMahalanobis()` calculates and returns the weighted distance between two vectors:

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

The covariance matrix may be calculated using the [calcCovarMatrix](#) function and then inverted using the [invert](#) function (preferably using `DECOMP_SVD` method, as the most accurate).

## max

---

Mat\_Expr<...> **max**(const Mat& *src1*, const Mat& *src2*)

Mat\_Expr<...> **max**(const Mat& *src1*, double *value*)

Mat\_Expr<...> **max**(double *value*, const Mat& *src1*)

void **max**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*)

void **max**(const Mat& *src1*, double *value*, Mat& *dst*)

void **max**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*)

void **max**(const MatND& *src1*, double *value*, MatND& *dst*)

Calculates per-element maximum of two arrays or array and a scalar

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array of the same size and type as *src1*
  - *value* – The real scalar value
  - *dst* – The destination array; will have the same size and type as *src1*

The functions `max` compute per-element maximum of two arrays:

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

or array and a scalar:

$$\text{dst}(I) = \max(\text{src1}(I), \text{value})$$

In the second variant, when the source array is multi-channel, each channel is compared with `value` independently.

The first 3 variants of the function listed above are actually a part of *Matrix Expressions*, they return the expression object that can be further transformed, or assigned to a matrix, or passed to a function etc.

See also: [min](#), [compare](#), [inRange](#), [minMaxLoc](#), [Matrix Expressions](#)

## mean

---

Scalar **mean**(const Mat& *mtx*)

Scalar **mean**(const Mat& *mtx*, const Mat& *mask*)

Scalar **mean**(const MatND& *mtx*)

Scalar **mean**(const MatND& *mtx*, const MatND& *mask*)

Calculates average (mean) of array elements

**Parameters:**

- *mtx* – The source array; it should have 1 to 4 channels (so that the result can be stored in *Scalar*)
- *mask* – The optional operation mask

The functions `mean` compute mean value  $M$  of array elements, independently for each channel, and return it:

$$N = \sum_{I: \text{mask}(I) \neq 0} 1$$

$$M_c = \left( \sum_{I: \text{mask}(I) \neq 0} \text{mtx}(I)_c \right) / N$$

When all the mask elements are 0's, the functions return `Scalar::all(0)`.

See also: [countNonZero](#), [meanStdDev](#), [norm](#), [minMaxLoc](#)

## meanStdDev

---

void **meanStdDev**(const Mat& *mtx*, Scalar& *mean*, Scalar& *stddev*, const Mat& *mask=Mat()*)

void **meanStdDev**(const MatND& *mtx*, Scalar& *mean*, Scalar& *stddev*, const MatND& *mask=MatND()*)

Calculates mean and standard deviation of array elements

**Parameters:**

- *mtx* – The source array; it should have 1 to 4 channels (so that

the results can be stored in *Scalar*'s)

- *mean* – The output parameter: computed mean value
- *stddev* – The output parameter: computed standard deviation
- *mask* – The optional operation mask

The functions `meanStdDev` compute the mean and the standard deviation  $M$  of array elements, independently for each channel, and return it via the output parameters:

$$N = \sum_{I, \text{mask}(I) \neq 0} 1$$

$$\text{mean}_c = \frac{\sum_{I: \text{mask}(I) \neq 0} \text{src}(I)_c}{N}$$

$$\text{stddev}_c = \sqrt{\sum_{I: \text{mask}(I) \neq 0} (\text{src}(I)_c - \text{mean}_c)^2}$$

When all the mask elements are 0's, the functions return `mean=stddev=Scalar::all(0)`. Note that the computed standard deviation is only the diagonal of the complete normalized covariance matrix. If the full matrix is needed, you can reshape the multi-channel array  $M \times N$  to the single-channel array  $M * N \times \text{mtx.channels}()$  (only possible when the matrix is continuous) and then pass the matrix to [calcCovarMatrix](#).

See also: [countNonZero](#), [mean](#), [norm](#), [minMaxLoc](#), [calcCovarMatrix](#)

## merge

---

```
void merge(const Mat* mv, size_t count, Mat& dst)
```

```
void merge(const vector<Mat>& mv, Mat& dst)
```

```
void merge(const MatND* mv, size_t count, MatND& dst)
```

```
void merge(const vector<MatND>& mv, MatND& dst)
```

Composes a multi-channel array from several single-channel arrays.

- Parameters:**
- *mv* – The source array or vector of the single-channel matrices to be merged. All the matrices in `mv` must have the same size and the same type
  - *count* – The number of source matrices when `mv` is a plain C array; must be greater than zero
  - *dst* – The destination array; will have the same size and the same depth as `mv[0]`, the number of channels will match

the number of source matrices

The functions `merge` merge several single-channel arrays (or rather interleave their elements) to make a single multi-channel array.

$$\text{dst}(I)_c = \text{mv}[c](I)$$

The function `split` does the reverse operation and if you need to merge several multi-channel images or shuffle channels in some other advanced way, use `mixChannels`

See also: `mixChannels`, `split`, `reshape`

## min

---

`Mat_Expr<...> min(const Mat& src1, const Mat& src2)`

`Mat_Expr<...> min(const Mat& src1, double value)`

`Mat_Expr<...> min(double value, const Mat& src1)`

`void min(const Mat& src1, const Mat& src2, Mat& dst)`

`void min(const Mat& src1, double value, Mat& dst)`

`void min(const MatND& src1, const MatND& src2, MatND& dst)`

`void min(const MatND& src1, double value, MatND& dst)`

Calculates per-element minimum of two arrays or array and a scalar

- Parameters:**
- `src1` – The first source array
  - `src2` – The second source array of the same size and type as `src1`
  - `value` – The real scalar value
  - `dst` – The destination array; will have the same size and type as `src1`

The functions `min` compute per-element minimum of two arrays:

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

or array and a scalar:

$$\text{dst}(I) = \min(\text{src1}(I), \text{value})$$

In the second variant, when the source array is multi-channel, each channel is compared with `value` independently.

The first 3 variants of the function listed above are actually a part of *Matrix Expressions*, they return the expression object that can be further transformed, or assigned to a matrix, or passed to a function etc.

See also: [max](#), [compare](#), [inRange](#), [minMaxLoc](#), [Matrix Expressions](#)

## minMaxLoc

---

```
void minMaxLoc(const Mat& src, double* minVal, double* maxVal=0, Point* minLoc=0, Point* maxLoc=0, const Mat& mask=Mat())
```

```
void minMaxLoc(const MatND& src, double* minVal, double* maxVal, int* minIdx=0, int* maxIdx=0, const MatND& mask=MatND())
```

```
void minMaxLoc(const SparseMat& src, double* minVal, double* maxVal, int* minIdx=0, int* maxIdx=0)
```

Finds global minimum and maximum in a whole array or sub-array

- Parameters:**
- `src` – The source single-channel array
  - `minVal` – Pointer to returned minimum value; NULL if not required
  - `maxVal` – Pointer to returned maximum value; NULL if not required
  - `minLoc` – Pointer to returned minimum location (in 2D case); NULL if not required
  - `maxLoc` – Pointer to returned maximum location (in 2D case); NULL if not required
  - `minIdx` – Pointer to returned minimum location (in nD case); NULL if not required, otherwise must point to an array of `src.dims` elements and the coordinates of minimum element in each dimensions will be stored sequentially there.
  - `maxIdx` – Pointer to returned maximum location (in nD case); NULL if not required

- *mask* – The optional mask used to select a sub-array

The functions `minMaxLoc` find minimum and maximum element values and their positions. The extremums are searched across the whole array, or, if `mask` is not an empty array, in the specified array region.

The functions do not work with multi-channel arrays. If you need to find minimum or maximum elements across all the channels, use *reshape* first to reinterpret the array as single-channel. Or you may extract the particular channel using [extractImageCOI](#) or [mixChannels](#) or [split](#).

in the case of a sparse matrix the minimum is found among non-zero elements only.

See also: [max](#), [min](#), [compare](#), [inRange](#), [extractImageCOI](#), [mixChannels](#), [split](#), [reshape](#).

## mixChannels

---

```
void mixChannels(const Mat* srcv, int nsrc, Mat* dstv, int ndst, const int* fromTo, size_t npairs)
```

```
void mixChannels(const MatND* srcv, int nsrc, MatND* dstv, int ndst, const int* fromTo, size_t npairs)
```

```
void mixChannels(const vector<Mat>& srcv, vector<Mat>& dstv, const int* fromTo, int npairs)
```

```
void mixChannels(const vector<MatND>& srcv, vector<MatND>& dstv, const int* fromTo, int npairs)
```

Copies specified channels from input arrays to the specified channels of output arrays

- Parameters:**
- *srcv* – The input array or vector of matrices. All the matrices must have the same size and the same depth
  - *nsrc* – The number of elements in *srcv*
  - *dstv* – The output array or vector of matrices. All the matrices must be allocated, their size and depth must be the same as in `srcv[0]`
  - *ndst* – The number of elements in *dstv*
  - *fromTo* – The array of index pairs, specifying which channels

are copied and where. `fromTo[k*2]` is the 0-based index of the input channel in `srcv` and `fromTo[k*2+1]` is the index of the output channel in `dstv`. Here the continuous channel numbering is used, that is, the first input image channels are indexed from 0 to `srcv[0].channels() - 1`, the second input image channels are indexed from `srcv[0].channels()` to `srcv[0].channels() + srcv[1].channels() - 1` etc., and the same scheme is used for the output image channels. As a special case, when `fromTo[k*2]` is negative, the corresponding output channel is filled with zero. `npairs`bgroup``{The number of pairs. In the latter case the parameter is not passed explicitly, but computed as `texttt{srcv.size()} (=texttt{dstv.size()})`}}

The functions `mixChannels` provide an advanced mechanism for shuffling image channels. [split](#) and [merge](#) and some forms of [cvtColor](#) are partial cases of `mixChannels`.

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

---

```
Mat rgba( 100, 100, CV_8UC4, Scalar(1,2,3,4) );
Mat bgr( rgba.rows, rgba.cols, CV_8UC3 );
Mat alpha( rgba.rows, rgba.cols, CV_8UC1 );

// forming array of matrices is quite efficient operations
// because the matrix data is not copied, only the headers
Mat out[] = { bgr, alpha };
// rgba[0] -> bgr[2], rgba[1] -> bgr[1],
// rgba[2] -> bgr[0], rgba[3] -> alpha[0]
int from_to[] = { 0,2, 1,1, 2,0, 3,3 };
mixChannels( &rgba, 1, out, 2, from_to, 4 );
```

---

Note that, unlike many other new-style C++ functions in OpenCV (see the introduction section and `Mat::create`), `mixChannels` requires the destination arrays be pre-allocated before calling the function.

See also: [split](#), [merge](#), [cvtColor](#)

## mulSpectrums

---

void **mulSpectrums**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, int *flags*, bool *conj=false*)

Performs per-element multiplication of two Fourier spectrums.

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array; must have the same size and the same type as *src1*
  - *dst* – The destination array; will have the same size and the same type as *src1*
  - *flags* – The same flags as passed to [dft](#); only the flag `DFT_ROWS` is checked for
  - *conj* – The optional flag that conjugate the second source array before the multiplication (true) or not (false)

The function `mulSpectrums` 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](#) and [idft](#), may be used to calculate convolution (pass `conj=false`) or correlation (pass `conj=true`) of two arrays rapidly. When the arrays are complex, they are simply multiplied (per-element) with optional conjugation of the second array elements. When the arrays are real, they assumed to be CCS-packed (see [dft](#) for details).

## multiply

---

```
void multiply(const Mat& src1, const Mat& src2, Mat& dst, double scale=1)
```

```
void multiply(const MatND& src1, const MatND& src2, MatND& dst, double scale=1)
```

Calculates the per-element scaled product of two arrays

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array of the same size and the same type as *src1*
  - *dst* – The destination array; will have the same size and the same type as *src1*
  - *scale* – The optional scale factor

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

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

There is also *Matrix Expressions*-friendly variant of the first function, see `Mat::mul`.

If you are looking for a matrix product, not per-element product, see [gemm](#).

See also: [add](#), [subtract](#), [divide](#), [Matrix Expressions](#), [scaleAdd](#), [addWeighted](#), [accumulate](#), [accumulateProduct](#), [accumulateSquare](#), [Mat::convertTo](#)

## mulTransposed

---

void **mulTransposed**(const Mat& *src*, Mat& *dst*, bool *aTa*, const Mat& *delta*=Mat(), double *scale*=1, int *rtype*=-1)

Calculates the product of a matrix and its transposition.

- Parameters:**
- *src* – The source matrix
  - *dst* – The destination square matrix
  - *aTa* – Specifies the multiplication ordering; see the description below
  - *delta* – The optional delta matrix, subtracted from *src* before the multiplication. When the matrix is empty (*delta*=Mat()), it's assumed to be zero, i.e. nothing is subtracted, otherwise if it has the same size as *src*, then it's simply subtracted, otherwise it is "repeated" (see [repeat](#)) to cover the full *src* and then subtracted. Type of the delta matrix, when it's not empty, must be the same as the type of created destination matrix, see the *rtype* description
  - *scale* – The optional scale factor for the matrix product
  - *rtype* – When it's negative, the destination matrix will have the same type as *src*. Otherwise, it will have `type=CV_MAT_DEPTH(rtype)`, which should be either **CV\_32F** or **CV\_64F**

The function `mulTransposed` calculates the product of *src* and its transposition:

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

if *aTa*=true, and

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

otherwise. The function is used to compute covariance matrix and with zero delta can be used as a faster substitute for general matrix product  $A * B$  when  $B = A^T$ .

See also: [calcCovarMatrix](#), [gemm](#), [repeat](#), [reduce](#)

## norm

---

```
double norm(const Mat& src1, int normType=NORM_L2)
```

```
double norm(const Mat& src1, const Mat& src2, int normType=NORM_L2)
```

```
double norm(const Mat& src1, int normType, const Mat& mask)
```

```
double norm(const Mat& src1, const Mat& src2, int normType, const Mat& mask)
```

```
double norm(const MatND& src1, int normType=NORM_L2, const MatND&
mask=MatND())
```

```
double norm(const MatND& src1, const MatND& src2, int normType=NORM_L2,
const MatND& mask=MatND())
```

```
double norm(const SparseMat& src, int normType)
```

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

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array of the same size and the same type as *src1*
  - *normType* – Type of the norm; see the discussion below
  - *mask* – The optional operation mask

The functions `norm` calculate the absolute norm of *src1* (when there is no *src2*):

$$\mathit{norm} = \begin{cases} \|\mathit{src1}\|_{L_\infty} = \max_I |\mathit{src1}(I)| & \text{if } \mathit{normType} = \text{NORM\_INF} \\ \|\mathit{src1}\|_{L_1} = \sum_I |\mathit{src1}(I)| & \text{if } \mathit{normType} = \text{NORM\_L1} \\ \|\mathit{src1}\|_{L_2} = \sqrt{\sum_I \mathit{src1}(I)^2} & \text{if } \mathit{normType} = \text{NORM\_L2} \end{cases}$$

or an absolute or relative difference norm if *src2* is there:

$$\mathit{norm} = \begin{cases} \|\mathit{src1} - \mathit{src2}\|_{L_\infty} = \max_I |\mathit{src1}(I) - \mathit{src2}(I)| & \text{if } \mathit{normType} = \text{NORM\_DIFF\_LINF} \\ \|\mathit{src1} - \mathit{src2}\|_{L_1} = \sum_I |\mathit{src1}(I) - \mathit{src2}(I)| & \text{if } \mathit{normType} = \text{NORM\_DIFF\_L1} \\ \|\mathit{src1} - \mathit{src2}\|_{L_2} = \sqrt{\sum_I (\mathit{src1}(I) - \mathit{src2}(I))^2} & \text{if } \mathit{normType} = \text{NORM\_DIFF\_L2} \end{cases}$$

or

$$norm = \begin{cases} \frac{\|src1 - src2\|_{L_\infty}}{\|src2\|_{L_\infty}} & \text{if } normType = NORM\_RELATIVE\_INF \\ \frac{\|src1 - src2\|_{L_1}}{\|src2\|_{L_1}} & \text{if } normType = NORM\_RELATIVE\_L1 \\ \frac{\|src1 - src2\|_{L_2}}{\|src2\|_{L_2}} & \text{if } normType = NORM\_RELATIVE\_L2 \end{cases}$$

The functions `norm` return the calculated norm.

When there is `mask` parameter, and it is not empty (then it should have type `CV_8U` and the same size as `src1`), the norm is computed only over the specified by the mask region.

A multiple-channel source arrays are treated as a single-channel, that is, the results for all channels are combined.

## normalize

---

```
void normalize(const Mat& src, Mat& dst, double alpha=1, double beta=0,
int normType=NORM_L2, int rtype=-1, const Mat& mask=Mat())
```

```
void normalize(const MatND& src, MatND& dst, double alpha=1, double
beta=0, int normType=NORM_L2, int rtype=-1, const MatND& mask=MatND())
```

```
void normalize(const SparseMat& src, SparseMat& dst, double alpha, int
normType)
```

Normalizes array's norm or the range

- Parameters:**
- `src` – The source array
  - `dst` – The destination array; will have the same size as `src`
  - `alpha` – The norm value to normalize to or the lower range boundary in the case of range normalization
  - `beta` – The upper range boundary in the case of range normalization; not used for norm normalization
  - `normType` – The normalization type, see the discussion
  - `rtype` – When the parameter is negative, the destination array will have the same type as `src`, otherwise it will have the same number of channels as `src` and the depth ``=CV\_MAT\_DEPTH(rtype)``
  - `mask` – The optional operation mask

The functions `normalize` scale and shift the source array elements, so that

$$\|dst\|_{L_p} = \mathbf{alpha}$$

(where  $p = \infty, 1$  or  $2$ ) when `normType=NORM_INF`, `NORM_L1` or `NORM_L2`, or so that

$$\min_I \text{dst}(I) = \text{alpha}, \max_I \text{dst}(I) = \text{beta}$$

when `normType=NORM_MINMAX` (for dense arrays only).

The optional mask specifies the sub-array to be normalize, that is, the norm or min-n-max are computed over the sub-array and then this sub-array is modified to be normalized. If you want to only use the mask to compute the norm or min-max, but modify the whole array, you can use [norm](#) and [Mat::convertScale/MatND::convertScale/crossbgroup\({SparseMat::convertScale}\)](#) separately.

in the case of sparse matrices, only the non-zero values are analyzed and transformed. Because of this, the range transformation for sparse matrices is not allowed, since it can shift the zero level.

See also: [norm](#), [Mat::convertScale](#), [MatND::convertScale](#), [SparseMat::convertScale](#)

## PCA

---

### Class for Principal Component Analysis

```
class PCA
{
public:
    // default constructor
    PCA();
    // computes PCA for a set of vectors stored as data rows
    PCA(const Mat& data, const Mat& mean, int flags, int maxC);
    // computes PCA for a set of vectors stored as data rows
    PCA& operator()(const Mat& data, const Mat& mean, int flags);
    // projects vector into the principal components space
    Mat project(const Mat& vec) const;
    void project(const Mat& vec, Mat& result) const;
    // reconstructs the vector from its PC projection
    Mat backProject(const Mat& vec) const;
    void backProject(const Mat& vec, Mat& result) const;

    // eigenvectors of the PC space, stored as the matrix row
    Mat eigenvectors;
    // the corresponding eigenvalues; not used for PCA computation
    Mat eigenvalues;
    // mean vector, subtracted from the projected vector
    // or added to the reconstructed vector
    Mat mean;
```

```

    Mat mean;
};

```

---

The class `PCA` is used to compute the special basis for a set of vectors. The basis will consist of eigenvectors of the covariance matrix computed from the input set of vectors. And also the class `PCA` can transform vectors to/from the new coordinate space, defined by the basis. Usually, in this new coordinate system each vector from the original set (and any linear combination of such vectors) can be quite accurately approximated by taking just the first few its components, corresponding to the eigenvectors of the largest eigenvalues of the covariance matrix. Geometrically it means that we compute projection of the vector to a subspace formed by a few eigenvectors corresponding to the dominant eigenvalues of the covariation matrix. And usually such a projection is very close to the original vector. That is, we can represent the original vector from a high-dimensional space with a much shorter vector consisting of the projected vector's coordinates in the subspace. Such a transformation is also known as Karhunen-Loeve Transform, or KLT. See [http://en.wikipedia.org/wiki/Principal\\_component\\_analysis](http://en.wikipedia.org/wiki/Principal_component_analysis)

The following sample is the function that takes two matrices. The first one stores the set of vectors (a row per vector) that is used to compute PCA, the second one stores another “test” set of vectors (a row per vector) that are first compressed with PCA, then reconstructed back and then the reconstruction error norm is computed and printed for each vector.

---

```

PCA compressPCA(const Mat& pcaset, int maxComponents,
               const Mat& testset, Mat& compressed)
{
    PCA pca(pcaset, // pass the data
           Mat(), // we do not have a pre-computed mean vect
           // so let the PCA engine to compute it
           CV_PCA_DATA_AS_ROW, // indicate that the vectors
                               // are stored as matrix rows
                               // (use CV_PCA_DATA_AS_COL if
                               // the matrix columns)
           maxComponents // specify, how many principal comp
           );
    // if there is no test data, just return the computed bas
    if( !testset.data )
        return pca;
    CV_Assert( testset.cols == pcaset.cols );

    compressed.create(testset.rows, maxComponents, testset.ty

    Mat reconstructed;
    for( int i = 0; i < testset.rows; i++ )
    {
        Mat vec = testset.row(i), coeffs = compressed.row(i);
        // compress the vector, the result will be stored

```

```

        // in the i-th row of the output matrix
        pca.project(vec, coeffs);
        // and then reconstruct it
        pca.backProject(coeffs, reconstructed);
        // and measure the error
        printf("%d. diff = %g\n", i, norm(vec, reconstructed,
    }
    return pca;
}

```

See also: [calcCovarMatrix](#), [mulTransposed](#), [SVD](#), [dft](#), [dct](#)

## perspectiveTransform

void **perspectiveTransform**(const Mat& *src*, Mat& *dst*, const Mat& *mtx*)

Performs perspective matrix transformation of vectors.

- Parameters:**
- *src* – The source two-channel or three-channel floating-point array; each element is 2D/3D vector to be transformed
  - *dst* – The destination array; it will have the same size and same type as *src*
  - *mtx* –  $3 \times 3$  or  $4 \times 4$  transformation matrix

The function `perspectiveTransform` transforms every element of *src*, by treating it as 2D or 3D vector, in the following way (here 3D vector transformation is shown; in the case of 2D vector transformation the *z* component is omitted):

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

where

$$(x', y', z', w') = \text{mat} \cdot [x \ y \ z \ 1]$$

and

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

Note that the function transforms a sparse set of 2D or 3D vectors. If you want to transform an image using perspective transformation, use [warpPerspective](#). If you have an inverse task, i.e. want to compute the most probable perspective transformation out of several pairs of corresponding points, you can use [getPerspectiveTransform](#) or [findHomography](#).

See also: [transform](#), [warpPerspective](#), [getPerspectiveTransform](#), [findHomography](#)

## phase

---

void **phase**(const Mat& *x*, const Mat& *y*, Mat& *angle*, bool *angleInDegrees*=false)

Calculates the rotation angle of 2d vectors

- Parameters:**
- *x* – The source floating-point array of x-coordinates of 2D vectors
  - *y* – The source array of y-coordinates of 2D vectors; must have the same size and the same type as *x*
  - *angle* – The destination array of vector angles; it will have the same size and same type as *x*
  - *angleInDegrees* – When it is true, the function will compute angle in degrees, otherwise they will be measured in radians

The function `phase` computes the rotation angle of each 2D vector that is formed from the corresponding elements of *x* and *y*:

$$\text{angle}(I) = \text{atan2}(y(I), x(I))$$

The angle estimation accuracy is  $\sim 0.3^\circ$ , when  $x(I)=y(I)=0$ , the corresponding `angle(I)` is set to 0.

See also:

## polarToCart

---

void **polarToCart**(const Mat& *magnitude*, const Mat& *angle*, Mat& *x*, Mat& *y*, bool *angleInDegrees*=false)

Computes x and y coordinates of 2D vectors from their magnitude and angle.

- Parameters:**
- *magnitude* – The source floating-point array of magnitudes of 2D vectors. It can be an empty matrix (`=Mat()`) - in this case the function assumes that all the magnitudes are =1. If it's not empty, it must have the same size and same type as *angle*
  - *angle* – The source floating-point array of angles of the 2D vectors
  - *x* – The destination array of x-coordinates of 2D vectors; will have the same size and the same type as *angle*
  - *y* – The destination array of y-coordinates of 2D vectors; will have the same size and the same type as *angle*
  - *angleInDegrees* – When it is true, the input angles are measured in degrees, otherwise they are measured in radians

The function `polarToCart` computes the cartesian coordinates of each 2D vector represented by the corresponding elements of `magnitude` and `angle`:

$$\begin{aligned}x(I) &= \text{magnitude}(I) \cos(\text{angle}(I)) \\y(I) &= \text{magnitude}(I) \sin(\text{angle}(I))\end{aligned}$$

The relative accuracy of the estimated coordinates is  $\sim 10^{-6}$ .

See also: [cartToPolar](#), [magnitude](#), [phase](#), [exp](#), [log](#), [pow](#), [sqrt](#)

## pow

---

void **pow**(const Mat& *src*, double *p*, Mat& *dst*)

void **pow**(const MatND& *src*, double *p*, MatND& *dst*)

Raises every array element to a power.

- Parameters:**
- *src* – The source array
  - *p* – The exponent of power
  - *dst* – The destination array; will have the same size and the same type as *src*

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

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

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 5th root of array `src`, shows:

---

```
Mat mask = src < 0;
pow(src, 1./5, dst);
subtract(Scalar::all(0), dst, dst, mask);
```

---

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

See also: [sqrt](#), [exp](#), [log](#), [cartToPolar](#), [polarToCart](#)

## randu

---

template<typename \_Tp> \_Tp **randu**()

void **randu**(Mat& *mtx*, const Scalar& *low*, const Scalar& *high*)

Generates a single uniformly-distributed random number or array of random numbers

- Parameters:**
- *mtx* – The output array of random numbers. The array must be pre-allocated and have 1 to 4 channels
  - *low* – The inclusive lower boundary of the generated random numbers
  - *high* – The exclusive upper boundary of the generated random numbers

The template functions `randu` generate and return the next uniformly-distributed random value of the specified type. `randu<int>()` is equivalent to `(int)theRNG()`; etc. See *RNG* description.

The second non-template variant of the function fills the matrix `mtx` with uniformly-distributed random numbers from the specified range:

$$\text{low}_c \leq \text{mtx}(I)_c < \text{high}_c$$

See also: *RNG*, [randn](#), [theRNG](#).

## randn

---

void **randn**(Mat& *mtx*, const Scalar& *mean*, const Scalar& *stddev*)

Fills array with normally distributed random numbers

- Parameters:**
- *mtx* – The output array of random numbers. The array must be pre-allocated and have 1 to 4 channels
  - *mean* – The mean value (expectation) of the generated random numbers
  - *stddev* – The standard deviation of the generated random numbers

The function `randn` fills the matrix `mtx` with normally distributed random numbers with the specified mean and standard deviation. `[cppfunc.saturatecast] bgroup({saturate_ cast})` is applied to the generated numbers (i.e. the values are clipped)

See also: [RNG](#), [randu](#)

## randShuffle

---

void **randShuffle**(Mat& *mtx*, double *iterFactor*=1., RNG\* *rng*=0)

Shuffles the array elements randomly

- Parameters:**
- *mtx* – The input/output numerical 1D array
  - *iterFactor* – The scale factor that determines the number of random swap operations. See the discussion
  - *rng* – The optional random number generator used for shuffling. If it is zero, `:ref: `theRNG`()` is used instead

The function `randShuffle` shuffles the specified 1D array by randomly choosing pairs of elements and swapping them. The number of such swap operations will be `mtx.rows*mtx.cols*iterFactor`

See also: [RNG](#), [sort](#)

## reduce

---

void **reduce**(const Mat& *mtx*, Mat& *vec*, int *dim*, int *reduceOp*, int *dtype*=-1)

Reduces a matrix to a vector

- Parameters:**
- *mtx* – The source 2D matrix
  - *vec* – The destination vector. Its size and type is defined by `dim` and `dtype` parameters
  - *dim* – The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row and 1 means that the matrix is reduced to a single column
  - *reduceOp* –  
The reduction operation, one of:
    - **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.
  - *dtype* – When it is negative, the destination vector will have the same type as the source matrix, otherwise, its type will be

```
CV_MAKE_TYPE(CV_MAT_DEPTH(dtype) ,
mtx.channels())
```

The function `reduce` 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.

See also: [repeat](#)

## repeat

---

```
void repeat(const Mat& src, int ny, int nx, Mat& dst)
```

```
Mat repeat(const Mat& src, int ny, int nx)
```

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

- Parameters:**
- `src` – The source array to replicate
  - `dst` – The destination array; will have the same type as `src`
  - `ny` – How many times the `src` is repeated along the vertical axis
  - `nx` – How many times the `src` is repeated along the horizontal axis

The functions [repeat](#) duplicate the source array one or more times along each of the two axes:

$$dst_{ij} = src_{i \bmod src.rows, j \bmod src.cols}$$

The second variant of the function is more convenient to use with *Matrix Expressions*

See also: [reduce](#), *Matrix Expressions*

## saturate\_cast

---

Template function for accurate conversion from one primitive type to another

```
template<typename _Tp> inline _Tp saturate_cast(unsigned char v)
```

```
template<typename _Tp> inline _Tp saturate_cast(signed char v)
```

```
template<typename _Tp> inline _Tp saturate_cast(unsigned short v)
```

```
template<typename _Tp> inline _Tp saturate_cast(signed short v)
```

```
template<typename _Tp> inline _Tp saturate_cast(int v)
```

```
template<typename _Tp> inline _Tp saturate_cast(unsigned int v)
```

```
template<typename _Tp> inline _Tp saturate_cast(float v)
```

```
template<typename _Tp> inline _Tp saturate_cast(double v)
```

- **v** - The function parameter

The functions `saturate_cast` resembles the standard C++ cast operations, such as `static_cast<T>()` etc. They perform an efficient and accurate conversion from one primitive type to another, see the introduction. “saturate” in the name means that when the input value `v` is out of range of the target type, the result will not be formed just by taking low bits of the input, but instead the value will be clipped. For example:

```
uchar a = saturate_cast<uchar>(-100); // a = 0 (UCHAR_MIN)
short b = saturate_cast<short>(33333.33333); // b = 32767
```

Such clipping is done when the target type is `unsigned char`, `signed char`, `unsigned short` or `signed short` - for 32-bit integers no clipping is done.

When the parameter is floating-point value and the target type is an integer (8-, 16- or 32-bit), the floating-point value is first rounded to the nearest integer and then clipped if needed (when the target type is 8- or 16-bit).

This operation is used in most simple or complex image processing functions in OpenCV.

See also: [add](#), [subtract](#), [multiply](#), [divide](#), [Mat::convertTo](#)

## scaleAdd

---

void **scaleAdd**(const Mat& *src1*, double *scale*, const Mat& *src2*, Mat& *dst*)

void **scaleAdd**(const MatND& *src1*, double *scale*, const MatND& *src2*, MatND& *dst*)

Calculates the sum of a scaled array and another array.

- Parameters:**
- *src1* – The first source array
  - *scale* – Scale factor for the first array
  - *src2* – The second source array; must have the same size and the same type as *src1*
  - *dst* – The destination array; will have the same size and the same type as *src1*

The function **cvScaleAdd()** is one of the classical primitive linear algebra operations, known as DAXPY or SAXPY in bgroup ([http://en.wikipedia.org/wiki/Basic\\_Linear\\_Algebra\\_Subprograms](http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms))bgroup ({BLAS}). It calculates the sum of a scaled array and another array:

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

The function can also be emulated with a matrix expression, for example:

---

```
Mat A(3, 3, CV_64F);
...
A.row(0) = A.row(1)*2 + A.row(2);
```

---

See also: [add](#), [addWeighted](#), [subtract](#), [Mat::dot](#), [Mat::convertTo](#), [Matrix Expressions](#)

## setIdentity

---

void **setIdentity**(Mat& *dst*, const Scalar& *value=Scalar(1)*)

Initializes a scaled identity matrix

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

The function [setIdentity](#) initializes a scaled identity matrix:

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

The function can also be emulated using the matrix initializers and the matrix expressions:

---

```
Mat A = Mat::eye(4, 3, CV_32F)*5;
// A will be set to [[5, 0, 0], [0, 5, 0], [0, 0, 5], [0,
```

---

See also: *Mat::zeros*, *Mat::ones*, *Matrix Expressions*, *Mat::setTo*, *Mat::operator=*,

## solve

---

bool **solve**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, int *flags=DECOMP\_LU*)

Solves one or more linear systems or least-squares problems.

- Parameters:**
- *src1* – The input matrix on the left-hand side of the system
  - *src2* – The input matrix on the right-hand side of the system
  - *dst* – The output solution
  - *flags* –
    - The solution (matrix inversion) method
      - **DECOMP\_LU** - Gaussian elimination with optimal pivot element chosen
      - **DECOMP\_CHOLESKY** - Cholesky  $LL^T$  factorization; the matrix *src1* must be symmetrical and positively defined
      - **DECOMP\_EIG** - Eigenvalue decomposition; the matrix *src1* must be symmetrical
      - **DECOMP\_SVD** - Singular value decomposition (SVD) method; the system can be over-defined and/or the matrix *src1* can be singular
      - **DECOMP\_QR** - QR factorization; the system can be over-defined and/or the matrix *src1* can be singular
      - **DECOMP\_NORMAL** - While all the previous flags are mutually exclusive, this flag can be used together with any of the previous. It means that the normal equations  $\text{src1}^T \cdot \text{src1} \cdot \text{dst} = \text{src1}^T \text{src2}$  are solved instead of the original system  $\text{src1} \cdot \text{dst} = \text{src2}$

The function `solve` solves a linear system or least-squares problem (the latter is possible with SVD or QR methods, or by specifying the flag `DECOMP_NORMAL`):

$$\text{dst} = \arg \min_X \|\text{src1} \cdot X - \text{src2}\|$$

If `DECOMP_LU` or `DECOMP_CHOLESKY` method is used, the function returns

1 if `src1` (or `src1Tsrc1`) is non-singular and 0 otherwise; in the latter case `dst` is not valid. Other methods find some pseudo-solution in the case of singular left-hand side part.

Note that if you want to find unity-norm solution of an under-defined singular system `src1 · dst = 0`, the function `solve` will not do the work. Use `SVD::solveZ` instead.

See also: [invert](#), [SVD](#), [eigen](#)

## solveCubic

---

void **solveCubic**(const Mat& *coeffs*, Mat& *roots*)

Finds the real roots of a cubic equation.

**Parameters:**

- *coeffs* – The equation coefficients, an array of 3 or 4 elements
- *roots* – The destination array of real roots which will have 1 or 3 elements

The function `solveCubic` finds the real roots of a cubic equation:

(if *coeffs* is a 4-element vector)

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

or (if *coeffs* is 3-element vector):

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

The roots are stored to `roots` array.

## solvePoly

---

void **solvePoly**(const Mat& *coeffs*, Mat& *roots*, int *maxIters*=20, int *fig*=100)

Finds the real or complex roots of a polynomial equation

**Parameters:**

- *coeffs* – The array of polynomial coefficients
- *roots* – The destination (complex) array of roots
- *maxIters* – The maximum number of iterations the algorithm does
- *fig* –

The function `solvePoly` finds real and complex roots of a polynomial

equation:

$$\text{coeffs}[0]x^n + \text{coeffs}[1]x^{n-1} + \dots + \text{coeffs}[n-1]x + \text{coeffs}[n] = 0$$

## sort

---

void **sort**(const Mat& *src*, Mat& *dst*, int *flags*)

Sorts each row or each column of a matrix

- Parameters:**
- *src* – The source single-channel array
  - *dst* – The destination array of the same size and the same type as *src*
  - *flags* –  
The operation flags, a combination of the following values:
    - **CV\_SORT\_EVERY\_ROW** - Each matrix row is sorted independently
    - **CV\_SORT\_EVERY\_COLUMN** - Each matrix column is sorted independently. This flag and the previous one are mutually exclusive
    - **CV\_SORT\_ASCENDING** - Each matrix row is sorted in the ascending order
    - **CV\_SORT\_DESCENDING** - Each matrix row is sorted in the descending order. This flag and the previous one are also mutually exclusive

The function `sort` sorts each matrix row or each matrix column in ascending or descending order. If you want to sort matrix rows or columns lexicographically, you can use STL `std::sort` generic function with the proper comparison predicate.

See also: [sortIdx](#), [randShuffle](#)

## sortIdx

---

void **sortIdx**(const Mat& *src*, Mat& *dst*, int *flags*)

Sorts each row or each column of a matrix

- Parameters:**
- *src* – The source single-channel array
  - *dst* – The destination integer array of the same size as *src*
  - *flags* –  
The operation flags, a combination of the following values:
    - **CV\_SORT\_EVERY\_ROW** - Each matrix row is sorted

- independently
- **CV\_SORT\_EVERY\_COLUMN** - Each matrix column is sorted independently. This flag and the previous one are mutually exclusive
- **CV\_SORT\_ASCENDING** - Each matrix row is sorted in the ascending order
- **CV\_SORT\_DESCENDING** - Each matrix row is sorted in the descending order. This flag and the previous one are also mutually exclusive

The function `sortIdx` sorts each matrix row or each matrix column in ascending or descending order. Instead of reordering the elements themselves, it stores the indices of sorted elements in the destination array. For example:

---

```
Mat A = Mat::eye(3,3,CV_32F), B;
sortIdx(A, B, CV_SORT_EVERY_ROW + CV_SORT_ASCENDING);
// B will probably contain
// (because of equal elements in A some permutations are p
// [[1, 2, 0], [0, 2, 1], [0, 1, 2]]
```

---

See also: [sort](#), [randShuffle](#)

## split

---

void **split**(const Mat& *mtx*, Mat\* *mv*)

void **split**(const Mat& *mtx*, vector<Mat>& *mv*)

void **split**(const MatND& *mtx*, MatND\* *mv*)

void **split**(const MatND& *mtx*, vector<MatND>& *mv*)

Divides multi-channel array into several single-channel arrays

- Parameters:**
- *mtx* – The source multi-channel array
  - *mv* – The destination array or vector of arrays; The number of arrays must match `mtx.channels()`. The arrays themselves will be reallocated if needed

The functions `split` split multi-channel array into separate single-channel arrays:

$$\mathbf{mv}[c](I) = \mathbf{mtx}(I)_c$$

If you need to extract a single-channel or do some other sophisticated channel permutation, use [mixChannels](#)

See also: [merge](#), [mixChannels](#), [cvtColor](#)

## sqrt

---

void **sqrt**(const Mat& *src*, Mat& *dst*)

void **sqrt**(const MatND& *src*, MatND& *dst*)

Calculates square root of array elements

- Parameters:**
- *src* – The source floating-point array
  - *dst* – The destination array; will have the same size and the same type as *src*

The functions `sqrt` calculate square root of each source array element. In the case of multi-channel arrays each channel is processed independently. The function accuracy is approximately the same as of the built-in `std::sqrt`.

See also: [pow](#), [magnitude](#)

## subtract

---

void **subtract**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*)

void **subtract**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, const Mat& *mask*)

void **subtract**(const Mat& *src1*, const Scalar& *sc*, Mat& *dst*, const Mat& *mask=Mat()*)

void **subtract**(const Scalar& *sc*, const Mat& *src2*, Mat& *dst*, const Mat& *mask=Mat()*)

void **subtract**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*)

void **subtract**(const MatND& *src1*, const MatND& *src2*, MatND& *dst*, const MatND& *mask*)

void **subtract**(const MatND& *src1*, const Scalar& *sc*, MatND& *dst*, const MatND& *mask=MatND()*)

void **subtract**(const Scalar& *sc*, const MatND& *src2*, MatND& *dst*, const MatND& *mask=MatND()*)

Calculates per-element difference between two arrays or array and a scalar

- Parameters:**
- *src1* – The first source array
  - *src2* – The second source array. It must have the same size and same type as *src1*
  - *sc* – Scalar; the first or the second input parameter
  - *dst* – The destination array; it will have the same size and same type as *src1*; see `Mat::create`
  - *mask* – The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `subtract` compute

- the difference between two arrays

$$\text{dst}(I) = \text{saturate}(\text{src1}(I) - \text{src2}(I)) \quad \text{if } \text{mask}(I) \neq 0$$

- the difference between array and a scalar:

$$\text{dst}(I) = \text{saturate}(\text{src1}(I) - \text{sc}) \quad \text{if } \text{mask}(I) \neq 0$$

- the difference between scalar and an array:

$$\text{dst}(I) = \text{saturate}(\text{sc} - \text{src2}(I)) \quad \text{if } \text{mask}(I) \neq 0$$

where *I* is multi-dimensional index of array elements.

The first function in the above list can be replaced with matrix expressions:

---

```
dst = src1 - src2;
dst -= src2; // equivalent to subtract(dst, src2, dst);
```

---

See also: [add](#), [addWeighted](#), [scaleAdd](#), [convertScale](#), [Matrix Expressions](#), `[cppfunc.saturatecast]bgroup({saturate_cast})`.

## SVD

---

Class for computing Singular Value Decomposition

```
class SVD
{
public:
    enum { MODIFY_A=1, NO_UV=2, FULL_UV=4 };
    // default empty constructor
    SVD();
    // decomposes m into u, w and vt: m = u*w*vt;
    // u and vt are orthogonal, w is diagonal
    SVD( const Mat& m, int flags=0 );
    // decomposes m into u, w and vt.
    SVD& operator ()( const Mat& m, int flags=0 );

    // finds such vector x, norm(x)=1, so that m*x = 0,
    // where m is singular matrix
    static void solveZ( const Mat& m, Mat& dst );
    // does back-substitution:
    // dst = vt.t()*inv(w)*u.t()*rhs ~ inv(m)*rhs
    void backSubst( const Mat& rhs, Mat& dst ) const;

    Mat u, w, vt;
};
```

---

The class **cvSVD()** is used to compute Singular Value Decomposition of a floating-point matrix and then use it to solve least-square problems, under-determined linear systems, invert matrices, compute condition numbers etc. For a bit faster operation you can pass `flags=SVD::MODIFY_A|...` to modify the decomposed matrix when it is not necessarily to preserve it. If you want to compute condition number of a matrix or absolute value of its determinant - you do not need `u` and `vt`, so you can pass `flags=SVD::NO_UV|...`. Another flag `FULL_UV` indicates that full-size `u` and `vt` must be computed, which is not necessary most of the time.

See also: [invert](#), [solve](#), [eigen](#), [determinant](#)

## sum

---

Scalar **sum**(const Mat& *mtx*)

Scalar **sum**(const MatND& *mtx*)

Calculates sum of array elements

**Parameter:** *mtx* – The source array; must have 1 to 4 channels

The functions `sum` calculate and return the sum of array elements, independently for each channel.

See also: [countNonZero](#), [mean](#), [meanStdDev](#), [norm](#), [minMaxLoc](#), [reduce](#)

## theRNG

---

### RNG& **theRNG**()

Returns the default random number generator

The function `theRNG` returns the default random number generator. For each thread there is separate random number generator, so you can use the function safely in multi-thread environments. If you just need to get a single random number using this generator or initialize an array, you can use [randu](#) or [randn](#) instead. But if you are going to generate many random numbers inside a loop, it will be much faster to use this function to retrieve the generator and then use `RNG::operator _Tp()`.

See also: [RNG](#), [randu](#), [randn](#)

## trace

---

### Scalar **trace**(const Mat& *mtx*)

Returns the trace of a matrix

**Parameter:** *mtx* – The source matrix

The function `trace` returns the sum of the diagonal elements of the matrix `mtx`.

$$\text{tr}(\text{mtx}) = \sum_i \text{mtx}(i, i)$$

## transform

---

### void **transform**(const Mat& *src*, Mat& *dst*, const Mat& *mtx*)

Performs matrix transformation of every array element.

**Parameters:** • *src* – The source array; must have as many channels (1 to 4)

- as `mtx.cols` or `mtx.cols-1`
- *dst* – The destination array; will have the same size and depth as *src* and as many channels as `mtx.rows`
- *mtx* – The transformation matrix

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

$$\text{dst}(I) = \text{mtx} \cdot \text{src}(I)$$

(when `mtx.cols=src.channels()`), or

$$\text{dst}(I) = \text{mtx} \cdot [\text{src}(I); 1]$$

(when `mtx.cols=src.channels()+1`)

That is, every element of an *N*-channel array *src* is considered as *N*-element vector, which is transformed using a  $M \times N$  or  $M \times N+1$  matrix *mtx* into an element of *M*-channel array *dst*.

The function may be used for geometrical transformation of *N*-dimensional points, arbitrary linear color space transformation (such as various kinds of RGB: $\rightarrow$ YUV transforms), shuffling the image channels and so forth.

See also: [perspectiveTransform](#), [getAffineTransform](#), [estimateRigidTransform](#), [warpAffine](#), [warpPerspective](#)

## transpose

---

void **transpose**(const Mat& *src*, Mat& *dst*)

Transposes a matrix

- Parameters:**
- *src* – The source array
  - *dst* – The destination array of the same type as *src*

The function [transpose](#) transposes the matrix *src*:

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

Note that no complex conjugation is done in the case of a complex matrix, it should be done separately if needed.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Dynamic Structures

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# 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` or the `Scalar` constructor) for color images and brightness for grayscale images. For color images the order channel is normally Blue, Green, Red, this is what `imshow`, `imread` and `imwrite` expect, so if you form a color using `Scalar` constructor, it should look like:

```
Scalar (blue_component, green_component, red_component[, alpha_component])
```

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using `cvtColor`.

