The OpenCV Reference Manual

Release 2.3.2

January 11, 2012

CONTENTS

INTRODUCTION

OpenCV (Open Source Computer Vision Library: http://opencv.willowgarage.com/wiki/) is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms. The document describes the so-called OpenCV 2.x API, which is essentially a C++ API, as opposite to the C-based OpenCV 1.x API. The latter is described in opencv1x.pdf.

OpenCV has a modular structure, which means that the package includes several shared or static libraries. The following modules are available:

- **core** - a compact module defining basic data structures, including the dense multi-dimensional array `Mat` and basic functions used by all other modules.

- **imgproc** - an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), color space conversion, histograms, and so on.

- **video** - a video analysis module that includes motion estimation, background subtraction, and object tracking algorithms.

- **calib3d** - basic multiple-view geometry algorithms, single and stereo camera calibration, object pose estimation, stereo correspondence algorithms, and elements of 3D reconstruction.

- **features2d** - salient feature detectors, descriptors, and descriptor matchers.

- **objdetect** - detection of objects and instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on).

- **highgui** - an easy-to-use interface to video capturing, image and video codecs, as well as simple UI capabilities.

- **gpu** - GPU-accelerated algorithms from different OpenCV modules.

- ... some other helper modules, such as FLANN and Google test wrappers, Python bindings, and others.

The further chapters of the document describe functionality of each module. But first, make sure to get familiar with the common API concepts used thoroughly in the library.

1.1 API Concepts

cv Namespace

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

```
#include "opencv2/core/core.hpp"
...
cv::Mat H = cv::findHomography(points1, points2, CV_RANSAC, 5);
...
```

or

```
#include "opencv2/core/core.hpp"
using namespace cv;
...
Mat H = findHomography(points1, points2, CV_RANSAC, 5 );
...
```

Some of the current or future OpenCV external names may 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()));
cv::log(a, a);
a /= std::log(2.);
```

Automatic Memory Management

OpenCV handles all the memory automatically.

First of all, `std::vector`, `Mat`, and other data structures used by the functions and methods have destructors that deallocate the underlying memory buffers when needed. This means that the destructors do not always deallocate the buffers as in case of `Mat`. They take into account possible data sharing. A destructor decrements the reference counter associated with the matrix data buffer. The buffer is deallocated if and only if the reference counter reaches zero, that is, when no other structures refer to the same buffer. Similarly, when a `Mat` instance is copied, no actual data is really copied. Instead, the reference counter is incremented to memorize that there is another owner of the same data. There is also the `Mat::clone` method that creates a full copy of the matrix data. See the example below:

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

// create another header for the same matrix;
// this is an instant operation, regardless of the matrix size.
Mat B = A;
// create another header for the 3-rd row of A; no data is copied either
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 A
// to the 3-rd row of A.
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 buffers),
// but the modified version of A will still be referenced by C,
// despite that C is just a single row of the original A
B.release();

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

You see that the use of `Mat` and other basic structures is simple. But what about high-level classes or even user data types created without taking automatic memory management into account? For them, OpenCV offers the `Ptr<>` template class that is similar to `std::shared_ptr` from C++ TR1. So, instead of using plain pointers:

```
T* ptr = new T(...);
```

you can use:

```
Ptr<T> ptr = new T(...);
```

That is, `Ptr<T> ptr` incapsulates a pointer to a `T` instance and a reference counter associated with the pointer. See the `Ptr` description for details.

Automatic Allocation of the Output Data

OpenCV deallocates the memory automatically, as well as automatically allocates the memory for output function parameters most of the time. So, if a function has one or more input arrays (`cv::Mat` instances) and some output arrays, the output arrays are automatically allocated or reallocated. The size and type of the output arrays are determined from the size and type of input arrays. If needed, the functions take extra parameters that help to figure out the output array properties.

Example:

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

using namespace cv;

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

    Mat frame, edges;
    namedWindow("edges",1);
    for(;;)
    {
        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 array `frame` is automatically allocated by the `>>` operator since the video frame resolution and the bit-depth is known to the video capturing module. The array `edges` is automatically allocated by the `cvtColor` function. It has the same size and the bit-depth as the input array. The number of channels is 1 because the color conversion code `CV_BGR2GRAY` is passed, which means a color to grayscale conversion. Note that `frame` and `edges` are allocated only once during the first execution of the loop body since all the next video frames have the same resolution. If you somehow change the video resolution, the arrays are automatically reallocated.

The key component of this technology is the `Mat::create` method. It takes the desired array size and type. If the array already has the specified size and type, the method does nothing. Otherwise, it releases the previously allocated data, if any (this part involves decrementing the reference counter and comparing it with zero), and then allocates a new

buffer of the required size. Most functions call the `Mat::create` method for each output array, and so the automatic output data allocation is implemented.

Some notable exceptions from this scheme are `cv::mixChannels`, `cv::RNG::fill`, and a few other functions and methods. They are not able to allocate the output array, so you have to do this in advance.

Saturation Arithmetics

As a computer vision library, OpenCV deals a lot with image pixels that are often encoded in a compact, 8- or 16-bit per channel, form and thus have a limited value range. Furthermore, certain operations on images, like color space conversions, brightness/contrast adjustments, sharpening, complex interpolation (bi-cubic, Lanczos) can produce values out of the available range. If you just store the lowest 8 (16) bits of the result, this results in visual artifacts and may affect a further image analysis. To solve this problem, the so-called *saturation* arithmetics is used. For example, to store r, the result of an operation, to an 8-bit image, you find the nearest value within the 0..255 range:

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

Similar rules are applied to 8-bit signed, 16-bit signed and unsigned types. This semantics is used everywhere in the library. In C++ code, it is done using the `saturate_cast<>` functions that resemble standard C++ cast operations. See below the implementation of the formula provided above:

```
I.at<uchar>(y, x) = saturate_cast<uchar>(r);
```

where `cv::uchar` is an OpenCV 8-bit unsigned integer type. In the optimized SIMD code, such SSE2 instructions as `paddusb`, `packuswb`, and so on are used. They help achieve exactly the same behavior as in C++ code.

Fixed Pixel Types. Limited Use of Templates

Templates is a great feature of C++ that enables implementation of very powerful, efficient and yet safe data structures and algorithms. However, the extensive use of templates may dramatically increase compilation time and code size. Besides, it is difficult to separate an interface and implementation when templates are used exclusively. This could be fine for basic algorithms but not good for computer vision libraries where a single algorithm may span thousands lines of code. Because of this and also to simplify development of bindings for other languages, like Python, Java, Matlab that do not have templates at all or have limited template capabilities, the current OpenCV implementation is based on polymorphism and runtime dispatching over templates. In those places where runtime dispatching would be too slow (like pixel access operators), impossible (generic `Ptr<>` implementation), or just very inconvenient (`saturate_cast<>()`) the current implementation introduces small template classes, methods, and functions. Anywhere else in the current OpenCV version the use of templates is limited.

Consequently, there is a limited fixed set of primitive data types the library can operate on. That is, array elements should have one of the following types:

- 8-bit unsigned integer (uchar)
- 8-bit signed integer (schar)
- 16-bit unsigned integer (ushort)
- 16-bit signed integer (short)
- 32-bit signed integer (int)
- 32-bit floating-point number (float)
- 64-bit floating-point number (double)
- a tuple of several elements where all elements have the same type (one of the above). An array whose elements are such tuples, are called multi-channel arrays, as opposite to the single-channel arrays, whose elements are

scalar values. The maximum possible number of channels is defined by the `CV_CN_MAX` constant, which is currently set to 512.

For these basic types, the following enumeration is applied:

```
enum { CV_8U=0, CV_8S=1, CV_16U=2, CV_16S=3, CV_32S=4, CV_32F=5, CV_64F=6 };
```

Multi-channel (n-channel) types can be specified using the following options:

- `CV_8UC1` ... `CV_64FC4` constants (for a number of channels from 1 to 4)

- `CV_8UC(n)` ... `CV_64FC(n)` or `CV_MAKETYPE(CV_8U, n)` ... `CV_MAKETYPE(CV_64F, n)` macros when the number of channels is more than 4 or unknown at the compilation time.

Note: `CV_32FC1 == CV_32F`, `CV_32FC2 == CV_32FC(2) == CV_MAKETYPE(CV_32F, 2)`, and `CV_MAKETYPE(depth, n) == ((x&7)<<3) + (n-1)`. This means that the constant type is formed from the `depth`, taking the lowest 3 bits, and the number of channels minus 1, taking the next `log2(CV_CN_MAX)` bits.

Examples:

```
Mat mtx(3, 3, CV_32F); // make a 3x3 floating-point matrix
Mat cmtx(10, 1, CV_64FC2); // make a 10x1 2-channel floating-point
                           // matrix (10-element complex vector)
Mat img(Size(1920, 1080), CV_8UC3); // make a 3-channel (color) image
                                    // of 1920 columns and 1080 rows.
Mat grayscale(image.size(), CV_MAKETYPE(image.depth(), 1)); // make a 1-channel image of
                                                            // the same size and same
                                                            // channel type as img
```

Arrays with more complex elements cannot be constructed or processed using OpenCV. Furthermore, each function or method can handle only a subset of all possible array types. Usually, the more complex the algorithm is, the smaller the supported subset of formats is. See below typical examples of such limitations:

- The face detection algorithm only works with 8-bit grayscale or color images.

- Linear algebra functions and most of the machine learning algorithms work with floating-point arrays only.

- Basic functions, such as `cv::add`, support all types.

- Color space conversion functions support 8-bit unsigned, 16-bit unsigned, and 32-bit floating-point types.

The subset of supported types for each function has been defined from practical needs and could be extended in future based on user requests.

InputArray and OutputArray

Many OpenCV functions process dense 2-dimensional or multi-dimensional numerical arrays. Usually, such functions take cpp:class:*Mat* as parameters, but in some cases it's more convenient to use `std::vector<>` (for a point set, for example) or `Matx<>` (for 3x3 homography matrix and such). To avoid many duplicates in the API, special "proxy" classes have been introduced. The base "proxy" class is `InputArray`. It is used for passing read-only arrays on a function input. The derived from `InputArray` class `OutputArray` is used to specify an output array for a function. Normally, you should not care of those intermediate types (and you should not declare variables of those types explicitly) - it will all just work automatically. You can assume that instead of `InputArray`/`OutputArray` you can always use `Mat`, `std::vector<>`, `Matx<>`, `Vec<>` or `Scalar`. When a function has an optional input or output array, and you do not have or do not want one, pass `cv::noArray()`.

Error Handling

OpenCV uses exceptions to signal critical errors. When the input data has a correct format and belongs to the specified value range, but the algorithm cannot succeed for some reason (for example, the optimization algorithm did not converge), it returns a special error code (typically, just a boolean variable).

The exceptions can be instances of the `cv::Exception` class or its derivatives. In its turn, `cv::Exception` is a derivative of `std::exception`. So it can be gracefully handled in the code using other standard C++ library components.

The exception is typically thrown either using the `CV_Error(errcode, description)` macro, or its printf-like `CV_Error_(errcode, printf-spec, (printf-args))` variant, or using the `CV_Assert(condition)` macro that checks the condition and throws an exception when it is not satisfied. For performance-critical code, there is `CV_DbgAssert(condition)` that is only retained in the Debug configuration. Due to the automatic memory management, all the intermediate buffers are automatically deallocated in case of a sudden error. You only need to add a try statement to catch exceptions, if needed:

```cpp
try
{
    ... // call OpenCV
}
catch( cv::Exception& e )
{
    const char* err_msg = e.what();
    std::cout << "exception caught: " << err_msg << std::endl;
}
```

Multi-threading and Re-enterability

The current OpenCV implementation is fully re-enterable. That is, the same function, the same *constant* method of a class instance, or the same *non-constant* method of different class instances can be called from different threads. Also, the same `cv::Mat` can be used in different threads because the reference-counting operations use the architecture-specific atomic instructions.

CORE. THE CORE FUNCTIONALITY

2.1 Basic Structures

DataType

class DataType

Template "trait" class for OpenCV primitive data types. A primitive OpenCV data type is one of `unsigned char`, `bool`, `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. Any primitive type from the list can be defined by an identifier in the form `CV_<bit-depth>{U|S|F}C(<number_of_channels>)`, for example: `uchar ~ CV_8UC1`, 3-element floating-point tuple `~ CV_32FC3`, and so on. A universal OpenCV structure that is able to store a single instance of such a primitive data type is `Vec`. Multiple instances of such a type can be stored in a `std::vector`, `Mat`, `Mat_`, `SparseMat`, `SparseMat_`, or any other container that is able to store `Vec` instances.

The `DataType` class is basically used to provide a 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 is not `DataType` itself that is used but its specialized versions, such as:

```cpp
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 = CV_8U };
};
...
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 this class is to convert compilation-time type information to an OpenCV-compatible data type identifier, for example:

```
// allocates a 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 /*, that is depth == CV_64F, channels == 2 */
cout << B.depth() << ", " << B.channels() << endl;
```

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

Point_

class `Point_`

Template class for 2D points specified by its coordinates x and y. An instance of the class is interchangeable with C structures, `CvPoint` and `CvPoint2D32f`. There is also a cast operator to convert point coordinates to the specified type. The conversion from floating-point coordinates to integer coordinates is done by rounding. Commonly, the conversion uses this operation for each of the coordinates. Besides the class members listed in the declaration above, the following operations on points are implemented:

```
pt1 = pt2 + pt3;
pt1 = pt2 - pt3;
pt1 = pt2 * a;
pt1 = a * pt2;
pt1 += pt2;
pt1 -= pt2;
pt1 *= a;
double value = norm(pt); // L2 norm
pt1 == pt2;
pt1 != pt2;
```

For your convenience, the following type aliases are defined:

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

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_

class `Point3_`

Template class for 3D points specified by its coordinates x, y and z. An instance of the class is interchangeable with the C structure `CvPoint2D32f`. Similarly to `Point_`, the coordinates of 3D points can be converted to another type. The vector arithmetic and comparison operations are also supported.

The following `Point3_<>` aliases are available:

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

Size_

class Size_

Template class for specfying the size of an image or rectangle. The class includes two members called `width` and `height`. 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 `Size_<>` aliases:

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

Rect_

class Rect_

Template class for 2D rectangles, described by the following parameters:

- Coordinates of the top-left corner. This is a 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.

- Rectangle width and height.

OpenCV typically assumes 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

$$x \leq pt.x < x + \text{width}, y \leq pt.y < y + \text{height}$$

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 a rectangle by a certain offset)

- `rect = rect ± size` (expanding or shrinking a 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 `rect2` and `rect3`)

- `rect &= rect1, rect |= rect1` (and the corresponding augmenting operations)

- `rect == rect1, rect != rect1` (rectangle comparison)

This is an example how the partial ordering on rectangles can be established (rect1 $\subseteq$ rect2):

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

For your convenience, the `Rect_<>` alias is available:

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

RotatedRect

class RotatedRect

Template class for rotated rectangles specified by the center, size, and the rotation angle in degrees.

TermCriteria

class TermCriteria

Template class defining termination criteria for iterative algorithms.

Matx

class Matx

Template class for small matrices whose type and size are known at compilation time:

```
template<typename _Tp, int m, int n> class Matx {...};

typedef Matx<float, 1, 2> Matx12f;
typedef Matx<double, 1, 2> Matx12d;
...
typedef Matx<float, 1, 6> Matx16f;
typedef Matx<double, 1, 6> Matx16d;

typedef Matx<float, 2, 1> Matx21f;
typedef Matx<double, 2, 1> Matx21d;
...
typedef Matx<float, 6, 1> Matx61f;
typedef Matx<double, 6, 1> Matx61d;

typedef Matx<float, 2, 2> Matx22f;
typedef Matx<double, 2, 2> Matx22d;
...
typedef Matx<float, 6, 6> Matx66f;
typedef Matx<double, 6, 6> Matx66d;
```

If you need a more flexible type, use `Mat` . The elements of the matrix M are accessible using the M(i,j) notation. Most of the common matrix operations (see also *Matrix Expressions*) are available. To do an operation on `Matx` that is not implemented, you can easily convert the matrix to `Mat` and backwards.

```
Matx33f m(1, 2, 3,
          4, 5, 6,
          7, 8, 9);
cout << sum(Mat(m*m.t())) << endl;
```

Vec

class Vec

Template class for short numerical vectors, a partial case of Matx:

```
template<typename _Tp, int n> class Vec : public Matx<_Tp, n, 1> {...};

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;
```

It is possible to convert Vec<T,2> to/from Point_, Vec<T,3> to/from Point3_ , and Vec<T,4> to CvScalar or Scalar_. Use operator[] to access the elements of Vec.

All the expected vector operations are also implemented:

- v1 = v2 + v3
- v1 = v2 - v3
- v1 = v2 * scale
- v1 = scale * v2
- v1 = -v2
- v1 += v2 and other augmenting operations
- v1 == v2, v1 != v2
- norm(v1) (euclidean norm)

The Vec class is commonly used to describe pixel types of multi-channel arrays. See Mat for details.

Scalar_

class Scalar_

Template class for a 4-element vector derived from Vec.

```
template<typename _Tp> class Scalar_ : public Vec<_Tp, 4> { ... };

typedef Scalar_<double> Scalar;
```

Being derived from `Vec<_Tp, 4>`, `Scalar_` and `Scalar` can be used just as typical 4-element vectors. In addition, they can be converted to/from `CvScalar`. The type `Scalar` is widely used in OpenCV to pass pixel values.

Range

class Range

Template class specifying a continuous subsequence (slice) of a sequence.

```
class Range
{
public:
    ...
    int start, end;
};
```

The class is used to specify a row or a 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 an inclusive left boundary of the range and `end` is an exclusive right boundary of the range. Such a half-opened interval is usually denoted as $[start, end)$.

The static method `Range::all()` returns a 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 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

class Ptr

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 counter
    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 `Ptr<_Tp>` class is a template class that wraps pointers of the corresponding type. It is similar to `shared_ptr` that is part of the Boost library (http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/shared_ptr.htm) and also part of the C++0x standard.

This class provides the following options:

- Default constructor, copy constructor, and assignment operator for an arbitrary C++ class or a C structure. For some objects, like files, windows, mutexes, sockets, and others, a copy constructor or an 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 be written in C. However, copy constructors and default constructors can simplify programming a lot. Besides, they are often required (for example, by STL containers). By wrapping a pointer to such a complex object `TObj` to `Ptr<TObj>` , you automatically get all of the necessary constructors and the assignment operator.

- *O(1)* complexity of the above-mentioned operations. While some structures, like `std::vector`, provide a copy constructor and an assignment operator, the operations may take a considerable amount of time if the data structures are large. But if the structures are put into `Ptr<>` , the overhead is small and independent of the data size.

- Automatic destruction, even for C structures. See the example below with `FILE*` .

- Heterogeneous collections of objects. The standard STL and most other C++ and OpenCV containers can store only 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 `base_class_t*` instead but then you loose the automatic memory management. Again, by using `Ptr<base_class_t>()` instead of the raw pointers, you can solve the problem.

The `Ptr` class 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 incapsulated in the `Ptr::delete_obj()` method that 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 `delete obj;` . However, if the object is deallocated in a different way, the specialized method should be created. For example, if you want to wrap `FILE` , the `delete_obj` may be implemented as follows:

```
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.
```

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 Mat and other C++ OpenCV classes that operate on the reference counters.

Mat

class Mat

OpenCV C++ n-dimensional dense array class

```
class CV_EXPORTS Mat
{
public:
    // ... a lot of methods ...
    ...

    /*! includes several bit-fields:
         - the magic signature
         - continuity flag
         - depth
         - number of channels
     */
    int flags;
    //! the array dimensionality, >= 2
    int dims;
    //! the number of rows and columns or (-1, -1) when the array has more than 2 dimensions
    int rows, cols;
    //! pointer to the data
    uchar* data;

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

    // other members
    ...
};
```

The class Mat represents an n-dimensional dense numerical single-channel or multi-channel array. It can be used to store real or complex-valued vectors and matrices, grayscale or color images, voxel volumes, vector fields, point clouds, tensors, histograms (though, very high-dimensional histograms may be better stored in a SparseMat). The

data layout of the array M is defined by the array `M.step[]`, so that the address of element $(i_0, ..., i_{M.dims-1})$, where $0 \le i_k < M.size[k]$, is computed as:

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

In case of a 2-dimensional array, the above formula is reduced to:

$$addr(M_{i,j}) = M.data + M.step[0] * i + M.step[1] * j$$

Note that `M.step[i] >= M.step[i+1]` (in fact, `M.step[i] >= M.step[i+1]*M.size[i+1]`). This means that 2-dimensional matrices are stored row-by-row, 3-dimensional matrices are stored plane-by-plane, and so on. `M.step[M.dims-1]` is minimal and always equal to the element size `M.elemSize()`.

So, the data layout in `Mat` is fully compatible with `CvMat`, `IplImage`, and `CvMatND` types from OpenCV 1.x. It is also compatible with the majority of dense array types from the standard toolkits and SDKs, such as Numpy (ndarray), Win32 (independent device bitmaps), and others, that is, with any array that uses *steps* (or *strides*) to compute the position of a pixel. Due to this compatibility, it is possible to make a `Mat` header for user-allocated data and process it in-place using OpenCV functions.

There are many different ways to create a `Mat` object. The most popular options are listed below:

- Use the `create(nrows, ncols, type)` method or the similar `Mat(nrows, ncols, type[, fillValue])` constructor. A new array of the specified size and type is allocated. `type` has the same meaning as in the `cvCreateMat` method. For example, `CV_8UC1` means a 8-bit single-channel array, `CV_32FC2` means a 2-channel (complex) floating-point array, and so on.

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

  As noted in the introduction to this chapter, `create()` allocates only a new array when the shape or type of the current array are different from the specified ones.

- Create a multi-dimensional array:

  ```
  // create a 100x100x100 8-bit array
  int sz[] = {100, 100, 100};
  Mat bigCube(3, sz, CV_8U, Scalar::all(0));
  ```

  It passes the number of dimensions =1 to the `Mat` constructor but the created array will be 2-dimensional with the number of columns set to 1. So, `Mat::dims` is always >= 2 (can also be 0 when the array is empty).

- Use a copy constructor or assignment operator where there can be an array or expression on the right side (see below). As noted in the introduction, the array assignment is an O(1) operation because it only copies the header and increases the reference counter. The `Mat::clone()` method can be used to get a full (deep) copy of the array when you need it.

- Construct a header for a part of another array. It can be a single row, single column, several rows, several columns, rectangular region in the array (called a *minor* in algebra) or a diagonal. Such operations are also O(1) because the new header references the same data. You can actually modify a part of the array using this feature, for example:

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

  // now copy the 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 a new 320x240 image
Mat img(Size(320,240),CV_8UC3);
// select a ROI
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);
```

Due to the additional `datastart` and `dataend` members, it is possible to compute a relative sub-array position in the main *container* array 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, y=5)
```

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

- Make a header for user-allocated data. It can be useful to do the following:

    1. Process "foreign" data using OpenCV (for example, when you implement a DirectShow* filter or a processing module for `gstreamer`, and so on). For example:

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

    2. Quickly initialize small matrices and/or get a super-fast element access.

       ```
       double m[3][3] = {{a, b, c}, {d, e, f}, {g, h, i}};
       Mat M = 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 `Mat`. 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 `Mat` 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* -> Mat
    CvMat oldmat = mtx; // convert Mat -> CvMat
    CV_Assert(oldmat.cols == img->width && oldmat.rows == img->height &&
        oldmat.data.ptr == (uchar*)img->imageData && oldmat.step == img->widthStep);
    ```

- Use MATLAB-style array initializers, `zeros()`, `ones()`, `eye()`, for example:

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

- Use a comma-separated initializer:

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

  With this approach, you first call a constructor of the Mat_ class with the proper parameters, and then you just put << operator followed by comma-separated values that can be constants, variables, expressions, and so on. Also, note the extra parentheses required to avoid compilation errors.

Once the array is created, it is automatically managed via a reference-counting mechanism. If the array header is built on top of user-allocated data, you should handle the data by yourself. The array data is deallocated when no one points to it. If you want to release the data pointed by a array header before the array destructor is called, use Mat::release() .

The next important thing to learn about the array class is element access. This manual already described how to compute an address of each array element. Normally, you are not required to use the formula directly in the code. If you know the array element type (which can be retrieved using the method Mat::type()), you can access the element M_{ij} of a 2-dimensional array as:

```
M.at<double>(i,j) += 1.f;
```

assuming that M is a double-precision floating-point array. There are several variants of the method at for a different number of dimensions.

If you need to process a whole row of a 2D array, the most efficient way is to get the pointer to the row first, and then just use the plain C operator [] :

```
// compute sum of positive matrix elements
// (assuming that M isa 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 one above, do not actually depend on the array shape. They just process elements of an array one by one (or elements from multiple arrays that have the same coordinates, for example, array addition). Such operations are called *element-wise*. It makes sense to check whether all the input/output arrays are continuous, namely, have no gaps at the end of each row. If yes, process them as a long single row:

```
// compute the 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 case of the continuous matrix, the outer loop body is executed just once. So, the overhead is smaller, which is especially noticeable in 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 variant
double sum=0;
MatConstIterator_<double> it = M.begin<double>(), it_end = M.end<double>();
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 (`Mat`), `s` for a scalar (`Scalar`), `alpha` for a real-valued scalar (`double`)):

- Addition, subtraction, negation: `A+B`, `A-B`, `A+s`, `A-s`, `s+A`, `s-A`, `-A`

- Scaling: `A*alpha`

- Per-element multiplication and division: `A.mul(B)`, `A/B`, `alpha/A`

- Matrix multiplication: `A*B`

- Transposition: `A.t()` (means 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 cmpop B`, `A cmpop alpha`, `alpha cmpop A`, where `cmpop` is one of : `>`, `>=`, `==`, `!=`, `<=`, `<`. The result of comparison is an 8-bit single channel mask whose elements are set to 255 (if the particular element or pair of elements satisfy the condition) or 0.

- Bitwise logical operations: `A logicop B`, `A logicop s`, `s logicop A`, `~A`, where `logicop` is one of : `&`, `|`, `^`.

- 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`, and others.

- Matrix initializers (`Mat::eye()`, `Mat::zeros()`, `Mat::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: Comma-separated initializers and probably some other operations may require additional explicit `Mat()` or `Mat_<T>()` constuctor calls to resolve a possible ambiguity.

Here are examples of matrix expressions:

```
// compute pseudo-inverse of A, equivalent to A.inv(DECOMP_SVD)
SVD svd(A);
Mat pinvA = svd.vt.t()*Mat::diag(1./svd.w)*svd.u.t();

// compute the new vector of parameters in the Levenberg-Marquardt algorithm
```

```
x -= (A.t()*A + lambda*Mat::eye(A.cols,A.cols,A.type())).inv(DECOMP_CHOLESKY)*(A.t()*err);

// sharpen image using "unsharp mask" algorithm
Mat blurred; double sigma = 1, threshold = 5, amount = 1;
GaussianBlur(img, blurred, Size(), sigma, sigma);
Mat lowContrastMask = abs(img - blurred) < threshold;
Mat sharpened = img*(1+amount) + blurred*(-amount);
img.copyTo(sharpened, lowContrastMask);
```

Below is the formal description of the `Mat` methods.

Mat::Mat

Various Mat constructors

C++: `Mat::Mat()`

C++: `Mat::Mat`(int **rows**, int **cols**, int **type**)

C++: `Mat::Mat`(Size **size**, int **type**)

C++: `Mat::Mat`(int **rows**, int **cols**, int **type**, const Scalar& **s**)

C++: `Mat::Mat`(Size **size**, int **type**, const Scalar& **s**)

C++: `Mat::Mat`(const Mat& **m**)

C++: `Mat::Mat`(int **rows**, int **cols**, int **type**, void* **data**, size_t **step**=AUTO_STEP)

C++: `Mat::Mat`(Size **size**, int **type**, void* **data**, size_t **step**=AUTO_STEP)

C++: `Mat::Mat`(const Mat& **m**, const Range& **rowRange**, const Range& **colRange**)

C++: `Mat::Mat`(const Mat& **m**, const Rect& **roi**)

C++: `Mat::Mat`(const CvMat* **m**, bool **copyData**=false)

C++: `Mat::Mat`(const IplImage* **img**, bool **copyData**=false)

C++: template<typename T, int n> explicit `Mat::Mat`(const Vec<T, n>& **vec**, bool **copyData**=true)

C++: template<typename T, int m, int n> explicit `Mat::Mat`(const Matx<T, m, n>& **vec**, bool **copyData**=true)

C++: template<typename T> explicit `Mat::Mat`(const vector<T>& **vec**, bool **copyData**=false)

C++: `Mat::Mat`(const MatExpr& **expr**)

C++: `Mat::Mat`(int **ndims**, const int* **sizes**, int **type**)

C++: `Mat::Mat`(int **ndims**, const int* **sizes**, int **type**, const Scalar& **s**)

C++: `Mat::Mat`(int **ndims**, const int* **sizes**, int **type**, void* **data**, const size_t* **steps**=0)

C++: `Mat::Mat`(const Mat& **m**, const Range* **ranges**)

> **Parameters**
>
> > **ndims** – Array dimensionality.
> >
> > **rows** – Number of rows in a 2D array.
> >
> > **cols** – Number of columns in a 2D array.
> >
> > **size** – 2D array size: `Size(cols, rows)` . In the `Size()` constructor, the number of rows and the number of columns go in the reverse order.

sizes – Array of integers specifying an n-dimensional array shape.

type – Array type. Use `CV_8UC1, ..., CV_64FC4` to create 1-4 channel matrices, or `CV_8UC(n), ..., CV_64FC(n)` to create multi-channel (up to `CV_MAX_CN` channels) matrices.

s – An optional value to initialize each matrix element with. To set all the matrix elements to the particular value after the construction, use the assignment operator `Mat::operator=(const Scalar& value)`.

data – Pointer to the user data. Matrix constructors that take `data` and `step` parameters do not allocate matrix data. Instead, they just initialize the matrix header that points to the specified data, which means that no data is copied. This operation is very efficient and can be used to process external data using OpenCV functions. The external data is not automatically deallocated, so you should take care of it.

step – Number of bytes each matrix row occupies. The value should include the padding bytes at the end of each row, if any. If the parameter is missing (set to `AUTO_STEP`), no padding is assumed and the actual step is calculated as `cols*elemSize()`. See `Mat::elemSize()`.

steps – Array of `ndims-1` steps in case of a multi-dimensional array (the last step is always set to the element size). If not specified, the matrix is assumed to be continuous.

m – Array that (as a whole or partly) is assigned to the constructed matrix. No data is copied by these constructors. Instead, the header pointing to `m` data or its sub-array is constructed and associated with it. The reference counter, if any, is incremented. So, when you modify the matrix formed using such a constructor, you also modify the corresponding elements of `m`. If you want to have an independent copy of the sub-array, use `Mat::clone()`.

img – Pointer to the old-style `IplImage` image structure. By default, the data is shared between the original image and the new matrix. But when `copyData` is set, the full copy of the image data is created.

vec – STL vector whose elements form the matrix. The matrix has a single column and the number of rows equal to the number of vector elements. Type of the matrix matches the type of vector elements. The constructor can handle arbitrary types, for which there is a properly declared `DataType`. This means that the vector elements must be primitive numbers or uni-type numerical tuples of numbers. Mixed-type structures are not supported. The corresponding constructor is explicit. Since STL vectors are not automatically converted to `Mat` instances, you should write `Mat(vec)` explicitly. Unless you copy the data into the matrix (`copyData=true`), no new elements will be added to the vector because it can potentially yield vector data reallocation, and, thus, the matrix data pointer will be invalid.

copyData – Flag to specify whether the underlying data of the STL vector or the old-style `CvMat` or `IplImage` should be copied to (`true`) or shared with (`false`) the newly constructed matrix. When the data is copied, the allocated buffer is managed using `Mat` reference counting mechanism. While the data is shared, the reference counter is NULL, and you should not deallocate the data until the matrix is not destructed.

rowRange – Range of the `m` rows to take. As usual, the range start is inclusive and the range end is exclusive. Use `Range::all()` to take all the rows.

colRange – Range of the `m` columns to take. Use `Range::all()` to take all the columns.

ranges – Array of selected ranges of `m` along each dimensionality.

expr – Matrix expression. See *Matrix Expressions*.

These are various constructors that form a matrix. As noted in the *Automatic Allocation of the Output Data*, often the default constructor is enough, and the proper matrix will be allocated by an OpenCV function. The constructed

matrix can further be assigned to another matrix or matrix expression or can be allocated with `Mat::create()` . In the former case, the old content is de-referenced.

Mat::~Mat

The Mat destructor.

C++: `Mat::~Mat()`

The matrix destructor calls `Mat::release()` .

Mat::operator =

Provides matrix assignment operators.

C++: Mat& `Mat::operator=`(const Mat& **m**)

C++: Mat& `Mat::operator=`(const MatExpr_Base& **expr**)

C++: Mat& `Mat::operator=`(const Scalar& **s**)

> **Parameters**
>
> > **m** – Assigned, right-hand-side matrix. Matrix assignment is an O(1) operation. This means that no data is copied but the data is shared and the reference counter, if any, is incremented. Before assigning new data, the old data is de-referenced via `Mat::release()` .
> >
> > **expr** – Assigned matrix expression object. As opposite to the first form of the assignment operation, the second form can reuse already allocated matrix if it has the right size and type to fit the matrix expression result. It is automatically handled by the real function that the matrix expressions is expanded to. For example, C=A+B is expanded to `add(A, B, C)` , and `add()` takes care of automatic C reallocation.
> >
> > **s** – Scalar assigned to each matrix element. The matrix size or type is not changed.

These are available assignment operators. Since they all are very different, make sure to read the operator parameters description.

Mat::operator MatExpr

Provides a `Mat` -to- `MatExpr` cast operator.

C++: `Mat::operator` MatExpr_<Mat, Mat>() const

The cast operator should not be called explicitly. It is used internally by the *Matrix Expressions* engine.

Mat::row

Creates a matrix header for the specified matrix row.

C++: Mat `Mat::row`(int **i**) const

> **Parameters**
>
> > **i** – A 0-based row index.

The method makes a new header for the specified matrix row and returns it. This is an O(1) operation, regardless of the matrix size. The underlying data of the new matrix is shared with the original matrix. Here is the example of one of the classical basic matrix processing operations, `axpy`, used by LU and many other algorithms:

```cpp
inline void matrix_axpy(Mat& A, int i, int j, double alpha)
{
    A.row(i) += A.row(j)*alpha;
}
```

Note: In the current implementation, the following code does not work as expected:

```cpp
Mat A;
...
A.row(i) = A.row(j); // will not work
```

This happens because `A.row(i)` forms a temporary header that is further assigned to another header. Remember that each of these operations is O(1), that is, no data is copied. Thus, the above assignment is not true if you may have expected the j-th row to be copied to the i-th row. To achieve that, you should either turn this simple assignment into an expression or use the `Mat::copyTo()` method:

```cpp
Mat A;
...
// works, but looks a bit obscure.
A.row(i) = A.row(j) + 0;

// this is a bit longe, but the recommended method.
A.row(j).copyTo(A.row(i));
```

Mat::col

Creates a matrix header for the specified matrix column.

C++: Mat Mat::**col**(int **j**) const

> **Parameters**
>
> > **j** – A 0-based column index.

The method makes a new header for the specified matrix column and returns it. This is an O(1) operation, regardless of the matrix size. The underlying data of the new matrix is shared with the original matrix. See also the `Mat::row()` description.

Mat::rowRange

Creates a matrix header for the specified row span.

C++: Mat Mat::**rowRange**(int **startrow**, int **endrow**) const

C++: Mat Mat::**rowRange**(const Range& **r**) const

> **Parameters**
>
> > **startrow** – An inclusive 0-based start index of the row span.
> >
> > **endrow** – An exclusive 0-based ending index of the row span.
> >
> > **r** – Range structure containing both the start and the end indices.

The method makes a new header for the specified row span of the matrix. Similarly to `Mat::row()` and `Mat::col()`, this is an O(1) operation.

Mat::colRange

Creates a matrix header for the specified row span.

C++: Mat `Mat::colRange`(int **startcol**, int **endcol**) `const`

C++: Mat `Mat::colRange`(const Range& **r**) `const`

> **Parameters**
>
>> **startcol** – An inclusive 0-based start index of the column span.
>>
>> **endcol** – An exclusive 0-based ending index of the column span.
>>
>> **r** – Range structure containing both the start and the end indices.

The method makes a new header for the specified column span of the matrix. Similarly to `Mat::row()` and `Mat::col()`, this is an O(1) operation.

Mat::diag

Extracts a diagonal from a matrix, or creates a diagonal matrix.

C++: Mat `Mat::diag`(int **d**) `const`

C++: **static** Mat `Mat::diag`(const Mat& **matD**)

> **Parameters**
>
>> **d** – Index of the diagonal, with the following values:
>>
>> – **d=0** is the main diagonal.
>>
>> – **d>0** is a diagonal from the lower half. For example, d=1 means the diagonal is set immediately below the main one.
>>
>> – **d<0** is a diagonal from the upper half. For example, d=1 means the diagonal is set immediately above the main one.
>>
>> **matD** – Single-column matrix that forms a diagonal matrix.

The method makes a new header for the specified matrix diagonal. The new matrix is represented as a single-column matrix. Similarly to `Mat::row()` and `Mat::col()`, this is an O(1) operation.

Mat::clone

Creates a full copy of the array and the underlying data.

C++: Mat `Mat::clone`() `const`

The method creates a full copy of the array. The original `step[]` is not taken into account. So, the array copy is a continuous array occupying `total()*elemSize()` bytes.

Mat::copyTo

Copies the matrix to another one.

C++: void Mat::**copyTo**(OutputArray **m**) const

C++: void Mat::**copyTo**(OutputArray **m**, InputArray **mask**) const

> **Parameters**
>
> > **m** – Destination matrix. If it does not have a proper size or type before the operation, it is reallocated.
> >
> > **mask** – Operation mask. Its non-zero elements indicate which matrix elements need to be copied.

The method copies the matrix data to another matrix. Before copying the data, the method invokes

```
m.create(this->size(), this->type);
```

so that the destination matrix is reallocated if needed. While `m.copyTo(m);` works flawlessly, the function does not handle the case of a partial overlap between the source and the destination matrices.

When the operation mask is specified, and the `Mat::create` call shown above reallocated the matrix, the newly allocated matrix is initialized with all zeros before copying the data.

Mat::convertTo

Converts an array to another datatype with optional scaling.

C++: void Mat::**convertTo**(OutputArray **m**, int **rtype**, double **alpha**=1, double **beta**=0) const

> **Parameters**
>
> > **m** – Destination matrix. If it does not have a proper size or type before the operation, it is reallocated.
> >
> > **rtype** – Desired destination matrix type or, rather, the depth since the number of channels are the same as the source has. If `rtype` is negative, the destination matrix will have the same type as the source.
> >
> > **alpha** – Optional scale factor.
> >
> > **beta** – Optional delta added to the scaled values.

The method converts source pixel values to the target datatype. `saturate_cast<>` is applied at the end to avoid possible overflows:

$$m(x, y) = saturate\_cast < rType > (\alpha(*this)(x, y) + \beta)$$

Mat::assignTo

Provides a functional form of `convertTo`.

C++: void Mat::**assignTo**(Mat& **m**, int **type**=-1) const

> **Parameters**
>
> > **m** – Destination array.
> >
> > **type** – Desired destination array depth (or -1 if it should be the same as the source type).

This is an internally used method called by the *Matrix Expressions* engine.

Mat::setTo

Sets all or some of the array elements to the specified value.

C++: Mat& Mat::**setTo**(const Scalar& **s**, InputArray **mask**=noArray())

> **Parameters**
>
> > **s** – Assigned scalar converted to the actual array type.
> >
> > **mask** – Operation mask of the same size as *this. This is an advanced variant of the Mat::operator=(const Scalar& s) operator.

Mat::reshape

Changes the shape and/or the number of channels of a 2D matrix without copying the data.

C++: Mat Mat::**reshape**(int **cn**, int **rows**=0) const

> **Parameters**
>
> > **cn** – New number of channels. If the parameter is 0, the number of channels remains the same.
> >
> > **rows** – New number of rows. If the parameter is 0, the number of rows remains the same.

The method makes a new matrix header for *this elements. The new matrix may have a different size and/or different number of channels. Any combination is possible if:

- No extra elements are included into the new matrix and no elements are excluded. Consequently, the product rows*cols*channels() must stay the same after the transformation.

- No data is copied. That is, this is an O(1) operation. Consequently, if you change the number of rows, or the operation changes the indices of elements row in some other way, the matrix must be continuous. See Mat::isContinuous() .

For example, if there is a set of 3D points stored as an STL vector, and you want to represent the points as a 3xN matrix, do the following:

```
std::vector<Point3f> vec;
...

Mat pointMat = Mat(vec). // convert vector to Mat, O(1) operation
                 reshape(1). // make Nx3 1-channel matrix out of Nx1 3-channel.
                             // Also, an O(1) operation
                 t(); // finally, transpose the Nx3 matrix.
                      // This involves copying all the elements
```

Mat::t

Transposes a matrix.

C++: MatExpr Mat::**t**() const

The method performs matrix transposition by means of matrix expressions. It does not perform the actual transposition but returns a temporary matrix transposition object that can be further used as a part of more complex matrix expressions or can be assigned to a matrix:

```
Mat A1 = A + Mat::eye(A.size(), A.type)*lambda;
Mat C = A1.t()*A1; // compute (A + lambda*I)^t * (A + lamda*I)
```

Mat::inv

Inverses a matrix.

C++: MatExpr `Mat::`**`inv`**(int **method**=DECOMP_LU) const

> **Parameters**
>
> > **method** – Matrix inversion method. Possible values are the following:
> >
> > – **DECOMP_LU** is the LU decomposition. The matrix must be non-singular.
> >
> > – **DECOMP_CHOLESKY** is the Cholesky LL^T decomposition for symmetrical positively defined matrices only. This type is about twice faster than LU on big matrices.
> >
> > – **DECOMP_SVD** is the SVD decomposition. If the matrix is singular or even non-square, the pseudo inversion is computed.

The method performs a matrix inversion by means of matrix expressions. This means that a temporary matrix inversion object is returned by the method and can be used further as a part of more complex matrix expressions or can be assigned to a matrix.

Mat::mul

Performs an element-wise multiplication or division of the two matrices.

C++: MatExpr `Mat::`**`mul`**(InputArray **m**, double **scale**=1) const

> **Parameters**
>
> > **m** – Another array of the same type and the same size as *this, or a matrix expression.
> >
> > **scale** – Optional scale factor.

The method returns a temporary object encoding per-element array multiplication, with optional scale. Note that this is not a matrix multiplication that corresponds to a simpler "*" operator.

Example:

```
Mat C = A.mul(5/B); // equivalent to divide(A, B, C, 5)
```

Mat::cross

Computes a cross-product of two 3-element vectors.

C++: Mat `Mat::`**`cross`**(InputArray **m**) const

> **Parameters**
>
> > **m** – Another cross-product operand.

The method computes a cross-product of two 3-element vectors. The vectors must be 3-element floating-point vectors of the same shape and size. The result is another 3-element vector of the same shape and type as operands.

Mat::dot

Computes a dot-product of two vectors.

C++: double `Mat::`**`dot`**(InputArray **m**) const

> **Parameters**

> **m** – Another dot-product operand.

The method computes a dot-product of two matrices. If the matrices are not single-column or single-row vectors, the top-to-bottom left-to-right scan ordering is used to treat them as 1D vectors. The vectors must have the same size and type. If the matrices have more than one channel, the dot products from all the channels are summed together.

Mat::zeros

Returns a zero array of the specified size and type.

C++: static MatExpr Mat::**zeros**(int **rows**, int **cols**, int **type**)

C++: static MatExpr Mat::**zeros**(Size **size**, int **type**)

C++: static MatExpr Mat::**zeros**(int **ndims**, const int* **sizes**, int **type**)

> **Parameters**
>
> > **ndims** – Array dimensionality.
> >
> > **rows** – Number of rows.
> >
> > **cols** – Number of columns.
> >
> > **size** – Alternative to the matrix size specification Size(cols, rows).
> >
> > **sizes** – Array of integers specifying the array shape.
> >
> > **type** – Created matrix type.