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as  $\text{Point}(x, y) \rightarrow \text{Point2f}(x * 2^{-\text{shift}}, y * 2^{-\text{shift}})$ . This feature is especially effective when rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

## circle

---

```
void circle(Mat& img, Point center, int radius, const Scalar& color, int thickness=1, int lineType=8, int shift=0)
```

Draws a circle

- Parameters:**
- `img` – Image where the circle is drawn
  - `center` – Center of the circle
  - `radius` – Radius of the circle
  - `color` – Circle color
  - `thickness` – Thickness of the circle outline if positive; negative

thickness means that a filled circle is to be drawn

- *lineType* – Type of the circle boundary, see [line](#) description
- *shift* – Number of fractional bits in the center coordinates and radius value

The function `circle` draws a simple or filled circle with a given center and radius.

## clipLine

---

bool **clipLine**(Size *imgSize*, Point& *pt1*, Point& *pt2*)

bool **clipLine**(Rect *imgRect*, Point& *pt1*, Point& *pt2*)

Clips the line against the image rectangle

- Parameters:**
- *imgSize* – The image size; the image rectangle will be `Rect(0, 0, imgSize.width, imgSize.height)`
  - *imgRect* – The image rectangle
  - *pt1* – The first line point
  - *pt2* – The second line point

The functions `clipLine` calculate a part of the line segment which is entirely within the specified rectangle. They return `false` if the line segment is completely outside the rectangle and `true` otherwise.

## ellipse

---

void **ellipse**(Mat& *img*, Point *center*, Size *axes*, double *angle*, double *startAngle*, double *endAngle*, const Scalar& *color*, int *thickness*=1, int *lineType*=8, int *shift*=0)

void **ellipse**(Mat& *img*, const RotatedRect& *box*, const Scalar& *color*, int *thickness*=1, int *lineType*=8)

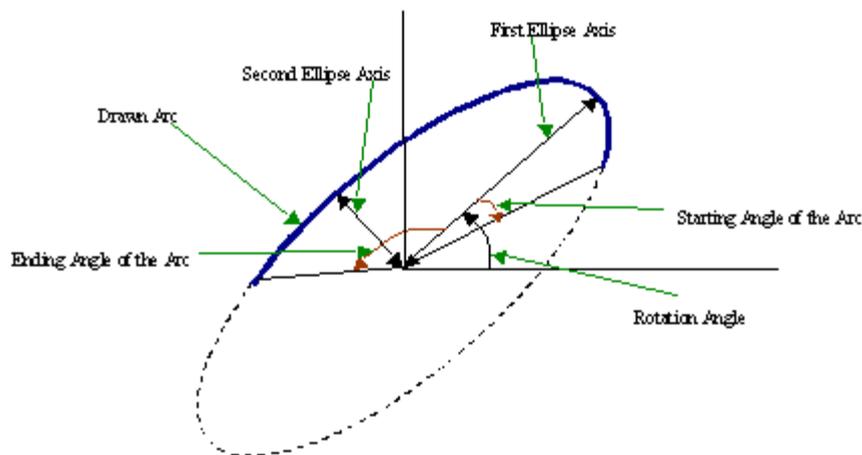
Draws a simple or thick elliptic arc or an fills ellipse sector.

- Parameters:**
- *img* – The image
  - *center* – Center of the ellipse
  - *axes* – Length of the ellipse axes
  - *angle* – The ellipse rotation angle in degrees
  - *startAngle* – Starting angle of the elliptic arc in degrees

- *endAngle* – Ending angle of the elliptic arc in degrees
- *box* – Alternative ellipse representation via a *RotatedRect*, i.e. the function draws an ellipse inscribed in the rotated rectangle
- *color* – Ellipse color
- *thickness* – Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn
- *lineType* – Type of the ellipse boundary, see [line](#) description
- *shift* – Number of fractional bits in the center coordinates and axes' values

The functions `ellipse` with less parameters draw an ellipse outline, a filled ellipse, an elliptic arc or a filled ellipse sector. A piecewise-linear curve is used to approximate the elliptic arc boundary. If you need more control of the ellipse rendering, you can retrieve the curve using [ellipse2Poly](#) and then render it with [polylines](#) or fill it with [fillPoly](#). If you use the first variant of the function and want to draw the whole ellipse, not an arc, pass `startAngle=0` and `endAngle=360`. The picture below explains the meaning of the parameters.

#### Parameters of Elliptic Arc



## ellipse2Poly

```
void ellipse2Poly(Point center, Size axes, int angle, int startAngle, int endAngle, int delta, vector<Point>& pts)
```

Approximates an elliptic arc with a polyline

- Parameters:**
- *center* – Center of the arc
  - *axes* – Half-sizes of the arc. See [ellipse](#)
  - *angle* – Rotation angle of the ellipse in degrees. See [ellipse](#)
  - *startAngle* – Starting angle of the elliptic arc in degrees

- *endAngle* – Ending angle of the elliptic arc in degrees
- *delta* – Angle between the subsequent polyline vertices. It defines the approximation accuracy.
- *pts* – The output vector of polyline vertices

The function `ellipse2Poly` computes the vertices of a polyline that approximates the specified elliptic arc. It is used by [ellipse](#).

## fillConvexPoly

---

void **fillConvexPoly**(Mat& *img*, const Point\* *pts*, int *npts*, const Scalar& *color*, int *lineType*=8, int *shift*=0)

Fills a convex polygon.

- Parameters:**
- *img* – Image
  - *pts* – The polygon vertices
  - *npts* – The number of polygon vertices
  - *color* – Polygon color
  - *lineType* – Type of the polygon boundaries, see [line](#) description
  - *shift* – The number of fractional bits in the vertex coordinates

The function `fillConvexPoly` draws a filled convex polygon. This function is much faster than the function `fillPoly` and can fill not only convex polygons but any monotonic polygon without self-intersections, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most (though, its top-most and/or the bottom edge could be horizontal).

## fillPoly

---

void **fillPoly**(Mat& *img*, const Point\*\* *pts*, const int\* *npts*, int *ncontours*, const Scalar& *color*, int *lineType*=8, int *shift*=0, Point *offset*=Point())

Fills the area bounded by one or more polygons

- Parameters:**
- *img* – Image
  - *pts* – Array of polygons, each represented as an array of points
  - *npts* – The array of polygon vertex counters
  - *ncontours* – The number of contours that bind the filled region
  - *color* – Polygon color
  - *lineType* – Type of the polygon boundaries, see [line](#) description

- *shift* – The number of fractional bits in the vertex coordinates

The function `fillPoly` fills an area bounded by several polygonal contours. The function can fill complex areas, for example, areas with holes, contours with self-intersections (some of their parts), and so forth.

## getTextSize

---

Size **getTextSize**(const string& *text*, int *fontFace*, double *fontScale*, int *thickness*, int\* *baseLine*)

Calculates the width and height of a text string.

- Parameters:**
- *text* – The input text string
  - *fontFace* – The font to use; see [putText](#)
  - *fontScale* – The font scale; see [putText](#)
  - *thickness* – The thickness of lines used to render the text; see [putText](#)
  - *baseLine* – The output parameter - y-coordinate of the baseline relative to the bottom-most text point

The function `getTextSize` calculates and returns size of the box that contain the specified text. That is, the following code will render some text, the tight box surrounding it and the baseline:

---

```
// Use "y" to show that the baseLine is about
string text = "Funny text inside the box";
int fontFace = FONT_HERSHEY_SCRIPT_SIMPLEX;
double fontScale = 2;
int thickness = 3;

Mat img(600, 800, CV_8UC3, Scalar::all(0));

int baseline=0;
Size textSize = getTextSize(text, fontFace,
                             fontScale, thickness, &baseline);
baseline += thickness;

// center the text
Point textOrg((img.cols - textSize.width)/2,
              (img.rows + textSize.height)/2);

// draw the box
rectangle(img, textOrg + Point(0, baseline),
          textOrg + Point(textSize.width, -textSize.height),
          Scalar(0,0,255));
// ... and the baseline first
line(img, textOrg + Point(0, thickness),
```

```

        textOrg + Point(textSize.width, thickness),
        Scalar(0, 0, 255));

    // then put the text itself
    putText(img, text, textOrg, fontFace, fontScale,
            Scalar::all(255), thickness, 8);

```

---

## line

---

void **line**(Mat& *img*, Point *pt1*, Point *pt2*, const Scalar& *color*, int *thickness*=1, int *lineType*=8, int *shift*=0)

Draws a line segment connecting two points

- Parameters:**
- *img* – The image
  - *pt1* – First point of the line segment
  - *pt2* – Second point of the line segment
  - *color* – Line color
  - *thickness* – Line thickness
  - *lineType* –
    - Type of the line:
      - **8** - (or omitted) 8-connected line.
      - **4** - 4-connected line.
      - **CV\_AA** - antialiased line.
  - *shift* – Number of fractional bits in the point coordinates

The function `line` draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image boundaries. 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)`.

## LineIterator

---

Class for iterating pixels on a raster line

```

class LineIterator
{
public:
    // creates iterators for the line connecting pt1 and pt2
    // the line will be clipped on the image boundaries
    // the line is 8-connected or 4-connected
    // If leftToRight=true, then the iteration is always done

```

```

// from the left-most point to the right most,
// not to depend on the ordering of pt1 and pt2 parameters
LineIterator(const Mat& img, Point pt1, Point pt2,
             int connectivity=8, bool leftToRight=false);
// returns pointer to the current line pixel
uchar* operator *();newline
// move the iterator to the next pixel
LineIterator& operator ++();newline
LineIterator operator ++(int);newline

// internal state of the iterator
uchar* ptr;newline
int err, count;newline
int minusDelta, plusDelta;newline
int minusStep, plusStep;newline
};

```

The class `LineIterator` is used to get each pixel of a raster line. It can be treated as versatile implementation of the Bresenham algorithm, where you can stop at each pixel and do some extra processing, for example, grab pixel values along the line, or draw a line with some effect (e.g. with XOR operation).

The number of pixels along the line is store in `LineIterator::count`.

```

// grabs pixels along the line (pt1, pt2)
// from 8-bit 3-channel image to the buffer
LineIterator it(img, pt1, pt2, 8);
vector<Vec3b> buf(it.count);

for(int i = 0; i < it.count; i++, ++it)
    buf[i] = *(const Vec3b)*it;

```

## rectangle

void **rectangle**(Mat& *img*, Point *pt1*, Point *pt2*, const Scalar& *color*, int *thickness*=1, int *lineType*=8, int *shift*=0)

Draws a simple, thick, or filled up-right rectangle.

- Parameters:**
- *img* – Image
  - *pt1* – One of the rectangle's vertices
  - *pt2* – Opposite to *pt1* rectangle vertex
  - *color* – Rectangle color or brightness (grayscale image)
  - *thickness* – Thickness of lines that make up the rectangle. Negative values, e.g. `CV_FILLED`, mean that the function has to draw a filled rectangle.
  - *lineType* – Type of the line, see [line](#) description
  - *shift* – Number of fractional bits in the point coordinates

The function `rectangle` draws a rectangle outline or a filled rectangle, which two opposite corners are `pt1` and `pt2`.

## polylines

---

```
void polylines(Mat& img, const Point** pts, const int* npts, int ncontours,
bool isClosed, const Scalar& color, int thickness=1, int lineType=8, int shift=0)
```

Draws several polygonal curves

- Parameters:**
- *img* – The image
  - *pts* – Array of polygonal curves
  - *npts* – Array of polygon vertex counters
  - *ncontours* – The number of curves
  - *isClosed* – Indicates whether the drawn polylines are closed or not. If they are closed, the function draws the line from the last vertex of each curve to its first vertex
  - *color* – Polyline color
  - *thickness* – Thickness of the polyline edges
  - *lineType* – Type of the line segments, see [line](#) description
  - *shift* – The number of fractional bits in the vertex coordinates

The function `polylines` draws one or more polygonal curves.

## putText

---

```
void putText(Mat& img, const string& text, Point org, int fontFace, double
fontScale, Scalar color, int thickness=1, int lineType=8, bool bottomLeftOrigin=false)
```

Draws a text string

- Parameters:**
- *img* – The image
  - *text* – The text string to be drawn
  - *org* – The bottom-left corner of the text string in the image
  - *fontFace* –
    - The font type, one of \* `FONT_HERSHEY_SIMPLEX` - ,
    - `FONT_HERSHEY_PLAIN` - ,
    - `FONT_HERSHEY_DUPLEX` - ,
    - `FONT_HERSHEY_COMPLEX` - ,
    - `FONT_HERSHEY_TRIPLEX` - ,
    - `FONT_HERSHEY_COMPLEX_SMALL` - ,
    - `FONT_HERSHEY_SCRIPT_SIMPLEX` - ○

r

- **FONT\_HERSHEY\_SCRIPT\_COMPLEX** - ,

where each of the font id's can be combined with

- **FONT\_HERSHEY\_ITALIC** - t

o get the slanted letters.

**param fontScale:**

The font scale factor that is multiplied by the font-specific base size

**param thickness:**

Thickness of the lines used to draw the text

**param**

The line type; see `line` for details

**lineType:**

**param bottomLeftOrigin:**

When true, the image data origin is at the bottom-left corner, otherwise it's at the top-left corner

The function `putText` draws a text string in the image. Symbols that can not be rendered using the specified font are replaced question marks. See [getTextSize](#) for a text rendering code example.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# XML/YAML Persistence

---

## FileStorage

---

The XML/YAML file storage class

```
class FileStorage
{
public:
    enum { READ=0, WRITE=1, APPEND=2 };
    enum { UNDEFINED=0, VALUE_EXPECTED=1, NAME_EXPECTED=2, IN
// the default constructor
FileStorage();
// the constructor that opens the file for reading
// (flags=FileStorage::READ) or writing (flags=FileStorag
FileStorage(const string& filename, int flags);
// wraps the already opened CvFileStorage*
FileStorage(CvFileStorage* fs);
// the destructor; closes the file if needed
virtual ~FileStorage();

// opens the specified file for reading (flags=FileStorag
// or writing (flags=FileStorage::WRITE)
virtual bool open(const string& filename, int flags);
// checks if the storage is opened
virtual bool isOpened() const;
// closes the file
virtual void release();

// returns the first top-level node
FileNode getFirstTopLevelNode() const;
// returns the root file node
// (it's the parent of the first top-level node)
FileNode root(int streamidx=0) const;
// returns the top-level node by name
FileNode operator[](const string& nodename) const;
FileNode operator[](const char* nodename) const;

// returns the underlying CvFileStorage*
CvFileStorage* operator *() { return fs; }
const CvFileStorage* operator *() const { return fs; }

// writes the certain number of elements of the specified
// (see DataType) without any headers
void writeRaw(const string& fmt, const uchar* vec, size_

// writes an old-style object (CvMat, CvMatND etc.)
void writeObj(const string& name, const void* obj);
```

```

// returns the default object name from the filename
// (used by cvSave() with the default object name etc.)
static string getDefaultObjectName(const string& filename

Ptr<CvFileStorage> fs;
string elname;
vector<char> structs;
int state;
};

```

---

## FileNode

---

### The XML/YAML file node class

---

```

class CV_EXPORTS FileNode
{
public:
    enum { NONE=0, INT=1, REAL=2, FLOAT=REAL, STR=3,
          STRING=STR, REF=4, SEQ=5, MAP=6, TYPE_MASK=7,
          FLOW=8, USER=16, EMPTY=32, NAMED=64 };
    FileNode();
    FileNode(const CvFileStorage* fs, const CvFileNode* node)
    FileNode(const FileNode& node);
    FileNode operator[] (const string& nodename) const;
    FileNode operator[] (const char* nodename) const;
    FileNode operator[] (int i) const;
    int type() const;
    int rawDataSize(const string& fmt) const;
    bool empty() const;
    bool isNone() const;
    bool isSeq() const;
    bool isMap() const;
    bool isInt() const;
    bool isReal() const;
    bool isString() const;
    bool isNamed() const;
    string name() const;
    size_t size() const;
    operator int() const;
    operator float() const;
    operator double() const;
    operator string() const;

    FileNodeIterator begin() const;
    FileNodeIterator end() const;

    void readRaw(const string& fmt, uchar* vec, size_t len)
    void* readObj() const;

```

```
// do not use wrapper pointer classes for better efficiency
const CvFileStorage* fs;
const CvFileNode* node;
};
```

---

## FileNodeIterator

---

The XML/YAML file node iterator class

---

```
class CV_EXPORTS FileNodeIterator
{
public:
    FileNodeIterator();
    FileNodeIterator(const CvFileStorage* fs,
                    const CvFileNode* node, size_t ofs=0);
    FileNodeIterator(const FileNodeIterator& it);
    FileNode operator *() const;
    FileNode operator ->() const;

    FileNodeIterator& operator ++();
    FileNodeIterator operator ++(int);
    FileNodeIterator& operator --();
    FileNodeIterator operator --(int);
    FileNodeIterator& operator += (int);
    FileNodeIterator& operator -= (int);

    FileNodeIterator& readRaw( const string& fmt, uchar* vec,
                              size_t maxCount=(size_t)INT_MA

    const CvFileStorage* fs;
    const CvFileNode* container;
    CvSeqReader reader;
    size_t remaining;
};
```

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Clustering and Search in Multi-Dimensional Spaces

## kmeans

double **kmeans**(const Mat& *samples*, int *clusterCount*, Mat& *labels*, TermCriteria *termcrit*, int *attempts*, int *flags*, Mat\* *centers*)

- Parameters:**
- *samples* – Floating-point matrix of input samples, one row per sample
  - *clusterCount* – The number of clusters to split the set by
  - *labels* – The input/output integer array that will store the cluster indices for every sample
  - *termcrit* – Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)
  - *attempts* – How many times the algorithm is executed using different initial labelings. The algorithm returns the labels that yield the best compactness (see the last function parameter)
  - *flags* –
    - It can take the following values:
      - **KMEANS\_RANDOM\_CENTERS** - Random initial centers are selected in each attempt
      - **KMEANS\_PP\_CENTERS** - Use kmeans++ center initialization by Arthur and Vassilvitskii
      - **KMEANS\_USE\_INITIAL\_LABELS** - During the first (and possibly the only) attempt, the function uses the user-supplied labels instead of computing them from the initial centers. For the second and further attempts, the function will use the random or semi-random centers (use one of **KMEANS\_\*\_CENTERS** flag to specify the exact method)
  - *centers* – The output matrix of the cluster centers, one row per each cluster center

The function `kmeans` implements a k-means algorithm that finds the centers of `clusterCount` clusters and groups the input samples around the clusters. On output, `labelsi` contains a 0-based cluster index for the sample stored in the *i*<sup>th</sup> row of the `samples` matrix.

The function returns the compactness measure, which is computed as

$$\sum_i \|\text{samples}_i - \text{centers}_{\text{labels}_i}\|^2$$

after every attempt; the best (minimum) value is chosen and the corresponding

labels and the compactness value are returned by the function. Basically, the user can use only the core of the function, set the number of attempts to 1, initialize labels each time using some custom algorithm and pass them with

(`flags` `=``KMEANS_USE_INITIAL_LABELS`) flag, and then choose the best (most-compact) clustering.

## partition

---

```
template<typename _Tp, class _EqPredicate> int partition(const
vector<_Tp>& vec, vector<int>& labels, _EqPredicate predicate=_EqPredicate())
```

Splits an element set into equivalency classes.

- Parameters:**
- *vec* – The set of elements stored as a vector
  - *labels* – The output vector of labels; will contain as many elements as *vec*. Each label *labels[i]* is 0-based cluster index of *vec[i]*
  - *predicate* – The equivalence predicate (i.e. pointer to a boolean function of two arguments or an instance of the class that has the method `bool operator()(const _Tp a, const _Tp b)`). The predicate returns true when the elements are certainly if the same class, and false if they may or may not be in the same class

The generic function `partition` implements an  $O(N^2)$  algorithm for splitting a set of  $N$  elements into one or more equivalency classes, as described in [http://en.wikipedia.org/wiki/Disjoint-set\\_data\\_structure](http://en.wikipedia.org/wiki/Disjoint-set_data_structure). The function returns the number of equivalency classes.

## Fast Approximate Nearest Neighbor Search

---

This section documents OpenCV's interface to the FLANN <http://people.cs.ubc.ca/~.05em.7ex.04emmariusm/flann> library. FLANN (Fast Library for Approximate Nearest Neighbors) is a library that contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. More information about FLANN can be found in .

## flann::Index

---

The FLANN nearest neighbor index class.

---

```
namespace flann
```

```

{
    class Index
    {
    public:
        Index(const Mat& features, const IndexParams& params);

        void knnSearch(const vector<float>& query,
                      vector<int>& indices,
                      vector<float>& dists,
                      int knn,
                      const SearchParams& params);
        void knnSearch(const Mat& queries,
                      Mat& indices,
                      Mat& dists,
                      int knn,
                      const SearchParams& params);

        int radiusSearch(const vector<float>& query,
                        vector<int>& indices,
                        vector<float>& dists,
                        float radius,
                        const SearchParams& params);
        int radiusSearch(const Mat& query,
                        Mat& indices,
                        Mat& dists,
                        float radius,
                        const SearchParams& params);

        void save(std::string filename);

        int veclen() const;

        int size() const;
    };
}

```

---

## flann::Index::Index

---

Index:: **Index**(const Mat& *features*, const IndexParams& *params*)

Constructs a nearest neighbor search index for a given dataset.

- Parameters:**
- *features* – Matrix of type CV\_32F containing the features(points) to num\_features x feature\_dimensionality.
  - *params* – Structure containing the index parameters. The type of index that the type of this parameter. The possible parameter types are:
    - **LinearIndexParams** - When passing an object of this type, the i force search.

---

```
struct LinearIndexParams : public IndexParams
{
};
```

---

- **KDTreeIndexParams** - When passing an object of this type the a set of randomized kd-trees which will be searched in parallel.

---

```
struct KDTreeIndexParams : public IndexParams
{
    KDTreeIndexParams( int trees = 4 );
};
```

---

- **trees** - The number of parallel kd-trees to use. Good value is 4.

- **KMeansIndexParams** - When passing an object of this type the hierarchical k-means tree.

---

```
struct KMeansIndexParams : public IndexParams
{
    KMeansIndexParams( int branching = 32,
                      int iterations = 11,
                      flann_centers_init_t centers_init =
                      float cb_index = 0.2 );
};
```

---

- **branching** - The branching factor to use for the hierarchical k-means tree.
- **iterations** - The maximum number of iterations to use stage when building the k-means tree. A value of -1 means clustering should be iterated until convergence
- **centers\_init** - The algorithm to use for selecting the initial cluster centers. The possible values are CLUSTERING\_RANDOM (the initial cluster centers randomly), CENTERS\_GONZALES (the initial cluster centers using Gonzales' algorithm) and CENTERS\_KMEANS (the initial cluster centers using the algorithm suggested in [1])
- **cb\_index** - This parameter (cluster boundary index) influences the hierarchical kmeans tree. When cb\_index is greater than zero, the kmeans domain to be explored is chosen to be the one with the largest value greater than zero also takes into account the size of the domain.

- **CompositeIndexParams** - When using a parameters object the hierarchical k-means tree and the randomized kd-trees

---

```
struct CompositeIndexParams : public IndexParams
{
    CompositeIndexParams( int trees = 4,
                        int branching = 32,
                        int iterations = 11,
                        flann_centers_init_t centers_init =
                        float cb_index = 0.2 );
};
```

---

```

flann_centers_init_t centers_init = CENT
float cb_index = 0.2 );
} ;

```

---

- **AutotunedIndexParams** - When passing an object of this type tuned to offer the best performance, by choosing the optimal (for hierarchical kmeans, linear) and parameters for the dataset provided.

```

struct AutotunedIndexParams : public IndexParams
{
AutotunedIndexParams( float target_precision = 0.01,
float build_weight = 0.01,
float memory_weight = 0,
float sample_fraction = 0.1 );
} ;

```

---

- **target\_precision** - Is a number between 0 and 1 specifying the approximate nearest-neighbor searches that return the most accurate results. Using a higher value for this parameter gives more accurate results but takes longer. The optimum value usually depends on the dataset.
- **build\_weight** - Specifies the importance of the index building step to the nearest-neighbor search time. In some applications it's important to build the index as fast as possible even if that leads to slightly longer search times.
- **memory\_weight** - Is used to specify the tradeoff between building time (and search time) and memory used by the index. A value of 1 means equal importance to the time spent and a value greater than 1 means more importance to the memory usage.
- **sample\_fraction** - Is a number between 0 and 1 indicating the fraction of the dataset to use in the automatic parameter configuration algorithm. Using a smaller fraction on the full dataset gives the most accurate results but can take longer than desired. In such case using this parameter helps speeding up this algorithm while still giving good results.

- **SavedIndexParams** - This object type is used for loading a previously saved index.

```

struct SavedIndexParams : public IndexParams
{
SavedIndexParams( std::string filename );
} ;

```

---

- **filename** - The filename in which the index was saved.

## flann::Index::knnSearch

---

void Index : : **knnSearch**(const vector<float>& *query*, vector<int>& *indices*, vector<float>& *dists*, int *knn*, const SearchParams& *params*)

Performs a K-nearest neighbor search for a given query point using the index.

- Parameters:**
- *query* – The query point
  - *indices* – Vector that will contain the indices of the K-nearest neighbors found. It must have at least *knn* size.
  - *dists* – Vector that will contain the distances to the K-nearest neighbors found. It must have at least *knn* size.
  - *knn* – Number of nearest neighbors to search for.
  - *params* – Search parameters

---

```
struct SearchParams {
    SearchParams(int checks = 32);
};
```

---

- *checks* – The number of times the tree(s) in the index should be recursively traversed. A higher value for this parameter would give better search precision, but also take more time. If automatic configuration was used when the index was created, the number of checks required to achieve the specified precision was also computed, in which case this parameter is ignored.

## flann::Index::knnSearch

---

void Index : : **knnSearch**(const Mat& *queries*, Mat& *indices*, Mat& *dists*, int *knn*, const SearchParams& *params*)

Performs a K-nearest neighbor search for multiple query points.

- Parameters:**
- *queries* – The query points, one per row
  - *indices* – Indices of the nearest neighbors found
  - *dists* – Distances to the nearest neighbors found
  - *knn* – Number of nearest neighbors to search for
  - *params* – Search parameters

## flann::Index::radiusSearch

---

```
int Index::radiusSearch(const vector<float>& query, vector<int>&
indices, vector<float>& dists, float radius, const SearchParams& params)
```

Performs a radius nearest neighbor search for a given query point.

- Parameters:**
- *query* – The query point
  - *indices* – Vector that will contain the indices of the points found within the search radius in decreasing order of the distance to the query point. If the number of neighbors in the search radius is bigger than the size of this vector, the ones that don't fit in the vector are ignored.
  - *dists* – Vector that will contain the distances to the points found within the search radius
  - *radius* – The search radius
  - *params* – Search parameters

## flann::Index::radiusSearch

---

```
int Index::radiusSearch(const Mat& query, Mat& indices, Mat&
dists, float radius, const SearchParams& params)
```

Performs a radius nearest neighbor search for multiple query points.

- Parameters:**
- *queries* – The query points, one per row
  - *indices* – Indices of the nearest neighbors found
  - *dists* – Distances to the nearest neighbors found
  - *radius* – The search radius
  - *params* – Search parameters

## flann::Index::save

---

```
void Index::save(std::string filename)
```

Saves the index to a file.

**Parameter:** *filename* – The file to save the index to

## flann::hierarchicalClustering

---

```
int hierarchicalClustering(const Mat& features, Mat&
centers, const KMeansIndexParams& params)
```

Clusters the given points by constructing a hierarchical k-means tree and choosing a cut in the tree that minimizes the cluster's variance.

- Parameters:**
- *features* – The points to be clustered
  - *centers* – The centers of the clusters obtained. The number of rows in this matrix represents the number of clusters desired, however, because of the way the cut in the hierarchical tree is chosen, the number of clusters computed will be the highest number of the form  $(branching - 1) * k + 1$  that's lower than the number of clusters desired, where *branching* is the tree's branching factor (see description of the KMeansIndexParams).
  - *params* – Parameters used in the construction of the hierarchical k-means tree

The function returns the number of clusters computed.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Utility and System Functions and Macros

---

## alignPtr

---

```
template<typename _Tp> _Tp* alignPtr(_Tp* ptr, int n=sizeof(_Tp))
```

Aligns pointer to the specified number of bytes

- Parameters:**
- *ptr* – The aligned pointer
  - *n* – The alignment size; must be a power of two

The function returns the aligned pointer of the same type as the input pointer:

$$(\_Tp^*)((((size\_t)ptr + n-1) \& -n))$$

## alignSize

---

```
size_t alignSize(size_t sz, int n)
```

Aligns a buffer size to the specified number of bytes

- Parameters:**
- *sz* – The buffer size to align
  - *n* – The alignment size; must be a power of two

The function returns the minimum number that is greater or equal to *sz* and is divisible by *n*:

$$(sz + n-1) \& -n$$

## allocate

---

```
template<typename _Tp> _Tp* allocate(size_t n)
```

Allocates an array of elements

- Parameter:** *n* – The number of elements to allocate

The generic function `allocate` allocates buffer for the specified number of elements. For each element the default constructor is called.

## deallocate

---

```
template<typename _Tp> void deallocate(_Tp* ptr, size_t n)
```

Allocates an array of elements

- Parameters:**
- *ptr* – Pointer to the deallocated buffer
  - *n* – The number of elements in the buffer

The generic function `deallocate` deallocates the buffer allocated with `allocate`. The number of elements must match the number passed to `allocate`.

## CV\_Assert

---

Checks a condition at runtime.

```
#define CV_Assert( expr ) ...
#define CV_DbgAssert( expr ) ...
```

**param expr:** The checked expression

The macros `CV_Assert` and `CV_DbgAssert` evaluate the specified expression and if it is 0, the macros raise an error (see [error](#)). The macro `CV_Assert` checks the condition in both Debug and Release configurations, while `CV_DbgAssert` is only retained in the Debug configuration.

## error

---

void **error**(const Exception& exc)

```
# define CV_Error( code, msg ) <...> #
define CV_Error_( code, args ) <...>
```

Signals an error and raises the exception

- Parameters:**
- *exc* – The exception to throw
  - *code* – The error code, normally, a negative value. The list of pre-defined error codes can be found in `cxerror.h`
  - *msg* – Text of the error message
  - *args* – printf-like formatted error message in parantheses

The function and the helper macros `CV_Error` and `CV_Error_` call the error handler. Currently, the error handler prints the error code (`exc.code`), the context (`exc.file`, `exc.line` and the error message `exc.err` to the standard error stream `stderr`. In Debug configuration it then provokes memory access violation, so that the execution stack and all the parameters can be analyzed in debugger. In Release configuration the exception `exc` is thrown.

The macro **CV\_Error\_** can be used to construct the error message on-fly to include some dynamic information, for example:

---

```
// note the extra parentheses around the formatted text me
CV_Error_(CV_StsOutOfRange,
          ("the matrix element (%d,%d)=%g is out of range",
           i, j, mtx.at<float>(i,j)))
```

---

## Exception

---

The exception class passed to error

---

```
class Exception
{
public:
    // various constructors and the copy operation
    Exception() { code = 0; line = 0; }
    Exception(int _code, const string& _err,
              const string& _func, const string& _file, int _
    Exception(const Exception& exc);newline
    Exception& operator = (const Exception& exc);newline

    // the error code
    int code;newline
    // the error text message
    string err;newline
    // function name where the error happened
    string func;newline
    // the source file name where the error happened
    string file;newline
    // the source file line where the error happened
    int line;
};
```

---

The class `Exception` encapsulates all or almost all the necessary information about the error happened in the program. The exception is usually constructed and thrown implicitly, via **CV\_Error** and **CV\_Error\_** macros, see [error](#).

## fastMalloc

---

void\* **fastMalloc**(size\_t size)

Allocates aligned memory buffer

**Parameter:** *size* – The allocated buffer size

The function allocates buffer of the specified size and returns it. When the buffer size is 16 bytes or more, the returned buffer is aligned on 16 bytes.

## fastFree

---

void **fastFree**(void\* *ptr*)

Deallocates memory buffer

**Parameter:** *ptr* – Pointer to the allocated buffer

The function deallocates the buffer, allocated with [fastMalloc](#). If NULL pointer is passed, the function does nothing.

## format

---

string **format**(const char\* *fmt*, ...)

Returns a text string formatted using printf-like expression

**Parameter:** *fmt* – The printf-compatible formatting specifiers

The function acts like `sprintf`, but forms and returns STL string. It can be used for form the error message in [Exception](#) constructor.

## getNumThreads

---

int **getNumThreads**()

Returns the number of threads used by OpenCV

The function returns the number of threads that is used by OpenCV.

See also: [setNumThreads](#), [getThreadNum](#).

## getThreadNum

---

int **getThreadNum**()

Returns index of the currently executed thread

The function returns 0-based index of the currently executed thread. The function

is only valid inside a parallel OpenMP region. When OpenCV is built without OpenMP support, the function always returns 0.

See also: [setNumThreads](#), [getNumThreads](#).

## getTickCount

---

int64 **getTickCount()**

Returns the number of ticks

The function returns the number of ticks since the certain event (e.g. when the machine was turned on). It can be used to initialize *RNG* or to measure a function execution time by reading the tick count before and after the function call. See also the tick frequency.

## getTickFrequency

---

double **getTickFrequency()**

Returns the number of ticks per second

The function returns the number of ticks per second. That is, the following code computes the executing time in seconds.

```
double t = (double)getTickCount();  
// do something ...  
t = ((double)getTickCount() - t)/getTickFrequency();
```

## setNumThreads

---

void **setNumThreads**(int *nthreads*)

Sets the number of threads used by OpenCV

**Parameter:** *nthreads* – The number of threads used by OpenCV

The function sets the number of threads used by OpenCV in parallel OpenMP regions. If *nthreads*=0, the function will use the default number of threads, which is usually equal to the number of the processing cores.

See also: [getNumThreads](#), [getThreadNum](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# cv. Image Processing and Computer Vision

---

- Image Filtering
  - BaseColumnFilter
  - BaseFilter
  - BaseRowFilter
  - FilterEngine
  - bilateralFilter
  - blur
  - borderInterpolate
  - boxFilter
  - buildPyramid
  - copyMakeBorder
  - createBoxFilter
  - createDerivFilter
  - createGaussianFilter
  - createLinearFilter
  - createMorphologyFilter
  - createSeparableLinearFilter
  - dilate
  - erode
  - filter2D
  - GaussianBlur
  - getDerivKernels
  - getGaussianKernel
  - getKernelType
  - getStructuringElement
  - medianBlur
  - morphologyEx
  - Laplacian
  - pyrDown
  - pyrUp
  - sepFilter2D
  - Sobel
  - Scharr
- Geometric Image Transformations
  - convertMaps
  - getAffineTransform
  - getPerspectiveTransform
  - getRectSubPix
  - getRotationMatrix2D
  - invertAffineTransform
  - remap
  - resize
  - warpAffine

- warpPerspective
- Miscellaneous Image Transformations
  - adaptiveThreshold
  - cvtColor
  - distanceTransform
  - floodFill
  - inpaint
  - integral
  - threshold
  - watershed
- Histograms
  - calcHist
  - calcBackProject
  - compareHist
  - equalizeHist
- Feature Detection
  - Canny
  - cornerEigenValsAndVecs
  - cornerHarris
  - cornerMinEigenVal
  - cornerSubPix
  - goodFeaturesToTrack
  - HoughCircles
  - HoughLines
  - HoughLinesP
  - perCornerDetect
  - KeyPoint
  - MSER
  - SURF
  - StarDetector
- Motion Analysis and Object Tracking
  - accumulate
  - accumulateSquare
  - accumulateProduct
  - accumulateWeighted
  - calcOpticalFlowPyrLK
  - calcOpticalFlowFarneback
  - updateMotionHistory
  - calcMotionGradient
  - calcGlobalOrientation
  - CamShift
  - meanShift
  - KalmanFilter
- Structural Analysis and Shape Descriptors
  - moments
  - HuMoments

- [findContours](#)
- [drawContours](#)
- [approxPolyDP](#)
- [arcLength](#)
- [boundingRect](#)
- [estimateRigidTransform](#)
- [estimateAffine3D](#)
- [contourArea](#)
- [convexHull](#)
- [fitEllipse](#)
- [fitLine](#)
- [isContourConvex](#)
- [minAreaRect](#)
- [minEnclosingCircle](#)
- [matchShapes](#)
- [pointPolygonTest](#)
- [Planar Subdivisions](#)
- [Object Detection](#)
  - [FeatureEvaluator](#)
  - [CascadeClassifier](#)
  - [groupRectangles](#)
  - [matchTemplate](#)
- [Camera Calibration and 3D Reconstruction](#)
  - [calibrateCamera](#)
  - [calibrationMatrixValues](#)
  - [composeRT](#)
  - [computeCorrespondEpilines](#)
  - [convertPointsHomogeneous](#)
  - [decomposeProjectionMatrix](#)
  - [drawChessboardCorners](#)
  - [findChessboardCorners](#)
  - [solvePnP](#)
  - [findFundamentalMat](#)
  - [findHomography](#)
  - [getDefaultNewCameraMatrix](#)
  - [getOptimalNewCameraMatrix](#)
  - [initCameraMatrix2D](#)
  - [initUndistortRectifyMap](#)
  - [matMulDeriv](#)
  - [projectPoints](#)
  - [reprojectImageTo3D](#)
  - [RQDecomp3x3](#)
  - [Rodrigues](#)
  - [StereoBM](#)
  - [stereoCalibrate](#)
  - [stereoRectify](#)

- [stereoRectifyUncalibrated](#)
- [undistort](#)
- [undistortPoints](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# 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  $3 \times 3$  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 (“constant border” extrapolation method) etc. OpenCV let the user to specify the extrapolation method; see the function [borderInterpolate](#) and discussion of `borderType` parameter in various functions below.

## BaseColumnFilter

---

Base class for filters with single-column kernels

```
class BaseColumnFilter
{
public:
    virtual ~BaseColumnFilter();

    // To be overridden by the user.
    //
    // runs filtering operation on the set of rows,
    // "dstcount + ksize - 1" rows on input,
    // "dstcount" rows on output,
    // each input and output row has "width" elements
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, int
                           dstcount, int width) = 0;
    // resets the filter state (may be needed for IIR filters
```

```

virtual void reset();

    int ksize; // the aperture size
    int anchor; // position of the anchor point,
                // normally not used during the processing
};

```

---

The class `BaseColumnFilter` is the base class for filtering data using single-column kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\text{dst}(x, y) = F(\text{src}[y](x), \text{src}[y + 1](x), \dots, \text{src}[y + \text{ksize} - 1](x))$$

where  $F$  is the filtering function, but, as it is represented as a class, it can produce any side effects, memorize previously processed data etc. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to `FilterEngine` constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: [BaseRowFilter](#), [BaseFilter](#), [FilterEngine](#), [getColumnSumFilter](#), [getLinearColumnFilter](#), [getMorphologyColumnFilter](#)

## BaseFilter

---

Base class for 2D image filters

```

class BaseFilter
{
public:
    virtual ~BaseFilter();

    // To be overridden by the user.
    //
    // runs filtering operation on the set of rows,
    // "dstcount + ksize.height - 1" rows on input,
    // "dstcount" rows on output,
    // each input row has "(width + ksize.width-1)*cn" elemen
    // each output row has "width*cn" elements.
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, in
                           int dstcount, int width, int cn)
    // resets the filter state (may be needed for IIR filters)
    virtual void reset();
    Size ksize;
    Point anchor;
};

```

---

The class `BaseFilter` is the base class for filtering data using 2D kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\text{dst}(x, y) = F(\text{src}[y](x), \text{src}[y](x + 1), \dots, \text{src}[y](x + \text{ksize.width} - 1), \\ \text{src}[y + 1](x), \text{src}[y + 1](x + 1), \dots, \text{src}[y + 1](x + \text{ksize.width} - 1), \\ \dots \\ \text{src}[y + \text{ksize.height} - 1](x), \\ \text{src}[y + \text{ksize.height} - 1](x + 1), \\ \dots \text{src}[y + \text{ksize.height} - 1](x + \text{ksize.width} - 1))$$

where  $F$  is the filtering function. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to `FilterEngine` constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: [BaseColumnFilter](#), [BaseRowFilter](#), [FilterEngine](#), [getLinearFilter](#), [getMorphologyFilter](#)

## BaseRowFilter

---

Base class for filters with single-row kernels

```
class BaseRowFilter
{
public:
    virtual ~BaseRowFilter();

    // To be overridden by the user.
    //
    // runs filtering operation on the single input row
    // of "width" element, each element is has "cn" channels.
    // the filtered row is written into "dst" buffer.
    virtual void operator()(const uchar* src, uchar* dst,
                           int width, int cn) = 0;

    int ksize, anchor;
};
```

---

The class `BaseRowFilter` is the base class for filtering data using single-row kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\text{dst}(x, y) = F(\text{src}[y](x), \text{src}[y](x + 1), \dots, \text{src}[y](x + \text{ksize.width} - 1))$$

where  $F$  is the filtering function. The class only defines the interface and is not used

directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to *FilterEngine* constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: [BaseColumnFilter](#), [Filter](#), [FilterEngine](#), [getLinearRowFilter](#), [getMorphologyRowFilter](#), [getRowSumFilter](#)

## FilterEngine

---

Generic image filtering class

```
class FilterEngine
{
public:
    // empty constructor
    FilterEngine();
    // builds a 2D non-separable filter (!_filter2D.empty())
    // a separable filter (!_rowFilter.empty() && !_columnFil
    // the input data type will be "srcType", the output data
    // the intermediate data type is "bufType".
    // _rowBorderType and _columnBorderType determine how the
    // will be extrapolated beyond the image boundaries.
    // _borderValue is only used when _rowBorderType and/or _
    // == cv::BORDER_CONSTANT
    FilterEngine(const Ptr<BaseFilter>& _filter2D,
                 const Ptr<BaseRowFilter>& _rowFilter,
                 const Ptr<BaseColumnFilter>& _columnFilter,
                 int srcType, int dstType, int bufType,
                 int _rowBorderType=BORDER_REPLICATE,
                 int _columnBorderType=-1, // use _rowBorderT
                 const Scalar& _borderValue=Scalar());
    virtual ~FilterEngine();
    // separate function for the engine initialization
    void init(const Ptr<BaseFilter>& _filter2D,
              const Ptr<BaseRowFilter>& _rowFilter,
              const Ptr<BaseColumnFilter>& _columnFilter,
              int srcType, int dstType, int bufType,
              int _rowBorderType=BORDER_REPLICATE, int _column
              const Scalar& _borderValue=Scalar());
    // starts filtering of the ROI in an image of size "whole
    // returns the starting y-position in the source image.
    virtual int start(Size wholeSize, Rect roi, int maxBufRow
    // alternative form of start that takes the image
    // itself instead of "wholeSize". Set isolated to true to
    // there are no real pixels outside of the ROI
    // (so that the pixels will be extrapolated using the spe
    virtual int start(const Mat& src, const Rect& srcRoi=Rect
```

```

        bool isolated=false, int maxBufRows=-1);
// processes the next portion of the source image,
// "srcCount" rows starting from "src" and
// stores the results to "dst".
// returns the number of produced rows
virtual int proceed(const uchar* src, int srcStep, int srcCount
                    uchar* dst, int dstStep);
// higher-level function that processes the whole
// ROI or the whole image with a single call
virtual void apply( const Mat& src, Mat& dst,
                   const Rect& srcRoi=Rect(0,0,-1,-1),
                   Point dstOfs=Point(0,0),
                   bool isolated=false);
bool isSeparable() const { return filter2D.empty(); }
// how many rows from the input image are not yet processed
int remainingInputRows() const;
// how many output rows are not yet produced
int remainingOutputRows() const;
...
// the starting and the ending rows in the source image
int startY, endY;

// pointers to the filters
Ptr<BaseFilter> filter2D;
Ptr<BaseRowFilter> rowFilter;
Ptr<BaseColumnFilter> columnFilter;
};

```

---

The class `FilterEngine` can be used to apply an arbitrary filtering operation to an image. It contains all the necessary intermediate buffers, it computes extrapolated values of the “virtual” pixels outside of the image etc. Pointers to the initialized `FilterEngine` instances are returned by various `create*Filter` functions, see below, and they are used inside high-level functions such as [filter2D](#), [erode](#), [dilate](#) etc, that is, the class is the workhorse in many of OpenCV filtering functions.

This class makes it easier (though, maybe not very easy yet) to combine filtering operations with other operations, such as color space conversions, thresholding, arithmetic operations, etc. By combining several operations together you can get much better performance because your data will stay in cache. For example, below is the implementation of Laplace operator for a floating-point images, which is a simplified implementation of [Laplacian](#):

---

```

void laplace_f(const Mat& src, Mat& dst)
{
    CV_Assert( src.type() == CV_32F );
    dst.create(src.size(), src.type());

    // get the derivative and smooth kernels for d2I/dx2.

```

```

// for d2I/dy2 we could use the same kernels, just swapped
Mat kd, ks;
getSobelKernels( kd, ks, 2, 0, ksize, false, ktype );

// let's process 10 source rows at once
int DELTA = std::min(10, src.rows);
Ptr<FilterEngine> Fxx = createSeparableLinearFilter(src.type
    dst.type(), kd, ks, Point(-1,-1), 0, borderType, borderType
Ptr<FilterEngine> Fyy = createSeparableLinearFilter(src.type
    dst.type(), ks, kd, Point(-1,-1), 0, borderType, borderType

int y = Fxx->start(src), dsty = 0, dy = 0;
Fyy->start(src);
const uchar* sptr = src.data + y*src.step;

// allocate the buffers for the spatial image derivatives;
// the buffers need to have more than DELTA rows, because at the
// last iteration the output may take max(kd.rows-1,ks.rows
// rows more than the input.
Mat Ixx( DELTA + kd.rows - 1, src.cols, dst.type() );
Mat Iyy( DELTA + kd.rows - 1, src.cols, dst.type() );

// inside the loop we always pass DELTA rows to the filter
// (note that the "proceed" method takes care of possible overflow
// it was given the actual image height in the "start" method)
// on output we can get:
// * < DELTA rows (the initial buffer accumulation stage)
// * = DELTA rows (settled state in the middle)
// * > DELTA rows (then the input image is over, but we generate
// "virtual" rows using the border mode and filter
// this variable number of output rows is dy.
// dsty is the current output row.
// sptr is the pointer to the first input row in the portion to p
for( ; dsty < dst.rows; sptr += DELTA*src.step, dsty += dy
{
    Fxx->proceed( sptr, (int)src.step, DELTA, Ixx.data, (
    dy = Fyy->proceed( sptr, (int)src.step, DELTA, d2y.data
    if( dy > 0 )
    {
        Mat dstripe = dst.rowRange(dsty, dsty + dy);
        add(Ixx.rowRange(0, dy), Iyy.rowRange(0, dy), dstripe
    }
}
}
}

```

If you do not need that much control of the filtering process, you can simply use the `FilterEngine::apply` method. Here is how the method is actually implemented:

---

```

void FilterEngine::apply(const Mat& src, Mat& dst,

```

```

    const Rect& srcRoi, Point dstOfs, bool isolated)
{
    // check matrix types
    CV_Assert( src.type() == srcType && dst.type() == dstType

    // handle the "whole image" case
    Rect _srcRoi = srcRoi;
    if( _srcRoi == Rect(0,0,-1,-1) )
        _srcRoi = Rect(0,0,src.cols,src.rows);

    // check if the destination ROI is inside the dst.
    // and FilterEngine::start will check if the source ROI is inside
    CV_Assert( dstOfs.x >= 0 && dstOfs.y >= 0 &&
        dstOfs.x + _srcRoi.width <= dst.cols &&
        dstOfs.y + _srcRoi.height <= dst.rows );

    // start filtering
    int y = start(src, _srcRoi, isolated);

    // process the whole ROI. Note that "endY - startY" is the total
    // of the source rows to process
    // (including the possible rows outside of srcRoi but inside the
    proceed( src.data + y*src.step,
        (int)src.step, endY - startY,
        dst.data + dstOfs.y*dst.step +
        dstOfs.x*dst.elemSize(), (int)dst.step );
}

```

---

Unlike the earlier versions of OpenCV, now the filtering operations fully support the notion of image ROI, that is, pixels outside of the ROI but inside the image can be used in the filtering operations. For example, you can take a ROI of a single pixel and filter it - that will be a filter response at that particular pixel (however, it's possible to emulate the old behavior by passing `isolated=false` to `FilterEngine::start` or `FilterEngine::apply`). You can pass the ROI explicitly to `FilterEngine::apply`, or construct a new matrix headers:

---

```

// compute dI/dx derivative at src(x,y)

// method 1:
// form a matrix header for a single value
float val1 = 0;
Mat dst1(1,1,CV_32F,&val1);

Ptr<FilterEngine> Fx = createDerivFilter(CV_32F, CV_32F,
    1, 0, 3, BORDER_REFLECT_101);
Fx->apply(src, Rect(x,y,1,1), Point(), dst1);

// method 2:
// form a matrix header for a single value
float val2 = 0;

```

```

Mat dst2(1,1,CV_32F,&val2);

Mat pix_roi(src, Rect(x,y,1,1));
Sobel(pix_roi, dst2, dst2.type(), 1, 0, 3, 1, 0, BORDER_REFLECT_101

printf("method1 = %g, method2 = %g\n", val1, val2);

```

---

Note on the data types. As it was mentioned in [BaseFilter](#) description, the specific filters can process data of any type, despite that `Base*Filter::operator()` only takes `uchar` pointers and no information about the actual types. To make it all work, the following rules are used:

- in case of separable filtering `FilterEngine::rowFilter` applied first. It transforms the input image data (of type `srcType`) to the intermediate results stored in the internal buffers (of type `bufType`). Then these intermediate results are processed as single-channel data with `FilterEngine::columnFilter` and stored in the output image (of type `dstType`). Thus, the input type for `rowFilter` is `srcType` and the output type is `bufType`; the input type for `columnFilter` is **`CV_MAT_DEPTH(bufType)`** and the output type is **`CV_MAT_DEPTH(dstType)`**.
- in case of non-separable filtering `bufType` must be the same as `srcType`. The source data is copied to the temporary buffer if needed and then just passed to `FilterEngine::filter2D`. That is, the input type for `filter2D` is `srcType` (`=`bufType``) and the output type is `dstType`.

See also: [BaseColumnFilter](#), [BaseFilter](#), [BaseRowFilter](#), [createBoxFilter](#), [createDerivFilter](#), [createGaussianFilter](#), [createLinearFilter](#), [createMorphologyFilter](#), [createSeparableLinearFilter](#)

## bilateralFilter

---

void **bilateralFilter**(const Mat& *src*, Mat& *dst*, int *d*, double *sigmaColor*, double *sigmaSpace*, int *borderType*=`BORDER_DEFAULT`)

Applies bilateral filter to the image

- Parameters:**
- *src* – The source 8-bit or floating-point, 1-channel or 3-channel image
  - *dst* – The destination image; will have the same size and the same type as *src*
  - *d* – The diameter of each pixel neighborhood, that is used during filtering. If it is non-positive, it's computed from *sigmaSpace*
  - *sigmaColor* – Filter sigma in the color space. Larger value of the parameter means that farther colors within the pixel neighborhood (see *sigmaSpace*) will be mixed together,

resulting in larger areas of semi-equal color

- *sigmaSpace* – Filter sigma in the coordinate space. Larger value of the parameter means that farther pixels will influence each other (as long as their colors are close enough; see *sigmaColor*). Then  $d > 0$ , it specifies the neighborhood size regardless of *sigmaSpace*, otherwise  $d$  is proportional to *sigmaSpace*

The function applies bilateral filtering to the input image, as described in [http://www.dai.ed.ac.uk/CVonline/LOCAL\\_COPIES/MANDUCHI1/Bilateral\\_Filtering.html](http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html)

## blur

---

void **blur**(const Mat& *src*, Mat& *dst*, Size *ksize*, Point *anchor*=Point(-1, -1), int *borderType*=BORDER\_DEFAULT)

Smooths image using normalized box filter

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same size and the same type as *src*
  - *ksize* – The smoothing kernel size
  - *anchor* – The anchor point. The default value Point(-1, -1) means that the anchor is at the kernel center
  - *borderType* – The border mode used to extrapolate pixels outside of the image

The function smooths the image using the kernel:

$$K = \frac{1}{\text{ksize.width} * \text{ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & 1 & 1 & \dots & 1 & 1 \end{bmatrix}$$

The call `blur(src, dst, ksize, anchor, borderType)` is equivalent to `boxFilter(src, dst, src.type(), anchor, true, borderType)`.

See also: [boxFilter](#), [bilateralFilter](#), [GaussianBlur](#), [medianBlur](#).

## borderInterpolate

---

int **borderInterpolate**(int *p*, int *len*, int *borderType*)

Computes source location of extrapolated pixel

- Parameters:**
- $p$  – 0-based coordinate of the extrapolated pixel along one of the axes, likely  $<0$  or  $\geq \text{len}$
  - $len$  – length of the array along the corresponding axis
  - $borderType$  – the border type, one of the `BORDER_*`, except for `BORDER_TRANSPARENT` and `BORDER_ISOLATED`. When  $borderType == BORDER_CONSTANT$  the function always returns -1, regardless of  $p$  and  $len$

The function computes and returns the coordinate of the donor pixel, corresponding to the specified extrapolated pixel when using the specified extrapolation border mode. For example, if we use `BORDER_WRAP` mode in the horizontal direction, `BORDER_REFLECT_101` in the vertical direction and want to compute value of the “virtual” pixel `Point(-5, 100)` in a floating-point image `img`, it will be

---

```
float val = img.at<float>(borderInterpolate(100, img.rows,
borderInterpolate(-5, img.cols,
```

---

Normally, the function is not called directly; it is used inside [FilterEngine](#) and [copyMakeBorder](#) to compute tables for quick extrapolation.

See also: [FilterEngine](#), [copyMakeBorder](#)

## boxFilter

---

void **boxFilter**(const Mat& *src*, Mat& *dst*, int *ddepth*, Size *ksize*, Point *anchor=Point(-1, -1)*, bool *normalize=true*, int *borderType=BORDER\_DEFAULT*)

Smooths image using box filter

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same size and the same type as *src*
  - *ksize* – The smoothing kernel size
  - *anchor* – The anchor point. The default value `Point(-1, -1)` means that the anchor is at the kernel center
  - *normalize* – Indicates, whether the kernel is normalized by its area or not
  - *borderType* – The border mode used to extrapolate pixels outside of the image

The function smooths the image using the kernel:

$$K = \alpha \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

where

$$\alpha = \begin{cases} \frac{1}{\text{ksize.width} * \text{ksize.height}} & \text{when } \text{normalize}=\text{true} \\ 1 & \text{otherwise} \end{cases}$$

Unnormalized box filter is useful for computing various integral characteristics over each pixel neighborhood, such as covariation matrices of image derivatives (used in dense optical flow algorithms, `[conerHarris]bgroup`({Harris corner detector}) etc.). If you need to compute pixel sums over variable-size windows, use [integral](#).

See also: [boxFilter](#), [bilateralFilter](#), [GaussianBlur](#), [medianBlur](#), [integral](#).

## buildPyramid

---

void **buildPyramid**(const Mat& *src*, vector<Mat>& *dst*, int *maxlevel*)

Constructs Gaussian pyramid for an image

- Parameters:**
- *src* – The source image; check [pyrDown](#) for the list of supported types
  - *dst* – The destination vector of `maxlevel+1` images of the same type as *src*; `dst[0]` will be the same as *src*, `dst[1]` is the next pyramid layer, a smoothed and down-sized *src* etc.
  - *maxlevel* – The 0-based index of the last (i.e. the smallest) pyramid layer; it must be non-negative

The function constructs a vector of images and builds the gaussian pyramid by recursively applying [pyrDown](#) to the previously built pyramid layers, starting from `dst[0]==src`.

## copyMakeBorder

---

void **copyMakeBorder**(const Mat& *src*, Mat& *dst*, int *top*, int *bottom*, int *left*, int *right*, int *borderType*, const Scalar& *value=Scalar()*)

Forms a border around the image

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same type as *src* and the size `Size(src.cols+left+right, src.rows+top+bottom)`
  - *top*, *bottom*, *left*, *right* – Specify how much pixels in each direction from the source image rectangle one needs to extrapolate, e.g. `top=1, bottom=1, left=1, right=1` mean that 1 pixel-wide border needs to be built
  - *borderType* – The border type; see [borderInterpolate](#)
  - *value* – The border value if `borderType==BORDER_CONSTANT`

The function copies the source image into the middle of the destination image. The areas to the left, to the right, above and below the copied source image will be filled with extrapolated pixels. This is not what [FilterEngine](#) or based on it filtering functions do (they extrapolate pixels on-fly), but what other more complex functions, including your own, may do to simplify image boundary handling.

The function supports the mode when *src* is already in the middle of *dst*. In this case the function does not copy *src* itself, but simply constructs the border, e.g.:

---

```
// let border be the same in all directions
int border=2;
// constructs a larger image to fit both the image and the
Mat gray_buf(rgb.rows + border*2, rgb.cols + border*2, rgb
// select the middle part of it w/o copying data
Mat gray(gray_canvas, Rect(border, border, rgb.cols, rgb.r
// convert image from RGB to grayscale
cvtColor(rgb, gray, CV_RGB2GRAY);
// form a border in-place
copyMakeBorder(gray, gray_buf, border, border,
                border, border, BORDER_REPLICATE);
// now do some custom filtering ...
...
```

---

See also: [borderInterpolate](#)

## createBoxFilter

---

`Ptr<FilterEngine> createBoxFilter(int srcType, int dstType, Size ksize, Point anchor=Point(-1, -1), bool normalize=true, int borderType=BORDER_DEFAULT)`

Ptr<BaseRowFilter> **getRowSumFilter**(int *srcType*, int *sumType*, int *ksize*, int *anchor*=-1)

Ptr<BaseColumnFilter> **getColumnSumFilter**(int *sumType*, int *dstType*, int *ksize*, int *anchor*=-1, double *scale*=1)

Returns box filter engine

- Parameters:**
- *srcType* – The source image type
  - *sumType* – The intermediate horizontal sum type; must have as many channels as *srcType*
  - *dstType* – The destination image type; must have as many channels as *srcType*
  - *ksize* – The aperture size
  - *anchor* – The anchor position with the kernel; negative values mean that the anchor is at the kernel center
  - *normalize* – Whether the sums are normalized or not; see [boxFilter](#)
  - *scale* – Another way to specify normalization in lower-level `getColumnSumFilter`
  - *borderType* – Which border type to use; see [borderInterpolate](#)

The function is a convenience function that retrieves horizontal sum primitive filter with `getRowSumFilter`, vertical sum filter with `getColumnSumFilter`, constructs new [FilterEngine](#) and passes both of the primitive filters there. The constructed filter engine can be used for image filtering with normalized or unnormalized box filter.

The function itself is used by [blur](#) and [boxFilter](#).

See also: [FilterEngine](#), [blur](#), [boxFilter](#).

## createDerivFilter

---

Ptr<FilterEngine> **createDerivFilter**(int *srcType*, int *dstType*, int *dx*, int *dy*, int *ksize*, int *borderType*=`BORDER_DEFAULT`)

Returns engine for computing image derivatives

- Parameters:**
- *srcType* – The source image type
  - *dstType* – The destination image type; must have as many channels as *srcType*
  - *dx* – The derivative order in respect with x
  - *dy* – The derivative order in respect with y
  - *ksize* – The aperture size; see [getDerivKernels](#)

- *borderType* – Which border type to use; see [borderInterpolate](#)

The function [createDerivFilter](#) is a small convenience function that retrieves linear filter coefficients for computing image derivatives using [getDerivKernels](#) and then creates a separable linear filter with [createSeparableLinearFilter](#). The function is used by [Sobel](#) and [Scharr](#).

See also: [createSeparableLinearFilter](#), [getDerivKernels](#), [Scharr](#), [Sobel](#).

## createGaussianFilter

---

Ptr<FilterEngine> **createGaussianFilter**(int *type*, Size *ksize*, double *sigmaX*, double *sigmaY*=0, int *borderType*=BORDER\_DEFAULT)

Returns engine for smoothing images with a Gaussian filter

- Parameters:**
- *type* – The source and the destination image type
  - *ksize* – The aperture size; see [getGaussianKernel](#)
  - *sigmaX* – The Gaussian sigma in the horizontal direction; see [getGaussianKernel](#)
  - *sigmaY* – The Gaussian sigma in the vertical direction; if 0, then  $\text{sigmaY} \leftarrow \text{sigmaX}$
  - *borderType* – Which border type to use; see [borderInterpolate](#)

The function [createGaussianFilter](#) computes Gaussian kernel coefficients and then returns separable linear filter for that kernel. The function is used by [GaussianBlur](#). Note that while the function takes just one data type, both for input and output, you can pass by this limitation by calling [getGaussianKernel](#) and then [createSeparableFilter](#) directly.

See also: [createSeparableLinearFilter](#), [getGaussianKernel](#), [GaussianBlur](#).

## createLinearFilter

---

Ptr<FilterEngine> **createLinearFilter**(int *srcType*, int *dstType*, const Mat& *kernel*, Point *\_anchor*=Point(-1, -1), double *delta*=0, int *rowBorderType*=BORDER\_DEFAULT, int *columnBorderType*=-1, const Scalar& *borderValue*=Scalar())

Ptr<BaseFilter> **getLinearFilter**(int *srcType*, int *dstType*, const Mat& *kernel*, Point *anchor*=Point(-1, -1), double *delta*=0, int *bits*=0)

Creates non-separable linear filter engine

- Parameters:**
- *srcType* – The source image type
  - *dstType* – The destination image type; must have as many channels as *srcType*
  - *kernel* – The 2D array of filter coefficients
  - *anchor* – The anchor point within the kernel; special value `Point(-1, -1)` means that the anchor is at the kernel center
  - *delta* – The value added to the filtered results before storing them
  - *bits* – When the kernel is an integer matrix representing fixed-point filter coefficients, the parameter specifies the number of the fractional bits
  - *rowBorderType*, *columnBorderType* – The pixel extrapolation methods in the horizontal and the vertical directions; see [borderInterpolate](#)
  - *borderValue* – Used in case of constant border

The function returns pointer to 2D linear filter for the specified kernel, the source array type and the destination array type. The function is a higher-level function that calls `getLinearFilter` and passes the retrieved 2D filter to [FilterEngine](#) constructor.

See also: [createSeparableLinearFilter](#), [FilterEngine](#), [filter2D](#)

## createMorphologyFilter

---

`Ptr<FilterEngine>` **createMorphologyFilter**(int *op*, int *type*, const Mat& *element*, Point *anchor*=Point(-1, -1), int *rowBorderType*=BORDER\_CONSTANT, int *columnBorderType*=-1, const Scalar& *borderValue*=morphologyDefaultBorderValue())

`Ptr<BaseFilter>` **getMorphologyFilter**(int *op*, int *type*, const Mat& *element*, Point *anchor*=Point(-1, -1))

`Ptr<BaseRowFilter>` **getMorphologyRowFilter**(int *op*, int *type*, int *esize*, int *anchor*=-1)

`Ptr<BaseColumnFilter>` **getMorphologyColumnFilter**(int *op*, int *type*, int *esize*, int *anchor*=-1)

```
static inline Scalar morphologyDefaultBorderValue()
{ return Scalar::all(DBL_MAX)
```

Creates engine for non-separable morphological operations

- Parameters:**
- *op* – The morphology operation id, MORPH\_ERODE or MORPH\_DILATE
  - *type* – The input/output image type
  - *element* – The 2D 8-bit structuring element for the morphological operation. Non-zero elements indicate the pixels that belong to the element
  - *esize* – The horizontal or vertical structuring element size for separable morphological operations
  - *anchor* – The anchor position within the structuring element; negative values mean that the anchor is at the center
  - *rowBorderType*, *columnBorderType* – The pixel extrapolation methods in the horizontal and the vertical directions; see [borderInterpolate](#)
  - *borderValue* – The border value in case of a constant border. The default value, ``morphologyDefaultBorderValue``, has the special meaning. It is transformed  $+\text{inf}$  for the erosion and to  $-\text{inf}$  for the dilation, which means that the minimum (maximum) is effectively computed only over the pixels that are inside the image.

The functions construct primitive morphological filtering operations or a filter engine based on them. Normally it's enough to use [createMorphologyFilter](#) or even higher-level [erode](#), [dilate](#) or [morphologyEx](#). Note, that [createMorphologyFilter](#) analyses the structuring element shape and builds a separable morphological filter engine when the structuring element is square.

See also: [erode](#), [dilate](#), [morphologyEx](#), [FilterEngine](#)

## createSeparableLinearFilter

---

```
Ptr<FilterEngine> createSeparableLinearFilter(int
srcType, int dstType, const Mat& rowKernel, const Mat& columnKernel, Point
anchor=Point(-1, -1), double delta=0, int rowBorderType=BORDER_DEFAULT, int
columnBorderType=-1, const Scalar& borderValue=Scalar())
```

```
Ptr<BaseColumnFilter> getLinearColumnFilter(int bufType, int
dstType, const Mat& columnKernel, int anchor, int symmetryType, double delta=0, int
bits=0)
```

Ptr<BaseRowFilter> **getLinearRowFilter**(int *srcType*, int *bufType*, const Mat& *rowKernel*, int *anchor*, int *symmetryType*)

Creates engine for separable linear filter

- Parameters:**
- *srcType* – The source array type
  - *dstType* – The destination image type; must have as many channels as *srcType*
  - *bufType* – The intermediate buffer type; must have as many channels as *srcType*
  - *rowKernel* – The coefficients for filtering each row
  - *columnKernel* – The coefficients for filtering each column
  - *anchor* – The anchor position within the kernel; negative values mean that anchor is positioned at the aperture center
  - *delta* – The value added to the filtered results before storing them
  - *bits* – When the kernel is an integer matrix representing fixed-point filter coefficients, the parameter specifies the number of the fractional bits
  - *rowBorderType*, *columnBorderType* – The pixel extrapolation methods in the horizontal and the vertical directions; see [borderInterpolate](#)
  - *borderValue* – Used in case of a constant border
  - *symmetryType* – The type of each of the row and column kernel; see [getKernelType](#).

The functions construct primitive separable linear filtering operations or a filter engine based on them. Normally it's enough to use [createSeparableLinearFilter](#) or even higher-level [sepFilter2D](#). The function [createMorphologyFilter](#) is smart enough to figure out the *symmetryType* for each of the two kernels, the intermediate *bufType*, and, if the filtering can be done in integer arithmetics, the number of *bits* to encode the filter coefficients. If it does not work for you, it's possible to call [getLinearColumnFilter](#), [getLinearRowFilter](#) directly and then pass them to [FilterEngine](#) constructor.

See also: [sepFilter2D](#), [createLinearFilter](#), [FilterEngine](#), [getKernelType](#)

## dilate

---

void **dilate**(const Mat& *src*, Mat& *dst*, const Mat& *element*, Point *anchor*=Point(-1, -1), int *iterations*=1, int *borderType*=BORDER\_CONSTANT, const Scalar& *borderValue*=morphologyDefaultBorderValue())

Dilates an image by using a specific structuring element.

- Parameters:**
- *src* – The source image
  - *dst* – The destination image. It will have the same size and the same type as *src*
  - *element* – The structuring element used for dilation. If `element=Mat()`, a  $3 \times 3$  rectangular structuring element is used
  - *anchor* – Position of the anchor within the element. The default value  $(-1, -1)$  means that the anchor is at the element center
  - *iterations* – The number of times dilation is applied
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)
  - *borderValue* – The border value in case of a constant border. The default value has a special meaning, see [createMorphologyFilter](#)

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:

$$\text{dst}(x, y) = \max_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

The function supports the in-place mode. Dilation can be applied several (*iterations*) times. In the case of multi-channel images each channel is processed independently.

See also: [erode](#), [morphologyEx](#), [createMorphologyFilter](#)

## erode

---

void **erode**(const Mat& *src*, Mat& *dst*, const Mat& *element*, Point *anchor*=Point(-1, -1), int *iterations*=1, int *borderType*=BORDER\_CONSTANT, const Scalar& *borderValue*=*morphologyDefaultBorderValue*())

Erodes an image by using a specific structuring element.

- Parameters:**
- *src* – The source image
  - *dst* – The destination image. It will have the same size and the same type as *src*
  - *element* – The structuring element used for dilation. If `element=Mat()`, a  $3 \times 3$  rectangular structuring element is used
  - *anchor* – Position of the anchor within the element. The default value  $(-1, -1)$  means that the anchor is at the element center
  - *iterations* – The number of times erosion is applied
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)
  - *borderValue* – The border value in case of a constant border.

The default value has a special meaning, see [createMorphologyFilter](#)

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:

$$\text{dst}(x, y) = \min_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. In the case of multi-channel images each channel is processed independently.

See also: [dilate](#), [morphologyEx](#), [createMorphologyFilter](#)

## filter2D

---

void **filter2D**(const Mat& *src*, Mat& *dst*, int *ddepth*, const Mat& *kernel*, Point *anchor*=Point(-1, -1), double *delta*=0, int *borderType*=BORDER\_DEFAULT)

Convolve an image with the kernel

- Parameters:**
- *src* – The source image
  - *dst* – The destination image. It will have the same size and the same number of channels as `src`
  - *ddepth* – The desired depth of the destination image. If it is negative, it will be the same as `src.depth()`
  - *kernel* – Convolution kernel (or rather a correlation kernel), a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using [split](#) and process them individually
  - *anchor* – The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor should lie within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center
  - *delta* – The optional value added to the filtered pixels before storing them in `dst`
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)

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 according to the specified border mode.

The function does actually computes correlation, not the convolution:

$$\text{dst}(x, y) = \sum_{\substack{0 \leq x' < \text{kernel.cols}, \\ 0 \leq y' < \text{kernel.rows}}} \text{kernel}(x', y') * \text{src}(x + x' - \text{anchor.x}, y + y' - \text{anchor.y})$$

That is, the kernel is not mirrored around the anchor point. If you need a real convolution, flip the kernel using [flip](#) and set the new anchor to  $(\text{kernel.cols} - \text{anchor.x} - 1, \text{kernel.rows} - \text{anchor.y} - 1)$ .

The function uses [dft]bgroup({DFT})-based algorithm in case of sufficiently large kernels ( $11 \times 11$ ) and the direct algorithm (that uses the engine retrieved by [createLinearFilter](#)) for small kernels.

See also: [sepFilter2D](#), [createLinearFilter](#), [dft](#), [matchTemplate](#)

## GaussianBlur

---

void **GaussianBlur**(const Mat& *src*, Mat& *dst*, Size *ksize*, double *sigmaX*, double *sigmaY*=0, int *borderType*=BORDER\_DEFAULT)

Smooths image using a Gaussian filter

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same size and the same type as *src*
  - *ksize* – The Gaussian kernel size; *ksize.width* and *ksize.height* can differ, but they both must be positive and odd. Or, they can be zero's, then they are computed from  $\text{sigma}^*$
  - *sigmaX*, *sigmaY* – The Gaussian kernel standard deviations in X and Y direction. If *sigmaY* is zero, it is set to be equal to *sigmaX*. If they are both zeros, they are computed from *ksize.width* and *ksize.height*, respectively, see [getGaussianKernel](#). To fully control the result regardless of possible future modification of all this semantics, it is recommended to specify all of *ksize*, *sigmaX* and *sigmaY*
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)

The function convolves the source image with the specified Gaussian kernel. In-place filtering is supported.

See also: [sepFilter2D](#), [filter2D](#), [blur](#), [boxFilter](#), [bilateralFilter](#), [medianBlur](#)

## getDerivKernels

---

void **getDerivKernels**(Mat& *kx*, Mat& *ky*, int *dx*, int *dy*, int *ksize*, bool *normalize=false*, int *ktype=CV\_32F*)

Returns filter coefficients for computing spatial image derivatives

- Parameters:**
- *kx* – The output matrix of row filter coefficients; will have type *ktype*
  - *ky* – The output matrix of column filter coefficients; will have type *ktype*
  - *dx* – The derivative order in respect with x
  - *dy* – The derivative order in respect with y
  - *ksize* – The aperture size. It can be **CV\_SCHARR**, 1, 3, 5 or 7
  - *normalize* – Indicates, whether to normalize (scale down) the filter coefficients or not. In theory the coefficients should have the denominator  $= 2^{ksize*2-dx-dy-2}$ . If you are going to filter floating-point images, you will likely want to use the normalized kernels. But if you compute derivatives of a 8-bit image, store the results in 16-bit image and wish to preserve all the fractional bits, you may want to set *normalize=false*.
  - *ktype* – The type of filter coefficients. It can be **CV\_32f** or **CV\_64F**

The function computes and returns the filter coefficients for spatial image derivatives. When *ksize=CV\_SCHARR*, the Scharr  $3 \times 3$  kernels are generated, see [Scharr](#). Otherwise, Sobel kernels are generated, see [Sobel](#). The filters are normally passed to [sepFilter2D](#) or to [createSeparableLinearFilter](#).

## getGaussianKernel

---

Mat **getGaussianKernel**(int *ksize*, double *sigma*, int *ktype=CV\_64F*)

Returns Gaussian filter coefficients

- Parameters:**
- *ksize* – The aperture size. It should be odd ( $ksize \bmod 2 = 1$ ) and positive.
  - *sigma* – The Gaussian standard deviation. If it is non-positive, it is computed from *ksize* as  $sigma = 0.3 * (ksize/2 - 1) + 0.8$
  - *ktype* – The type of filter coefficients. It can be **CV\_32f** or **CV\_64F**

## CV\_64F

The function computes and returns the `ksize × 1` matrix of Gaussian filter coefficients:

$$G_i = \alpha * e^{-\frac{(i-(ksize-1)/2)^2}{(2*\sigma)^2}},$$

where  $i = 0..ksize - 1$  and  $\alpha$  is the scale factor chosen so that  $\sum_i G_i = 1$

Two of such generated kernels can be passed to [sepFilter2D](#) or to [createSeparableLinearFilter](#) that will automatically detect that these are smoothing kernels and handle them accordingly. Also you may use the higher-level [GaussianBlur](#).

See also: [sepFilter2D](#), [createSeparableLinearFilter](#), [getDerivKernels](#), [getStructuringElement](#), [GaussianBlur](#).

## getKernelType

---

int **getKernelType**(const Mat& *kernel*, Point *anchor*)

Returns the kernel type

**Parameters:**

- *kernel* – 1D array of the kernel coefficients to analyze
- *anchor* – The anchor position within the kernel

The function analyzes the kernel coefficients and returns the corresponding kernel type:

- **KERNEL\_GENERAL** - Generic kernel - when there is no any type of symmetry or other properties
- **KERNEL\_SYMMETRICAL** - The kernel is symmetrical: `kerneli == kernelksize-i-1` and the anchor is at the center
- **KERNEL\_ASYMMETRICAL** - The kernel is asymmetrical: `kerneli == -kernelksize-i-1` and the anchor is at the center
- **KERNEL\_SMOOTH** - All the kernel elements are non-negative and sum to 1. E.g. the Gaussian kernel is both smooth kernel and symmetrical, so the function will return `KERNEL_SMOOTH | KERNEL_SYMMETRICAL`
- **KERNEL\_INTEGER** - All the kernel coefficients are integer numbers. This flag can be combined with `KERNEL_SYMMETRICAL` or `KERNEL_ASYMMETRICAL`

## getStructuringElement

---

Mat **getStructuringElement**(int *shape*, Size *esize*, Point *anchor*=Point(-1, -1))

Returns the structuring element of the specified size and shape for morphological operations

**Parameters:**

- *shape* –

The element shape, one of:

- MORPH\_RECT - rectangular structuring element

$$E_{ij} = 1$$

- MORPH\_ELLIPSE - elliptic structuring element, i.e. a filled ellipse inscribed into the rectangle `Rect(0, 0, esize.width, 0. esize.height)`
- MORPH\_CROSS - cross-shaped structuring element:

$$E_{ij} = \begin{cases} 1 & \text{if } i=\text{anchor.y} \text{ or } j=\text{anchor.x} \\ 0 & \text{otherwise} \end{cases}$$

- *esize* – Size of the structuring element
- *anchor* – The anchor position within the element. The default value `(-1, -1)` means that the anchor is at the center. Note that only the cross-shaped element's shape depends on the anchor position; in other cases the anchor just regulates by how much the result of the morphological operation is shifted

The function constructs and returns the structuring element that can be then passed to [createMorphologyFilter](#), [erode](#), [dilate](#) or [morphologyEx](#). But also you can construct an arbitrary binary mask yourself and use it as the structuring element.

## medianBlur

---

void **medianBlur**(const Mat& *src*, Mat& *dst*, int *ksize*)

Smooths image using median filter

- Parameters:**
- *src* – The source 1-, 3- or 4-channel image. When *ksize* is 3 or 5, the image depth should be **CV\_8U**, **CV\_16U** or **CV\_32F**. For larger aperture sizes it can only be **CV\_8U**
  - *dst* – The destination array; will have the same size and the same type as *src*
  - *ksize* – The aperture linear size. It must be odd and more than 1, i.e. 3, 5, 7 ...

The function smooths image using the median filter with `ksize × ksize` aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

See also: [bilateralFilter](#), [blur](#), [boxFilter](#), [GaussianBlur](#)

## morphologyEx

---

void **morphologyEx**(const Mat& *src*, Mat& *dst*, int *op*, const Mat& *element*, Point *anchor*=Point(-1, -1), int *iterations*=1, int *borderType*=BORDER\_CONSTANT, const Scalar& *borderValue*=morphologyDefaultBorderValue())

Performs advanced morphological transformations

- Parameters:**
- *src* – Source image
  - *dst* – Destination image. It will have the same size and the same type as *src*
  - *element* – Structuring element
  - *op* –  
Type of morphological operation, one of the following:
    - **MORPH\_OPEN** - opening
    - **MORPH\_CLOSE** - closing
    - **MORPH\_GRADIENT** - morphological gradient
    - **MORPH\_TOPHAT** - “top hat”
    - **MORPH\_BLACKHAT** - “black hat”
  - *iterations* – Number of times erosion and dilation are applied
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)
  - *borderValue* – The border value in case of a constant border. The default value has a special meaning, see [createMorphologyFilter](#)

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$\text{dst} = \text{open}(\text{src}, \text{element}) = \text{dilate}(\text{erode}(\text{src}, \text{element}))$$

Closing:

$$\text{dst} = \text{close}(\text{src}, \text{element}) = \text{erode}(\text{dilate}(\text{src}, \text{element}))$$

Morphological gradient:

$$\text{dst} = \text{morph\_grad}(\text{src}, \text{element}) = \text{dilate}(\text{src}, \text{element}) - \text{erode}(\text{src}, \text{element})$$

“Top hat”:

$$\text{dst} = \text{tophat}(\text{src}, \text{element}) = \text{src} - \text{open}(\text{src}, \text{element})$$

“Black hat”:

$$\text{dst} = \text{blackhat}(\text{src}, \text{element}) = \text{close}(\text{src}, \text{element}) - \text{src}$$

Any of the operations can be done in-place.

See also: [dilate](#), [erode](#), [createMorphologyFilter](#)

## Laplacian

---

void **Laplacian**(const Mat& *src*, Mat& *dst*, int *ddepth*, int *ksize*=1, double *scale*=1, double *delta*=0, int *borderType*=BORDER\_DEFAULT)

Calculates the Laplacian of an image

- Parameters:**
- *src* – Source image
  - *dst* – Destination image; will have the same size and the same number of channels as *src*
  - *ddepth* – The desired depth of the destination image
  - *ksize* – The aperture size used to compute the second-derivative filters, see [getDerivKernels](#). It must be positive and odd
  - *scale* – The optional scale factor for the computed Laplacian values (by default, no scaling is applied, see [getDerivKernels](#))
  - *delta* – The optional delta value, added to the results prior to storing them in *dst*
  - *borderType* – The pixel extrapolation method, see [borderInterpolate](#)

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\text{dst} = \Delta \text{src} = \frac{\partial^2 \text{src}}{\partial x^2} + \frac{\partial^2 \text{src}}{\partial y^2}$$

This is done when *ksize* > 1. When *ksize* == 1, the Laplacian is computed by filtering the image with the following 3 × 3 aperture:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

See also: [Sobel](#), [Scharr](#)

## pyrDown

---

void **pyrDown**(const Mat& *src*, Mat& *dst*, const Size& *dsize*=Size())

Smooths an image and downsamples it.

- Parameters:**
- *src* – The source image
  - *dst* – The destination image. It will have the specified size and the same type as *src*
  - *dsize* – Size of the destination image. By default it is computed as `Size((src.cols+1)/2, (src.rows+1)/2)`. But in any case the following conditions should be satisfied:

$$\begin{aligned} |\text{dsize.width} * 2 - \text{src.cols}| &\leq 2 \\ |\text{dsize.height} * 2 - \text{src.rows}| &\leq 2 \end{aligned}$$

The function performs the downsampling step of the Gaussian pyramid construction. First it convolves the source image with the kernel:

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

and then downsamples the image by rejecting even rows and columns.

## pyrUp

---

void **pyrUp**(const Mat& *src*, Mat& *dst*, const Size& *dsize*=Size())

Upsamples an image and then smooths it

- Parameters:**
- *src* – The source image
  - *dst* – The destination image. It will have the specified size and the same type as *src*
  - *dsize* – Size of the destination image. By default it is computed as `Size(src.cols*2, src.rows*2)`. But in any case the following conditions should be satisfied:

$$\begin{aligned} |\text{dsize.width} - \text{src.cols} * 2| &\leq (\text{dsize.width} \bmod 2) \\ |\text{dsize.height} - \text{src.rows} * 2| &\leq (\text{dsize.height} \bmod 2) \end{aligned}$$

The function performs the upsampling step of the Gaussian pyramid construction (it can actually be used to construct the Laplacian pyramid). First it upsamples the source image by injecting even zero rows and columns and then convolves the result with the same kernel as in [pyrDown](#), multiplied by 4.

## sepFilter2D

---

void **sepFilter2D**(const Mat& *src*, Mat& *dst*, int *ddepth*, const Mat& *rowKernel*, const Mat& *columnKernel*, Point *anchor*=Point(-1, -1), double *delta*=0, int *borderType*=BORDER\_DEFAULT)

Applies separable linear filter to an image

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same size and the same number of channels as *src*
  - *ddepth* – The destination image depth
  - *rowKernel* – The coefficients for filtering each row
  - *columnKernel* – The coefficients for filtering each column
  - *anchor* – The anchor position within the kernel; The default value (-1, 1) means that the anchor is at the kernel center
  - *delta* – The value added to the filtered results before storing them
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)

The function applies a separable linear filter to the image. That is, first, every row of *src* is filtered with 1D kernel *rowKernel*. Then, every column of the result is filtered with 1D kernel *columnKernel* and the final result shifted by *delta* is stored in *dst*.

See also: [createSeparableLinearFilter](#), [filter2D](#), [Sobel](#), [GaussianBlur](#), [boxFilter](#), [blur](#).

## Sobel

---

void **Sobel**(const Mat& *src*, Mat& *dst*, int *ddepth*, int *xorder*, int *yorder*, int *ksize*=3, double *scale*=1, double *delta*=0, int *borderType*=BORDER\_DEFAULT)

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

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same size and the same number of channels as *src*
  - *ddepth* – The destination image depth
  - *xorder* – Order of the derivative x
  - *yorder* – Order of the derivative y
  - *ksize* – Size of the extended Sobel kernel, must be 1, 3, 5 or 7

- *scale* – The optional scale factor for the computed derivative values (by default, no scaling is applied, see [getDerivKernels](#))
- *delta* – The optional delta value, added to the results prior to storing them in `dst`
- *borderType* – The pixel extrapolation method, see [borderInterpolate](#)

In all cases except 1, an `ksize` × `ksize` separable kernel will be used to calculate the derivative. When `ksize` = 1, a 3 × 1 or 1 × 3 kernel will be used (i.e. no Gaussian smoothing is done). `ksize` = 1 can only be used for the first or the second x- or y- derivatives.

There is also the special value `ksize` = `CV_SCHARR` (-1) that corresponds to a 3 × 3 Scharr filter that may give more accurate results than a 3 × 3 Sobel. The 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:

$$\mathbf{dst} = \frac{\partial^{xorder+yorder} \mathbf{src}}{\partial x^{xorder} \partial y^{yorder}}$$

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, `ksize` = 3) or (`xorder` = 0, `yorder` = 1, `ksize` = 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:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

See also: [Scharr](#), [Lapacian](#), [sepFilter2D](#), [filter2D](#), [GaussianBlur](#)

## Scharr

---

void **Scharr**(const Mat& *src*, Mat& *dst*, int *ddepth*, int *xorder*, int *yorder*, double *scale*=1, double *delta*=0, int *borderType*=*BORDER\_DEFAULT*)

Calculates the first x- or y- image derivative using Scharr operator

- Parameters:**
- *src* – The source image
  - *dst* – The destination image; will have the same size and the same number of channels as *src*
  - *ddepth* – The destination image depth
  - *xorder* – Order of the derivative x
  - *yorder* – Order of the derivative y
  - *scale* – The optional scale factor for the computed derivative values (by default, no scaling is applied, see [getDerivKernels](#))
  - *delta* – The optional delta value, added to the results prior to storing them in *dst*
  - *borderType* – The pixel extrapolation method, see [borderInterpolate](#)

The function computes the first x- or y- spatial image derivative using Scharr operator. The call

```
Scharr(src, dst, ddepth, xorder, yorder, scale, delta, borderType)
```

is equivalent to

```
Sobel(src, dst, ddepth, xorder, yorder, CV_SCHARR, scale, delta
```

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# 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:

$$\text{dst}(x, y) = \text{src}(f_x(x, y), f_y(x, y))$$

In the case when the user specifies the forward mapping:  $\langle g_x, g_y \rangle : \text{src} \rightarrow \text{dst}$ , the OpenCV functions first compute the corresponding inverse mapping:  $\langle f_x, f_y \rangle : \text{dst} \rightarrow \text{src}$  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:

- 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.
- 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 `bgroup` ([http://en.wikipedia.org/wiki/Multivariate\\_interpolation](http://en.wikipedia.org/wiki/Multivariate_interpolation)) `bgroup` (`{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*.

## convertMaps

---

void **convertMaps**(const Mat& *map1*, const Mat& *map2*, Mat& *dstmap1*, Mat& *dstmap2*, int *dstmap1type*, bool *nninterpolation=false*)

Converts image transformation maps from one representation to another

- Parameters:**
- *map1* – The first input map of type **CV\_16SC2** or **CV\_32FC1** or **CV\_32FC2**
  - *map2* – The second input map of type **CV\_16UC1** or **CV\_32FC1** or none (empty matrix), respectively
  - *dstmap1* – The first output map; will have type *dstmap1type* and the same size as *src*
  - *dstmap2* – The second output map
  - *dstmap1type* – The type of the first output map; should be **CV\_16SC2**, **CV\_32FC1** or **CV\_32FC2**
  - *nninterpolation* – Indicates whether the fixed-point maps will be used for nearest-neighbor or for more complex interpolation

The function converts a pair of maps for [remap](#) from one representation to another. The following options  $(map1.type(), map2.type()) \rightarrow (dstmap1.type(), dstmap2.type())$  are supported:

- $(CV\_32FC1, CV\_32FC1) \rightarrow (CV\_16SC2, CV\_16UC1)$ . This is the most frequently used conversion operation, in which the original floating-point maps (see [remap](#)) are converted to more compact and much faster fixed-point representation. The first output array will contain the rounded coordinates and the second array (created only when *nninterpolation=false*) will contain indices in the interpolation tables.
- $(CV\_32FC2) \rightarrow (CV\_16SC2, CV\_16UC1)$ . The same as above, but the original maps are stored in one 2-channel matrix.
- the reverse conversion. Obviously, the reconstructed floating-point maps will not be exactly the same as the originals.

See also: [remap](#), [undisort](#), [initUndistortRectifyMap](#)

## getAffineTransform

---

Mat **getAffineTransform**(const Point2f *src*[], const Point2f *dst*[])

Calculates the affine transform from 3 pairs of the corresponding points

- Parameters:**
- *src* – Coordinates of a triangle vertices in the source image
  - *dst* – Coordinates of the corresponding triangle vertices in the destination image

The function calculates the  $2 \times 3$  matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \text{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$\text{dst}(i) = (x'_i, y'_i), \text{src}(i) = (x_i, y_i), i = 0, 1, 2$$

See also: [warpAffine](#), [transform](#)

## getPerspectiveTransform

---

Mat **getPerspectiveTransform**(const Point2f *src*[], const Point2f *dst*[])

Calculates the perspective transform from 4 pairs of the corresponding points

**Parameters:**

- *src* – Coordinates of a quadrangle vertices in the source image
- *dst* – Coordinates of the corresponding quadrangle vertices in the destination image

The function calculates the  $3 \times 3$  matrix of a perspective transform such that:

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \text{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$\text{dst}(i) = (x'_i, y'_i), \text{src}(i) = (x_i, y_i), i = 0, 1, 2$$

See also: [findHomography](#), [warpPerspective](#), [perspectiveTransform](#)

## getRectSubPix

---

void **getRectSubPix**(const Mat& *image*, Size *patchSize*, Point2f *center*, Mat& *dst*, int *patchType*=-1)

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

**Parameters:**

- *src* – Source image
- *patchSize* – Size of the extracted patch
- *center* – Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image
- *dst* – The extracted patch; will have the size *patchSize*

and the same number of channels as `src`

- `patchType` – The depth of the extracted pixels. By default they will have the same depth as `src`

The function `getRectSubPix` extracts pixels from `src`:

$$dst(x, y) = src(x + center.x - (dst.cols - 1) * 0.5, y + center.y - (dst.r$$

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 (see [borderInterpolate](#)) is used to extrapolate the pixel values outside of the image.

See also: [warpAffine](#), [warpPerspective](#)

## getRotationMatrix2D

---

Mat **getRotationMatrix2D**(Point2f *center*, double *angle*, double *scale*)

Calculates the affine matrix of 2d rotation.

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

The function calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1 - \alpha) \cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x - (1 - \alpha) \cdot center.y \end{bmatrix}$$

where

$$\begin{aligned} \alpha &= scale \cdot \cos angle, \\ \beta &= scale \cdot \sin angle \end{aligned}$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

See also: [getAffineTransform](#), [warpAffine](#), [transform](#)

## invertAffineTransform

---

void **invertAffineTransform**(const Mat& *M*, Mat& *iM*)

Inverts an affine transformation

- Parameters:**
- *M* – The original affine transformation
  - *iM* – The output reverse affine transformation

The function computes inverse affine transformation represented by  $2 \times 3$  matrix *M*:

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

The result will also be a  $2 \times 3$  matrix of the same type as *M*.

## remap

---

void **remap**(const Mat& *src*, Mat& *dst*, const Mat& *map1*, const Mat& *map2*, int *interpolation*, int *borderMode*=*BORDER\_CONSTANT*, const Scalar& *borderValue*=*Scalar()*)

Applies a generic geometrical transformation to an image.

- Parameters:**
- *src* – Source image
  - *dst* – Destination image. It will have the same size as *map1* and the same type as *src*
  - *map1* – The first map of type **CV\_16SC2**, **CV\_32FC1** or **CV\_32FC2**. See [convertMaps](#)
  - *map2* – The second map of type **CV\_16UC1**, **CV\_32FC1** or none (empty map), respectively
  - *interpolation* – The interpolation method, see [resize](#). The method **INTER\_AREA** is not supported by this function
  - *borderMode* – The pixel extrapolation method, see [borderInterpolate](#). When the `borderMode=BORDER_TRANSPARENT`, it means that the pixels in the destination image that corresponds to the “outliers” in the source image are not modified by the function
  - *borderValue* – A value used in the case of a constant border. By default it is 0

The function `remap` transforms the source image using the specified map:

$$\text{dst}(x, y) = \text{src}(\text{map}_x(x, y), \text{map}_y(x, y))$$

Where values of pixels with non-integer coordinates are computed using one of the available interpolation methods. *map<sub>x</sub>* and *map<sub>y</sub>* can be encoded as separate floating-point maps, interleaved floating-point maps or fixed-point maps. The function can not operate in-place.