The method returns a Matlab-style zero array initializer. It can be used to quickly form a constant array as a function parameter, part of a matrix expression, or as a matrix initializer.

```
Mat A;
A = Mat::zeros(3, 3, CV_32F);
```

In the example above, a new matrix is allocated only if A is not a 3x3 floating-point matrix. Otherwise, the existing matrix A is filled with zeros.

Mat::ones

Returns an array of all 1's of the specified size and type.

C++: static MatExpr Mat::**ones**(int **rows**, int **cols**, int **type**)

C++: static MatExpr Mat::**ones**(Size **size**, int **type**)

C++: static MatExpr Mat::**ones**(int **ndims**, const int* **sizes**, int **type**)

> **Parameters**
>
> > **ndims** – Array dimensionality.
> >
> > **rows** – Number of rows.
> >
> > **cols** – Number of columns.
> >
> > **size** – Alternative to the matrix size specification Size(cols, rows).
> >
> > **sizes** – Array of integers specifying the array shape.
> >
> > **type** – Created matrix type.

The method returns a Matlab-style 1's array initializer, similarly to `Mat::zeros()`. Note that using this method you can initialize an array with an arbitrary value, using the following Matlab idiom:

```
Mat A = Mat::ones(100, 100, CV_8U)*3; // make 100x100 matrix filled with 3.
```

The above operation does not form a 100x100 matrix of 1's and then multiply it by 3. Instead, it just remembers the scale factor (3 in this case) and use it when actually invoking the matrix initializer.

Mat::eye

Returns an identity matrix of the specified size and type.

C++: **static** MatExpr `Mat::eye`(int **rows**, int **cols**, int **type**)

C++: **static** MatExpr `Mat::eye`(Size **size**, int **type**)

>> **Parameters**

>>> **rows** – Number of rows.

>>> **cols** – Number of columns.

>>> **size** – Alternative matrix size specification as `Size(cols, rows)`.

>>> **type** – Created matrix type.

The method returns a Matlab-style identity matrix initializer, similarly to `Mat::zeros()`. Similarly to `Mat::ones()`, you can use a scale operation to create a scaled identity matrix efficiently:

```
// make a 4x4 diagonal matrix with 0.1's on the diagonal.
Mat A = Mat::eye(4, 4, CV_32F)*0.1;
```

Mat::create

Allocates new array data if needed.

C++: void `Mat::create`(int **rows**, int **cols**, int **type**)

C++: void `Mat::create`(Size **size**, int **type**)

C++: void `Mat::create`(int **ndims**, const int* **sizes**, int **type**)

>> **Parameters**

>>> **ndims** – New array dimensionality.

>>> **rows** – New number of rows.

>>> **cols** – New number of columns.

>>> **size** – Alternative new matrix size specification: `Size(cols, rows)`

>>> **sizes** – Array of integers specifying a new array shape.

>>> **type** – New matrix type.

This is one of the key `Mat` methods. Most new-style OpenCV functions and methods that produce arrays call this method for each output array. The method uses the following algorithm:

1. If the current array shape and the type match the new ones, return immediately. Otherwise, de-reference the previous data by calling `Mat::release()`.

2. Initialize the new header.

3. Allocate the new data of `total()*elemSize()` bytes.

4. Allocate the new, associated with the data, reference counter and set it to 1.

Such a scheme makes the memory management robust and efficient at the same time and helps avoid extra typing for you. This means that usually there is no need to explicitly allocate output arrays. That is, instead of writing:

```
Mat color;
...
Mat gray(color.rows, color.cols, color.depth());
cvtColor(color, gray, CV_BGR2GRAY);
```

you can simply write:

```
Mat color;
...
Mat gray;
cvtColor(color, gray, CV_BGR2GRAY);
```

because `cvtColor`, as well as the most of OpenCV functions, calls `Mat::create()` for the output array internally.

Mat::addref

Increments the reference counter.

C++: void `Mat::`**`addref`**`()`

The method increments the reference counter associated with the matrix data. If the matrix header points to an external data set (see `Mat::Mat()`), the reference counter is NULL, and the method has no effect in this case. Normally, to avoid memory leaks, the method should not be called explicitly. It is called implicitly by the matrix assignment operator. The reference counter increment is an atomic operation on the platforms that support it. Thus, it is safe to operate on the same matrices asynchronously in different threads.

Mat::release

Decrements the reference counter and deallocates the matrix if needed.

C++: void `Mat::`**`release`**`()`

The method decrements the reference counter associated with the matrix data. When the reference counter reaches 0, the matrix data is deallocated and the data and the reference counter pointers are set to NULL's. If the matrix header points to an external data set (see `Mat::Mat()`), the reference counter is NULL, and the method has no effect in this case.

This method can be called manually to force the matrix data deallocation. But since this method is automatically called in the destructor, or by any other method that changes the data pointer, it is usually not needed. The reference counter decrement and check for 0 is an atomic operation on the platforms that support it. Thus, it is safe to operate on the same matrices asynchronously in different threads.

Mat::resize

Changes the number of matrix rows.

C++: void `Mat::`**`resize`**`(size_t **sz**)`

C++: void `Mat::`**`resize`**`(size_t **sz**, const Scalar& **s**)`

      Parameters

> **sz** – New number of rows.

> **s** – Value assigned to the newly added elements.

The methods change the number of matrix rows. If the matrix is reallocated, the first `min(Mat::rows, sz)` rows are preserved. The methods emulate the corresponding methods of the STL vector class.

Mat::reserve

Reserves space for the certain number of rows.

C++: void `Mat::`**`reserve`**(size_t **sz**)

> **Parameters**

>> **sz** – Number of rows.

The method reserves space for `sz` rows. If the matrix already has enough space to store `sz` rows, nothing happens. If the matrix is reallocated, the first `Mat::rows` rows are preserved. The method emulates the corresponding method of the STL vector class.

Mat::push_back

Adds elements to the bottom of the matrix.

C++: template<typename T> void `Mat::`**`push_back`**(const T& **elem**)

C++: void `Mat::`**`push_back`**(const Mat& **elem**)

> **Parameters**

>> **elem** – Added element(s).

The methods add one or more elements to the bottom of the matrix. They emulate the corresponding method of the STL vector class. When `elem` is `Mat` , its type and the number of columns must be the same as in the container matrix.

Mat::pop_back

Removes elements from the bottom of the matrix.

C++: template<typename T> void `Mat::`**`pop_back`**(size_t **nelems**=1)

> **Parameters**

>> **nelems** – Number of removed rows. If it is greater than the total number of rows, an exception is thrown.

The method removes one or more rows from the bottom of the matrix.

Mat::locateROI

Locates the matrix header within a parent matrix.

C++: void `Mat::`**`locateROI`**(Size& **wholeSize**, Point& **ofs**) const

> **Parameters**

>> **wholeSize** – Output parameter that contains the size of the whole matrix containing `*this` is a part.

> **ofs** – Output parameter that contains an offset of `*this` inside the whole matrix.

After you extracted a submatrix from a matrix using `Mat::row()`, `Mat::col()`, `Mat::rowRange()`, `Mat::colRange()` , and others, the resultant submatrix points just to the part of the original big matrix. However, each submatrix contains information (represented by `datastart` and `dataend` fields) that helps reconstruct the original matrix size and the position of the extracted submatrix within the original matrix. The method `locateROI` does exactly that.

Mat::adjustROI

Adjusts a submatrix size and position within the parent matrix.

C++: Mat& Mat::**adjustROI**(int **dtop**, int **dbottom**, int **dleft**, int **dright**)

> **Parameters**
>
> > **dtop** – Shift of the top submatrix boundary upwards.
> >
> > **dbottom** – Shift of the bottom submatrix boundary downwards.
> >
> > **dleft** – Shift of the left submatrix boundary to the left.
> >
> > **dright** – Shift of the right submatrix boundary to the right.

The method is complimentary to `Mat::locateROI()` . The typical use of these functions is to determine the submatrix position within the parent matrix and then shift the position somehow. Typically, it can be required for filtering operations when pixels outside of the ROI should be taken into account. When all the method parameters are positive, the ROI needs to grow in all directions by the specified amount, for example:

```
A.adjustROI(2, 2, 2, 2);
```

In this example, the matrix size is increased by 4 elements in each direction. The matrix is shifted by 2 elements to the left and 2 elements up, which brings in all the necessary pixels for the filtering with the 5x5 kernel.

It is your responsibility to make sure `adjustROI` does not cross the parent matrix boundary. If it does, the function signals an error.

The function is used internally by the OpenCV filtering functions, like `filter2D()` , morphological operations, and so on.

See Also:

`copyMakeBorder()`

Mat::operator()

Extracts a rectangular submatrix.

C++: Mat Mat::**operator()**(Range **rowRange**, Range **colRange**) const

C++: Mat Mat::**operator()**(const Rect& **roi**) const

C++: Mat Mat::**operator()**(const Ranges* **ranges**) const

> **Parameters**
>
> > **rowRange** – Start and end row of the extracted submatrix. The upper boundary is not included. To select all the rows, use `Range::all()`.
> >
> > **colRange** – Start and end column of the extracted submatrix. The upper boundary is not included. To select all the columns, use `Range::all()`.

> **roi** – Extracted submatrix specified as a rectangle.

> **ranges** – Array of selected ranges along each array dimension.

The operators make a new header for the specified sub-array of `*this` . They are the most generalized forms of `Mat::row()`, `Mat::col()`, `Mat::rowRange()`, and `Mat::colRange()` . For example, A(Range(0, 10), Range::all()) is equivalent to `A.rowRange(0, 10)` . Similarly to all of the above, the operators are O(1) operations, that is, no matrix data is copied.

Mat::operator CvMat

Creates the `CvMat` header for the matrix.

C++: Mat::**operator** CvMat() const

The operator creates the `CvMat` header for the matrix without copying the underlying data. The reference counter is not taken into account by this operation. Thus, you should make sure than the original matrix is not deallocated while the `CvMat` header is used. The operator is useful for intermixing the new and the old OpenCV API's, for example:

```
Mat img(Size(320, 240), CV_8UC3);
...

CvMat cvimg = img;
mycvOldFunc( &cvimg, ...);
```

where `mycvOldFunc` is a function written to work with OpenCV 1.x data structures.

Mat::operator IplImage

Creates the `IplImage` header for the matrix.

C++: Mat::**operator** IplImage() const

The operator creates the `IplImage` header for the matrix without copying the underlying data. You should make sure than the original matrix is not deallocated while the `IplImage` header is used. Similarly to `Mat::operator CvMat` , the operator is useful for intermixing the new and the old OpenCV API's.

Mat::total

Returns the total number of array elements.

C++: size_t Mat::**total**() const

The method returns the number of array elements (a number of pixels if the array represents an image).

Mat::isContinuous

Reports whether the matrix is continuous or not.

C++: bool Mat::**isContinuous**() const

The method returns `true` if the matrix elements are stored continuously without gaps at the end of each row. Otherwise, it returns `false`. Obviously, `1x1` or `1xN` matrices are always continuous. Matrices created with `Mat::create()` are always continuous. But if you extract a part of the matrix using `Mat::col()`, `Mat::diag()` , and so on, or constructed a matrix header for externally allocated data, such matrices may no longer have this property.

The continuity flag is stored as a bit in the `Mat::flags` field and is computed automatically when you construct a matrix header. Thus, the continuity check is a very fast operation, though theoretically it could be done as follows:

```cpp
// alternative implementation of Mat::isContinuous()
bool myCheckMatContinuity(const Mat& m)
{
    //return (m.flags & Mat::CONTINUOUS_FLAG) != 0;
    return m.rows == 1 || m.step == m.cols*m.elemSize();
}
```

The method is used in quite a few of OpenCV functions. The point is that element-wise operations (such as arithmetic and logical operations, math functions, alpha blending, color space transformations, and others) do not depend on the image geometry. Thus, if all the input and output arrays are continuous, the functions can process them as very long single-row vectors. The example below illustrates how an alpha-blending function can be implemented.

```cpp
template<typename T>
void alphaBlendRGBA(const Mat& src1, const Mat& src2, Mat& dst)
{
    const float alpha_scale = (float)std::numeric_limits<T>::max(),
                inv_scale = 1.f/alpha_scale;

    CV_Assert( src1.type() == src2.type() &&
               src1.type() == CV_MAKETYPE(DataType<T>::depth, 4) &&
               src1.size() == src2.size());
    Size size = src1.size();
    dst.create(size, src1.type());

    // here is the idiom: check the arrays for continuity and,
    // if this is the case,
    // treat the arrays as 1D vectors
    if( src1.isContinuous() && src2.isContinuous() && dst.isContinuous() )
    {
        size.width *= size.height;
        size.height = 1;
    }
    size.width *= 4;

    for( int i = 0; i < size.height; i++ )
    {
        // when the arrays are continuous,
        // the outer loop is executed only once
        const T* ptr1 = src1.ptr<T>(i);
        const T* ptr2 = src2.ptr<T>(i);
        T* dptr = dst.ptr<T>(i);

        for( int j = 0; j < size.width; j += 4 )
        {
            float alpha = ptr1[j+3]*inv_scale, beta = ptr2[j+3]*inv_scale;
            dptr[j] = saturate_cast<T>(ptr1[j]*alpha + ptr2[j]*beta);
            dptr[j+1] = saturate_cast<T>(ptr1[j+1]*alpha + ptr2[j+1]*beta);
            dptr[j+2] = saturate_cast<T>(ptr1[j+2]*alpha + ptr2[j+2]*beta);
            dptr[j+3] = saturate_cast<T>((1 - (1-alpha)*(1-beta))*alpha_scale);
        }
    }
}
```

This approach, while being very simple, can boost the performance of a simple element-operation by 10-20 percents, especially if the image is rather small and the operation is quite simple.

Another OpenCV idiom in this function, a call of `Mat::create()` for the destination array, that allocates the destination array unless it already has the proper size and type. And while the newly allocated arrays are always continuous, you still need to check the destination array because `Mat::create()` does not always allocate a new matrix.

Mat::elemSize

Returns the matrix element size in bytes.

C++: size_t Mat::**elemSize**() const

The method returns the matrix element size in bytes. For example, if the matrix type is `CV_16SC3`, the method returns `3*sizeof(short)` or 6.

Mat::elemSize1

Returns the size of each matrix element channel in bytes.

C++: size_t Mat::**elemSize1**() const

The method returns the matrix element channel size in bytes, that is, it ignores the number of channels. For example, if the matrix type is `CV_16SC3`, the method returns `sizeof(short)` or 2.

Mat::type

Returns the type of a matrix element.

C++: int Mat::**type**() const

The method returns a matrix element type. This is an identifier compatible with the `CvMat` type system, like `CV_16SC3` or 16-bit signed 3-channel array, and so on.

Mat::depth

Returns the depth of a matrix element.

C++: int Mat::**depth**() const

The method returns the identifier of the matrix element depth (the type of each individual channel). For example, for a 16-bit signed 3-channel array, the method returns `CV_16S`. A complete list of matrix types contains the following values:

- `CV_8U` - 8-bit unsigned integers (`0..255`)
- `CV_8S` - 8-bit signed integers (`-128..127`)
- `CV_16U` - 16-bit unsigned integers (`0..65535`)
- `CV_16S` - 16-bit signed integers (`-32768..32767`)
- `CV_32S` - 32-bit signed integers (`-2147483648..2147483647`)
- `CV_32F` - 32-bit floating-point numbers (`-FLT_MAX..FLT_MAX, INF, NAN`)
- `CV_64F` - 64-bit floating-point numbers (`-DBL_MAX..DBL_MAX, INF, NAN`)

Mat::channels

Returns the number of matrix channels.

C++: int Mat::**channels**() const

The method returns the number of matrix channels.

Mat::step1

Returns a normalized step.

C++: size_t Mat::**step1**() const

The method returns a matrix step divided by Mat::elemSize1() . It can be useful to quickly access an arbitrary matrix element.

Mat::size

Returns a matrix size.

C++: Size Mat::**size**() const

The method returns a matrix size: Size(cols, rows) . When the matrix is more than 2-dimensional, the returned size is (-1, -1).

Mat::empty

Returns true if the array has no elemens.

C++: bool Mat::**empty**() const

The method returns true if Mat::total() is 0 or if Mat::data is NULL. Because of pop_back() and resize() methods M.total() == 0 does not imply that M.data == NULL .

Mat::ptr

Returns a pointer to the specified matrix row.

C++: uchar* Mat::**ptr**(int **i**=0)

C++: const uchar* Mat::**ptr**(int **i**=0) const

C++: template<typename _Tp> _Tp* Mat::**ptr**(int **i**=0)

C++: template<typename _Tp> const _Tp* Mat::**ptr**(int **i**=0) const

> **Parameters**
>
> > **i** – A 0-based row index.

The methods return uchar* or typed pointer to the specified matrix row. See the sample in Mat::isContinuous() to know how to use these methods.

Mat::at

Returns a reference to the specified array element.

C++: template<typename T> T& Mat::**at**(int **i**) const

C++: template<typename T> const T& Mat::**at**(int **i**) const

C++: template<typename T> T& Mat::**at**(int **i**, int **j**)

C++: template<typename T> const T& Mat::**at**(int **i**, int **j**) const

C++: template<typename T> T& Mat::**at**(Point **pt**)

C++: template<typename T> const T& Mat::**at**(Point **pt**) const

C++: template<typename T> T& Mat::**at**(int **i**, int **j**, int **k**)

C++: template<typename T> const T& Mat::**at**(int **i**, int **j**, int **k**) const

C++: template<typename T> T& Mat::**at**(const int* **idx**)

C++: template<typename T> const T& Mat::**at**(const int* **idx**) const

> **Parameters**
>
> > **i** – Index along the dimension 0
> >
> > **j** – Index along the dimension 1
> >
> > **k** – Index along the dimension 2
> >
> > **pt** – Element position specified as `Point(j,i)`.
> >
> > **idx** – Array of `Mat::dims` indices.

The template methods return a reference to the specified array element. For the sake of higher performance, the index range checks are only performed in the Debug configuration.

Note that the variants with a single index (i) can be used to access elements of single-row or single-column 2-dimensional arrays. That is, if, for example, A is a `1 x N` floating-point matrix and B is an `M x 1` integer matrix, you can simply write `A.at<float>(k+4)` and `B.at<int>(2*i+1)` instead of `A.at<float>(0,k+4)` and `B.at<int>(2*i+1,0)`, respectively.

The example below initializes a Hilbert matrix:

```
Mat H(100, 100, CV_64F);
for(int i = 0; i < H.rows; i++)
    for(int j = 0; j < H.cols; j++)
        H.at<double>(i,j)=1./(i+j+1);
```

Mat::begin

Returns the matrix iterator and sets it to the first matrix element.

C++: template<typename _Tp> MatIterator_<_Tp> Mat::**begin**()

C++: template<typename _Tp> MatConstIterator_<_Tp> Mat::**begin**() const

The methods return the matrix read-only or read-write iterators. The use of matrix iterators is very similar to the use of bi-directional STL iterators. In the example below, the alpha blending function is rewritten using the matrix iterators:

```cpp
template<typename T>
void alphaBlendRGBA(const Mat& src1, const Mat& src2, Mat& dst)
{
    typedef Vec<T, 4> VT;

    const float alpha_scale = (float)std::numeric_limits<T>::max(),
                inv_scale = 1.f/alpha_scale;

    CV_Assert( src1.type() == src2.type() &&
               src1.type() == DataType<VT>::type &&
               src1.size() == src2.size());
    Size size = src1.size();
    dst.create(size, src1.type());

    MatConstIterator_<VT> it1 = src1.begin<VT>(), it1_end = src1.end<VT>();
    MatConstIterator_<VT> it2 = src2.begin<VT>();
    MatIterator_<VT> dst_it = dst.begin<VT>();

    for( ; it1 != it1_end; ++it1, ++it2, ++dst_it )
    {
        VT pix1 = *it1, pix2 = *it2;
        float alpha = pix1[3]*inv_scale, beta = pix2[3]*inv_scale;
        *dst_it = VT(saturate_cast<T>(pix1[0]*alpha + pix2[0]*beta),
                     saturate_cast<T>(pix1[1]*alpha + pix2[1]*beta),
                     saturate_cast<T>(pix1[2]*alpha + pix2[2]*beta),
                     saturate_cast<T>((1 - (1-alpha)*(1-beta))*alpha_scale));
    }
}
```

Mat::end

Returns the matrix iterator and sets it to the after-last matrix element.

C++: template<typename _Tp> MatIterator_<_Tp> Mat::**end**()

C++: template<typename _Tp> MatConstIterator_<_Tp> Mat::**end**() const

The methods return the matrix read-only or read-write iterators, set to the point following the last matrix element.

Mat_

class Mat_

Template matrix class derived from Mat .

```cpp
template<typename _Tp> class Mat_ : public Mat
{
public:
    // ... some specific methods
    //         and
    // no new extra fields
};
```

The class Mat_<_Tp> is a "thin" template wrapper on top of the Mat class. It does not have any extra data fields. Nor this class nor Mat has any virtual methods. Thus, references or pointers to these two classes can be freely but carefully converted one to another. For example:

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

While Mat 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 the compilation 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);
```

To use Mat_ for multi-channel images/matrices, pass Vec as a Mat_ parameter:

```
// allocate a 320x240 color image and fill it with green (in RGB space)
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);
```

NAryMatIterator

class **NAryMatIterator**

n-ary multi-dimensional array iterator.

```
class CV_EXPORTS NAryMatIterator
{
public:
    //! the default constructor
    NAryMatIterator();
    //! the full constructor taking arbitrary number of n-dim matrices
    NAryMatIterator(const Mat** arrays, Mat* planes, int narrays=-1);
    //! the separate iterator initialization method
    void init(const Mat** arrays, Mat* planes, int narrays=-1);

    //! proceeds to the next plane of every iterated matrix
    NAryMatIterator& operator ++();
    //! proceeds to the next plane of every iterated matrix (postfix increment operator)
    NAryMatIterator operator ++(int);

    ...
    int nplanes; // the total number of planes
};
```

Use the class to implement unary, binary, and, generally, n-ary element-wise operations on multi-dimensional arrays. Some of the arguments of an n-ary function may be continuous arrays, some may be not. It is possible to use conventional `MatIterator` 's for each array but incrementing all of the iterators after each small operations may be a big overhead. In this case consider using `NAryMatIterator` to iterate through several matrices simultaneously as long as they have the same geometry (dimensionality and all the dimension sizes are the same). On each iteration `it.planes[0]`, `it.planes[1]` , ... will be the slices of the corresponding matrices.

The example below illustrates how you can compute a normalized and threshold 3D color histogram:

```
void computeNormalizedColorHist(const Mat& image, Mat& hist, int N, double minProb)
{
    const int histSize[] = {N, N, N};

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

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

    // the loop below assumes that the image
    // is a 8-bit 3-channel. 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;
        hist.at<float>(pix[0]*N/256, pix[1]*N/256, pix[2]*N/256) += 1.f;
    }

    minProb *= image.rows*image.cols;
    Mat plane;
    NAryMatIterator it(&hist, &plane, 1);
    double s = 0;
    // iterate through the matrix. on each iteration
    // it.planes[*] (of type Mat) will be set to the current plane.
    for(int p = 0; p < it.nplanes; p++, ++it)
    {
        threshold(it.planes[0], it.planes[0], minProb, 0, THRESH_TOZERO);
        s += sum(it.planes[0])[0];
    }

    s = 1./s;
    it = NAryMatIterator(&hist, &plane, 1);
    for(int p = 0; p < it.nplanes; p++, ++it)
        it.planes[0] *= s;
}
```

SparseMat

class **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 array 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 an 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 an 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);

// creates a 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, the result will
// be a single-column matrix.
void copyTo( Mat& m ) const;
// converts arbitrary sparse matrix to dense matrix.
// multiplies all the matrix elements by the specified scalar
void convertTo( SparseMat& m, int rtype, double alpha=1 ) const;
// converts sparse matrix to dense matrix with optional type conversion and scaling.
// 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;

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

// reallocates sparse matrix. If it was already of the proper size and type,
// it is simply cleared with clear(), otherwise,
// the old matrix is released (using release()) and the new one is allocated.
void create(int dims, const int* _sizes, int _type);
// sets all the matrix elements to 0, which means clearing the hash table.
void clear();
// manually increases reference counter to the header.
void addref();
// decreses 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
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 n-D case.
//
// return pointer to the matrix element.
//  if the element is there (it is non-zero), the pointer to it is returned
//  if it is not there and createMissing=false, NULL pointer is returned
```

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

// higher-level element access functions:
// ref<_Tp>(i0,...[,hashval]) - equivalent to *(_Tp*)ptr(i0,...true[,hashval]).
//     always return valid reference to the element.
//     If it does not exist, it is created.
// find<_Tp>(i0,...[,hashval]) - equivalent to (_const Tp*)ptr(i0,...false[,hashval]).
//     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 : _Tp(); }
//     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) const;
template<typename _Tp> const _Tp* find(int i0, size_t* hashval=0) const;

// 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) const;
template<typename _Tp> const _Tp* find(int i0, int i1, size_t* hashval=0) const;

// 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) const;
template<typename _Tp> const _Tp* find(int i0, int i1, int i2, size_t* hashval=0) const;

// 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) const;
template<typename _Tp> const _Tp* find(const int* idx, size_t* hashval=0) const;

// erase the specified matrix element.
// when there is no such an 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() const;
    template<typename _Tp> SparseMatIterator_<_Tp> end();
    template<typename _Tp> SparseMatConstIterator_<_Tp> end() const;

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

    ////////////// some internally used 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` 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 is up to you to detect such elements and delete them using `SparseMat::erase`). The non-zero elements are stored in a hash table that grows when it is filled 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`), for example:

```
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.ref<float>(idx) += 1.f;
}
```

- Sparse matrix iterators. They are similar to `MatIterator` but different from `NAryMatIterator`. That is, the iteration loop is familiar to STL 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("
    printf(":
    s += *it;
}
printf("Element sum is
```

If you run this loop, you will notice that elements are not enumerated in a logical order (lexicographical, and so

on). They come in the same order as they are stored in the hash table (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 may happen due to possible buffer reallocation.

- Combination of the above 2 methods when you need to process 2 or more sparse matrices simultaneously. For example, 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 is 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 an element with the same index in the second matrix.
        // since the hash value depends only on the element index,
        // reuse the hash value stored in the node
        float bvalue = _b->value<float>(anode->idx,&anode->hashval);
        ccorr += avalue*bvalue;
    }
    return ccorr;
}
```

SparseMat_

class **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>::type
    SparseMat_();
    SparseMat_(int dims, const int* _sizes);
    SparseMat_(const SparseMat& m);
    SparseMat_(const SparseMat_& m);
    SparseMat_(const Mat& m);
    SparseMat_(const CvSparseMat* m);
    // assignment operators; data type conversion is done when necessary
    SparseMat_& operator = (const SparseMat& m);
    SparseMat_& operator = (const SparseMat_& m);
    SparseMat_& operator = (const Mat& m);

    // equivalent to the correspoding parent class methods
    SparseMat_ clone() const;
```

```
    void create(int dims, const int* _sizes);
    operator CvSparseMat*() const;

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

    // more convenient element access operations.
    // ref() is retained (but <_Tp> specification is not needed anymore);
    // 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) const;
    _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 created in the same way as Mat_ . It simplifies notation of some operations.

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

2.2 Basic C Structures and Operations

The section describes the main data structures, used by the OpenCV 1.x API, and the basic functions to create and process the data structures.

CvPoint

struct CvPoint

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

> int **x**
> > x-coordinate

> int **y**
> > y-coordinate

C: CvPoint **cvPoint**(int **x**, int **y**)
> constructs CvPoint structure.

C: CvPoint **cvPointFrom32f**(CvPoint32f **pt**)
> converts CvPoint2D32f to CvPoint.

See Also:

Point_

CvPoint2D32f

struct CvPoint2D32f

2D point with floating-point coordinates.

> float **x**
>> x-coordinate

> float **y**
>> y-coordinate

C: CvPoint2D32f **cvPoint2D32f**(float **x**, float **y**)
> constructs CvPoint2D32f structure.

C: CvPoint2D32f **cvPointTo32f**(CvPoint **pt**)
> converts CvPoint to CvPoint2D32f.

See Also:

Point_

CvPoint3D32f

struct CvPoint3D32f

3D point with floating-point coordinates

> float **x**
>> x-coordinate

> float **y**
>> y-coordinate

> float **z**
>> z-coordinate

C: CvPoint3D32f **cvPoint3D32f**(float **x**, float **y**, float **z**)
> constructs CvPoint3D32f structure.

See Also:

Point3_

CvPoint2D64f

struct CvPoint2D64f

2D point with double-precision floating-point coordinates.

> double **x**
>> x-coordinate

      double **y**
          y-coordinate

C: CvPoint2D64f **cvPoint2D64f**(double **x**, double **y**)
      constructs `CvPoint2D64f` structure.

See Also:

Point_

CvPoint3D64f

struct `CvPoint3D64f`

3D point with double-precision floating-point coordinates.

      double **x**
          x-coordinate

      double **y**
          y-coordinate

      double **z**

C: CvPoint3D64f **cvPoint3D64f**(double **x**, double **y**, double **z**)
      constructs `CvPoint3D64f` structure.

See Also:

Point3_

CvSize

struct `CvSize`

Size of a rectangle or an image.

      int **width**
          Width of the rectangle

      int **height**
          Height of the rectangle

C: CvSize **cvSize**(int **width**, int **height**)
      constructs `CvSize` structure.

See Also:

Size_

CvSize2D32f

struct `CvSize2D32f`

Sub-pixel accurate size of a rectangle.

      float **width**
          Width of the rectangle

      float **height**
          Height of the rectangle

C: CvSize2D32f **cvSize2D23f**(float **width**, float **height**)
>   constructs `CvSize2D32f` structure.

See Also:

Size_

CvRect

struct `CvRect`

Stores coordinates of a rectangle.

>   int **x**
>>   x-coordinate of the top-left corner

>   int **y**
>>   y-coordinate of the top-left corner (sometimes bottom-left corner)

>   int **width**
>>   Width of the rectangle

>   int **height**
>>   Height of the rectangle

C: CvRect **cvRect**(int **x**, int **y**, int **width**, int **height**)
>   constructs `CvRect` structure.

See Also:

Rect_

CvScalar

struct `CvScalar`

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

>   double[4] **val**

See Also:

Scalar_

CvTermCriteria

struct `CvTermCriteria`

Termination criteria for iterative algorithms.

>   int **type**
>>   type of the termination criteria, one of:

>>   • `CV_TERMCRIT_ITER` - stop the algorithm after `max_iter` iterations at maximum.

>>   • `CV_TERMCRIT_EPS` - stop the algorithm after the achieved algorithm-dependent accuracy becomes lower than `epsilon`.

>>   • `CV_TERMCRIT_ITER+CV_TERMCRIT_EPS` - stop the algorithm after `max_iter` iterations or when the achieved accuracy is lower than `epsilon`, whichever comes the earliest.

int **max_iter**
    Maximum number of iterations

double **epsilon**
    Required accuracy

See Also:

TermCriteria

CvMat

struct CvMat

A multi-channel dense matrix.

int **type**
    CvMat signature (CV_MAT_MAGIC_VAL) plus type of the elements. Type of the matrix elements can be retrieved using CV_MAT_TYPE macro:

```
int type = CV_MAT_TYPE(matrix->type);
```

For description of possible matrix elements, see Mat.

int **step**
    Full row length in bytes

int* **refcount**
    Underlying data reference counter

union **data**
    Pointers to the actual matrix data:

    • ptr - pointer to 8-bit unsigned elements

    • s - pointer to 16-bit signed elements

    • i - pointer to 32-bit signed elements

    • fl - pointer to 32-bit floating-point elements

    • db - pointer to 64-bit floating-point elements

int **rows**
    Number of rows

int **cols**
    Number of columns

Matrix elements are stored row by row. Element (i, j) (i - 0-based row index, j - 0-based column index) of a matrix can be retrieved or modified using CV_MAT_ELEM macro:

```
uchar pixval = CV_MAT_ELEM(grayimg, uchar, i, j)
CV_MAT_ELEM(cameraMatrix, float, 0, 2) = image.width*0.5f;
```

To access multiple-channel matrices, you can use CV_MAT_ELEM(matrix, type, i, j*nchannels + channel_idx).

CvMat is now obsolete; consider using Mat instead.

CvMatND

struct CvMatND

Multi-dimensional dense multi-channel array.

    int **type**
        A `CvMatND` signature (CV_MATND_MAGIC_VAL) plus the type of elements. Type of the matrix elements can be retrieved using CV_MAT_TYPE macro:

```
int type = CV_MAT_TYPE(ndmatrix->type);
```

    int **dims**
        The number of array dimensions

    int* **refcount**
        Underlying data reference counter

    union **data**
        Pointers to the actual matrix data

        • ptr - pointer to 8-bit unsigned elements

        • s - pointer to 16-bit signed elements

        • i - pointer to 32-bit signed elements

        • fl - pointer to 32-bit floating-point elements

        • db - pointer to 64-bit floating-point elements

    array **dim**
        Arrays of pairs (array size along the i-th dimension, distance between neighbor elements along i-th dimension):

```
for(int i = 0; i < ndmatrix->dims; i++)
    printf("size[i] = %d, step[i] = %d\n", ndmatrix->dim[i].size, ndmatrix->dim[i].step);
```

CvMatND is now obsolete; consider using `Mat` instead.

CvSparseMat

struct CvSparseMat

Multi-dimensional sparse multi-channel array.

    int **type**
        A `CvSparseMat` signature (CV_SPARSE_MAT_MAGIC_VAL) plus the type of sparse matrix elements. Similarly to `CvMat` and `CvMatND`, use `CV_MAT_TYPE()` to retrieve type of the elements.

    int **dims**
        Number of dimensions

    int* **refcount**
        Underlying reference counter. Not used.

    CvSet* **heap**
        A pool of hash table nodes

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

int **hashsize**
> Size of the hash table

int[] **size**
> Array of dimension sizes

IplImage

struct IplImage

IPL image header

int **nSize**
> `sizeof(IplImage)`

int **ID**
> Version, always equals 0

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

int **alphaChannel**
> Ignored by OpenCV

int **depth**
> Channel depth in bits + the optional sign bit (`IPL_DEPTH_SIGN`). The supported depths are:
>
> - `IPL_DEPTH_8U` - unsigned 8-bit integer. Equivalent to `CV_8U` in matrix types.
> - `IPL_DEPTH_8S` - signed 8-bit integer. Equivalent to `CV_8S` in matrix types.
> - `IPL_DEPTH_16U` - unsigned 16-bit integer. Equivalent to `CV_16U` in matrix types.
> - `IPL_DEPTH_16S` - signed 8-bit integer. Equivalent to `CV_16S` in matrix types.
> - `IPL_DEPTH_32S` - signed 32-bit integer. Equivalent to `CV_32S` in matrix types.
> - `IPL_DEPTH_32F` - single-precision floating-point number. Equivalent to `CV_32F` in matrix types.
> - `IPL_DEPTH_64F` - double-precision floating-point number. Equivalent to `CV_64F` in matrix types.

char[] **colorModel**
> Ignored by OpenCV.

char[] **channelSeq**
> Ignored by OpenCV

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

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

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

int **width**
> Image width in pixels

int **height**
    Image height in pixels

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

IplImage* **maskROI**
    Must be NULL in OpenCV

void* **imageId**
    Must be NULL in OpenCV

void* **tileInfo**
    Must be NULL in OpenCV

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

char* **imageData**
    A pointer to the aligned image data. Do not assign imageData directly. Use `SetData()`.

int **widthStep**
    The size of an aligned image row, in bytes.

int[] **BorderMode**
    Border completion mode, ignored by OpenCV

int[] **BorderConst**
    Constant border value, ignored by OpenCV

char* **imageDataOrigin**
    A pointer to the origin of the image data (not necessarily aligned). This is used for image dealloca-
    tion.

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

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

CvArr

struct CvArr

This is the "metatype" used *only* as a function parameter. It denotes that the function accepts arrays of multiple types,
such as IplImage*, CvMat* or even CvSeq* sometimes. The particular array type is determined at runtime by analyz-
ing the first 4 bytes of the header. In C++ interface the role of `CvArr` is played by `InputArray` and `OutputArray`.

ClearND

Clears a specific array element.

C: void **cvClearND**(CvArr* **arr**, int* **idx**)

Python: cv.**ClearND**(arr, idx) → None

> **Parameters**

>> **arr** – Input array

> **idx** – Array of the element indices

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

CloneImage

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

C: IplImage* **cvCloneImage**(const IplImage* **image**)

Python: cv.**CloneImage**(image) → copy

> **Parameters**
>
> > **image** – The original image

CloneMat

Creates a full matrix copy.

C: CvMat* **cvCloneMat**(const CvMat* **mat**)

Python: cv.**CloneMat**(mat) → copy

> **Parameters**
>
> > **mat** – Matrix to be copied

Creates a full copy of a matrix and returns a pointer to the copy. Note that the matrix copy is compacted, that is, it will not have gaps between rows.

CloneMatND

Creates full copy of a multi-dimensional array and returns a pointer to the copy.

C: CvMatND* **cvCloneMatND**(const CvMatND* **mat**)

Python: cv.**CloneMatND**(mat) → copy

> **Parameters**
>
> > **mat** – Input array

CloneSparseMat

Creates full copy of sparse array.

C: CvSparseMat* **cvCloneSparseMat**(const CvSparseMat* **mat**)

> **Parameters**
>
> > **mat** – Input array

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

ConvertScale

Converts one array to another with optional linear transformation.

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

Python: cv.**ConvertScale**(src, dst, scale=1.0, shift=0.0) → None

Python: cv.**Convert**(src, dst) → None

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

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

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

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

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

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

Copy

Copies one array to another.

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

Python: cv.**Copy**(src, dst, mask=None) → None

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

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

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

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

CreateData

Allocates array data

C: void **cvCreateData**(CvArr* **arr**)

Python: cv.**CreateData**(arr) → None

> **Parameters**

>> **arr** – Array header

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

CreateImage

Creates an image header and allocates the image data.

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

Python: cv.**CreateImage**(size, depth, channels) → image

> **Parameters**

>> **size** – Image width and height

>> **depth** – Bit depth of image elements. See `IplImage` for valid depths.

>> **channels** – Number of channels per pixel. See `IplImage` for details. This function only creates images with interleaved channels.

This function call is equivalent to the following code:

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

CreateImageHeader

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

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

Python: cv.**CreateImageHeader**(size, depth, channels) → image

> **Parameters**

>> **size** – Image width and height

>> **depth** – Image depth (see `CreateImage()`)

>> **channels** – Number of channels (see `CreateImage()`)

CreateMat

Creates a matrix header and allocates the matrix data.

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

Python: cv.**CreateMat**(rows, cols, type) → mat

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

The function call is equivalent to the following code:

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

CreateMatHeader

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

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

Python: cv.**CreateMatHeader**(rows, cols, type) → mat

> **Parameters**
>
>> **rows** – Number of rows in the matrix
>>
>> **cols** – Number of columns in the matrix
>>
>> **type** – Type of the matrix elements, see CreateMat()

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

CreateMatND

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

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

Python: cv.**CreateMatND**(dims, type) → None

> **Parameters**
>
>> **dims** – Number of array dimensions. This must not exceed CV_MAX_DIM (32 by default, but can be changed at build time).
>>
>> **sizes** – Array of dimension sizes.
>>
>> **type** – Type of array elements, see CreateMat() .

This function call is equivalent to the following code:

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

CreateMatNDHeader

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

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

Python: cv.**CreateMatNDHeader**(dims, type) → None

>> **Parameters**

>>> **dims** – Number of array dimensions

>>> **sizes** – Array of dimension sizes

>>> **type** – Type of array elements, see `CreateMat()`

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

CreateSparseMat

Creates sparse array.

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

>> **Parameters**

>>> **dims** – Number of array dimensions. In contrast to the dense matrix, the number of dimensions is practically unlimited (up to 2^{16}).

>>> **sizes** – Array of dimension sizes

>>> **type** – Type of array elements. The same as for CvMat

The function allocates a multi-dimensional sparse array. Initially the array contain no elements, that is `PtrND()` and other related functions will return 0 for every index.

CrossProduct

Calculates the cross product of two 3D vectors.

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

Python: cv.**CrossProduct**(src1, src2, dst) → None

>> **Parameters**

>>> **src1** – The first source vector

>>> **src2** – The second source vector

>>> **dst** – The destination vector

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

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

or:

$$\mathtt{dst}_1 = \mathtt{src1}_2 \mathtt{src2}_3 - \mathtt{src1}_3 \mathtt{src2}_2$$
$$\mathtt{dst}_2 = \mathtt{src1}_3 \mathtt{src2}_1 - \mathtt{src1}_1 \mathtt{src2}_3$$
$$\mathtt{dst}_3 = \mathtt{src1}_1 \mathtt{src2}_2 - \mathtt{src1}_2 \mathtt{src2}_1$$

DotProduct

Calculates the dot product of two arrays in Euclidian metrics.

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

Python: cv.**DotProduct**(src1, src2) → double

> **Parameters**
>
> > **src1** – The first source array
> >
> > **src2** – The second source array

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

$$src1 \bullet src2 = \sum_{I} (src1(I)src2(I))$$

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

Get?D

C: CvScalar **cvGet1D**(const CvArr* **arr**, int **idx0**)

C: CvScalar **cvGet2D**(const CvArr* **arr**, int **idx0**, int **idx1**)

C: CvScalar **cvGet3D**(const CvArr* **arr**, int **idx0**, int **idx1**, int **idx2**)

C: CvScalar **cvGetND**(const CvArr* **arr**, int* **idx**)

Python: cv.**Get1D**(arr, idx) → scalar

Python: cv.**Get2D**(arr, idx0, idx1) → scalar

Python: cv.**Get3D**(arr, idx0, idx1, idx2) → scalar

Python: cv.**GetND**(arr, indices) → scalar
> Return a specific array element.

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

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

GetCol(s)

Returns one of more array columns.

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

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

Python: cv.**GetCol**(arr, col) → submat

Python: cv.**GetCols**(arr, startCol, endCol) → submat

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

The functions return the header, corresponding to a specified column span of the input array. That is, no data is copied. Therefore, any modifications of the submatrix will affect the original array. If you need to copy the columns, use `CloneMat()`. `cvGetCol(arr, submat, col)` is a shortcut for `cvGetCols(arr, submat, col, col+1)`.

GetDiag

Returns one of array diagonals.

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

Python: cv.**GetDiag**(arr, diag=0) → submat

>
> **Parameters**
>
>> **arr** – Input array
>>
>> **submat** – Pointer to the resulting sub-array header
>>
>> **diag** – Index of the array diagonal. Zero value corresponds to the main diagonal, -1 corresponds to the diagonal above the main, 1 corresponds to the diagonal below the main, and so forth.

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

GetDims

Return number of array dimensions

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

Python: cv.**GetDims**(arr) → list

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

The function returns the array dimensionality and the array of dimension sizes. In the case of `IplImage` or *CvMat* it always returns 2 regardless of number of image/matrix rows. For example, the following code calculates total number of array elements:

```
int sizes[CV_MAX_DIM];
int i, total = 1;
int dims = cvGetDims(arr, size);
```

```
for(i = 0; i < dims; i++ )
    total *= sizes[i];
```

GetDimSize

Returns array size along the specified dimension.

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

> **Parameters**
>
> > **arr** – Input array
> >
> > **index** – Zero-based dimension index (for matrices 0 means number of rows, 1 means number of columns; for images 0 means height, 1 means width)

GetElemType

Returns type of array elements.

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

Python: cv.**GetElemType**(arr) → int

> **Parameters**
>
> > **arr** – Input array

The function returns type of the array elements. In the case of `IplImage` the type is converted to `CvMat`-like representation. For example, if the image has been created as:

```
IplImage* img = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 3);
```

The code `cvGetElemType(img)` will return `CV_8UC3`.

GetImage

Returns image header for arbitrary array.