## resize

---

void **resize**(const Mat& *src*, Mat& *dst*, Size *dsize*, double *fx*=0, double *fy*=0, int *interpolation*=INTER\_LINEAR)

Resizes an image

- Parameters:**
- *src* – Source image
  - *dst* – Destination image. It will have size *dsize* (when it is non-zero) or the size computed from `src.size()` and *fx* and *fy*. The type of *dst* will be the same as of *src*.
  - *dsize* – The destination image size. If it is zero, then it is computed as:  

$$dsize = \text{Size}(\text{round}(fx * src.cols), \text{round}(fy * src.rows))$$

. Either *dsize* or both *fx* or *fy* must be non-zero.

**param** *fx* The scale factor along the horizontal axis. When 0, it is computed as

**fx:**  $(double)dsize.width/src.cols$

**param** *fy* The scale factor along the vertical axis. When 0, it is computed as

**fy:**  $(double)dsize.height/src.rows$

**param interpolation:**

The interpolation method:

- **INTER\_NEAREST** - nearest-neighbor interpolation
- **INTER\_LINEAR** - bilinear interpolation (used by default)
- **INTER\_AREA** - resampling using pixel area relation. It may be the preferred method for image decimation, as it gives moire-free results. But when the image is zoomed, it is similar to the INTER\_NEAREST method
- **INTER\_CUBIC** - bicubic interpolation over 4x4 pixel neighborhood
- **INTER\_LANCZOS4** - Lanczos interpolation over 8x8 pixel neighborhood

The function `resize` resizes an image *src* down to or up to the specified size. Note that the initial *dst* type or size are not taken into account. Instead the size and type are derived from the *src*, *dsize*, *fx* and *fy*. If you want to resize *src* so that it fits the pre-created *dst*, you may call the function as:

---

```
// explicitly specify dsize=dst.size(); fx and fy will be
resize(src, dst, dst.size(), 0, 0, interpolation);
```

---

If you want to decimate the image by factor of 2 in each direction, you can call the function this way:

---

```
// specify fx and fy and let the function to compute the c
resize(src, dst, Size(), 0.5, 0.5, interpolation);
```

---

See also: [warpAffine](#), [warpPerspective](#), [remap](#).

## warpAffine

---

void **warpAffine**(const Mat& src, Mat& dst, const Mat& M, Size dsize, int flags=INTER\_LINEAR, int borderMode=BORDER\_CONSTANT, const Scalar& borderValue=Scalar())

Applies an affine transformation to an image.

- Parameters:**
- *src* – Source image
  - *dst* – Destination image; will have size *dsize* and the same type as *src*
  - *M* –  $2 \times 3$  transformation matrix
  - *dsize* – Size of the destination image
  - *flags* – A combination of interpolation methods, see [resize](#), and the optional flag WARP\_INVERSE\_MAP that means that *M* is the inverse transformation (*dst* → *src*)
  - *borderMode* – The pixel extrapolation method, see [borderInterpolate](#). When the *borderMode*=BORDER\_TRANSPARENT, it means that the pixels in the destination image that corresponds to the “outliers” in the source image are not modified by the function
  - *borderValue* – A value used in case of a constant border. By default it is 0

The function `warpAffine` transforms the source image using the specified matrix:

$$\text{dst}(x, y) = \text{src}(M_{11}x + M_{12}y + M_{13}, M_{21}x + M_{22}y + M_{23})$$

when the flag WARP\_INVERSE\_MAP is set. Otherwise, the transformation is first inverted with [invertAffineTransform](#) and then put in the formula above instead of *M*. The function can not operate in-place.

See also: [warpPerspective](#), [resize](#), [remap](#), [getRectSubPix](#), [transform](#)

## warpPerspective

---

void **warpPerspective**(const Mat& src, Mat& dst, const Mat& M, Size dsize, int flags=INTER\_LINEAR, int borderMode=BORDER\_CONSTANT, const Scalar& borderValue=Scalar())

Applies a perspective transformation to an image.

- Parameters:**
- *src* – Source image
  - *dst* – Destination image; will have size `dsize` and the same type as `src`
  - $M$  –  $3 \times 3$  transformation matrix
  - *dsize* – Size of the destination image
  - *flags* – A combination of interpolation methods, see [resize](#), and the optional flag `WARP_INVERSE_MAP` that means that  $M$  is the inverse transformation ( $\text{dst} \rightarrow \text{src}$ )
  - *borderMode* – The pixel extrapolation method, see [borderInterpolate](#). When the `borderMode=BORDER_TRANSPARENT`, it means that the pixels in the destination image that corresponds to the “outliers” in the source image are not modified by the function
  - *borderValue* – A value used in case of a constant border. By default it is 0

The function `warpPerspective` transforms the source image using the specified matrix:

$$\text{dst}(x, y) = \text{src} \left( \frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right)$$

when the flag `WARP_INVERSE_MAP` is set. Otherwise, the transformation is first inverted with [invert](#) and then put in the formula above instead of  $M$ . The function can not operate in-place.

See also: [warpAffine](#), [resize](#), [remap](#), [getRectSubPix](#), [perspectiveTransform](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Miscellaneous Image Transformations

## adaptiveThreshold

void **adaptiveThreshold**(const Mat& *src*, Mat& *dst*, double *maxValue*, int *adaptiveMethod*, int *thresholdType*, int *blockSize*, double *C*)

Applies an adaptive threshold to an array.

- Parameters:**
- *src* – Source 8-bit single-channel image
  - *dst* – Destination image; will have the same size and the same type as *src*
  - *maxValue* – The non-zero value assigned to the pixels for which the condition is satisfied. See the discussion
  - *adaptiveMethod* – Adaptive thresholding algorithm to use, \* **ADAPTIVE\_THRESH\_MEAN\_C** - o

r

- **ADAPTIVE\_THRESH\_GAUSSIAN\_C** - (

see the discussion)

**param thresholdType:**

Thresholding type; must be one of \* **THRESH\_BINARY** - o

r

- **THRESH\_BINARY\_INV** - None

**param blockSize:**

The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

**param C:** The constant subtracted from the mean or weighted mean (see the discussion); normally, it's positive, but may be zero or negative as well

The function transforms a grayscale image to a binary image according to the formulas:

- **THRESH\_BINARY** -

$$dst(x, y) = \begin{cases} \text{maxValue} & \text{if } src(x, y) > T(x, y) \\ 0 & \text{otherwise} \end{cases}$$

- **THRESH\_BINARY\_INV** -

$$dst(x, y) = \begin{cases} 0 & \text{if } src(x, y) > T(x, y) \\ \text{maxValue} & \text{otherwise} \end{cases}$$

where  $T(x, y)$  is a threshold calculated individually for each pixel.

- For the method `ADAPTIVE_THRESH_MEAN_C` the threshold value  $T(x, y)$  is the mean of a `blockSize` × `blockSize` neighborhood of  $(x, y)$ , minus `C`.
- For the method `ADAPTIVE_THRESH_GAUSSIAN_C` the threshold value  $T(x, y)$  is the weighted sum (i.e. cross-correlation with a Gaussian window) of a `blockSize` × `blockSize` neighborhood of  $(x, y)$ , minus `C`. The default sigma (standard deviation) is used for the specified `blockSize`, see [getGaussianKernel](#).

The function can process the image in-place.

See also: [threshold](#), [blur](#), [GaussianBlur](#)

## cvtColor

---

void **cvtColor**(const Mat& *src*, Mat& *dst*, int *code*, int *dstCn*=0)

Converts image from one color space to another

- Parameters:**
- *src* – The source image, 8-bit unsigned, 16-bit unsigned (`CV_16UC...`) or single-precision floating-point
  - *dst* – The destination image; will have the same size and the same depth as *src*
  - *code* – The color space conversion code; see the discussion
  - *dstCn* – The number of channels in the destination image; if the parameter is 0, the number of the channels will be derived automatically from *src* and the *code*

The function converts the input image from one color space to another. In the case of transformation to-from RGB color space the ordering of the channels should be specified explicitly (RGB or BGR).

The conventional ranges for R, G and B channel values are:

- 0 to 255 for `CV_8U` images
- 0 to 65535 for `CV_16U` images and
- 0 to 1 for `CV_32F` images.

Of course, in the case of linear transformations the range does not matter, but in the non-linear cases the input RGB image should be normalized to the proper value range in order to get the correct results, e.g. for RGB: *math: \rightarrow L^\*u^\*v^\** transformation. For example, if you have a 32-bit floating-point image directly converted from 8-bit image without any scaling, then it will have 0..255 value range, instead of the assumed by the function 0..1. So, before calling `:func: cvtColor()`, you need first to scale the image down:

---

```
img *= 1./255;
cvtColor(img, img, CV_BGR2Luv);
```

---

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:

$$\text{RGB[A]} \text{ to Gray: } Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

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:

---

```
cvtColor(src, bwsrc, CV_RGB2GRAY);
```

---

Some more advanced channel reordering can also be done with [mixChannels](#).

- RGB ↔ 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 ↔ YCrCb JPEG (a.k.a. YCC) (**CV\_BGR2YCrCb**, **CV\_RGB2YCrCb**, **CV\_YCrCb2BGR**, **CV\_YCrCb2RGB**)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + \textit{delta}$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + \textit{delta}$$

$$R \leftarrow Y + 1.403 \cdot (Cr - \textit{delta})$$

$$G \leftarrow Y - 0.344 \cdot (Cr - \textit{delta}) - 0.714 \cdot (Cb - \textit{delta})$$

$$B \leftarrow Y + 1.773 \cdot (Cb - \textit{delta})$$

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  $\leftrightarrow$  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) \*

$$V \leftarrow -65535V, S \leftarrow -65535S, H \leftarrow -H$$

- 32-bit images \*

H, S, V are left as is

- RGB  $\leftrightarrow$  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  $0 \leq V \leq 1, 0 \leq S \leq 1, 0 \leq H \leq 360$ .

The values are then converted to the destination data type:

8-bit images \*

$$V \leftarrow 255 \cdot V, S \leftarrow 255 \cdot S, H \leftarrow H/2 \text{ (to fit to 0 to 255)}$$

- 16-bit images (currently not supported) \*

$$V \leftarrow -65535 \cdot V, S \leftarrow -65535 \cdot S, H \leftarrow -H$$

- 32-bit images \*

H, S, V are left as is

- RGB  $\leftrightarrow$  CIE L\*a\*b\* (**CV\_BGR2Lab**, **CV\_RGB2Lab**, **CV\_Lab2BGR**, **CV\_Lab2RGB**) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

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

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

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

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

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

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

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t \leq 0.008856 \end{cases}$$

and

$$\textit{delta} = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output  $0 \leq L \leq 100, -127 \leq a \leq 127, -127 \leq b \leq 127$

The values are then converted to the destination data type:

8-bit images \*

$$L \leftarrow L * 255/100, a \leftarrow a + 128, b \leftarrow b + 128$$

- 16-bit images \*  
currently not supported
- 32-bit images \*  
L, a, b are left as is
- RGB  $\leftrightarrow$  CIE L\*u\*v\* (**CV\_BGR2Luv**, **CV\_RGB2Luv**, **CV\_Luv2BGR**, **CV\_Luv2RGB**) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

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

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

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

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

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

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

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 Charles Poynton site <http://www.poynton.com/ColorFAQ.html>

- Bayer  $\rightarrow$  RGB (**CV\_BayerBG2BGR**, **CV\_BayerGB2BGR**,  
**CV\_BayerRG2BGR**, **CV\_BayerGR2BGR**,  
**CV\_BayerBG2RGB**, **CV\_BayerGB2RGB**,  
**CV\_BayerRG2RGB**, **CV\_BayerGR2RGB**) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

```

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

```

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.

## distanceTransform

---

void **distanceTransform**(const Mat& *src*, Mat& *dst*, int *distanceType*, int *maskSize*)

void **distanceTransform**(const Mat& *src*, Mat& *dst*, Mat& *labels*, int *distanceType*, int *maskSize*)

Calculates the distance to the closest zero pixel for each pixel of the source image.

- Parameters:**
- *src* – 8-bit, single-channel (binary) source image
  - *dst* – Output image with calculated distances; will be 32-bit floating-point, single-channel image of the same size as *src*
  - *distanceType* – Type of distance; can be **CV\_DIST\_L1**, **CV\_DIST\_L2** or **CV\_DIST\_C**
  - *maskSize* – Size of the distance transform mask; can be 3, 5 or **CV\_DIST\_MASK\_PRECISE** (the latter option is only supported by the first of the functions). In the case of

**CV\_DIST\_L1** or **CV\_DIST\_C** distance type the parameter is forced to 3, because a  $3 \times 3$  mask gives the same result as a  $5 \times 5$  or any larger aperture.

- *labels* – The optional output 2d array of labels - the discrete Voronoi diagram; will have type **CV\_32SC1** and the same size as *src*. See the discussion

The functions `distanceTransform` calculate the approximate or precise distance from every binary image pixel to the nearest zero pixel. (for zero image pixels the distance will obviously be zero).

When `maskSize == CV_DIST_MASK_PRECISE` and `distanceType == CV_DIST_L2`, the function runs the algorithm described in .

In other cases the algorithm is used, that is, for pixel the function finds the shortest path to the nearest zero pixel consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a  $5 \times 5$  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  $5 \times 5$  mask gives more accurate results). For *a*, *b* and *c* OpenCV uses the values suggested in the original paper:

<b>CV_DIST_C</b>	( $3 \times 3$ )	<i>a</i> = 1, <i>b</i> = 1
<b>CV_DIST_L1</b>	( $3 \times 3$ )	<i>a</i> = 1, <i>b</i> = 2
<b>CV_DIST_L2</b>	( $3 \times 3$ )	<i>a</i> =0.955, <i>b</i> =1.3693
<b>CV_DIST_L2</b>	( $5 \times 5$ )	<i>a</i> =1, <i>b</i> =1.4, <i>c</i> =2.1969

Typically, for a fast, coarse distance estimation **CV\_DIST\_L2**, a  $3 \times 3$  mask is used, and for a more accurate distance estimation **CV\_DIST\_L2**, a  $5 \times 5$  mask or the precise algorithm is used. Note that both the precise and the approximate algorithms are linear on the number of pixels.

The second variant of the function does not only compute the minimum distance for each pixel  $(x, y)$ , but it also identifies the nearest the nearest connected component consisting of zero pixels. Index of the component is stored in `labels(x, y)`. The connected components of zero pixels are also found and marked by the function.

In this mode the complexity is still linear. That is, the function provides a very fast way to compute Voronoi diagram for the binary image. Currently, this second variant can only use the approximate distance transform algorithm.

## floodFill

---

int **floodFill**(Mat& *image*, Point *seed*, Scalar *newVal*, Rect\* *rect*=0, Scalar *loDiff*=Scalar(), Scalar *upDiff*=Scalar(), int *flags*=4)

int **floodFill**(Mat& *image*, Mat& *mask*, Point *seed*, Scalar *newVal*, Rect\* *rect*=0, Scalar *loDiff*=Scalar(), Scalar *upDiff*=Scalar(), int *flags*=4)

Fills a connected component with the given color.

- Parameters:**
- *image* – Input/output 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the `FLOODFILL_MASK_ONLY` flag is set (in the second variant of the function; see below)
  - *mask* – (For the second function only) Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller. The function uses and updates the mask, so the user takes responsibility of initializing the `mask` content. Flood-filling 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  $(x, y)$  in `image` will correspond to the pixel  $(x + 1, y + 1)$  in the `mask`
  - *seed* – The starting point
  - *newVal* – New value of the repainted domain pixels
  - *loDiff* – 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
  - *upDiff* – 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
  - *rect* – The optional output parameter that the function sets to the minimum bounding rectangle of the repainted domain
  - *flags* –  
The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:
    - **FLOODFILL\_FIXED\_RANGE** - if set, the difference between the current pixel and seed pixel is considered,

otherwise the difference between neighbor pixels is considered (i.e. the range is floating)

- **FLOODFILL\_MASK\_ONLY** - (for the second variant only) if set, the function does not change the image (`newVal` is ignored), but fills the mask

The functions `floodFill` fill a connected component starting from the seed point with the specified color. The connectivity is determined by the color/brightness closeness of the neighbor pixels. The pixel at  $(x, y)$  is considered to belong to the repainted domain if:

grayscale image, floating range \*

$$\text{src}(x', y') - \text{loDiff} \leq \text{src}(x, y) \leq \text{src}(x', y') + \text{upDiff}$$

- grayscale image, fixed range \*

$$\text{src}(\text{seed}.x, \text{seed}.y) - \text{loDiff} \leq \text{src}(x, y) \leq \text{src}(\text{seed}.x, \text{seed}.y) + \text{upDiff}$$

- color image, floating range \*

$$\text{src}(x', y')_r - \text{loDiff}_r \leq \text{src}(x, y)_r \leq \text{src}(x', y')_r + \text{upDiff}_r$$

$$\text{src}(x', y')_g - \text{loDiff}_g \leq \text{src}(x, y)_g \leq \text{src}(x', y')_g + \text{upDiff}_g$$

$$\text{src}(x', y')_b - \text{loDiff}_b \leq \text{src}(x, y)_b \leq \text{src}(x', y')_b + \text{upDiff}_b$$

- color image, fixed range \*

$$\text{src}(\text{seed}.x, \text{seed}.y)_r - \text{loDiff}_r \leq \text{src}(x, y)_r \leq \text{src}(\text{seed}.x, \text{seed}.y)_r$$

$$\text{src}(\text{seed}.x, \text{seed}.y)_g - \text{loDiff}_g \leq \text{src}(x, y)_g \leq \text{src}(\text{seed}.x, \text{seed}.y)_g$$

$$\text{src}(\text{seed}.x, \text{seed}.y)_b - \text{loDiff}_b \leq \text{src}(x, y)_b \leq \text{src}(\text{seed}.x, \text{seed}.y)_b$$

where  $\text{src}(x', y')$  is the value of one of pixel neighbors that is already known to belong to the component. 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.

By using these functions you can either mark a connected component with the specified color in-place, or build a mask and then extract the contour or copy the region to another image etc. Various modes of the function are demonstrated in `floodfill.c` sample.

See also: [findContours](#)

## inpaint

---

void **inpaint**(const Mat& *src*, const Mat& *inpaintMask*, Mat& *dst*, double *inpaintRadius*, int *flags*)

Inpaints the selected region in the image.

- Parameters:**
- *src* – The input 8-bit 1-channel or 3-channel image.
  - *inpaintMask* – The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.
  - *dst* – The output image; will have the same size and the same type as `src`
  - *inpaintRadius* – The radius of a circular neighborhood of each point inpainted that is considered by the algorithm.
  - *flags* –  
The inpainting method, one of the following:
    - **INPAINT\_NS** - Navier-Stokes based method.
    - **INPAINT\_TELEA** - The method by Alexandru Telea

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. See <http://en.wikipedia.org/wiki/Inpainting> for more details.

## integral

---

void **integral**(const Mat& *image*, Mat& *sum*, int *sdepth*=-1)

void **integral**(const Mat& *image*, Mat& *sum*, Mat& *sqsum*, int *sdepth*=-1)

void **integral**(const Mat& *image*, Mat& *sum*, Mat& *sqsum*, Mat& *tilted*, int *sdepth*=-1)

Calculates the integral of an image.

- Parameters:**
- *image* – The source image,  $W \times H$ , 8-bit or floating-point (32f or 64f)
  - *sum* – The integral image,  $(W + 1) \times (H + 1)$ , 32-bit integer or floating-point (32f or 64f)
  - *sqsum* – The integral image for squared pixel values,  $(W + 1) \times (H + 1)$ , double precision floating-point (64f)
  - *tilted* – The integral for the image rotated by 45 degrees,

$(W + 1) \times (H + 1)$ , the same data type as `sum`

- `sdepth` – The desired depth of the integral and the tilted integral images, `CV_32S`, `CV_32F` or `CV_64F`

The functions `integral` calculate one or more integral images for the source image as following:

$$\begin{aligned} \text{sum}(X, Y) &= \sum_{x < X, y < Y} \text{image}(x, y) \\ \text{sqsum}(X, Y) &= \sum_{x < X, y < Y} \text{image}(x, y)^2 \\ \text{tilted}(X, Y) &= \sum_{y < Y, \text{abs}(x - X) < y} \text{image}(x, y) \end{aligned}$$

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:

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

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

## threshold

---

double **threshold**(const Mat& *src*, Mat& *dst*, double *thresh*, double *maxVal*, int *thresholdType*)

Applies a fixed-level threshold to each array element

- Parameters:**
- *src* – Source array (single-channel, 8-bit or 32-bit floating point)
  - *dst* – Destination array; will have the same size and the same type as `src`
  - *thresh* – Threshold value
  - *maxVal* – Maximum value to use with `THRESH_BINARY` and `THRESH_BINARY_INV` thresholding types
  - *thresholdType* – Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image ([compare](#) 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:

- **THRESH\_BINARY** -

$$\text{dst}(x, y) = \begin{cases} \text{maxVal} & \text{if } \text{src}(x, y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- **THRESH\_BINARY\_INV** -

$$\text{dst}(x, y) = \begin{cases} 0 & \text{if } \text{src}(x, y) > \text{thresh} \\ \text{maxVal} & \text{otherwise} \end{cases}$$

- **THRESH\_TRUNC** -

$$\text{dst}(x, y) = \begin{cases} \text{threshold} & \text{if } \text{src}(x, y) > \text{thresh} \\ \text{src}(x, y) & \text{otherwise} \end{cases}$$

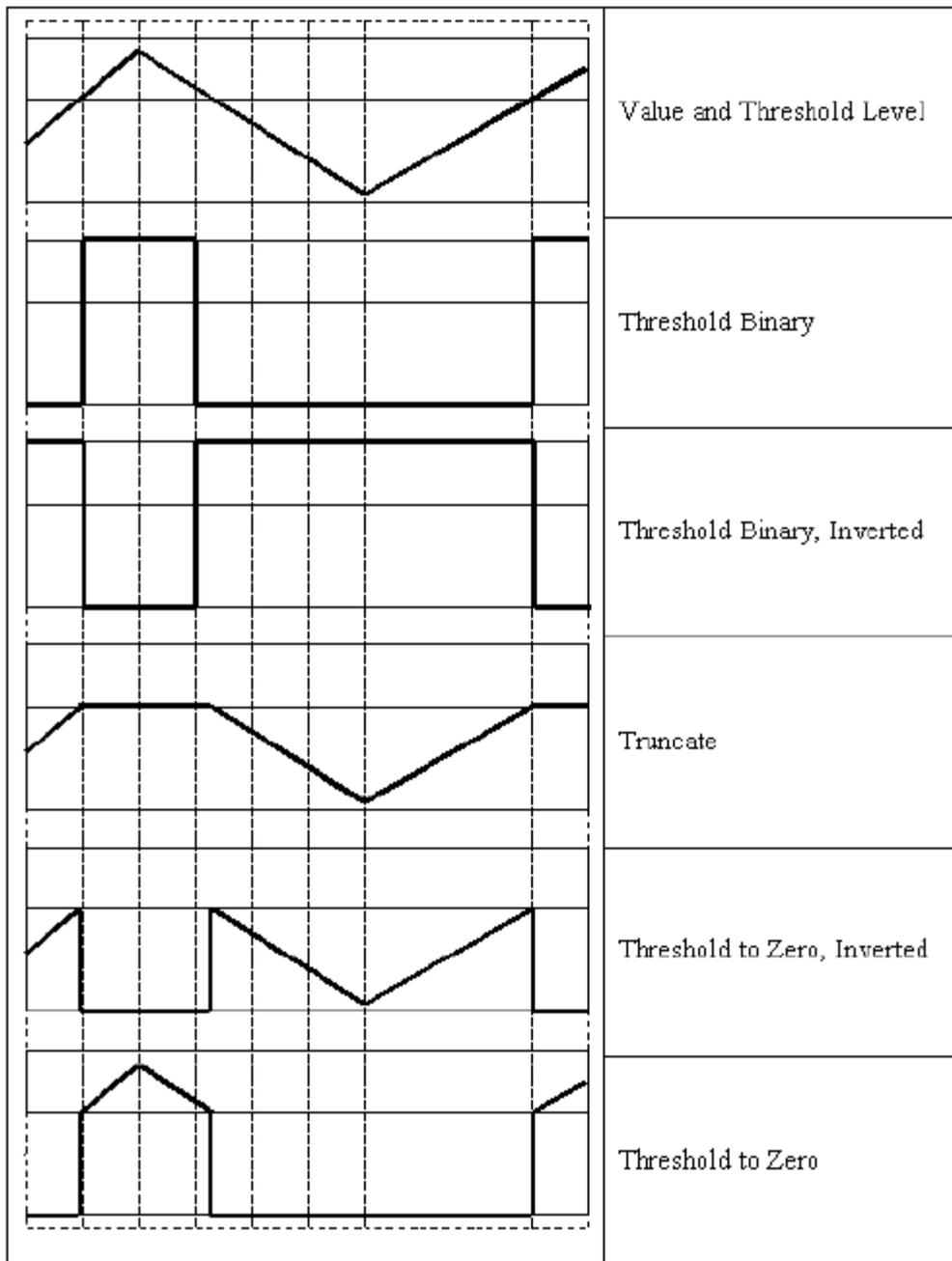
- **THRESH\_TOZERO** -

$$\text{dst}(x, y) = \begin{cases} \text{src}(x, y) & \text{if } \text{src}(x, y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- **THRESH\_TOZERO\_INV** -

$$\text{dst}(x, y) = \begin{cases} 0 & \text{if } \text{src}(x, y) > \text{thresh} \\ \text{src}(x, y) & \text{otherwise} \end{cases}$$

Also, the special value `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.



See also: [adaptiveThreshold](#), [findContours](#), [compare](#), [min](#), [max](#)

## watershed

void **watershed**(const Mat& *image*, Mat& *markers*)

Does marker-based image segmentation using watershed algorithm

- Parameters:**
- *image* – The input 8-bit 3-channel image.
  - *markers* – The input/output 32-bit single-channel image (map) of markers. It should have the same size as *image*

The function implements one of the variants of watershed, non-parametric

marker-based segmentation algorithm, described in . Before passing the image to the function, user has to outline roughly the desired regions in the image `markers` with positive ( $> 0$ ) indices, i.e. every region is represented as one or more connected components with the pixel values 1, 2, 3 etc (such markers can be retrieved from a binary mask using `findContours` and `drawContours`, see `watershed.cpp` demo). The markers will be “seeds” of the future image regions. All the other pixels in `markers`, which relation to the outlined regions is not known and should be defined by the algorithm, should be set to 0's. On the output of the function, each pixel in `markers` is set to one of values of the “seed” components, or to -1 at boundaries between the regions.

Note, that it is not necessary that every two neighbor connected components are separated by a watershed boundary (-1's pixels), for example, in case when such tangent components exist in the initial marker image. Visual demonstration and usage example of the function can be found in OpenCV samples directory; see `watershed.cpp` demo.

See also: [findContours](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Histograms

## calcHist

void **calcHist**(const Mat\* *arrays*, int *narrays*, const int\* *channels*, const Mat& *mask*, MatND& *hist*, int *dims*, const int\* *histSize*, const float\*\* *ranges*, bool *uniform=true*, bool *accumulate=false*)

void **calcHist**(const Mat\* *arrays*, int *narrays*, const int\* *channels*, const Mat& *mask*, SparseMat& *hist*, int *dims*, const int\* *histSize*, const float\*\* *ranges*, bool *uniform=true*, bool *accumulate=false*)

Calculates histogram of a set of arrays

- Parameters:**
- *arrays* – Source arrays. They all should have the same depth, **CV\_32F**, and the same size. Each of them can have an arbitrary of channels
  - *narrays* – The number of source arrays
  - *channels* – The list of *dims* channels that are used to compute histogram. The first array channels are numerated from 0 to `arrays[0].channels()-1`, the second array channels are counted `arrays[0].channels()` to `arrays[0].channels()+arrays[1].channels()-1` etc.
  - *mask* – The optional mask. If the matrix is not empty, it must be 8-bit of the same size as `arrays[i]`. The non-zero mask elements are the array elements that are counted in the histogram
  - *hist* – The output histogram, a dense or sparse *dims*-dimensional
  - *dims* – The histogram dimensionality; must be positive and not greater than `CV_MAX_DIMS` (=32 in the current OpenCV version)
  - *histSize* – The array of histogram sizes in each dimension
  - *ranges* – The array of *dims* arrays of the histogram bin boundaries in each dimension. When the histogram is uniform (`uniform=true`), then for each dimension it's enough to specify the lower (inclusive) boundary  $L_0$  of the first histogram bin and the upper (exclusive) boundary  $U_{\text{histSize}[i]-1}$  for the last histogram bin `histSize[i]-1`. That is, in the case of uniform histogram each of `ranges[i]` is an array of 2 elements. When the histogram is not uniform (`uniform=false`), then each of `ranges[i]` contains `histSize[i]+1` elements:  $L_0, U_0 = L_1, U_1 = L_2, \dots, U_{\text{histSize}[i]-2} = L_{\text{histSize}[i]-1}, U_{\text{histSize}[i]-1}$ . The array elements, which are not between  $L_0$  and  $U_{\text{histSize}[i]-1}$ , are not counted in the histogram
  - *uniform* – Indicates whether the histogram is uniform or not, see above
  - *accumulate* – Accumulation flag. If it is set, the histogram is not cleared at the beginning (when it is allocated). This feature allows user to compute

single histogram from several sets of arrays, or to update the histogram

The functions `calcHist` calculate the histogram of one or more arrays. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input arrays. The sample below shows how to compute 2D Hue-Saturation histogram for a color image

---

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

using namespace cv;

int main( int argc, char** argv )
{
    Mat src;
    if( argc != 2 || !(src=imread(argv[1], 1)).data )
        return -1;

    Mat hsv;
    cvtColor(src, hsv, CV_BGR2HSV);

    // let's quantize the hue to 30 levels
    // and the saturation to 32 levels
    int hbins = 30, sbins = 32;
    int histSize[] = {hbins, sbins};
    // hue varies from 0 to 179, see cvtColor
    float hranges[] = { 0, 180 };
    // saturation varies from 0 (black-gray-white) to
    // 255 (pure spectrum color)
    float sranges[] = { 0, 256 };
    float* ranges[] = { hranges, sranges };
    MatND hist;
    // we compute the histogram from the 0-th and 1-st channels
    int channels[] = {0, 1};

    calcHist( &hsv, 1, channels, Mat(), // do not use mask
             hist, 2, histSize, ranges,
             true, // the histogram is uniform
             false );
    double maxVal=0;
    minMaxLoc(hist, 0, &maxVal, 0, 0);

    int scale = 10;
    Mat histImg = Mat::zeros(sbins*scale, hbins*10, CV_8UC

    for( int h = 0; h < hbins; h++ )
        for( int s = 0; s < sbins; s++ )
        {
            float binVal = hist.at<float>(h, s);
```

```

        int intensity = cvRound(binVal*255/maxValue);
        cvRectangle( histImg, Point(h*scale, s*scale),
                    Point( (h+1)*scale - 1, (s+1)*scale
                    Scalar::all(intensity),
                    CV_FILLED );
    }

    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "H-S Histogram", 1 );
    imshow( "H-S Histogram", histImg );

    waitKey();
}

```

---

## calcBackProject

---

void **calcBackProject**(const Mat\* *arrays*, int *narrays*, const int\* *channels*, const MatND& *hist*, Mat& *backProject*, const float\*\* *ranges*, double *scale*=1, bool *uniform*=true)

void **calcBackProject**(const Mat\* *arrays*, int *narrays*, const int\* *channels*, const SparseMat& *hist*, Mat& *backProject*, const float\*\* *ranges*, double *scale*=1, bool *uniform*=true)

Calculates the back projection of a histogram.

- Parameters:**
- *arrays* – Source arrays. They all should have the same depth, **CV\_8U** or **CV\_32F**, and the same size. Each of them can have an arbitrary number of channels
  - *narrays* – The number of source arrays
  - *channels* – The list of channels that are used to compute the back projection. The number of channels must match the histogram dimensionality. The first array channels are numerated from 0 to `arrays[0].channels()-1`, the second array channels are counted from `arrays[0].channels()` to `arrays[0].channels() + arrays[1].channels()-1` etc.
  - *hist* – The input histogram, a dense or sparse
  - *backProject* – Destination back projection array; will be a single-channel array of the same size and the same depth as `arrays[0]`
  - *ranges* – The array of arrays of the histogram bin boundaries in each dimension. See [calcHist](#)

- *scale* – The optional scale factor for the output back projection
- *uniform* – Indicates whether the histogram is uniform or not, see above

The functions `calcBackProject` calculate the back project of the histogram. That is, similarly to `calcHist`, at each location  $(x, y)$  the function collects the values from the selected channels in the input images and finds the corresponding histogram bin. But instead of incrementing it, the function reads the bin value, scales it by *scale* and stores in `backProject(x, y)`. In terms of statistics, the function computes probability of each element value in respect with the empirical probability distribution represented by the histogram. Here is how, for example, you can find and track a bright-colored object in a scene:

- Before the tracking, show the object to the camera such that covers almost the whole frame. Calculate a hue histogram. The histogram will likely have a strong maximums, corresponding to the dominant colors in the object.
- During the tracking, calculate back projection of a hue plane of each input video frame using that pre-computed histogram. Threshold the back projection to suppress weak colors. It may also have sense to suppress pixels with non sufficient color saturation and too dark or too bright pixels.
- Find connected components in the resulting picture and choose, for example, the largest component.

That is the approximate algorithm of [CamShift](#) color object tracker.

See also: [calcHist](#)

## compareHist

---

double **compareHist**(const MatND& *H1*, const MatND& *H2*, int *method*)

double **compareHist**(const SparseMat& *H1*, const SparseMat& *H2*, int *method*)

Compares two histograms

- Parameters:**
- *H1* – The first compared histogram
  - *H2* – The second compared histogram of the same size as *H1*
  - *method* –  
The 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 functions `compareHist` compare two dense or two sparse histograms using the specified method:

Correlation (method=`CV_COMP_CORREL`) \*

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H}_1)(H_2(I) - \bar{H}_2)}{\sqrt{\sum_I (H_1(I) - \bar{H}_1)^2 \sum_I (H_2(I) - \bar{H}_2)^2}}$$

where

$$\bar{H}_k = \frac{1}{N} \sum_J H_k(J)$$

and  $N$  is the total 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`) \*

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H}_1 \bar{H}_2 N^2} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

The function returns  $d(H_1, H_2)$ .

While the function works well with 1-, 2-, 3-dimensional dense histograms, it may not be suitable for high-dimensional sparse histograms, where, because of aliasing and sampling problems the coordinates of non-zero histogram bins can slightly shift. To compare such histograms or more general sparse configurations of weighted points, consider using the *calcEMD* function.

## equalizeHist

---

void **equalizeHist**(const Mat& *src*, Mat& *dst*)

Equalizes the histogram of a grayscale image.

- Parameters:**
- *src* – The source 8-bit single channel image
  - *dst* – The destination image; will have the same size and the same type as *src*

The function equalizes the histogram of the input image using the following algorithm:

- calculate the histogram  $H$  for *src*.
- normalize the histogram so that the sum of histogram bins is 255.
- compute the integral of the histogram:

$$H'_i = \sum_{0 \leq j < i} H(j)$$

- transform the image using  $H'$  as a look-up table:  
 $\text{dst}(x, y) = H'(\text{src}(x, y))$

The algorithm normalizes the brightness and increases the contrast of the image.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Feature Detection

---

## Canny

---

void **Canny**(const Mat& *image*, Mat& *edges*, double *threshold1*, double *threshold2*, int *apertureSize*=3, bool *L2gradient*=false)

Finds edges in an image using Canny algorithm.

- Parameters:**
- *image* – Single-channel 8-bit input image
  - *edges* – The output edge map. It will have the same size and the same type as *image*
  - *threshold1* – The first threshold for the hysteresis procedure
  - *threshold2* – The second threshold for the hysteresis procedure
  - *apertureSize* – Aperture size for the [Sobel](#) operator
  - *L2gradient* – Indicates, whether the more accurate  $L_2$  norm  $= \sqrt{(dI/dx)^2 + (dI/dy)^2}$  should be used to compute the image gradient magnitude (*L2gradient*=true), or a faster default  $L_1$  norm  $= |dI/dx| + |dI/dy|$  is enough (*L2gradient*=false)

The function finds edges in the input image *image* and marks them in the output map *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, see [http://en.wikipedia.org/wiki/Canny\\_edge\\_detector](http://en.wikipedia.org/wiki/Canny_edge_detector)

## cornerEigenValsAndVecs

---

void **cornerEigenValsAndVecs**(const Mat& *src*, Mat& *dst*, int *blockSize*, int *apertureSize*, int *borderType*=BORDER\_DEFAULT)

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

- Parameters:**
- *src* – Input single-channel 8-bit or floating-point image
  - *dst* – Image to store the results. It will have the same size as *src* and the type **CV\_32FC(6)**
  - *blockSize* – Neighborhood size (see discussion)
  - *apertureSize* – Aperture parameter for the [Sobel](#) operator
  - *boderType* – Pixel extrapolation method; see [borderInterpolate](#)

For every pixel *P*, the function `cornerEigenValsAndVecs` considers a *blockSize* × *blockSize* neighborhood  $S(p)$ . It calculates the covariation matrix of derivatives over the neighborhood as:

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

Where the derivatives are computed using [Sobel](#) operator.

After that it finds eigenvectors and eigenvalues of  $M$  and stores them into destination image in the form  $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$  where

$\lambda_1, \lambda_2$  \*

are the eigenvalues of  $M$ ; not sorted

- $x_1, y_1$  \*  
are the eigenvectors corresponding to  $\lambda_1$
- $x_2, y_2$  \*  
are the eigenvectors corresponding to  $\lambda_2$

The output of the function can be used for robust edge or corner detection.

See also: [cornerMinEigenVal](#), [cornerHarris](#), [preCornerDetect](#)

## cornerHarris

---

void **cornerHarris**(const Mat& *src*, Mat& *dst*, int *blockSize*, int *apertureSize*, double *k*, int *borderType*=BORDER\_DEFAULT)

Harris edge detector.

- Parameters:**
- *src* – Input single-channel 8-bit or floating-point image
  - *dst* – Image to store the Harris detector responses; will have type **CV\_32FC1** and the same size as *src*
  - *blockSize* – Neighborhood size (see the discussion of [cornerEigenValsAndVecs](#))
  - *apertureSize* – Aperture parameter for the [Sobel](#) operator
  - *k* – Harris detector free parameter. See the formula below
  - *boderType* – Pixel extrapolation method; see [borderInterpolate](#)

The function runs the Harris edge detector on the image. Similarly to [cornerMinEigenVal](#) and [cornerEigenValsAndVecs](#), for each pixel  $(x, y)$  it calculates a  $2 \times 2$  gradient covariation matrix  $M^{(x,y)}$  over a  $\text{blockSize} \times \text{blockSize}$  neighborhood. Then, it computes the following characteristic:

$$\text{dst}(x, y) = \det M^{(x,y)} - k \cdot (\text{tr} M^{(x,y)})^2$$

Corners in the image can be found as the local maxima of this response map.

## cornerMinEigenVal

---

void **cornerMinEigenVal**(const Mat& *src*, Mat& *dst*, int *blockSize*, int *apertureSize*=3, int *borderType*=*BORDER\_DEFAULT*)

Calculates the minimal eigenvalue of gradient matrices for corner detection.

- Parameters:**
- *src* – Input single-channel 8-bit or floating-point image
  - *dst* – Image to store the minimal eigenvalues; will have type **CV\_32FC1** and the same size as *src*
  - *blockSize* – Neighborhood size (see the discussion of [cornerEigenValsAndVecs](#))
  - *apertureSize* – Aperture parameter for the [Sobel](#) operator
  - *boderType* – Pixel extrapolation method; see [borderInterpolate](#)

The function is similar to [cornerEigenValsAndVecs](#) but it calculates and stores only the minimal eigenvalue of the covariation matrix of derivatives, i.e.  $\min(\lambda_1, \lambda_2)$  in terms of the formulae in [cornerEigenValsAndVecs](#) description.

## cornerSubPix

---

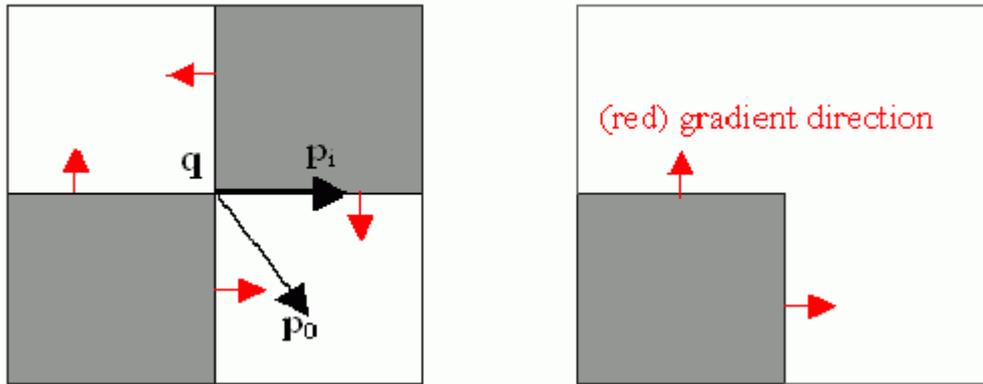
void **cornerSubPix**(const Mat& *image*, vector<Point2f>& *corners*, Size *winSize*, Size *zeroZone*, TermCriteria *criteria*)

Refines the corner locations.

- Parameters:**
- *image* – Input image
  - *corners* – Initial coordinates of the input corners; refined coordinates on output
  - *winSize* – Half of the side length of the search window. For example, if `winSize=Size(5,5)`, then a  $5 * 2 + 1 \times 5 * 2 + 1 = 11 \times 11$  search window would be used
  - *zeroZone* – Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size
  - *criteria* – Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The *criteria* may

specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below.



Sub-pixel accurate corner locator is based on the observation that every vector from the center  $q$  to a point  $p$  located within a neighborhood of  $q$  is orthogonal to the image gradient at  $p$  subject to image and measurement noise. Consider the expression:

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

where  $DI_{p_i}$  is the image gradient at the one of the points  $p_i$  in a neighborhood of  $q$ . The value of  $q$  is to be found such that  $\epsilon_i$  is minimized. A system of equations may be set up with  $\epsilon_i$  set to zero:

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

where the gradients are summed within a neighborhood (“search window”) of  $q$ . Calling the first gradient term  $G$  and the second gradient term  $b$  gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center  $q$  and then iterates until the center keeps within a set threshold.

## goodFeaturesToTrack

```
void goodFeaturesToTrack(const Mat& image, vector<Point2f>&
corners, int maxCorners, double qualityLevel, double minDistance, const Mat&
mask=Mat(), int blockSize=3, bool useHarrisDetector=false, double k=0.04)
```

Determines strong corners on an image.

**Parameters:**

- $image$  – The input 8-bit or floating-point 32-bit, single-channel image

- *corners* – The output vector of detected corners
- *maxCorners* – The maximum number of corners to return. If there are more corners than that will be found, the strongest of them will be returned
- *qualityLevel* – Characterizes the minimal accepted quality of image corners; the value of the parameter is multiplied by the by the best corner quality measure (which is the min eigenvalue, see [cornerMinEigenVal](#), or the Harris function response, see [cornerHarris](#)). The corners, which quality measure is less than the product, will be rejected. For example, if the best corner has the quality measure = 1500, and the `qualityLevel=0.01`, then all the corners which quality measure is less than 15 will be rejected.
- *minDistance* – The minimum possible Euclidean distance between the returned corners
- *mask* – The optional region of interest. If the image is not empty (then it needs to have the type `CV_8UC1` and the same size as `image`), it will specify the region in which the corners are detected
- *blockSize* – Size of the averaging block for computing derivative covariation matrix over each pixel neighborhood, see [cornerEigenValsAndVecs](#)
- *useHarrisDetector* – Indicates, whether to use [`cornerHarris`] `bgroup({Harris})` operator or [cornerMinEigenVal](#)
- *k* – Free parameter of Harris detector

The function finds the most prominent corners in the image or in the specified image region, as described in :

- the function first calculates the corner quality measure at every source image pixel using the [cornerMinEigenVal](#) or [cornerHarris](#)
- then it performs non-maxima suppression (the local maxima in  $3 \times 3$  neighborhood are retained).
- the next step rejects the corners with the minimal eigenvalue less than  $qualityLevel \cdot \max_{x,y} qualityMeasureMap(x, y)$ .
- the remaining corners are then sorted by the quality measure in the descending order.
- finally, the function throws away each corner  $pt_j$  if there is a stronger corner  $pt_i$  ( $i < j$ ) such that the distance between them is less than `minDistance`

The function can be used to initialize a point-based tracker of an object.

See also: [cornerMinEigenVal](#), [cornerHarris](#), [calcOpticalFlowPyrLK](#), [estimateRigidMotion](#), [PlanarObjectDetector](#), [OneWayDescriptor](#)

## HoughCircles

---

void **HoughCircles**(Mat& *image*, vector<Vec3f>& *circles*, int *method*, double *dp*, double *minDist*, double *param1*=100, double *param2*=100, int *minRadius*=0, int *maxRadius*=0)

Finds circles in a grayscale image using a Hough transform.

- Parameters:**
- *image* – The 8-bit, single-channel, grayscale input image
  - *circles* – The output vector of found circles. Each vector is encoded as 3-element floating-point vector  $(x, y, radius)$
  - *method* – Currently, the only implemented method is **CV\_HOUGH\_GRADIENT**, which is basically 21HT, described in .
  - *dp* – The inverse ratio of the accumulator resolution to the image resolution. For example, if  $dp=1$ , the accumulator will have the same resolution as the input image, if  $dp=2$  - accumulator will have half as big width and height, etc
  - *minDist* – Minimum distance between the centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed
  - *param1* – The first method-specific parameter. in the case of **CV\_HOUGH\_GRADIENT** it is the higher threshold of the two passed to *Canny* edge detector (the lower one will be twice smaller)
  - *param2* – The second method-specific parameter. in the case of **CV\_HOUGH\_GRADIENT** it is the accumulator threshold at the center detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first
  - *minRadius* – Minimum circle radius
  - *maxRadius* – Maximum circle radius

The function finds circles in a grayscale image using some modification of Hough transform. Here is a short usage example:

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

using namespace cv;

int main(int argc, char** argv)
{
    Mat img, gray;
    if( argc != 2 && !(img=imread(argv[1], 1)).data)
```

```

        return -1;
    cvtColor(img, gray, CV_BGR2GRAY);
    // smooth it, otherwise a lot of false circles may be detected
    GaussianBlur( gray, gray, 9, 9, 2, 2 );
    vector<Vec3f> circles;
    houghCircles(gray, circles, CV_HOUGH_GRADIENT,
                2, gray->rows/4, 200, 100 );
    for( size_t i = 0; i < circles.size(); i++ )
    {
        Point center(cvRound(circles[i][0]), cvRound(circles
        int radius = cvRound(circles[i][2]));
        // draw the circle center
        circle( img, center, 3, Scalar(0,255,0), -1, 8, 0
        // draw the circle outline
        circle( img, center, radius, Scalar(0,0,255), 3,
    }
    namedWindow( "circles", 1 );
    imshow( "circles", img );
    return 0;
}

```

Note that usually the function detects the circles' centers well, however it may fail to find the correct radii. You can assist the function by specifying the radius range (`minRadius` and `maxRadius`) if you know it, or you may ignore the returned radius, use only the center and find the correct radius using some additional procedure.