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

Python: cv.**GetImage**(arr) → iplimage

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

The function returns the image header for the input array that can be a matrix (`CvMat`) or image (`IplImage`). In the case of an image the function simply returns the input pointer. In the case of `CvMat` it initializes an `imageHeader` structure with the parameters of the input matrix. Note that if we transform `IplImage` to `CvMat` using `GetMat()` and then transform `CvMat` back to IplImage using this function, we will get different headers if the ROI is set in the original image.

GetImageCOI

Returns the index of the channel of interest.

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

Python: cv.**GetImageCOI**(image) → channel

> **Parameters**

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

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

GetImageROI

Returns the image ROI.

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

Python: cv.**GetImageROI**(image) → CvRect

> **Parameters**

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

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

GetMat

Returns matrix header for arbitrary array.

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

Python: cv.**GetMat**(arr, allowND=0) → cvmat

> **Parameters**

> > **arr** – Input array

> > **header** – Pointer to `CvMat` structure used as a temporary buffer

> > **coi** – Optional output parameter for storing COI

> > **allowND** – If non-zero, the function accepts multi-dimensional dense arrays (CvMatND*)
> > and returns 2D matrix (if CvMatND has two dimensions) or 1D matrix (when CvMatND
> > has 1 dimension or more than 2 dimensions). The `CvMatND` array must be continuous.

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

The function provides an easy way to handle both types of arrays - `IplImage` and `CvMat` using the same code. Input array must have non-zero data pointer, otherwise the function will report an error.

See Also:

`GetImage()`, `cvarrToMat()`.

Note: If the input array is `IplImage` with planar data layout and COI set, the function returns the pointer to the selected plane and `COI == 0`. This feature allows user to process `IplImage` strctures with planar data layout, even though OpenCV does not support such images.

GetNextSparseNode

Returns the next sparse matrix element

C: CvSparseNode* **cvGetNextSparseNode**(CvSparseMatIterator* **matIterator**)

>> **Parameters**

>>>> **matIterator** – Sparse array iterator

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

```
// print all the non-zero sparse matrix elements and compute their sum
double sum = 0;
int i, dims = cvGetDims(sparsemat);
CvSparseMatIterator it;
CvSparseNode* node = cvInitSparseMatIterator(sparsemat, &it);

for(; node != 0; node = cvGetNextSparseNode(&it))
{
    /* get pointer to the element indices */
    int* idx = CV_NODE_IDX(array, node);
    /* get value of the element (assume that the type is CV_32FC1) */
    float val = *(float*)CV_NODE_VAL(array, node);
    printf("M");
    for(i = 0; i < dims; i++ )
        printf("[%d]", idx[i]);
    printf("=%g\n", val);

    sum += val;
}

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

GetRawData

Retrieves low-level information about the array.

C: void **cvGetRawData**(const CvArr* **arr**, uchar** **data**, int* **step**=NULL, CvSize* **roiSize**=NULL)

>> **Parameters**

>>>> **arr** – Array header

>>>> **data** – Output pointer to the whole image origin or ROI origin if ROI is set

>>>> **step** – Output full row length in bytes

>>>> **roiSize** – Output ROI size

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

The following example shows how to get access to array elements. It computes absolute values of the array elements

```
float* data;
int step;
CvSize size;

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

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

GetReal?D

Return a specific element of single-channel 1D, 2D, 3D or nD array.

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

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

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

C: double **cvGetRealND**(const CvArr* **arr**, int* **idx**)

Python: cv.**GetReal1D**(arr, idx0) → float

Python: cv.**GetReal2D**(arr, idx0, idx1) → float

Python: cv.**GetReal3D**(arr, idx0, idx1, idx2) → float

Python: cv.**GetRealND**(arr, idx) → float

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

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

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

GetRow(s)

Returns array row or row span.

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

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

Python: cv.**GetRow**(arr, row) → submat

Python: cv.**GetRows**(arr, startRow, endRow, deltaRow=1) → submat

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

The functions return the header, corresponding to a specified row/row span of the input array. `cvGetRow(arr, submat, row)` is a shortcut for `cvGetRows(arr, submat, row, row+1)`.

GetSize

Returns size of matrix or image ROI.

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

Python: cv.**GetSize**(arr)-> (width, height)

> **Parameters**
>> **arr** – array header

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

GetSubRect

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

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

Python: cv.**GetSubRect**(arr, rect) → submat

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

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

DecRefData

Decrements an array data reference counter.

C: void **cvDecRefData**(CvArr* **arr**)

> **Parameters**
>> **arr** – Pointer to an array header

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

IncRefData

Increments array data reference counter.

C: int **cvIncRefData**(CvArr* **arr**)

> **Parameters**
>
> > **arr** – Array header

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

InitImageHeader

Initializes an image header that was previously allocated.

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

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

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

InitMatHeader

Initializes a pre-allocated matrix header.

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

> **Parameters**
>
> > **mat** – A pointer to the matrix header to be initialized
> >
> > **rows** – Number of rows in the matrix
> >
> > **cols** – Number of columns in the matrix
> >
> > **type** – Type of the matrix elements, see `CreateMat()` .
> >
> > **data** – Optional: data pointer assigned to the matrix header

step – Optional: full row width in bytes of the assigned data. By default, the minimal possible step is used which assumes there are no gaps between subsequent rows of the matrix.

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

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

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

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

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

cvMatMulAdd(&Ma, &Mb, 0, &Mc);
// the c array now contains the product of a (3x4) and b (4x3)
```

InitMatNDHeader

Initializes a pre-allocated multi-dimensional array header.

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

> **Parameters**

>> **mat** – A pointer to the array header to be initialized

>> **dims** – The number of array dimensions

>> **sizes** – An array of dimension sizes

>> **type** – Type of array elements, see `CreateMat()`

>> **data** – Optional data pointer assigned to the matrix header

InitSparseMatIterator

Initializes sparse array elements iterator.

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

> **Parameters**

>> **mat** – Input array

>> **matIterator** – Initialized iterator

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

Mat

Initializes matrix header (lightweight variant).

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

>> **Parameters**

>>> **rows** – Number of rows in the matrix

>>> **cols** – Number of columns in the matrix

>>> **type** – Type of the matrix elements - see `CreateMat()`

>>> **data** – Optional data pointer assigned to the matrix header

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

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

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

Ptr?D

Return pointer to a particular array element.

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

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

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

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

>> **Parameters**

>>> **arr** – Input array

>>> **idx0** – The first zero-based component of the element index

>>> **idx1** – The second zero-based component of the element index

>>> **idx2** – The third zero-based component of the element index

>>> **idx** – Array of the element indices

>>> **type** – Optional output parameter: type of matrix elements

>>> **createNode** – Optional input parameter for sparse matrices. Non-zero value of the parameter means that the requested element is created if it does not exist already.

>>> **precalcHashval** – Optional input parameter for sparse matrices. If the pointer is not NULL, the function does not recalculate the node hash value, but takes it from the specified location. It is useful for speeding up pair-wise operations (TODO: provide an example)

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

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

All these as well as other functions accessing array elements (`GetND()` , `GetRealND()` , `Set()` , `SetND()` , `SetRealND()`) raise an error in case if the element index is out of range.

ReleaseData

Releases array data.

C: void **cvReleaseData**(CvArr* **arr**)

> **Parameters**
>
>> **arr** – Array header

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

ReleaseImage

Deallocates the image header and the image data.

C: void **cvReleaseImage**(IplImage** **image**)

> **Parameters**
>
>> **image** – Double pointer to the image header

This call is a shortened form of

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

ReleaseImageHeader

Deallocates an image header.

C: void **cvReleaseImageHeader**(IplImage** **image**)

> **Parameters**
>
>> **image** – Double pointer to the image header

This call is an analogue of

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

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

ReleaseMat

Deallocates a matrix.

C: void **cvReleaseMat**(CvMat** **mat**)

> **Parameters**
>
> > **mat** – Double pointer to the matrix

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

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

ReleaseMatND

Deallocates a multi-dimensional array.

C: void **cvReleaseMatND**(CvMatND** **mat**)

> **Parameters**
>
> > **mat** – Double pointer to the array

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

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

ReleaseSparseMat

Deallocates sparse array.

C: void **cvReleaseSparseMat**(CvSparseMat** **mat**)

> **Parameters**
>
> > **mat** – Double pointer to the array

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

ResetImageROI

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

C: void **cvResetImageROI**(IplImage* **image**)

Python: cv.**ResetImageROI**(image) → None

> **Parameters**
>
> > **image** – A pointer to the image header

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

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

Reshape

Changes shape of matrix/image without copying data.

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

Python: cv.**Reshape**(arr, newCn, newRows=0) → cvmat

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

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

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

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

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

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

ReshapeMatND

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

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

Python: cv.**ReshapeMatND**(arr, newCn, newDims) → cvmat

> **Parameters**
>
> > **arr** – Input array
> >
> > **sizeofHeader** – Size of output header to distinguish between IplImage, CvMat and CvMatND output headers
> >
> > **header** – Output header to be filled
> >
> > **newCn** – New number of channels. newCn = 0 means that the number of channels remains unchanged.

> > **newDims** – New number of dimensions. `newDims = 0` means that the number of dimensions remains the same.
>
> > **newSizes** – Array of new dimension sizes. Only `newDims-1` values are used, because the total number of elements must remain the same. Thus, if `newDims = 1`, `newSizes` array is not used.

The function is an advanced version of `Reshape()` that can work with multi-dimensional arrays as well (though it can work with ordinary images and matrices) and change the number of dimensions.

Below are the two samples from the `Reshape()` description rewritten using `ReshapeMatND()` :

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


...


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

Set

Sets every element of an array to a given value.

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

Python: cv.**Set**(arr, value, mask=None) → None

> > **Parameters**

> > > **arr** – The destination array

> > > **value** – Fill value

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

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

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

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

Set?D

Change the particular array element.

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

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

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

C: void **cvSetND**(CvArr* **arr**, int* **idx**, CvScalar **value**)

Python: cv.**Set1D**(arr, idx, value) → None

Python: cv.**Set2D**(arr, idx0, idx1, value) → None

Python: cv.**Set3D**(arr, idx0, idx1, idx2, value) → None

Python: cv.**SetND**(arr, indices, value) → None

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

The functions assign the new value to a particular array element. In the case of a sparse array the functions create the node if it does not exist yet.

SetData

Assigns user data to the array header.

C: void **cvSetData**(CvArr* **arr**, void* **data**, int **step**)

Python: cv.**SetData**(arr, data, step) → None

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

The function assigns user data to the array header. Header should be initialized before using `cvCreateMatHeader()`, `cvCreateImageHeader()`, `cvCreateMatNDHeader()`, `cvInitMatHeader()`, `cvInitImageHeader()` or `cvInitMatNDHeader()`.

SetImageCOI

Sets the channel of interest in an IplImage.

C: void **cvSetImageCOI**(IplImage* **image**, int **coi**)

Python: cv.**SetImageCOI**(image, coi) → None

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

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

SetImageROI

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

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

Python: cv.**SetImageROI**(image, rect) → None

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

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

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

SetReal?D

Change a specific array element.

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

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

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

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

Python: cv.**SetReal1D**(arr, idx, value) → None

Python: cv.**SetReal2D**(arr, idx0, idx1, value) → None

Python: cv.**SetReal3D**(arr, idx0, idx1, idx2, value) → None

Python: cv.**SetRealND**(arr, indices, value) → None

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

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

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

SetZero

Clears the array.

C: void **cvSetZero**(CvArr* **arr**)

Python: cv.**SetZero**(arr) → None

>>> **Parameters**

>>>> **arr** – Array to be cleared

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

mGet

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

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

Python: cv.**mGet**(mat, row, col) → double

>>> **Parameters**

>>>> **mat** – Input matrix

>>>> **row** – The zero-based index of row

>>>> **col** – The zero-based index of column

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

mSet

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

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

Python: cv.**mSet**(mat, row, col, value) → None

>>> **Parameters**

>>>> **mat** – The matrix

>>>> **row** – The zero-based index of row

>>>> **col** – The zero-based index of column

>>>> **value** – The new value of the matrix element

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

SetIPLAllocators

Makes OpenCV use IPL functions for allocating IplImage and IplROI structures.

C: void **cvSetIPLAllocators**(Cv_iplCreateImageHeader **create_header**, Cv_iplAllocateImageData **allocate_data**, Cv_iplDeallocate **deallocate**, Cv_iplCreateROI **create_roi**, Cv_iplCloneImage **clone_image**)

Normally, the function is not called directly. Instead, a simple macro CV_TURN_ON_IPL_COMPATIBILITY() is used that calls cvSetIPLAllocators and passes there pointers to IPL allocation functions.

```
...
CV_TURN_ON_IPL_COMPATIBILITY()
...
```

RNG

Initializes a random number generator state.

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

Python: cv.**RNG**(seed=-1LL) → CvRNG

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

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

See Also:

the C++ class `RNG` replaced `CvRNG`.

RandArr

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

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

Python: cv.**RandArr**(rng, arr, distType, param1, param2) → None

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

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

See Also:

`randu()`, `randn()`, `RNG::fill()`.

RandInt

Returns a 32-bit unsigned integer and updates RNG.

C: unsigned int **cvRandInt**(CvRNG* **rng**)

Python: cv.**RandInt**(rng) → unsigned

> **Parameters**
>
> > **rng** – CvRNG state initialized by `RNG()`.

The function returns a uniformly-distributed random 32-bit unsigned integer and updates the RNG state. It is similar to the rand() function from the C runtime library, except that OpenCV functions always generates a 32-bit random number, regardless of the platform.

RandReal

Returns a floating-point random number and updates RNG.

C: double **cvRandReal**(CvRNG* **rng**)

Python: cv.**RandReal**(rng) → double

> **Parameters**
>
> > **rng** – RNG state initialized by `RNG()`

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

fromarray

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

Python: cv.**fromarray**(object, allowND=False) → CvMat

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

If the object supports the array interface , return a `CvMat` or `CvMatND`, depending on `allowND` flag:

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

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

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

Note: In the new Python wrappers (**cv2** module) the function is not needed, since cv2 can process Numpy arrays (and this is the only supported array type).

2.3 Dynamic Structures

The section describes OpenCV 1.x API for creating growable sequences and other dynamic data structures allocated in `CvMemStorage`. If you use the new C++, Python, Java etc interface, you will unlikely need this functionality. Use `std::vector` or other high-level data structures.

CvMemStorage

struct `CvMemStorage`

A storage for various OpenCV dynamic data structures, such as `CvSeq`, `CvSet` etc.

> CvMemBlock* **bottom**
> > the first memory block in the double-linked list of blocks
>
> CvMemBlock* **top**
> > the current partially allocated memory block in the list of blocks
>
> CvMemStorage* **parent**
> > the parent storage (if any) from which the new memory blocks are borrowed.
>
> int **free_space**
> > number of free bytes in the `top` block
>
> int **block_size**
> > the total size of the memory blocks

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

A new memory buffer that may be allocated explicitly by `MemStorageAlloc()` function or implicitly by higher-level functions, such as `SeqPush()`, `GraphAddEdge()` etc.

The buffer is put in the end of already allocated space in the `top` memory block, if there is enough free space. After allocation, `free_space` is decreased by the size of the allocated buffer plus some padding to keep the proper alignment. When the allocated buffer does not fit into the available portion of `top`, the next storage block from the list is taken as `top` and `free_space` is reset to the whole block size prior to the allocation.

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

CvMemBlock

struct `CvMemBlock`

The structure `CvMemBlock` represents a single block of memory storage. The actual data in the memory blocks follows the header.

CvMemStoragePos

struct CvMemStoragePos

The structure stores the position in the memory storage. It is used by SaveMemStoragePos() and RestoreMemStoragePos().

CvSeq

struct CvSeq

Dynamically growing sequence.

> int **flags**
>> sequence flags, including the sequence signature (CV_SEQ_MAGIC_VAL or CV_SET_MAGIC_VAL), type of the elements and some other information about the sequence.

> int **header_size**
>> size of the sequence header. It should be sizeof(CvSeq) at minimum. See CreateSeq().

> CvSeq* **h_prev**

> CvSeq* **h_next**

> CvSeq* **v_prev**

> CvSeq* **v_next**
>> pointers to another sequences in a sequence tree. Sequence trees are used to store hierarchical contour structures, retrieved by FindContours()

> int **total**
>> the number of sequence elements

> int **elem_size**
>> size of each sequence element in bytes

> CvMemStorage* **storage**
>> memory storage where the sequence resides. It can be a NULL pointer.

> CvSeqBlock* **first**
>> pointer to the first data block

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

CvSlice

struct CvSlice
> A sequence slice. In C++ interface the class Range should be used instead.

> int **start_index**
>> inclusive start index of the sequence slice

> int **end_index**
>> exclusive end index of the sequence slice

There are helper functions to construct the slice and to compute its length:

C: CvSlice **cvSlice**(int **start**, int **end**)

```
#define CV_WHOLE_SEQ_END_INDEX 0x3fffffff
#define CV_WHOLE_SEQ  cvSlice(0, CV_WHOLE_SEQ_END_INDEX)
```

C: int **cvSliceLength**(CvSlice **slice**, const CvSeq* **seq**)
    Calculates the sequence slice length

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

1. SliceLength() is called to determine the length of the slice,

2. start_index of the slice is normalized similarly to the argument of GetSeqElem() (i.e., negative indices are allowed). The actual slice to process starts at the normalized start_index and lasts SliceLength() elements (again, assuming the sequence is a circular structure).

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

CvSet

struct CvSet

The structure CvSet is a base for OpenCV 1.x sparse data structures. It is derived from CvSeq and includes an additional member free_elems - a list of free nodes. Every node of the set, whether free or not, is an element of the underlying sequence. While there are no restrictions on elements of dense sequences, the set (and derived structures) elements must start with an integer field and be able to fit CvSetElem structure, because these two fields (an integer followed by a pointer) are required for the organization of a node set with the list of free nodes. If a node is free, the flags field is negative (the most-significant bit, or MSB, of the field is set), and the next_free points to the next free node (the first free node is referenced by the free_elems field of CvSet). And if a node is occupied, the flags field is positive and contains the node index that may be retrieved using the (set_elem->flags & CV_SET_ELEM_IDX_MASK) expressions, the rest of the node content is determined by the user. In particular, the occupied nodes are not linked as the free nodes are, so the second field can be used for such a link as well as for some different purpose. The macro CV_IS_SET_ELEM(set_elem_ptr) can be used to determined whether the specified node is occupied or not.

Initially the set and the free node list are empty. When a new node is requested from the set, it is taken from the list of free nodes, which is then updated. If the list appears to be empty, a new sequence block is allocated and all the nodes within the block are joined in the list of free nodes. Thus, the total field of the set is the total number of nodes both occupied and free. When an occupied node is released, it is added to the list of free nodes. The node released last will be occupied first.

CvSet is used to represent graphs (CvGraph), sparse multi-dimensional arrays (CvSparseMat), and planar subdivisions (CvSubdiv2D).

CvGraph

struct CvGraph

The structure CvGraph is a base for graphs used in OpenCV 1.x. It inherits from CvSet, that is, it is considered as a set of vertices. Besides, it contains another set as a member, a set of graph edges. Graphs in OpenCV are represented using adjacency lists format.

CvGraphScanner

struct CvGraphScanner

The structure CvGraphScanner is used for depth-first graph traversal. See discussion of the functions below.

CvTreeNodeIterator

struct CvTreeNodeIterator

The structure CvTreeNodeIterator is used to traverse trees of sequences.

ClearGraph

Clears a graph.

C: void **cvClearGraph**(CvGraph* **graph**)

       Parameters

              graph – Graph

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

ClearMemStorage

Clears memory storage.

C: void **cvClearMemStorage**(CvMemStorage* **storage**)

       Parameters

              storage – Memory storage

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

ClearSeq

Clears a sequence.

C: void **cvClearSeq**(CvSeq* **seq**)

       Parameters

              seq – Sequence

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

Note: It is impossible to deallocate a sequence, i.e. free space in the memory storage occupied by the sequence. Instead, call `ClearMemStorage()` or `ReleaseMemStorage()` from time to time somewhere in a top-level processing loop.

ClearSet

Clears a set.

C: void **cvClearSet**(CvSet* **setHeader**)

> **Parameters**
>
> > **setHeader** – Cleared set

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

CloneGraph

Clones a graph.

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

> **Parameters**
>
> > **graph** – The graph to copy
> >
> > **storage** – Container for the copy

The function creates a full copy of the specified graph. If the graph vertices or edges have pointers to some external data, it can still be shared between the copies. The vertex and edge indices in the new graph may be different from the original because the function defragments the vertex and edge sets.

CloneSeq

Creates a copy of a sequence.

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

Python: cv.**CloneSeq**(seq, storage) → None

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

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

The call cvCloneSeq(seq, storage) is equivalent to cvSeqSlice(seq, CV_WHOLE_SEQ, storage, 1).

CreateChildMemStorage

Creates child memory storage.

C: CvMemStorage* **cvCreateChildMemStorage**(CvMemStorage* **parent**)

> **Parameters**

> **parent** – Parent memory storage

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

Child storage is useful in the following situation. Imagine that the user needs to process dynamic data residing in a given storage area and put the result back to that same storage area. With the simplest approach, when temporary data is resided in the same storage area as the input and output data, the storage area will look as follows after processing:

Dynamic data processing without using child storage
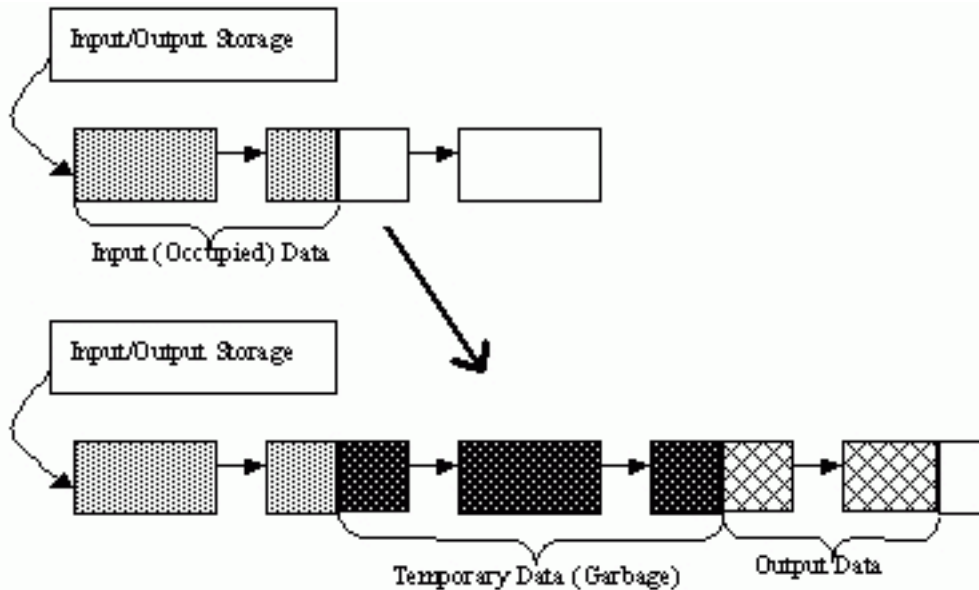
That is, garbage appears in the middle of the storage. However, if one creates a child memory storage at the beginning of processing, writes temporary data there, and releases the child storage at the end, no garbage will appear in the source/destination storage:

Dynamic data processing using a child storage
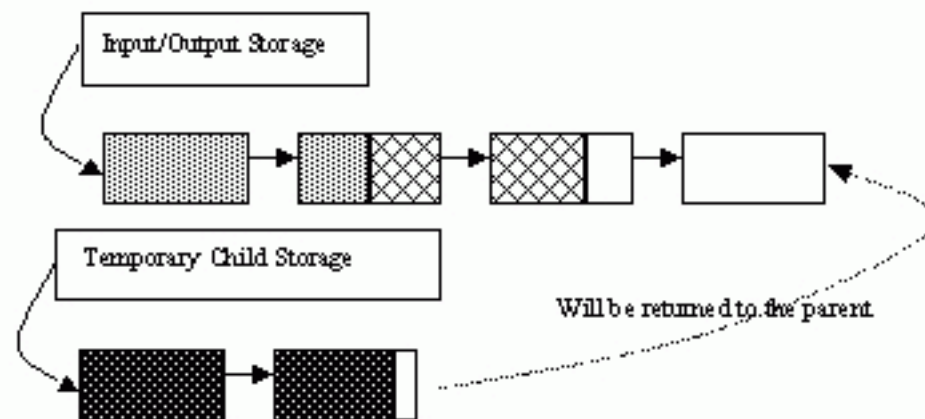
CreateGraph

Creates an empty graph.

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

> **Parameters**
>
> > **graph_flags** – Type of the created graph. Usually, it is either `CV_SEQ_KIND_GRAPH` for generic unoriented graphs and `CV_SEQ_KIND_GRAPH | CV_GRAPH_FLAG_ORIENTED` for generic oriented graphs.
> >
> > **header_size** – Graph header size; may not be less than `sizeof(CvGraph)`
> >
> > **vtx_size** – Graph vertex size; the custom vertex structure must start with `CvGraphVtx` (use `CV_GRAPH_VERTEX_FIELDS()`)
> >
> > **edge_size** – Graph edge size; the custom edge structure must start with `CvGraphEdge` (use `CV_GRAPH_EDGE_FIELDS()`)
> >
> > **storage** – The graph container

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

CreateGraphScanner

Creates structure for depth-first graph traversal.

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

> **Parameters**
>
> > **graph** – Graph
> >
> > **vtx** – Initial vertex to start from. If NULL, the traversal starts from the first vertex (a vertex with the minimal index in the sequence of vertices).
> >
> > **mask** – Event mask indicating which events are of interest to the user (where `NextGraphItem()` function returns control to the user) It can be `CV_GRAPH_ALL_ITEMS` (all events are of interest) or a combination of the following flags:
> >
> > – **CV_GRAPH_VERTEX** stop at the graph vertices visited for the first time
> >
> > – **CV_GRAPH_TREE_EDGE** stop at tree edges (`tree edge` is the edge connecting the last visited vertex and the vertex to be visited next)
> >
> > – **CV_GRAPH_BACK_EDGE** stop at back edges (`back edge` is an edge connecting the last visited vertex with some of its ancestors in the search tree)
> >
> > – **CV_GRAPH_FORWARD_EDGE** stop at forward edges (`forward edge` is an edge conecting the last visited vertex with some of its descendants in the search tree. The forward edges are only possible during oriented graph traversal)
> >
> > – **CV_GRAPH_CROSS_EDGE** stop at cross edges (`cross edge` is an edge connecting different search trees or branches of the same tree. The `cross edges` are only possible during oriented graph traversal)
> >
> > – **CV_GRAPH_ANY_EDGE** stop at any edge (`tree, back, forward` , and `cross edges`)

– **CV_GRAPH_NEW_TREE** stop in the beginning of every new search tree. When the traversal procedure visits all vertices and edges reachable from the initial vertex (the visited vertices together with tree edges make up a tree), it searches for some unvisited vertex in the graph and resumes the traversal process from that vertex. Before starting a new tree (including the very first tree when `cvNextGraphItem` is called for the first time) it generates a `CV_GRAPH_NEW_TREE` event. For unoriented graphs, each search tree corresponds to a connected component of the graph.

– **CV_GRAPH_BACKTRACKING** stop at every already visited vertex during backtracking - returning to already visited vertexes of the traversal tree.

The function creates a structure for depth-first graph traversal/search. The initialized structure is used in the `NextGraphItem()` function - the incremental traversal procedure.

CreateMemStorage

Creates memory storage.

C: CvMemStorage* **cvCreateMemStorage**(int **blockSize**=0)

Python: cv.**CreateMemStorage**(blockSize=0) → memstorage

>   **Parameters**

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

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

CreateSeq

Creates a sequence.

C: CvSeq* **cvCreateSeq**(int **seqFlags**, int **headerSize**, int **elemSize**, CvMemStorage* **storage**)

>   **Parameters**

>>   **seqFlags** – Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be set to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.

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

>>   **elemSize** – Size of the sequence elements in bytes. The size must be consistent with the sequence type. For example, for a sequence of points to be created, the element type `CV_SEQ_ELTYPE_POINT` should be specified and the parameter `elemSize` must be equal to `sizeof(CvPoint)` .

>>   **storage** – Sequence location

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

CreateSet

Creates an empty set.

C: CvSet* **cvCreateSet**(int **set_flags**, int **header_size**, int **elem_size**, CvMemStorage* **storage**)

> **Parameters**
>
>> **set_flags** – Type of the created set
>>
>> **header_size** – Set header size; may not be less than `sizeof(CvSet)`
>>
>> **elem_size** – Set element size; may not be less than `CvSetElem`
>>
>> **storage** – Container for the set

The function creates an empty set with a specified header size and element size, and returns the pointer to the set. This function is just a thin layer on top of `CreateSeq()`.

CvtSeqToArray

Copies a sequence to one continuous block of memory.

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

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

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

EndWriteSeq

Finishes the process of writing a sequence.

C: CvSeq* **cvEndWriteSeq**(CvSeqWriter* **writer**)

> **Parameters**
>
>> **writer** – Writer state

The function finishes the writing process and returns the pointer to the written sequence. The function also truncates the last incomplete sequence block to return the remaining part of the block to memory storage. After that, the sequence can be read and modified safely. See `StartWriteSeq()` and `StartAppendToSeq()`

FindGraphEdge

Finds an edge in a graph.

C: CvGraphEdge* **cvFindGraphEdge**(const CvGraph* **graph**, int **start_idx**, int **end_idx**)

> **Parameters**
>
>> **graph** – Graph
>>
>> **start_idx** – Index of the starting vertex of the edge

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

```
#define cvGraphFindEdge cvFindGraphEdge
```

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

FindGraphEdgeByPtr

Finds an edge in a graph by using its pointer.

C: CvGraphEdge* **cvFindGraphEdgeByPtr**(const CvGraph* **graph**, const CvGraphVtx* **startVtx**, const Cv-GraphVtx* **endVtx**)

>> **Parameters**

>>> **graph** – Graph

>>> **startVtx** – Pointer to the starting vertex of the edge

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

```
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

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

FlushSeqWriter

Updates sequence headers from the writer.

C: void **cvFlushSeqWriter**(CvSeqWriter* **writer**)

>> **Parameters**

>>> **writer** – Writer state

The function is intended to enable the user to read sequence elements, whenever required, during the writing process, e.g., in order to check specific conditions. The function updates the sequence headers to make reading from the sequence possible. The writer is not closed, however, so that the writing process can be continued at any time. If an algorithm requires frequent flushes, consider using `SeqPush()` instead.

GetGraphVtx

Finds a graph vertex by using its index.

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

>> **Parameters**

>>> **graph** – Graph

>>> **vtx_idx** – Index of the vertex

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

GetSeqElem

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

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

>> **Parameters**

>>> **seq** – Sequence

>>> **index** – Index of element

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

The function finds the element with the given index in the sequence and returns the pointer to it. If the element is not found, the function returns 0. The function supports negative indices, where -1 stands for the last sequence element, -2 stands for the one before last, etc. If the sequence is most likely to consist of a single sequence block or the desired element is likely to be located in the first block, then the macro CV_GET_SEQ_ELEM(elemType, seq, index) should be used, where the parameter elemType is the type of sequence elements (CvPoint for example), the parameter seq is a sequence, and the parameter index is the index of the desired element. The macro checks first whether the desired element belongs to the first block of the sequence and returns it if it does; otherwise the macro calls the main function GetSeqElem . Negative indices always cause the GetSeqElem() call. The function has O(1) time complexity assuming that the number of blocks is much smaller than the number of elements.

GetSeqReaderPos

Returns the current reader position.

C: int **cvGetSeqReaderPos**(CvSeqReader* **reader**)

>> **Parameters**

>>> **reader** – Reader state

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

GetSetElem

Finds a set element by its index.

C: CvSetElem* **cvGetSetElem**(const CvSet* **setHeader**, int **index**)

>> **Parameters**

>>> **setHeader** – Set

>>> **index** – Index of the set element within a sequence

The function finds a set element by its index. The function returns the pointer to it or 0 if the index is invalid or the corresponding node is free. The function supports negative indices as it uses GetSeqElem() to locate the node.

GraphAddEdge

Adds an edge to a graph.

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

>> **Parameters**

> > > **graph** – Graph
> > >
> > > **start_idx** – Index of the starting vertex of the edge
> > >
> > > **end_idx** – Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.
> > >
> > > **edge** – Optional input parameter, initialization data for the edge
> > >
> > > **inserted_edge** – Optional output parameter to contain the address of the inserted edge

The function connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same, or there is some other critical situation. In the latter case (i.e., when the result is negative), the function also reports an error by default.

GraphAddEdgeByPtr

Adds an edge to a graph by using its pointer.

C: int **cvGraphAddEdgeByPtr**(CvGraph* **graph**, CvGraphVtx* **start_vtx**, CvGraphVtx* **end_vtx**, const Cv-GraphEdge* **edge**=NULL, CvGraphEdge** **inserted_edge**=NULL)

> **Parameters**
>
> > > **graph** – Graph
> > >
> > > **start_vtx** – Pointer to the starting vertex of the edge
> > >
> > > **end_vtx** – Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.
> > >
> > > **edge** – Optional input parameter, initialization data for the edge
> > >
> > > **inserted_edge** – Optional output parameter to contain the address of the inserted edge within the edge set

The function connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already, and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e., when the result is negative), the function also reports an error by default.

GraphAddVtx

Adds a vertex to a graph.

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

> **Parameters**
>
> > > **graph** – Graph
> > >
> > > **vtx** – Optional input argument used to initialize the added vertex (only user-defined fields beyond `sizeof(CvGraphVtx)` are copied)
> > >
> > > **inserted_vertex** – Optional output argument. If not `NULL` , the address of the new vertex is written here.

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

GraphEdgeIdx

Returns the index of a graph edge.

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

> **Parameters**
>
>> **graph** – Graph
>>
>> **edge** – Pointer to the graph edge

The function returns the index of a graph edge.

GraphRemoveEdge

Removes an edge from a graph.

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

> **Parameters**
>
>> **graph** – Graph
>>
>> **start_idx** – Index of the starting vertex of the edge
>>
>> **end_idx** – Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

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

GraphRemoveEdgeByPtr

Removes an edge from a graph by using its pointer.

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

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

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

GraphRemoveVtx

Removes a vertex from a graph.

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

> **Parameters**
>
>> **graph** – Graph
>>
>> **vtx_idx** – Index of the removed vertex

The function removes a vertex from a graph together with all the edges incident to it. The function reports an error if the input vertex does not belong to the graph. The return value is the number of edges deleted, or -1 if the vertex does not belong to the graph.

GraphRemoveVtxByPtr

Removes a vertex from a graph by using its pointer.

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

> **Parameters**
>
> > **graph** – Graph
> >
> > **vtx** – Pointer to the removed vertex

The function removes a vertex from the graph by using its pointer together with all the edges incident to it. The function reports an error if the vertex does not belong to the graph. The return value is the number of edges deleted, or -1 if the vertex does not belong to the graph.

GraphVtxDegree

Counts the number of edges indicent to the vertex.

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

> **Parameters**
>
> > **graph** – Graph
> >
> > **vtxIdx** – Index of the graph vertex

The function returns the number of edges incident to the specified vertex, both incoming and outgoing. To count the edges, the following code is used:

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

The macro `CV_NEXT_GRAPH_EDGE( edge, vertex )` returns the edge incident to `vertex` that follows after `edge` .

GraphVtxDegreeByPtr

Finds an edge in a graph.

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

> **Parameters**
>
> > **graph** – Graph
> >
> > **vtx** – Pointer to the graph vertex

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

GraphVtxIdx

Returns the index of a graph vertex.

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

> **Parameters**
>
> > **graph** – Graph
> >
> > **vtx** – Pointer to the graph vertex

The function returns the index of a graph vertex.

InitTreeNodeIterator

Initializes the tree node iterator.

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

> **Parameters**
>
> > **tree_iterator** – Tree iterator initialized by the function
> >
> > **first** – The initial node to start traversing from
> >
> > **max_level** – The maximal level of the tree (`first` node assumed to be at the first level) to traverse up to. For example, 1 means that only nodes at the same level as `first` should be visited, 2 means that the nodes on the same level as `first` and their direct children should be visited, and so forth.

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

InsertNodeIntoTree

Adds a new node to a tree.

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

> **Parameters**
>
> > **node** – The inserted node
> >
> > **parent** – The parent node that is already in the tree
> >
> > **frame** – The top level node. If `parent` and `frame` are the same, the `v_prev` field of `node` is set to NULL rather than `parent` .

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

MakeSeqHeaderForArray

Constructs a sequence header for an array.

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

> **Parameters**
>
> > **seq_type** – Type of the created sequence

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

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

MemStorageAlloc

Allocates a memory buffer in a storage block.

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

> **Parameters**
>
> > **storage** – Memory storage
> >
> > **size** – Buffer size

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

MemStorageAllocString

Allocates a text string in a storage block.

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

> **Parameters**
>
> > **storage** – Memory storage
> >
> > **ptr** – The string
> >
> > **len** – Length of the string (not counting the ending `NUL`) . If the parameter is negative, the function computes the length.

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

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

NextGraphItem

Executes one or more steps of the graph traversal procedure.

C: int **cvNextGraphItem**(CvGraphScanner* **scanner**)

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

The function traverses through the graph until an event of interest to the user (that is, an event, specified in the `mask` in the `CreateGraphScanner()` call) is met or the traversal is completed. In the first case, it returns one of the events listed in the description of the `mask` parameter above and with the next call it resumes the traversal. In the latter case, it returns CV_GRAPH_OVER (-1). When the event is CV_GRAPH_VERTEX , CV_GRAPH_BACKTRACKING , or CV_GRAPH_NEW_TREE , the currently observed vertex is stored in `scanner-:math:`>`vtx` . And if the event is edge-related, the edge itself is stored at `scanner-:math:`>`edge` , the previously visited vertex - at `scanner-:math:`>`vtx` and the other ending vertex of the edge - at `scanner-:math:`>`dst` .

NextTreeNode

Returns the currently observed node and moves the iterator toward the next node.

C: void* **cvNextTreeNode**(CvTreeNodeIterator* **tree_iterator**)

> **Parameters**
>
> > **tree_iterator** – Tree iterator initialized by the function

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

PrevTreeNode

Returns the currently observed node and moves the iterator toward the previous node.

C: void* **cvPrevTreeNode**(CvTreeNodeIterator* **tree_iterator**)

> **Parameters**
>
> > **tree_iterator** – Tree iterator initialized by the function

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

ReleaseGraphScanner

Completes the graph traversal procedure.

C: void **cvReleaseGraphScanner**(CvGraphScanner** **scanner**)

> **Parameters**
>
> > **scanner** – Double pointer to graph traverser

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

ReleaseMemStorage

Releases memory storage.

C: void **cvReleaseMemStorage**(CvMemStorage** **storage**)

> **Parameters**
>
> > **storage** – Pointer to the released storage

The function deallocates all storage memory blocks or returns them to the parent, if any. Then it deallocates the storage header and clears the pointer to the storage. All child storage associated with a given parent storage block must be released before the parent storage block is released.

RestoreMemStoragePos

Restores memory storage position.

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

> **Parameters**
>
> > **storage** – Memory storage
> >
> > **pos** – New storage top position

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

SaveMemStoragePos

Saves memory storage position.

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

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

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

SeqElemIdx

Returns the index of a specific sequence element.

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

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

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

SeqInsert

Inserts an element in the middle of a sequence.

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

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

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

SeqInsertSlice

Inserts an array in the middle of a sequence.

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

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

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

SeqInvert

Reverses the order of sequence elements.

C: void **cvSeqInvert**(CvSeq* **seq**)

> **Parameters**
>
>> **seq** – Sequence

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

SeqPop

Removes an element from the end of a sequence.

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

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

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

SeqPopFront

Removes an element from the beginning of a sequence.

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

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

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

SeqPopMulti

Removes several elements from either end of a sequence.

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

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

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

SeqPush

Adds an element to the end of a sequence.

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

> **Parameters**
>
> > **seq** – Sequence
> >
> > **element** – Added element

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

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

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

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

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

SeqPushFront

Adds an element to the beginning of a sequence.

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

> **Parameters**
>
> > **seq** – Sequence
> >
> > **element** – Added element

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

SeqPushMulti

Pushes several elements to either end of a sequence.

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

> **Parameters**
>
> > **seq** – Sequence
> >
> > **elements** – Added elements
> >
> > **count** – Number of elements to push
> >
> > **in_front** – The flags specifying which end of the modified sequence.
> >
> > > – **CV_BACK** the elements are added to the end of the sequence
> > >
> > > – **CV_FRONT** the elements are added to the beginning of the sequence

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

SeqRemove

Removes an element from the middle of a sequence.

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

> **Parameters**
>
> > **seq** – Sequence
> >
> > **index** – Index of removed element

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

SeqRemoveSlice

Removes a sequence slice.

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

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

The function removes a slice from the sequence.

SeqSearch

Searches for an element in a sequence.

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

> **Parameters**
>
> > **seq** – The sequence
> >
> > **elem** – The element to look for
> >
> > **func** – The comparison function that returns negative, zero or positive value depending on the relationships among the elements (see also SeqSort())
> >
> > **is_sorted** – Whether the sequence is sorted or not
> >
> > **elem_idx** – Output parameter; index of the found element
> >
> > **userdata** – The user parameter passed to the compasion function; helps to avoid global variables in some cases

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

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

SeqSlice

Makes a separate header for a sequence slice.

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

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

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

SeqSort

Sorts sequence element using the specified comparison function.

C: void **cvSeqSort**(CvSeq* **seq**, CvCmpFunc **func**, void* **userdata**=NULL)

> **Parameters**
>
>> **seq** – The sequence to sort
>>
>> **func** – The comparison function that returns a negative, zero, or positive value depending on the relationships among the elements (see the above declaration and the example below) - a similar function is used by `qsort` from C runline except that in the latter, `userdata` is not used
>>
>> **userdata** – The user parameter passed to the compasion function; helps to avoid global variables in some cases

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

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

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

...

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

```
int i;

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

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

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

cvReleaseMemStorage( &storage );
```

SetAdd

Occupies a node in the set.

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

> **Parameters**
>
> > **setHeader** – Set
> >
> > **elem** – Optional input argument, an inserted element. If not NULL, the function copies the data to the allocated node (the MSB of the first integer field is cleared after copying).
> >
> > **inserted_elem** – Optional output argument; the pointer to the allocated cell

The function allocates a new node, optionally copies input element data to it, and returns the pointer and the index to the node. The index value is taken from the lower bits of the `flags` field of the node. The function has O(1) complexity; however, there exists a faster function for allocating set nodes (see `SetNew()`).

SetNew

Adds an element to a set (fast variant).

C: CvSetElem* **cvSetNew**(CvSet* **setHeader**)

> **Parameters**
>
> > **setHeader** – Set

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

SetRemove

Removes an element from a set.

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

Parameters

setHeader – Set

index – Index of the removed element

The function removes an element with a specified index from the set. If the node at the specified location is not occupied, the function does nothing. The function has O(1) complexity; however, `SetRemoveByPtr()` provides a quicker way to remove a set element if it is located already.

SetRemoveByPtr

Removes a set element based on its pointer.

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

Parameters

setHeader – Set

elem – Removed element

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

SetSeqBlockSize

Sets up sequence block size.

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

Parameters

seq – Sequence

deltaElems – Desirable sequence block size for elements

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

SetSeqReaderPos

Moves the reader to the specified position.

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

Parameters

reader – Reader state

index – The destination position. If the positioning mode is used (see the next parameter), the actual position will be index mod `reader->seq->total` .

is_relative – If it is not zero, then `index` is a relative to the current position

The function moves the read position to an absolute position or relative to the current position.

StartAppendToSeq

Initializes the process of writing data to a sequence.

C: void **cvStartAppendToSeq**(CvSeq* **seq**, CvSeqWriter* **writer**)

>   **Parameters**

>   >   **seq** – Pointer to the sequence

>   >   **writer** – Writer state; initialized by the function

The function initializes the process of writing data to a sequence. Written elements are added to the end of the sequence by using the `CV_WRITE_SEQ_ELEM( written_elem, writer )` macro. Note that during the writing process, other operations on the sequence may yield an incorrect result or even corrupt the sequence (see description of `FlushSeqWriter()`, which helps to avoid some of these problems).