See also: [fitEllipse](#), [minEnclosingCircle](#)

## HoughLines

void **HoughLines**(Mat& *image*, vector<Vec2f>& *lines*, double *rho*, double *theta*, int *threshold*, double *srn*=0, double *stn*=0)

Finds lines in a binary image using standard Hough transform.

- Parameters:**
- *image* – The 8-bit, single-channel, binary source image. The image may be modified by the function
  - *lines* – The output vector of lines. Each line is represented by a two-element vector  $(\rho, \theta)$ .  $\rho$  is the distance from the coordinate origin  $(0, 0)$  (top-left corner of the image) and  $\theta$  is the line rotation angle in radians ( $0 \sim$  vertical line,  $\pi/2 \sim$  horizontal line)
  - *rho* – Distance resolution of the accumulator in pixels
  - *theta* – Angle resolution of the accumulator in radians
  - *threshold* – The accumulator threshold parameter. Only those lines are returned that get enough votes ( $>$  `threshold`)

- *srn* – For the multi-scale Hough transform it is the divisor for the distance resolution *rho*. The coarse accumulator distance resolution will be *rho* and the accurate accumulator resolution will be *rho/srn*. If both *srn*=0 and *stn*=0 then the classical Hough transform is used, otherwise both these parameters should be positive.
- *stn* – For the multi-scale Hough transform it is the divisor for the distance resolution *theta*

The function implements standard or standard multi-scale Hough transform algorithm for line detection. See [HoughLinesP](#) for the code example.

## HoughLinesP

---

void **HoughLinesP**(Mat& *image*, vector<Vec4i>& *lines*, double *rho*, double *theta*, int *threshold*, double *minLineLength*=0, double *maxLineGap*=0)

Finds lines segments in a binary image using probabilistic Hough transform.

- Parameters:**
- *image* – The 8-bit, single-channel, binary source image. The image may be modified by the function
  - *lines* – The output vector of lines. Each line is represented by a 4-element vector  $(x_1, y_1, x_2, y_2)$ , where  $(x_1, y_1)$  and  $(x_2, y_2)$  are the ending points of each line segment detected.
  - *rho* – Distance resolution of the accumulator in pixels
  - *theta* – Angle resolution of the accumulator in radians
  - *threshold* – The accumulator threshold parameter. Only those lines are returned that get enough votes ( $> \text{threshold}$ )
  - *minLineLength* – The minimum line length. Line segments shorter than that will be rejected
  - *maxLineGap* – The maximum allowed gap between points on the same line to link them.

The function implements probabilistic Hough transform algorithm for line detection, described in . Below is line detection example:

```

/* This is a standalone program. Pass an image name as a i
of the program. Switch between standard and probabilistic
by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

using namespace cv;

int main(int argc, char** argv)
{

```

```

Mat src, dst, color_dst;
if( argc != 2 || !(src=imread(argv[1], 0)).data)
    return -1;

Canny( src, dst, 50, 200, 3 );
cvtColor( dst, color_dst, CV_GRAY2BGR );

#if 0
vector<Vec2f> lines;
HoughLines( dst, lines, 1, CV_PI/180, 100 );

for( size_t i = 0; i < lines.size(); i++ )
{
    float rho = lines[i][0];
    float theta = lines[i][1];
    double a = cos(theta), b = sin(theta);
    double x0 = a*rho, y0 = b*rho;
    Point pt1(cvRound(x0 + 1000*(-b)),
              cvRound(y0 + 1000*(a)));
    Point pt2(cvRound(x0 - 1000*(-b)),
              cvRound(y0 - 1000*(a)));
    line( color_dst, pt1, pt2, Scalar(0,0,255), 3, 8 );
}
#else
vector<Vec4i> lines;
HoughLinesP( dst, lines, 1, CV_PI/180, 80, 30, 10 );
for( size_t i = 0; i < lines.size(); i++ )
{
    line( color_dst, Point(lines[i][0], lines[i][1]),
          Point(lines[i][2], lines[i][3]), Scalar(0,0,255)
        );
}
#endif
namedWindow( "Source", 1 );
imshow( "Source", src );

namedWindow( "Detected Lines", 1 );
imshow( "Detected Lines", color_dst );

waitKey(0);
return 0;
}

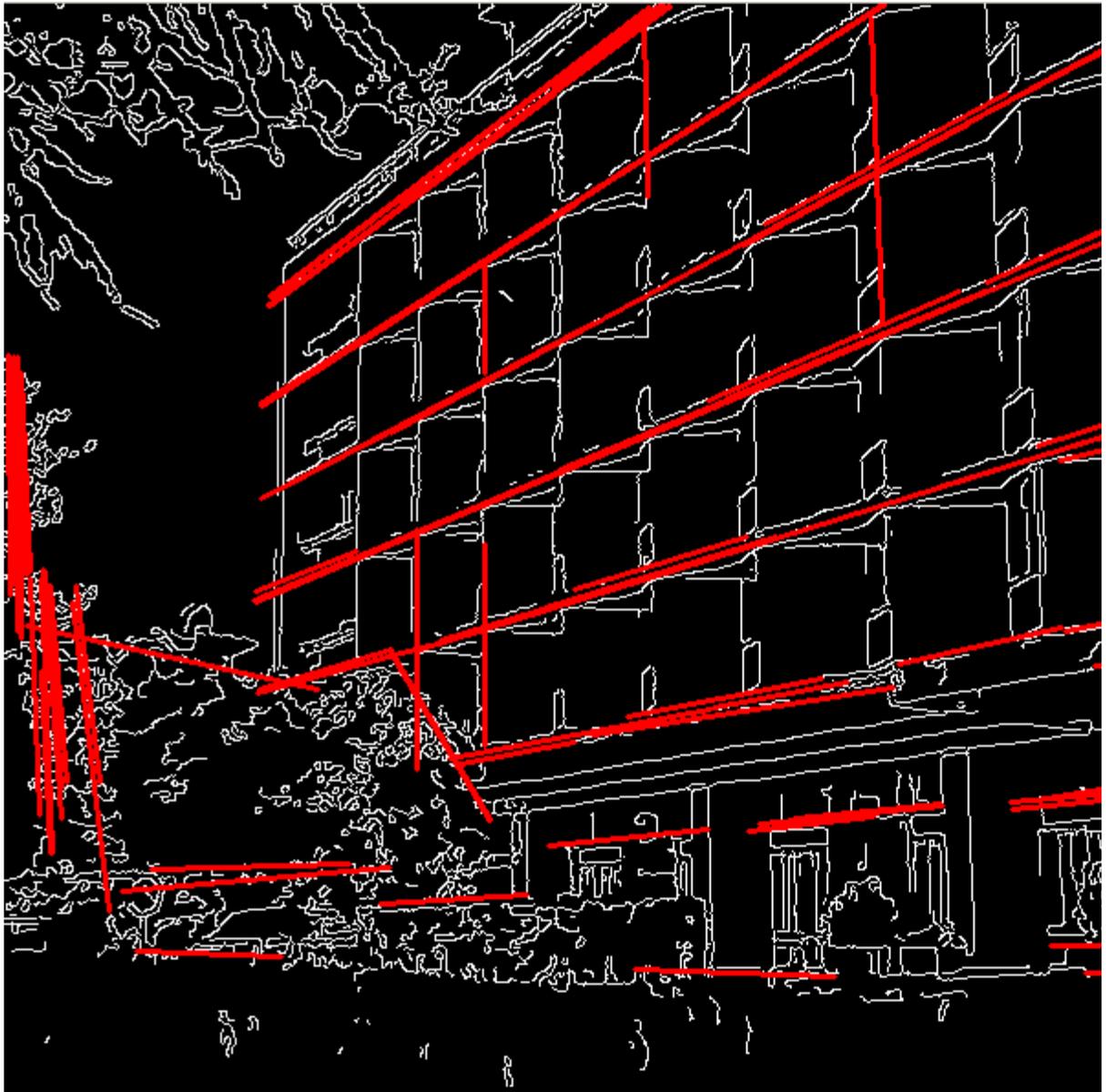
```

---

This is the sample picture the function parameters have been tuned for:



And this is the output of the above program in the case of probabilistic Hough transform



## perCornerDetect

```
void preCornerDetect(const Mat& src, Mat& dst, int apertureSize, int borderType=BORDER_DEFAULT)
```

Calculates the feature map for corner detection

- Parameters:**
- *src* – The source single-channel 8-bit or floating-point image
  - *dst* – The output image; will have type **CV\_32F** and the same size as *src*
  - *apertureSize* – Aperture size of [Sobel](#)
  - *borderType* – The pixel extrapolation method; see [borderInterpolate](#)

The function calculates the complex spatial derivative-based function of the source image

$$\text{dst} = (D_x \text{src})^2 \cdot D_{yy} \text{src} + (D_y \text{src})^2 \cdot D_{xx} \text{src} - 2D_x \text{src} \cdot D_y \text{src} \cdot D_{xy} \text{src}$$

where  $D_x$ ,  $D_y$  are the first image derivatives,  $D_{xx}$ ,  $D_{yy}$  are the second image derivatives and  $D_{xy}$  is the mixed derivative.

The corners can be found as local maximums of the functions, as shown below:

---

```
Mat corners, dilated_corners;
preCornerDetect(image, corners, 3);
// dilation with 3x3 rectangular structuring element
dilate(corners, dilated_corners, Mat(), 1);
Mat corner_mask = corners == dilated_corners;
```

---

## Keypoint

---

Data structure for salient point detectors

---

```
Keypoint
{
public:
    // default constructor
    Keypoint();
    // two complete constructors
    Keypoint(Point2f _pt, float _size, float _angle=-1,
             float _response=0, int _octave=0, int _class_id=-1);
    Keypoint(float x, float y, float _size, float _angle=-1,
             float _response=0, int _octave=0, int _class_id=-1);
    // coordinate of the point
    Point2f pt;
    // feature size
    float size;
    // feature orintation in degrees
    // (has negative value if the orientation
    // is not defined/not computed)
    float angle;
    // feature strength
    // (can be used to select only
    // the most prominent key points)
    float response;
    // scale-space octave in which the feature has been found
    // may correlate with the size
    int octave;
    // point (can be used by feature
    // classifiers or object detectors)
    int class_id;
};
```

```
// reading/writing a vector of keypoints to a file storage
void write(FileStorage& fs, const string& name, const vector<
void read(const FileNode& node, vector<KeyPoint>& keypoints);
```

---

## MSER

---

### Maximally-Stable Extremal Region Extractor

```
class MSER : public CvMSERParams
{
public:
    // default constructor
    MSER();
    // constructor that initializes all the algorithm paramet
    MSER( int _delta, int _min_area, int _max_area,
         float _max_variation, float _min_diversity,
         int _max_evolution, double _area_threshold,
         double _min_margin, int _edge_blur_size );
    // runs the extractor on the specified image; returns the
    // each encoded as a contour (vector<Point>, see findCont
    // the optional mask marks the area where MSERs are searc
    void operator() (Mat& image, vector<vector<Point> >& msers
};
```

---

The class encapsulates all the parameters of MSER (see [http://en.wikipedia.org/wiki/Maximally\\_stable\\_extremal\\_regions](http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions)) extraction algorithm.

## SURF

---

Class for extracting Speeded Up Robust Features from an image.

```
class SURF : public CvSURFParams
{
public:
    // default constructor
    SURF();
    // constructor that initializes all the algorithm paramet
    SURF(double _hessianThreshold, int _nOctaves=4,
         int _nOctaveLayers=2, bool _extended=false);
    // returns the number of elements in each descriptor (64
    int descriptorSize() const;
    // detects keypoints using fast multi-scale Hessian detec
    void operator() (const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints) const;
    // detects keypoints and computes the SURF descriptors fc
    void operator() (const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints,
```

```

        vector<float>& descriptors,
        bool useProvidedKeypoints=false) const;
};

```

---

The class SURF implements Speeded Up Robust Features descriptor . There is fast multi-scale Hessian keypoint detector that can be used to find the keypoints (which is the default option), but the descriptors can be also computed for the user-specified keypoints. The function can be used for object tracking and localization, image stitching etc. See the `find_obj.cpp` demo in OpenCV samples directory.

## StarDetector

---

Implements Star keypoint detector

---

```

class StarDetector : CvStarDetectorParams
{
public:
    // default constructor
    StarDetector();
    // the full constructor initialized all the algorithm par
    // maxSize - maximum size of the features. The following
    // values of the parameter are supported:
    // 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90,
    // responseThreshold - threshold for the approximated lap
    // used to eliminate weak features. The larger it is
    // the less features will be retrieved
    // lineThresholdProjected - another threshold for the lap
    // eliminate edges
    // lineThresholdBinarized - another threshold for the fea
    // size to eliminate edges.
    // The larger the 2 threshold, the more points you get.
    StarDetector(int maxSize, int responseThreshold,
                 int lineThresholdProjected,
                 int lineThresholdBinarized,
                 int suppressNonmaxSize);

    // finds keypoints in an image
    void operator()(const Mat& image, vector<KeyPoint>& keypo
};

```

---

The class implements a modified version of CenSurE keypoint detector described in

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Motion Analysis and Object Tracking

---

## accumulate

---

void **accumulate**(const Mat& *src*, Mat& *dst*, const Mat& *mask=Mat()*)

Adds image to the accumulator.

- Parameters:**
- *src* – The input image, 1- or 3-channel, 8-bit or 32-bit floating point
  - *dst* – The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point
  - *mask* – Optional operation mask

The function adds *src*, or some of its elements, to *dst*:

$$\text{dst}(x, y) \leftarrow \text{dst}(x, y) + \text{src}(x, y) \quad \text{if } \text{mask}(x, y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.

The functions `accumulate*` can be used, for example, to collect statistic of background of a scene, viewed by a still camera, for the further foreground-background segmentation.

See also: [accumulateSquare](#), [accumulateProduct](#), [accumulateWeighted](#)

## accumulateSquare

---

void **accumulateSquare**(const Mat& *src*, Mat& *dst*, const Mat& *mask=Mat()*)

Adds the square of the source image to the accumulator.

- Parameters:**
- *src* – The input image, 1- or 3-channel, 8-bit or 32-bit floating point
  - *dst* – The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point
  - *mask* – Optional operation mask

The function adds the input image *src* or its selected region, raised to power 2, to the accumulator *dst*:

$$\text{dst}(x, y) \leftarrow \text{dst}(x, y) + \text{src}(x, y)^2 \quad \text{if } \text{mask}(x, y) \neq 0$$

The function supports multi-channel images; each channel is processed

independently.

See also: [accumulateSquare](#), [accumulateProduct](#), [accumulateWeighted](#)

## accumulateProduct

---

void **accumulateProduct**(const Mat& *src1*, const Mat& *src2*, Mat& *dst*, const Mat& *mask=Mat()*)

Adds the per-element product of two input images to the accumulator.

- Parameters:**
- *src1* – The first input image, 1- or 3-channel, 8-bit or 32-bit floating point
  - *src2* – The second input image of the same type and the same size as *src1*
  - *dst* – Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point
  - *mask* – Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator *dst*:

$$\text{dst}(x, y) \leftarrow \text{dst}(x, y) + \text{src1}(x, y) \cdot \text{src2}(x, y) \quad \text{if } \text{mask}(x, y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.

See also: [accumulate](#), [accumulateSquare](#), [accumulateWeighted](#)

## accumulateWeighted

---

void **accumulateWeighted**(const Mat& *src*, Mat& *dst*, double *alpha*, const Mat& *mask=Mat()*)

Updates the running average.

- Parameters:**
- *src* – The input image, 1- or 3-channel, 8-bit or 32-bit floating point
  - *dst* – The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point
  - *alpha* – Weight of the input image
  - *mask* – Optional operation mask

The function calculates the weighted sum of the input image *src* and the accumulator *dst* so that *dst* becomes a running average of frame sequence:

$$\text{dst}(x, y) \leftarrow (1 - \alpha) \cdot \text{dst}(x, y) + \alpha \cdot \text{src}(x, y) \quad \text{if } \text{mask}(x, y) \neq 0$$

that is, `alpha` regulates the update speed (how fast the accumulator “forgets” about earlier images). The function supports multi-channel images; each channel is processed independently.

See also: [accumulate](#), [accumulateSquare](#), [accumulateProduct](#)

## calcOpticalFlowPyrLK

```
void calcOpticalFlowPyrLK(const Mat& prevImg, const Mat&
nextImg, const vector<Point2f>& prevPts, vector<Point2f>& nextPts, vector<uchar>&
status, vector<float>& err, Size winSize=Size(15, 15), int maxLevel=3, TermCriteria
criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01), double
derivLambda=0.5, int flags=0)
```

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

- Parameters:**
- *prevImg* – The first 8-bit single-channel or 3-channel input image
  - *nextImg* – The second input image of the same size and the same type as `prevImg`
  - *prevPts* – Vector of points for which the flow needs to be found
  - *nextPts* – The output vector of points containing the calculated new positions of the input features in the second image
  - *status* – The output status vector. Each element of the vector is set to 1 if the flow for the corresponding features has been found, 0 otherwise
  - *err* – The output vector that will contain the difference between patches around the original and moved points
  - *winSize* – Size of the search window at each pyramid level
  - *maxLevel* – 0-based maximal pyramid level number. If 0, pyramids are not used (single level), if 1, two levels are used etc.
  - *criteria* – Specifies the termination criteria of the iterative search algorithm (after the specified maximum number of iterations `criteria.maxCount` or when the search window moves by less than `criteria.epsilon`)
  - *derivLambda* – The relative weight of the spatial image derivatives impact to the optical flow estimation. If `derivLambda=0`, only the image intensity is used, `derivLambda=1`, only derivatives are used. Any other values between 0 and 1 means that both derivatives and the image intensity are used (in the corresponding proportions).

- *flags* –  
The operation flags:
  - **OPTFLOW\_USE\_INITIAL\_FLOW** - use initial estimations stored in `nextPts`. If the flag is not set, then initially  $\text{nextPts} \leftarrow \text{prevPts}$

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids, see .

## calcOpticalFlowFarneback

---

void **calcOpticalFlowFarneback**(const Mat& *prevImg*, const Mat& *nextImg*, Mat& *flow*, double *pyrScale*, int *levels*, int *winsize*, int *iterations*, int *polyN*, double *polySigma*, int *flags*)

Computes dense optical flow using Gunnar Farneback's algorithm

- Parameters:**
- *prevImg* – The first 8-bit single-channel input image
  - *nextImg* – The second input image of the same size and the same type as `prevImg`
  - *flow* – The computed flow image; will have the same size as `prevImg` and type **CV\_32FC2**
  - *pyrScale* – Specifies the image scale (<1) to build the pyramids for each image. `pyrScale=0.5` means the classical pyramid, where each next layer is twice smaller than the previous
  - *levels* – The number of pyramid layers, including the initial image. `levels=1` means that no extra layers are created and only the original images are used
  - *winsize* – The averaging window size; The larger values increase the algorithm robustness to image noise and give more chances for fast motion detection, but yield more blurred motion field
  - *iterations* – The number of iterations the algorithm does at each pyramid level
  - *polyN* – Size of the pixel neighborhood used to find polynomial expansion in each pixel. The larger values mean that the image will be approximated with smoother surfaces, yielding more robust algorithm and more blurred motion field. Typically, `polyN=5` or `7`
  - *polySigma* – Standard deviation of the Gaussian that is used to smooth derivatives that are used as a basis for the polynomial expansion. For `polyN=5` you can set `polySigma=1.1`, for `polyN=7` a good value would be `polySigma=1.5`

- *flags* –  
The operation flags; can be a combination of the following:
  - **OPTFLOW\_USE\_INITIAL\_FLOW** - Use the input `flow` as the initial flow approximation
  - **OPTFLOW\_FARNEBACK\_GAUSSIAN** - Use a Gaussian `winsize` × `winsize` filter instead of box filter of the same size for optical flow estimation. Usually, this option gives more accurate flow than with a box filter, at the cost of lower speed (and normally `winsize` for a Gaussian window should be set to a larger value to achieve the same level of robustness)

The function finds optical flow for each `prevImg` pixel using the algorithm so that

$$\text{prevImg}(x, y) \sim \text{nextImg}(\text{flow}(x, y)[0], \text{flow}(x, y)[1])$$

## updateMotionHistory

---

void **updateMotionHistory**(const Mat& *silhouette*, Mat& *mhi*, double *timestamp*, double *duration*)

Updates the motion history image by a moving silhouette.

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

The function updates the motion history image as following:

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

That is, MHI pixels where motion occurs are set to the current `timestamp`, while the pixels where motion happened last time a long time ago are cleared.

The function, together with [calcMotionGradient](#) and [calcGlobalOrientation](#), implements the motion templates technique, described in and . See also the OpenCV sample `motempl.c` that demonstrates the use of all the motion template functions.

## calcMotionGradient

---

void **calcMotionGradient**(const Mat& *mhi*, Mat& *mask*, Mat& *orientation*, double *delta1*, double *delta2*, int *apertureSize*=3)

Calculates the gradient orientation of a motion history image.

- Parameters:**
- *mhi* – Motion history single-channel floating-point image
  - *mask* – The output mask image; will have the type **CV\_8UC1** and the same size as *mhi*. Its non-zero elements will mark pixels where motion gradient data is correct
  - *orientation* – The output motion gradient orientation image; will have the same type and the same size as *mhi*. Each pixel of it will contain the orientation in degrees, from 0 to 360.
  - *delta1*, *delta2* – The minimal and maximal allowed difference between *mhi* values in a pixel neighborhood. That is, the function finds the minimum ( $m(x, y)$ ) and maximum ( $M(x, y)$ ) *mhi* values over  $3 \times 3$  neighborhood of each pixel and marks the motion orientation at  $(x, y)$  as valid only if  $\min(\text{delta1}, \text{delta2}) \leq M(x, y) - m(x, y) \leq \max(\text{delta1}, \text{delta2})$
  - *apertureSize* – The aperture size of *Sobel* operator

The function calculates the gradient orientation at each pixel  $(x, y)$  as:

$$\text{orientation}(x, y) = \arctan \frac{dmhi/dy}{dmhi/dx}$$

(in fact, *fastArctan* and *phase* are used, so that the computed angle is measured in degrees and covers the full range 0..360). Also, the *mask* is filled to indicate pixels where the computed angle is valid.

## calcGlobalOrientation

---

double **calcGlobalOrientation**(const Mat& *orientation*, const Mat& *mask*, const Mat& *mhi*, double *timestamp*, double *duration*)

Calculates the global motion orientation in some selected region.

- Parameters:**
- *orientation* – Motion gradient orientation image, calculated by the function *calcMotionGradient*
  - *mask* – Mask image. It may be a conjunction of a valid gradient mask, also calculated by *calcMotionGradient*, and the mask of the region, whose direction needs to be calculated
  - *mhi* – The motion history image, calculated by

### [updateMotionHistory](#)

- *timestamp* – The timestamp passed to [updateMotionHistory](#)
- *duration* – Maximal duration of motion track in milliseconds, passed to [updateMotionHistory](#)

The function calculates the average motion direction in the selected region and returns the angle between 0 degrees and 360 degrees. The average direction is computed from the weighted orientation histogram, where a recent motion has larger weight and the motion occurred in the past has smaller weight, as recorded in `mhi`.

## CamShift

---

RotatedRect **CamShift**(const Mat& *problmage*, Rect& *window*, TermCriteria *criteria*)

Finds the object center, size, and orientation

- Parameters:**
- *problmage* – Back projection of the object histogram; see [calcBackProject](#)
  - *window* – Initial search window
  - *criteria* – Stop criteria for the underlying [meanShift](#)

The function implements the CAMSHIFT object tracking algorithm *Bradski98*. First, it finds an object center using [meanShift](#) and then adjust the window size and finds the optimal rotation. The function returns the rotated rectangle structure that includes the object position, size and the orientation. The next position of the search window can be obtained with `RotatedRect::boundingRect()`.

See the OpenCV sample `camshiftdemo.c` that tracks colored objects.

## meanShift

---

int **meanShift**(const Mat& *problmage*, Rect& *window*, TermCriteria *criteria*)

Finds the object on a back projection image.

- Parameters:**
- *problmage* – Back projection of the object histogram; see [calcBackProject](#)
  - *window* – Initial search window
  - *criteria* – The stop criteria for the iterative search algorithm

The function implements iterative object search algorithm. It takes the object back

projection on input and the initial position. The mass center in window of the back projection image is computed and the search window center shifts to the mass center. The procedure is repeated until the specified number of iterations `criteria.maxCount` is done or until the window center shifts by less than `criteria.epsilon`. The algorithm is used inside [CamShift](#) and, unlike [CamShift](#), the search window size or orientation do not change during the search. You can simply pass the output of [calcBackProject](#) to this function, but better results can be obtained if you pre-filter the back projection and remove the noise (e.g. by retrieving connected components with [findContours](#), throwing away contours with small area ([contourArea](#)) and rendering the remaining contours with [drawContours](#))

## KalmanFilter

---

### Kalman filter class

```
class KalmanFilter
{
public:
    KalmanFilter();
    KalmanFilter(int dynamParams, int measureParams, int control
void init(int dynamParams, int measureParams, int control
// predicts statePre from statePost
const Mat& predict(const Mat& control=Mat());
// corrects statePre based on the input measurement vectc
// and stores the result to statePost.
const Mat& correct(const Mat& measurement);

    Mat statePre; // predicted state (x'(k)):
// x(k)=A*x(k-1)+B*u(k)
    Mat statePost; // corrected state (x(k)):
// x(k)=x'(k)+K(k)*(z(k)-H*x'(k)
    Mat transitionMatrix; // state transition matrix (A)
    Mat controlMatrix; // control matrix (B)
// (it is not used if there is
    Mat measurementMatrix; // measurement matrix (H)
    Mat processNoiseCov; // process noise covariance matri
    Mat measurementNoiseCov; // measurement noise covariance n
    Mat errorCovPre; // priori error estimate covarian
// P'(k)=A*P(k-1)*At + Q)*/
    Mat gain; // Kalman gain matrix (K(k)):
// K(k)=P'(k)*Ht*inv(H*P'(k)*H
    Mat errorCovPost; // posteriori error estimate cova
// P(k)=(I-K(k)*H)*P'(k)
    ...
};
```

The class implements standard Kalman filter [http://en.wikipedia.org/wiki/Kalman\\_filter](http://en.wikipedia.org/wiki/Kalman_filter). However, you can modify `transitionMatrix`, `controlMatrix` and `measurementMatrix` to get the extended Kalman filter functionality. See the OpenCV sample `kalman.c`

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Structural Analysis and Shape Descriptors

## moments

Moments **moments** (const Mat& *array*, bool *binaryImage=false*)

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

where **cvMoments ()** is defined as:

```
class Moments
{
public:
    Moments();
    Moments(double m00, double m10, double m01, double m20,
          double m02, double m30, double m21, double m12,
          Moments( const CvMoments& moments );
    operator CvMoments() const;

    // spatial moments
    double m00, m10, m01, m20, m11, m02, m30, m21, m12, m
    // central moments
    double mu20, mu11, mu02, mu30, mu21, mu12, mu03;
    // central normalized moments
    double nu20, nu11, nu02, nu30, nu21, nu12, nu03;
};
```

- Parameters:**
- *array* – A raster image (single-channel, 8-bit or floating-point 2D array) or an array ( $1 \times N$  or  $N \times 1$ ) of 2D points (`Point` or `Point2f`)
  - *binaryImage* – (For images only) If it is true, then all the non-zero image pixels are treated as 1's

The function computes moments, up to the 3rd order, of a vector shape or a rasterized shape. In case of a raster image, the spatial moments **Moments::m<sub>ji</sub>** are computed as:

$$m_{ji} = \sum_{x,y} (\text{array}(x,y) \cdot x^j \cdot y^i),$$

the central moments **Moments::mu<sub>ji</sub>** are computed as:

$$mu_{ji} = \sum_{x,y} (\text{array}(x,y) \cdot (x - \bar{x})^j \cdot (y - \bar{y})^i)$$

where  $(\bar{x}, \bar{y})$  is the mass center:

$$\bar{x} = \frac{m_{10}}{m_{00}}, \quad \bar{y} = \frac{m_{01}}{m_{00}}$$

and the normalized central moments `Moments::nuij` are computed as:

$$\text{nu}_{ji} = \frac{m_{ji}}{m_{00}^{(i+j)/2+1}}$$

Note that  $m_{00} = m_{00}$ ,  $\text{nu}_{00} = 1$ ,  $\text{nu}_{10} = m_{10} = m_{01} = m_{10} = 0$ , hence the values are not stored.

The moments of a contour are defined in the same way, but computed using Green's formula (see [http://en.wikipedia.org/wiki/Green\\_theorem](http://en.wikipedia.org/wiki/Green_theorem)), therefore, because of a limited raster resolution, the moments computed for a contour will be slightly different from the moments computed for the same contour rasterized.

See also: [contourArea](#), [arcLength](#)

## HuMoments

void **HuMoments**(const Moments& *moments*, double *hu*[7])

Calculates the seven Hu invariants.

- Parameters:**
- *moments* – The input moments, computed with [moments](#)
  - *hu* – The output Hu invariants

The function calculates the seven Hu invariants, see [http://en.wikipedia.org/wiki/Image\\_moment](http://en.wikipedia.org/wiki/Image_moment), that are defined as:

$$\begin{aligned} h[0] &= \eta_{20} + \eta_{02} \\ h[1] &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\ h[2] &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\ h[3] &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\ h[4] &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03}) \\ h[5] &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\ h[6] &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03}) \end{aligned}$$

where  $\eta_{ji}$  stand for `Moments::nuji`.

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.

See also: [matchShapes](#)

## findContours

---

```
void findContours(const Mat& image, vector<vector<Point> >&
contours, vector<Vec4i>& hierarchy, int mode, int method, Point offset=Point())
```

```
void findContours(const Mat& image, vector<vector<Point> >&
contours, int mode, int method, Point offset=Point())
```

Finds the contours in a binary image.

- Parameters:**
- *image* – The source, an 8-bit single-channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as binary. You can use [compare](#), [inRange](#), [threshold](#), [adaptiveThreshold](#), [Canny](#) etc. to create a binary image out of a grayscale or color one. The function modifies the image by extracting the contours
  - *contours* – The detected contours. Each contour is stored as a vector of points
  - *hierarchy* – The optional output vector that will contain information about the image topology. It will have as many elements as the number of contours. For each contour `contours[i]`, the elements `hierarchy[i][0]`, `hierarchy[i][1]`, `hierarchy[i][2]`, `hierarchy[i][3]` will be set to 0-based indices of the next and previous contours at the same hierarchical level, the first child contour and the parent contour, respectively. If some contour `i` there is no next, previous, parent or nested contour, the corresponding elements of `hierarchy[i]` will be negative
  - *mode* – The contour retrieval mode
    - **RETR\_EXTERNAL** - retrieves only the extreme outer contours; all other contours are discarded. set `hierarchy[i][2]=hierarchy[i][3]=-1` for all contours
    - **RETR\_LIST** - retrieves all of the contours without establishing any hierarchical relationships
    - **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. If inside a hole of a connected component there is another contour, it will still be put on the top level
    - **CV\_RETR\_TREE** - retrieves all of the contours and reconstructs a full hierarchy of nested contours. This full hierarchy is built and stored in `hierarchy`. See `contours.c` demo
  - *method* –

The contour approximation method.

- **CV\_CHAIN\_APPROX\_NONE** - stores absolutely all the contour points. That is, every 2 points of a contour stored with this method are 8-connected neighbors of each other
- **CV\_CHAIN\_APPROX\_SIMPLE** - compresses horizontal, vertical and diagonal segments and leaves only their end points. E.g. an upright rectangular contour will be encoded with 4 points
- **CV\_CHAIN\_APPROX\_TC89\_L1, CV\_CHAIN\_APPROX\_TC89\_K1** - applies one of the flavors of the Teh-Chin chain approximation algorithm; see [Teh-Chin](#)
- *offset* – The optional offset, by which every contour point is shifted. It is useful if the contours are extracted from the image ROI and they should be analyzed in the whole image context

The function retrieves contours from the binary image using the algorithm [findContours](#). The contours are a useful tool for shape analysis and object detection and recognition. See `squares.c` in the OpenCV sample directory.

## drawContours

---

```
void drawContours(Mat& image, const vector<vector<Point> >& contours, int contourIdx, const Scalar& color, int thickness=1, int lineType=8, const vector<Vec4i>& hierarchy=vector<Vec4i>(), int maxLevel=INT_MAX, Point offset=Point())
```

Draws contours' outlines or filled contours.

- Parameters:**
- *image* – The destination image
  - *contours* – All the input contours. Each contour is stored as a point vector
  - *contourIdx* – Indicates the contour to draw. If it is negative, all the contours are drawn
  - *color* – The contours' color
  - *thickness* – Thickness of lines the contours are drawn with. If it is negative (e.g. `thickness=CV_FILLED`), the contour interiors are drawn.
  - *lineType* – The line connectivity; see [line](#) description
  - *hierarchy* – The optional information about hierarchy. It is only needed if you want to draw only some of the contours (see `maxLevel`)
  - *maxLevel* – Maximal level for drawn contours. If 0, only the specified contour is drawn. If 1, the function draws the contour (s) and all the nested contours. If 2, the function draws the contours, all the nested contours and all the nested into nested contours etc. This parameter is only taken into account

when there is hierarchy available.

- *offset* – The optional contour shift parameter. Shift all the drawn contours by the specified *offset* =  $(dx, dy)$

The function draws contour outlines in the image if **thickness**  $\geq 0$  or fills the area bounded by the contours if **thickness**  $< 0$ . Here is the example on how to retrieve connected components from the binary image and label them

---

```

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

using namespace cv;

int main( int argc, char** argv )
{
    Mat src;
    // the first command line parameter must be file name
    // (black-n-white) image
    if( argc != 2 || !(src=imread(argv[1], 0)).data)
        return -1;

    Mat dst = Mat::zeros(src.rows, src.cols, CV_8UC3);

    src = src > 1;
    namedWindow( "Source", 1 );
    imshow( "Source", src );

    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    findContours( src, contours, hierarchy,
                 CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );

    // iterate through all the top-level contours,
    // draw each connected component with its own random color
    int idx = 0;
    for( ; idx >= 0; idx = hierarchy[idx][0] )
    {
        Scalar color( rand()&255, rand()&255, rand()&255 );
        drawContours( dst, contours, idx, color, CV_FILLED );
    }

    namedWindow( "Components", 1 );
    showImage( "Components", dst );
    waitKey(0);
}

```

---

## approxPolyDP

---

void **approxPolyDP**(const Mat& *curve*, vector<Point>& *approxCurve*, double *epsilon*, bool *closed*)

void **approxPolyDP**(const Mat& *curve*, vector<Point2f>& *approxCurve*, double *epsilon*, bool *closed*)

Approximates polygonal curve(s) with the specified precision.

- Parameters:**
- *curve* – The polygon or curve to approximate. Must be  $1 \times N$  or  $N \times 1$  matrix of type **CV\_32SC2** or **CV\_32FC2**. You can also pass `vector<Point>` or `vector<Point2f>` that will be automatically converted to the matrix of the proper size and type
  - *approxCurve* – The result of the approximation; The type should match the type of the input curve
  - *epsilon* – Specifies the approximation accuracy. This is the maximum distance between the original curve and its approximation
  - *closed* – If true, the approximated curve is closed (i.e. its first and last vertices are connected), otherwise it's not

The functions `approxPolyDP` approximate a curve or a polygon with another curve/polygon with less vertices, so that the distance between them is less or equal to the specified precision. It used Douglas-Peucker algorithm [http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker\\_algorithm](http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm)

## arcLength

---

double **arcLength**(const Mat& *curve*, bool *closed*)

Calculates a contour perimeter or a curve length.

- Parameters:**
- *curve* – The input vector of 2D points, represented by **CV\_32SC2** or **CV\_32FC2** matrix or by `vector<Point>` or `vector<Point2f>`
  - *closed* – Indicates, whether the curve is closed or not

The function computes the curve length or the closed contour perimeter.

## boundingRect

---

Rect **boundingRect**(const Mat& *points*)

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

**Parameter:** *points* – The input 2D point set, represented by **CV\_32SC2** or **CV\_32FC2** matrix or by `vector<Point>` or `vector<Point2f>`

The function calculates and returns the minimal up-right bounding rectangle for the specified point set.

## estimateRigidTransform

---

Mat **estimateRigidTransform**(const Mat& *srcpt*, const Mat& *dstpt*, bool *fullAffine*)

Computes optimal affine transformation between two 2D point sets

**Parameters:**

- *srcpt* – The first input 2D point set
- *dst* – The second input 2D point set of the same size and the same type as *A*
- *fullAffine* – If true, the function finds the optimal affine transformation with no any additional restrictions (i.e. there are 6 degrees of freedom); otherwise, the class of transformations to choose from is limited to combinations of translation, rotation and uniform scaling (i.e. there are 5 degrees of freedom)

The function finds the optimal affine transform  $[A|b]$  (a  $2 \times 3$  floating-point matrix) that approximates best the transformation from  $\text{srcpt}_i$  to  $\text{dstpt}_i$ :

$$[A^*|b^*] = \arg \min_{[A|b]} \sum_i \|\text{dstpt}_i - A\text{srcpt}_i^T - b\|^2$$

where  $[A|b]$  can be either arbitrary (when `fullAffine=true`) or have form

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ -a_{12} & a_{11} & b_2 \end{bmatrix}$$

when `fullAffine=false`.

See also: [getAffineTransform](#), [getPerspectiveTransform](#), [findHomography](#)

## estimateAffine3D

---

int **estimateAffine3D**(const Mat& *srcpt*, const Mat& *dstpt*, Mat& *out*, `vector<uchar>& outliers`, double `ransacThreshold = 3.0`, double `confidence = 0.99`)

Computes optimal affine transformation between two 3D point sets

- Parameters:**
- *srcpt* – The first input 3D point set
  - *dstpt* – The second input 3D point set
  - *out* – The output 3D affine transformation matrix  $3 \times 4$
  - *outliers* – The output vector indicating which points are outliers
  - *ransacThreshold* – The maximum reprojection error in RANSAC algorithm to consider a point an inlier
  - *confidence* – The confidence level, between 0 and 1, with which the matrix is estimated

The function estimates the optimal 3D affine transformation between two 3D point sets using RANSAC algorithm.

## contourArea

---

double **contourArea**(const Mat& *contour*)

Calculates the contour area

**Parameter:** *contour* – The contour vertices, represented by **CV\_32SC2** or **CV\_32FC2** matrix or by `vector<Point>` or `vector<Point2f>`

The function computes the contour area. Similarly to [moments](#) the area is computed using the Green formula, thus the returned area and the number of non-zero pixels, if you draw the contour using [drawContours](#) or [fillPoly](#), can be different. Here is a short example:

---

```
vector<Point> contour;
contour.push_back(Point2f(0, 0));
contour.push_back(Point2f(10, 0));
contour.push_back(Point2f(10, 10));
contour.push_back(Point2f(5, 4));

double area0 = contourArea(contour);
vector<Point> approx;
approxPolyDP(contour, approx, 5, true);
double area1 = contourArea(approx);

cout << "area0 =" << area0 << endl <<
      "area1 =" << area1 << endl <<
      "approx poly vertices" << approx.size() << endl;
```

---

## convexHull

---

void **convexHull**(const Mat& *points*, vector<int>& *hull*, bool *clockwise=false*)

void **convexHull**(const Mat& *points*, vector<Point>& *hull*, bool *clockwise=false*)

void **convexHull**(const Mat& *points*, vector<Point2f>& *hull*, bool *clockwise=false*)

Finds the convex hull of a point set.

- Parameters:**
- *points* – The input 2D point set, represented by **CV\_32SC2** or **CV\_32FC2** matrix or by `vector<Point>` or `vector<Point2f>`
  - *hull* – The output convex hull. It is either a vector of points that form the hull, or a vector of 0-based point indices of the hull points in the original array (since the set of convex hull points is a subset of the original point set).
  - *clockwise* – If true, the output convex hull will be oriented clockwise, otherwise it will be oriented counter-clockwise. Here, the usual screen coordinate system is assumed - the origin is at the top-left corner, x axis is oriented to the right, and y axis is oriented downwards.

The functions find the convex hull of a 2D point set using Sklansky's algorithm that has  $O(N \log N)$  or  $O(N)$  complexity (where  $N$  is the number of input points), depending on how the initial sorting is implemented (currently it is  $O(N \log N)$ ). See the OpenCV sample `convexhull.c` that demonstrates the use of the different function variants.

## fitEllipse

---

RotatedRect **fitEllipse**(const Mat& *points*)

Fits an ellipse around a set of 2D points.

**Parameter:** *points* – The input 2D point set, represented by **CV\_32SC2** or **CV\_32FC2** matrix or by `vector<Point>` or `vector<Point2f>`

The function calculates the ellipse that fits best (in least-squares sense) a set of 2D points. It returns the rotated rectangle in which the ellipse is inscribed.

## fitLine

---

void **fitLine**(const Mat& *points*, Vec4f& *line*, int *distType*, double *param*, double *reps*, double *aeps*)

void **fitLine**(const Mat& *points*, Vec6f& *line*, int *distType*, double *param*, double *reps*, double *aeps*)

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

- Parameters:**
- *points* – The input 2D point set, represented by **CV\_32SC2** or **CV\_32FC2** matrix or by `vector<Point>`, `vector<Point2f>`, `vector<Point3i>` or `vector<Point3f>`
  - *line* – The output line parameters. In the case of a 2d fitting, it is a vector of 4 floats ( $v_x, v_y, x_0, y_0$ ) where  $(v_x, v_y)$  is a normalized vector collinear to the line and  $(x_0, y_0)$  is some point on the line. in the case of a 3D fitting it is vector of 6 floats ( $v_x, v_y, v_z, x_0, y_0, z_0$ ) where  $(v_x, v_y, v_z)$  is a normalized vector collinear to the line and  $(x_0, y_0, z_0)$  is some point on the line
  - *distType* – The distance used by the M-estimator (see the discussion)
  - *param* – Numerical parameter (C) for some types of distances, if 0 then some optimal value is chosen
  - *reps*, *aeps* – Sufficient accuracy for the radius (distance between the coordinate origin and the line) and angle, respectively; 0.01 would be a good default value for both.

The functions `fitLine` fit 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:

distType=CV\_DIST\_L2 \*

$$\rho(r) = r^2/2 \quad (\text{the simplest and the fastest least-squares method})$$

- distType=CV\_DIST\_L1 \*

$$\rho(r) = r$$

- distType=CV\_DIST\_L12 \*

$$\rho(r) = 2 \cdot \left( \sqrt{1 + \frac{r^2}{2}} - 1 \right)$$

- distType=CV\_DIST\_FAIR \*

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

- distType=CV\_DIST\_WELLSCH \*

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

- distType=CV\_DIST\_HUBER \*

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where } C = 1.345$$

The algorithm is based on the M-estimator (<http://en.wikipedia.org/wiki/M-estimator>) technique, that iteratively fits the line using weighted least-squares algorithm and after each iteration the weights  $w_i$  are adjusted to be inversely proportional to  $\rho(r_i)$ .

## isContourConvex

---

bool **isContourConvex**(const Mat& *contour*)

Tests contour convexity.

**Parameter:** *contour* – The tested contour, a matrix of type **CV\_32SC2** or **CV\_32FC2**, or vector<Point> or vector<Point2f>

The function tests whether the input contour is convex or not. The contour must be simple, i.e. without self-intersections, otherwise the function output is undefined.

## minAreaRect

---

RotatedRect **minAreaRect**(const Mat& *points*)

Finds the minimum area rotated rectangle enclosing a 2D point set.

**Parameter:** *points* – The input 2D point set, represented by **CV\_32SC2** or



$$I_1(A, B) = \sum_{i=1..7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

- method=CV\_CONTOUR\_MATCH\_I2 \*

$$I_2(A, B) = \sum_{i=1..7} |m_i^A - m_i^B|$$

- method=CV\_CONTOUR\_MATCH\_I3 \*

$$I_3(A, B) = \sum_{i=1..7} \frac{|m_i^A - m_i^B|}{|m_i^A|}$$

where

$$m_i^A = \text{sign}(h_i^A) \cdot \log h_i^A$$

$$m_i^B = \text{sign}(h_i^B) \cdot \log h_i^B$$

and  $h_i^A, h_i^B$  are the Hu moments of  $A$  and  $B$  respectively.