StartReadSeq

Initializes the process of sequential reading from a sequence.

C: void **cvStartReadSeq**(const CvSeq* **seq**, CvSeqReader* **reader**, int **reverse**=0)

>   **Parameters**

>   >   **seq** – Sequence

>   >   **reader** – Reader state; initialized by the function

>   >   **reverse** – Determines the direction of the sequence traversal. If `reverse` is 0, the reader is positioned at the first sequence element; otherwise it is positioned at the last element.

The function initializes the reader state. After that, all the sequence elements from the first one down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM( read_elem, reader )` in the case of forward reading and by using `CV_REV_READ_SEQ_ELEM( read_elem, reader )` in the case of reverse reading. Both macros put the sequence element to `read_elem` and move the reading pointer toward the next element. A circular structure of sequence blocks is used for the reading process, that is, after the last element has been read by the macro `CV_READ_SEQ_ELEM`, the first element is read when the macro is called again. The same applies to `CV_REV_READ_SEQ_ELEM`. There is no function to finish the reading process, since it neither changes the sequence nor creates any temporary buffers. The reader field `ptr` points to the current element of the sequence that is to be read next. The code below demonstrates how to use the sequence writer and reader.

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

cvStartAppendToSeq( seq, &writer );
for( i = 0; i < 10; i++ )
{
    int val = rand()
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("
}
cvEndWriteSeq( &writer );

cvStartReadSeq( seq, &reader, 0 );
for( i = 0; i < seq->total; i++ )
{
```

```
    int val;
#if 1
    CV_READ_SEQ_ELEM( val, reader );
    printf("
#else /* alternative way, that is prefferable if sequence elements are large,
        or their size/type is unknown at compile time */
    printf("
    CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
#endif
}
...

cvReleaseStorage( &storage );
```

StartWriteSeq

Creates a new sequence and initializes a writer for it.

C: void **cvStartWriteSeq**(int **seq_flags**, int **header_size**, int **elem_size**, CvMemStorage* **storage**, CvSe-
qWriter* **writer**)

> **Parameters**

>> **seq_flags** – Flags of the created sequence. If the sequence is not passed to any function
>> working with a specific type of sequences, the sequence value may be equal to 0; otherwise
>> the appropriate type must be selected from the list of predefined sequence types.

>> **header_size** – Size of the sequence header. The parameter value may not be less than
>> `sizeof(CvSeq)` . If a certain type or extension is specified, it must fit within the base type
>> header.

>> **elem_size** – Size of the sequence elements in bytes; must be consistent with the sequence
>> type. For example, if a sequence of points is created (element type `CV_SEQ_ELTYPE_POINT`
>>), then the parameter `elem_size` must be equal to `sizeof(CvPoint)` .

>> **storage** – Sequence location

>> **writer** – Writer state; initialized by the function

The function is a combination of `CreateSeq()` and `StartAppendToSeq()` . The pointer to the created sequence is
stored at `writer->seq` and is also returned by the `EndWriteSeq()` function that should be called at the end.

TreeToNodeSeq

Gathers all node pointers to a single sequence.

C: CvSeq* **cvTreeToNodeSeq**(const void* **first**, int **header_size**, CvMemStorage* **storage**)

> **Parameters**

>> **first** – The initial tree node

>> **header_size** – Header size of the created sequence (sizeof(CvSeq) is the most frequently
>> used value)

>> **storage** – Container for the sequence

The function puts pointers of all nodes reacheable from `first` into a single sequence. The pointers are written
sequentially in the depth-first order.

2.4 Operations on Arrays

abs

Computes an absolute value of each matrix element.

C++: MatExpr **abs**(const Mat& **src**)

C++: MatExpr **abs**(const MatExpr& **src**)

> **Parameters**
>
> > **src** – Matrix or matrix expression.

abs is a meta-function that is expanded to one of `absdiff()` forms:

- `C = abs(A-B)` is equivalent to `absdiff(A, B, C)`
- `C = abs(A)` is equivalent to `absdiff(A, Scalar::all(0), C)`
- `C = Mat_<Vec<uchar,n> >(abs(A*alpha + beta))` is equivalent to `convertScaleAbs` (A, C, alpha, beta)

The output matrix has the same size and the same type as the input one except for the last case, where C is `depth=CV_8U`
.

> **See Also:**
>
> *Matrix Expressions*, `absdiff()`

absdiff

Computes the per-element absolute difference between two arrays or between an array and a scalar.

C++: void **absdiff**(InputArray **src1**, InputArray **src2**, OutputArray **dst**)

Python: cv2.**absdiff**(src1, src2$\big[$, dst$\big]$) $\rightarrow$ dst

C: void **cvAbsDiff**(const CvArr* **src1**, const CvArr* **src2**, CvArr* **dst**)

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

Python: cv.**AbsDiff**(src1, src2, dst) $\rightarrow$ None

Python: cv.**AbsDiffS**(src, dst, value) $\rightarrow$ None

> **Parameters**
>
> > **src1** – First input array or a scalar.
> >
> > **src2** – Second input array or a scalar.
> >
> > **dst** – Destination array that has the same size and type as `src1` (or `src2`).

The function absdiff computes:

- Absolute difference between two arrays when they have the same size and type:

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

- Absolute difference between an array and a scalar when the second array is constructed from `Scalar` or has as many elements as the number of channels in `src1`:

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

- Absolute difference between a scalar and an array when the first array is constructed from `Scalar` or has as many elements as the number of channels in `src2`:

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

where `I` is a multi-dimensional index of array elements. In case of multi-channel arrays, each channel is processed independently.

See Also:

`abs()`

add

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

C++: void **add**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, InputArray **mask**=noArray(), int **dtype**=-1)

Python: cv2.**add**(src1, src2[, dst[, mask[, dtype]]]) → dst

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

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

Python: cv.**Add**(src1, src2, dst, mask=None) → None

Python: cv.**AddS**(src, value, dst, mask=None) → None

> **Parameters**

>> **src1** – First source array or a scalar.

>> **src2** – Second source array or a scalar.

>> **dst** – Destination array that has the same size and number of channels as the input array(s). The depth is defined by `dtype` or `src1`/`src2`.

>> **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.

>> **dtype** – Optional depth of the output array. See the discussion below.

The function `add` computes:

- Sum of two arrays when both input arrays have the same size and the same number of channels:

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

- Sum of an array and a scalar when `src2` is constructed from `Scalar` or has the same number of elements as `src1.channels()`:

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

- Sum of a scalar and an array when `src1` is constructed from `Scalar` or has the same number of elements as `src2.channels()`:

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

where `I` is a multi-dimensional index of array elements. In case of multi-channel arrays, each channel is processed independently.

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

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

The input arrays and the destination array can all have the same or different depths. For example, you can add a 16-bit unsigned array to a 8-bit signed array and store the sum as a 32-bit floating-point array. Depth of the output array is determined by the `dtype` parameter. In the second and third cases above, as well as in the first case, when `src1.depth() == src2.depth()`, dtype can be set to the default -1. In this case, the output array will have the same depth as the input array, be it `src1`, `src2` or both.

See Also:

`subtract()`, `addWeighted()`, `scaleAdd()`, `Mat::convertTo()`, *Matrix Expressions*

addWeighted

Computes the weighted sum of two arrays.

C++: void **addWeighted**(InputArray **src1**, double **alpha**, InputArray **src2**, double **beta**, double **gamma**, OutputArray **dst**, int **dtype**=-1)

Python: cv2.**addWeighted**(src1, alpha, src2, beta, gamma[, dst[, dtype]]) → dst

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

Python: cv.**AddWeighted**(src1, alpha, src2, beta, gamma, dst) → None

> **Parameters**

>> **src1** – First source array.

>> **alpha** – Weight for the first array elements.

>> **src2** – Second source array of the same size and channel number as `src1` .

>> **beta** – Weight for the second array elements.

>> **dst** – Destination array that has the same size and number of channels as the input arrays.

>> **gamma** – Scalar added to each sum.

>> **dtype** – Optional depth of the destination array. When both input arrays have the same depth, `dtype` can be set to -1, which will be equivalent to `src1.depth()`.

The function `addWeighted` calculates 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 a multi-dimensional index of array elements. In case of multi-channel arrays, each channel is processed independently.

The function can be replaced with a matrix expression:

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

See Also:

`add()`, `subtract()`, `scaleAdd()`, `Mat::convertTo()`, *Matrix Expressions*

bitwise_and

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

C++: void **bitwise_and**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**bitwise_and**(src1, src2[, dst[, mask]]) → dst

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

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

Python: cv.**And**(src1, src2, dst, mask=None) → None

Python: cv.**AndS**(src, value, dst, mask=None) → None

> **Parameters**

>> **src1** – First source array or a scalar.

>> **src2** – Second source array or a scalar.

>> **dst** – Destination arrayb that has the same size and type as the input array(s).

>> **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.

The function computes the per-element bit-wise logical conjunction for:

- Two arrays when `src1` and `src2` have the same size:

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

- An array and a scalar when `src2` is constructed from `Scalar` or has the same number of elements as `src1.channels()`:

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

- A scalar and an array when `src1` is constructed from `Scalar` or has the same number of elements as `src2.channels()`:

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

In case of floating-point arrays, their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. In case of multi-channel arrays, each channel is processed independently. In the second and third cases above, the scalar is first converted to the array type.

bitwise_not

Inverts every bit of an array.

C++: void **bitwise_not**(InputArray **src**, OutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**bitwise_not**(src[, dst[, mask]]) → dst

C: void **cvNot**(const CvArr* **src**, CvArr* **dst**)

Python: cv.**Not**(src, dst) → None

> **Parameters**

>> **src** – Source array.

> > **dst** – Destination array that has the same size and type as the input array.
>
> > **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.

The function computes per-element bit-wise inversion of the source array:

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

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

bitwise_or

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

C++: void **bitwise_or**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**bitwise_or**(src1, src2[, dst[, mask]]) → dst

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

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

Python: cv.**Or**(src1, src2, dst, mask=None) → None

Python: cv.**OrS**(src, value, dst, mask=None) → None

> **Parameters**

> > **src1** – First source array or a scalar.

> > **src2** – Second source array or a scalar.

> > **dst** – Destination array that has the same size and type as the input array(s).

> > **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.

The function computes the per-element bit-wise logical disjunction for:

- Two arrays when `src1` and `src2` have the same size:

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

- An array and a scalar when `src2` is constructed from `Scalar` or has the same number of elements as `src1.channels()`:

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

- A scalar and an array when `src1` is constructed from `Scalar` or has the same number of elements as `src2.channels()`:

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

In case of floating-point arrays, their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. In case of multi-channel arrays, each channel is processed independently. In the second and third cases above, the scalar is first converted to the array type.

---

bitwise_xor

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

C++: void **bitwise_xor**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**bitwise_xor**(src1, src2[, dst[, mask]]) → dst

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

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

Python: cv.**Xor**(src1, src2, dst, mask=None) → None

Python: cv.**XorS**(src, value, dst, mask=None) → None

> **Parameters**
>
> > **src1** – First source array or a scalar.
> >
> > **src2** – Second source array or a scalar.
> >
> > **dst** – Destination array that has the same size and type as the input array(s).
> >
> > **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.

The function computes the per-element bit-wise logical "exclusive-or" operation for:

* Two arrays when `src1` and `src2` have the same size:

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

* An array and a scalar when `src2` is constructed from `Scalar` or has the same number of elements as `src1.channels()`:

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

* A scalar and an array when `src1` is constructed from `Scalar` or has the same number of elements as `src2.channels()`:

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

In case of floating-point arrays, their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. In case of multi-channel arrays, each channel is processed independently. In the 2nd and 3rd cases above, the scalar is first converted to the array type.

calcCovarMatrix

Calculates the covariance matrix of a set of vectors.

C++: void **calcCovarMatrix**(const Mat* **samples**, int **nsamples**, Mat& **covar**, Mat& **mean**, int **flags**, int **ctype**=CV_64F)

C++: void **calcCovarMatrix**(InputArray **samples**, OutputArray **covar**, OutputArray **mean**, int **flags**, int **ctype**=CV_64F)

Python: cv2.**calcCovarMatrix**(samples, flags[, covar[, mean[, ctype]]]) → covar, mean

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

Python: cv.**CalcCovarMatrix**(vects, covMat, avg, flags) → None

> **Parameters**

>> **samples** – Samples stored either as separate matrices or as rows/columns of a single matrix.

>> **nsamples** – Number of samples when they are stored separately.

>> **covar** – Output covariance matrix of the type `ctype` and square size.

>> **mean** – Input or output (depending on the flags) array as the average value of the input vectors.

>> **flags** – Operation flags as a combination of the following values:

>> – **CV_COVAR_SCRAMBLED** The output covariance matrix is calculated as:

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

>> The covariance matrix will be `nsamples x 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 match the eigenvalues of the true covariance matrix. 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:

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

>> `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 in advance, or if the covariance matrix is calculated by parts. In this case, `mean` is not a mean vector of the input sub-set of vectors but rather the mean vector of the whole set.

>> – **CV_COVAR_SCALE** If the flag is specified, the covariance matrix is scaled. In the "normal" mode, `scale` is `1./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 (`scale=1`).

>> – **CV_COVAR_ROWS** [Only useful in the second variant of the function] If the flag is specified, 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] If the flag is specified, 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

Calculates the magnitude and angle of 2D vectors.

C++: void **cartToPolar**(InputArray **x**, InputArray **y**, OutputArray **magnitude**, OutputArray **angle**, bool **angleInDegrees**=false)

Python: cv2.**cartToPolar**(x, y[, magnitude[, angle[, angleInDegrees]]]) → magnitude, angle

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

Python: cv.**CartToPolar**(x, y, magnitude, angle=None, angleInDegrees=0) → None

> **Parameters**
>
> > **x** – Array of x-coordinates. This must be a single-precision or double-precision floating-point array.
> >
> > **y** – Array of y-coordinates that must have the same size and same type as x .
> >
> > **magnitude** – Destination array of magnitudes of the same size and type as x .
> >
> > **angle** – Destination array of angles that has the same size and type as x . The angles are measured in radians (from 0 to 2*Pi) or in degrees (0 to 360 degrees).
> >
> > **angleInDegrees** – Flag indicating whether the angles are measured in radians, which is the default mode, or in degrees.

The function cartToPolar calculates either the magnitude, angle, or both for every 2D vector (x(I),y(I)):

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

The angles are calculated with accuracy about 0.3 degrees. For the point (0,0), the angle is set to 0.

checkRange

Checks every element of an input array for invalid values.

C++: bool **checkRange**(InputArray **src**, bool **quiet**=true, Point* **pos**=0, double **minVal**=-DBL_MAX, double **maxVal**=DBL_MAX)

Python: cv2.**checkRange**(a[, quiet[, minVal[, maxVal]]]) → retval, pt

> **Parameters**
>
> > **src** – Array to check.
> >
> > **quiet** – Flag indicating whether the functions quietly return false when the array elements are out of range or they throw an exception.
> >
> > **pos** – 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** – Inclusive lower boundary of valid values range.
> >
> > **maxVal** – Exclusive upper boundary of valid values range.

The functions checkRange check that every array element is neither NaN nor infinite. When minVal < -DBL_MAX and maxVal < DBL_MAX , the functions also check that each value is between minVal and maxVal . In 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 != NULL). Then, the functions either return false (when quiet=true) or throw an exception.

compare

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

C++: void **compare**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, int **cmpop**)

Python: cv2.**compare**(src1, src2, cmpop[, dst]) → dst

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

Python: cv.**Cmp**(src1, src2, dst, cmpOp) → None

C: void **cvCmpS**(const CvArr* **src1**, double **src2**, CvArr* **dst**, int **cmpOp**)

Python: cv.**CmpS**(src1, src2, dst, cmpOp) → None

   Parameters

     src1 – First source array or a scalar (in the case of cvCmp, cv.Cmp, cvCmpS, cv.CmpS it is always an array). When it is array, it must have a single channel.

     src2 – Second source array or a scalar (in the case of cvCmp and cv.Cmp it is always an array; in the case of cvCmpS, cv.CmpS it is always a scalar). When it is array, it must have a single channel.

     dst – Destination array that has the same size as the input array(s) and type= CV_8UC1 .

     cmpop – Flag specifying the relation between the elements to be checked.

      – **CMP_EQ** src1 equal to src2.

      – **CMP_GT** src1 greater than src2.

      – **CMP_GE** src1 greater than or equal to src2.

      – **CMP_LT** src1 less than src2.

      – **CMP_LE** src1 less than or equal to src2.

      – **CMP_NE** src1 not equal to src2.

The function compares:

- Elements of two arrays when src1 and src2 have the same size:

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

- Elements of src1 with a scalar src2' when ''src2 is constructed from Scalar or has a single element:

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

- src1 with elements of src2 when src1 is constructed from Scalar or has a single element:

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

When the comparison result is true, the corresponding element of destination array is set to 255. 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

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

C++: void **completeSymm**(InputOutputArray **mtx**, bool **lowerToUpper**=false)

Python: cv2.**completeSymm**(mtx[, lowerToUpper]) → None

> **Parameters**
>
>> **mtx** – Input-output floating-point square matrix.
>>
>> **lowerToUpper** – Operation flag. If it is 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

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

C++: void **convertScaleAbs**(InputArray **src**, OutputArray **dst**, double **alpha**=1, double **beta**=0)

Python: cv2.**convertScaleAbs**(src[, dst[, alpha[, beta]]]) → dst

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

Python: cv.**ConvertScaleAbs**(src, dst, scale=1.0, shift=0.0) → None

> **Parameters**
>
>> **src** – Source array.
>>
>> **dst** – Destination array.
>>
>> **alpha** – Optional scale factor.
>>
>> **beta** – Optional delta added to the scaled values.

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

$$dst(I) = saturate\_cast<uchar>(|src(I) * alpha + beta|)$$

In 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 the Mat::convertTo method (or by using matrix expressions) and then by computing an 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

Counts non-zero array elements.

C++: int **countNonZero**(InputArray **mtx**)

Python: cv2.**countNonZero**(src) → retval

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

Python: cv.**CountNonZero**(arr) → int

> **Parameters**

> > **mtx** – Single-channel array.

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

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

See Also:

`mean()`, `meanStdDev()`, `norm()`, `minMaxLoc()`, `calcCovarMatrix()`

cvarrToMat

Converts `CvMat`, `IplImage` , or `CvMatND` to `Mat`.

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

> **Parameters**

> > **src** – Source `CvMat`, `IplImage` , or `CvMatND` .

> > **copyData** – When it is false (default value), no data is copied and only the new header is created. In this case, the original array should not be deallocated while the new matrix header is used. If the parameter is true, all the data is copied and you may deallocate the original array right after the conversion.

> > **allowND** – When it is true (default value), `CvMatND` is converted to 2-dimensional `Mat`, if it is possible (see the discussion below). If it is not possible, or when the parameter is false, the function will report an error.

> > **coiMode** – Parameter specifying how the IplImage COI (when set) is handled.

> > – If `coiMode=0` and COI is set, the function reports an error.

> > – If `coiMode=1` , the function never reports an error. Instead, it returns the header to the whole original image and you 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. In this case, you have to preserve the data until the new header is destructed. Otherwise, when `copyData=true` , the new buffer is allocated

---

and managed as if you created a new matrix from scratch and copied the data there. That is, `cvarrToMat(src, true)` is equivalent to `cvarrToMat(src, false).clone()` (assuming that COI is not set). The function provides a 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 a 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, for example:
printf("%g\n", cvNorm(&C1, 0, CV_L2));
```

Normally, the function is used to convert an old-style 2D array (`CvMat` or `IplImage`) to `Mat` . However, the function can also take `CvMatND` as an input and create `Mat()` for it, if it is 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 an arbitrary element-wise function.

The last parameter, `coiMode` , specifies how to deal with an image with COI set. By default, it is 0 and the function reports an error when an image with COI comes in. And `coiMode=1` means that no error is signalled. You have to check COI presence and handle it manually. The modern structures, such as `Mat` and `MatND` do not support COI natively. To process an individual channel of a new-style array, you need either to organize a loop over the array (for example, 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 needed (using `mixChannels()` or `insertImageCOI()` , respectively).

See Also:

`cvGetImage()`, `cvGetMat()`, `extractImageCOI()`, `insertImageCOI()`, `mixChannels()`

dct

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

C++: void **dct**(InputArray **src**, OutputArray **dst**, int **flags**=0)

Python: cv2.**dct**(src[, dst[, flags]]) → dst

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

Python: cv.**DCT**(src, dst, flags) → None

> **Parameters**
>
>> **src** – Source floating-point array.
>>
>> **dst** – Destination array of the same size and type as `src` .
>>
>> **flags** – Transformation flags as a combination of the following values:
>>
>> – **DCT_INVERSE** performs an inverse 1D or 2D transform instead of the default forward transform.
>>
>> – **DCT_ROWS** performs a forward or inverse transform of every individual row of the input matrix. This flag enables you to transform multiple vectors simultaneously and

can be used to decrease the overhead (which is sometimes several times larger than the processing itself) to perform 3D and higher-dimensional transforms and so forth.

The function `dct` performs a forward or inverse discrete Cosine transform (DCT) of a 1D or 2D floating-point array:

- Forward Cosine transform of a 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 a 1D vector of N elements:

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

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

- Forward 2D Cosine transform of M x N matrix:

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

- Inverse 2D Cosine transform of M x N matrix:

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

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 a forward 1D or 2D transform. Otherwise, it is an inverse 1D or 2D transform.

- If `(flags & DCT_ROWS) != 0`, the function performs a 1D transform of each row.

- If the array is a single column or a single row, the function performs a 1D transform.

- If none of the above is true, the function performs a 2D transform.

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

Also, the function 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 N1 >= N can be computed as:

```
size_t getOptimalDCTSize(size_t N) { return 2*getOptimalDFTSize((N+1)/2); }
N1 = getOptimalDCTSize(N);
```

See Also:

`dft()`, `getOptimalDFTSize()`, `idct()`

dft

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

C++: void **dft**(InputArray **src**, OutputArray **dst**, int **flags**=0, int **nonzeroRows**=0)

Python: cv2.**dft**(src[, dst[, flags[, nonzeroRows]]]) → dst

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

Python: cv.**DFT**(src, dst, flags, nonzeroRows=0) → None

> **Parameters**

>> **src** – Source array that could be real or complex.

>> **dst** – Destination array whose size and type depends on the `flags` .

>> **flags** – Transformation flags representing a combination of the following values:

>> – **DFT_INVERSE** performs an inverse 1D or 2D transform instead of the default forward transform.

>> – **DFT_SCALE** scales the result: divide it by the number of array elements. Normally, it is combined with `DFT_INVERSE` .

>> – **DFT_ROWS** performs a forward or inverse transform of every individual row of the input matrix. This flag enables you to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself) to perform 3D and higher-dimensional transforms and so forth.

>> – **DFT_COMPLEX_OUTPUT** performs a forward transformation of 1D or 2D real array. The result, though being a complex array, has complex-conjugate symmetry (*CCS*, see the function description below for details). Such an array can be packed into a 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 a full complex array (for simpler spectrum analysis, and so on). Pass the flag to enable the function to produce a full-size complex output array.

>> – **DFT_REAL_OUTPUT** performs an inverse transformation of a 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), the output is a 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 into a real array and inverse transformation is executed, the function treats the input as a packed complex-conjugate symmetrical array. So, the output will also be a real array.

>> **nonzeroRows** – When the parameter is not zero, 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 save some time. This technique is very useful for computing array cross-correlation or convolution using DFT.

The function performs one of the following:

• Forward the Fourier transform of a 1D vector of `N` elements:

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

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

- Inverse the Fourier transform of a 1D vector of `N` elements:

$$X' = \left(F^{(N)}\right)^{-1} \cdot Y = \left(F^{(N)}\right)^{*} \cdot y$$
$$X = (1/N) \cdot X,$$

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

- Forward the 2D Fourier transform of a `M x N` matrix:

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

- Inverse the 2D Fourier transform of a `M x N` matrix:

$$X' = \left(F^{(M)}\right)^{*} \cdot Y \cdot \left(F^{(N)}\right)^{*}$$
$$X = \frac{1}{M \cdot N} \cdot X'$$

In case of real (single-channel) data, the output spectrum of the forward Fourier transform or input spectrum of the inverse Fourier transform can be represented in a packed format called *CCS* (complex-conjugate-symmetrical). It was borrowed from IPL (Intel* Image Processing Library). Here is how 2D *CCS* spectrum looks:

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

In case of 1D transform of a real vector, the output looks like the first row of the matrix above.

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

- If `DFT_ROWS` is set or the input array has a single row or single column, the function performs a 1D forward or inverse transform of each row of a matrix when `DFT_ROWS` is set. Otherwise, it performs a 2D transform.

- If the input array is real and `DFT_INVERSE` is not set, the function performs a forward 1D or 2D transform:

    - When `DFT_COMPLEX_OUTPUT` is set, the output is a complex matrix of the same size as input.

    - When `DFT_COMPLEX_OUTPUT` is not set, the output is a real matrix of the same size as input. In case of 2D transform, it uses the packed format as shown above. In case of a single 1D transform, it looks like the first row of the matrix above. In case of multiple 1D transforms (when using the `DCT_ROWS` flag), each row of the output matrix looks like the first row of the matrix above.

- If the input array is complex and either `DFT_INVERSE` or `DFT_REAL_OUTPUT` are not set, the output is a complex array of the same size as input. The function performs a 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`.

- When `DFT_INVERSE` is set and the input array is real, or it is complex but `DFT_REAL_OUTPUT` is set, the output is a real array of the same size as input. The function performs a 1D or 2D inverse transformation of the whole input array or each individual row, depending on the flags `DFT_INVERSE` and `DFT_ROWS`.

If `DFT_SCALE` is set, the scaling is done after the transformation.

Unlike `dct()` , the function supports arrays of arbitrary size. But only those arrays are processed efficiently, whose 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 the `getOptimalDFTSize()` method.

The sample below illustrates how to compute a DFT-based convolution of two 2D real arrays:

```
void convolveDFT(InputArray A, InputArray B, OutputArray C)
{
    // reallocate the output array if needed
    C.create(abs(A.rows - B.rows)+1, abs(A.cols - B.cols)+1, A.type());
    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 0's
    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 tempB, respectively
    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,
    // you need only the first C.rows of them, and thus you
    // 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 automatically
}
```

To optimize this sample, consider the following approaches:

- Since nonzeroRows != 0 is passed to the forward transform calls and since A and B are copied to the top-left corners of tempA and tempB, respectively, it is 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, you can compute convolution by parts. To do this, you need to split the destination array C into multiple tiles. 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 a slowdown because of bad cache locality. So, there is an optimal tile size somewhere in the middle.

- If different tiles in C can be computed in parallel and, thus, the convolution is done by parts, the loop can be threaded.

All of the above improvements have been implemented in matchTemplate() and filter2D() . Therefore, by using them, you can get the performance even better than with the above theoretically optimal implementation. Though,

those two functions actually compute cross-correlation, not convolution, so you need to "flip" the second convolution operand B vertically and horizontally using `flip()` .

See Also:

`dct()` , `getOptimalDFTSize()` , `mulSpectrums()` , `filter2D()` , `matchTemplate()` , `flip()` , `cartToPolar()` , `magnitude()` , `phase()`

divide

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

C++: void **divide**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, double **scale**=1, int **dtype**=-1)

C++: void **divide**(double **scale**, InputArray **src2**, OutputArray **dst**, int **dtype**=-1)

Python: cv2.**divide**(src1, src2[, dst[, scale[, dtype]]]) $\rightarrow$ dst

Python: cv2.**divide**(scale, src2[, dst[, dtype]]) $\rightarrow$ dst

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

Python: cv.**Div**(src1, src2, dst, scale) $\rightarrow$ None

> **Parameters**

>> **src1** – First source array.

>> **src2** – Second source array of the same size and type as `src1` .

>> **scale** – Scalar factor.

>> **dst** – Destination array of the same size and type as `src2` .

>> **dtype** – Optional depth of the destination array. If it is `-1`, `dst` will have depth `src2.depth()`. In case of an array-by-array division, you can only pass `-1` when `src1.depth()==src2.depth()`.

The functions `divide` divide one array by another:

$$dst(I) = saturate(src1(I)*scale/src2(I))$$

or a scalar by an array when there is no `src1` :

$$dst(I) = saturate(scale/src2(I))$$

When `src2(I)` is zero, `dst(I)` will also be zero. Different channels of multi-channel arrays are processed independently.

See Also:

`multiply()` , `add()` , `subtract()` , *Matrix Expressions*

determinant

Returns the determinant of a square floating-point matrix.

C++: double **determinant**(InputArray **mtx**)

Python: cv2.**determinant**(mtx) $\rightarrow$ retval

C: double **cvDet**(const CvArr* **mtx**)

Python: cv.**Det**(mtx) $\rightarrow$ double

Parameters

mtx – Input matrix that must have `CV_32FC1` or `CV_64FC1` type and square size.

The function `determinant` computes and returns the 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 with partial pivoting.

For symmetric positively-determined matrices, it is also possible to use `eigen()` decomposition to compute the determinant.

See Also:

`trace()`, `invert()`, `solve()`, `eigen()`, *Matrix Expressions*

eigen

C++: bool **eigen**(InputArray **src**, OutputArray **eigenvalues**, int **lowindex**=-1, int **highindex**=-1)

C++: bool **eigen**(InputArray **src**, OutputArray **eigenvalues**, OutputArray **eigenvectors**, int **lowindex**=-1, int **highindex**=-1)

C: void **cvEigenVV**(CvArr* **src**, CvArr* **eigenvectors**, CvArr* **eigenvalues**, double **eps**=0, int **lowindex**=-1, int **highindex**=-1)

Python: cv.**EigenVV**(src, eigenvectors, eigenvalues, eps, lowindex=-1, highindex=-1) → None
Computes eigenvalues and eigenvectors of a symmetric matrix.

Python: cv2.**eigen**(src, computeEigenvectors[, eigenvalues[, eigenvectors[, lowindex[, highindex]]]])
→ retval, eigenvalues, eigenvectors

Parameters

src – Input matrix that must have `CV_32FC1` or `CV_64FC1` type, square size and be symmetrical ($src^T == src$).

eigenvalues – Output vector of eigenvalues of the same type as `src` . The eigenvalues are stored in the descending order.

eigenvectors – Output matrix of eigenvectors. It has the same size and 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. The parameter is ignored in the current implementation.

highindex – Optional index of smallest eigenvalue/-vector to calculate. The parameter is ignored in the current implementation.

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

`src*eigenvectors.row(i).t() = eigenvalues.at<srcType>(i)*eigenvectors.row(i).t()`

Note: in the new and the old interfaces different ordering of eigenvalues and eigenvectors parameters is used.

See Also:

`completeSymm()`, PCA

exp

Calculates the exponent of every array element.

C++: void **exp**(InputArray **src**, OutputArray **dst**)

Python: cv2.**exp**(src[, dst]) → dst

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

Python: cv.**Exp**(src, dst) → None

> **Parameters**

>> **src** – Source array.

>> **dst** – Destination array of the same size and type as `src`.

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

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

The maximum relative error is about `7e-6` for single-precision input and less than `1e-10` for double-precision input. Currently, the function converts denormalized values to zeros on output. Special values (NaN, Inf) are not handled.

See Also:

log() , cartToPolar() , polarToCart() , phase() , pow() , sqrt() , magnitude()

extractImageCOI

Extracts the selected image channel.

C++: void **extractImageCOI**(const CvArr* **src**, OutputArray **dst**, int **coi**=-1)

> **Parameters**

>> **src** – Source array. It should be a pointer to `CvMat` or `IplImage` .

>> **dst** – Destination array with a single channel and the same size and depth as `src` .

>> **coi** – If the parameter is `>=0` , it specifies the channel to extract. If it is `<0` and `src` is a pointer to `IplImage` with a valid COI set, the selected COI is extracted.

The function extractImageCOI is used to extract an 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()

flip

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

C++: void **flip**(InputArray **src**, OutputArray **dst**, int **flipCode**)

Python: cv2.**flip**(src, flipCode[, dst]) → dst

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

Python: cv.**Flip**(src, dst=None, flipMode=0) → None

Parameters

src – Source array.

dst – Destination array of the same size and type as `src`.

flipCode – Flag to specify how to flip the array. 0 means flipping around the x-axis. Positive value (for example, 1) means flipping around y-axis. Negative value (for example, -1) means flipping around both axes. See 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):

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

The example scenarios of using the function are the following:

- Vertical flipping of the image (`flipCode == 0`) to switch between top-left and bottom-left image origin. This is a typical operation in video processing on Microsoft Windows* OS.

- Horizontal flipping of the image with the 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 the subsequent shift and absolute difference calculation to check for a central symmetry (`flipCode < 0`).

- Reversing the order of point arrays (`flipCode > 0` or `flipCode == 0`).

See Also:

`transpose()` , `repeat()` , `completeSymm()`

gemm

Performs generalized matrix multiplication.

C++: void **gemm**(InputArray **src1**, InputArray **src2**, double **alpha**, InputArray **src3**, double **beta**, OutputArray **dst**, int **flags**=0)

Python: cv2.**gemm**(src1, src2, alpha, src3, gamma$\big[$, dst$\big[$, flags $\big]\big]$) $\rightarrow$ dst

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

Python: cv.**GEMM**(src1, src2, alphs, src3, beta, dst, tABC=0) $\rightarrow$ None

Parameters

src1 – First multiplied input matrix that should have `CV_32FC1` , `CV_64FC1` , `CV_32FC2` , or `CV_64FC2` type.

src2 – Second multiplied input matrix of the same type as `src1`.

alpha – Weight of the matrix product.

src3 – Third optional delta matrix added to the matrix product. It should have the same type as `src1` and `src2`.

beta – Weight of `src3`.

dst – Destination matrix. It has 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 similar to the `gemm` functions 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. For example, the above call can be replaced with:

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

See Also:

`mulTransposed()` , `transform()` , *Matrix Expressions*

getConvertElem

Returns a conversion function for a single pixel.

C++: ConvertData **getConvertElem**(int **fromType**, int **toType**)

C++: ConvertScaleData **getConvertScaleElem**(int **fromType**, int **toType**)

> **Parameters**
>
> > **fromType** – Source pixel type.
> >
> > **toType** – 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. It 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.

`ConvertData` and `ConvertScaleData` are defined as:

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

See Also:

`Mat::convertTo()` , `SparseMat::convertTo()`

getOptimalDFTSize

Returns the optimal DFT size for a given vector size.

C++: int **getOptimalDFTSize**(int **vecsize**)

Python: cv2.**getOptimalDFTSize**(vecsize) → retval

C: int **cvGetOptimalDFTSize**(int **size0**)

Python: cv.**GetOptimalDFTSize**(size0) → int

> **Parameters**
>
>> **vecsize** – Vector size.

DFT performance is not a monotonic function of a vector size. Therefore, when you compute convolution of two arrays or perform the spectral analysis of an 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 whose size is a power-of-two (2, 4, 8, 16, 32, ...) are the fastest to process. Though, the arrays whose size is a product of 2's, 3's, and 5's (for example, 300 = 5*5*3*2*2) are also processed quite efficiently.

The function `getOptimalDFTSize` returns the minimum number `N` that is greater than or equal to `vecsize` so that the DFT of a vector of size `N` can be computed efficiently. In the current implementation $N = 2^p * 3^q * 5^r$ for some integer `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

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

C++: void **idct**(InputArray **src**, OutputArray **dst**, int **flags**=0)

Python: cv2.**idct**(src[, dst[, flags]]) → dst

> **Parameters**
>
>> **src** – Source floating-point single-channel array.
>>
>> **dst** – Destination array of the same size and type as `src` .
>>
>> **flags** – Operation flags.

`idct(src, dst, flags)` is equivalent to `dct(src, dst, flags | DCT_INVERSE)`.

See Also:

`dct()` , `dft()` , `idft()` , `getOptimalDFTSize()`

idft

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

C++: void **idft**(InputArray **src**, OutputArray **dst**, int **flags**=0, int **outputRows**=0)

Python: cv2.**idft**(src[, dst[, flags[, nonzeroRows]]]) → dst

> **Parameters**
>
>> **src** – Source floating-point real or complex array.

>   **dst** – Destination array whose size and type depend on the `flags` .
>
>   **flags** – Operation flags. See `dft()` .
>
>   **nonzeroRows** – Number of `dst` rows to compute. The rest of the rows 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: None of `dft` and `idft` scales the result by default. So, 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

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

C++: void **inRange**(InputArray **src**, InputArray **lowerb**, InputArray **upperb**, OutputArray **dst**)

Python: cv2.**inRange**(src, lowerb, upperb[, dst]) → dst

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

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

Python: cv.**InRange**(src, lower, upper, dst) → None

Python: cv.**InRangeS**(src, lower, upper, dst) → None

>   **Parameters**
>
>   > **src** – First source array.
>   >
>   > **lowerb** – Inclusive lower boundary array or a scalar.
>   >
>   > **upperb** – Inclusive upper boundary array or a scalar.
>   >
>   > **dst** – Destination array of the same size as `src` and `CV_8U` type.

The function checks the range as follows:

* For every element of a single-channel input array:

$$\texttt{dst}(I) = \texttt{lowerb}(I)_0 \le \texttt{src}(I)_0 \le \texttt{upperb}(I)_0$$

* For two-channel arrays:

$$\texttt{dst}(I) = \texttt{lowerb}(I)_0 \le \texttt{src}(I)_0 \le \texttt{upperb}(I)_0 \wedge \texttt{lowerb}(I)_1 \le \texttt{src}(I)_1 \le \texttt{upperb}(I)_1$$

* and so forth.

That is, `dst` (I) is set to 255 (all `1` -bits) if `src` (I) is within the specified 1D, 2D, 3D, ... box and 0 otherwise.

When the lower and/or upper bounary parameters are scalars, the indexes (`I`) at `lowerb` and `upperb` in the above formulas should be omitted.

invert

Finds the inverse or pseudo-inverse of a matrix.

C++: double **invert**(InputArray **src**, OutputArray **dst**, int **method**=DECOMP_LU)

Python: cv2.**invert**(src[, dst[, flags]]) → retval, dst

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

Python: cv.**Invert**(src, dst, method=CV_LU) → double

> **Parameters**
>
> > **src** – Source floating-point M x N matrix.
> >
> > **dst** – Destination matrix of N x M size and the same type as src .
> >
> > **flags** – Inversion method :
> >
> > – **DECOMP_LU** Gaussian elimination with the optimal pivot element chosen.
> >
> > – **DECOMP_SVD** Singular value decomposition (SVD) method.
> >
> > – **DECOMP_CHOLESKY** Cholesky decomposion. The matrix must be symmetrical and positively defined.

The function invert inverts the matrix src and stores the result in dst . When the matrix src is singular or non-square, the function computes the pseudo-inverse matrix (the dst matrix) so that norm(src*dst - I) is minimal, where I is an identity matrix.

In case of the 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 case of the DECOMP_SVD method, the function returns the inverse 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 that should also be symmetrical and positively defined. In this case, the function stores the inverted matrix in dst and returns non-zero. Otherwise, it returns 0.

See Also:

solve(), SVD

log

Calculates the natural logarithm of every array element.

C++: void **log**(InputArray **src**, OutputArray **dst**)

Python: cv2.**log**(src[, dst]) → dst

C: void **cvLog**(const CvArr* **src**, CvArr* **dst**)

Python: cv.**Log**(src, dst) → None

> **Parameters**
>
> > **src** – Source array.
> >
> > **dst** – Destination array of the same size and type as src .

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

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

where C is a large negative number (about -700 in the current implementation). The maximum relative error is about 7e-6 for single-precision input and less than 1e-10 for double-precision input. Special values (NaN, Inf) are not handled.

See Also:

exp(), cartToPolar(), polarToCart(), phase(), pow(), sqrt(), magnitude()

LUT

Performs a look-up table transform of an array.

C++: void **LUT**(InputArray **src**, InputArray **lut**, OutputArray **dst**)

Python: cv2.**LUT**(src, lut[, dst[, interpolation]]) → dst

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

Python: cv.**LUT**(src, dst, lut) → None

>   **Parameters**

>>   **src** – Source array of 8-bit elements.

>>   **lut** – Look-up table of 256 elements. In 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 of the same size and the same number of channels as src , and the same depth as lut .

The function LUT 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:

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

where

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

See Also:

convertScaleAbs(), Mat::convertTo()

magnitude

Calculates the magnitude of 2D vectors.

C++: void **magnitude**(InputArray **x**, InputArray **y**, OutputArray **magnitude**)

Python: cv2.**magnitude**(x, y[, magnitude]) → magnitude

>   **Parameters**

>>   **x** – Floating-point array of x-coordinates of the vectors.

y – Floating-point array of y-coordinates of the vectors. It must have the same size as x .

dst – Destination array of the same size and type as x .

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

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

See Also:

`cartToPolar()`, `polarToCart()`, `phase()`, `sqrt()`

Mahalanobis

Calculates the Mahalanobis distance between two vectors.

C++: double **Mahalanobis**(InputArray **vec1**, InputArray **vec2**, InputArray **icovar**)

Python: cv2.**Mahalanobis**(v1, v2, icovar) → retval

C: double **cvMahalanobis**(const CvArr* **vec1**, const CvArr* **vec2**, CvArr* **icovar**)

Python: cv.**Mahalanobis**(vec1, vec2, icovar) → None

> **Parameters**

>> **vec1** – First 1D source vector.

>> **vec2** – Second 1D source vector.

>> **icovar** – Inverse covariance matrix.

The function `Mahalanobis` 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 the `DECOMP_SVD` method, as the most accurate).

max

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

C++: MatExpr **max**(const Mat& **src1**, const Mat& **src2**)

C++: MatExpr **max**(const Mat& **src1**, double **value**)

C++: MatExpr **max**(double **value**, const Mat& **src1**)

C++: void **max**(InputArray **src1**, InputArray **src2**, OutputArray **dst**)

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

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

Python: cv2.**max**(src1, src2[, dst]) → dst

C: void **cvMax**(const CvArr* **src1**, const CvArr* **src2**, CvArr* **dst**)

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

Python: cv.**Max**(src1, src2, dst) → None

Python: cv.**MaxS**(src, value, dst) → None

>   **Parameters**

>>   **src1** – First source array.

>>   **src2** – Second source array of the same size and type as src1 .

>>   **value** – Real scalar value.

>>   **dst** – Destination array of the same size and type as src1 .

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

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

or array and a scalar:

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

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 an expression object that can be further either transformed/ assigned to a matrix, or passed to a function, and so on.

See Also:

min(), compare(), inRange(), minMaxLoc(), *Matrix Expressions*

mean

Calculates an average (mean) of array elements.

C++: Scalar **mean**(InputArray **src**, InputArray **mask**=noArray())

Python: cv2.**mean**(src[, mask]) → retval

C: CvScalar **cvAvg**(const CvArr* **src**, const CvArr* **mask**=NULL)

Python: cv.**Avg**(src, mask=None) → CvScalar

>   **Parameters**

>>   **src** – Source array that should have from 1 to 4 channels so that the result can be stored in Scalar_ .

>>   **mask** – Optional operation mask.

The function mean computes the mean value M of array elements, independently for each channel, and return it:

$$N = \sum_{I: \, \mathtt{mask}(I) \neq 0} 1$$
$$M_c = \left(\sum_{I: \, \mathtt{mask}(I) \neq 0} \mathtt{mtx}(I)_c \right) / N$$

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

See Also:

countNonZero(), meanStdDev(), norm(), minMaxLoc()

meanStdDev

Calculates a mean and standard deviation of array elements.

C++: void **meanStdDev**(InputArray **src**, OutputArray **mean**, OutputArray **stddev**, InputArray **mask**=noArray())

Python: cv2.**meanStdDev**(src[, mean[, stddev[, mask]]]) → mean, stddev

C: void **cvAvgSdv**(const CvArr* **src**, CvScalar* **mean**, CvScalar* **stdDev**, const CvArr* **mask**=NULL)

Python: cv.**AvgSdv**(src, mask=None)-> (mean, stdDev)

> **Parameters**
>
>> **src** – Source array that should have from 1 to 4 channels so that the results can be stored in `Scalar_` 's.
>>
>> **mean** – Output parameter: computed mean value.
>>
>> **stddev** – Output parameter: computed standard deviation.
>>
>> **mask** – Optional operation mask.

The function `meanStdDev` computes the mean and the standard deviation M of array elements independently for each channel and returns it via the output parameters:

$$
\begin{aligned}
N &= \sum_{I,\text{mask}(I)\neq 0} 1 \\
\text{mean}_c &= \frac{\sum_{I:\ \text{mask}(I)\neq 0} \text{src}(I)_c}{N} \\
\text{stddev}_c &= \sqrt{\sum_{I:\ \text{mask}(I)\neq 0} \left(\text{src}(I)_c - \text{mean}_c\right)^2}
\end{aligned}
$$

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

Note: 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 x N` to the single-channel array `M*N x mtx.channels()` (only possible when the matrix is continuous) and then pass the matrix to `calcCovarMatrix()`.

See Also:

`countNonZero()`, `mean()`, `norm()`, `minMaxLoc()`, `calcCovarMatrix()`

merge

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

C++: void **merge**(const Mat* **mv**, size_t **count**, OutputArray **dst**)

C++: void **merge**(const vector<Mat>& **mv**, OutputArray **dst**)

Python: cv2.**merge**(mv[, dst]) → dst

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

Python: cv.**Merge**(src0, src1, src2, src3, dst) → None

> **Parameters**
>
>> **mv** – Source array or vector of matrices to be merged. All the matrices in `mv` must have the same size and the same depth.
>>
>> **count** – Number of source matrices when `mv` is a plain C array. It must be greater than zero.

---

dst – Destination array of the same size and the same depth as `mv[0]` . The number of channels will be the total number of channels in the matrix array.

The functions `merge` merge several arrays to make a single multi-channel array. That is, each element of the output array will be a concatenation of the elements of the input arrays, where elements of i-th input array are treated as `mv[i].channels()`-element vectors.

The function `split()` does the reverse operation. If you need to shuffle channels in some other advanced way, use `mixChannels()` .

See Also:

`mixChannels()`, `split()`, `Mat::reshape()`

min

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

C++: MatExpr **min**(const Mat& **src1**, const Mat& **src2**)

C++: MatExpr **min**(const Mat& **src1**, double **value**)

C++: MatExpr **min**(double **value**, const Mat& **src1**)

C++: void **min**(InputArray **src1**, InputArray **src2**, OutputArray **dst**)

C++: void **min**(const Mat& **src1**, const Mat& **src2**, Mat& **dst**)

C++: void **min**(const Mat& **src1**, double **value**, Mat& **dst**)

Python: cv2.**min**(src1, src2[, dst]) → dst

C: void **cvMin**(const CvArr* **src1**, const CvArr* **src2**, CvArr* **dst**)

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

Python: cv.**Min**(src1, src2, dst) → None

Python: cv.**MinS**(src, value, dst) → None

> **Parameters**

>> **src1** – First source array.

>> **src2** – Second source array of the same size and type as `src1` .

>> **value** – Real scalar value.

>> **dst** – Destination array of the same size and type as `src1` .

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

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

or array and a scalar:

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

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

The first three variants of the function listed above are actually a part of *Matrix Expressions* . They return the expression object that can be further either transformed/assigned to a matrix, or passed to a function, and so on.

See Also:

`max()`, `compare()`, `inRange()`, `minMaxLoc()`, *Matrix Expressions*

minMaxIdx

Finds the global minimum and maximum in an array

C++: void **minMaxIdx**(InputArray **src**, double* **minVal**, double* **maxVal**, int* **minIdx**=0, int* **maxIdx**=0, InputArray **mask**=noArray())

> **Parameters**
>
> > **src** – Source single-channel array.
> >
> > **minVal** – Pointer to the returned minimum value. NULL is used if not required.
> >
> > **maxVal** – Pointer to the returned maximum value. NULL is used if not required.
> >
> > **minIdx** – Pointer to the returned minimum location (in nD case). NULL is used if not required. Otherwise, it must point to an array of `src.dims` elements. The coordinates of the minimum element in each dimension are stored there sequentially.
> >
> > ---
> > **Note:** When `minIdx` is not NULL, it must have at least 2 elements (as well as `maxIdx`), even if `src` is a single-row or single-column matrix. In OpenCV (following MATLAB) each array has at least 2 dimensions, i.e. single-row matrix is `Mx1` matrix (and therefore `minIdx/maxIdx` will be `(i1,0)/(i2,0)`) and single-column matrix is `1xN` matrix (and therefore `minIdx/maxIdx` will be `(0,j1)/(0,j2)`).
> >
> > ---
> >
> > **maxIdx** – Pointer to the returned maximum location (in nD case). NULL is used if not required.

The function `minMaxIdx` finds the 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 function does not work with multi-channel arrays. If you need to find minimum or maximum elements across all the channels, use `Mat::reshape()` first to reinterpret the array as single-channel. Or you may extract the particular channel using either `extractImageCOI()`, or `mixChannels()`, or `split()`.

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

minMaxLoc

Finds the global minimum and maximum in an array.

C++: void **minMaxLoc**(InputArray **src**, double* **minVal**, double* **maxVal**=0, Point* **minLoc**=0, Point* **maxLoc**=0, InputArray **mask**=noArray())

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

Python: cv2.**minMaxLoc**(src[, mask]) → minVal, maxVal, minLoc, maxLoc

C: void **cvMinMaxLoc**(const CvArr* **arr**, double* **minVal**, double* **maxVal**, CvPoint* **minLoc**=NULL, CvPoint* **maxLoc**=NULL, const CvArr* **mask**=NULL)

Python: cv.**MinMaxLoc**(arr, mask=None)-> (minVal, maxVal, minLoc, maxLoc)

> **Parameters**
>
> > **src** – Source single-channel array.
> >
> > **minVal** – Pointer to the returned minimum value. NULL is used if not required.
> >
> > **maxVal** – Pointer to the returned maximum value. NULL is used if not required.

> > > **minLoc** – Pointer to the returned minimum location (in 2D case). `NULL` is used if not required.
> > >
> > > **maxLoc** – Pointer to the returned maximum location (in 2D case). `NULL` is used if not required.
> > >
> > > **mask** – Optional mask used to select a sub-array.

The functions `minMaxLoc` find the 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 `Mat::reshape()` first to reinterpret the array as single-channel. Or you may extract the particular channel using either `extractImageCOI()`, or `mixChannels()`, or `split()`.

See Also:

`max()`, `min()`, `compare()`, `inRange()`, `extractImageCOI()`, `mixChannels()`, `split()`, `Mat::reshape()`

mixChannels

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

C++: void **mixChannels**(const Mat* **src**, int **nsrc**, Mat* **dst**, int **ndst**, const int* **fromTo**, size_t **npairs**)

C++: void **mixChannels**(const vector<Mat>& **src**, vector<Mat>& **dst**, const int* **fromTo**, int **npairs**)

Python: cv2.**mixChannels**(src, dst, fromTo) → None

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

Python: cv.**MixChannels**(src, dst, fromTo) → None

> > Parameters

> > > **src** – Input array or vector of matrices. All the matrices must have the same size and the same depth.
> > >
> > > **nsrc** – Number of matrices in `src`.
> > >
> > > **dst** – Output array or vector of matrices. All the matrices *must be allocated*. Their size and depth must be the same as in `src[0]`.
> > >
> > > **ndst** – Number of matrices in `dst`.
> > >
> > > **fromTo** – Array of index pairs specifying which channels are copied and where. `fromTo[k*2]` is a 0-based index of the input channel in `src`. `fromTo[k*2+1]` is an index of the output channel in `dst`. The continuous channel numbering is used: the first input image channels are indexed from 0 to `src[0].channels()-1`, the second input image channels are indexed from `src[0].channels()` to `src[0].channels() + src[1].channels()-1`, and so on. 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** – Number of index pairs in `fromTo`.

The functions `mixChannels` provide an advanced mechanism for shuffling image channels.

`split()` and `merge()` and some forms of `cvtColor()` are partial cases of `mixChannels`.

In the example below, the code splits a 4-channel RGBA image into a 3-channel BGR (with R and B channels swapped) and a 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 an array of matrices is a quite efficient operation,
// 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: Unlike many other new-style C++ functions in OpenCV (see the introduction section and `Mat::create()`), `mixChannels` requires the destination arrays to be pre-allocated before calling the function.

See Also:

`split()`, `merge()`, `cvtColor()`

mulSpectrums

Performs the per-element multiplication of two Fourier spectrums.

C++: void **mulSpectrums**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, int **flags**, bool **conj**=false)

Python: cv2.**mulSpectrums**(a, b, flags[, c[, conjB]]) → c

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

Python: cv.**MulSpectrums**(src1, src2, dst, flags) → None

> **Parameters**
>
> > **src1** – First source array.
> >
> > **src2** – Second source array of the same size and type as `src1`.
> >
> > **dst** – Destination array of the same size and type as `src1`.
> >
> > **flags** – Operation flags. Currently, the only supported flag is `DFT_ROWS`, which indicates that each row of `src1` and `src2` is an independent 1D Fourier spectrum.
> >
> > **conj** – Optional flag that conjugates the second source array before the multiplication (true) or not (false).

The function `mulSpectrums` performs the 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=false`) of two arrays rapidly. When the arrays are complex, they are simply multiplied (per element) with an optional conjugation of the second-array elements. When the arrays are real, they are assumed to be CCS-packed (see `dft()` for details).

multiply

Calculates the per-element scaled product of two arrays.

C++: void **multiply**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, double **scale**=1)

Python: cv2.**multiply**(src1, src2[, dst[, scale[, dtype]]]) → dst

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

Python: cv.**Mul**(src1, src2, dst, scale) → None

> **Parameters**

> > **src1** – First source array.

> > **src2** – Second source array of the same size and the same type as src1 .

> > **dst** – Destination array of the same size and type as src1 .

> > **scale** – Optional scale factor.

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

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

There is also a *Matrix Expressions* -friendly variant of the first function. See Mat::mul() .

For a not-per-element matrix product, see gemm() .

See Also:

add(), subtract(), divide(), *Matrix Expressions*, scaleAdd(), addWeighted(), accumulate(), accumulateProduct(), accumulateSquare(), Mat::convertTo()

mulTransposed

Calculates the product of a matrix and its transposition.

C++: void **mulTransposed**(InputArray **src**, OutputArray **dst**, bool **aTa**, InputArray **delta**=noArray(), double **scale**=1, int **rtype**=-1)

Python: cv2.**mulTransposed**(src, aTa[, dst[, delta[, scale[, dtype]]]]) → dst

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

Python: cv.**MulTransposed**(src, dst, order, delta=None, scale) → None

> **Parameters**

> > **src** – Source single-channel matrix. Note that unlike gemm(), the function can multiply not only floating-point matrices.

> > **dst** – Destination square matrix.

> > **aTa** – Flag specifying the multiplication ordering. See the description below.

> > **delta** – Optional delta matrix subtracted from src before the multiplication. When the matrix is empty (delta=noArray()), it is assumed to be zero, that is, nothing is subtracted. If it has the same size as src , it is simply subtracted. Otherwise, it is "repeated" (see repeat()) to cover the full src and then subtracted. Type of the delta matrix, when it is not empty, must be the same as the type of created destination matrix. See the rtype parameter description below.

> > **scale** – Optional scale factor for the matrix product.

> > **rtype** – Optional type of the destination matrix. When it is negative, the destination matrix will have the same type as src . Otherwise, it will be type=CV_MAT_DEPTH(rtype) that should be either CV_32F or CV_64F .

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

$$dst = scale(src - delta)^T(src - delta)$$

if `aTa=true`, and

$$dst = scale(src - delta)(src - delta)^T$$

otherwise. The function is used to compute the covariance matrix. With zero delta, it can be used as a faster substitute for general matrix product A*B when B=A'

See Also:

`calcCovarMatrix()`, `gemm()`, `repeat()`, `reduce()`

norm

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

C++: double **norm**(InputArray **src1**, int **normType**=NORM_L2, InputArray **mask**=noArray())

C++: double **norm**(InputArray **src1**, InputArray **src2**, int **normType**, InputArray **mask**=noArray())

C++: double **norm**(const SparseMat& **src**, int **normType**)

Python: cv2.**norm**(src1[, normType[, mask]]) $\rightarrow$ retval

Python: cv2.**norm**(src1, src2[, normType[, mask]]) $\rightarrow$ retval

C: double **cvNorm**(const CvArr* **arr1**, const CvArr* **arr2**=NULL, int **normType**=CV_L2, const CvArr* **mask**=NULL)

Python: cv.**Norm**(arr1, arr2, normType=CV_L2, mask=None) $\rightarrow$ double

> **Parameters**
>> **src1** – First source array.
>>
>> **src2** – Second source array of the same size and the same type as `src1`.
>>
>> **normType** – Type of the norm. See the details below.
>>
>> **mask** – Optional operation mask. It must have the same size as `src1` and CV_8UC1 type.

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

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

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

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

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 the `mask` parameter is specified and it is not empty, the norm is computed only over the region specified by the mask.

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

normalize

Normalizes the norm or value range of an array.

C++: void **normalize**(const InputArray **src**, OutputArray **dst**, double **alpha**=1, double **beta**=0, int **norm-Type**=NORM_L2, int **rtype**=-1, InputArray **mask**=noArray())

C++: void **normalize**(const SparseMat& **src**, SparseMat& **dst**, double **alpha**, int **normType**)

Python: cv2.**normalize**(src[, dst[, alpha[, beta[, norm_type[, dtype[, mask]]]]]]) → dst

> **Parameters**
>
> > **src** – Source array.
> >
> > **dst** – Destination array of the same size as `src` .
> >
> > **alpha** – Norm value to normalize to or the lower range boundary in case of the range normalization.
> >
> > **beta** – Upper range boundary in case ofthe range normalization. It is not used for the norm normalization.
> >
> > **normType** – Normalization type. See the details below.
> >
> > **rtype** – When the parameter is negative, the destination array has the same type as `src`. Otherwise, it has the same number of channels as `src` and the depth =CV_MAT_DEPTH(rtype) .
> >
> > **mask** – Optional operation mask.

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

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

(where p=Inf, 1 or 2) when `normType=NORM_INF, NORM_L1,` or `NORM_L2`, respectively; or so that

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

when `normType=NORM_MINMAX` (for dense arrays only). The optional mask specifies a sub-array to be normalized. This means that 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::convertTo()`.

In 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::convertTo()`, `SparseMat::convertTo()`

PCA

class PCA

Principal Component Analysis class.

The class is used to compute a special basis for a set of vectors. The basis will consist of eigenvectors of the co-variance matrix computed from the input set of vectors. The class PCA can also 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 its first few components, corresponding to the eigenvectors of the largest eigenvalues of the covariance matrix. Geometrically it means that you compute a projection of the vector to a subspace formed by a few eigenvectors corresponding to the dominant eigenvalues of the covariance matrix. And usually such a projection is very close to the original vector. So, you 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 .

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

```
PCA compressPCA(InputArray pcaset, int maxComponents,
                const Mat& testset, OutputArray compressed)
{
    PCA pca(pcaset, // pass the data
            Mat(), // there is no pre-computed mean vector,
                   // 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 vectors are
                                // the matrix columns)
            maxComponents // specify how many principal components to retain
            );
    // if there is no test data, just return the computed basis, ready-to-use
    if( !testset.data )
        return pca;
    CV_Assert( testset.cols == pcaset.cols );

    compressed.create(testset.rows, maxComponents, testset.type());

    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("
    }
    return pca;
}
```

See Also:

calcCovarMatrix(), mulTransposed(), SVD, dft(), dct()

PCA::PCA

PCA constructors

C++: `PCA::`**`PCA`**`()`

C++: `PCA::`**`PCA`**`(`InputArray **data**, InputArray **mean**, int **flags**, int **maxComponents**=0)

> **Parameters**
>
> > **data** – Input samples stored as matrix rows or matrix columns.
> >
> > **mean** – Optional mean value. If the matrix is empty (`noArray()`), the mean is computed from the data.
> >
> > **flags** – Operation flags. Currently the parameter is only used to specify the data layout.
> >
> > – **CV_PCA_DATA_AS_ROW** indicates that the input samples are stored as matrix rows.
> >
> > – **CV_PCA_DATA_AS_COL** indicates that the input samples are stored as matrix columns.
> >
> > **maxComponents** – Maximum number of components that PCA should retain. By default, all the components are retained.

The default constructor initializes an empty PCA structure. The second constructor initializes the structure and calls `PCA::operator()` .

PCA::operator ()

Performs Principal Component Analysis of the supplied dataset.

C++: PCA& PCA::**`operator()`**`(`InputArray **data**, InputArray **mean**, int **flags**, int **maxComponents**=0)

Python: `cv2.`**`PCACompute`**`(`data$\left[$, mean$\left[$, eigenvectors$\left[$, maxComponents $\right]\right]\right]$) $\rightarrow$ mean, eigenvectors

> **Parameters**
>
> > **data** – Input samples stored as the matrix rows or as the matrix columns.
> >
> > **mean** – Optional mean value. If the matrix is empty (`noArray()`), the mean is computed from the data.
> >
> > **flags** – Operation flags. Currently the parameter is only used to specify the data layout.
> >
> > – **CV_PCA_DATA_AS_ROW** indicates that the input samples are stored as matrix rows.
> >
> > – **CV_PCA_DATA_AS_COL** indicates that the input samples are stored as matrix columns.
> >
> > **maxComponents** – Maximum number of components that PCA should retain. By default, all the components are retained.

The operator performs PCA of the supplied dataset. It is safe to reuse the same PCA structure for multiple datasets. That is, if the structure has been previously used with another dataset, the existing internal data is reclaimed and the new `eigenvalues`, `eigenvectors` , and `mean` are allocated and computed.

The computed eigenvalues are sorted from the largest to the smallest and the corresponding eigenvectors are stored as `PCA::eigenvectors` rows.

PCA::project

Projects vector(s) to the principal component subspace.

C++: Mat PCA::**project**(InputArray **vec**) const

C++: void PCA::**project**(InputArray **vec**, OutputArray **result**) const

Python: cv2.**PCAProject**(vec, mean, eigenvectors[, result]) → result

> **Parameters**
>
> > **vec** – Input vector(s). They must have the same dimensionality and the same layout as the input data used at PCA phase. That is, if CV_PCA_DATA_AS_ROW are specified, then vec.cols==data.cols (vector dimensionality) and vec.rows is the number of vectors to project. The same is true for the CV_PCA_DATA_AS_COL case.
> >
> > **result** – Output vectors. In case of CV_PCA_DATA_AS_COL , the output matrix has as many columns as the number of input vectors. This means that result.cols==vec.cols and the number of rows match the number of principal components (for example, maxComponents parameter passed to the constructor).

The methods project one or more vectors to the principal component subspace, where each vector projection is represented by coefficients in the principal component basis. The first form of the method returns the matrix that the second form writes to the result. So the first form can be used as a part of expression while the second form can be more efficient in a processing loop.

PCA::backProject

Reconstructs vectors from their PC projections.

C++: Mat PCA::**backProject**(InputArray **vec**) const

C++: void PCA::**backProject**(InputArray **vec**, OutputArray **result**) const

Python: cv2.**PCABackProject**(vec, mean, eigenvectors[, result]) → result

> **Parameters**
>
> > **vec** – Coordinates of the vectors in the principal component subspace. The layout and size are the same as of PCA::project output vectors.
> >
> > **result** – Reconstructed vectors. The layout and size are the same as of PCA::project input vectors.

The methods are inverse operations to PCA::project() . They take PC coordinates of projected vectors and reconstruct the original vectors. Unless all the principal components have been retained, the reconstructed vectors are different from the originals. But typically, the difference is small if the number of components is large enough (but still much smaller than the original vector dimensionality). As a result, PCA is used.

perspectiveTransform

Performs the perspective matrix transformation of vectors.

C++: void **perspectiveTransform**(InputArray **src**, OutputArray **dst**, InputArray **mtx**)

Python: cv2.**perspectiveTransform**(src, m[, dst]) → dst

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

Python: cv.**PerspectiveTransform**(src, dst, mat) → None

Parameters

src – Source two-channel or three-channel floating-point array. Each element is a 2D/3D vector to be transformed.

dst – Destination array of the same size and type as `src` .

mtx – 3x3 or 4x4 floating-point transformation matrix.

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

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

where

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

and

$$w = \left\{ \begin{array}{ll} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{array} \right.$$

Here a 3D vector transformation is shown. In case of a 2D vector transformation, the `z` component is omitted.

Note: 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 problem, that is, you 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

Calculates the rotation angle of 2D vectors.

C++: void **phase**(InputArray **x**, InputArray **y**, OutputArray **angle**, bool **angleInDegrees**=false)

Python: `cv2.phase`(x, y[, angle[, angleInDegrees]]) → angle

Parameters

x – Source floating-point array of x-coordinates of 2D vectors.

y – Source array of y-coordinates of 2D vectors. It must have the same size and the same type as `x` .

angle – Destination array of vector angles. It has the same size and same type as `x` .

angleInDegrees – When it is true, the function computes the angle in degrees. Otherwise, they are 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` :

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

The angle estimation accuracy is about 0.3 degrees. When `x(I)=y(I)=0` , the corresponding `angle(I)` is set to 0.

---

polarToCart

Computes x and y coordinates of 2D vectors from their magnitude and angle.

C++: void **polarToCart**(InputArray **magnitude**, InputArray **angle**, OutputArray **x**, OutputArray **y**, bool **angleInDegrees**=false)

Python: cv2.**polarToCart**(magnitude, angle[, x[, y[, angleInDegrees]]]) → x, y

C: void **cvPolarToCart**(const CvArr* **magnitude**, const CvArr* **angle**, CvArr* **x**, CvArr* **y**, int **angleInDegrees**=0)

Python: cv.**PolarToCart**(magnitude, angle, x, y, angleInDegrees=0) → None

> **Parameters**

>> **magnitude** – 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 is not empty, it must have the same size and type as angle .

>> **angle** – Source floating-point array of angles of 2D vectors.

>> **x** – Destination array of x-coordinates of 2D vectors. It has the same size and type as angle.

>> **y** – Destination array of y-coordinates of 2D vectors. It has the same size and 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 :

$$x(I) = \mathtt{magnitude}(I)\cos(\mathtt{angle}(I))$$
$$y(I) = \mathtt{magnitude}(I)\sin(\mathtt{angle}(I))$$

The relative accuracy of the estimated coordinates is about 1e-6.

See Also:

cartToPolar(), magnitude(), phase(), exp(), log(), pow(), sqrt()

pow

Raises every array element to a power.

C++: void **pow**(InputArray **src**, double **p**, OutputArray **dst**)

Python: cv2.**pow**(src, power[, dst]) → dst

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

Python: cv.**Pow**(src, dst, power) → None

> **Parameters**

>> **src** – Source array.

>> **p** – Exponent of power.

>> **dst** – Destination array of the same size and type as src .

The function pow raises every element of the input array to p :

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

So, 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. In the example below, 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()

RNG

class RNG

Random number generator. It encapsulates the state (currently, a 64-bit integer) and has methods to return scalar random values and to fill arrays with random values. Currently it supports uniform and Gaussian (normal) distributions. The generator uses Multiply-With-Carry algorithm, introduced by G. Marsaglia (http://en.wikipedia.org/wiki/Multiply-with-carry). Gaussian-distribution random numbers are generated using the Ziggurat algorithm (http://en.wikipedia.org/wiki/Ziggurat_algorithm), introduced by G. Marsaglia and W. W. Tsang.

RNG::RNG

The constructors

C++: RNG::**RNG**()

C++: RNG::**RNG**(uint64 **state**)

> **Parameters**
>
> > **state** – 64-bit value used to initialize the RNG.

These are the RNG constructors. The first form sets the state to some pre-defined value, equal to `2**32-1` in the current implementation. The second form sets the state to the specified value. If you passed `state=0` , the constructor uses the above default value instead to avoid the singular random number sequence, consisting of all zeros.

RNG::next

Returns the next random number.

C++: unsigned int RNG::**next**()

The method updates the state using the MWC algorithm and returns the next 32-bit random number.

RNG::operator T

Returns the next random number of the specified type.

C++: RNG::**operator** uchar()

C++: RNG::**operator** schar()

C++: RNG::**operator** ushort()

C++: RNG::**operator** short int()

C++: RNG::**operator** int()

C++: RNG::**operator** unsigned int()

C++: RNG::**operator** float()

C++: RNG::**operator** double()

Each of the methods updates the state using the MWC algorithm and returns the next random number of the specified type. In case of integer types, the returned number is from the available value range for the specified type. In case of floating-point types, the returned value is from [0,1) range.

RNG::operator ()

Returns the next random number.

C++: unsigned int RNG::**operator()**()

C++: unsigned int RNG::**operator()**(unsigned int **N**)

> **Parameters**

> > **N** – Upper non-inclusive boundary of the returned random number.

The methods transform the state using the MWC algorithm and return the next random number. The first form is equivalent to RNG::next() . The second form returns the random number modulo N , which means that the result is in the range [0, N) .

RNG::uniform

Returns the next random number sampled from the uniform distribution.

C++: int RNG::**uniform**(int **a**, int **b**)

C++: float RNG::**uniform**(float **a**, float **b**)

C++: double RNG::**uniform**(double **a**, double **b**)

> **Parameters**

> > **a** – Lower inclusive boundary of the returned random numbers.

> > **b** – Upper non-inclusive boundary of the returned random numbers.

The methods transform the state using the MWC algorithm and return the next uniformly-distributed random number of the specified type, deduced from the input parameter type, from the range [a, b) . There is a nuance illustrated by the following sample:

```
RNG rng;

// always produces 0
double a = rng.uniform(0, 1);

// produces double from [0, 1)
double a1 = rng.uniform((double)0, (double)1);

// produces float from [0, 1)
double b = rng.uniform(0.f, 1.f);

// produces double from [0, 1)
```

```
double c = rng.uniform(0., 1.);

// may cause compiler error because of ambiguity:
//  RNG::uniform(0, (int)0.999999)? or RNG::uniform((double)0, 0.99999)?
double d = rng.uniform(0, 0.999999);
```

The compiler does not take into account the type of the variable to which you assign the result of `RNG::uniform` . The only thing that matters to the compiler is the type of a and b parameters. So, if you want a floating-point random number, but the range boundaries are integer numbers, either put dots in the end, if they are constants, or use explicit type cast operators, as in the a1 initialization above.

RNG::gaussian

Returns the next random number sampled from the Gaussian distribution.

C++: double RNG::**gaussian**(double **sigma**)

>   **Parameters**

>>   **sigma** – Standard deviation of the distribution.

The method transforms the state using the MWC algorithm and returns the next random number from the Gaussian distribution `N(0,sigma)` . That is, the mean value of the returned random numbers is zero and the standard deviation is the specified `sigma` .

RNG::fill

Fills arrays with random numbers.

C++: void RNG::**fill**(InputOutputArray **mat**, int **distType**, InputArray **a**, InputArray **b**)

>   **Parameters**

>>   **mat** – 2D or N-dimensional matrix. Currently matrices with more than 4 channels are not supported by the methods. Use `Mat::reshape()` as a possible workaround.

>>   **distType** – Distribution type, `RNG::UNIFORM` or `RNG::NORMAL` .

>>   **a** – First distribution parameter. In case of the uniform distribution, this is an inclusive lower boundary. In case of the normal distribution, this is a mean value.

>>   **b** – Second distribution parameter. In case of the uniform distribution, this is a non-inclusive upper boundary. In case of the normal distribution, this is a standard deviation (diagonal of the standard deviation matrix or the full standard deviation matrix).

Each of the methods fills the matrix with the random values from the specified distribution. As the new numbers are generated, the RNG state is updated accordingly. In case of multiple-channel images, every channel is filled independently, which means that RNG cannot generate samples from the multi-dimensional Gaussian distribution with non-diagonal covariance matrix directly. To do that, the method generates samples from multi-dimensional standard Gaussian distribution with zero mean and identity covariation matrix, and then transforms them using `transform()` to get samples from the specified Gaussian distribution.

randu

Generates a single uniformly-distributed random number or an array of random numbers.

C++: template<typename _Tp> _Tp **randu**()

C++: void **randu**(InputOutputArray **mtx**, InputArray **low**, InputArray **high**)

Python: cv2.**randu**(dst, low, high) → None

>>> **Parameters**

>>>> **mtx** – Output array of random numbers. The array must be pre-allocated.

>>>> **low** – Inclusive lower boundary of the generated random numbers.

>>>> **high** – 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 an equivalent to (int)theRNG(); , and so on. See RNG description.

The second non-template variant of the function fills the matrix mtx with uniformly-distributed random numbers from the specified range:

$$\texttt{low}_c \leq \texttt{mtx(I)}_c < \texttt{high}_c$$

See Also:

RNG, randn(), theRNG()

randn

Fills the array with normally distributed random numbers.

C++: void **randn**(InputOutputArray **mtx**, InputArray **mean**, InputArray **stddev**)

Python: cv2.**randn**(dst, mean, stddev) → None

>>> **Parameters**

>>>> **mtx** – Output array of random numbers. The array must be pre-allocated and have 1 to 4 channels.

>>>> **mean** – Mean value (expectation) of the generated random numbers.

>>>> **stddev** – Standard deviation of the generated random numbers. It can be either a vector (in which case a diagonal standard deviation matrix is assumed) or a square matrix.

The function randn fills the matrix mtx with normally distributed random numbers with the specified mean vector and the standard deviation matrix. The generated random numbers are clipped to fit the value range of the destination array data type.

See Also:

RNG, randu()

randShuffle

Shuffles the array elements randomly.

C++: void **randShuffle**(InputOutputArray **mtx**, double **iterFactor**=1., RNG* **rng**=0)

Python: cv2.**randShuffle**(src$\left[$, dst$\left[$, iterFactor$\right]\right]$) → dst

>>> **Parameters**

>>>> **mtx** – Input/output numerical 1D array.

> **iterFactor** – Scale factor that determines the number of random swap operations. See the details below.
>
> **rng** – Optional random number generator used for shuffling. If it is zero, `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

Reduces a matrix to a vector.

C++: void **reduce**(InputArray **mtx**, OutputArray **vec**, int **dim**, int **reduceOp**, int **dtype**=-1)

Python: cv2.**reduce**(src, dim, rtype$\left[, \text{dst}\left[, \text{dtype}\right]\right]$) → dst

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

Python: cv.**Reduce**(src, dst, dim=-1, op=CV_REDUCE_SUM) → None

> **Parameters**
>
> > **mtx** – Source 2D matrix.
> >
> > **vec** – Destination vector. Its size and type is defined by `dim` and `dtype` parameters.
> >
> > **dim** – Dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row. 1 means that the matrix is reduced to a single column.
> >
> > **reduceOp** – Reduction operation that could be one of the following:
> >
> > – **CV_REDUCE_SUM** The output is the sum of all rows/columns of the matrix.
> >
> > – **CV_REDUCE_AVG** The output is the mean vector of all rows/columns of the matrix.
> >
> > – **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all rows/columns of the matrix.
> >
> > – **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all rows/columns of the matrix.
> >
> > **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 the 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 a raster image. In case of `CV_REDUCE_SUM` and `CV_REDUCE_AVG` , the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

See Also:

`repeat()`

repeat

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

C++: void **repeat**(InputArray **src**, int **ny**, int **nx**, OutputArray **dst**)

C++: Mat **repeat**(InputArray **src**, int **ny**, int **nx**)

Python: cv2.**repeat**(src, ny, nx[, dst]) → dst

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

Python: cv.**Repeat**(src, dst) → None

>> **Parameters**

>>> **src** – Source array to replicate.

>>> **dst** – Destination array of the same type as src .

>>> **ny** – Flag to specify how many times the src is repeated along the vertical axis.

>>> **nx** – Flag to specify 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:

$$\mathtt{dst}_{ij} = \mathtt{src}_{i \mod \mathtt{src.rows}, \, j \mod \mathtt{src.cols}}$$

The second variant of the function is more convenient to use with *Matrix Expressions* .

See Also:

reduce(), *Matrix Expressions*

scaleAdd

Calculates the sum of a scaled array and another array.

C++: void **scaleAdd**(InputArray **src1**, double **scale**, InputArray **src2**, OutputArray **dst**)

Python: cv2.**scaleAdd**(src1, alpha, src2[, dst]) → dst

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

Python: cv.**ScaleAdd**(src1, scale, src2, dst) → None

>> **Parameters**

>>> **src1** – First source array.

>>> **scale** – Scale factor for the first array.

>>> **src2** – Second source array of the same size and type as src1 .

>>> **dst** – Destination array of the same size and type as src1 .

The function scaleAdd is one of the classical primitive linear algebra operations, known as DAXPY or SAXPY in BLAS. It calculates the sum of a scaled array and another array:

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

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

Initializes a scaled identity matrix.

C++: void **setIdentity**(InputOutputArray **dst**, const Scalar& **value**=Scalar(1))

Python: cv2.**setIdentity**(mtx$\big[$, s$\big]$) $\rightarrow$ None

C: void **cvSetIdentity**(CvArr* **mat**, CvScalar **value**=cvRealScalar(1))

Python: cv.**SetIdentity**(mat, value=1) $\rightarrow$ None

> **Parameters**
>
> > **dst** – Matrix to initialize (not necessarily square).
> >
> > **value** – Value to assign to diagonal elements.

The function setIdentity() initializes a scaled identity matrix:

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

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, 0, 0]]
```

See Also:

Mat::zeros(), Mat::ones(), *Matrix Expressions*, Mat::setTo(), Mat::operator=()

solve

Solves one or more linear systems or least-squares problems.

C++: bool **solve**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, int **flags**=DECOMP_LU)

Python: cv2.**solve**(src1, src2$\big[$, dst$\big[$, flags$\big]\big]$) $\rightarrow$ retval, dst

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

Python: cv.**Solve**(A, B, X, method=CV_LU) $\rightarrow$ None

> **Parameters**
>
> > **src1** – Input matrix on the left-hand side of the system.
> >
> > **src2** – Input matrix on the right-hand side of the system.
> >
> > **dst** – Output solution.
> >
> > **flags** – 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 $\mathsf{src1}^\mathsf{T} \cdot \mathsf{src1} \cdot \mathsf{dst} = \mathsf{src1}^\mathsf{T}\mathsf{src2}$ are solved instead of the original system $\mathsf{src1} \cdot \mathsf{dst} = \mathsf{src2}$.

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`):

$$\mathsf{dst} = \arg\min_X \|\mathsf{src1} \cdot X - \mathsf{src2}\|$$

If `DECOMP_LU` or `DECOMP_CHOLESKY` method is used, the function returns 1 if `src1` (or $\mathsf{src1}^\mathsf{T}\mathsf{src1}$) is non-singular. Otherwise, it returns 0. In the latter case, `dst` is not valid. Other methods find a pseudo-solution in case of a singular left-hand side part.

Note: If you want to find a unity-norm solution of an under-defined singular system $\mathsf{src1} \cdot \mathsf{dst} = 0$, the function `solve` will not do the work. Use `SVD::solveZ()` instead.

See Also:

`invert()`, `SVD`, `eigen()`

solveCubic

Finds the real roots of a cubic equation.

C++: void **solveCubic**(InputArray **coeffs**, OutputArray **roots**)

Python: cv2.**solveCubic**(coeffs[, roots]) → retval, roots

C: void **cvSolveCubic**(const CvArr* **coeffs**, CvArr* **roots**)

Python: cv.**SolveCubic**(coeffs, roots) → None

> **Parameters**
>
> > **coeffs** – Equation coefficients, an array of 3 or 4 elements.
> >
> > **roots** – Destination array of real roots that has 1 or 3 elements.

The function `solveCubic` finds the real roots of a cubic equation:

- if `coeffs` is a 4-element vector:

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

- if `coeffs` is a 3-element vector:

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

The roots are stored in the `roots` array.

solvePoly

Finds the real or complex roots of a polynomial equation.

C++: void **solvePoly**(InputArray **coeffs**, OutputArray **roots**, int **maxIters**=300)

Python: cv2.**solvePoly**(coeffs[, roots[, maxIters]]) → retval, roots

> **Parameters**
>
> > **coeffs** – Array of polynomial coefficients.
> >
> > **roots** – Destination (complex) array of roots.
> >
> > **maxIters** – Maximum number of iterations the algorithm does.

The function solvePoly finds real and complex roots of a polynomial equation:

$$\texttt{coeffs}[n]x^n + \texttt{coeffs}[n-1]x^{n-1} + ... + \texttt{coeffs}[1]x + \texttt{coeffs}[0] = 0$$

sort

Sorts each row or each column of a matrix.

C++: void **sort**(InputArray **src**, OutputArray **dst**, int **flags**)

Python: cv2.**sort**(src, flags[, dst]) → dst

> **Parameters**
>
> > **src** – Source single-channel array.
> >
> > **dst** – Destination array of the same size and type as src .
> >
> > **flags** – 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. So you should pass two operation flags to get desired behaviour. 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

Sorts each row or each column of a matrix.

C++: void **sortIdx**(InputArray **src**, OutputArray **dst**, int **flags**)

Python: cv2.**sortIdx**(src, flags[, dst]) → dst

> **Parameters**

> **src** – Source single-channel array.
>
> **dst** – Destination integer array of the same size as `src` .
>
> **flags** – Operation flags that could be 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 the ascending or descending order. So you should pass two operation flags to get desired behaviour. 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 possible):
// [[1, 2, 0], [0, 2, 1], [0, 1, 2]]
```

See Also:

`sort()`, `randShuffle()`

split

Divides a multi-channel array into several single-channel arrays.

C++: void **split**(const Mat& **mtx**, Mat* **mv**)

C++: void **split**(const Mat& **mtx**, vector<Mat>& **mv**)

Python: cv2.**split**(m, mv) $\rightarrow$ None

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

Python: cv.**Split**(src, dst0, dst1, dst2, dst3) $\rightarrow$ None

> **Parameters**
>
> > **mtx** – Source multi-channel array.
> >
> > **mv** – Destination array or vector of arrays. In the first variant of the function the number of arrays must match `mtx.channels()` . The arrays themselves are reallocated, if needed.

The functions `split` split a multi-channel array into separate single-channel arrays:

$$\mathtt{mv}[\mathrm{c}](\mathrm{I}) = \mathtt{mtx}(\mathrm{I})_{\mathrm{c}}$$

If you need to extract a single channel or do some other sophisticated channel permutation, use `mixChannels()` .

See Also:

`merge()`, `mixChannels()`, `cvtColor()`

sqrt

Calculates a quare root of array elements.

C++: void **sqrt**(InputArray **src**, OutputArray **dst**)

Python: cv2.**sqrt**(src[, dst]) → dst

C: float **cvSqrt**(float **value**)

Python: cv.**Sqrt**(value) → float

> **Parameters**
>
>> **src** – Source floating-point array.
>>
>> **dst** – Destination array of the same size and type as `src` .

The functions `sqrt` calculate a square root of each source array element. In case of multi-channel arrays, each channel is processed independently. The accuracy is approximately the same as of the built-in `std::sqrt` .

See Also:

`pow()`, `magnitude()`

subtract

Calculates the per-element difference between two arrays or array and a scalar.

C++: void **subtract**(InputArray **src1**, InputArray **src2**, OutputArray **dst**, InputArray **mask**=noArray(), int **dtype**=-1)

Python: cv2.**subtract**(src1, src2[, dst[, mask[, dtype]]]) → dst

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

C: void **cvSubRS**(const CvArr* **src1**, CvScalar **src2**, CvArr* **dst**, const CvArr* **mask**=NULL)

C: void **cvSubS**(const CvArr* **src1**, CvScalar **src2**, CvArr* **dst**, const CvArr* **mask**=NULL)

Python: cv.**Sub**(src1, src2, dst, mask=None) → None

Python: cv.**SubRS**(src1, src2, dst, mask=None) → None

Python: cv.**SubS**(src1, src2, dst, mask=None) → None

> **Parameters**
>
>> **src1** – First source array or a scalar.
>>
>> **src2** – Second source array or a scalar.
>>
>> **dst** – Destination array of the same size and the same number of channels as the input array.
>>
>> **mask** – Optional operation mask. This is an 8-bit single channel array that specifies elements of the destination array to be changed.
>>
>> **dtype** – Optional depth of the output array. See the details below.

The function `subtract` computes:

- Difference between two arrays, when both input arrays have the same size and the same number of channels:

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

- Difference between an array and a scalar, when `src2` is constructed from `Scalar` or has the same number of elements as `src1.channels()`:

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

- Difference between a scalar and an array, when `src1` is constructed from `Scalar` or has the same number of elements as `src2.channels()`:

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

- The reverse difference between a scalar and an array in the case of `SubRS`:

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

where I is a multi-dimensional index of array elements. In case of multi-channel arrays, each channel is processed independently.

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

```
dst = src1 - src2;
dst -= src1; // equivalent to subtract(dst, src1, dst);
```

The input arrays and the destination array can all have the same or different depths. For example, you can subtract to 8-bit unsigned arrays and store the difference in a 16-bit signed array. Depth of the output array is determined by `dtype` parameter. In the second and third cases above, as well as in the first case, when `src1.depth() == src2.depth()`, `dtype` can be set to the default `-1`. In this case the output array will have the same depth as the input array, be it `src1`, `src2` or both.

See Also:

`add()`, `addWeighted()`, `scaleAdd()`, `Mat::convertTo()`, *Matrix Expressions*

SVD

class SVD

Class for computing Singular Value Decomposition of a floating-point matrix. The Singular Value Decomposition is used to solve least-square problems, under-determined linear systems, invert matrices, compute condition numbers, and so on.

For a faster operation, you can pass `flags=SVD::MODIFY_A|...` to modify the decomposed matrix when it is not necessary to preserve it. If you want to compute a condition number of a matrix or an absolute value of its determinant, you do not need u and vt . 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()`

SVD::SVD

The constructors.

C++: `SVD::`**`SVD`**`()`

C++: `SVD::`**`SVD`**`(InputArray **A**, int `**flags**`=0 )`

      Parameters

>   **src** – Decomposed matrix.
>
>   **flags** – Operation flags.
>
>   – **SVD::MODIFY_A** Use the algorithm to modify the decomposed matrix. It can save
>   space and speed up processing.
>
>   – **SVD::NO_UV** Indicate that only a vector of singular values w is to be computed, while u
>   and vt will be set to empty matrices.
>
>   – **SVD::FULL_UV** When the matrix is not square, by default the algorithm produces u
>   and vt matrices of sufficiently large size for the further A reconstruction. If, however,
>   FULL_UV flag is specified, u and vt will be full-size square orthogonal matrices.

The first constructor initializes an empty SVD structure. The second constructor initializes an empty SVD structure and
then calls `SVD::operator()`.

SVD::operator ()

Performs SVD of a matrix.

C++: SVD& SVD::**operator()**(InputArray **src**, int **flags**=0)

>   **Parameters**
>
>   >   **src** – Decomposed matrix.
>   >
>   >   **flags** – Operation flags.
>   >
>   >   – **SVD::MODIFY_A** Use the algorithm to modify the decomposed matrix. It can save
>   >   space and speed up processing.
>   >
>   >   – **SVD::NO_UV** Use only singular values. The algorithm does not compute u and vt
>   >   matrices.
>   >
>   >   – **SVD::FULL_UV** When the matrix is not square, by default the algorithm produces u
>   >   and vt matrices of sufficiently large size for the further A reconstruction. If, however, the
>   >   FULL_UV flag is specified, u and vt are full-size square orthogonal matrices.

The operator performs the singular value decomposition of the supplied matrix. The u,``vt`` , and the vector of singular
values w are stored in the structure. The same SVD structure can be reused many times with different matrices. Each
time, if needed, the previous u,``vt`` , and w are reclaimed and the new matrices are created, which is all handled by
`Mat::create()`.