## pointPolygonTest

---

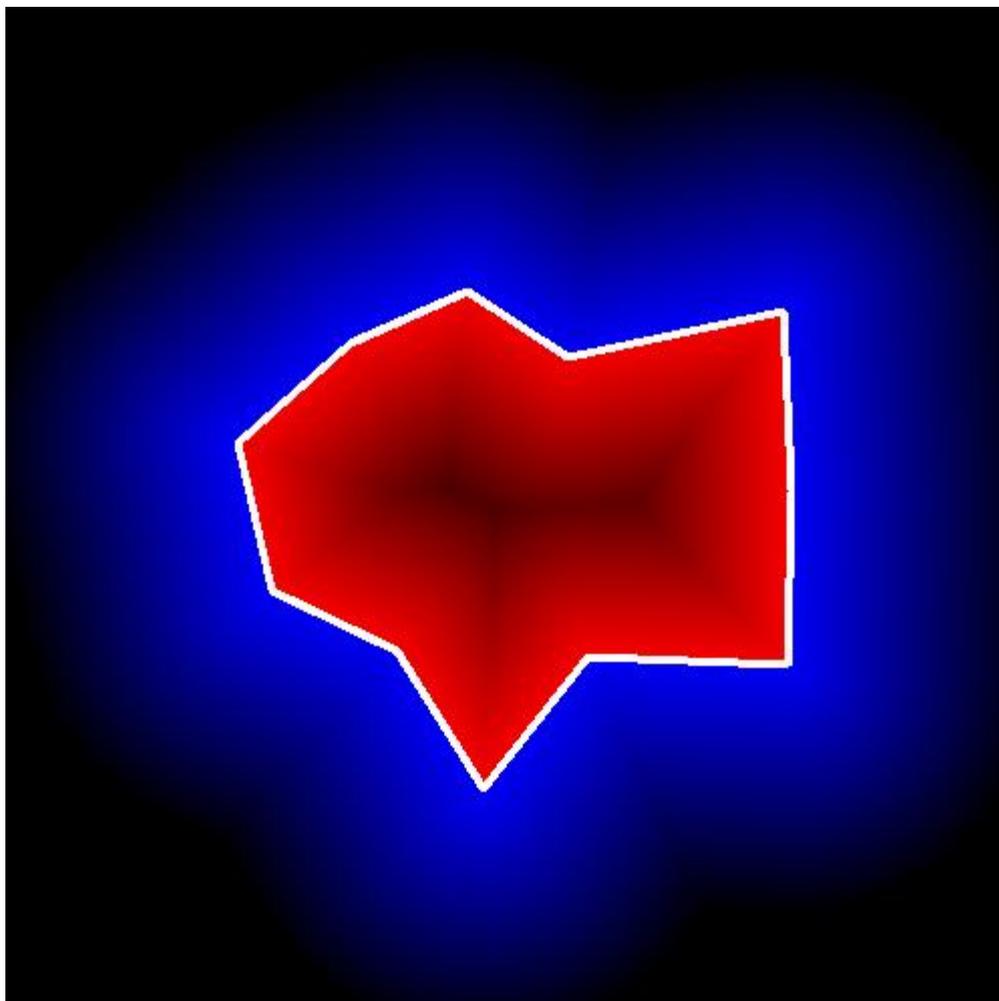
double **pointPolygonTest**(const Mat& *contour*, Point2f *pt*, bool *measureDist*)

Performs point-in-contour test.

- Parameters:**
- *contour* – The input contour
  - *pt* – The point tested against the contour
  - *measureDist* – If true, the function estimates the signed distance from the point to the nearest contour edge; otherwise, the function only checks if the point is inside or not.

The function determines whether the point is inside a contour, outside, or lies on an edge (or coincides with a vertex). It returns positive (inside), negative (outside) or zero (on an edge) value, correspondingly. When *measureDist*=false, the return value is +1, -1 and 0, respectively. Otherwise, the return value 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.



## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Planar Subdivisions

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Object Detection

---

## FeatureEvaluator

---

Base class for computing feature values in cascade classifiers

```
class FeatureEvaluator
{
public:
    // feature type
    enum { HAAR = 0, LBP = 1 };
    virtual ~FeatureEvaluator();
    // reads parameters of the features from a FileStorage node
    virtual bool read(const FileNode& node);
    // returns a full copy of the feature evaluator
    virtual Ptr<FeatureEvaluator> clone() const;
    // returns the feature type (HAAR or LBP for now)
    virtual int getFeatureType() const;

    // sets the image in which to compute the features
    // (called by CascadeClassifier::setImage)
    virtual bool setImage(const Mat& image, Size origWinSize)
    // sets window in the current image in which the features
    // will be computed (called by CascadeClassifier::runAt)
    virtual bool setWindow(Point p);

    // computes value of an ordered (numerical) feature #feat
    virtual double calcOrd(int featureIdx) const;
    // computes value of a categorical feature #featureIdx
    virtual int calcCat(int featureIdx) const;

    // static function that constructs feature evaluator
    // of the specific feature type (HAAR or LBP for now)
    static Ptr<FeatureEvaluator> create(int type);
};
```

---

## CascadeClassifier

---

The cascade classifier class for object detection

```
class CascadeClassifier
{
public:
    enum { BOOST = 0 };

    // default constructor
```

```

CascadeClassifier();
// load the classifier from file
CascadeClassifier(const string& filename);
// the destructor
~CascadeClassifier();

// checks if the classifier has been loaded or not
bool empty() const;
// loads the classifier from file. The previous content is destroyed
bool load(const string& filename);
// reads the classifier from a FileStorage node.
bool read(const FileNode& node);
// detects objects of different sizes in the input image.
// the detected objects are returned as a list of rectangles.
// scaleFactor specifies how much the image size
// is reduced at each image scale.
// minNeighbors specifies how many neighbors should
// each candidate rectangle have to retain it.
// flags - ignored
// minSize - the minimum possible object size.
// Objects smaller than that are ignored.
void detectMultiScale( const Mat& image,
                      vector<Rect>& objects,
                      double scaleFactor=1.1,
                      int minNeighbors=3, int flags=0,
                      Size minSize=Size());

// sets the image for detection
// (called by detectMultiScale at each image level)
bool setImage( Ptr<FeatureEvaluator>& feval, const Mat& image
// runs the detector at the specified point
// (the image that the detector is working with should be set
// by setImage)
int runAt( Ptr<FeatureEvaluator>& feval, Point pt );

bool is_stump_based;

int stageType;
int featureType;
int ncategories;
Size origWinSize;

Ptr<FeatureEvaluator> feval;
Ptr<CvHaarClassifierCascade> oldCascade;
};

```

---

## groupRectangles

---

```

void groupRectangles(vector<Rect>& rectList, int groupThreshold,
double eps=0.2)

```

Groups the object candidate rectangles

- Parameters:**
- *rectList* – The input/output vector of rectangles. On output there will be retained and grouped rectangles
  - *groupThreshold* – The minimum possible number of rectangles, minus 1, in a group of rectangles to retain it.
  - *eps* – The relative difference between sides of the rectangles to merge them into a group

The function is a wrapper for a generic function [partition](#). It clusters all the input rectangles using the rectangle equivalence criteria, that combines rectangles that have similar sizes and similar locations (the similarity is defined by *eps*). When *eps*=0, no clustering is done at all. If *eps*  $\rightarrow +\infty$ , all the rectangles will be put in one cluster. Then, the small clusters, containing less than or equal to *groupThreshold* rectangles, will be rejected. In each other cluster the average rectangle will be computed and put into the output rectangle list.

## matchTemplate

---

void **matchTemplate**(const Mat& *image*, const Mat& *templ*, Mat& *result*, int *method*)

Compares a template against overlapped image regions.

- Parameters:**
- *image* – Image where the search is running; should be 8-bit or 32-bit floating-point
  - *templ* – Searched template; must be not greater than the source image and have the same data type
  - *result* – A map of comparison results; will be single-channel 32-bit floating-point. If *image* is  $W \times H$  and *templ* is  $w \times h$  then *result* will be  $(W - w + 1) \times (H - h + 1)$
  - *method* – Specifies the comparison method (see below)

The function 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 available comparison methods (*I* denotes *image*, *T* template, *R* result). The summation is done over template and/or the image patch:  $x' = 0 \dots w - 1, y' = 0 \dots h - 1$

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 (when **CV\_TM\_SQDIFF** was used) or maximums (when **CV\_TM\_CCORR** or **CV\_TM\_CCOEFF** was used) 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). That is, the function can take a color template and a color image; the result will still be a single-channel image, which is easier to analyze.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# 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.  $(c_x, c_y)$  is a principal point (that is usually at the image center), and  $f_x, f_y$  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:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$x'' = x'(1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_1x'y' + p_2(r^2 + 2x'^2)$$

$$y'' = y'(1 + k_1r^2 + k_2r^4 + k_3r^6) + p_1(r^2 + 2y'^2) + 2p_2x'y'$$

where  $r^2 = x'^2 + y'^2$

$$u = f_x * x'' + c_x$$

$$v = f_y * y'' + c_y$$

$k_1, k_2, k_3$  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

$$(k_1, k_2, p_1, p_2[, k_3])$$

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  $320 \times 240$  resolution, absolutely the same distortion coefficients can be used for images of  $640 \times 480$  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.

## calibrateCamera

---

```
double calibrateCamera(const vector<vector<Point3f> >&
objectPoints, const vector<vector<Point2f> >& imagePoints, Size imageSize, Mat&
cameraMatrix, Mat& distCoeffs, vector<Mat>& rvecs, vector<Mat>& tvecs, int flags=0)
```

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

calibration pattern.

- Parameters:**
- *objectPoints* – The vector of vectors of points on the calibration pattern in its coordinate system, one vector per view. If the the same calibration pattern is shown in each view and it's fully visible then all the vectors will be the same, although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0
  - *imagePoints* – The vector of vectors of the object point projections on the calibration pattern views, one vector per a view. The projections must be in the same order as the corresponding object points.
  - *imageSize* – Size of the image, used only to initialize the intrinsic camera matrix
  - *cameraMatrix* – The output 3x3 floating-point camera matrix

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.lf$$

- CV\_CALIB\_USE\_INTRINSIC\_GUESS** and/or **CV\_CALIB\_FIX\_ASPECT\_RATIO** are specified, some or all of  $f_x$ ,  $f_y$ ,  $c_x$ ,  $c_y$  must be initialized before calling the function
- *distCoeffs* – The output 5x1 or 1x5 vector of distortion coefficients ( $k_1, k_2, p_1, p_2, k_3$ ).
  - *rvecs* – The output vector of rotation vectors (see [Rodrigues](#)), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view ( $k=0..M-1$ )
  - *tvecs* – The output vector 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  $f_x$ ,  $f_y$ ,  $c_x$ ,  $c_y$  that are optimized further. Otherwise,  $(c_x, c_y)$  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 [solvePnP](#) 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  $f_y$  as a free parameter, the ratio  $f_x/f_y$  stays the same as in the input `cameraMatrix`. When **CV\_CALIB\_USE\_INTRINSIC\_GUESS** is not set, the actual input values of  $f_x$  and  $f_y$  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.

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:

- 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).
- The the initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using [solvePnP](#)
- 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 [projectPoints](#).

The function returns the final re-projection error.

Note: if you're using a non-square (=non-NxN) grid and [findChessboardCorners](#) for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from  $(w/2 - 0.5, h/2 - 0.5)$ ,

and / or large differences between  $f_x$  and  $f_y$  (ratios of 10:1 or more)), then you've probably used `patternSize=cvSize(rows,cols)`, but should use `patternSize=cvSize(cols,rows)` in [findChessboardCorners](#).

See also: [findChessboardCorners](#), [solvePnP](#), [initCameraMatrix2D](#), [stereoCalibrate](#), [undistort](#)

## calibrationMatrixValues

---

void **calibrationMatrixValues**(const Mat& *cameraMatrix*, Size *imageSize*, double *apertureWidth*, double *apertureHeight*, double& *fovX*, double& *fovY*, double& *focalLength*, Point2d& *principalPoint*, double& *aspectRatio*)

Computes some useful camera characteristics from the camera matrix

- Parameters:**
- *cameraMatrix* – The input camera matrix that can be estimated by [calibrateCamera](#) or [stereoCalibrate](#)
  - *imageSize* – The input image size in pixels
  - *apertureWidth* – Physical width of the sensor
  - *apertureHeight* – Physical height of the sensor
  - *fovX* – The output field of view in degrees along the horizontal sensor axis
  - *fovY* – The output field of view in degrees along the vertical sensor axis
  - *focalLength* – The focal length of the lens in mm
  - *principalPoint* – The principal point in pixels
  - *aspectRatio* –  $f_y/f_x$

The function computes various useful camera characteristics from the previously estimated camera matrix.

## composeRT

---

void **composeRT**(const Mat& *rvec1*, const Mat& *tvec1*, const Mat& *rvec2*, const Mat& *tvec2*, Mat& *rvec3*, Mat& *tvec3*)

void **composeRT**(const Mat& *rvec1*, const Mat& *tvec1*, const Mat& *rvec2*, const Mat& *tvec2*, Mat& *rvec3*, Mat& *tvec3*, Mat& *dr3dr1*, Mat& *dr3dt1*, Mat& *dr3dr2*, Mat& *dr3dt2*, Mat& *dt3dr1*, Mat& *dt3dt1*, Mat& *dt3dr2*, Mat& *dt3dt2*)

Combines two rotation-and-shift transformations

- Parameters:**
- *rvec1* – The first rotation vector

- *tvec1* – The first translation vector
- *rvec2* – The second rotation vector
- *tvec2* – The second translation vector
- *rvec3* – The output rotation vector of the superposition
- *tvec3* – The output translation vector of the superposition
- *d??d??* – The optional output derivatives of *rvec3* or *tvec3* w.r.t. *rvec?* or *tvec?*

The functions compute:

$$\begin{aligned} \mathbf{rvec3} &= \mathbf{rodrigues}^{-1}(\mathbf{rodrigues}(\mathbf{rvec2}) \cdot \mathbf{rodrigues}(\mathbf{rvec1})) \\ \mathbf{tvec3} &= \mathbf{rodrigues}(\mathbf{rvec2}) \cdot \mathbf{tvec1} + \mathbf{tvec2} \end{aligned}$$

where *rodrigues* denotes a rotation vector to rotation matrix transformation, and *rodrigues*<sup>-1</sup> denotes the inverse transformation, see [Rodrigues](#).

Also, the functions can compute the derivatives of the output vectors w.r.t the input vectors (see [matMulDeriv](#)). The functions are used inside [stereoCalibrate](#) but can also be used in your own code where Levenberg-Marquardt or another gradient-based solver is used to optimize a function that contains matrix multiplication.

## computeCorrespondEpilines

---

void **computeCorrespondEpilines**(const Mat& *points*, int *whichImage*, const Mat& *F*, vector<Vec3f>& *lines*)

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

- Parameters:**
- *points* – The input points.  $N \times 1$  or  $1 \times N$  matrix of type **CV\_32FC2** or `vector<Point2f>`
  - *whichImage* – Index of the image (1 or 2) that contains the points
  - *F* – The fundamental matrix that can be estimated using [findFundamentalMat](#) or [stereoRectify](#).
  - *lines* – The output vector of the corresponding to the points epipolar lines in the other image. 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

---

void **convertPointsHomogeneous**(const Mat& *src*,  
vector<Point3f>& *dst*)

void **convertPointsHomogeneous**(const Mat& *src*,  
vector<Point2f>& *dst*)

Convert points to/from homogenous coordinates.

- Parameters:**
- *src* – The input array or vector of 2D or 3D points
  - *dst* – The output vector of 3D or 2D points, respectively

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

$$(x, y[, z], w) \rightarrow (x', y'[, z'])$$

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.

## decomposeProjectionMatrix

---

void **decomposeProjectionMatrix**(const Mat& *projMatrix*,  
Mat& *cameraMatrix*, Mat& *rotMatrix*, Mat& *transVect*)

void **decomposeProjectionMatrix**(const Mat& *projMatrix*, Mat& *cameraMatrix*, Mat& *rotMatrix*, Mat& *transVect*, Mat& *rotMatrixX*, Mat& *rotMatrixY*, Mat& *rotMatrixZ*, Vec3d& *eulerAngles*)

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

- Parameters:**
- *P* – The 3x4 input projection matrix P
  - *cameraMatrix* – The output 3x3 camera matrix K
  - *rotMatrix* – The output 3x3 external rotation matrix R
  - *transVect* – The output 4x1 translation vector T
  - *rotMatrX* – Optional 3x3 rotation matrix around x-axis
  - *rotMatrY* – Optional 3x3 rotation matrix around y-axis
  - *rotMatrZ* – Optional 3x3 rotation matrix around z-axis
  - *eulerAngles* – Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on [RQDecomp3x3](#).

## drawChessboardCorners

---

void **drawChessboardCorners**(Mat& *image*, Size *patternSize*, const Mat& *corners*, bool *patternWasFound*)

Renders the detected chessboard corners.

- Parameters:**
- *image* – The destination image; it must be an 8-bit color image
  - *patternSize* – The number of inner corners per chessboard row and column. (`patternSize = cvSize(points_per_row, points_per_colum) = cvSize(columns, rows)`)
  - *corners* – The array of corners detected
  - *patternWasFound* – Indicates whether the complete board was found or not. One may just pass the return value `FindChessboardCorners`bgroup({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

---

```
bool findChessboardCorners(const Mat& image, Size
patternSize, vector<Point2f>& corners, int
flags=CV_CALIB_CB_ADAPTIVE_THRESH+CV_CALIB_CB_NORMALIZE_IMAGE)
```

Finds the positions of the internal corners of the chessboard.

- Parameters:**
- *image* – Source chessboard view; it must be an 8-bit grayscale or color image
  - *patternSize* – The number of inner corners per chessboard row and column ( *patternSize* = *cvSize*(*points\_per\_row*,*points\_per\_colum*) = *cvSize*(*columns*,*rows*) )
  - *corners* – The output array of corners detected
  - *flags* – Various operation flags, can be 0 or a combination of the following values:
    - **CV\_CALIB\_CB\_ADAPTIVE\_THRESH** - use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
    - **CV\_CALIB\_CB\_NORMALIZE\_IMAGE** - normalize the image gamma with [equalizeHist](#) before applying fixed or adaptive thresholding.
    - **CV\_CALIB\_CB\_FILTER\_QUADS** - use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

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 [cornerSubPix](#).

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).

## solvePnP

---

void **solvePnP**(const Mat& *objectPoints*, const Mat& *imagePoints*, const Mat& *cameraMatrix*, const Mat& *distCoeffs*, Mat& *rvec*, Mat& *tvec*, bool *useExtrinsicGuess=false*)

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

- Parameters:**
- *objectPoints* – The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points. Can also pass `vector<Point3f>` here.
  - *imagePoints* – The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points. Can also pass `vector<Point2f>` here.
  - *cameraMatrix* – The input camera matrix  $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$
  - *distCoeffs* – The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients  $(k_1, k_2, p_1, p_2, k_3)$ . If it is NULL, all of the distortion coefficients are set to 0
  - *rvec* – The output rotation vector (see [Rodrigues](#)) that (together with *tvec*) brings points from the model coordinate system to the camera coordinate system
  - *tvec* – The output translation vector

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 [projectPoints](#)) `objectPoints`.

## findFundamentalMat

---

Mat **findFundamentalMat**(const Mat& *points1*, const Mat& *points2*, vector<uchar>& *status*, int *method=FM\_RANSAC*, double *param1=3.*, double *param2=0.99*)

Mat **findFundamentalMat**(const Mat& *points1*, const Mat& *points2*, int *method=FM\_RANSAC*, double *param1=3.*, double *param2=0.99*)

Calculates the fundamental matrix from the corresponding points in two images.

- Parameters:**
- *points1* – Array of  $N$  points from the first image.. The point coordinates should be floating-point (single or double precision)
  - *points2* – Array of the second image points of the same size and format as *points1*
  - *method* – Method for computing the fundamental matrix
    - **CV\_FM\_7POINT** - for a 7-point algorithm.  $N = 7$
    - **CV\_FM\_8POINT** - for an 8-point algorithm.  $N \geq 8$
    - **CV\_FM\_RANSAC** - for the RANSAC algorithm.  $N \geq 8$
    - **CV\_FM\_LMEDS** - for the LMedS algorithm.  $N \geq 8$
  - *param1* – The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise
  - *param2* – The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct
  - *status* – The 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 found fundamental matrix. Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (  $9 \times 3$  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.

---

```
// Example. Estimation of fundamental matrix using RANSAC
int point_count = 100;
vector<Point2f> points1(point_count);
vector<Point2f> points2(point_count);

// initialize the points here ... */
for( int i = 0; i < point_count; i++ )
```

```

{
    points1[i] = ...;
    points2[i] = ...;
}

Mat fundamental_matrix =
    findFundamentalMat(points1, points2, FM_RANSAC, 3, 0.99);

```

## findHomography

Mat **findHomography**(const Mat& *srcPoints*, const Mat& *dstPoints*,  
Mat& *status*, int *method*=0, double *ransacReprojThreshold*=0)

Mat **findHomography**(const Mat& *srcPoints*, const Mat& *dstPoints*,  
vector<uchar>& *status*, int *method*=0, double *ransacReprojThreshold*=0)

Mat **findHomography**(const Mat& *srcPoints*, const Mat& *dstPoints*, int  
*method*=0, double *ransacReprojThreshold*=0)

Finds the perspective transformation between two planes.

**param srcPoints:**

Coordinates of the points in the original plane, a matrix of  
vector<Point2f>.

**param dstPoints:**

Coordinates of the points in the target plane, a matrix of  
vector<Point2f>.

**param method:** The method used to computed homography matrix; one of the follow

- **0** - a regular method using all the points
- **CV\_RANSAC** - RANSAC-based robust method
- **CV\_LMEDS** - Least-Median robust method

**param ransacReprojThreshold:**

The maximum allowed reprojection error to treat a point pair as an  
method only). That is, if

$$\|dstPoints_i - convertPointHomogeneous(HsrcPoints_i)\| >$$

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.

**param status:** The optional output mask set by a robust method

(**CV\_RANSAC** or **CV\_LMEDS**). Note that the input mask

values are ignored.

The functions find and return 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](#)

## getDefaultNewCameraMatrix

---

Mat **getDefaultNewCameraMatrix**(const Mat& *cameraMatrix*, Size *imgSize*=Size(), bool *centerPrincipalPoint*=false)

Returns the default new camera matrix

- Parameters:**
- *cameraMatrix* – The input camera matrix
  - *imageSize* – The camera view image size in pixels
  - *centerPrincipalPoint* – Indicates whether in the new camera matrix the principal point should be at the image center or not

The function returns the camera matrix that is either an exact copy of the input *cameraMatrix* (when *centerPrincipalPoint*=false), or the modified one (when *centerPrincipalPoint*=true).

In the latter case the new camera matrix will be:

$$\begin{bmatrix} f_x & 0 & (\text{imgSize.width} - 1) * 0.5 \\ 0 & f_y & (\text{imgSize.height} - 1) * 0.5 \\ 0 & 0 & 1 \end{bmatrix},$$

where  $f_x$  and  $f_y$  are (0,0) and (1,1) elements of *cameraMatrix*, respectively.

By default, the undistortion functions in OpenCV (see `initUndistortRectifyMap`, `undistort`) do not move the principal point. However, when you work with stereo, it's important to move the principal points in both views to the same y-coordinate (which is required by most of stereo correspondence algorithms), and maybe to the same x-coordinate too. So you can form the new camera matrix for each view, where the principal points will be at the center.

## getOptimalNewCameraMatrix

---

Mat **getOptimalNewCameraMatrix**(const Mat& *cameraMatrix*, const Mat& *distCoeffs*, Size *imageSize*, double *alpha*, Size *newImgSize*=Size(), Rect\* *validPixROI*=0)

Returns the new camera matrix based on the free scaling parameter

- Parameters:**
- *cameraMatrix* – The input camera matrix
  - *distCoeffs* – The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients ( $k_1, k_2, p_1, p_2, k_3$ ).
  - *imageSize* – The original image size

- *alpha* – The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see [stereoRectify](#)
- *newImageSize* – The image size after rectification. By default it will be set to `imageSize`.
- *validPixROI* – The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1`, `roi2` description in [stereoRectify](#)

The function computes and returns 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 `bgroup({newImageSize})` should be passed to [initUndistortRectifyMap](#) to produce the maps for [remap](#).

## initCameraMatrix2D

---

Mat **initCameraMatrix2D**(const vector<vector<Point3f> >& *objectPoints*, const vector<vector<Point2f> >& *imagePoints*, Size *imageSize*, double *aspectRatio*=1.)

Finds the initial camera matrix from the 3D-2D point correspondences

- Parameters:**
- *objectPoints* – The vector of vectors of the object points. See [calibrateCamera](#)
  - *imagePoints* – The vector of vectors of the corresponding image points. See [calibrateCamera](#)
  - *imageSize* – The image size in pixels; used to initialize the principal point
  - *aspectRatio* – If it is zero or negative, both  $f_x$  and  $f_y$  are estimated independently. Otherwise  $f_x = f_y * \text{aspectRatio}$

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.

## initUndistortRectifyMap

---

void **initUndistortRectifyMap**(const Mat& *cameraMatrix*,

const Mat& *distCoeffs*, const Mat& *R*, const Mat& *newCameraMatrix*, Size *size*, int *m1type*, Mat& *map1*, Mat& *map2*)

Computes the undistortion and rectification transformation map.

**Parameters:**

- *cameraMatrix* – The input camera matrix  $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
- *distCoeffs* – The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients ( $k_1, k_2, p_1, p_2, k_3$ ).
- *R* – The optional 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 assumed
- *newCameraMatrix* – The new camera matrix  $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$
- *size* – The undistorted image size
- *m1type* – The type of the first output map, can be **CV\_32FC1** or **CV\_16SC2**. See [convertMaps](#)
- *map1* – The first output map
- *map2* – The second output map

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:

$$\begin{aligned}
x &\leftarrow (u - c'_x) / f'_x \\
y &\leftarrow (v - c'_y) / f'_y \\
[X Y W]^T &\leftarrow R^{-1} * [x y 1]^T \\
x' &\leftarrow X/W \\
y' &\leftarrow Y/W \\
x'' &\leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2 (r^2 + 2x'^2) \\
y'' &\leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1 (r^2 + 2y'^2) + 2p_2 x' y' \\
map_x(u, v) &\leftarrow x'' f_x + c_x \\
map_y(u, v) &\leftarrow y'' f_y + c_y
\end{aligned}$$

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

$$R = \text{cameraMatrix}^{-1} \cdot H \cdot \text{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

## matMulDeriv

---

void **matMulDeriv**(const Mat& A, const Mat& B, Mat& dABdA, Mat& dABdB)

Computes partial derivatives of the matrix product w.r.t each multiplied matrix

- Parameters:**
- $A$  – The first multiplied matrix
  - $B$  – The second multiplied matrix
  - $dABdA$  – The first output derivative matrix  $d(A*B) / dA$  of size  $A.rows * B.cols \times A.rows * A.cols$
  - $dABdB$  – The second output derivative matrix  $d(A*B) / dB$  of size  $A.rows * B.cols \times B.rows * B.cols$

The function computes the partial derivatives of the elements of the matrix product  $A * B$  w.r.t. the elements of each of the two input matrices. The function is used to compute Jacobian matrices in [stereoCalibrate](#), but can also be used in any other similar optimization function.

## projectPoints

---

void **projectPoints**(const Mat& *objectPoints*, const Mat& *rvec*, const Mat& *tvec*, const Mat& *cameraMatrix*, const Mat& *distCoeffs*, vector<Point2f>& *imagePoints*)

void **projectPoints**(const Mat& *objectPoints*, const Mat& *rvec*, const Mat& *tvec*, const Mat& *cameraMatrix*, const Mat& *distCoeffs*, vector<Point2f>& *imagePoints*, Mat& *dpdrot*, Mat& *dpd*, Mat& *dpdf*, Mat& *dpdc*, Mat& *dpddist*, double *aspectRatio*=0)

Project 3D points on to an image plane.

- Parameters:**
- *objectPoints* – The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel (or vector<Point3f>), where N is the number of points in the view
  - *rvec* – The rotation vector, see [Rodrigues](#)
  - *tvec* – The translation vector

- *cameraMatrix* – The camera matrix  $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

- *distCoeffs* – The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients ( $k_1, k_2, p_1, p_2, k_3$ ). If it is empty, all of the distortion coefficients are considered 0's
- *imagePoints* – The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel (or vector<Point2f>)
- *dpdrot* – Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector
- *dpd* – Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector
- *dpdf* – Optional 2Nx2 matrix of derivatives of image points with respect to  $f_x$  and  $f_y$
- *dpdc* – Optional 2Nx2 matrix of derivatives of image points with respect to  $c_x$  and  $c_y$
- *dpddist* – Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in [calibrateCamera](#), [solvePnP](#) and [stereoCalibrate](#). The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can

compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

## reprojectImageTo3D

---

void **reprojectImageTo3D**(const Mat& *disparity*, Mat& *\_3dImage*, const Mat& *Q*, bool *handleMissingValues=false*)

Reprojects disparity image to 3D space.

- Parameters:**
- *disparity* – The input single-channel 16-bit signed or 32-bit floating-point disparity image
  - *\_3dImage* – The output 3-channel floating-point image of the same size as *disparity*. Each element of *\_3dImage* (*x*, *y*) will contain the 3D coordinates of the point (*x*, *y*), computed from the disparity map.
  - *Q* – The  $4 \times 4$  perspective transformation matrix that can be obtained with [stereoRectify](#)
  - *handleMissingValues* – If true, when the pixels with the minimal disparity (that corresponds to the outliers; see [StereoBM](#)) 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  $d = \text{disparity}(x, y)$  it computes:

$$\begin{aligned} [X \ Y \ Z \ W]^T &= Q * [x \ y \ \text{disparity}(x, y) \ 1]^T \\ \text{3dImage}(x, y) &= (X/W, Y/W, Z/W) \end{aligned}$$

The matrix *Q* can be arbitrary  $4 \times 4$  matrix, e.g. the one computed by [stereoRectify](#). To reproject a sparse set of points `bggroup({(x,y,d),...})` to 3D space, use [perspectiveTransform](#).

## RQDecomp3x3

---

void **RQDecomp3x3**(const Mat& *M*, Mat& *R*, Mat& *Q*)

Vec3d **RQDecomp3x3**(const Mat& *M*, Mat& *R*, Mat& *Q*, Mat& *Qx*, Mat& *Qy*, Mat& *Qz*)

Computes the 'RQ' decomposition of 3x3 matrices.

- Parameters:**
- $M$  – The 3x3 input matrix
  - $R$  – The output 3x3 upper-triangular matrix
  - $Q$  – The output 3x3 orthogonal matrix
  - $Q_x$  – Optional 3x3 rotation matrix around x-axis
  - $Q_y$  – Optional 3x3 rotation matrix around y-axis
  - $Q_z$  – Optional 3x3 rotation matrix around z-axis

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 (as the return value) that could be used in OpenGL.

## Rodrigues

---

void **Rodrigues**(const Mat& *src*, Mat& *dst*)

void **Rodrigues**(const Mat& *src*, Mat& *dst*, Mat& *jacobian*)

Converts a rotation matrix to a rotation vector or vice versa.

- Parameters:**
- *src* – The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)
  - *dst* – The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively
  - *jacobian* – Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow \text{norm}(r)$$

$$r \leftarrow r/\theta$$

$$R = \cos \theta I + (1 - \cos \theta) r r^T + \sin \theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

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

[calibrateCamera](#), [stereoCalibrate](#) or [solvePnP](#).

## StereoBM

The class for computing stereo correspondence using block matching algorithm.

---

```
// Block matching stereo correspondence algorithm\par
class StereoBM
{
    enum { NORMALIZED_RESPONSE = CV_STEREO_BM_NORMALIZED_RESP
          BASIC_PRESET=CV_STEREO_BM_BASIC,
          FISH_EYE_PRESET=CV_STEREO_BM_FISH_EYE,
          NARROW_PRESET=CV_STEREO_BM_NARROW };

    StereoBM();
    // the preset is one of ..._PRESET above.
    // ndisparities is the size of disparity range,
    // in which the optimal disparity at each pixel is search
    // SADWindowSize is the size of averaging window used to
    // (larger values mean better robustness to noise, but
    StereoBM(int preset, int ndisparities=0, int SADWindowSize
    // separate initialization function
    void init(int preset, int ndisparities=0, int SADWindowSize
    // computes the disparity for the two rectified 8-bit sin
    // the disparity will be 16-bit signed image of the same
    void operator()( const Mat& left, const Mat& right, Mat&

    Ptr<CvStereoBMState> state;
};
```

---

## stereoCalibrate

```
double stereoCalibrate(const vector<vector<Point3f> >&
objectPoints, const vector<vector<Point2f> >& imagePoints1, const
vector<vector<Point2f> >& imagePoints2, Mat& cameraMatrix1, Mat& distCoeffs1,
Mat& cameraMatrix2, Mat& distCoeffs2, Size imageSize, Mat& R, Mat& T, Mat& E,
Mat& F, TermCriteria criteria = TermCriteria
(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6), int
flags=CALIB_FIX_INTRINSIC)
```

Calibrates stereo camera.

**Parameters:**

- *objectPoints* – The vector of vectors of points on the calibration pattern in its coordinate system, one vector per view. If the the same calibration pattern is shown in each view and it's fully visible then all the vectors will be the same,

although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0

- *imagePoints1* – The vector of vectors of the object point projections on the calibration pattern views from the 1st camera, one vector per a view. The projections must be in the same order as the corresponding object points.
- *imagePoints2* – The vector of vectors of the object point projections on the calibration pattern views from the 2nd camera, one vector per a view. The projections must be in the same order as the corresponding object points.
- *cameraMatrix1*, *cameraMatrix2* – The input/output first and

second camera matrices, respectively: 
$$\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix},$$
  
 $j = 0, 1.$  If any of

**CV\_CALIB\_USE\_INTRINSIC\_GUESS**,

**CV\_CALIB\_FIX\_ASPECT\_RATIO**,

**CV\_CALIB\_FIX\_INTRINSIC**

or

**CV\_CALIB\_FIX\_FOCAL\_LENGTH** are specified, some or all of the matrices' components must be initialized; see the flags description

- *distCoeffs1*, *distCoeffs2* – The input/output lens distortion coefficients for the first and the second cameras, 4x1, 5x1, 1x4 or 1x5 floating-point vectors  $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}, k_3^{(j)})$ ,  $j = 0, 1.$  If any of **CV\_CALIB\_FIX\_K1**, **CV\_CALIB\_FIX\_K2** or **CV\_CALIB\_FIX\_K3** is specified, then the corresponding elements of the distortion coefficients must be initialized.
- *imageSize* – Size of the image, used only to initialize intrinsic camera matrix.
- *R* – The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.
- *T* – The output translation vector between the cameras' coordinate systems.
- *E* – The output essential matrix.
- *F* – The output fundamental matrix.
- *criteria* – The termination criteria for the iterative optimization algorithm.
- *flags* – Different flags, may be 0 or combination of the following values:

- **CV\_CALIB\_FIX\_INTRINSIC** - If it is set, `cameraMatrix?`, as well as `distCoeffs?` are fixed, so that only  $R$ ,  $T$ ,  $E$  and  $F$  are estimated.
- **CV\_CALIB\_USE\_INTRINSIC\_GUESS** - The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.
- **CV\_CALIB\_FIX\_PRINCIPAL\_POINT** - The principal points are fixed during the optimization.
- **CV\_CALIB\_FIX\_FOCAL\_LENGTH** -  $f_x^{(j)}$  and  $f_y^{(j)}$  are fixed.
- **CV\_CALIB\_FIX\_ASPECT\_RATIO** -  $f_y^{(j)}$  is optimized, but the ratio  $f_x^{(j)} / f_y^{(j)}$  is fixed.
- **CV\_CALIB\_FIX\_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\_K2**, **CV\_CALIB\_FIX\_K3** - Fixes the corresponding radial distortion coefficient (the coefficient must be passed to the function)

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientatation of the 2 cameras is fixed, and if we computed poses of an object relative to the first camera and to the second camera,  $(R_1, T_1)$  and  $(R_2, T_2)$ , respectively (that can be done with [solvePnP](#)), 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 [calibrateCamera](#)), 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 [calibrateCamera](#), the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

## stereoRectify

```
void stereoRectify(const Mat& cameraMatrix1, const Mat&
distCoeffs1, const Mat& cameraMatrix2, const Mat& distCoeffs2, Size imageSize,
const Mat& R, const Mat& T, Mat& R1, Mat& R2, Mat& P1, Mat& P2, Mat& Q, int
flags=CV_CALIB_ZERO_DISPARIITY)
```

```
void stereoRectify(const Mat& cameraMatrix1, const Mat&
distCoeffs1, const Mat& cameraMatrix2, const Mat& distCoeffs2, Size imageSize,
const Mat& R, const Mat& T, Mat& R1, Mat& R2, Mat& P1, Mat& P2, Mat& Q, double
alpha, Size newImageSize=Size(), Rect* roi1=0, Rect* roi2=0, int
flags=CV_CALIB_ZERO_DISPARIITY)
```

Computes rectification transforms for each head of a calibrated stereo camera.

**Parameters:**

- *cameraMatrix1*, *cameraMatrix2* – The camera matrices

$$\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$$

- *distCoeffs1*, *distCoeffs2* – The input distortion coefficients for each camera,  $k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}, k_3^{(j)}$
- *imageSize* – Size of the image used for stereo calibration.
- *R* – The rotation matrix between the 1st and the 2nd cameras' coordinate systems.
- *T* – The translation vector between the cameras' coordinate systems.
- *R1*, *R2* – The output  $3 \times 3$  rectification transforms (rotation matrices) for the first and the second cameras, respectively.
- *P1*, *P2* – The output  $3 \times 4$  projection matrices in the new (rectified) coordinate systems.
- *Q* – The output  $4 \times 4$  disparity-to-depth mapping matrix, see

[reprojectImageTo3D](#).

- *flags* – The operation flags; may be 0 or **CV\_CALIB\_ZERO\_DISPARITY**. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.
- *alpha* – The free scaling parameter. If it is -1 or absent, the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.
- *newImageSize* – The new image resolution after rectification. The same size should be passed to [initUndistortRectifyMap](#), see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.
- *roi1, roi2* – 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:

- 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:

$$P1 = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$P2 = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where  $T_x$  is horizontal shift between the cameras and  $cx_1 = cx_2$  if **CV\_CALIB\_ZERO\_DISPARITY** is set.

- 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. P1 and P2 will look as:

$$P1 = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$P2 = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

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

bool **stereoRectifyUncalibrated**(const Mat& *points1*, const Mat& *points2*, const Mat& *F*, Size *imgSize*, Mat& *H1*, Mat& *H2*, double *threshold*=5)

Computes rectification transform for uncalibrated stereo camera.

- Parameters:**
- *points1*, *points2* – The 2 arrays of corresponding 2D points. The same formats as in [findFundamentalMat](#) are supported
  - *F* – The input fundamental matrix. It can be computed from the same set of point pairs using [findFundamentalMat](#).
  - *imgSize* – Size of the image.
  - *H1*, *H2* – The output rectification homography matrices for the first and for the second images.
  - *threshold* – The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which  $|\text{points2}[i]^T * F * \text{points1}[i]| > \text{threshold}$ ) 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  $H_1$  and  $H_2$ . The function implements the algorithm .

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 [calibrateCamera](#) and then the images can be corrected using [undistort](#), or just the point coordinates can be corrected with [undistortPoints](#).

## undistort

---

void **undistort**(const Mat& *src*, Mat& *dst*, const Mat& *cameraMatrix*, const Mat& *distCoeffs*, const Mat& *newCameraMatrix*=Mat())

Transforms an image to compensate for lens distortion.

- Parameters:**
- *src* – The input (distorted) image
  - *dst* – The output (corrected) image; will have the same size and the same type as *src*
  - *cameraMatrix* – The input camera matrix  $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
  - *distCoeffs* – The vector of distortion coefficients,  $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}, [k_3^{(j)}])$
  - *newCameraMatrix* – Camera matrix of the distorted image. By default it is the same as *cameraMatrix*, but you may additionally scale and shift the result by using some different matrix

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 [calibrateCamera](#). If the resolution of images is different from the used at the calibration stage,  $f_x$ ,  $f_y$ ,  $c_x$  and  $c_y$  need to be scaled accordingly, while the distortion coefficients remain the same.

## undistortPoints

---

```
void undistortPoints(const Mat& src, vector<Point2f>& dst, const
Mat& cameraMatrix, const Mat& distCoeffs, const Mat& R=Mat(), const Mat& P=Mat
())
```

```
void undistortPoints(const Mat& src, Mat& dst, const Mat&
cameraMatrix, const Mat& distCoeffs, const Mat& R=Mat(), const Mat& P=Mat())
```

Computes the ideal point coordinates from the observed point coordinates.

- Parameters:**
- *src* – The observed point coordinates, same format as `imagePoints` in [projectPoints](#)
  - *dst* – The output ideal point coordinates, after undistortion and reverse perspective transformation.

- *cameraMatrix* – The camera matrix 
$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$
- *distCoeffs* – The vector of distortion coefficients,  $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}, [k_3^{(j)}])$
- *R* – The rectification transformation in object space (3x3 matrix). R1 or R2, computed by [stereoRectify](#) can be passed here. If the matrix is empty, the identity transformation is used
- *P* – The new camera matrix (3x3) or the new projection matrix (3x4). P1 or P2, computed by [stereoRectify](#) can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to [undistort](#) 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 [projectPoints](#) (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 empty).

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# highgui. High-level GUI and Media IO

---

- [User Interface](#)
  - [createTrackbar](#)
  - [getTrackbarPos](#)
  - [imshow](#)
  - [namedWindow](#)
  - [setTrackbarPos](#)
  - [waitKey](#)
- [Reading and Writing Images and Video](#)
  - [imdecode](#)
  - [imencode](#)
  - [imread](#)
  - [imwrite](#)
  - [VideoCapture](#)
  - [VideoWriter](#)

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# User Interface

---

## createTrackbar

---

int **createTrackbar**(const string& *trackbarname*, const string& *winname*, int\* *value*, int *count*, TrackbarCallback *onChange* CV\_DEFAULT(0), void\* *userdata* CV\_DEFAULT(0))

Creates a trackbar and attaches it to the specified window

- Parameters:**
- *trackbarname* – Name of the created trackbar.
  - *winname* – Name of the window which will be used as a parent of the created trackbar.
  - *value* – The optional pointer to an integer variable, whose value will reflect the position of the slider. Upon creation, the slider position is defined by this variable.
  - *count* – The maximal position of the slider. The minimal position is always 0.
  - *onChange* – Pointer to the function to be called every time the slider changes position. This function should be prototyped as `void Foo(int, void*) ;`, where the first parameter is the trackbar position and the second parameter is the user data (see the next parameter). If the callback is NULL pointer, then no callbacks is called, but only *value* is updated
  - *userdata* – The user data that is passed as-is to the callback; it can be used to handle trackbar events without using global variables

The function `createTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable *value* to be synchronized with trackbar position and specifies a callback function *onChange* to be called on the trackbar position change. The created trackbar is displayed on the top of the given window.

## getTrackbarPos

---

int **getTrackbarPos**(const string& *trackbarname*, const string& *winname*)

Returns the trackbar position.

- Parameters:**
- *trackbarname* – Name of the trackbar.
  - *winname* – Name of the window which is the parent of the trackbar.

The function returns the current position of the specified trackbar.

## imshow

---

void **imshow**(const string& *winname*, const Mat& *image*)

Displays the image in the specified window

- Parameters:**
- *winname* – Name of the window.
  - *image* – Image to be shown.

The function `imshow` 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  $[0, 255 \times 256]$  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  $[0, 255]$ .

## namedWindow

---

void **namedWindow**(const string& *winname*, int *flags*)

Creates a window.

- Parameters:**
- *name* – Name of the window in the window caption that may be used as a window identifier.
  - *flags* – Flags of the window. Currently the only supported flag is **CV\_WINDOW\_AUTOSIZE**. If this is set, the window size is automatically adjusted to fit the displayed image (see [imshow](#)), and the user can not change the window size manually.

The function `namedWindow` 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.

## setTrackbarPos

---

void **setTrackbarPos**(const string& *trackbarname*, const string& *winname*, int *pos*)

Sets the trackbar position.

- Parameters:**
- *trackbarname* – Name of the trackbar.
  - *winname* – Name of the window which is the parent of trackbar.
  - *pos* – The new position.

The function sets the position of the specified trackbar in the specified window.

## waitKey

---

int **waitKey**(int *delay=0*)

Waits for a pressed key.

- Parameter:** *delay* – Delay in milliseconds. 0 is the special value that means “forever”

The function `waitKey` waits for key event infinitely (when **`delay ≤ 0`**) or for `delay` milliseconds, when it's positive. 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.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Reading and Writing Images and Video

---

## imdecode

---

Mat **imdecode**(const Mat& *buf*, int *flags*)

- Parameters:**
- *buf* – The input array of vector of bytes
  - *flags* – The same flags as in [imread](#)

The function reads image from the specified buffer in memory. If the buffer is too short or contains invalid data, the empty matrix will be returned.

See [imread](#) for the list of supported formats and the flags description.

## imencode

---

bool **imencode**(const string& *ext*, const Mat& *img*, vector<uchar>& *buf*, const vector<int>& *params*=vector<int>())

- Parameters:**
- *ext* – The file extension that defines the output format
  - *img* – The image to be written
  - *buf* – The output buffer; resized to fit the compressed image
  - *params* – The format-specific parameters; see [imwrite](#)

The function compresses the image and stores it in the memory buffer, which is resized to fit the result. See [imwrite](#) for the list of supported formats and the flags description.

## imread

---

Mat **imread**(const string& *filename*, int *flags*=1)

Loads an image from a file.