SVD::compute

Performs SVD of a matrix

C++: **static** void SVD::**compute**(InputArray **src**, OutputArray **w**, OutputArray **u**, OutputArray **vt**, int **flags**=0
>   )

C++: **static** void SVD::**compute**(InputArray **src**, OutputArray **w**, int **flags**=0)

Python: cv2.**SVDecomp**(src[, w[, u[, vt[, flags]]]]) → w, u, vt

C: void **cvSVD**(CvArr* **src**, CvArr* **w**, CvArr* **u**=NULL, CvArr* **v**=NULL, int **flags**=0)

Python: cv.**SVD**(src, w, u=None, v=None, flags=0) → None

>   **Parameters**
>
>   >   **src** – Decomposed matrix

> **w** – Computed singular values
>
> **u** – Computed left singular vectors
>
> **v** – Computed right singular vectors
>
> **vt** – Transposed matrix of right singular values
>
> **flags** – Opertion flags - see `SVD::SVD()`.

The methods/functions perform SVD of matrix. Unlike `SVD::SVD` constructor and `SVD::operator()`, they store the results to the user-provided matrices.

```
Mat A, w, u, vt;
SVD::compute(A, w, u, vt);
```

SVD::solveZ

Solves an under-determined singular linear system.

C++: **static** void SVD::**solveZ**(InputArray **src**, OutputArray **dst**)

> **Parameters**
>
> > **src** – Left-hand-side matrix.
> >
> > **dst** – Found solution.

The method finds a unit-length solution x of a singular linear system A*x = 0. Depending on the rank of A, there can be no solutions, a single solution or an infinite number of solutions. In general, the algorithm solves the following problem:

$$dst = \arg \min_{x:\|x\|=1} \|src \cdot x\|$$

SVD::backSubst

Performs a singular value back substitution.

C++: void SVD::**backSubst**(InputArray **rhs**, OutputArray **dst**) const

C++: **static** void SVD::**backSubst**(InputArray **w**, InputArray **u**, InputArray **vt**, InputArray **rhs**, OutputArray **dst**)

Python: cv2.**SVBackSubst**(w, u, vt, rhs[, dst]) → dst

C: void **cvSVBkSb**(const CvArr* **w**, const CvArr* **u**, const CvArr* **v**, const CvArr* **rhs**, CvArr* **dst**, int **flags**)

Python: cv.**SVBkSb**(w, u, v, rhs, dst, flags) → None

> **Parameters**
>
> > **w** – Singular values
> >
> > **u** – Left singular vectors
> >
> > **v** – Right singular vectors
> >
> > **vt** – Transposed matrix of right singular vectors.
> >
> > **rhs** – Right-hand side of a linear system (u*w*v')*dst = rhs to be solved, where A has been previously decomposed.
> >
> > **dst** – Found solution of the system.

The method computes a back substitution for the specified right-hand side:

$$x = vt^T \cdot diag(w)^{-1} \cdot u^T \cdot rhs \sim A^{-1} \cdot rhs$$

Using this technique you can either get a very accurate solution of the convenient linear system, or the best (in the least-squares terms) pseudo-solution of an overdetermined linear system.

Note: Explicit SVD with the further back substitution only makes sense if you need to solve many linear systems with the same left-hand side (for example, `src`). If all you need is to solve a single system (possibly with multiple `rhs` immediately available), simply call `solve()` add pass `DECOMP_SVD` there. It does absolutely the same thing.

---

sum

Calculates the sum of array elements.

C++: Scalar **sum**(InputArray **arr**)

Python: `cv2.sumElems`(arr) → retval

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

Python: `cv.Sum`(arr) → CvScalar

> **Parameters**
>
>> **arr** – Source array that must have from 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

Returns the default random number generator.

C++: RNG& **theRNG**()

The function `theRNG` returns the default random number generator. For each thread, there is a 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 is much faster to use this function to retrieve the generator and then use `RNG::operator _Tp()` .

See Also:

`RNG`, `randu()`, `randn()`

trace

Returns the trace of a matrix.

C++: Scalar **trace**(InputArray **mat**)

Python: `cv2.trace`(mat) → retval

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

Python: cv.**Trace**(mat) → CvScalar

> **Parameters**
>
>> **mtx** – Source matrix.

The function trace returns the sum of the diagonal elements of the matrix mtx .

$$tr(mtx) = \sum_i mtx(i, i)$$

transform

Performs the matrix transformation of every array element.

C++: void **transform**(InputArray **src**, OutputArray **dst**, InputArray **mtx**)

Python: cv2.**transform**(src, mtx[, dst]) → dst

C: void **cvTransform**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mtx**, const CvMat* **shiftvec**=NULL)

Python: cv.**Transform**(src, dst, mtx, shiftvec=None) → None

> **Parameters**
>
>> **src** – Source array that must have as many channels (1 to 4) as mtx.cols or mtx.cols-1.
>>
>> **dst** – Destination array of the same size and depth as src . It has as many channels as mtx.rows .
>>
>> **mtx** – Transformation 2x2 or 2x3 floating-point matrix.
>>
>> **shiftvec** – Optional translation vector (when mtx is 2x2)

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

$$dst(I) = mtx \cdot src(I)$$

(when mtx.cols=src.channels()), or

$$dst(I) = mtx \cdot [src(I); 1]$$

(when mtx.cols=src.channels()+1)

Every element of the N -channel array src is interpreted as N -element vector that is transformed using the M x N or M x (N+1) matrix mtx to M-element vector - the corresponding element of the destination 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 to YUV transforms), shuffling the image channels, and so forth.

See Also:

perspectiveTransform(), getAffineTransform(), estimateRigidTransform(), warpAffine(), warpPerspective()

transpose

Transposes a matrix.

C++: void **transpose**(InputArray **src**, OutputArray **dst**)

Python: cv2.**transpose**(src[, dst]) → dst

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

Python: cv.**Transpose**(src, dst) → None

> **Parameters**
>
> > **src** – Source array.
> >
> > **dst** – Destination array of the same type as src .

The function transpose() transposes the matrix src :

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

Note: No complex conjugation is done in case of a complex matrix. It it should be done separately if needed.

2.5 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 an 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 channel ordering is normally *Blue, Green, Red*. This is what imshow(), imread(), and imwrite() expect. So, if you form a color using the 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 used color space. 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. This means that 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 $Point(x, y) \rightarrow Point2f(x * 2^{-shift}, y * 2^{-shift})$. This feature is especially effective when rendering antialiased shapes.

Note: The functions do not support alpha-transparency when the target image is 4-channel. In this case, 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

Draws a circle.

C++: void **circle**(Mat& **img**, Point **center**, int **radius**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

Python: cv2.**circle**(img, center, radius, color[, thickness[, lineType[, shift]]]) → None

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

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

---

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 the `line()` description.

shift – Number of fractional bits in the coordinates of the center and in the radius value.

The function `circle` draws a simple or filled circle with a given center and radius.

clipLine

Clips the line against the image rectangle.

C++: bool **clipLine**(Size **imgSize**, Point& **pt1**, Point& **pt2**)

C++: bool **clipLine**(Rect **imgRect**, Point& **pt1**, Point& **pt2**)

Python: cv2.**clipLine**(imgRect, pt1, pt2) → retval, pt1, pt2

C: int **cvClipLine**(CvSize **imgSize**, CvPoint* **pt1**, CvPoint* **pt2**)

Python: cv.**ClipLine**(imgSize, pt1, pt2) -> (clippedPt1, clippedPt2)

Parameters

imgSize – Image size. The image rectangle is Rect(0, 0, imgSize.width, imgSize.height).

imgRect – Image rectangle.

pt1 – First line point.

pt2 – Second line point.

The functions `clipLine` calculate a part of the line segment that is entirely within the specified rectangle. They return `false` if the line segment is completely outside the rectangle. Otherwise, they return `true` .

ellipse

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

C++: 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)

C++: void **ellipse**(Mat& **img**, const RotatedRect& **box**, const Scalar& **color**, int **thickness**=1, int **lineType**=8)

Python: cv2.**ellipse**(img, center, axes, angle, startAngle, endAngle, color[, thickness[, lineType[, shift]]]) → None

Python: cv2.**ellipse**(img, box, color[, thickness[, lineType]]) → None

C: void **cvEllipse**(CvArr* **img**, CvPoint **center**, CvSize **axes**, double **angle**, double **startAngle**, double **endAngle**, CvScalar **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

Python: cv.**Ellipse**(img, center, axes, angle, startAngle, endAngle, color, thickness=1, lineType=8, shift=0)
$\rightarrow$ None

C: void **cvEllipseBox**(CvArr* **img**, CvBox2D **box**, CvScalar **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

Python: cv.**EllipseBox**(img, box, color, thickness=1, lineType=8, shift=0) $\rightarrow$ None

> **Parameters**
>
>> **img** – Image.
>>
>> **center** – Center of the ellipse.
>>
>> **axes** – Length of the ellipse axes.
>>
>> **angle** – 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 `RotatedRect` or `CvBox2D`. This means that 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 the `line()` description.
>>
>> **shift** – Number of fractional bits in the coordinates of the center and values of axes.

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 figure below explains the meaning of the parameters.

Figure 1. Parameters of Elliptic Arc
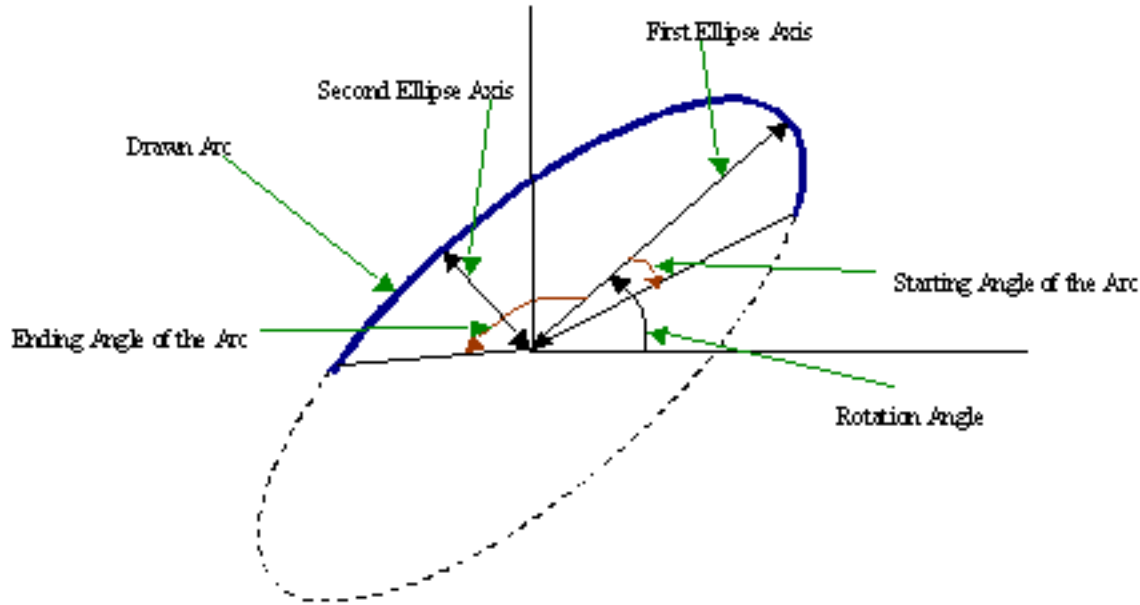
ellipse2Poly

Approximates an elliptic arc with a polyline.

C++: void **ellipse2Poly**(Point **center**, Size **axes**, int **angle**, int **startAngle**, int **endAngle**, int **delta**, vector<Point>& **pts**)

Python: cv2.**ellipse2Poly**(center, axes, angle, arcStart, arcEnd, delta) → pts

>   **Parameters**

>>   **center** – Center of the arc.

>>   **axes** – Half-sizes of the arc. See the `ellipse()` for details.

>>   **angle** – Rotation angle of the ellipse in degrees. See the `ellipse()` for details.

>>   **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** – 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

Fills a convex polygon.

C++: void **fillConvexPoly**(Mat& **img**, const Point* **pts**, int **npts**, const Scalar& **color**, int **lineType**=8, int **shift**=0)

Python: cv2.**fillConvexPoly**(img, points, color[, lineType[, shift]]) → None

C: void **cvFillConvexPoly**(CvArr* **img**, CvPoint* **pts**, int **npts**, CvScalar **color**, int **lineType**=8, int **shift**=0)

Python: cv.**FillConvexPoly**(img, pn, color, lineType=8, shift=0) → None

> **Parameters**
>
> > **img** – Image.
> >
> > **pts** – Polygon vertices.
> >
> > **npts** – Number of polygon vertices.
> >
> > **color** – Polygon color.
> >
> > **lineType** – Type of the polygon boundaries. See the line() description.
> >
> > **shift** – 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 . It can fill not only convex polygons but any monotonic polygon without self-intersections, that is, 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

Fills the area bounded by one or more polygons.

C++: void **fillPoly**(Mat& **img**, const Point** **pts**, const int* **npts**, int **ncontours**, const Scalar& **color**, int **lineType**=8, int **shift**=0, Point **offset**=Point())

Python: cv2.**fillPoly**(img, pts, color[, lineType[, shift[, offset]]]) → None

C: void **cvFillPoly**(CvArr* **img**, CvPoint** **pts**, int* **npts**, int **contours**, CvScalar **color**, int **lineType**=8, int **shift**=0)

Python: cv.**FillPoly**(img, polys, color, lineType=8, shift=0) → None

> **Parameters**
>
> > **img** – Image.
> >
> > **pts** – Array of polygons where each polygon is represented as an array of points.
> >
> > **npts** – Array of polygon vertex counters.
> >
> > **ncontours** – Number of contours that bind the filled region.
> >
> > **color** – Polygon color.
> >
> > **lineType** – Type of the polygon boundaries. See the line() description.
> >
> > **shift** – 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 thier parts), and so forth.

getTextSize

Calculates the width and height of a text string.

C++: Size **getTextSize**(const string& **text**, int **fontFace**, double **fontScale**, int **thickness**, int* **baseLine**)

Python: cv2.**getTextSize**(text, fontFace, fontScale, thickness) → retval, baseLine

C: void **cvGetTextSize**(const char* **textString**, const CvFont* **font**, CvSize* **textSize**, int* **baseline**)

Python: cv.**GetTextSize**(textString, font)-> (textSize, baseline)

> **Parameters**

>> **text** – Input text string.

>> **fontFace** – Font to use. See the `putText()` for details.

>> **fontScale** – Font scale. See the `putText()` for details.

>> **thickness** – Thickness of lines used to render the text. See `putText()` for details.

>> **baseLine** – Output parameter - y-coordinate of the baseline relative to the bottom-most text point.

The function `getTextSize` calculates and returns the size of a box that contains the specified text. That is, the following code renders some text, the tight box surrounding it, and the baseline:

```cpp
// 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);
```

InitFont

Initializes font structure (OpenCV 1.x API).

C: void **cvInitFont**(CvFont* **font**, int **fontFace**, double **hscale**, double **vscale**, double **shear**=0, int **thickness**=1, int **lineType**=8)

> **Parameters**

>> **font** – Pointer to the font structure initialized by the function

> **fontFace** – Font name identifier. Only a subset of Hershey fonts http://sources.isc.org/utils/misc/hershey-font.txt are supported now:
>
> – **CV_FONT_HERSHEY_SIMPLEX** normal size sans-serif font
>
> – **CV_FONT_HERSHEY_PLAIN** small size sans-serif font
>
> – **CV_FONT_HERSHEY_DUPLEX** normal size sans-serif font (more complex than `CV_FONT_HERSHEY_SIMPLEX`)
>
> – **CV_FONT_HERSHEY_COMPLEX** normal size serif font
>
> – **CV_FONT_HERSHEY_TRIPLEX** normal size serif font (more complex than `CV_FONT_HERSHEY_COMPLEX`)
>
> – **CV_FONT_HERSHEY_COMPLEX_SMALL** smaller version of `CV_FONT_HERSHEY_COMPLEX`
>
> – **CV_FONT_HERSHEY_SCRIPT_SIMPLEX** hand-writing style font
>
> – **CV_FONT_HERSHEY_SCRIPT_COMPLEX** more complex variant of `CV_FONT_HERSHEY_SCRIPT_SIMPLEX`
>
> The parameter can be composited from one of the values above and an optional `CV_FONT_ITALIC` flag, which indicates italic or oblique font.
>
> **hscale** – Horizontal scale. If equal to `1.0f` , the characters have the original width depending on the font type. If equal to `0.5f` , the characters are of half the original width.
>
> **vscale** – Vertical scale. If equal to `1.0f` , the characters have the original height depending on the font type. If equal to `0.5f` , the characters are of half the original height.
>
> **shear** – Approximate tangent of the character slope relative to the vertical line. A zero value means a non-italic font, `1.0f` means about a 45 degree slope, etc.
>
> **thickness** – Thickness of the text strokes
>
> **lineType** – Type of the strokes, see `line()` description

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

See Also:

`PutText()`

line

Draws a line segment connecting two points.

C++: void **line**(Mat& **img**, Point **pt1**, Point **pt2**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

Python: cv2.**line**(img, pt1, pt2, color[, thickness[, lineType[, shift]]]) → None

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

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

> **Parameters**
>
> > **img** – 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, you may use the macro `CV_RGB(r, g, b)` .

LineIterator

class `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 *();
    // move the iterator to the next pixel
    LineIterator& operator ++();
    LineIterator operator ++(int);

    // internal state of the iterator
    uchar* ptr;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;
};
```

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 an effect (for example, with XOR operation).

The number of pixels along the line is stored 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

Draws a simple, thick, or filled up-right rectangle.

C++: void **rectangle**(Mat& **img**, Point **pt1**, Point **pt2**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

C++: void **rectangle**(Mat& **img**, Rect **r**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

Python: cv2.**rectangle**(img, pt1, pt2, color[, thickness[, lineType[, shift]]]) → None

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

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

> **Parameters**
>
>> **img** – Image.
>>
>> **pt1** – Vertex of the rectangle.
>>
>> **pt2** – Vertex of the recangle opposite to `pt1` .
>>
>> **r** – Alternative specification of the drawn rectangle.
>>
>> **color** – Rectangle color or brightness (grayscale image).
>>
>> **thickness** – Thickness of lines that make up the rectangle. Negative values, like `CV_FILLED`, mean that the function has to draw a filled rectangle.
>>
>> **lineType** – Type of the line. See the `line()` description.
>>
>> **shift** – Number of fractional bits in the point coordinates.

The function `rectangle` draws a rectangle outline or a filled rectangle whose two opposite corners are `pt1` and `pt2`, or `r.tl()` and `r.br()-Point(1,1)`.

polylines

Draws several polygonal curves.

C++: 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)

Python: cv2.**polylines**(img, pts, isClosed, color[, thickness[, lineType[, shift]]]) → None

C: void **cvPolyLine**(CvArr* **img**, CvPoint** **pts**, int* **npts**, int **contours**, int **isClosed**, CvScalar **color**, int **thickness**=1, int **lineType**=8, int **shift**=0)

Python: cv.**PolyLine**(img, polys, isClosed, color, thickness=1, lineType=8, shift=0) → None

> **Parameters**
>
>> **img** – Image.
>>
>> **pts** – Array of polygonal curves.
>>
>> **npts** – Array of polygon vertex counters.
>>
>> **ncontours** – Number of curves.

isClosed – Flag indicating whether the drawn polylines are closed or not. If they are closed, the function draws a 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 the `line()` description.

shift – Number of fractional bits in the vertex coordinates.

The function `polylines` draws one or more polygonal curves.

putText

Draws a text string.

C++: void **putText**(Mat& **img**, const string& **text**, Point **org**, int **fontFace**, double **fontScale**, Scalar **color**, int **thickness**=1, int **lineType**=8, bool **bottomLeftOrigin**=false)

Python: cv2.**putText**(img, text, org, fontFace, fontScale, color[, thickness[, linetype[, bottomLeftOrigin]]]) → None

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

Python: cv.**PutText**(img, text, org, font, color) → None

> **Parameters**

> > **img** – Image.

> > **text** – Text string to be drawn.

> > **org** – Bottom-left corner of the text string in the image.

> > **font** – CvFont structure initialized using `InitFont()`.

> > **fontFace** – 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`, or `FONT_HERSHEY_SCRIPT_COMPLEX`, where each of the font ID's can be combined with `FONT_HERSHEY_ITALIC` to get the slanted letters.

> > **fontScale** – Font scale factor that is multiplied by the font-specific base size.

> > **color** – Text color.

> > **thickness** – Thickness of the lines used to draw a text.

> > **lineType** – Line type. See the `line` for details.

> > **bottomLeftOrigin** – When true, the image data origin is at the bottom-left corner. Otherwise, it is at the top-left corner.

The function `putText` renders the specified text string in the image. Symbols that cannot be rendered using the specified font are replaced by question marks. See `getTextSize()` for a text rendering code example.

2.6 XML/YAML Persistence

XML/YAML file storages. Writing to a file storage.

You can store and then restore various OpenCV data structures to/from XML (http://www.w3c.org/XML) or YAML (http://www.yaml.org) formats. Also, it is possible store and load arbitrarily complex data structures, which include OpenCV data structures, as well as primitive data types (integer and floating-point numbers and text strings) as their elements.

Use the following procedure to write something to XML or YAML:

1. Create new FileStorage and open it for writing. It can be done with a single call to FileStorage::FileStorage() constructor that takes a filename, or you can use the default constructor and then call FileStorage::open(). Format of the file (XML or YAML) is determined from the filename extension (".xml" and ".yml"/".yaml", respectively)

2. Write all the data you want using the streaming operator >>, just like in the case of STL streams.

3. Close the file using FileStorage::release(). FileStorage destructor also closes the file.

Here is an example:

```cpp
#include "opencv2/opencv.hpp"
#include <time.h>

using namespace cv;

int main(int, char** argv)
{
    FileStorage fs("test.yml", FileStorage::WRITE);

    fs << "frameCount" << 5;
    time_t rawtime; time(&rawtime);
    fs << "calibrationDate" << asctime(localtime(&rawtime));
    Mat cameraMatrix = (Mat_<double>(3,3) << 1000, 0, 320, 0, 1000, 240, 0, 0, 1);
    Mat distCoeffs = (Mat_<double>(5,1) << 0.1, 0.01, -0.001, 0, 0);
    fs << "cameraMatrix" << cameraMatrix << "distCoeffs" << distCoeffs;
    fs << "features" << "[";
    for( int i = 0; i < 3; i++ )
    {
        int x = rand() % 640;
        int y = rand() % 480;
        uchar lbp = rand() % 256;

        fs << "{:" << "x" << x << "y" << y << "lbp" << "[:";
        for( int j = 0; j < 8; j++ )
            fs << ((lbp >> j) & 1);
        fs << "]" << "}";
    }
    fs << "]";
    fs.release();
    return 0;
}
```

The sample above stores to XML and integer, text string (calibration date), 2 matrices, and a custom structure "feature", which includes feature coordinates and LBP (local binary pattern) value. Here is output of the sample:

```
%YAML:1.0
frameCount: 5
```

```
calibrationDate: "Fri Jun 17 14:09:29 2011\n"
cameraMatrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1000., 0., 320., 0., 1000., 240., 0., 0., 1. ]
distCoeffs: !!opencv-matrix
   rows: 5
   cols: 1
   dt: d
   data: [ 1.0000000000000001e-01, 1.0000000000000000e-02,
       -1.0000000000000000e-03, 0., 0. ]
features:
   - { x:167, y:49, lbp:[ 1, 0, 0, 1, 1, 0, 1, 1 ] }
   - { x:298, y:130, lbp:[ 0, 0, 0, 1, 0, 0, 1, 1 ] }
   - { x:344, y:158, lbp:[ 1, 1, 0, 0, 0, 0, 1, 0 ] }
```

As an exercise, you can replace ".yml" with ".xml" in the sample above and see, how the corresponding XML file will look like.

Several things can be noted by looking at the sample code and the output:

- The produced YAML (and XML) consists of heterogeneous collections that can be nested. There are 2 types of collections: named collections (mappings) and unnamed collections (sequences). In mappings each element has a name and is accessed by name. This is similar to structures and `std::map` in C/C++ and dictionaries in Python. In sequences elements do not have names, they are accessed by indices. This is similar to arrays and `std::vector` in C/C++ and lists, tuples in Python. "Heterogeneous" means that elements of each single collection can have different types.

    Top-level collection in YAML/XML is a mapping. Each matrix is stored as a mapping, and the matrix elements are stored as a sequence. Then, there is a sequence of features, where each feature is represented a mapping, and lbp value in a nested sequence.

- When you write to a mapping (a structure), you write element name followed by its value. When you write to a sequence, you simply write the elements one by one. OpenCV data structures (such as cv::Mat) are written in absolutely the same way as simple C data structures - using "**<<**" operator.

- To write a mapping, you first write the special string "**{**" to the storage, then write the elements as pairs (`fs << <element_name> << <element_value>`) and then write the closing "**}**".

- To write a sequence, you first write the special string "**[**", then write the elements, then write the closing "**]**".

- In YAML (but not XML), mappings and sequences can be written in a compact Python-like inline form. In the sample above matrix elements, as well as each feature, including its lbp value, is stored in such inline form. To store a mapping/sequence in a compact form, put ":" after the opening character, e.g. use "**{:**" instead of "**{**" and "**[:**" instead of "**[**". When the data is written to XML, those extra ":" are ignored.

Reading data from a file storage.

To read the previously written XML or YAML file, do the following:

1. Open the file storage using `FileStorage::FileStorage()` constructor or `FileStorage::open()` method. In the current implementation the whole file is parsed and the whole representation of file storage is built in memory as a hierarchy of file nodes (see `FileNode`)

2. Read the data you are interested in. Use `FileStorage::operator []()`, `FileNode::operator []()` and/or `FileNodeIterator`.

3. Close the storage using FileStorage::release().

Here is how to read the file created by the code sample above:

```
FileStorage fs2("test.yml", FileStorage::READ);

// first method: use (type) operator on FileNode.
int frameCount = (int)fs2["frameCount"];

std::string date;
// second method: use FileNode::operator >>
fs2["calibrationDate"] >> date;

Mat cameraMatrix2, distCoeffs2;
fs2["cameraMatrix"] >> cameraMatrix2;
fs2["distCoeffs"] >> distCoeffs2;

cout << "frameCount: " << frameCount << endl
     << "calibration date: " << date << endl
     << "camera matrix: " << cameraMatrix2 << endl
     << "distortion coeffs: " << distCoeffs2 << endl;

FileNode features = fs2["features"];
FileNodeIterator it = features.begin(), it_end = features.end();
int idx = 0;
std::vector<uchar> lbpval;

// iterate through a sequence using FileNodeIterator
for( ; it != it_end; ++it, idx++ )
{
    cout << "feature #" << idx << ": ";
    cout << "x=" << (int)(*it)["x"] << ", y=" << (int)(*it)["y"] << ", lbp: (";
    // you can also easily read numerical arrays using FileNode >> std::vector operator.
    (*it)["lbp"] >> lbpval;
    for( int i = 0; i < (int)lbpval.size(); i++ )
        cout << " " << (int)lbpval[i];
    cout << ")" << endl;
}
fs.release();
```

FileStorage

class FileStorage

XML/YAML file storage class that incapsulates all the information necessary for writing or reading data to/from file.

FileNode

class FileNode

The class FileNode represents each element of the file storage, be it a matrix, a matrix element or a top-level node, containing all the file content. That is, a file node may contain either a singe value (integer, floating-point value or a text string), or it can be a sequence of other file nodes, or it can be a mapping. Type of the file node can be determined using FileNode::type() method.

FileNodeIterator

class FileNodeIterator

The class `FileNodeIterator` is used to iterate through sequences and mappings. A standard STL notation, with `node.begin()`, `node.end()` denoting the beginning and the end of a sequence, stored in `node`. See the data reading sample in the beginning of the section.

2.7 XML/YAML Persistence (C API)

The section describes the OpenCV 1.x API for reading and writing data structures to/from XML or YAML files. It is now recommended to use the new C++ interface for reading and writing data.

CvFileStorage

struct CvFileStorage

The structure `CvFileStorage` is a "black box" representation of the file storage associated with a file on disk. Several functions that are described below take `CvFileStorage*` as inputs and allow the user to save or to load hierarchical collections that consist of scalar values, standard CXCore objects (such as matrices, sequences, graphs), and user-defined objects.

OpenCV can read and write data in XML (http://www.w3c.org/XML) or YAML (http://www.yaml.org) formats. Below is an example of 3x3 floating-point identity matrix A, stored in XML and YAML files using CXCore functions:

XML:

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

YAML:

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

As it can be seen from the examples, XML uses nested tags to represent hierarchy, while YAML uses indentation for that purpose (similar to the Python programming language).

The same functions can read and write data in both formats; the particular format is determined by the extension of the opened file, ".xml" for XML files and ".yml" or ".yaml" for YAML.

CvFileNode

struct CvFileNode

File storage node. When XML/YAML file is read, it is first parsed and stored in the memory as a hierarchical collection of nodes. Each node can be a "leaf", that is, contain a single number or a string, or be a collection of other nodes. Collections are also referenced to as "structures" in the data writing functions. There can be named collections (mappings), where each element has a name and is accessed by a name, and ordered collections (sequences), where elements do not have names, but rather accessed by index.

int **tag**
     type of the file node:

          • CV_NODE_NONE - empty node

          • CV_NODE_INT - an integer

          • CV_NODE_REAL - a floating-point number

          • CV_NODE_STR - text string

          • CV_NODE_SEQ - a sequence

          • CV_NODE_MAP - a mapping

     type of the node can be retrieved using `CV_NODE_TYPE(node->tag)` macro.

CvTypeInfo* **info**
     optional pointer to the user type information. If you look at the matrix representation in XML and YAML, shown above, you may notice `type_id="opencv-matrix"` or `!!opencv-matrix` strings. They are used to specify that the certain element of a file is a representation of a data structure of certain type ("opencv-matrix" corresponds to `CvMat`). When a file is parsed, such type identifiers are passed to `FindType()` to find type information and the pointer to it is stored in the file node. See `CvTypeInfo` for more details.

union **data**
     the node data, declared as:

```
union
{
    double f; /* scalar floating-point number */
    int i;    /* scalar integer number */
    CvString str; /* text string */
    CvSeq* seq; /* sequence (ordered collection of file nodes) */
    struct CvMap* map; /* map (collection of named file nodes) */
} data;
```

     Primitive nodes are read using `ReadInt()`, `ReadReal()` and `ReadString()`. Sequences are read by iterating through `node->data.seq` (see "Dynamic Data Structures" section). Mappings are read using `GetFileNodeByName()`. Nodes with the specified type (so that `node->info != NULL`) can be read using `Read()`.

CvAttrList

struct **CvAttrList**

List of attributes.

```
typedef struct CvAttrList
{
    const char** attr; /* NULL-terminated array of (attribute_name,attribute_value) pairs */
    struct CvAttrList* next; /* pointer to next chunk of the attributes list */
}
CvAttrList;
```

```
/* initializes CvAttrList structure */
inline CvAttrList cvAttrList( const char** attr=NULL, CvAttrList* next=NULL );

/* returns attribute value or 0 (NULL) if there is no such attribute */
const char* cvAttrValue( const CvAttrList* attr, const char* attr_name );
```

In the current implementation, attributes are used to pass extra parameters when writing user objects (see `Write()`). XML attributes inside tags are not supported, aside from the object type specification (`type_id` attribute).

CvTypeInfo

struct CvTypeInfo

Type information.

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

typedef struct CvTypeInfo
{
    int flags; /* not used */
    int header_size; /* sizeof(CvTypeInfo) */
    struct CvTypeInfo* prev; /* previous registered type in the list */
    struct CvTypeInfo* next; /* next registered type in the list */
    const char* type_name; /* type name, written to file storage */

    /* methods */
    CvIsInstanceFunc is_instance; /* checks if the passed object belongs to the type */
    CvReleaseFunc release; /* releases object (memory etc.) */
    CvReadFunc read; /* reads object from file storage */
    CvWriteFunc write; /* writes object to file storage */
    CvCloneFunc clone; /* creates a copy of the object */
}
CvTypeInfo;
```

The structure contains information about one of the standard or user-defined types. Instances of the type may or may not contain a pointer to the corresponding `CvTypeInfo` structure. In any case, there is a way to find the type info structure for a given object using the `TypeOf()` function. Aternatively, type info can be found by type name using `FindType()`, which is used when an object is read from file storage. The user can register a new type with `RegisterType()` that adds the type information structure into the beginning of the type list. Thus, it is possible to create specialized types from generic standard types and override the basic methods.

Clone

Makes a clone of an object.

C: void* **cvClone**(const void* **structPtr**)

> **Parameters**
>
> > **structPtr** – The object to clone

The function finds the type of a given object and calls `clone` with the passed object. Of course, if you know the object type, for example, `structPtr` is `CvMat*`, it is faster to call the specific function, like `CloneMat()`.

EndWriteStruct

Finishes writing to a file node collection.

C: void **cvEndWriteStruct**(CvFileStorage* **fs**)

>>**Parameters**

>>>>**fs** – File storage

See Also:

`StartWriteStruct()`.

FindType

Finds a type by its name.

C: CvTypeInfo* **cvFindType**(const char* **typeName**)

>>**Parameters**

>>>>**typeName** – Type name

The function finds a registered type by its name. It returns NULL if there is no type with the specified name.

FirstType

Returns the beginning of a type list.

C: CvTypeInfo* **cvFirstType**(void **None**)

The function returns the first type in the list of registered types. Navigation through the list can be done via the `prev` and `next` fields of the `CvTypeInfo` structure.

GetFileNode

Finds a node in a map or file storage.

C: CvFileNode* **cvGetFileNode**(CvFileStorage* **fs**, CvFileNode* **map**, const CvStringHashNode* **key**, int **createMissing**=0)

>**Parameters**

>>>**fs** – File storage

>>>**map** – The parent map. If it is NULL, the function searches a top-level node. If both `map` and `key` are NULLs, the function returns the root file node - a map that contains top-level nodes.

>>>**key** – Unique pointer to the node name, retrieved with `GetHashedKey()`

>>>**createMissing** – Flag that specifies whether an absent node should be added to the map

The function finds a file node. It is a faster version of `GetFileNodeByName()` (see `GetHashedKey()` discussion). Also, the function can insert a new node, if it is not in the map yet.

GetFileNodeByName

Finds a node in a map or file storage.

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

>> **Parameters**

>>> **fs** – File storage

>>> **map** – The parent map. If it is NULL, the function searches in all the top-level nodes (streams), starting with the first one.

>>> **name** – The file node name

The function finds a file node by name. The node is searched either in map or, if the pointer is NULL, among the top-level file storage nodes. Using this function for maps and GetSeqElem() (or sequence reader) for sequences, it is possible to nagivate through the file storage. To speed up multiple queries for a certain key (e.g., in the case of an array of structures) one may use a combination of GetHashedKey() and GetFileNode().

GetFileNodeName

Returns the name of a file node.

C: const char* **cvGetFileNodeName**(const CvFileNode* **node**)

>> **Parameters**

>>> **node** – File node

The function returns the name of a file node or NULL, if the file node does not have a name or if node is NULL.

GetHashedKey

Returns a unique pointer for a given name.

C: CvStringHashNode* **cvGetHashedKey**(CvFileStorage* **fs**, const char* **name**, int **len**=-1, int **createMissing**=0)

>> **Parameters**

>>> **fs** – File storage

>>> **name** – Literal node name

>>> **len** – Length of the name (if it is known apriori), or -1 if it needs to be calculated

>>> **createMissing** – Flag that specifies, whether an absent key should be added into the hash table

The function returns a unique pointer for each particular file node name. This pointer can be then passed to the GetFileNode() function that is faster than GetFileNodeByName() because it compares text strings by comparing pointers rather than the strings' content.

Consider the following example where an array of points is encoded as a sequence of 2-entry maps:

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

Then, it is possible to get hashed "x" and "y" pointers to speed up decoding of the points.

```c
#include "cxcore.h"

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

    if( CV_NODE_IS_SEQ(points->tag) )
    {
        CvSeq* seq = points->data.seq;
        int i, total = seq->total;
        CvSeqReader reader;
        cvStartReadSeq( seq, &reader, 0 );
        for( i = 0; i < total; i++ )
        {
            CvFileNode* pt = (CvFileNode*)reader.ptr;
#if 1 /* faster variant */
            CvFileNode* xnode = cvGetFileNode( fs, pt, x_key, 0 );
            CvFileNode* ynode = cvGetFileNode( fs, pt, y_key, 0 );
            assert( xnode && CV_NODE_IS_INT(xnode->tag) &&
                    ynode && CV_NODE_IS_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#elif 1 /* slower variant; does not use x_key & y_key */
            CvFileNode* xnode = cvGetFileNodeByName( fs, pt, "x" );
            CvFileNode* ynode = cvGetFileNodeByName( fs, pt, "y" );
            assert( xnode && CV_NODE_IS_INT(xnode->tag) &&
                    ynode && CV_NODE_IS_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#else /* the slowest yet the easiest to use variant */
            int x = cvReadIntByName( fs, pt, "x", 0 /* default value */ );
            int y = cvReadIntByName( fs, pt, "y", 0 /* default value */ );
#endif
            CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
            printf("
        }
    }
    cvReleaseFileStorage( &fs );
    return 0;
}
```

Please note that whatever method of accessing a map you are using, it is still much slower than using plain sequences; for example, in the above example, it is more efficient to encode the points as pairs of integers in a single numeric sequence.

GetRootFileNode

Retrieves one of the top-level nodes of the file storage.

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

> **Parameters**
>
> > **fs** – File storage

> **stream_index** – Zero-based index of the stream. See `StartNextStream()` . In most cases, there is only one stream in the file; however, there can be several.

The function returns one of the top-level file nodes. The top-level nodes do not have a name, they correspond to the streams that are stored one after another in the file storage. If the index is out of range, the function returns a NULL pointer, so all the top-level nodes can be iterated by subsequent calls to the function with `stream_index=0,1,...`, until the NULL pointer is returned. This function can be used as a base for recursive traversal of the file storage.

Load

Loads an object from a file.

C: void* **cvLoad**(const char* **filename**, CvMemStorage* **storage**=NULL, const char* **name**=NULL, const char** **realName**=NULL)

Python: cv.**Load**(filename, storage=None, name=None) → generic

> **Parameters**
>
> > **filename** – File name
> >
> > **storage** – Memory storage for dynamic structures, such as `CvSeq` or `CvGraph` . It is not used for matrices or images.
> >
> > **name** – Optional object name. If it is NULL, the first top-level object in the storage will be loaded.
> >
> > **realName** – Optional output parameter that will contain the name of the loaded object (useful if `name=NULL`)

The function loads an object from a file. It basically reads the specified file, find the first top-level node and calls `Read()` for that node. If the file node does not have type information or the type information can not be found by the type name, the function returns NULL. After the object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as a sequence, contour, or graph, one should pass a valid memory storage destination to the function.

OpenFileStorage

Opens file storage for reading or writing data.

C: CvFileStorage* **cvOpenFileStorage**(const char* **filename**, CvMemStorage* **memstorage**, int **flags**)

> **Parameters**
>
> > **filename** – Name of the file associated with the storage
> >
> > **memstorage** – Memory storage used for temporary data and for storing dynamic structures, such as `CvSeq` or `CvGraph` . If it is NULL, a temporary memory storage is created and used.
> >
> > **flags** – Can be one of the following:
> >
> > > – **CV_STORAGE_READ** the storage is open for reading
> > >
> > > – **CV_STORAGE_WRITE** the storage is open for writing

The function opens file storage for reading or writing data. In the latter case, a new file is created or an existing file is rewritten. The type of the read or written file is determined by the filename extension: `.xml` for XML and `.yml` or `.yaml` for YAML. The function returns a pointer to the `CvFileStorage` structure.

Read

Decodes an object and returns a pointer to it.

C: void* **cvRead**(CvFileStorage* **fs**, CvFileNode* **node**, CvAttrList* **attributes**=NULL)

> **Parameters**
>
> > **fs** – File storage
> >
> > **node** – The root object node
> >
> > **attributes** – Unused parameter

The function decodes a user object (creates an object in a native representation from the file storage subtree) and returns it. The object to be decoded must be an instance of a registered type that supports the read method (see CvTypeInfo). The type of the object is determined by the type name that is encoded in the file. If the object is a dynamic structure, it is created either in memory storage and passed to OpenFileStorage() or, if a NULL pointer was passed, in temporary memory storage, which is released when ReleaseFileStorage() is called. Otherwise, if the object is not a dynamic structure, it is created in a heap and should be released with a specialized function or by using the generic Release().

ReadByName

Finds an object by name and decodes it.

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

> **Parameters**
>
> > **fs** – File storage
> >
> > **map** – The parent map. If it is NULL, the function searches a top-level node.
> >
> > **name** – The node name
> >
> > **attributes** – Unused parameter

The function is a simple superposition of GetFileNodeByName() and Read().

ReadInt

Retrieves an integer value from a file node.

C: int **cvReadInt**(const CvFileNode* **node**, int **defaultValue**=0)

> **Parameters**
>
> > **node** – File node
> >
> > **defaultValue** – The value that is returned if node is NULL

The function returns an integer that is represented by the file node. If the file node is NULL, the defaultValue is returned (thus, it is convenient to call the function right after GetFileNode() without checking for a NULL pointer). If the file node has type CV_NODE_INT, then node->data.i is returned. If the file node has type CV_NODE_REAL, then node->data.f is converted to an integer and returned. Otherwise the error is reported.

ReadIntByName

Finds a file node and returns its value.

C: int **cvReadIntByName**(const CvFileStorage* **fs**, const CvFileNode* **map**, const char* **name**, int **default-Value**=0)

>   **Parameters**

>>   **fs** – File storage

>>   **map** – The parent map. If it is NULL, the function searches a top-level node.

>>   **name** – The node name

>>   **defaultValue** – The value that is returned if the file node is not found

The function is a simple superposition of `GetFileNodeByName()` and `ReadInt()`.

ReadRawData

Reads multiple numbers.

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

>   **Parameters**

>>   **fs** – File storage

>>   **src** – The file node (a sequence) to read numbers from

>>   **dst** – Pointer to the destination array

>>   **dt** – Specification of each array element. It has the same format as in `WriteRawData()`.

The function reads elements from a file node that represents a sequence of scalars.

ReadRawDataSlice

Initializes file node sequence reader.

C: void **cvReadRawDataSlice**(const CvFileStorage* **fs**, CvSeqReader* **reader**, int **count**, void* **dst**, const char* **dt**)

>   **Parameters**

>>   **fs** – File storage

>>   **reader** – The sequence reader. Initialize it with `StartReadRawData()`.

>>   **count** – The number of elements to read

>>   **dst** – Pointer to the destination array

>>   **dt** – Specification of each array element. It has the same format as in `WriteRawData()`.

The function reads one or more elements from the file node, representing a sequence, to a user-specified array. The total number of read sequence elements is a product of `total` and the number of components in each array element. For example, if `dt=2if`, the function will read `total*3` sequence elements. As with any sequence, some parts of the file node sequence can be skipped or read repeatedly by repositioning the reader using `SetSeqReaderPos()`.

ReadReal

Retrieves a floating-point value from a file node.

C: double **cvReadReal**(const CvFileNode* **node**, double **defaultValue**=0.)

> **Parameters**
>
> > **node** – File node
> >
> > **defaultValue** – The value that is returned if `node` is NULL

The function returns a floating-point value that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after `GetFileNode()` without checking for a NULL pointer). If the file node has type `CV_NODE_REAL` , then `node->data.f` is returned. If the file node has type `CV_NODE_INT` , then `node-:math:'>'data.f` is converted to floating-point and returned. Otherwise the result is not determined.

ReadRealByName

Finds a file node and returns its value.

C: double **cvReadRealByName**(const CvFileStorage* **fs**, const CvFileNode* **map**, const char* **name**, double
defaultValue=0.)

> **Parameters**
>
> > **fs** – File storage
> >
> > **map** – The parent map. If it is NULL, the function searches a top-level node.
> >
> > **name** – The node name
> >
> > **defaultValue** – The value that is returned if the file node is not found

The function is a simple superposition of `GetFileNodeByName()` and `ReadReal()` .

ReadString

Retrieves a text string from a file node.

C: const char* **cvReadString**(const CvFileNode* **node**, const char* **defaultValue**=NULL)

> **Parameters**
>
> > **node** – File node
> >
> > **defaultValue** – The value that is returned if `node` is NULL

The function returns a text string that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after `GetFileNode()` without checking for a NULL pointer). If the file node has type `CV_NODE_STR` , then `node-:math:'>'data.str.ptr` is returned. Otherwise the result is not determined.

ReadStringByName

Finds a file node by its name and returns its value.

C: const char* **cvReadStringByName**(const CvFileStorage* **fs**, const CvFileNode* **map**, const char* **name**,
const char* **defaultValue**=NULL)

> **Parameters**
>
> > **fs** – File storage
> >
> > **map** – The parent map. If it is NULL, the function searches a top-level node.
> >
> > **name** – The node name
> >
> > **defaultValue** – The value that is returned if the file node is not found

The function is a simple superposition of `GetFileNodeByName()` and `ReadString()` .

RegisterType

Registers a new type.

C: void **cvRegisterType**(const CvTypeInfo* **info**)

> **Parameters**
>
> > **info** – Type info structure

The function registers a new type, which is described by `info` . The function creates a copy of the structure, so the user should delete it after calling the function.

Release

Releases an object.

C: void **cvRelease**(void** **structPtr**)

> **Parameters**
>
> > **structPtr** – Double pointer to the object

The function finds the type of a given object and calls `release` with the double pointer.

ReleaseFileStorage

Releases file storage.

C: void **cvReleaseFileStorage**(CvFileStorage** **fs**)

> **Parameters**
>
> > **fs** – Double pointer to the released file storage

The function closes the file associated with the storage and releases all the temporary structures. It must be called after all I/O operations with the storage are finished.

Save

Saves an object to a file.

C: void **cvSave**(const char* **filename**, const void* **structPtr**, const char* **name**=NULL, const char* **comment**=NULL, CvAttrList **attributes**=cvAttrList())

Python: cv.**Save**(filename, structPtr, name=None, comment=None) → None

> **Parameters**

> **filename** – File name
>
> **structPtr** – Object to save
>
> **name** – Optional object name. If it is NULL, the name will be formed from `filename` .
>
> **comment** – Optional comment to put in the beginning of the file
>
> **attributes** – Optional attributes passed to `Write()`

The function saves an object to a file. It provides a simple interface to `Write()` .

StartNextStream

Starts the next stream.

C: void **cvStartNextStream**(CvFileStorage* **fs**)

> **Parameters**
>
> > **fs** – File storage

The function finishes the currently written stream and starts the next stream. In the case of XML the file with multiple streams looks like this:

```
<opencv_storage>
<!-- stream #1 data -->
</opencv_storage>
<opencv_storage>
<!-- stream #2 data -->
</opencv_storage>
...
```

The a YAML file will look like this:

```
%YAML:1.0
# stream #1 data
...
---
# stream #2 data
```

This is useful for concatenating files or for resuming the writing process.

StartReadRawData

Initializes the file node sequence reader.

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

> **Parameters**
>
> > **fs** – File storage
> >
> > **src** – The file node (a sequence) to read numbers from
> >
> > **reader** – Pointer to the sequence reader

The function initializes the sequence reader to read data from a file node. The initialized reader can be then passed to `ReadRawDataSlice()`.

StartWriteStruct

Starts writing a new structure.

C: void **cvStartWriteStruct**(CvFileStorage* **fs**, const char* **name**, int **struct_flags**, const char* **type-Name**=NULL, CvAttrList **attributes**=cvAttrList())

> **Parameters**
>
>> **fs** – File storage
>>
>> **name** – Name of the written structure. The structure can be accessed by this name when the storage is read.
>>
>> **struct_flags** – A combination one of the following values:
>>
>> – **CV_NODE_SEQ** the written structure is a sequence (see discussion of `CvFileStorage`), that is, its elements do not have a name.
>>
>> – **CV_NODE_MAP** the written structure is a map (see discussion of `CvFileStorage`), that is, all its elements have names.
>>
>> One and only one of the two above flags must be specified
>>
>> **CV_NODE_FLOW** – the optional flag that makes sense only for YAML streams. It means that the structure is written as a flow (not as a block), which is more compact. It is recommended to use this flag for structures or arrays whose elements are all scalars.
>>
>> **typeName** – Optional parameter - the object type name. In case of XML it is written as a `type_id` attribute of the structure opening tag. In the case of YAML it is written after a colon following the structure name (see the example in `CvFileStorage` description). Mainly it is used with user objects. When the storage is read, the encoded type name is used to determine the object type (see `CvTypeInfo` and `FindType()`).
>>
>> **attributes** – This parameter is not used in the current implementation

The function starts writing a compound structure (collection) that can be a sequence or a map. After all the structure fields, which can be scalars or structures, are written, `EndWriteStruct()` should be called. The function can be used to group some objects or to implement the `write` function for a some user object (see `CvTypeInfo`).

TypeOf

Returns the type of an object.

C: CvTypeInfo* **cvTypeOf**(const void* **structPtr**)

> **Parameters**
>
>> **structPtr** – The object pointer

The function finds the type of a given object. It iterates through the list of registered types and calls the `is_instance` function/method for every type info structure with that object until one of them returns non-zero or until the whole list has been traversed. In the latter case, the function returns NULL.

UnregisterType

Unregisters the type.

C: void **cvUnregisterType**(const char* **typeName**)

> **Parameters**

> **typeName** – Name of an unregistered type

The function unregisters a type with a specified name. If the name is unknown, it is possible to locate the type info by an instance of the type using `TypeOf()` or by iterating the type list, starting from `FirstType()`, and then calling `cvUnregisterType(info->typeName)`.

Write

Writes an object to file storage.

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

> **Parameters**
>
>> **fs** – File storage
>>
>> **name** – Name of the written object. Should be NULL if and only if the parent structure is a sequence.
>>
>> **ptr** – Pointer to the object
>>
>> **attributes** – The attributes of the object. They are specific for each particular type (see the dicsussion below).

The function writes an object to file storage. First, the appropriate type info is found using `TypeOf()`. Then, the `write` method associated with the type info is called.

Attributes are used to customize the writing procedure. The standard types support the following attributes (all the `dt` attributes have the same format as in `WriteRawData()`):

1. CvSeq

   • **header_dt** description of user fields of the sequence header that follow CvSeq, or CvChain (if the sequence is a Freeman chain) or CvContour (if the sequence is a contour or point sequence)

   • **dt** description of the sequence elements.

   • **recursive** if the attribute is present and is not equal to "0" or "false", the whole tree of sequences (contours) is stored.

2. CvGraph

   • **header_dt** description of user fields of the graph header that follows CvGraph;

   • **vertex_dt** description of user fields of graph vertices

   • **edge_dt** description of user fields of graph edges (note that the edge weight is always written, so there is no need to specify it explicitly)

Below is the code that creates the YAML file shown in the `CvFileStorage` description:

```
#include "cxcore.h"

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

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

    cvReleaseFileStorage( &fs );
    cvReleaseMat( &mat );
```

```
    return 0;
}
```

WriteComment

Writes a comment.

C: void **cvWriteComment**(CvFileStorage* **fs**, const char* **comment**, int **eolComment**)

> **Parameters**
>
> > **fs** – File storage
> >
> > **comment** – The written comment, single-line or multi-line
> >
> > **eolComment** – If non-zero, the function tries to put the comment at the end of current line. If the flag is zero, if the comment is multi-line, or if it does not fit at the end of the current line, the comment starts a new line.

The function writes a comment into file storage. The comments are skipped when the storage is read.

WriteFileNode

Writes a file node to another file storage.

C: void **cvWriteFileNode**(CvFileStorage* **fs**, const char* **new_node_name**, const CvFileNode* **node**, int **embed**)

> **Parameters**
>
> > **fs** – Destination file storage
> >
> > **new_file_node** – New name of the file node in the destination file storage. To keep the existing name, use `cvGetFileNodeName()`
> >
> > **node** – The written node
> >
> > **embed** – If the written node is a collection and this parameter is not zero, no extra level of hiararchy is created. Instead, all the elements of node are written into the currently written structure. Of course, map elements can only be embedded into another map, and sequence elements can only be embedded into another sequence.

The function writes a copy of a file node to file storage. Possible applications of the function are merging several file storages into one and conversion between XML and YAML formats.

WriteInt

Writes an integer value.

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

> **Parameters**
>
> > **fs** – File storage
> >
> > **name** – Name of the written value. Should be NULL if and only if the parent structure is a sequence.
> >
> > **value** – The written value

The function writes a single integer value (with or without a name) to the file storage.

WriteRawData

Writes multiple numbers.

C: void **cvWriteRawData**(CvFileStorage* **fs**, const void* **src**, int **len**, const char* **dt**)

> **Parameters**
>
> > **fs** – File storage
> >
> > **src** – Pointer to the written array
> >
> > **len** – Number of the array elements to write
> >
> > **dt** – Specification of each array element that has the following format (`[count]{'u'|'c'|'w'|'s'|'i'|'f'|'d'})...` where the characters correspond to fundamental C types:
> >
> > – **u** 8-bit unsigned number
> >
> > – **c** 8-bit signed number
> >
> > – **w** 16-bit unsigned number
> >
> > – **s** 16-bit signed number
> >
> > – **i** 32-bit signed number
> >
> > – **f** single precision floating-point number
> >
> > – **d** double precision floating-point number
> >
> > – **r pointer, 32 lower bits of which are written as a signed integer. The type can be used to store structures w** example, `2if` means that each array element is a structure of 2 integers, followed by a single-precision floating-point number. The equivalent notations of the above specification are ' `iif` ', ' `2i1f` ' and so forth. Other examples: u means that the array consists of bytes, and `2d` means the array consists of pairs of doubles.

The function writes an array, whose elements consist of single or multiple numbers. The function call can be replaced with a loop containing a few `WriteInt()` and `WriteReal()` calls, but a single call is more efficient. Note that because none of the elements have a name, they should be written to a sequence rather than a map.

WriteReal

Writes a floating-point value.

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

> **Parameters**
>
> > **fs** – File storage
> >
> > **name** – Name of the written value. Should be NULL if and only if the parent structure is a sequence.
> >
> > **value** – The written value

The function writes a single floating-point value (with or without a name) to file storage. Special values are encoded as follows: NaN (Not A Number) as .NaN, infinity as +.Inf or -.Inf.

The following example shows how to use the low-level writing functions to store custom structures, such as termination criteria, without registering a new type.

```
void write_termcriteria( CvFileStorage* fs, const char* struct_name,
                         CvTermCriteria* termcrit )
{
    cvStartWriteStruct( fs, struct_name, CV_NODE_MAP, NULL, cvAttrList(0,0));
    cvWriteComment( fs, "termination criteria", 1 ); // just a description
    if( termcrit->type & CV_TERMCRIT_ITER )
        cvWriteInteger( fs, "max_iterations", termcrit->max_iter );
    if( termcrit->type & CV_TERMCRIT_EPS )
        cvWriteReal( fs, "accuracy", termcrit->epsilon );
    cvEndWriteStruct( fs );
}
```

WriteString

Writes a text string.

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

> **Parameters**
>
> > **fs** – File storage
> >
> > **name** – Name of the written string . Should be NULL if and only if the parent structure is a sequence.
> >
> > **str** – The written text string
> >
> > **quote** – If non-zero, the written string is put in quotes, regardless of whether they are required. Otherwise, if the flag is zero, quotes are used only when they are required (e.g. when the string starts with a digit or contains spaces).

The function writes a text string to file storage.

2.8 Clustering

kmeans

Finds centers of clusters and groups input samples around the clusters.

C++: double **kmeans**(InputArray **samples**, int **clusterCount**, InputOutputArray **labels**, TermCriteria **criteria**, int **attempts**, int **flags**, OutputArray **centers**=noArray())

Python: cv2.**kmeans**(data, K, criteria, attempts, flags[, bestLabels[, centers]]) → retval, bestLabels, centers

C: int **cvKMeans2**(const CvArr* **samples**, int **nclusters**, CvArr* **labels**, CvTermCriteria **criteria**, int **attempts**=1, CvRNG* **rng**=0, int **flags**=0, CvArr* **centers**=0, double* **compactness**=0)

Python: cv.**KMeans2**(samples, nclusters, labels, criteria) → None

> **Parameters**
>
> > **samples** – Floating-point matrix of input samples, one row per sample.
> >
> > **clusterCount** – Number of clusters to split the set by.
> >
> > **labels** – Input/output integer array that stores the cluster indices for every sample.

criteria – The algorithm termination criteria, that is, the maximum number of iterations and/or the desired accuracy. The accuracy is specified as `criteria.epsilon`. As soon as each of the cluster centers moves by less than `criteria.epsilon` on some iteration, the algorithm stops.

attempts – Flag to specify the number of 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 – Flag that can take the following values:

– **KMEANS_RANDOM_CENTERS** Select random initial centers in each attempt.

– **KMEANS_PP_CENTERS** Use `kmeans++` center initialization by Arthur and Vassilvit-skii [Arthur2007].

– **KMEANS_USE_INITIAL_LABELS** During the first (and possibly the only) attempt, use the user-supplied labels instead of computing them from the initial centers. For the second and further attempts, use the random or semi-random centers. Use one of `KMEANS_*_CENTERS` flag to specify the exact method.

centers – 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. As an output, $labels_i$ contains a 0-based cluster index for the sample stored in the ith row of the `samples` matrix.

The function returns the compactness measure that is computed as

$$\sum_i \|\texttt{samples}_i - \texttt{centers}_{\texttt{labels}_i}\|^2$$

after every attempt. The best (minimum) value is chosen and the corresponding labels and the compactness value are returned by the function. Basically, you can use only the core of the function, set the number of attempts to 1, initialize labels each time using a custom algorithm, pass them with the (`flags = KMEANS_USE_INITIAL_LABELS`) flag, and then choose the best (most-compact) clustering.

partition

Splits an element set into equivalency classes.

C++: template<typename _Tp, class _EqPredicate> int **partition**(const vector<_Tp>& **vec**, vec-tor<int>& **labels**, _EqPredicate **predicate**=_EqPredicate())

> **Parameters**

> **vec** – Set of elements stored as a vector.

> **labels** – Output vector of labels. It contains as many elements as `vec`. Each label `labels[i]` is a 0-based cluster index of `vec[i]` .

> **predicate** – Equivalence predicate (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 in the same class, and returns `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 . The function returns the number of equivalency classes.

2.9 Utility and System Functions and Macros

alignPtr

Aligns a pointer to the specified number of bytes.

C++: template<typename _Tp> _Tp* **alignPtr**(_Tp* **ptr**, int **n**=sizeof(_Tp))

> **Parameters**
>
> > **ptr** – Aligned pointer.
> >
> > **n** – Alignment size that 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

Aligns a buffer size to the specified number of bytes.

C++: size_t **alignSize**(size_t **sz**, int **n**)

> **Parameters**
>
> > **sz** – Buffer size to align.
> >
> > **n** – Alignment size that must be a power of two.

The function returns the minimum number that is greater or equal to `sz` and is divisble by `n` :

$$(sz + n-1) \ \& \ -n$$

allocate

Allocates an array of elements.

C++: template<typename _Tp> _Tp* **allocate**(size_t **n**)

> **Parameters**
>
> > **n** – Number of elements to allocate.

The generic function `allocate` allocates a buffer for the specified number of elements. For each element, the default constructor is called.

deallocate

Deallocates an array of elements.

C++: template<typename _Tp> void **deallocate**(_Tp* **ptr**, size_t **n**)

> **Parameters**
>
> > **ptr** – Pointer to the deallocated buffer.
> >
> > **n** – 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()` .

fastAtan2

Calculates the angle of a 2D vector in degrees.

C++: float **fastAtan2**(float **y**, float **x**)

Python: cv2.**fastAtan2**(y, x) → retval

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

Python: cv.**FastArctan**(y, x) → float

> **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 to 360 degrees. The accuracy is about 0.3 degrees.

cubeRoot

Computes the cube root of an argument.

C++: float **cubeRoot**(float **val**)

Python: cv2.**cubeRoot**(val) → retval

C: float **cvCbrt**(float **val**)

Python: cv.**Cbrt**(val) → float

> **Parameters**
>
> > **val** – A function argument.

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

Ceil

Rounds floating-point number to the nearest integer not smaller than the original.

C: int **cvCeil**(double **value**)

Python: cv.**Ceil**(value) → int

> **Parameters**
>
> > **value** – floating-point number. If the value is outside of INT_MIN ... INT_MAX range, the result is not defined.

The function computes an integer i such that:

$$i - 1 < \mathtt{value} \le i$$

Floor

Rounds floating-point number to the nearest integer not larger than the original.

C: int **cvFloor**(double **value**)

Python: cv.**Floor**(value) → int

> **Parameters**
>
> > **value** – floating-point number. If the value is outside of INT_MIN ... INT_MAX range, the result is not defined.

The function computes an integer i such that:

$$i \leq \text{value} < i + 1$$

Round

Rounds floating-point number to the nearest integer

C: int **cvRound**(double **value**)

Python: cv.**Round**(value) → int

> **Parameters**
>
> > **value** – floating-point number. If the value is outside of INT_MIN ... INT_MAX range, the result is not defined.

IsInf

Determines if the argument is Infinity.

C: int **cvIsInf**(double **value**)

Python: cv.**IsInf**(value) → int

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

The function returns 1 if the argument is a plus or minus infinity (as defined by IEEE754 standard) and 0 otherwise.

IsNaN

Determines if the argument is Not A Number.

C: int **cvIsNaN**(double **value**)

Python: cv.**IsNaN**(value) → int

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

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

CV_Assert

Checks a condition at runtime and throws exception if it fails

C++: `CV_Assert`(expr **None**)

The macros `CV_Assert` (and `CV_DbgAssert`) evaluate the specified expression. 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

Signals an error and raises an exception.

C++: void **error**(const Exception& **exc**)

C: int **cvError**(int **status**, const char* **funcName**, const char* **errMsg**, const char* **filename**, int **line**)

> **Parameters**
>
> > **exc** – Exception to throw.
> >
> > **code** – Error code. Normally, it is 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 parentheses.

The function and the helper macros `CV_Error` and `CV_Error_`:

```
#define CV_Error( code, msg ) error(...)
#define CV_Error_( code, args ) 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 the Debug configuration, it then provokes memory access violation, so that the execution stack and all the parameters can be analyzed by the debugger. In the Release configuration, the exception `exc` is thrown.

The macro `CV_Error_` can be used to construct an error message on-fly to include some dynamic information, for example:

```
// note the extra parentheses around the formatted text message
CV_Error_(CV_StsOutOfRange,
    ("the matrix element (
    i, j, mtx.at<float>(i,j)))
```

Exception

class `Exception`

Exception class passed to an 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 _line);
```

```
Exception(const Exception& exc);
Exception& operator = (const Exception& exc);

// the error code
int code;
// the error text message
string err;
// function name where the error happened
string func;
// the source file name where the error happened
string file;
// the source file line where the error happened
int line;
};
```

The class `Exception` encapsulates all or almost all 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

Allocates an aligned memory buffer.

C++: void* **fastMalloc**(size_t **size**)

C: void* **cvAlloc**(size_t **size**)

> **Parameters**
>
> > **size** – Allocated buffer size.

The function allocates the buffer of the specified size and returns it. When the buffer size is 16 bytes or more, the returned buffer is aligned to 16 bytes.

fastFree

Deallocates a memory buffer.

C++: void **fastFree**(void* **ptr**)

C: void **cvFree**(void** **pptr**)

> **Parameters**
>
> > **ptr** – Pointer to the allocated buffer.
> >
> > **pptr** – Double pointer to the allocated buffer

The function deallocates the buffer allocated with `fastMalloc()` . If NULL pointer is passed, the function does nothing. C version of the function clears the pointer *pptr to avoid problems with double memory deallocation.

format

Returns a text string formatted using the `printf`-like expression.

string format(const char* fmt, ...)

> **Parameters**
>
> > **fmt** – `printf` -compatible formatting specifiers.

The function acts like `sprintf` but forms and returns an STL string. It can be used to form an error message in the `Exception()` constructor.

checkHardwareSupport

Returns true if the specified feature is supported by the host hardware.

C++: bool **checkHardwareSupport**(int **feature**)

C: int **cvCheckHardwareSupport**(int **feature**)

Python: **checkHardwareSupport**(feature) $\rightarrow$ Bool

> **Parameters**
>
> > **feature** – The feature of interest, one of:
> >
> > – `CV_CPU_MMX` - MMX
> >
> > – `CV_CPU_SSE` - SSE
> >
> > – `CV_CPU_SSE2` - SSE 2
> >
> > – `CV_CPU_SSE3` - SSE 3
> >
> > – `CV_CPU_SSSE3` - SSSE 3
> >
> > – `CV_CPU_SSE4_1` - SSE 4.1
> >
> > – `CV_CPU_SSE4_2` - SSE 4.2
> >
> > – `CV_CPU_POPCNT` - POPCOUNT
> >
> > – `CV_CPU_AVX` - AVX

The function returns true if the host hardware supports the specified feature. When user calls `setUseOptimized(false)`, the subsequent calls to `checkHardwareSupport()` will return false until `setUseOptimized(true)` is called. This way user can dynamically switch on and off the optimized code in OpenCV.

getNumThreads

Returns the number of threads used by OpenCV.

C++: int **getNumThreads**()

The function returns the number of threads that is used by OpenCV.

See Also:

`setNumThreads()`, `getThreadNum()`

getThreadNum

Returns the index of the currently executed thread.

C++: int **getThreadNum**()

The function returns a 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

Returns the number of ticks.

C++: int64 **getTickCount**()

Python: cv2.**getTickCount**() → retval

The function returns the number of ticks after the certain event (for example, 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

Returns the number of ticks per second.

C++: double **getTickFrequency**()

Python: cv2.**getTickFrequency**() → retval

The function returns the number of ticks per second. That is, the following code computes the execution time in seconds:

```
double t = (double)getTickCount();
// do something ...
t = ((double)getTickCount() - t)/getTickFrequency();
```

getCPUTickCount

Returns the number of CPU ticks.

C++: int64 **getCPUTickCount**()

Python: cv2.**getCPUTickCount**() → retval

The function returns the current number of CPU ticks on some architectures (such as x86, x64, PowerPC). On other platforms the function is equivalent to getTickCount. It can also be used for very accurate time measurements, as well as for RNG initialization. Note that in case of multi-CPU systems a thread, from which getCPUTickCount is called, can be suspended and resumed at another CPU with its own counter. So, theoretically (and practically) the subsequent calls to the function do not necessary return the monotonously increasing values. Also, since a modern CPU varies the CPU frequency depending on the load, the number of CPU clocks spent in some code cannot be directly converted to time units. Therefore, getTickCount is generally a preferable solution for measuring execution time.

saturate_cast

Template function for accurate conversion from one primitive type to another.

C++: template<...> _Tp **saturate_cast**(_Tp2 **v**)

> **Parameters**

> > **v** – Function parameter.

The functions saturate_cast resemble the standard C++ cast operations, such as static_cast<T>() and others. They perform an efficient and accurate conversion from one primitive type to another (see the introduction chapter). saturate in the name means that when the input value v is out of the range of the target type, the result is not formed just by taking low bits of the input, but instead the value is clipped. For example:

```
uchar a = saturate_cast<uchar>(-100); // a = 0 (UCHAR_MIN)
short b = saturate_cast<short>(33333.33333); // b = 32767 (SHRT_MAX)
```

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 a 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 the simplest or most complex image processing functions in OpenCV.

See Also:

`add()`, `subtract()`, `multiply()`, `divide()`, `Mat::convertTo()`

setNumThreads

Sets the number of threads used by OpenCV.

C++: void **setNumThreads**(int **nthreads**)

> **Parameters**
>
>> **nthreads** – 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 uses the default number of threads that is usually equal to the number of the processing cores.

See Also:

`getNumThreads()`, `getThreadNum()`

setUseOptimized

Enables or disables the optimized code.

C++: void **setUseOptimized**(bool **onoff**)

Python: cv2.**setUseOptimized**(onoff) → None

C: int **cvUseOptimized**(int **onoff**)

> **Parameters**
>
>> **onoff** – The boolean flag specifying whether the optimized code should be used (onoff=true) or not (onoff=false).

The function can be used to dynamically turn on and off optimized code (code that uses SSE2, AVX, and other instructions on the platforms that support it). It sets a global flag that is further checked by OpenCV functions. Since the flag is not checked in the inner OpenCV loops, it is only safe to call the function on the very top level in your application where you can be sure that no other OpenCV function is currently executed.

By default, the optimized code is enabled unless you disable it in CMake. The current status can be retrieved using `useOptimized`.

useOptimized

Returns the status of optimized code usage.

C++: bool **useOptimized**()

Python: cv2.**useOptimized**() → retval

The function returns `true` if the optimized code is enabled. Otherwise, it returns `false`.

IMGPROC. IMAGE PROCESSING

3.1 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). It means that for each pixel location (x, y) in the source image (normally, rectangular), its 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 values, and so on. The computed response is stored in 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 support 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 you want to smooth an image using a Gaussian 3×3 filter, then, when processing the left-most pixels in each row, you need pixels to the left of them, that is, outside of the image. You can let these pixels be the same as the left-most image pixels ("replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("constant border" extrapolation method), and so on. OpenCV enables you to specify the extrapolation method. For details, see the function borderInterpolate() and discussion of the borderType parameter in various functions below.

BaseColumnFilter

class BaseColumnFilter

Base class for filters with single-column kernels.

```
class BaseColumnFilter
{
public:
    virtual ~BaseColumnFilter();

    // To be overriden by the user.
    //
    // runs a 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 dststep,
                            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 a base class for filtering data using single-column kernels. Filtering does not have to be a linear operation. In general, it could be written as follows:

$$\mathtt{dst}(x,y) = F(\mathtt{src[y](x)}, \; \mathtt{src[y+1](x)}, \; ..., \; \mathtt{src[y+ksize-1](x)})$$

where F is a filtering function but, as it is represented as a class, it can produce any side effects, memorize previously processed data, and so on. The class only defines an 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 the `FilterEngine` constructor. While the filtering operation interface uses the `uchar` type, a particular implementation is not limited to 8-bit data.

See Also:

`BaseRowFilter`, `BaseFilter`, `FilterEngine`, `getColumnSumFilter()`, `getLinearColumnFilter()`, `getMorphologyColumnFilter()`

BaseFilter

class `BaseFilter`

Base class for 2D image filters.

```
class BaseFilter
{
public:
    virtual ~BaseFilter();

    // To be overriden by the user.
    //
    // runs a 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" elements
    // each output row has "width*cn" elements.
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, int dststep,
                            int dstcount, int width, int cn) = 0;
    // resets the filter state (may be needed for IIR filters)
    virtual void reset();
    Size ksize;
    Point anchor;
};
```

The class `BaseFilter` is a base class for filtering data using 2D kernels. Filtering does not have to be a linear operation. In general, it could be written as follows:

$$\begin{aligned}
\mathtt{dst}(x,y) = F(&\mathtt{src[y](x)}, \; \mathtt{src[y](x+1)}, \; ..., \; \mathtt{src[y](x+ksize.width-1)}, \\
&\mathtt{src[y+1](x)}, \; \mathtt{src[y+1](x+1)}, \; ..., \; \mathtt{src[y+1](x+ksize.width-1)}, \\
&... \\
&\mathtt{src[y+ksize.height-1](x)}, \\
&\mathtt{src[y+ksize.height-1](x+1)}, \\
&...\mathtt{src[y+ksize.height-1](x+ksize.width-1)})
\end{aligned}$$

where F is a filtering function. The class only defines an 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 the `FilterEngine` constructor. While the filtering operation interface uses the `uchar` type, a particular implementation is not limited to 8-bit data.

See Also:

`BaseColumnFilter`, `BaseRowFilter`, `FilterEngine`, `getLinearFilter()`, `getMorphologyFilter()`

BaseRowFilter

class `BaseRowFilter`

Base class for filters with single-row kernels.

```cpp
class BaseRowFilter
{
public:
    virtual ~BaseRowFilter();

    // To be overriden 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 a base class for filtering data using single-row kernels. Filtering does not have to be a linear operation. In general, it could be written as follows:

$$dst(x, y) = F(src[y](x), \ src[y](x + 1), \ ..., \ src[y](x + ksize.width - 1))$$

where F is a filtering function. The class only defines an 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 the `FilterEngine` constructor. While the filtering operation interface uses the `uchar` type, a particular implementation is not limited to 8-bit data.

See Also:

`BaseColumnFilter`, `BaseFilter`, `FilterEngine`, `getLinearRowFilter()`, `getMorphologyRowFilter()`, `getRowSumFilter()`

FilterEngine

class `FilterEngine`

Generic image filtering class.

```cpp
class FilterEngine
{
public:
    // empty constructor
    FilterEngine();
    // builds a 2D non-separable filter (!_filter2D.empty()) or
    // a separable filter (!_rowFilter.empty() && !_columnFilter.empty())
```

```
    // the input data type will be "srcType", the output data type will be "dstType",
    // the intermediate data type is "bufType".
    // _rowBorderType and _columnBorderType determine how the image
    // will be extrapolated beyond the image boundaries.
    // _borderValue is only used when _rowBorderType and/or _columnBorderType
    // == 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 _rowBorderType by default
                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 _columnBorderType=-1,
            const Scalar& _borderValue=Scalar());
    // starts filtering of the ROI in an image of size "wholeSize".
    // returns the starting y-position in the source image.
    virtual int start(Size wholeSize, Rect roi, int maxBufRows=-1);
    // alternative form of start that takes the image
    // itself instead of "wholeSize". Set isolated to true to pretend that
    // there are no real pixels outside of the ROI
    // (so that the pixels are extrapolated using the specified border modes)
    virtual int start(const Mat& src, const Rect& srcRoi=Rect(0,0,-1,-1),
                    bool isolated=false, int maxBufRows=-1);
    // processes the next portion of the source image,
    // "srcCount" rows starting from "src" and
    // stores the results in "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, computes extrapolated values of the "virtual" pixels outside of the image, and so on. 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()`, and others. Thus, the class plays a key role in many of OpenCV filtering functions.

This class makes it easier to combine filtering operations with other operations, such as color space conversions, thresholding, arithmetic operations, and others. By combining several operations together you can get much better performance because your data will stay in cache. For example, see below the implementation of the Laplace operator for 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 consider using the same kernels, just swapped
    Mat kd, ks;
    getSobelKernels( kd, ks, 2, 0, ksize, false, ktype );

    // 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, Scalar() );
    Ptr<FilterEngine> Fyy = createSeparableLinearFilter(src.type(),
        dst.type(), ks, kd, Point(-1,-1), 0, borderType, borderType, Scalar() );

    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-1)
    // 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 always pass DELTA rows to the filter
    // (note that the "proceed" method takes care of possibe overflow, since
    // it was given the actual image height in the "start" method)
    // on output you can get:
    //  * < DELTA rows (initial buffer accumulation stage)
    //  * = DELTA rows (settled state in the middle)
    //  * > DELTA rows (when the input image is over, generate
    //                  "virtual" rows using the border mode and filter them)
    // 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 process
    for( ; dsty < dst.rows; sptr += DELTA*src.step, dsty += dy )
    {
        Fxx->proceed( sptr, (int)src.step, DELTA, Ixx.data, (int)Ixx.step );
        dy = Fyy->proceed( sptr, (int)src.step, DELTA, d2y.data, (int)Iyy.step );
        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. The method is implemented as follows:

```cpp
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 dst.
    // and FilterEngine::start will check if the source ROI is inside src.
    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 number
    // of the source rows to process
    // (including the possible rows outside of srcRoi but inside the source image)
    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. This will be a filter response at that particular pixel. However, it is 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 new matrix headers:

```cpp
// 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 =
```

Explore the data types. As it was mentioned in the 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 is 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

Applies the bilateral filter to an image.

C++: void **bilateralFilter**(InputArray **src**, OutputArray **dst**, int **d**, double **sigmaColor**, double **sigmaSpace**, int **borderType**=BORDER_DEFAULT)

Python: cv2.**bilateralFilter**(src, d, sigmaColor, sigmaSpace[, dst[, borderType]]) → dst

> **Parameters**
>
> > **src** – Source 8-bit or floating-point, 1-channel or 3-channel image.
> >
> > **dst** – Destination image of the same size and type as src .
> >
> > **d** – Diameter of each pixel neighborhood that is used during filtering. If it is non-positive, it is computed from sigmaSpace .
> >
> > **sigmaColor** – Filter sigma in the color space. A 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. A larger value of the parameter means that farther pixels will influence each other as long as their colors are close enough (see sigmaColor). When 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 bilateralFilter can reduce unwanted noise very well while keeping edges fairly sharp. However, it is very slow compared to most filters.

Sigma values: For simplicity, you can set the 2 sigma values to be the same. If they are small (< 10), the filter will not have much effect, whereas if they are large (> 150), they will have a very strong effect, making the image look "cartoonish".

Filter size: Large filters (d > 5) are very slow, so it is recommended to use d=5 for real-time applications, and perhaps d=9 for offline applications that need heavy noise filtering.

This filter does not work inplace.

blur

Smoothes an image using the normalized box filter.

C++: void **blur**(InputArray **src**, OutputArray **dst**, Size **ksize**, Point **anchor**=Point(-1,-1), int **border-Type**=BORDER_DEFAULT)

Python: cv2.**blur**(src, ksize[, dst[, anchor[, borderType]]]) → dst

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image of the same size and type as `src` .
> >
> > **ksize** – Smoothing kernel size.
> >
> > **anchor** – Anchor point. The default value `Point(-1,-1)` means that the anchor is at the kernel center.
> >
> > **borderType** – Border mode used to extrapolate pixels outside of the image.

The function smoothes an image using the kernel:

$$K = \frac{1}{\texttt{ksize.width*ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ & & \cdots\cdots\cdots & & \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

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

Computes the source location of an extrapolated pixel.

C++: int **borderInterpolate**(int **p**, int **len**, int **borderType**)

Python: cv2.**borderInterpolate**(p, len, borderType) → retval

> **Parameters**
>
> > **p** – 0-based coordinate of the extrapolated pixel along one of the axes, likely <0 or >= `len` .
> >
> > **len** – Length of the array along the corresponding axis.
> >
> > **borderType** – 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 a donor pixel corresponding to the specified extrapolated pixel when using the specified extrapolation border mode. For example, if you 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 looks like:

```
float val = img.at<float>(borderInterpolate(100, img.rows, BORDER_REFLECT_101),
                          borderInterpolate(-5, img.cols, BORDER_WRAP));
```

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

Smoothes an image using the box filter.

C++: void **boxFilter**(InputArray **src**, OutputArray **dst**, int **ddepth**, Size **ksize**, Point **anchor**=Point(-1,-1), bool **normalize**=true, int **borderType**=BORDER_DEFAULT)

Python: cv2.**boxFilter**(src, ddepth, ksize[, dst[, anchor[, normalize[, borderType]]]]) → dst

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image of the same size and type as `src` .
> >
> > **ksize** – Smoothing kernel size.
> >
> > **anchor** – Anchor point. The default value `Point(-1,-1)` means that the anchor is at the kernel center.
> >
> > **normalize** – Flag specifying whether the kernel is normalized by its area or not.
> >
> > **borderType** – Border mode used to extrapolate pixels outside of the image.

The function smoothes an image using the kernel:

$$K = \alpha \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ & & \cdots\cdots\cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

where

$$\alpha = \begin{cases} \frac{1}{\text{ksize.width*ksize.height}} & \text{when normalize=true} \\ 1 & \text{otherwise} \end{cases}$$

Unnormalized box filter is useful for computing various integral characteristics over each pixel neighborhood, such as covariance matrices of image derivatives (used in dense optical flow algorithms, and so on). If you need to compute pixel sums over variable-size windows, use `integral()` .

See Also:

`boxFilter()`, `bilateralFilter()`, `GaussianBlur()`, `medianBlur()`, `integral()`

buildPyramid

Constructs the Gaussian pyramid for an image.

C++: void **buildPyramid**(InputArray **src**, OutputArrayOfArrays **dst**, int **maxlevel**)

> **Parameters**

> **src** – Source image. Check `pyrDown()` for the list of supported types.
>
> **dst** – 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` , and so on.
>
> **maxlevel** – 0-based index of the last (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

Forms a border around an image.

C++: void **copyMakeBorder**(InputArray **src**, OutputArray **dst**, int **top**, int **bottom**, int **left**, int **right**, int **borderType**, const Scalar& **value**=Scalar())

Python: cv2.**copyMakeBorder**(src, top, bottom, left, right, borderType[, dst[, value]]) → dst

C: void **cvCopyMakeBorder**(const CvArr* **src**, CvArr* **dst**, CvPoint **offset**, int **bordertype**, CvScalar **value**=cvScalarAll(0))

Python: cv.**CopyMakeBorder**(src, dst, offset, bordertype, value=(0, 0, 0, 0)) → None