- Parameters:**
- *filename* – Name of file to be loaded.
  - *flags* – Specifies color type of the loaded image:
    - $>0$  – the loaded image is forced to be a 3-channel color image
    - $=0$  – the loaded image is forced to be grayscale
    - $<0$  – the loaded image will be loaded as-is (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 if  $flags \geq 0$ ).

The function `imread` loads an image from the specified file and returns it. If the

image can not be read (because of missing file, improper permissions, unsupported or invalid format), the function returns empty matrix (`Mat::data==NULL`). Currently, the following file formats are supported:

- Windows bitmaps - `*.bmp`, `*.dib` (always supported)
- JPEG files - `*.jpeg`, `*.jpg`, `*.jpe` (see Note2)
- JPEG 2000 files - `*.jp2` (see Note2)
- Portable Network Graphics - `*.png` (see Note2)
- Portable image format - `*.pbm`, `*.pgm`, `*.ppm` (always supported)
- Sun rasters - `*.sr`, `*.ras` (always supported)
- TIFF files - `*.tiff`, `*.tif` (see Note2)

Note1: The function determines type of the image by the content, not by the file extension.

Note2: On Windows and MacOSX the shipped with OpenCV image codecs (libjpeg, libpng, libtiff and libjasper) are used by default; so OpenCV can always read JPEGs, PNGs and TIFFs. On MacOSX there is also the option to use native MacOSX image readers. But beware that currently these native image loaders give images with somewhat different pixel values, because of the embedded into MacOSX color management.

On Linux, BSD flavors and other Unix-like open-source operating systems OpenCV looks for the supplied with OS image codecs. Please, install the relevant packages (do not forget the development files, e.g. “libjpeg-dev” etc. in Debian and Ubuntu) in order to get the codec support, or turn on `OPENCV_BUILD_3RDPARTY_LIBS` flag in CMake.

## imwrite

---

```
bool imwrite(const string& filename, const Mat& img, const vector<int>&
params=vector<int>())
```

Saves an image to a specified file.

- Parameters:**
- *filename* – Name of the file.
  - *img* – The image to be saved.
  - *params* –  
The format-specific save parameters, encoded as pairs `paramId_1, paramValue_1, paramId_2, paramValue_2, ...`. The following parameters are currently supported:
    - In the case of JPEG it can be a quality (`CV_IMWRITE_JPEG_QUALITY`), from 0 to 100 (the higher is the better), 95 by default.

- In the case of PNG it can be the compression level (**CV\_IMWRITE\_PNG\_COMPRESSION**), from 0 to 9 (the higher value means smaller size and longer compression time), 3 by default.
- In the case of PPM, PGM or PBM it can a binary format flag (**CV\_IMWRITE\_PXM\_BINARY**), 0 or 1, 1 by default.

The function `imwrite` saves the image to the specified file. The image format is chosen based on the filename extension, see [imread](#) for the list of extensions. Only 8-bit (or 16-bit in the case of PNG, JPEG 2000 and TIFF) 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 `Mat::convertTo`, and `cvtColor` to convert it before saving, or use the universal XML I/O functions to save the image to XML or YAML format.

## VideoCapture

---

Class for video capturing from video files or cameras

```
class VideoCapture
{
public:
    // the default constructor
    VideoCapture();
    // the constructor that opens video file
    VideoCapture(const string& filename);
    // the constructor that starts streaming from the camera
    VideoCapture(int device);

    // the destructor
    virtual ~VideoCapture();

    // opens the specified video file
    virtual bool open(const string& filename);

    // starts streaming from the specified camera by its id
    virtual bool open(int device);

    // returns true if the file was open successfully or if t
    // has been initialized succesfully
    virtual bool isOpened() const;

    // closes the camera stream or the video file
    // (automatically called by the destructor)
    virtual void release();

    // grab the next frame or a set of frames from a multi-he
```

```

    // returns false if there are no more frames
    virtual bool grab();
    // reads the frame from the specified video stream
    // (non-zero channel is only valid for multi-head camera live st.
    virtual bool retrieve(Mat& image, int channel=0);
    // equivalent to grab() + retrieve(image, 0);
    virtual VideoCapture& operator >> (Mat& image);

    // sets the specified property propId to the specified value
    virtual bool set(int propId, double value);
    // retrieves value of the specified property
    virtual double get(int propId);

protected:
    ...
};

```

---

The class provides C++ video capturing API. Here is how the class can be used:

```

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

using namespace cv;

int main(int, char**)
{
    VideoCapture cap(0); // open the default camera
    if(!cap.isOpened()) // check if we succeeded
        return -1;

    Mat edges;
    namedWindow("edges",1);
    for(;;)
    {
        Mat frame;
        cap >> frame; // get a new frame from camera
        cvtColor(frame, edges, CV_BGR2GRAY);
        GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
        Canny(edges, edges, 0, 30, 3);
        imshow("edges", edges);
        if(waitKey(30) >= 0) break;
    }
    // the camera will be deinitialized automatically in Vide
    return 0;
}

```

---

## VideoWriter

Video writer class

---

```
class VideoWriter
{
public:
    // default constructor
    VideoWriter();
    // constructor that calls open
    VideoWriter(const string& filename, int fourcc,
               double fps, Size frameSize, bool isColor=true

    // the destructor
    virtual ~VideoWriter();

    // opens the file and initializes the video writer.
    // filename - the output file name.
    // fourcc - the codec
    // fps - the number of frames per second
    // frameSize - the video frame size
    // isColor - specifies whether the video stream is color
    virtual bool open(const string& filename, int fourcc,
                    double fps, Size frameSize, bool isColor

    // returns true if the writer has been initialized succes
    virtual bool isOpened() const;

    // writes the next video frame to the stream
    virtual VideoWriter& operator << (const Mat& image);

protected:
    ...
};
```

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# ml. Machine Learning

---

- **Statistical Models**
  - CvStatModel
  - CvStatModel::CvStatModel
  - CvStatModel::CvStatModel(...)
  - CvStatModel::CvStatModel
  - CvStatModel::clear
  - CvStatModel::save
  - CvStatModel::load
  - CvStatModel::write
  - CvStatModel::read
  - CvStatModel::train
  - CvStatModel::predict
- **Normal Bayes Classifier**
  - CvNormalBayesClassifier
  - CvNormalBayesClassifier::train
  - CvNormalBayesClassifier::predict
- **K Nearest Neighbors**
  - CvKNearest
  - CvKNearest::train
  - CvKNearest::find\_nearest
  - Example. Classification of 2D samples from a Gaussian mixture with the k-nearest classifier
- **Support Vector Machines**
  - CvSVM
  - CvSVMParams
  - CvSVM::train
  - CvSVM::train\_auto
  - CvSVM::get\_default\_grid
  - CvSVM::get\_params
  - CvSVM::get\_support\_vector\*
- **Decision Trees**
  - Predicting with Decision Trees
  - Training Decision Trees
  - Variable importance
  - CvDTreeSplit
  - CvDTreeNode
  - CvDTreeParams
  - CvDTreeTrainData
  - CvDTree
  - CvDTree::train
  - CvDTree::predict
- **Boosting**
  - CvBoostParams

- [CvBoostTree](#)
- [CvBoost](#)
- [CvBoost::train](#)
- [CvBoost::predict](#)
- [CvBoost::prune](#)
- [CvBoost::get\\_weak\\_predictors](#)
- [Random Trees](#)
  - [CvRTParams](#)
  - [CvRTrees](#)
  - [CvRTrees::train](#)
  - [CvRTrees::predict](#)
  - [CvRTrees::get\\_var\\_importance](#)
  - [CvRTrees::get\\_proximity](#)
- [Expectation-Maximization](#)
  - [CvEMParams](#)
  - [CvEM](#)
  - [CvEM::train](#)
- [Neural Networks](#)
  - [CvANN\\_MLP\\_TrainParams](#)
  - [CvANN\\_MLP](#)
  - [CvANN\\_MLP::create](#)
  - [CvANN\\_MLP::train](#)

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different sets of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class 'CvStatModel' that all the other ML classes are derived from.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Statistical Models

---

## CvStatModel

---

Base class for the statistical models in ML.

```
class CvStatModel
{
public:
    /* CvStatModel(); */
    /* CvStatModel( const CvMat* train_data ... ); */

    virtual ~CvStatModel();

    virtual void clear()=0;

    /* virtual bool train( const CvMat* train_data, [int tfla
CvMat* responses, ...,
    [const CvMat* var_idx,] ..., [const CvMat* sample_idx,]
    [const CvMat* var_type,] ..., [const CvMat* missing_mask
<misc_training_alg_params> ... ]=0;
    */

    /* virtual float predict( const CvMat* sample ... ) const

    virtual void save( const char* filename, const char* name
    virtual void load( const char* filename, const char* name

    virtual void write( CvFileStorage* storage, const char* n
    virtual void read( CvFileStorage* storage, CvFileNode* no
};
```

In this declaration some methods are commented off. Actually, these are methods for which there is no unified API (with the exception of the default constructor), however, there are many similarities in the syntax and semantics that are briefly described below in this section, as if they are a part of the base class.

## CvStatModel::CvStatModel

---

`CvStatModel::CvStatModel()`

Default constructor.

Each statistical model class in ML has a default constructor without parameters. This constructor is useful for 2-stage model construction, when the default constructor is followed by `train()` or `load()`.

## CvStatModel::CvStatModel(...)

---

`CvStatModel::CvStatModel`(const `CvMat*` train\_data ...)

Training constructor.

Most ML classes provide single-step construct and train constructors. This constructor is equivalent to the default constructor, followed by the `train()` method with the parameters that are passed to the constructor.

## CvStatModel::~CvStatModel

---

`CvStatModel::~CvStatModel`()

Virtual destructor.

The destructor of the base class is declared as virtual, so it is safe to write the following code:

---

```
CvStatModel* model;
if( use\_svm )
    model = new CvSVM(... /* SVM params */);
else
    model = new CvDTree(... /* Decision tree params */);
...
delete model;
```

---

Normally, the destructor of each derived class does nothing, but in this instance it calls the overridden method `clear()` that deallocates all the memory.

## CvStatModel::clear

---

`void CvStatModel::clear`()

Deallocates memory and resets the model state.

The method `clear` does the same job as the destructor; it deallocates all the memory occupied by the class members. But the object itself is not destructed, and can be reused further. This method is called from the destructor, from the `train` methods of the derived classes, from the methods `load()`, `read()` or even explicitly by the user.

## CvStatModel::save

---

```
void CvStatModel::save(const char* filename, const char* name=0)
```

Saves the model to a file.

The method `save` stores the complete model state to the specified XML or YAML file with the specified name or default name (that depends on the particular class). Data persistence functionality from CxCore is used.

## CvStatModel::load

---

```
void CvStatModel::load(const char* filename, const char* name=0)
```

Loads the model from a file.

The method `load` loads the complete model state with the specified name (or default model-dependent name) from the specified XML or YAML file. The previous model state is cleared by `clear()`.

Note that the method is virtual, so any model can be loaded using this virtual method. However, unlike the C types of OpenCV that can be loaded using the generic `crossbgroup({cvLoad})`, here the model type must be known, because an empty model must be constructed beforehand. This limitation will be removed in the later ML versions.

## CvStatModel::write

---

```
void CvStatModel::write(CvFileStorage* storage, const char* name)
```

Writes the model to file storage.

The method `write` stores the complete model state to the file storage with the specified name or default name (that depends on the particular class). The method is called by `save()`.

## CvStatModel::read

---

```
void CvStatModel::read(CvFileStorage* storage, CvFileNode* node)
```

Reads the model from file storage.

The method `read` restores the complete model state from the specified node of the file storage. The node must be located by the user using the function `GetFileNodeByName`.

The previous model state is cleared by `clear()`.

## CvStatModel::train

```
bool CvStatModel::train(const CvMat* train_data, [int tflag, ] ..., const
CvMat* responses, ..., [const CvMat* var_idx, ] ..., [const CvMat* sample_idx, ] ...
[const CvMat* var_type, ] ..., [const CvMat* missing_mask, ]
<misc_training_alg_params> ...)
```

Trains the model.

The method trains the statistical model using a set of input feature vectors and the corresponding output values (responses). Both input and output vectors/values are passed as matrices. By default the input feature vectors are stored as `train_data` rows, i.e. all the components (features) of a training vector are stored continuously. However, some algorithms can handle the transposed representation, when all values of each particular feature (component/input variable) over the whole input set are stored continuously. If both layouts are supported, the method includes `tflag` parameter that specifies the orientation:

- `tflag=CV_ROW_SAMPLE` means that the feature vectors are stored as rows,
- `tflag=CV_COL_SAMPLE` means that the feature vectors are stored as columns.

The `train_data` must have a **CV\_32FC1** (32-bit floating-point, single-channel) format. Responses are usually stored in the 1d vector (a row or a column) of **CV\_32SC1** (only in the classification problem) or **CV\_32FC1** format, one value per input vector (although some algorithms, like various flavors of neural nets, take vector responses).

For classification problems the responses are discrete class labels; for regression problems the responses are values of the function to be approximated. Some algorithms can deal only with classification problems, some - only with regression problems, and some can deal with both problems. In the latter case the type of output variable is either passed as separate parameter, or as a last element of `var_type` vector:

- **CV\_VAR\_CATEGORICAL** means that the output values are discrete class labels,
- **CV\_VAR\_ORDERED (=CV\_VAR\_NUMERICAL)** means that the

output values are ordered, i.e. 2 different values can be compared as numbers, and this is a regression problem

The types of input variables can be also specified using `var_type`. Most algorithms can handle only ordered input variables.

Many models in the ML may be trained on a selected feature subset, and/or on a selected sample subset of the training set. To make it easier for the user, the method `train` usually includes `var_idx` and `sample_idx` parameters. The former identifies variables (features) of interest, and the latter identifies samples of interest. Both vectors are either integer (**CV\_32SC1**) vectors, i.e. lists of 0-based indices, or 8-bit (**CV\_8UC1**) masks of active variables/samples. The user may pass `NULL` pointers instead of either of the arguments, meaning that all of the variables/samples are used for training.

Additionally some algorithms can handle missing measurements, that is when certain features of certain training samples have unknown values (for example, they forgot to measure a temperature of patient A on Monday). The parameter `missing_mask`, an 8-bit matrix the same size as `train_data`, is used to mark the missed values (non-zero elements of the mask).

Usually, the previous model state is cleared by `clear()` before running the training procedure. However, some algorithms may optionally update the model state with the new training data, instead of resetting it.

## CvStatModel::predict

---

```
float CvStatModel::predict(const CvMat* sample[,  
<prediction_params>]) const
```

Predicts the response for the sample.

The method is used to predict the response for a new sample. In the case of classification the method returns the class label, in the case of regression - the output function value. The input sample must have as many components as the `train_data` passed to `train` contains. If the `var_idx` parameter is passed to `train`, it is remembered and then is used to extract only the necessary components from the input sample in the method `predict`.

The suffix “const” means that prediction does not affect the internal model state, so the method can be safely called from within different threads.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Normal Bayes Classifier

---

This is a simple classification model assuming that feature vectors from each class are normally distributed (though, not necessarily independently distributed), so the whole data distribution function is assumed to be a Gaussian mixture, one component per class. Using the training data the algorithm estimates mean vectors and covariance matrices for every class, and then it uses them for prediction.

[Fukunaga90] K. Fukunaga. Introduction to Statistical Pattern Recognition. second ed., New York: Academic Press, 1990.

## CvNormalBayesClassifier

---

Bayes classifier for normally distributed data.

```
class CvNormalBayesClassifier : public CvStatModel
{
public:
    CvNormalBayesClassifier();
    virtual ~CvNormalBayesClassifier();

    CvNormalBayesClassifier( const CvMat* _train_data, const
        const CvMat* _var_idx=0, const CvMat* _sample_idx=0 )

    virtual bool train( const CvMat* _train_data, const CvMat
        const CvMat* _var_idx = 0, const CvMat* _sample_idx=0

    virtual float predict( const CvMat* _samples, CvMat* resu
    virtual void clear();

    virtual void save( const char* filename, const char* name
    virtual void load( const char* filename, const char* name

    virtual void write( CvFileStorage* storage, const char* n
    virtual void read( CvFileStorage* storage, CvFileNode* no
protected:
    ...
};
```

---

## CvNormalBayesClassifier::train

---

```
bool CvNormalBayesClassifier::train(const CvMat* _train_data,
const CvMat* _responses, const CvMat* _var_idx =0, const CvMat* _sample_idx=0,
bool update=false)
```

Trains the model.

The method trains the Normal Bayes classifier. It follows the conventions of the generic `train` “method” with the following limitations: only `CV_ROW_SAMPLE` data layout is supported; the input variables are all ordered; the output variable is categorical (i.e. elements of `_responses` must be integer numbers, though the vector may have `CV_32FC1` type), and missing measurements are not supported.

In addition, there is an `update` flag that identifies whether the model should be trained from scratch (`update=false`) or should be updated using the new training data (`update=true`).

## CvNormalBayesClassifier::predict

---

```
float CvNormalBayesClassifier::predict(const CvMat* samples,  
CvMat* results=0) const
```

Predicts the response for sample(s)

The method `predict` estimates the most probable classes for the input vectors. The input vectors (one or more) are stored as rows of the matrix `samples`. In the case of multiple input vectors, there should be one output vector `results`. The predicted class for a single input vector is returned by the method.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# K Nearest Neighbors

---

The algorithm caches all of the training samples, and predicts the response for a new sample by analyzing a certain number (K) of the nearest neighbors of the sample (using voting, calculating weighted sum etc.) The method is sometimes referred to as “learning by example”, because for prediction it looks for the feature vector with a known response that is closest to the given vector.

## CvKNearest

---

K Nearest Neighbors model.

```
class CvKNearest : public CvStatModel
{
public:

    CvKNearest();
    virtual ~CvKNearest();

    CvKNearest( const CvMat* _train_data, const CvMat* _respo
                const CvMat* _sample_idx=0, bool _is_regressi

    virtual bool train( const CvMat* _train_data, const CvMat
                        const CvMat* _sample_idx=0, bool is_r
                        int _max_k=32, bool _update_base=false

    virtual float find_nearest( const CvMat* _samples, int k,
                                const float** neighbors=0, CvMat* neighbor_responses=

    virtual void clear();
    int get_max_k() const;
    int get_var_count() const;
    int get_sample_count() const;
    bool is_regression() const;

protected:
    ...
};
```

---

## CvKNearest::train

---

```
bool CvKNearest::train(const CvMat* _train_data, const CvMat*
    _responses, const CvMat* _sample_idx=0, bool is_regression=false, int _max_k=32,
    bool _update_base=false)
```

Trains the model.

The method trains the K-Nearest model. It follows the conventions of generic `train` “method” with the following limitations: only `CV_ROW_SAMPLE` data layout is supported, the input variables are all ordered, the output variables can be either categorical (`is_regression=false`) or ordered (`is_regression=true`), variable subsets (`var_idx`) and missing measurements are not supported.

The parameter `_max_k` specifies the number of maximum neighbors that may be passed to the method `find_nearest`.

The parameter `_update_base` specifies whether the model is trained from scratch (`_update_base=false`), or it is updated using the new training data (`_update_base=true`). In the latter case the parameter `_max_k` must not be larger than the original value.

## CvKNearest::find\_nearest

---

```
float CvKNearest::find_nearest(const CvMat* _samples, int k,
CvMat* results=0, const float** neighbors=0, CvMat* neighbor_responses=0, CvMat*
dist=0) const
```

Finds the neighbors for the input vectors.

For each input vector (which are the rows of the matrix `_samples`) the method finds the  $k \leq \text{get\_max\_k}()$  nearest neighbor. In the case of regression, the predicted result will be a mean value of the particular vector’s neighbor responses. In the case of classification the class is determined by voting.

For custom classification/regression prediction, the method can optionally return pointers to the neighbor vectors themselves (`neighbors`, an array of `k*_samples->rows` pointers), their corresponding output values (`neighbor_responses`, a vector of `k*_samples->rows` elements) and the distances from the input vectors to the neighbors (`dist`, also a vector of `k*_samples->rows` elements).

For each input vector the neighbors are sorted by their distances to the vector.

If only a single input vector is passed, all output matrices are optional and the predicted value is returned by the method.

## Example. Classification of 2D samples from a Gaussian mixture with the k-nearest classifier

---

```

#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    const int K = 10;
    int i, j, k, accuracy;
    float response;
    int train_sample_count = 100;
    CvRNG rng_state = cvRNG(-1);
    CvMat* trainData = cvCreateMat( train_sample_count, 2, CV_32FC1 );
    CvMat* trainClasses = cvCreateMat( train_sample_count, 1, CV_32FC1 );
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    float _sample[2];
    CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
    cvZero( img );

    CvMat trainData1, trainData2, trainClasses1, trainClasses2;

    // form the training samples
    cvGetRows( trainData, &trainData1, 0, train_sample_count/2,
               cvRandArr( &rng_state, &trainData1, CV_RAND_NORMAL, cvScalar(0) ) );

    cvGetRows( trainData, &trainData2, train_sample_count/2,
               cvRandArr( &rng_state, &trainData2, CV_RAND_NORMAL, cvScalar(0) ) );

    cvGetRows( trainClasses, &trainClasses1, 0, train_sample_count/2,
               cvSet( &trainClasses1, cvScalar(1) ) );

    cvGetRows( trainClasses, &trainClasses2, train_sample_count/2,
               cvSet( &trainClasses2, cvScalar(2) ) );

    // learn classifier
    CvKNearest knn( trainData, trainClasses, 0, false, K );
    CvMat* nearests = cvCreateMat( 1, K, CV_32FC1 );

    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;

            // estimates the response and get the neighbors'
            response = knn.find_nearest( &sample, K, 0, 0, nearests );

            // compute the number of neighbors representing t
            for( k = 0, accuracy = 0; k < K; k++ )
            {
                if( nearests->data.fl[k] == response )
                    accuracy++;
            }
        }
    }
}

```

```
    }
    // highlight the pixel depending on the accuracy (or con.
    cvSet2D( img, i, j, response == 1 ?
        (accuracy > 5 ? CV_RGB(180,0,0) : CV_RGB(180,
        (accuracy > 5 ? CV_RGB(0,180,0) : CV_RGB(120,
    }
}

// display the original training samples
for( i = 0; i < train_sample_count/2; i++ )
{
    CvPoint pt;
    pt.x = cvRound(trainData1.data.fl[i*2]);
    pt.y = cvRound(trainData1.data.fl[i*2+1]);
    cvCircle( img, pt, 2, CV_RGB(255,0,0), CV_FILLED );
    pt.x = cvRound(trainData2.data.fl[i*2]);
    pt.y = cvRound(trainData2.data.fl[i*2+1]);
    cvCircle( img, pt, 2, CV_RGB(0,255,0), CV_FILLED );
}

cvNamedWindow( "classifier result", 1 );
cvShowImage( "classifier result", img );
cvWaitKey(0);

cvReleaseMat( &trainClasses );
cvReleaseMat( &trainData );
return 0;
}
```

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Support Vector Machines

---

Originally, support vector machines (SVM) was a technique for building an optimal (in some sense) binary (2-class) classifier. Then the technique has been extended to regression and clustering problems. SVM is a partial case of kernel-based methods, it maps feature vectors into higher-dimensional space using some kernel function, and then it builds an optimal linear discriminating function in this space (or an optimal hyper-plane that fits into the training data, ...). in the case of SVM the kernel is not defined explicitly. Instead, a distance between any 2 points in the hyper-space needs to be defined.

The solution is optimal in a sense that the margin between the separating hyper-plane and the nearest feature vectors from the both classes (in the case of 2-class classifier) is maximal. The feature vectors that are the closest to the hyper-plane are called “support vectors”, meaning that the position of other vectors does not affect the hyper-plane (the decision function).

There are a lot of good references on SVM. Here are only a few ones to start with.

- [Burges98] C. Burges. “A tutorial on support vector machines for pattern recognition”, Knowledge Discovery and Data Mining 2(2), 1998. (available online at <http://citeseer.ist.psu.edu/burges98tutorial.html>).
- LIBSVM - A Library for Support Vector Machines. By Chih-Chung Chang and Chih-Jen Lin (<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>)

## CvSVM

---

Support Vector Machines.

```
class CvSVM : public CvStatModel
{
public:
    // SVM type
    enum { C_SVC=100, NU_SVC=101, ONE_CLASS=102, EPS_SVR=103,

    // SVM kernel type
    enum { LINEAR=0, POLY=1, RBF=2, SIGMOID=3 };

    // SVM params type
    enum { C=0, GAMMA=1, P=2, NU=3, COEF=4, DEGREE=5 };

    CvSVM();
    virtual ~CvSVM();

    CvSVM( const CvMat* _train_data, const CvMat* _responses,
           const CvMat* _var_idx=0, const CvMat* _sample_idx=
```

```

    CvSVMParams _params=CvSVMParams() );

    virtual bool train( const CvMat* _train_data, const CvMat
                        const CvMat* _var_idx=0, const CvMat*
                        CvSVMParams _params=CvSVMParams() );

    virtual bool train_auto( const CvMat* _train_data, const
                             const CvMat* _var_idx, const CvMat* _sample_idx, CvSVMParams
                             int k_fold = 10,
                             CvParamGrid C_grid      = get_default_grid(CvSVM::C),
                             CvParamGrid gamma_grid  = get_default_grid(CvSVM::GAMMA
                             CvParamGrid p_grid      = get_default_grid(CvSVM::P),
                             CvParamGrid nu_grid     = get_default_grid(CvSVM::NU),
                             CvParamGrid coef_grid   = get_default_grid(CvSVM::COEF
                             CvParamGrid degree_grid  = get_default_grid(CvSVM::DEGREE

    virtual float predict( const CvMat* _sample ) const;
    virtual int get_support_vector_count() const;
    virtual const float* get_support_vector(int i) const;
    virtual CvSVMParams get_params() const { return params; };
    virtual void clear();

    static CvParamGrid get_default_grid( int param_id );

    virtual void save( const char* filename, const char* name
    virtual void load( const char* filename, const char* name

    virtual void write( CvFileStorage* storage, const char* name
    virtual void read( CvFileStorage* storage, CvFileNode* node
    int get_var_count() const { return var_idx ? var_idx->cols

protected:
    ...
};

```

## CvSVMParams

SVM training parameters.

```

struct CvSVMParams
{
    CvSVMParams();
    CvSVMParams( int _svm_type, int _kernel_type,
                 double _degree, double _gamma, double _coef0
                 double _C, double _nu, double _p,
                 CvMat* _class_weights, CvTermCriteria _term_

    int svm_type;
    int kernel_type;

```

```

double      degree; // for poly
double      gamma;  // for poly/rbf/sigmoid
double      coef0;  // for poly/sigmoid

double      C;      // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVC
double      nu;     // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_C_SVC
double      p;      // for CV_SVM_EPS_SVR
CvMat*      class_weights; // for CV_SVM_C_SVC
CvTermCriteria term_crit; // termination criteria
};

```

The structure must be initialized and passed to the training method of [CvSVM](#).

## CvSVM::train

```

bool CvSVM::train(const CvMat* _train_data, const CvMat* _responses,
const CvMat* _var_idx=0, const CvMat* _sample_idx=0, CvSVMParams
_params=CvSVMParams())

```

Trains SVM.

The method trains the SVM model. It follows the conventions of the generic `train` “method” with the following limitations: only the `CV_ROW_SAMPLE` data layout is supported, the input variables are all ordered, the output variables can be either categorical (`_params.svm_type=CvSVM::C_SVC` or `_params.svm_type=CvSVM::NU_SVC`), or ordered (`_params.svm_type=CvSVM::EPS_SVR` or `_params.svm_type=CvSVM::NU_SVR`), or not required at all (`_params.svm_type=CvSVM::ONE_CLASS`), missing measurements are not supported.

All the other parameters are gathered in [CvSVMParams](#) structure.

## CvSVM::train\_auto

```

train_auto(const CvMat* _train_data, const CvMat* _responses, const
CvMat* _var_idx, const CvMat* _sample_idx, CvSVMParams params, int k_fold = 10,
CvParamGrid C_grid = get_default_grid(CvSVM::C), CvParamGrid gamma_grid =
get_default_grid(CvSVM::GAMMA), CvParamGrid p_grid = get_default_grid
(CvSVM::P), CvParamGrid nu_grid = get_default_grid(CvSVM::NU), CvParamGrid
coef_grid = get_default_grid(CvSVM::COEF), CvParamGrid degree_grid =
get_default_grid(CvSVM::DEGREE))

```

Trains SVM with optimal parameters.

**Parameter:** `k_fold` – Cross-validation parameter. The training set is divided

into `k_fold` subsets, one subset being used to train the model, the others forming the test set. So, the SVM algorithm is executed `k_fold` times.

The method trains the SVM model automatically by choosing the optimal parameters `C`, `gamma`, `p`, `nu`, `coef0`, `degree` from [CvSVMParams](#). By optimal one means that the cross-validation estimate of the test set error is minimal. The parameters are iterated by a logarithmic grid, for example, the parameter `gamma` takes the values in the set ( *min*, *min \* step*, *min \* step*<sup>2</sup>, ... *min \* step*<sup>*n*</sup> ) where *min* is `gamma_grid.min_val`, *step* is `gamma_grid.step`, and *n* is the maximal index such, that

$$\text{gamma\_grid.min\_val} * \text{gamma\_grid.step}^n < \text{gamma\_grid.max\_val}$$

So `step` must always be greater than 1.

If there is no need in optimization in some parameter, the according grid step should be set to any value less or equal to 1. For example, to avoid optimization in `gamma` one should set `gamma_grid.step = 0`, `gamma_grid.min_val`, `gamma_grid.max_val` being arbitrary numbers. In this case, the value `params.gamma` will be taken for `gamma`.

And, finally, if the optimization in some parameter is required, but there is no idea of the corresponding grid, one may call the function [CvSVM::get\\_default\\_grid](#). In order to generate a grid, say, for `gamma`, call `CvSVM::get_default_grid(CvSVM::GAMMA)`.

This function works for the case of classification (`params.svm_type=CvSVM::C_SVC` or `params.svm_type=CvSVM::NU_SVC`) as well as for the regression (`params.svm_type=CvSVM::EPS_SVR` or `params.svm_type=CvSVM::NU_SVR`). If `params.svm_type=CvSVM::ONE_CLASS`, no optimization is made and the usual SVM with specified in `params` parameters is executed.

## CvSVM::get\_default\_grid

---

CvParamGrid CvSVM::get\_default\_grid(int param\_id)

Generates a grid for the SVM parameters.

**Parameter:** *param\_id* –

Must be one of the following:

- CvSVM::C -
- CvSVM::GAMMA -
- CvSVM::P -

- **CvSVM::NU** -
- **CvSVM::COEF** -
- **CvSVM::DEGREE** -

The grid will be generated for the parameter with this ID.

The function generates a grid for the specified parameter of the SVM algorithm. The grid may be passed to the function **CvSVM::train\_auto**.

## CvSVM::get\_params

---

CvSVMParams CvSVM::**get\_params**() const

Returns the current SVM parameters.

This function may be used to get the optimal parameters that were obtained while automatically training **CvSVM::train\_auto**.

## CvSVM::get\_support\_vector\*

---

int CvSVM::**get\_support\_vector\_count**() const

const float\* CvSVM::**get\_support\_vector**(int *i*) const

Retrieves the number of support vectors and the particular vector.

The methods can be used to retrieve the set of support vectors.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/mailling list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.



# Decision Trees

---

The ML classes discussed in this section implement Classification And Regression Tree algorithms, which are described in `bgroup({#paper_Breiman84})bgroup({[Breiman84]})`.

The class `CvDTree` represents a single decision tree that may be used alone, or as a base class in tree ensembles (see *Boosting* and *Random Trees*).

A decision tree is a binary tree (i.e. tree where each non-leaf node has exactly 2 child nodes). It can be used either for classification, when each tree leaf is marked with some class label (multiple leafs may have the same label), or for regression, when each tree leaf is also assigned a constant (so the approximation function is piecewise constant).

## Predicting with Decision Trees

---

To reach a leaf node, and to obtain a response for the input feature vector, the prediction procedure starts with the root node. From each non-leaf node the procedure goes to the left (i.e. selects the left child node as the next observed node), or to the right based on the value of a certain variable, whose index is stored in the observed node. The variable can be either ordered or categorical. In the first case, the variable value is compared with the certain threshold (which is also stored in the node); if the value is less than the threshold, the procedure goes to the left, otherwise, to the right (for example, if the weight is less than 1 kilogram, the procedure goes to the left, else to the right). And in the second case the discrete variable value is tested to see if it belongs to a certain subset of values (also stored in the node) from a limited set of values the variable could take; if yes, the procedure goes to the left, else - to the right (for example, if the color is green or red, go to the left, else to the right). That is, in each node, a pair of entities (`variable_index`, `decision_rule` (threshold/subset)) is used. This pair is called a split (split on the variable `variable_index`). Once a leaf node is reached, the value assigned to this node is used as the output of prediction procedure.

Sometimes, certain features of the input vector are missed (for example, in the darkness it is difficult to determine the object color), and the prediction procedure may get stuck in the certain node (in the mentioned example if the node is split by color). To avoid such situations, decision trees use so-called surrogate splits. That is, in addition to the best “primary” split, every tree node may also be split on one or more other variables with nearly the same results.

## Training Decision Trees

---

The tree is built recursively, starting from the root node. All of the training data (feature vectors and the responses) is used to split the root node. In each node the optimum decision rule (i.e. the best “primary” split) is found based on some criteria (in ML `gini` “purity” criteria is used for classification, and sum of squared errors is used for regression). Then, if necessary, the surrogate splits are found that resemble the results of the primary split on the training data; all of the data is divided using the primary and the surrogate splits (just like it is done in the prediction procedure) between the left and the right child node. Then the procedure recursively splits both left and right nodes. At each node the recursive procedure may stop (i.e. stop splitting the node further) in one of the following cases:

- `bgroup`({depth of the tree branch being constructed has reached the specified maximum value.})
- `bgroup`({number of training samples in the node is less than the specified threshold, when it is not statistically representative to split the node further.})
- `bgroup`({all the samples in the node belong to the same class (or, in the case of regression, the variation is too small).})
- `bgroup`({the best split found does not give any noticeable improvement compared to a random choice.})

When the tree is built, it may be pruned using a cross-validation procedure, if necessary. That is, some branches of the tree that may lead to the model overfitting are cut off. Normally this procedure is only applied to standalone decision trees, while tree ensembles usually build small enough trees and use their own protection schemes against overfitting.

## Variable importance

---

Besides the obvious use of decision trees - prediction, the tree can be also used for various data analysis. One of the key properties of the constructed decision tree algorithms is that it is possible to compute importance (relative decisive power) of each variable. For example, in a spam filter that uses a set of words occurred in the message as a feature vector, the variable importance rating can be used to determine the most “spam-indicating” words and thus help to keep the dictionary size reasonable.

Importance of each variable is computed over all the splits on this variable in the tree, primary and surrogate ones. Thus, to compute variable

importance correctly, the surrogate splits must be enabled in the training parameters, even if there is no missing data.

[Breiman84] Breiman, L., Friedman, J. Olshen, R. and Stone, C. (1984), “Classification and Regression Trees”, Wadsworth.

## CvDTreeSplit

---

Decision tree node split.

```
struct CvDTreeSplit
{
    int var_idx;
    int inversed;
    float quality;
    CvDTreeSplit* next;
    union
    {
        int subset[2];
        struct
        {
            float c;
            int split_point;
        }
        ord;
    };
};
```

---

## CvDTreeNode

---

Decision tree node.

```
struct CvDTreeNode
{
    int class_idx;
    int Tn;
    double value;

    CvDTreeNode* parent;
    CvDTreeNode* left;
    CvDTreeNode* right;

    CvDTreeSplit* split;

    int sample_count;
    int depth;
    ...
};
```

---

```
};
```

Other numerous fields of **CvDTreeNode** are used internally at the training stage.

## CvDTreeParams

---

Decision tree training parameters.

```

struct CvDTreeParams
{
    int max_categories;
    int max_depth;
    int min_sample_count;
    int cv_folds;
    bool use_surrogates;
    bool use_lse_rule;
    bool truncate_pruned_tree;
    float regression_accuracy;
    const float* priors;

    CvDTreeParams() : max_categories(10), max_depth(INT_MAX),
        cv_folds(10), use_surrogates(true), use_lse_rule(true),
        truncate_pruned_tree(true), regression_accuracy(0.01f)
    {}

    CvDTreeParams( int _max_depth, int _min_sample_count,
        float _regression_accuracy, bool _use_surr,
        int _max_categories, int _cv_folds,
        bool _use_lse_rule, bool _truncate_pruned,
        const float* _priors );
};

```

---

The structure contains all the decision tree training parameters. There is a default constructor that initializes all the parameters with the default values tuned for standalone classification tree. Any of the parameters can be overridden then, or the structure may be fully initialized using the advanced variant of the constructor.

## CvDTreeTrainData

---

Decision tree training data and shared data for tree ensembles.

```

struct CvDTreeTrainData
{
    CvDTreeTrainData();
    CvDTreeTrainData( const CvMat* _train_data, int _tflag,
        const CvMat* _responses, const CvMat* _
        const CvMat* _sample_idx=0, const CvMat

```

```

        const CvMat* _missing_mask=0,
        const CvDTreeParams& _params=CvDTreeParams
        bool _shared=false, bool _add_labels=false
virtual ~CvDTreeTrainData();

virtual void set_data( const CvMat* _train_data, int tflag
        const CvMat* _responses, const CvMat
        const CvMat* _sample_idx=0, const CvMat
        const CvMat* _missing_mask=0,
        const CvDTreeParams& _params=CvDTreeParams
        bool _shared=false, bool _add_labels
        bool _update_data=false );

virtual void get_vectors( const CvMat* _subsample_idx,
        float* values, uchar* missing, float* responses,
        bool get_class_idx=false );

virtual CvDTreeNode* subsample_data( const CvMat* _subsample_idx

virtual void write_params( CvFileStorage* fs );
virtual void read_params( CvFileStorage* fs, CvFileNode*

// release all the data
virtual void clear();

int get_num_classes() const;
int get_var_type(int vi) const;
int get_work_var_count() const;

virtual int* get_class_labels( CvDTreeNode* n );
virtual float* get_ord_responses( CvDTreeNode* n );
virtual int* get_labels( CvDTreeNode* n );
virtual int* get_cat_var_data( CvDTreeNode* n, int vi );
virtual CvPair32s32f* get_ord_var_data( CvDTreeNode* n, int
virtual int get_child_buf_idx( CvDTreeNode* n );

////////////////////////////////////

virtual bool set_params( const CvDTreeParams& params );
virtual CvDTreeNode* new_node( CvDTreeNode* parent, int count
        int storage_idx, int offset

virtual CvDTreeSplit* new_split_ord( int vi, float cmp_val
        int split_point, int inversed, float quality
virtual CvDTreeSplit* new_split_cat( int vi, float quality
virtual void free_node_data( CvDTreeNode* node );
virtual void free_train_data();
virtual void free_node( CvDTreeNode* node );

int sample_count, var_all, var_count, max_c_count;
int ord_var_count, cat_var_count;

```

```

bool have_labels, have_priors;
bool is_classifier;

int buf_count, buf_size;
bool shared;

CvMat* cat_count;
CvMat* cat_ofs;
CvMat* cat_map;

CvMat* counts;
CvMat* buf;
CvMat* direction;
CvMat* split_buf;

CvMat* var_idx;
CvMat* var_type; // i-th element =
                  //   k<0 - ordered
                  //   k>=0 - categorical, see k-th element of ca

CvMat* priors;

CvDTreeParams params;

CvMemStorage* tree_storage;
CvMemStorage* temp_storage;

CvDTreeNode* data_root;

CvSet* node_heap;
CvSet* split_heap;
CvSet* cv_heap;
CvSet* nv_heap;

CvRNG rng;
};

```

---

This structure is mostly used internally for storing both standalone trees and tree ensembles efficiently. Basically, it contains 3 types of information:

- `bgroup`({The training parameters, an instance of `cross`{`CvDTreeParams`}.})
- `bgroup`({The training data, preprocessed in order to find the best splits more efficiently. For tree ensembles this preprocessed data is reused by all the trees. Additionally, the training data characteristics that are shared by all trees in the ensemble are stored here: variable types, the number of classes, class label compression map etc.})
- `bgroup`({Buffers, memory storages for tree nodes, splits and other elements of the trees constructed.})

There are 2 ways of using this structure. In simple cases (e.g. a standalone tree, or

the ready-to-use “black box” tree ensemble from ML, like *Random Trees* or *Boosting*) there is no need to care or even to know about the structure - just construct the needed statistical model, train it and use it. The **CvDTreeTrainData** structure will be constructed and used internally. However, for custom tree algorithms, or another sophisticated cases, the structure may be constructed and used explicitly. The scheme is the following:

- The structure is initialized using the default constructor, followed by `set_data` (or it is built using the full form of constructor). The parameter `_shared` must be set to `true`.
- One or more trees are trained using this data, see the special form of the method **`CvDTree::train`**.
- Finally, the structure can be released only after all the trees using it are released.

## CvDTree

---

Decision tree.

```
class CvDTree : public CvStatModel
{
public:
    CvDTree();
    virtual ~CvDTree();

    virtual bool train( const CvMat* _train_data, int _tflag,
                      const CvMat* _responses, const CvMat*
                      const CvMat* _sample_idx=0, const CvM
                      const CvMat* _missing_mask=0,
                      CvDTreeParams params=CvDTreeParams()

    virtual bool train( CvDTreeTrainData* _train_data,
                      const CvMat* _subsample_idx );

    virtual CvDTreeNode* predict( const CvMat* _sample,
                                  const CvMat* _missing_data_mask=0
                                  bool raw_mode=false ) const
    virtual const CvMat* get_var_importance();
    virtual void clear();

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name )

    // special read & write methods for trees in the tree ens
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                      CvDTreeTrainData* data );
    virtual void write( CvFileStorage* fs );
```

```

const CvDTreeNode* get_root() const;
int get_pruned_tree_idx() const;
CvDTreeTrainData* get_data();

```

#### protected:

```

virtual bool do_train( const CvMat* _subsample_idx );

virtual void try_split_node( CvDTreeNode* n );
virtual void split_node_data( CvDTreeNode* n );
virtual CvDTreeSplit* find_best_split( CvDTreeNode* n );
virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n );
virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n );
virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n,
virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n,
virtual CvDTreeSplit* find_surrogate_split_ord( CvDTreeNode* n,
virtual CvDTreeSplit* find_surrogate_split_cat( CvDTreeNode* n );
virtual double calc_node_dir( CvDTreeNode* node );
virtual void complete_node_dir( CvDTreeNode* node );
virtual void cluster_categories( const int* vectors, int
    int var_count, int* sums, int k, int* cluster_labels );

virtual void calc_node_value( CvDTreeNode* node );

virtual void prune_cv();
virtual double update_tree_rnc( int T, int fold );
virtual int cut_tree( int T, int fold, double min_alpha );
virtual void free_prune_data(bool cut_tree);
virtual void free_tree();

virtual void write_node( CvFileStorage* fs, CvDTreeNode* node );
virtual void write_split( CvFileStorage* fs, CvDTreeSplit* split );
virtual CvDTreeNode* read_node( CvFileStorage* fs,
    CvFileNode* node,
    CvDTreeNode* parent );
virtual CvDTreeSplit* read_split( CvFileStorage* fs, CvFileNode* node );
virtual void write_tree_nodes( CvFileStorage* fs );
virtual void read_tree_nodes( CvFileStorage* fs, CvFileNode* node );

CvDTreeNode* root;

int pruned_tree_idx;
CvMat* var_importance;

CvDTreeTrainData* data;
};

```

---

## CvDTree::train

---

```
bool CvDTree::train(const CvMat* _train_data, int _tflag, const CvMat*
_responses, const CvMat* _var_idx=0, const CvMat* _sample_idx=0, const CvMat*
_var_type=0, const CvMat* _missing_mask=0, CvDTreeParams
params=CvDTreeParams())
```

```
bool CvDTree::train(CvDTreeTrainData* _train_data, const CvMat*
_subsample_idx)
```

Trains a decision tree.

There are 2 `train` methods in **CvDTree**.

The first method follows the generic **CvStatModel::train** conventions, it is the most complete form. Both data layouts (`_tflag=CV_ROW_SAMPLE` and `_tflag=CV_COL_SAMPLE`) are supported, as well as sample and variable subsets, missing measurements, arbitrary combinations of input and output variable types etc. The last parameter contains all of the necessary training parameters, see the [CvDTreeParams](#) description.

The second method `train` is mostly used for building tree ensembles. It takes the pre-constructed [CvDTreeTrainData](#) instance and the optional subset of training set. The indices in `_subsample_idx` are counted relatively to the `_sample_idx`, passed to **CvDTreeTrainData** constructor. For example, if `_sample_idx=[1, 5, 7, 100]`, then `_subsample_idx=[0, 3]` means that the samples `[1, 100]` of the original training set are used.

## CvDTree::predict

---

```
CvDTreeNode* CvDTree::predict(const CvMat* _sample, const CvMat*
_missing_data_mask=0, bool raw_mode=false) const
```

Returns the leaf node of the decision tree corresponding to the input vector.

The method takes the feature vector and the optional missing measurement mask on input, traverses the decision tree and returns the reached leaf node on output. The prediction result, either the class label or the estimated function value, may be retrieved as the `value` field of the [CvDTreeNode](#) structure, for example: `dtree->predict(sample,mask)->value`.

The last parameter is normally set to `false`, implying a regular input. If it is `true`, the method assumes that all the values of the discrete input variables have been already normalized to `0` to `num_of_categoriesi - 1` ranges. (as the decision tree uses such normalized representation internally). It is useful for faster prediction with tree ensembles. For ordered input variables the flag is not

used.

Example: Building A Tree for Classifying Mushrooms. See the `mushroom.cpp` sample that demonstrates how to build and use the decision tree.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Boosting

A common machine learning task is supervised learning. In supervised learning, the goal is to learn the functional relationship  $F : y = F(x)$  between the input  $x$  and the output  $y$ . Predicting the qualitative output is called classification, while predicting the quantitative output is called regression.

Boosting is a powerful learning concept, which provide a solution to the supervised classification learning task. It combines the performance of many “weak” classifiers to produce a powerful ‘committee’ *HTF01*. A weak classifier is only required to be better than chance, and thus can be very simple and computationally inexpensive. Many of them smartly combined, however, results in a strong classifier, which often outperforms most ‘monolithic’ strong classifiers such as SVMs and Neural Networks.

Decision trees are the most popular weak classifiers used in boosting schemes. Often the simplest decision trees with only a single split node per tree (called stumps) are sufficient.

The boosted model is based on  $N$  training examples  $(x_i, y_i) \ 1 \leq i \leq N$  with  $x_i \in R^K$  and  $y_i \in \{-1, +1\}$ .  $x_i$  is a  $K$ -component vector. Each component encodes a feature relevant for the learning task at hand. The desired two-class output is encoded as  $-1$  and  $+1$ .

Different variants of boosting are known such as Discrete Adaboost, Real AdaBoost, LogitBoost, and Gentle AdaBoost *FHT98*. All of them are very similar in their overall structure. Therefore, we will look only at the standard two-class Discrete AdaBoost algorithm as shown in the box below. Each sample is initially assigned the same weight (step 2). Next a weak classifier  $f_m(x)$  is trained on the weighted training data (step 3a). Its weighted training error and scaling factor  $c_m$  is computed (step 3b). The weights are increased for training samples, which have been misclassified (step 3c). All weights are then normalized, and the process of finding the next weak classifier continues for another  $M-1$  times. The final classifier  $F(x)$  is the sign of the weighted sum over the individual weak classifiers (step 4).

- Given  $N$  examples  $(x_i, y_i) \ 1 \leq i \leq N$  with  $x_i \in R^K, y_i \in \{-1, +1\}$ .
- Start with weights  $w_i = 1/N, i = 1, \dots, N$ .
- Repeat for  $m = 1, 2, \dots, M$ :
- Fit the classifier  $f_m(x) \in \{-1, 1\}$ , using weights  $w_i$  on the training data.
- Compute  $err_m = E_w[1_{(y \neq f_m(x))}]$ ;  $c_m = \log((1 - err_m)/err_m)$ .
- Set  $w_i \leftarrow w_i \exp[c_m 1_{(y_i \neq f_m(x_i))}]$ ,  $i = 1, 2, \dots, N$ , and renormalize so that  $\sum_i w_i = 1$ .
- Output the classifier sign:  $\text{sign}[\sum_{m=1}^M c_m f_m(x)]$ .

Two-class Discrete AdaBoost Algorithm: Training (steps 1 to 3) and Evaluation (step 4) NOTE: As well as the classical boosting methods, the current implementation supports 2-class classifiers only. For  $M > 2$  classes there is the *AdaBoost.MH algorithm*, described in [:ref:`FHT98](#), that reduces the problem to the 2-class problem, yet with a much larger training set.

In order to reduce computation time for boosted models without substantially losing accuracy, the influence trimming technique may be employed. As the training algorithm proceeds and the number of trees in the ensemble is increased, a larger number of the training samples are classified correctly and with increasing confidence, thereby those samples receive smaller weights on the subsequent iterations. Examples with very low relative weight have small impact on training of the weak classifier. Thus such examples may be excluded during the weak classifier training without having much effect on the induced classifier. This process is controlled with the `weight_trim_rate` parameter. Only examples with the summary fraction `weight_trim_rate` of the total weight mass are used in the weak classifier training. Note that the weights for all training examples are recomputed at each training iteration. Examples deleted at a particular iteration may be used again for learning some of the weak classifiers further [FHT98](#).

[HTF01] Hastie, T., Tibshirani, R., Friedman, J. H. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. 2001.

[FHT98] Friedman, J. H., Hastie, T. and Tibshirani, R. Additive Logistic Regression: a Statistical View of Boosting. Technical Report, Dept. of Statistics, Stanford University, 1998.

## CvBoostParams

---

Boosting training parameters.

```
struct CvBoostParams : public CvDTreeParams
{
    int boost_type;
    int weak_count;
    int split_criteria;
    double weight_trim_rate;

    CvBoostParams();
    CvBoostParams( int boost_type, int weak_count, double wei
                    int max_depth, bool use_surrogates, const
};
```

---

The structure is derived from [CvDTreeParams](#), but not all of the decision tree parameters are supported. In particular, cross-validation is not supported.

## CvBoostTree

---

Weak tree classifier.

```
class CvBoostTree: public CvDTree
{
public:
    CvBoostTree();
    virtual ~CvBoostTree();

    virtual bool train( CvDTreeTrainData* _train_data,
                      const CvMat* subsample_idx, CvBoost*
virtual void scale( double s );
virtual void read( CvFileStorage* fs, CvFileNode* node,
                  CvBoost* ensemble, CvDTreeTrainData* _
virtual void clear();

protected:
    ...
    CvBoost* ensemble;
};
```

---

The weak classifier, a component of the boosted tree classifier [CvBoost](#), is a derivative of [CvDTree](#). Normally, there is no need to use the weak classifiers directly, however they can be accessed as elements of the sequence [CvBoost::weak](#), retrieved by [CvBoost::get\\_weak\\_predictors](#).

Note, that in the case of LogitBoost and Gentle AdaBoost each weak predictor is a regression tree, rather than a classification tree. Even in the case of Discrete AdaBoost and Real AdaBoost the [CvBoostTree::predict](#) return value ([CvDTreeNode::value](#)) is not the output class label; a negative value “votes” for class 0, a positive - for class 1. And the votes are weighted. The weight of each individual tree may be increased or decreased using the method [CvBoostTree::scale](#).

## CvBoost

---

Boosted tree classifier.

```
class CvBoost : public CvStatModel
{
public:
    // Boosting type
```

```

enum { DISCRETE=0, REAL=1, LOGIT=2, GENTLE=3 };

// Splitting criteria
enum { DEFAULT=0, GINI=1, MISCLASS=3, SQERR=4 };

CvBoost();
virtual ~CvBoost();

CvBoost( const CvMat* _train_data, int _tflag,
         const CvMat* _responses, const CvMat* _var_idx=0
         const CvMat* _sample_idx=0, const CvMat* _var_type
         const CvMat* _missing_mask=0,
         CvBoostParams params=CvBoostParams() );

virtual bool train( const CvMat* _train_data, int _tflag,
                  const CvMat* _responses, const CvMat* _var_idx=0
                  const CvMat* _sample_idx=0, const CvMat* _var_type
                  const CvMat* _missing_mask=0,
                  CvBoostParams params=CvBoostParams(),
                  bool update=false );

virtual float predict( const CvMat* _sample, const CvMat*
                      CvMat* weak_responses=0, CvSlice slice
                      bool raw_mode=false ) const;

virtual void prune( CvSlice slice );

virtual void clear();

virtual void write( CvFileStorage* storage, const char* name
virtual void read( CvFileStorage* storage, CvFileNode* node

CvSeq* get_weak_predictors();
const CvBoostParams& get_params() const;
...

protected:
virtual bool set_params( const CvBoostParams& _params );
virtual void update_weights( CvBoostTree* tree );
virtual void trim_weights();
virtual void write_params( CvFileStorage* fs );
virtual void read_params( CvFileStorage* fs, CvFileNode*

CvDTreeTrainData* data;
CvBoostParams params;
CvSeq* weak;
...
};

```

---

## CvBoost::train

---

```
bool CvBoost::train(const CvMat* _train_data, int _tflag, const CvMat*
_responses, const CvMat* _var_idx=0, const CvMat* _sample_idx=0, const CvMat*
_var_type=0, const CvMat* _missing_mask=0, CvBoostParams
params=CvBoostParams(), bool update=false)
```

Trains a boosted tree classifier.

The train method follows the common template; the last parameter `update` specifies whether the classifier needs to be updated (i.e. the new weak tree classifiers added to the existing ensemble), or the classifier needs to be rebuilt from scratch. The responses must be categorical, i.e. boosted trees can not be built for regression, and there should be 2 classes.

## CvBoost::predict

---

```
float CvBoost::predict(const CvMat* sample, const CvMat* missing=0,
CvMat* weak_responses=0, CvSlice slice=CV_WHOLE_SEQ, bool
raw_mode=false) const
```

Predicts a response for the input sample.

The method `CvBoost::predict` runs the sample through the trees in the ensemble and returns the output class label based on the weighted voting.

## CvBoost::prune

---

```
void CvBoost::prune(CvSlice slice)
```

Removes the specified weak classifiers.

The method removes the specified weak classifiers from the sequence. Note that this method should not be confused with the pruning of individual decision trees, which is currently not supported.

## CvBoost::get\_weak\_predictors

---

```
CvSeq* CvBoost::get_weak_predictors()
```

Returns the sequence of weak tree classifiers.

The method returns the sequence of weak classifiers. Each element of the sequence is a pointer to a `CvBoostTree` class (or, probably, to some of its derivatives).

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Random Trees

---

Random trees have been introduced by Leo Breiman and Adele Cutler: <http://www.stat.berkeley.edu/users/breiman/RandomForests/>. The algorithm can deal with both classification and regression problems. Random trees is a collection (ensemble) of tree predictors that is called forest further in this section (the term has been also introduced by L. Breiman). The classification works as follows: the random trees classifier takes the input feature vector, classifies it with every tree in the forest, and outputs the class label that received the majority of “votes”. In the case of regression the classifier response is the average of the responses over all the trees in the forest.

All the trees are trained with the same parameters, but on the different training sets, which are generated from the original training set using the bootstrap procedure: for each training set we randomly select the same number of vectors as in the original set ( $=N$ ). The vectors are chosen with replacement. That is, some vectors will occur more than once and some will be absent. At each node of each tree trained not all the variables are used to find the best split, rather than a random subset of them. With each node a new subset is generated, however its size is fixed for all the nodes and all the trees. It is a training parameter, set to  $\sqrt{\text{number\_of\_variables}}$  by default. None of the trees that are built are pruned.

In random trees there is no need for any accuracy estimation procedures, such as cross-validation or bootstrap, or a separate test set to get an estimate of the training error. The error is estimated internally during the training. When the training set for the current tree is drawn by sampling with replacement, some vectors are left out (so-called oob (out-of-bag) data). The size of oob data is about  $N/3$ . The classification error is estimated by using this oob-data as following:

- Get a prediction for each vector, which is oob relatively to the  $i$ -th tree, using the very  $i$ -th tree.
- After all the trees have been trained, for each vector that has ever been oob, find the class-“winner” for it (i.e. the class that has got the majority of votes in the trees, where the vector was oob) and compare it to the ground-truth response.
- Then the classification error estimate is computed as ratio of number of misclassified oob vectors to all the vectors in the original data. In the case of regression the oob-error is computed as the squared error for oob vectors difference divided by the total number of vectors.

## References:

- Machine Learning, Wald I, July 2002. <http://stat-www.berkeley.edu/users/breiman/wald2002-1.pdf>
- Looking Inside the Black Box, Wald II, July 2002. <http://stat-www.berkeley.edu/users/breiman/wald2002-2.pdf>
- Software for the Masses, Wald III, July 2002. <http://stat-www.berkeley.edu/users/breiman/wald2002-3.pdf>
- And other articles from the web site [http://www.stat.berkeley.edu/users/breiman/RandomForests/cc\\_home.htm](http://www.stat.berkeley.edu/users/breiman/RandomForests/cc_home.htm).

## CvRTParams

---

Training Parameters of Random Trees.

```

struct CvRTParams : public CvDTreeParams
{
    bool calc_var_importance;
    int nactive_vars;
    CvTermCriteria term_crit;

    CvRTParams() : CvDTreeParams( 5, 10, 0, false, 10, 0, false,
        calc_var_importance(false), nactive_vars(0) )
    {
        term_crit = cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_COST, 10, 1.0 );
    }

    CvRTParams( int _max_depth, int _min_sample_count,
        float _regression_accuracy, bool _use_surrogate,
        int _max_categories, const float* _priors,
        bool _calc_var_importance,
        int _nactive_vars, int max_tree_count,
        float forest_accuracy, int termcrit_type );
};

```

---

The set of training parameters for the forest is the superset of the training parameters for a single tree. However, Random trees do not need all the functionality/features of decision trees, most noticeably, the trees are not pruned, so the cross-validation parameters are not used.

## CvRTrees

---

Random Trees.

```

class CvRTrees : public CvStatModel
{

```

```

public:
    CvRTrees();
    virtual ~CvRTrees();
    virtual bool train( const CvMat* _train_data, int _tflag,
                     const CvMat* _responses, const CvMat*
                     const CvMat* _sample_idx=0, const CvMat
                     const CvMat* _missing_mask=0,
                     CvRTPParams params=CvRTPParams() );
    virtual float predict( const CvMat* sample, const CvMat*
                          const

    virtual void clear();

    virtual const CvMat* get_var_importance();
    virtual float get_proximity( const CvMat* sample_1, const

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name );

    CvMat* get_active_var_mask();
    CvRNG* get_rng();

    int get_tree_count() const;
    CvForestTree* get_tree(int i) const;

protected:

    bool grow_forest( const CvTermCriteria term_crit );

    // array of the trees of the forest
    CvForestTree** trees;
    CvDTreeTrainData* data;
    int ntrees;
    int nclasses;
    ...
};

```

---

## CvRTrees::train

---

`bool CvRTrees::train(const CvMat* train_data, int tflag, const CvMat* responses, const CvMat* comp_idx=0, const CvMat* sample_idx=0, const CvMat* var_type=0, const CvMat* missing_mask=0, CvRTPParams params=CvRTPParams())`  
 Trains the Random Trees model.

The method `CvRTrees::train` is very similar to the first form of `CvDTree::train`()` and follows the generic method `:ctype:`CvStatModel::train` conventions. All of the specific to the algorithm training parameters are passed as a `CvRTPParams`

instance. The estimate of the training error (`oob-error`) is stored in the protected class member `oob_error`.

## CvRTrees::predict

---

`double CvRTrees::predict(const CvMat* sample, const CvMat* missing=0) const`

Predicts the output for the input sample.

The input parameters of the prediction method are the same as in [CvDTree::predict](#), but the return value type is different. This method returns the cumulative result from all the trees in the forest (the class that receives the majority of voices, or the mean of the regression function estimates).

## CvRTrees::get\_var\_importance

---

`const CvMat* CvRTrees::get_var_importance() const`

Retrieves the variable importance array.

The method returns the variable importance vector, computed at the training stage when `:ref: `CvRTParams`::calc_var_importance` is set. If the training flag is not set, then the NULL pointer is returned. This is unlike decision trees, where variable importance can be computed anytime after the training.

## CvRTrees::get\_proximity

---

`float CvRTrees::get_proximity(const CvMat* sample_1, const CvMat* sample_2) const`

Retrieves the proximity measure between two training samples.

The method returns proximity measure between any two samples (the ratio of the those trees in the ensemble, in which the samples fall into the same leaf node, to the total number of the trees).

Example: Prediction of mushroom goodness using random trees classifier

---

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

```

#include <ctype.h>
#include "ml.h"

int main( void )
{
    CvStatModel*      cls = NULL;
    CvFileStorage*   storage = cvOpenFileStorage( "Mushroom.xml"
                                                NULL, CV_STORAGE_READ );

    CvMat*           data = (CvMat*)cvReadByName( storage, NULL
                                                "data", CV_MatType_32SC1 );
    CvMat            train_data, test_data;
    CvMat            response;
    CvMat*           missed = NULL;
    CvMat*           comp_idx = NULL;
    CvMat*           sample_idx = NULL;
    CvMat*           type_mask = NULL;
    int              resp_col = 0;
    int              i, j;
    CvRTreesParams   params;
    CvTreeClassifierTrainParams cart_params;
    const int        ntrain_samples = 1000;
    const int        ntest_samples  = 1000;
    const int        nvars = 23;

    if( data == NULL || data->cols != nvars )
    {
        puts( "Error in source data" );
        return -1;
    }

    cvGetSubRect( data, &train_data, cvRect( 0, 0, nvars, ntrain_samples ) );
    cvGetSubRect( data, &test_data, cvRect( 0, ntrain_samples,
        ntrain_samples + ntest_samples ) );

    resp_col = 0;
    cvGetCol( &train_data, &response, resp_col );

    /* create missed variable matrix */
    missed = cvCreateMat( train_data.rows, train_data.cols,
        CV_MatType_8UC1 );
    for( i = 0; i < train_data.rows; i++ )
        for( j = 0; j < train_data.cols; j++ )
            CV_MAT_ELEM( *missed, uchar, i, j )
                = (uchar)( CV_MAT_ELEM( train_data, float, i, j ) < 0 );

    /* create comp_idx vector */
    comp_idx = cvCreateMat( 1, train_data.cols-1, CV_32SC1 );
    for( i = 0; i < train_data.cols; i++ )
    {
        if( i < resp_col ) CV_MAT_ELEM( *comp_idx, int, 0, i ) = i;
        if( i > resp_col ) CV_MAT_ELEM( *comp_idx, int, 0, i-1 ) = i;
    }
}

```

```

/* create sample_idx vector */
sample_idx = cvCreateMat(1, train_data.rows, CV_32SC1);
for( j = i = 0; i < train_data.rows; i++ )
{
    if(CV_MAT_ELEM(response, float, i, 0) < 0) continue;
    CV_MAT_ELEM(*sample_idx, int, 0, j) = i;
    j++;
}
sample_idx->cols = j;

/* create type mask */
type_mask = cvCreateMat(1, train_data.cols+1, CV_8UC1);
cvSet( type_mask, cvRealScalar(CV_VAR_CATEGORICAL), 0);

// initialize training parameters
cvSetDefaultParamTreeClassifier((CvStatModelParams*)&cart_params);
cart_params.wrong_feature_as_unknown = 1;
params.tree_params = &cart_params;
params.term_crit.max_iter = 50;
params.term_crit.epsilon = 0.1;
params.term_crit.type = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS

puts("Random forest results");
cls = cvCreateRTreesClassifier( &train_data,
                               CV_ROW_SAMPLE,
                               &response,
                               (CvStatModelParams*)&
                               params,
                               comp_idx,
                               sample_idx,
                               type_mask,
                               missed );

if( cls )
{
    CvMat sample = cvMat( 1, nvars, CV_32FC1, test_data
    CvMat test_resp;
    int wrong = 0, total = 0;
    cvGetCol( &test_data, &test_resp, resp_col);
    for( i = 0; i < ntest_samples; i++, sample.data.fl
    {
        if( CV_MAT_ELEM(test_resp, float, i, 0) >= 0 )
        {
            float resp = cls->predict( cls, &sample, NULL
            wrong += (fabs(resp-response.data.fl[i]) >
            total++;
        }
    }
    printf( "Test set error = %.2f\n", wrong*100.f/(float
}
else
    puts("Error forest creation");

```

```
    cvReleaseMat (&missed);  
    cvReleaseMat (&sample_idx);  
    cvReleaseMat (&comp_idx);  
    cvReleaseMat (&type_mask);  
    cvReleaseMat (&data);  
    cvReleaseStatModel (&cls);  
    cvReleaseFileStorage (&storage);  
    return 0;  
}
```

---

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

# Expectation-Maximization

The EM (Expectation-Maximization) algorithm estimates the parameters of the multivariate probability density function in the form of a Gaussian mixture distribution with a specified number of mixtures.

Consider the set of the feature vectors  $x_1, x_2, \dots, x_N$  :  $N$  vectors from a  $d$ -dimensional Euclidean space drawn from a Gaussian mixture:

$$p(x; a_k, S_k, \pi_k) = \sum_{k=1}^m \pi_k p_k(x), \quad \pi_k \geq 0, \quad \sum_{k=1}^m \pi_k = 1,$$

$$p_k(x) = \varphi(x; a_k, S_k) = \frac{1}{(2\pi)^{d/2} |S_k|^{1/2}} \exp \left\{ -\frac{1}{2} (x - a_k)^T S_k^{-1} (x - a_k) \right\},$$

where  $m$  is the number of mixtures,  $p_k$  is the normal distribution density with the mean  $a_k$  and covariance matrix  $S_k$ ,  $\pi_k$  is the weight of the  $k$ -th mixture. Given the number of mixtures  $M$  and the samples  $x_i, i = 1..N$  the algorithm finds the maximum-likelihood estimates (MLE) of the all the mixture parameters, i.e.  $a_k, S_k$  and  $\pi_k$  :

$$L(x, \theta) = \log p(x, \theta) = \sum_{i=1}^N \log \left( \sum_{k=1}^m \pi_k p_k(x) \right) \rightarrow \max_{\theta \in \Theta},$$

$$\Theta = \left\{ (a_k, S_k, \pi_k) : a_k \in \mathbb{R}^d, S_k = S_k^T > 0, S_k \in \mathbb{R}^{d \times d}, \pi_k \geq 0, \sum_{k=1}^m \pi_k = 1 \right\}.$$

EM algorithm is an iterative procedure. Each iteration of it includes two steps. At the first step (Expectation-step, or E-step), we find a probability  $p_{i,k}$  (denoted  $\alpha_{i,k}$  in the formula below) of sample  $i$  to belong to mixture  $k$  using the currently available mixture parameter estimates:

$$\alpha_{ki} = \frac{\pi_k \varphi(x; a_k, S_k)}{\sum_{j=1}^m \pi_j \varphi(x; a_j, S_j)}.$$

At the second step (Maximization-step, or M-step) the mixture parameter estimates are refined using the computed probabilities:

$$\pi_k = \frac{1}{N} \sum_{i=1}^N \alpha_{ki}, \quad a_k = \frac{\sum_{i=1}^N \alpha_{ki} x_i}{\sum_{i=1}^N \alpha_{ki}}, \quad S_k = \frac{\sum_{i=1}^N \alpha_{ki} (x_i - a_k)(x_i - a_k)^T}{\sum_{i=1}^N \alpha_{ki}},$$

Alternatively, the algorithm may start with the M-step when the initial

values for  $P_{i,k}$  can be provided. Another alternative when  $P_{i,k}$  are unknown, is to use a simpler clustering algorithm to pre-cluster the input samples and thus obtain initial  $P_{i,k}$ . Often (and in ML) the *KMeans2* algorithm is used for that purpose.

One of the main that EM algorithm should deal with is the large number of parameters to estimate. The majority of the parameters sits in covariance matrices, which are  $d \times d$  elements each (where  $d$  is the feature space dimensionality). However, in many practical problems the covariance matrices are close to diagonal, or even to  $\mu_k * I$ , where  $I$  is identity matrix and  $\mu_k$  is mixture-dependent “scale” parameter. So a robust computation scheme could be to start with the harder constraints on the covariance matrices and then use the estimated parameters as an input for a less constrained optimization problem (often a diagonal covariance matrix is already a good enough approximation).

References:

- Bilmes98 J. A. Bilmes. A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models. Technical Report TR-97-021, International Computer Science Institute and Computer Science Division, University of California at Berkeley, April 1998.

## CvEMParams

---

Parameters of the EM algorithm.

```

struct CvEMParams
{
    CvEMParams() : nclusters(10), cov_mat_type(CvEM::COV_MAT_
        start_step(CvEM::START_AUTO_STEP), probs(0), weights(
            covs(0)

        {
            term_crit=cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRI
                100, FLT_EPSILON );
        }

    CvEMParams( int _nclusters, int _cov_mat_type=1/*CvEM::CC
        int _start_step=0/*CvEM::START_AUTO_STEP*/,
        CvTermCriteria _term_crit=cvTermCriteria(
            CV_TERMCRIT_ITER+CV_TERMCRI
                100, FLT_EPSILON),
        CvMat* _probs=0, CvMat* _weights=0,
        CvMat* _means=0, CvMat** _covs=0 ) :
        nclusters(_nclusters), cov_mat_type(_cov_mat_
        start_step(_start_step),
        probs(_probs), weights(_weights), means(_mean

```

```

        term_crit(_term_crit)
    {}

    int nclusters;
    int cov_mat_type;
    int start_step;
    const CvMat* probs;
    const CvMat* weights;
    const CvMat* means;
    const CvMat** covs;
    CvTermCriteria term_crit;
};

```

---

The structure has 2 constructors, the default one represents a rough rule-of-thumb, with another one it is possible to override a variety of parameters, from a single number of mixtures (the only essential problem-dependent parameter), to the initial values for the mixture parameters.

## CvEM

---

EM model.

```

class CV_EXPORTS CvEM : public CvStatModel
{
public:
    // Type of covariance matrices
    enum { COV_MAT_SPHERICAL=0, COV_MAT_DIAGONAL=1, COV_MAT_G

    // The initial step
    enum { START_E_STEP=1, START_M_STEP=2, START_AUTO_STEP=0

    CvEM();
    CvEM( const CvMat* samples, const CvMat* sample_idx=0,
          CvEMParams params=CvEMParams(), CvMat* labels=0 );
    virtual ~CvEM();

    virtual bool train( const CvMat* samples, const CvMat* sa
                        CvEMParams params=CvEMParams(), CvMat

    virtual float predict( const CvMat* sample, CvMat* probs
    virtual void clear();

    int get_nclusters() const { return params.nclusters; }
    const CvMat* get_means() const { return means; }
    const CvMat** get_covs() const { return covs; }
    const CvMat* get_weights() const { return weights; }
    const CvMat* get_probs() const { return probs; }

protected:

```

```

virtual void set_params( const CvEMParams& params,
                        const CvVectors& train_data );
virtual void init_em( const CvVectors& train_data );
virtual double run_em( const CvVectors& train_data );
virtual void init_auto( const CvVectors& samples );
virtual void kmeans( const CvVectors& train_data, int nclusters
                    CvMat* labels, CvTermCriteria criteria
                    const CvMat* means );

CvEMParams params;
double log_likelihood;

CvMat* means;
CvMat** covs;
CvMat* weights;
CvMat* probs;

CvMat* log_weight_div_det;
CvMat* inv_eigen_values;
CvMat** cov_rotate_mats;
};

```

---

## CvEM::train

---

`void CvEM::train(const CvMat* samples, const CvMat* sample_idx=0, CvEMParams params=CvEMParams(), CvMat* labels=0)`

Estimates the Gaussian mixture parameters from the sample set.

Unlike many of the ML models, EM is an unsupervised learning algorithm and it does not take responses (class labels or the function values) on input. Instead, it computes the *MLE* of the Gaussian mixture parameters from the input sample set, stores all the parameters inside the structure:  $p_{i,k}$  in `probs`,  $a_k$  in `means`,  $S_k$  in `covs[k]`,  $\pi_k$  in `weights` and optionally computes the output “class label” for each sample:  $labels_i = \arg \max_k(p_{i,k}), i = 1..N$  (i.e. indices of the most-probable mixture for each sample).

The trained model can be used further for prediction, just like any other classifier. The model trained is similar to the *Bayes classifier*.

Example: Clustering random samples of multi-Gaussian distribution using EM

---

```

#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    const int N = 4;

```

```

const int N1 = (int)sqrt((double)N);
const CvScalar colors[] = \cvdefCpp{0,0,255}},{{0,255,0}},
                        {{0,255,255}},{{255,255,0}}
                        ;

int i, j;
int nsamples = 100;
CvRNG rng_state = cvRNG(-1);
CvMat* samples = cvCreateMat( nsamples, 2, CV_32FC1 );
CvMat* labels = cvCreateMat( nsamples, 1, CV_32SC1 );
IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
float _sample[2];
CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
CvEM em_model;
CvEMParams params;
CvMat samples_part;

cvReshape( samples, samples, 2, 0 );
for( i = 0; i < N; i++ )
{
    CvScalar mean, sigma;

    // form the training samples
    cvGetRows( samples, &samples_part, i*nsamples/N,
              (i+1)*nsamples/N );
    mean = cvScalar(((i%N1)+1.)*img->width/(N1+1),
                  ((i/N1)+1.)*img->height/(N1+1));
    sigma = cvScalar(30,30);
    cvRandArr( &rng_state, &samples_part, CV_RAND_NORMAL,
              mean, sigma
              );
}
cvReshape( samples, samples, 1, 0 );

// initialize model's parameters
params.covs          = NULL;
params.means         = NULL;
params.weights       = NULL;
params.probs         = NULL;
params.nclusters     = N;
params.cov_mat_type  = CvEM::COV_MAT_SPHERICAL;
params.start_step    = CvEM::START_AUTO_STEP;
params.term_crit.max_iter = 10;
params.term_crit.epsilon = 0.1;
params.term_crit.type  = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

// cluster the data
em_model.train( samples, 0, params, labels );

#if 0
// the piece of code shows how to repeatedly optimize the model
// with less-constrained parameters
//(COV_MAT_DIAGONAL instead of COV_MAT_SPHERICAL)

```

```
// when the output of the first stage is used as input for the
CvEM em_model2;
params.cov_mat_type = CvEM::COV_MAT_DIAGONAL;
params.start_step = CvEM::START_E_STEP;
params.means = em_model.get_means();
params.covs = (const CvMat**)em_model.get_covs();
params.weights = em_model.get_weights();

em_model2.train( samples, 0, params, labels );
// to use em_model2, replace em_model.predict()
// with em_model2.predict() below
#endif
// classify every image pixel
cvZero( img );
for( i = 0; i < img->height; i++ )
{
    for( j = 0; j < img->width; j++ )
    {
        CvPoint pt = cvPoint(j, i);
        sample.data.fl[0] = (float)j;
        sample.data.fl[1] = (float)i;
        int response = cvRound(em_model.predict( &sample, NULL ));
        CvScalar c = colors[response];

        cvCircle( img, pt, 1, cvScalar(c.val[0]*0.75,
            c.val[1]*0.75,c.val[2]*0.75), CV_FILLED );
    }
}

//draw the clustered samples
for( i = 0; i < nsamples; i++ )
{
    CvPoint pt;
    pt.x = cvRound(samples->data.fl[i*2]);
    pt.y = cvRound(samples->data.fl[i*2+1]);
    cvCircle( img, pt, 1, colors[labels->data.i[i]], CV_FILLED );
}

cvNamedWindow( "EM-clustering result", 1 );
cvShowImage( "EM-clustering result", img );
cvWaitKey(0);

cvReleaseMat( &samples );
cvReleaseMat( &labels );
return 0;
}
```

---

## Help and Feedback

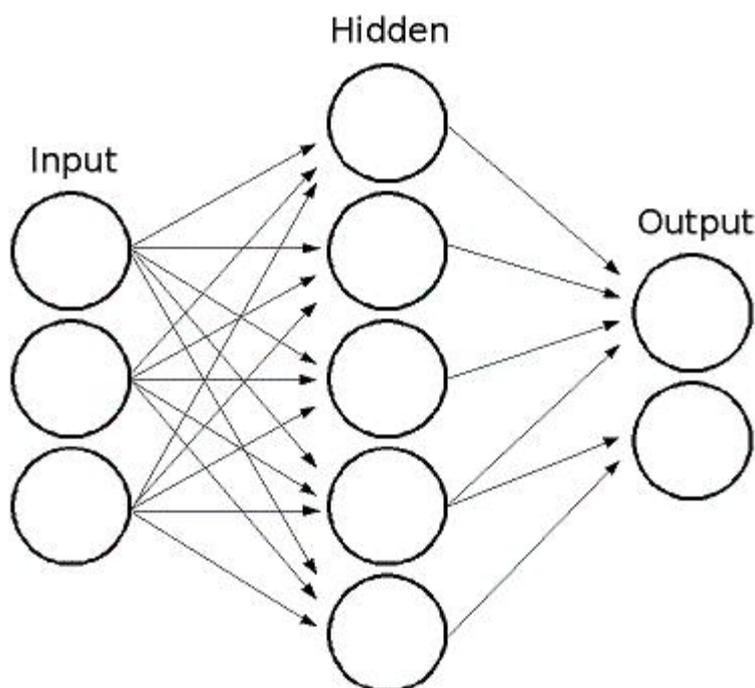
You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.

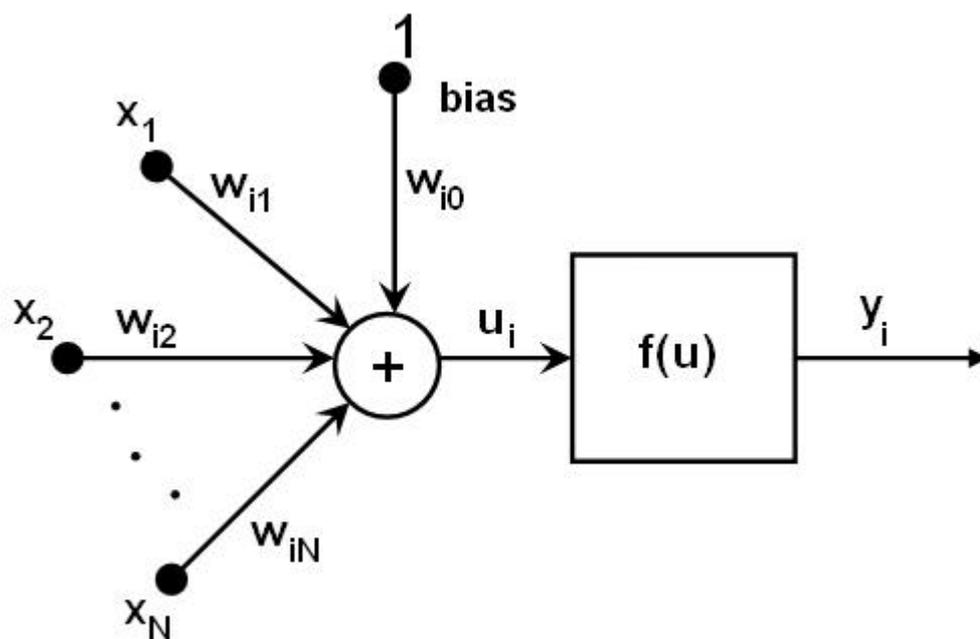
## Neural Networks

---

ML implements feed-forward artificial neural networks, more particularly, multi-layer perceptrons (MLP), the most commonly used type of neural networks. MLP consists of the input layer, output layer and one or more hidden layers. Each layer of MLP includes one or more neurons that are directionally linked with the neurons from the previous and the next layer. Here is an example of a 3-layer perceptron with 3 inputs, 2 outputs and the hidden layer including 5 neurons:



All the neurons in MLP are similar. Each of them has several input links (i.e. it takes the output values from several neurons in the previous layer on input) and several output links (i.e. it passes the response to several neurons in the next layer). The values retrieved from the previous layer are summed with certain weights, individual for each neuron, plus the bias term, and the sum is transformed using the activation function  $f$  that may be also different for different neurons. Here is the picture:



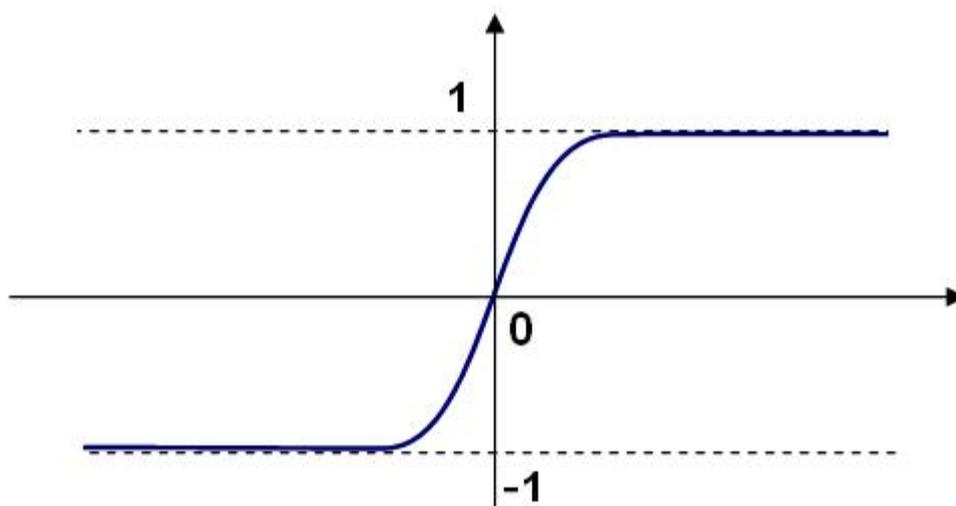
In other words, given the outputs  $x_j$  of the layer  $n$ , the outputs  $y_i$  of the layer  $n + 1$  are computed as:

$$u_i = \sum_j (w_{i,j}^{n+1} * x_j) + w_{i,bias}^{n+1}$$

$$y_i = f(u_i)$$

Different activation functions may be used, ML implements 3 standard ones:

- Identity function (**CvANN\_MLP::IDENTITY**):  $f(x) = x$
- Symmetrical sigmoid (**CvANN\_MLP::SIGMOID\_SYM**):  $f(x) = \beta * (1 - e^{-\alpha x}) / (1 + e^{-\alpha x})$ , the default choice for MLP; the standard sigmoid with  $\beta = 1, \alpha = 1$  is shown below:



- Gaussian function (**CvANN\_MLP::GAUSSIAN**):

$$f(x) = \beta e^{-\alpha x + x}, \text{ not completely supported by the moment.}$$

In ML all the neurons have the same activation functions, with the same free parameters  $(\alpha, \beta)$  that are specified by user and are not altered by the training algorithms.

So the whole trained network works as follows: It takes the feature vector on input, the vector size is equal to the size of the input layer, when the values are passed as input to the first hidden layer, the outputs of the hidden layer are computed using the weights and the activation functions and passed further downstream, until we compute the output layer.

So, in order to compute the network one needs to know all the weights  $w_{i,j}^{(n+1)}$ . The weights are computed by the training algorithm. The algorithm takes a training set: multiple input vectors with the corresponding output vectors, and iteratively adjusts the weights to try to make the network give the desired response on the provided input vectors.

The larger the network size (the number of hidden layers and their sizes), the more is the potential network flexibility, and the error on the training set could be made arbitrarily small. But at the same time the learned network will also “learn” the noise present in the training set, so the error on the test set usually starts increasing after the network size reaches some limit. Besides, the larger networks are train much longer than the smaller ones, so it is reasonable to preprocess the data (using *CalcPCA* or similar technique) and train a smaller network on only the essential features.

Another feature of the MLP's is their inability to handle categorical data as is, however there is a workaround. If a certain feature in the input or output (i.e. in the case of  $n$ -class classifier for  $n > 2$ ) layer is categorical and can take  $M > 2$  different values, it makes sense to represent it as binary tuple of  $M$  elements, where  $i$ -th element is 1 if and only if the feature is equal to the  $i$ -th value out of  $M$  possible. It will increase the size of the input/output layer, but will speedup the training algorithm convergence and at the same time enable “fuzzy” values of such variables, i.e. a tuple of probabilities instead of a fixed value.

ML implements 2 algorithms for training MLP's. The first is the classical random sequential back-propagation algorithm and the second (default one) is batch RPROP algorithm.

References:

- <http://en.wikipedia.org/wiki/Backpropagation>. Wikipedia article about the back-propagation algorithm.
- 25. LeCun, L. Bottou, G.B. Orr and K.-R. Muller, “Efficient

backprop”, in Neural Networks—Tricks of the Trade, Springer Lecture Notes in Computer Sciences 1524, pp.5-50, 1998.

- 13. Riedmiller and H. Braun, “A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm”, Proc. ICNN, San Francisco (1993).

## CvANN\_MLP\_TrainParams

---

Parameters of the MLP training algorithm.

```

struct CvANN_MLP_TrainParams
{
    CvANN_MLP_TrainParams();
    CvANN_MLP_TrainParams( CvTermCriteria term_crit, int train_method,
                          double param1, double param2=0 );
    ~CvANN_MLP_TrainParams();

    enum { BACKPROP=0, RPROP=1 };

    CvTermCriteria term_crit;
    int train_method;

    // backpropagation parameters
    double bp_dw_scale, bp_moment_scale;

    // rprop parameters
    double rp_dw0, rp_dw_plus, rp_dw_minus, rp_dw_min, rp_dw_max;
};

```

---

The structure has default constructor that initializes parameters for RPROP algorithm. There is also more advanced constructor to customize the parameters and/or choose backpropagation algorithm. Finally, the individual parameters can be adjusted after the structure is created.

## CvANN\_MLP

---

MLP model.

```

class CvANN_MLP : public CvStatModel
{
public:
    CvANN_MLP();
    CvANN_MLP( const CvMat* _layer_sizes,
              int _activ_func=SIGMOID_SYM,
              double _f_param1=0, double _f_param2=0 );
};

```

```

virtual ~CvANN_MLP();

virtual void create( const CvMat* _layer_sizes,
                    int _activ_func=SIGMOID_SYM,
                    double _f_param1=0, double _f_param2

virtual int train( const CvMat* _inputs, const CvMat* _outputs
                  const CvMat* _sample_weights,
                  const CvMat* _sample_idx=0,
                  CvANN_MLP_TrainParams _params = CvANN_MLP_Tra:
                  int flags=0 );
virtual float predict( const CvMat* _inputs,
                      CvMat* _outputs ) const;

virtual void clear();

// possible activation functions
enum { IDENTITY = 0, SIGMOID_SYM = 1, GAUSSIAN = 2 };

// available training flags
enum { UPDATE_WEIGHTS = 1, NO_INPUT_SCALE = 2, NO_OUTPUT_SCALE

virtual void read( CvFileStorage* fs, CvFileNode* node );
virtual void write( CvFileStorage* storage, const char* name

int get_layer_count() { return layer_sizes ? layer_sizes-
const CvMat* get_layer_sizes() { return layer_sizes; }

protected:

virtual bool prepare_to_train( const CvMat* _inputs, const
const CvMat* _sample_weights, const CvMat* _sample_idx
CvANN_MLP_TrainParams _params,
CvVectors* _ivecs, CvVectors* _ovecs, double** _sw

// sequential random backpropagation
virtual int train_backprop( CvVectors _ivecs, CvVectors _ovecs
const double* _sw );

// RPROP algorithm
virtual int train_rprop( CvVectors _ivecs, CvVectors _ovecs
const double* _sw );

virtual void calc_activ_func( CvMat* xf, const double* bias
virtual void calc_activ_func_deriv( CvMat* xf, CvMat* deriv
const double* bias ) const
virtual void set_activ_func( int _activ_func=SIGMOID_SYM,
double _f_param1=0, double _f_param2

virtual void init_weights();
virtual void scale_input( const CvMat* _src, CvMat* _dst
virtual void scale_output( const CvMat* _src, CvMat* _dst

```

```

virtual void calc_input_scale( const CvVectors* vecs, int
virtual void calc_output_scale( const CvVectors* vecs, int

virtual void write_params( CvFileStorage* fs );
virtual void read_params( CvFileStorage* fs, CvFileNode*

CvMat* layer_sizes;
CvMat* wbuf;
CvMat* sample_weights;
double** weights;
double f_param1, f_param2;
double min_val, max_val, min_val1, max_val1;
int activ_func;
int max_count, max_buf_sz;
CvANN_MLP_TrainParams params;
CvRNG rng;
};

```

---

Unlike many other models in ML that are constructed and trained at once, in the MLP model these steps are separated. First, a network with the specified topology is created using the non-default constructor or the method `create`. All the weights are set to zeros. Then the network is trained using the set of input and output vectors. The training procedure can be repeated more than once, i.e. the weights can be adjusted based on the new training data.

## CvANN\_MLP::create

---

```

void CvANN_MLP::create(const CvMat* _layer_sizes, int
  _activ_func=SIGMOID_SYM, double _f_param1=0, double _f_param2=0)

```

Constructs the MLP with the specified topology

- Parameters:**
- `_layer_sizes` – The integer vector specifies the number of neurons in each layer including the input and output layers.
  - `_activ_func` – Specifies the activation function for each neuron; one of `CvANN_MLP::IDENTITY`, `CvANN_MLP::SIGMOID_SYM` and `CvANN_MLP::GAUSSIAN`.
  - `_f_param1, _f_param2` – Free parameters of the activation function,  $\alpha$  and  $\beta$ , respectively. See the formulas in the introduction section.

The method creates a MLP network with the specified topology and assigns the same activation function to all the neurons.

## CvANN\_MLP::train

---

```
int CvANN_MLP::train(const CvMat* _inputs, const CvMat* _outputs, const
CvMat* _sample_weights, const CvMat* _sample_idx=0, CvANN_MLP_TrainParams
_params = CvANN_MLP_TrainParams(), int flags=0)
```

Trains/updates MLP.

- Parameters:**
- *\_inputs* – A floating-point matrix of input vectors, one vector per row.
  - *\_outputs* – A floating-point matrix of the corresponding output vectors, one vector per row.
  - *\_sample\_weights* – (RPROP only) The optional floating-point vector of weights for each sample. Some samples may be more important than others for training, and the user may want to raise the weight of certain classes to find the right balance between hit-rate and false-alarm rate etc.
  - *\_sample\_idx* – The optional integer vector indicating the samples (i.e. rows of *\_inputs* and *\_outputs*) that are taken into account.
  - *\_params* – The training params. See **CvANN\_MLP\_TrainParams** description.
  - *\_flags* –  
The various parameters to control the training algorithm. May be a combination of the following:
    - **UPDATE\_WEIGHTS = 1** - algorithm updates the network weights, rather than computes them from scratch (in the latter case the weights are initialized using Nguyen-Widrow algorithm).
    - **NO\_INPUT\_SCALE** - algorithm does not normalize the input vectors. If this flag is not set, the training algorithm normalizes each input feature independently, shifting its mean value to 0 and making the standard deviation =1. If the network is assumed to be updated frequently, the new training data could be much different from original one. In this case user should take care of proper normalization.
    - **NO\_OUTPUT\_SCALE** - algorithm does not normalize the output vectors. If the flag is not set, the training algorithm normalizes each output features independently, by transforming it to the certain range depending on the activation function used.

This method applies the specified training algorithm to compute/adjust the network weights. It returns the number of done iterations.

## Help and Feedback

You did not find what you were looking for?

- Try the **FAQ**.
- Ask a question in the **user group/ mailing list**.
- If you think something is missing or wrong in the documentation, please file a **bug report**.