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image of the same type as `src` and the size `Size(src.cols+left+right, src.rows+top+bottom)` .
> >
> > **top** –
> >
> > **bottom** –
> >
> > **left** –
> >
> > **right** – Parameter specifying how many pixels in each direction from the source image rectangle to extrapolate. For example, `top=1, bottom=1, left=1, right=1` mean that 1 pixel-wide border needs to be built.
> >
> > **borderType** – Border type. See `borderInterpolate()` for details.
> >
> > **value** – 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 filtering functions based on it 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, for example:

```
// let border be the same in all directions
int border=2;
// constructs a larger image to fit both the image and the border
Mat gray_buf(rgb.rows + border*2, rgb.cols + border*2, rgb.depth());
// select the middle part of it w/o copying data
Mat gray(gray_canvas, Rect(border, border, rgb.cols, rgb.rows));
// 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

Returns a box filter engine.

C++: Ptr<FilterEngine> **createBoxFilter**(int **srcType**, int **dstType**, Size **ksize**, Point **anchor**=Point(-1,-1), bool **normalize**=true, int **borderType**=BORDER_DEFAULT)

C++: Ptr<BaseRowFilter> **getRowSumFilter**(int **srcType**, int **sumType**, int **ksize**, int **anchor**=-1)

C++: Ptr<BaseColumnFilter> **getColumnSumFilter**(int **sumType**, int **dstType**, int **ksize**, int **anchor**=-1, double **scale**=1)

> **Parameters**
>
>> **srcType** – Source image type.
>>
>> **sumType** – Intermediate horizontal sum type that must have as many channels as `srcType`.
>>
>> **dstType** – Destination image type that must have as many channels as `srcType`.
>>
>> **ksize** – Aperture size.
>>
>> **anchor** – Anchor position with the kernel. Negative values mean that the anchor is at the kernel center.
>>
>> **normalize** – Flag specifying whether the sums are normalized or not. See boxFilter() for details.
>>
>> **scale** – Another way to specify normalization in lower-level `getColumnSumFilter`.
>>
>> **borderType** – Border type to use. See borderInterpolate() .

The function is a convenience function that retrieves the 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

Returns an engine for computing image derivatives.

C++: Ptr<FilterEngine> **createDerivFilter**(int **srcType**, int **dstType**, int **dx**, int **dy**, int **ksize**, int **border-Type**=BORDER_DEFAULT)

> **Parameters**
>
>> **srcType** – Source image type.

> **dstType** – Destination image type that must have as many channels as `srcType` .
>
> **dx** – Derivative order in respect of x.
>
> **dy** – Derivative order in respect of y.
>
> **ksize** – Aperture size See `getDerivKernels()` .
>
> **borderType** – 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

Returns an engine for smoothing images with the Gaussian filter.

C++: Ptr<FilterEngine> **createGaussianFilter**(int **type**, Size **ksize**, double **sigmaX**, double **sigmaY**=0, int **borderType**=BORDER_DEFAULT)

> **Parameters**
>
> > **type** – Source and destination image type.
> >
> > **ksize** – Aperture size. See `getGaussianKernel()` .
> >
> > **sigmaX** – Gaussian sigma in the horizontal direction. See `getGaussianKernel()` .
> >
> > **sigmaY** – Gaussian sigma in the vertical direction. If 0, then `sigmaY` ← `sigmaX` .
> >
> > **borderType** – Border type to use. See `borderInterpolate()` .

The function `createGaussianFilter()` computes Gaussian kernel coefficients and then returns a 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 this limitation by calling `getGaussianKernel()` and then `createSeparableLinearFilter()` directly.

See Also:

`createSeparableLinearFilter()`, `getGaussianKernel()`, `GaussianBlur()`

createLinearFilter

Creates a non-separable linear filter engine.

C++: Ptr<FilterEngine> **createLinearFilter**(int **srcType**, int **dstType**, InputArray **kernel**, Point **_anchor**=Point(-1,-1), double **delta**=0, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1, const Scalar& **borderValue**=Scalar())

C++: Ptr<BaseFilter> **getLinearFilter**(int **srcType**, int **dstType**, InputArray **kernel**, Point **anchor**=Point(-1,-1), double **delta**=0, int **bits**=0)

> **Parameters**
>
> > **srcType** – Source image type.
> >
> > **dstType** – Destination image type that must have as many channels as `srcType` .

> > **kernel** – 2D array of filter coefficients.
>
> > **anchor** – Anchor point within the kernel. Special value `Point(-1,-1)` means that the anchor is at the kernel center.
>
> > **delta** – Value added to the filtered results before storing them.
>
> > **bits** – Number of the fractional bits. the parameter is used when the kernel is an integer matrix representing fixed-point filter coefficients.
>
> > **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.
>
> > **columnBorderType** – Pixel extrapolation method in the horizontal direction.
>
> > **borderValue** – Border vaule used in case of a constant border.

The function returns a pointer to a 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 the `FilterEngine` constructor.

See Also:

`createSeparableLinearFilter()`, `FilterEngine`, `filter2D()`

createMorphologyFilter

Creates an engine for non-separable morphological operations.

C++: Ptr<FilterEngine> **createMorphologyFilter**(int **op**, int **type**, InputArray **element**, Point **anchor**=Point(-1,-1), int **rowBorderType**=BORDER_CONSTANT, int **columnBorderType**=-1, const Scalar& **borderValue**=morphologyDefaultBorderValue())

C++: Ptr<BaseFilter> **getMorphologyFilter**(int **op**, int **type**, InputArray **element**, Point **anchor**=Point(-1,-1))

C++: Ptr<BaseRowFilter> **getMorphologyRowFilter**(int **op**, int **type**, int **esize**, int **anchor**=-1)

C++: Ptr<BaseColumnFilter> **getMorphologyColumnFilter**(int **op**, int **type**, int **esize**, int **anchor**=-1)

C++: Scalar **morphologyDefaultBorderValue**()

> > **Parameters**
>
> > > **op** – Morphology operation ID, `MORPH_ERODE` or `MORPH_DILATE` .
> >
> > > **type** – Input/output image type.
> >
> > > **element** – 2D 8-bit structuring element for a morphological operation. Non-zero elements indicate the pixels that belong to the element.
> >
> > > **esize** – Horizontal or vertical structuring element size for separable morphological operations.
> >
> > > **anchor** – Anchor position within the structuring element. Negative values mean that the anchor is at the kernel center.
> >
> > > **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.
> >
> > > **columnBorderType** – Pixel extrapolation method in the horizontal direction.

> **borderValue** – Border value in case of a constant border. The default value, `morphologyDefaultBorderValue` , has a special meaning. It is transformed $+\inf$ for the erosion and to $-\inf$ 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 is enough to use `createMorphologyFilter()` or even higher-level `erode()`, `dilate()` , or `morphologyEx()` . Note that `createMorphologyFilter()` analyzes the structuring element shape and builds a separable morphological filter engine when the structuring element is square.

See Also:

`erode()`, `dilate()`, `morphologyEx()`, `FilterEngine`

createSeparableLinearFilter

Creates an engine for a separable linear filter.

C++: Ptr<FilterEngine> **createSeparableLinearFilter**(int **srcType**, int **dstType**, InputArray **rowKernel**, InputArray **columnKernel**, Point **anchor**=Point(-1,-1), double **delta**=0, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1, const Scalar& **borderValue**=Scalar())

C++: Ptr<BaseColumnFilter> **getLinearColumnFilter**(int **bufType**, int **dstType**, InputArray **columnKernel**, int **anchor**, int **symmetryType**, double **delta**=0, int **bits**=0)

C++: Ptr<BaseRowFilter> **getLinearRowFilter**(int **srcType**, int **bufType**, InputArray **rowKernel**, int **anchor**, int **symmetryType**)

> Parameters

> > **srcType** – Source array type.

> > **dstType** – Destination image type that must have as many channels as `srcType` .

> > **bufType** – Intermediate buffer type that must have as many channels as `srcType` .

> > **rowKernel** – Coefficients for filtering each row.

> > **columnKernel** – Coefficients for filtering each column.

> > **anchor** – Anchor position within the kernel. Negative values mean that anchor is positioned at the aperture center.

> > **delta** – Value added to the filtered results before storing them.

> > **bits** – Number of the fractional bits. The parameter is used when the kernel is an integer matrix representing fixed-point filter coefficients.

> > **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.

> > **columnBorderType** – Pixel extrapolation method in the horizontal direction.

> > **borderValue** – Border value used in case of a constant border.

> > **symmetryType** – Type of each row and column kernel. See `getKernelType()` .

The functions construct primitive separable linear filtering operations or a filter engine based on them. Normally it is 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 filtering can be done in integer arithmetics, the number of `bits` to encode the filter coefficients. If it does not work for you, it is possible to call `getLinearColumnFilter`,''`getLinearRowFilter`'' directly and then pass them to the `FilterEngine` constructor.

See Also:

`sepFilter2D()`, `createLinearFilter()`, `FilterEngine`, `getKernelType()`

dilate

Dilates an image by using a specific structuring element.

C++: void **dilate**(InputArray **src**, OutputArray **dst**, InputArray **element**, Point **anchor**=Point(-1,-1), int **iterations**=1, int **borderType**=BORDER_CONSTANT, const Scalar& **borderValue**=morphologyDefaultBorderValue())

Python: cv2.**dilate**(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]) → dst

C: void **cvDilate**(const CvArr* **src**, CvArr* **dst**, IplConvKernel* **element**=NULL, int **iterations**=1)

Python: cv.**Dilate**(src, dst, element=None, iterations=1) → None

> **Parameters**
>
>> **src** – Source image.
>>
>> **dst** – Destination image of the same size and type as `src` .
>>
>> **element** – Structuring element used for dilation. If `element=Mat()` , a `3 x 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** – Number of times dilation is applied.
>>
>> **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.
>>
>> **borderValue** – Border value in case of a constant border. The default value has a special meaning. See `createMorphologyFilter()` for details.

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:

$$\mathtt{dst}(x,y) = \max_{(x',y'): \, \mathtt{element}(x',y') \neq 0} \mathtt{src}(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. In case of multi-channel images, each channel is processed independently.

See Also:

`erode()`, `morphologyEx()`, `createMorphologyFilter()`

erode

Erodes an image by using a specific structuring element.

C++: void **erode**(InputArray **src**, OutputArray **dst**, InputArray **element**, Point **anchor**=Point(-1,-1), int **iterations**=1, int **borderType**=BORDER_CONSTANT, const Scalar& **borderValue**=morphologyDefaultBorderValue())

Python: cv2.**erode**(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]) $\rightarrow$ dst

C: void **cvErode**(const CvArr* **src**, CvArr* **dst**, IplConvKernel* **element**=NULL, int **iterations**=1)

Python: cv.**Erode**(src, dst, element=None, iterations=1) $\rightarrow$ None

> **Parameters**
>
>> **src** – Source image.
>>
>> **dst** – Destination image of the same size and type as src .
>>
>> **element** – Structuring element used for erosion. If element=Mat() , a 3 x 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** – Number of times erosion is applied.
>>
>> **borderType** – Pixel extrapolation method. See borderInterpolate() for details.
>>
>> **borderValue** – Border value in case of a constant border. The default value has a special meaning. See createMorphologyFilter() for details.

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:

$$\mathtt{dst}(x,y) = \min_{(x',y'): \, \mathtt{element}(x',y') \neq 0} \mathtt{src}(x+x',y+y')$$

The function supports the in-place mode. Erosion can be applied several (iterations) times. In case of multi-channel images, each channel is processed independently.

See Also:

dilate(), morphologyEx(), createMorphologyFilter()

filter2D

Convolves an image with the kernel.

C++: void **filter2D**(InputArray **src**, OutputArray **dst**, int **ddepth**, InputArray **kernel**, Point **anchor**=Point(-1,-1), double **delta**=0, int **borderType**=BORDER_DEFAULT)

Python: cv2.**filter2D**(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]]) $\rightarrow$ dst

C: void **cvFilter2D**(const CvArr* **src**, CvArr* **dst**, const CvMat* **kernel**, CvPoint **anchor**=cvPoint(-1, -1))

Python: cv.**Filter2D**(src, dst, kernel, anchor=(-1, -1)) $\rightarrow$ None

> **Parameters**
>
>> **src** – Source image.
>>
>> **dst** – Destination image of the same size and the same number of channels as src .
>>
>> **ddepth** – 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** – 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** – Optional value added to the filtered pixels before storing them in `dst`.
>
> **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.

The function applies an arbitrary linear filter to an 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 compute correlation, not the convolution:

$$\mathtt{dst}(x,y) = \sum_{\substack{0 \le x' < \mathtt{kernel.cols}, \\ 0 \le y' < \mathtt{kernel.rows}}} \mathtt{kernel}(x',y') * \mathtt{src}(x + x' - \mathtt{anchor.x}, y + y' - \mathtt{anchor.y})$$

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 `(kernel.cols - anchor.x - 1, kernel.rows - anchor.y - 1)`.

The function uses the DFT-based algorithm in case of sufficiently large kernels (~``11 x 11`` or larger) and the direct algorithm (that uses the engine retrieved by `createLinearFilter()`) for small kernels.

See Also:

`sepFilter2D()`, `createLinearFilter()`, `dft()`, `matchTemplate()`

GaussianBlur

Smoothes an image using a Gaussian filter.

C++: void **GaussianBlur**(InputArray **src**, OutputArray **dst**, Size **ksize**, double **sigmaX**, double **sigmaY**=0, int **borderType**=BORDER_DEFAULT)

Python: cv2.**GaussianBlur**(src, ksize, sigma1[, dst[, sigma2[, borderType]]]) → dst

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image of the same size and type as `src`.
> >
> > **ksize** – Gaussian kernel size. `ksize.width` and `ksize.height` can differ but they both must be positive and odd. Or, they can be zero's and then they are computed from `sigma*`.
> >
> > **sigmaX** – Gaussian kernel standard deviation in X direction.
> >
> > **sigmaY** – Gaussian kernel standard deviation in Y direction. If `sigmaY` is zero, it is set to be equal to `sigmaX`. If both sigmas are zeros, they are computed from `ksize.width` and `ksize.height`, respectively. See `getGaussianKernel()` for details. To fully control the result regardless of possible future modifications of all this semantics, it is recommended to specify all of `ksize`, `sigmaX`, and `sigmaY`.
> >
> > **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.

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

Returns filter coefficients for computing spatial image derivatives.

C++: void **getDerivKernels**(OutputArray **kx**, OutputArray **ky**, int **dx**, int **dy**, int **ksize**, bool **normalize**=false, int **ktype**=CV_32F)

Python: cv2.**getDerivKernels**(dx, dy, ksize[, kx[, ky[, normalize[, ktype]]]]) → kx, ky

> **Parameters**

>> **kx** – Output matrix of row filter coefficients. It has the type `ktype` .

>> **ky** – Output matrix of column filter coefficients. It has the type `ktype` .

>> **dx** – Derivative order in respect of x.

>> **dy** – Derivative order in respect of y.

>> **ksize** – Aperture size. It can be `CV_SCHARR` , 1, 3, 5, or 7.

>> **normalize** – Flag indicating whether to normalize (scale down) the filter coefficients or not. Theoretically, the coefficients should have the denominator $= 2^{\texttt{ksize}*2-\texttt{dx}-\texttt{dy}-2}$. If you are going to filter floating-point images, you are likely to use the normalized kernels. But if you compute derivatives of an 8-bit image, store the results in a 16-bit image, and wish to preserve all the fractional bits, you may want to set `normalize=false` .

>> **ktype** – 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 3×3 kernels are generated (see `Scharr()`). Otherwise, Sobel kernels are generated (see `Sobel()`). The filters are normally passed to `sepFilter2D()` or to `createSeparableLinearFilter()` .

getGaussianKernel

Returns Gaussian filter coefficients.

C++: Mat **getGaussianKernel**(int **ksize**, double **sigma**, int **ktype**=CV_64F)

Python: cv2.**getGaussianKernel**(ksize, sigma[, ktype]) → retval

> **Parameters**

>> **ksize** – Aperture size. It should be odd (`ksize` mod $2 = 1$) and positive.

>> **sigma** – Gaussian standard deviation. If it is non-positive, it is computed from `ksize` as `sigma = 0.3*((ksize-1)*0.5 - 1) + 0.8` .

>> **ktype** – Type of filter coefficients. It can be `CV_32f` or `CV_64F` .

The function computes and returns the `ksize` $\times 1$ matrix of Gaussian filter coefficients:

$$G_i = \alpha * e^{-(i-(\texttt{ksize}-1)/2)^2/(2*\texttt{sigma})^2},$$

where $i = 0..\texttt{ksize} - 1$ and α 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()`. Those functions automatically recognize smoothing kernels (a symmetrical kernel with sum of weights equal to 1) and handle them accordingly. You may also use the higher-level `GaussianBlur()`.

See Also:

`sepFilter2D()`, `createSeparableLinearFilter()`, `getDerivKernels()`, `getStructuringElement()`, `GaussianBlur()`

getKernelType

Returns the kernel type.

C++: int **getKernelType**(InputArray **kernel**, Point **anchor**)

> **Parameters**
>
>> **kernel** – 1D array of the kernel coefficients to analyze.
>>
>> **anchor** – Anchor position within the kernel.

The function analyzes the kernel coefficients and returns the corresponding kernel type:

- **KERNEL_GENERAL** The kernel is generic. It is used when there is no any type of symmetry or other properties.
- **KERNEL_SYMMETRICAL** The kernel is symmetrical: $kernel_i == kernel_{ksize-i-1}$, and the anchor is at the center.
- **KERNEL_ASYMMETRICAL** The kernel is asymmetrical: $kernel_i == -kernel_{ksize-i-1}$, and the anchor is at the center.
- **KERNEL_SMOOTH** All the kernel elements are non-negative and summed to 1. For example, the Gaussian kernel is both smooth kernel and symmetrical, so the function returns `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

Returns a structuring element of the specified size and shape for morphological operations.

C++: Mat **getStructuringElement**(int **shape**, Size **ksize**, Point **anchor**=Point(-1,-1))

Python: cv2.**getStructuringElement**(shape, ksize[, anchor]) → retval

C: IplConvKernel* **cvCreateStructuringElementEx**(int **cols**, int **rows**, int **anchorX**, int **anchorY**, int **shape**, int* **values**=NULL)

Python: cv.**CreateStructuringElementEx**(cols, rows, anchorX, anchorY, shape, values=None) → kernel

> **Parameters**
>
>> **shape** – Element shape that could be one of the following:
>>
>> – **MORPH_RECT** - a rectangular structuring element:
>>
>> $$E_{ij} = 1$$
>>
>> – **MORPH_ELLIPSE** - an elliptic structuring element, that is, a filled ellipse inscribed into the rectangle `Rect(0, 0, esize.width, 0.esize.height)`
>>
>> – **MORPH_CROSS** - a cross-shaped structuring element:
>>
>> $$E_{ij} = \begin{cases} 1 & \text{if i=anchor.y or j=anchor.x} \\ 0 & \text{otherwise} \end{cases}$$
>>
>> – **CV_SHAPE_CUSTOM** - custom structuring element (OpenCV 1.x API)
>>
>> **ksize** – Size of the structuring element.

> **cols** – Width of the structuring element
>
> **rows** – Height of the structuring element
>
> **anchor** – Anchor position within the element. The default value $(-1, -1)$ means that the anchor is at the center. Note that only the shape of a cross-shaped element depends on the anchor position. In other cases the anchor just regulates how much the result of the morphological operation is shifted.
>
> **anchorX** – x-coordinate of the anchor
>
> **anchorY** – y-coordinate of the anchor
>
> **values** – integer array of `cols''*''rows` elements that specifies the custom shape of the structuring element, when `shape=CV_SHAPE_CUSTOM`.

The function constructs and returns the structuring element that can be further passed to `createMorphologyFilter()`, `erode()`, `dilate()` or `morphologyEx()` . But you can also construct an arbitrary binary mask yourself and use it as the structuring element.

Note: When using OpenCV 1.x C API, the created structuring element `IplConvKernel* element` must be released in the end using `cvReleaseStructuringElement(&element)`.

---

medianBlur

Smoothes an image using the median filter.

C++: void **medianBlur**(InputArray **src**, OutputArray **dst**, int **ksize**)

Python: cv2.**medianBlur**(src, ksize[, dst]) $\rightarrow$ dst

> **Parameters**
>
> > **src** – 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** – Destination array of the same size and type as `src` .
> >
> > **ksize** – Aperture linear size. It must be odd and greater than 1, for example: 3, 5, 7 ...

The function smoothes an image using the median filter with the `ksize` $\times$ `ksize` aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

See Also:

`bilateralFilter()`, `blur()`, `boxFilter()`, `GaussianBlur()`

morphologyEx

Performs advanced morphological transformations.

C++: void **morphologyEx**(InputArray **src**, OutputArray **dst**, int **op**, InputArray **element**, Point **anchor**=Point(-1,-1), int **iterations**=1, int **borderType**=BORDER_CONSTANT, const Scalar& **borderValue**=morphologyDefaultBorderValue())

Python: cv2.**morphologyEx**(src, op, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]) $\rightarrow$ dst

C: void **cvMorphologyEx**(const CvArr* **src**, CvArr* **dst**, CvArr* **temp**, IplConvKernel* **element**, int **operation**, int **iterations**=1)

Python: cv.**MorphologyEx**(src, dst, temp, element, operation, iterations=1) → None

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image of the same size and type as `src` .
> >
> > **element** – Structuring element.
> >
> > **op** – Type of a morphological operation that can be one of the following:
> >
> > – **MORPH_OPEN** - an opening operation
> >
> > – **MORPH_CLOSE** - a closing operation
> >
> > – **MORPH_GRADIENT** - a morphological gradient
> >
> > – **MORPH_TOPHAT** - "top hat"
> >
> > – **MORPH_BLACKHAT** - "black hat"
> >
> > **iterations** – Number of times erosion and dilation are applied.
> >
> > **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.
> >
> > **borderValue** – Border value in case of a constant border. The default value has a special meaning. See `createMorphologyFilter()` for details.

The function can perform advanced morphological transformations using an erosion and dilation as basic operations.

Opening operation:

$$\texttt{dst} = \mathrm{open}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\mathrm{erode}(\texttt{src}, \texttt{element}))$$

Closing operation:

$$\texttt{dst} = \mathrm{close}(\texttt{src}, \texttt{element}) = \mathrm{erode}(\mathrm{dilate}(\texttt{src}, \texttt{element}))$$

Morphological gradient:

$$\texttt{dst} = \mathrm{morph\_grad}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\texttt{src}, \texttt{element}) - \mathrm{erode}(\texttt{src}, \texttt{element})$$

"Top hat":

$$\texttt{dst} = \mathrm{tophat}(\texttt{src}, \texttt{element}) = \texttt{src} - \mathrm{open}(\texttt{src}, \texttt{element})$$

"Black hat":

$$\texttt{dst} = \mathrm{blackhat}(\texttt{src}, \texttt{element}) = \mathrm{close}(\texttt{src}, \texttt{element}) - \texttt{src}$$

Any of the operations can be done in-place.

See Also:

`dilate()`, `erode()`, `createMorphologyFilter()`

Laplacian

Calculates the Laplacian of an image.

C++: void **Laplacian**(InputArray **src**, OutputArray **dst**, int **ddepth**, int **ksize**=1, double **scale**=1, double **delta**=0, int **borderType**=BORDER_DEFAULT)

Python: cv2.**Laplacian**(src, ddepth[, dst[, ksize[, scale[, delta[, borderType]]]]]) → dst

C: void **cvLaplace**(const CvArr* **src**, CvArr* **dst**, int **ksize**=3)

Python: cv.**Laplace**(src, dst, ksize=3) → None

>   **Parameters**

>   >   **src** – Source image.

>   >   **dst** – Destination image of the same size and the same number of channels as `src` .

>   >   **ddepth** – Desired depth of the destination image.

>   >   **ksize** – Aperture size used to compute the second-derivative filters. See `getDerivKernels()` for details. The size must be positive and odd.

>   >   **scale** – Optional scale factor for the computed Laplacian values. By default, no scaling is applied. See `getDerivKernels()` for details.

>   >   **delta** – Optional delta value that is added to the results prior to storing them in `dst` .

>   >   **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\mathtt{dst} = \Delta\mathtt{src} = \frac{\partial^2\mathtt{src}}{\partial x^2} + \frac{\partial^2\mathtt{src}}{\partial y^2}$$

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

Smoothes an image and downsamples it.

C++: void **pyrDown**(InputArray **src**, OutputArray **dst**, const Size& **dstsize**=Size())

Python: cv2.**pyrDown**(src[, dst[, dstsize]]) → dst

C: void **cvPyrDown**(const CvArr* **src**, CvArr* **dst**, int **filter**=CV_GAUSSIAN_5x5)

Python: cv.**PyrDown**(src, dst, filter=CV_GAUSSIAN_5X5) → None

>   **Parameters**

>   >   **src** – Source image.

>   >   **dst** – Destination image. It has the specified size and the same type as `src` .

>   >   **dstsize** – 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:

$$|\mathtt{dstsize.width} * 2 - \mathtt{src.cols}| \leq 2$$
$$|\mathtt{dstsize.height} * 2 - \mathtt{src.rows}| \leq 2$$

The function performs the downsampling step of the Gaussian pyramid construction. First, it convolves the source image with the kernel:

$$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Then, it downsamples the image by rejecting even rows and columns.

pyrUp

Upsamples an image and then smoothes it.

C++: void **pyrUp**(InputArray **src**, OutputArray **dst**, const Size& **dstsize**=Size())

Python: cv2.**pyrUp**(src[, dst[, dstsize]]) → dst

> **Parameters**

>> **src** – Source image.

>> **dst** – Destination image. It has the specified size and the same type as `src` .

>> **dstsize** – 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:

$$|\texttt{dstsize.width} - \texttt{src.cols} * 2| \leq (\texttt{dstsize.width} \mod 2)$$
$$|\texttt{dstsize.height} - \texttt{src.rows} * 2| \leq (\texttt{dstsize.height} \mod 2)$$

The function performs the upsampling step of the Gaussian pyramid construction though 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.

pyrMeanShiftFiltering

Performs initial step of meanshift segmentation of an image.

C++: void **pyrMeanShiftFiltering**(InputArray **src**, OutputArray **dst**, double **sp**, double **sr**, int **maxLevel**=1, TermCriteria **term-crit**=TermCriteria(TermCriteria::MAX_ITER+TermCriteria::EPS,5,1))

Python: cv2.**pyrMeanShiftFiltering**(src, sp, sr[, dst[, maxLevel[, termcrit]]]) → dst

C: void **cvPyrMeanShiftFiltering**(const CvArr* **src**, CvArr* **dst**, double **sp**, double **sr**, int **max_level**=1, CvTermCriteria **term-crit**=cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,5,1))

Python: cv.**PyrMeanShiftFiltering**(src, dst, sp, sr, maxLevel=1, term-crit=(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 5, 1)) → None

> **Parameters**

>> **src** – The source 8-bit, 3-channel image.

>> **dst** – The destination image of the same format and the same size as the source.

> **sp** – The spatial window radius.
>
> **sr** – The color window radius.
>
> **maxLevel** – Maximum level of the pyramid for the segmentation.
>
> **termcrit** – Termination criteria: when to stop meanshift iterations.

The function implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X,Y) neighborhood in the joint space-color hyperspace is considered:

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

where (R,G,B) and (r,g,b) are the vectors of color components at (X,Y) and (x,y), respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value (X',Y') and average color vector (R',G',B') are found and they act as the neighborhood center on the next iteration:

$$(X, Y) \ (X', Y'), (R, G, B) \ (R', G', B').$$

After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration):

$$I(X, Y) < -(R*, G*, B*)$$

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

sepFilter2D

Applies a separable linear filter to an image.

C++: void **sepFilter2D**(InputArray **src**, OutputArray **dst**, int **ddepth**, InputArray **rowKernel**, InputArray **columnKernel**, Point **anchor**=Point(-1,-1), double **delta**=0, int **borderType**=BORDER_DEFAULT)

Python: cv2.**sepFilter2D**(src, ddepth, kernelX, kernelY[, dst[, anchor[, delta[, borderType]]]]) → dst

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image of the same size and the same number of channels as `src` .
> >
> > **ddepth** – Destination image depth.
> >
> > **rowKernel** – Coefficients for filtering each row.
> >
> > **columnKernel** – Coefficients for filtering each column.

> **anchor** – Anchor position within the kernel. The default value $(-1, 1)$ means that the anchor is at the kernel center.

> **delta** – Value added to the filtered results before storing them.

> **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.

The function applies a separable linear filter to the image. That is, first, every row of `src` is filtered with the 1D kernel `rowKernel`. Then, every column of the result is filtered with the 1D kernel `columnKernel`. The final result shifted by `delta` is stored in `dst`.

See Also:

`createSeparableLinearFilter()`, `filter2D()`, `Sobel()`, `GaussianBlur()`, `boxFilter()`, `blur()`

Smooth

Smooths the image in one of several ways.

C: void **cvSmooth**(const CvArr* **src**, CvArr* **dst**, int **smoothtype**=CV_GAUSSIAN, int **param1**=3, int **param2**=0, double **param3**=0, double **param4**=0)

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

> **Parameters**

>> **src** – The source image

>> **dst** – The destination image

>> **smoothtype** – Type of the smoothing:

>>> – **CV_BLUR_NO_SCALE** linear convolution with `param1` $\times$ `param2` box kernel (all 1's). If you want to smooth different pixels with different-size box kernels, you can use the integral image that is computed using `integral()`

>>> – **CV_BLUR** linear convolution with `param1`$\times$`param2` box kernel (all 1's) with subsequent scaling by $1/(\text{param1} \cdot \text{param2})$

>>> – **CV_GAUSSIAN** linear convolution with a `param1` $\times$ `param2` Gaussian kernel

>>> – **CV_MEDIAN** median filter with a `param1` $\times$ `param1` square aperture

>>> – **CV_BILATERAL** bilateral filter with a `param1` $\times$ `param1` square aperture, color sigma= `param3` and spatial sigma= `param4` . If `param1=0` , the aperture square side is set to `cvRound(param4*1.5)*2+1` . Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

>> **param1** – The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)

>> **param2** – The second parameter of the smoothing operation, the aperture height. Ignored by `CV_MEDIAN` and `CV_BILATERAL` methods. In the case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1` . Otherwise it must be a positive odd number.

>> **param3** – In the case of a Gaussian parameter this parameter may specify Gaussian σ (standard deviation). If it is zero, it is calculated from the kernel size:

$$\sigma = 0.3(n/2 - 1) + 0.8 \quad \text{where} \quad n = \begin{array}{l} \texttt{param1} \text{ for horizontal kernel} \\ \texttt{param2} \text{ for vertical kernel} \end{array}$$

Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function smooths an image using one of several methods. Every of the methods has some features and restrictions listed below:

- Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to `Sobel()` and `Laplacian()`) and 32-bit floating point to 32-bit floating-point format.

- Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

- Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

Note: The function is now obsolete. Use `GaussianBlur()`, `blur()`, `medianBlur()` or `bilateralFilter()`.

Sobel

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

C++: void **Sobel**(InputArray **src**, OutputArray **dst**, int **ddepth**, int **xorder**, int **yorder**, int **ksize**=3, double **scale**=1, double **delta**=0, int **borderType**=BORDER_DEFAULT)

Python: cv2.**Sobel**(src, ddepth, dx, dy$\left[\right.$, dst$\left[\right.$, ksize$\left[\right.$, scale$\left[\right.$, delta$\left[\right.$, borderType $\left.\right]\left.\right]\left.\right]\left.\right]\left.\right]$) → dst

C: void **cvSobel**(const CvArr* **src**, CvArr* **dst**, int **xorder**, int **yorder**, int **apertureSize**=3)

Python: cv.**Sobel**(src, dst, xorder, yorder, apertureSize=3) → None

> **Parameters**

> > **src** – Source image.

> > **dst** – Destination image of the same size and the same number of channels as `src` .

> > **ddepth** – Destination image depth.

> > **xorder** – Order of the derivative x.

> > **yorder** – Order of the derivative y.

> > **ksize** – Size of the extended Sobel kernel. It must be 1, 3, 5, or 7.

> > **scale** – Optional scale factor for the computed derivative values. By default, no scaling is applied. See `getDerivKernels()` for details.

> > **delta** – Optional delta value that is added to the results prior to storing them in `dst` .

> > **borderType** – Pixel extrapolation method. See `borderInterpolate()` for details.

In all cases except one, the `ksize × ksize` separable kernel is used to calculate the derivative. When `ksize = 1`, the 3×1 or 1×3 kernel is used (that is, 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 the 3×3 Scharr filter that may give more accurate results than the 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 an image derivative by convolving the image with the appropriate kernel:

$$dst = \frac{\partial^{xorder+yorder} src}{\partial x^{xorder} \partial y^{yorder}}$$

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:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The second case corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

See Also:

Scharr(), Laplacian(), sepFilter2D(), filter2D(), GaussianBlur()

Scharr

Calculates the first x- or y- image derivative using Scharr operator.

C++: void **Scharr**(InputArray **src**, OutputArray **dst**, int **ddepth**, int **xorder**, int **yorder**, double **scale**=1, double **delta**=0, int **borderType**=BORDER_DEFAULT)

Python: cv2.**Scharr**(src, ddepth, dx, dy[, dst[, scale[, delta[, borderType]]]]) → dst

> **Parameters**

>> **src** – Source image.

>> **dst** – Destination image of the same size and the same number of channels as src .

>> **ddepth** – Destination image depth.

>> **xorder** – Order of the derivative x.

>> **yorder** – Order of the derivative y.

>> **scale** – Optional scale factor for the computed derivative values. By default, no scaling is applied. See getDerivKernels() for details.

>> **delta** – Optional delta value that is added to the results prior to storing them in dst .

>> **borderType** – Pixel extrapolation method. See borderInterpolate() for details.

The function computes the first x- or y- spatial image derivative using the 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, borderType).
```

3.2 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. 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:

The functions in this section perform various geometrical transformations of 2D images. 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: .. math:

```
\texttt{dst} (x,y)= \texttt{src} (f_x(x,y), f_y(x,y))
```

In case when you specify the forward mapping $\langle g_x, g_y \rangle : \texttt{src} \to \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping $\langle f_x, f_y \rangle : \texttt{dst} \to \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic `remap()` and to the simplest and the fastest `resize()` , need to solve two 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) , either one of $f_x(x, y)$, or $f_y(x, y)$, or both of them may fall outside of the image. In this case, an extrapolation method needs to be used. OpenCV provides the same selection of extrapolation methods as in the filtering functions. In addition, it provides the method `BORDER_TRANSPARENT` . This 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. This means that $\langle f_x, f_y \rangle$ can be either an affine or perspective transformation, or radial lens distortion correction, and so on. So, a pixel value 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 can be used. This is called a nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods , where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x, y), f_y(x, y))$, and then the value of the polynomial at $(f_x(x, y), f_y(x, y))$ is taken as the interpolated pixel value. In OpenCV, you can choose between several interpolation methods. See `resize()` for details.

convertMaps

Converts image transformation maps from one representation to another.

C++: void **convertMaps**(InputArray **map1**, InputArray **map2**, OutputArray **dstmap1**, OutputArray **dstmap2**, int **dstmap1type**, bool **nninterpolation**=false)

Python: cv2.**convertMaps**(map1, map2, dstmap1type[, dstmap1[, dstmap2[, nninterpolation]]]) → dstmap1, dstmap2

> Parameters
>
> > **map1** – The first input map of type `CV_16SC2` , `CV_32FC1` , or `CV_32FC2` .
> >
> > **map2** – The second input map of type `CV_16UC1` , `CV_32FC1` , or none (empty matrix), respectively.
> >
> > **dstmap1** – The first output map that has the type `dstmap1type` and the same size as `src` .
> >
> > **dstmap2** – The second output map.

> **dstmap1type** – Type of the first output map that should be `CV_16SC2` , `CV_32FC1` , or `CV_32FC2` .

> **nninterpolation** – Flag indicating whether the fixed-point maps are used for the nearest-neighbor or for a more complex interpolation.

The function converts a pair of maps for `remap()` from one representation to another. The following options (`(map1.type(), map2.type())` → `(dstmap1.type(), dstmap2.type())`) are supported:

- `(CV_32FC1, CV_32FC1)` → `(CV_16SC2, CV_16UC1)` . This is the most frequently used conversion operation, in which the original floating-point maps (see `remap()`) are converted to a more compact and much faster fixed-point representation. The first output array contains the rounded coordinates and the second array (created only when `nninterpolation=false`) contains indices in the interpolation tables.

- `(CV_32FC2)` → `(CV_16SC2, CV_16UC1)` . The same as above but the original maps are stored in one 2-channel matrix.

- Reverse conversion. Obviously, the reconstructed floating-point maps will not be exactly the same as the originals.

See Also:

`remap()`, `undistort()`, `initUndistortRectifyMap()`

getAffineTransform

Calculates an affine transform from three pairs of the corresponding points.

C++: Mat **getAffineTransform**(const Point2f* **src**, const Point2f* **dst**)

Python: cv2.**getAffineTransform**(src, dst) → retval

C: CvMat* **cvGetAffineTransform**(const CvPoint2D32f* **src**, const CvPoint2D32f* **dst**, CvMat* **mapMatrix**)

Python: cv.**GetAffineTransform**(src, dst, mapMatrix) → None

> **Parameters**

> > **src** – Coordinates of triangle vertices in the source image.

> > **dst** – Coordinates of the corresponding triangle vertices in the destination image.

The function calculates the 2×3 matrix of an affine transform so that:

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x_i', y_i'), src(i) = (x_i, y_i), i = 0, 1, 2$$

See Also:

`warpAffine()`, `transform()`

getPerspectiveTransform

Calculates a perspective transform from four pairs of the corresponding points.

C++: Mat **getPerspectiveTransform**(const Point2f* **src**, const Point2f* **dst**)

Python: cv2.**getPerspectiveTransform**(src, dst) → retval

C: CvMat* **cvGetPerspectiveTransform**(const CvPoint2D32f* **src**, const CvPoint2D32f* **dst**, CvMat* **mapMatrix**)

Python: cv.**GetPerspectiveTransform**(src, dst, mapMatrix) → None

> **Parameters**

>> **src** – Coordinates of quadrangle vertices in the source image.

>> **dst** – Coordinates of the corresponding quadrangle vertices in the destination image.

The function calculates the 3×3 matrix of a perspective transform so that:

$$\begin{bmatrix} t_i x_i' \\ t_i y_i' \\ t_i \end{bmatrix} = \mathtt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x_i', y_i'), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

See Also:

findHomography(), warpPerspective(), perspectiveTransform()

getRectSubPix

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

C++: void **getRectSubPix**(InputArray **image**, Size **patchSize**, Point2f **center**, OutputArray **dst**, int **patchType**=-1)

Python: cv2.**getRectSubPix**(image, patchSize, center[, patch[, patchType]]) → patch

C: void **cvGetRectSubPix**(const CvArr* **src**, CvArr* **dst**, CvPoint2D32f **center**)

Python: cv.**GetRectSubPix**(src, dst, center) → None

> **Parameters**

>> **src** – Source image.

>> **patchSize** – Size of the extracted patch.

>> **center** – Floating point coordinates of the center of the extracted rectangle within the source image. The center must be inside the image.

>> **dst** – Extracted patch that has the size patchSize and the same number of channels as src .

>> **patchType** – Depth of the extracted pixels. By default, they have the same depth as src .

The function getRectSubPix extracts pixels from src :

$$dst(x, y) = src(x + \mathtt{center.x} - (\mathtt{dst.cols} - 1) * 0.5, y + \mathtt{center.y} - (\mathtt{dst.rows} - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multi-channel images is processed independently. While the center of the rectangle 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

Calculates an affine matrix of 2D rotation.

C++: Mat **getRotationMatrix2D**(Point2f **center**, double **angle**, double **scale**)

Python: cv2.**getRotationMatrix2D**(center, angle, scale) → retval

C: CvMat* **cv2DRotationMatrix**(CvPoint2D32f **center**, double **angle**, double **scale**, CvMat* **mapMatrix**)

Python: cv.**GetRotationMatrix2D**(center, angle, scale, mapMatrix) → None

> **Parameters**

>> **center** – Center of the rotation in the source image.

>> **angle** – Rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner).

>> **scale** – Isotropic scale factor.

>> **mapMatrix** – The output affine transformation, 2x3 floating-point matrix.

The function calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} + (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot \cos \texttt{angle},$$
$$\beta = \texttt{scale} \cdot \sin \texttt{angle}$$

The transformation maps the rotation center to itself. If this is not the target, adjust the shift.

See Also:

getAffineTransform(), warpAffine(), transform()

invertAffineTransform

Inverts an affine transformation.

C++: void **invertAffineTransform**(InputArray **M**, OutputArray **iM**)

Python: cv2.**invertAffineTransform**(M[, iM]) → iM

> **Parameters**

>> **M** – Original affine transformation.

>> **iM** – Output reverse affine transformation.

The function computes an inverse affine transformation represented by 2×3 matrix M :

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

The result is also a 2×3 matrix of the same type as M .

LogPolar

Remaps an image to log-polar space.

C: void **cvLogPolar**(const CvArr* **src**, CvArr* **dst**, CvPoint2D32f **center**, double **M**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS)

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

> **Parameters**
>> **src** – Source image
>>
>> **dst** – Destination image
>>
>> **center** – The transformation center; where the output precision is maximal
>>
>> **M** – Magnitude scale parameter. See below
>>
>> **flags** – A combination of interpolation methods and the following optional flags:
>>
>>> – **CV_WARP_FILL_OUTLIERS** fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero
>>>
>>> – **CV_WARP_INVERSE_MAP** See below

The function `cvLogPolar` transforms the source image using the following transformation:

• Forward transformation (`CV_WARP_INVERSE_MAP` is not set):

$$dst(\phi, \rho) = src(x, y)$$

• Inverse transformation (`CV_WARP_INVERSE_MAP` is set):

$$dst(x, y) = src(\phi, \rho)$$

where

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

remap

Applies a generic geometrical transformation to an image.

C++: void **remap**(InputArray **src**, OutputArray **dst**, InputArray **map1**, InputArray **map2**, int **interpolation**, int **borderMode**=BORDER_CONSTANT, const Scalar& **borderValue**=Scalar())

Python: cv2.**remap**(src, map1, map2, interpolation[, dst[, borderMode[, borderValue]]]) → dst

C: void **cvRemap**(const CvArr* **src**, CvArr* **dst**, const CvArr* **mapx**, const CvArr* **mapy**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar **fillval**=cvScalarAll(0))

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

> **Parameters**

>> **src** – Source image.

>> **dst** – Destination image. It has the same size as `map1` and the same type as `src` .

>> **map1** – The first map of either (x,y) points or just x values having the type CV_16SC2 , CV_32FC1 , or CV_32FC2 . See `convertMaps()` for details on converting a floating point representation to fixed-point for speed.

>> **map2** – The second map of y values having the type CV_16UC1 , CV_32FC1 , or none (empty map if map1 is (x,y) points), respectively.

>> **interpolation** – Interpolation method (see `resize()`). The method INTER_AREA is not supported by this function.

>> **borderMode** – Pixel extrapolation method (see `borderInterpolate()`). When 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** – Value used in case of a constant border. By default, it is 0.

The function remap transforms the source image using the specified map:

$$\mathtt{dst}(x,y) = \mathtt{src}(map_x(x,y), map_y(x,y))$$

where values of pixels with non-integer coordinates are computed using one of available interpolation methods. map_x and map_y can be encoded as separate floating-point maps in map_1 and map_2 respectively, or interleaved floating-point maps of (x,y) in map_1 , or fixed-point maps created by using `convertMaps()` . The reason you might want to convert from floating to fixed-point representations of a map is that they can yield much faster (~2x) remapping operations. In the converted case, map_1 contains pairs (cvFloor(x), cvFloor(y)) and map_2 contains indices in a table of interpolation coefficients.

This function cannot operate in-place.

resize

Resizes an image.

C++: void **resize**(InputArray **src**, OutputArray **dst**, Size **dsize**, double **fx**=0, double **fy**=0, int **interpolation**=INTER_LINEAR)

Python: cv2.**resize**(src, dsize[, dst[, fx[, fy[, interpolation]]]]) → dst

C: void **cvResize**(const CvArr* **src**, CvArr* **dst**, int **interpolation**=CV_INTER_LINEAR)

Python: cv.**Resize**(src, dst, interpolation=CV_INTER_LINEAR) → None

> **Parameters**

>> **src** – Source image.

>> **dst** – Destination image. It has the size dsize (when it is non-zero) or the size computed from src.size() , fx , and fy . The type of dst is the same as of src .

>> **dsize** – Destination image size. If it is zero, it is computed as:

>>> dsize = Size(round(fx*src.cols), round(fy*src.rows))

>> Either dsize or both fx and fy must be non-zero.

> **fx** – Scale factor along the horizontal axis. When it is 0, it is computed as
>
> $$\texttt{(double)dsize.width/src.cols}$$
>
> **fy** – Scale factor along the vertical axis. When it is 0, it is computed as
>
> $$\texttt{(double)dsize.height/src.rows}$$
>
> **interpolation** – Interpolation method:
>
> – **INTER_NEAREST** - a nearest-neighbor interpolation
>
> – **INTER_LINEAR** - a bilinear interpolation (used by default)
>
> – **INTER_AREA** - resampling using pixel area relation. It may be a 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** - a bicubic interpolation over 4x4 pixel neighborhood
>
> – **INTER_LANCZOS4** - a Lanczos interpolation over 8x8 pixel neighborhood

The function `resize` resizes the 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 follows:

```
// explicitly specify dsize=dst.size(); fx and fy will be computed from that.
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 compute the destination image size.
resize(src, dst, Size(), 0.5, 0.5, interpolation);
```

To shrink an image, it will generally look best with CV_INTER_AREA interpolation, whereas to enlarge an image, it will generally look best with CV_INTER_CUBIC (slow) or CV_INTER_LINEAR (faster but still looks OK).

See Also:

warpAffine(), warpPerspective(), remap()

warpAffine

Applies an affine transformation to an image.

C++: void **warpAffine**(InputArray **src**, OutputArray **dst**, InputArray **M**, Size **dsize**, int **flags**=INTER_LINEAR, int **borderMode**=BORDER_CONSTANT, const Scalar& **borderValue**=Scalar())

Python: cv2.**warpAffine**(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) → dst

C: void **cvWarpAffine**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mapMatrix**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar **fillval**=cvScalarAll(0))

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

C: void **cvGetQuadrangleSubPix**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mapMatrix**)

Python: cv.**GetQuadrangleSubPix**(src, dst, mapMatrix) → None

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image that has the size `dsize` and the same type as `src` .
> >
> > **M** – 2×3 transformation matrix.
> >
> > **dsize** – Size of the destination image.
> >
> > **flags** – Combination of interpolation methods (see `resize()`) and the optional flag `WARP_INVERSE_MAP` that means that `M` is the inverse transformation (`dst` $\rightarrow$ `src`).
> >
> > **borderMode** – Pixel extrapolation method (see `borderInterpolate()`). When `borderMode=BORDER_TRANSPARENT` , it means that the pixels in the destination image corresponding to the "outliers" in the source image are not modified by the function.
> >
> > **borderValue** – 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 cannot operate in-place.

See Also:

`warpPerspective()`, `resize()`, `remap()`, `getRectSubPix()`, `transform()`

Note: `cvGetQuadrangleSubPix` is similar to `cvWarpAffine`, but the outliers are extrapolated using replication border mode.

---

warpPerspective

Applies a perspective transformation to an image.

C++: void **warpPerspective**(InputArray **src**, OutputArray **dst**, InputArray **M**, Size **dsize**, int **flags**=INTER_LINEAR, int **borderMode**=BORDER_CONSTANT, const Scalar& **borderValue**=Scalar())

Python: cv2.**warpPerspective**(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) $\rightarrow$ dst

C: void **cvWarpPerspective**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mapMatrix**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar **fillval**=cvScalarAll(0))

Python: cv.**WarpPerspective**(src, dst, mapMatrix, flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS, fillval=(0, 0, 0, 0)) $\rightarrow$ None

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image that has the size `dsize` and the same type as `src` .
> >
> > **M** – 3×3 transformation matrix.
> >
> > **dsize** – Size of the destination image.
> >
> > **flags** – Combination of interpolation methods (see `resize()`) and the optional flag `WARP_INVERSE_MAP` that means that `M` is the inverse transformation (`dst` $\rightarrow$ `src`).

> **borderMode** – Pixel extrapolation method (see `borderInterpolate()`). When `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** – Value used in case of a constant border. By default, it is 0.

The function `warpPerspective` transforms the source image using the specified matrix:

$$\texttt{dst}(x, y) = \texttt{src}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right)$$

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 cannot operate in-place.

See Also:

`warpAffine()`, `resize()`, `remap()`, `getRectSubPix()`, `perspectiveTransform()`

initUndistortRectifyMap

Computes the undistortion and rectification transformation map.

C++: void **initUndistortRectifyMap**(InputArray **cameraMatrix**, InputArray **distCoeffs**, InputArray **R**, InputArray **newCameraMatrix**, Size **size**, int **m1type**, OutputArray **map1**, OutputArray **map2**)

Python: cv2.**initUndistortRectifyMap**(cameraMatrix, distCoeffs, R, newCameraMatrix, size, m1type[, map1[, map2]]) → map1, map2

C: void **cvInitUndistortRectifyMap**(const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, const Cv-Mat* **R**, const CvMat* **newCameraMatrix**, CvArr* **map1**, CvArr* **map2**)

C: void **cvInitUndistortMap**(const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, CvArr* **map1**, CvArr* **map2**)

Python: cv.**InitUndistortRectifyMap**(cameraMatrix, distCoeffs, R, newCameraMatrix, map1, map2) → None

Python: cv.**InitUndistortMap**(cameraMatrix, distCoeffs, map1, map2) → None

> **Parameters**

> > **cameraMatrix** – Input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.

> > **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

> > **R** – Optional rectification transformation in the object space (3x3 matrix). `R1` or `R2` , computed by `stereoRectify()` can be passed here. If the matrix is empty, the identity transformation is assumed. In `cvInitUndistortMap` R assumed to be an identity matrix.

> > **newCameraMatrix** – New camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$.

> > **size** – Undistorted image size.

> > **m1type** – Type of the first output map that can be `CV_32FC1` or `CV_16SC2` . See `convertMaps()` for details.

> > **map1** – The first output map.

> **map2** – The second output map.

The function computes the joint undistortion and rectification transformation and represents the result in the form of maps for `remap()` . The undistorted image looks like original, as if it is captured with a camera using the camera matrix =newCameraMatrix and zero distortion. In case of a monocular camera, `newCameraMatrix` is usually equal to `cameraMatrix` , or it can be computed by `getOptimalNewCameraMatrix()` for a better control over scaling. In case of a stereo camera, `newCameraMatrix` is normally set to P1 or P2 computed by `stereoRectify()` .

Also, this new camera is 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 case of a 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 (that is, in the original image from camera). The following process is applied:

$$
\begin{aligned}
&x \leftarrow (u - c'_x)/f'_x \\
&y \leftarrow (v - c'_y)/f'_y \\
&[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\
&x' \leftarrow X/W \\
&y' \leftarrow Y/W \\
&x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x'y' + p_2(r^2 + 2x'^2) \\
&y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x'y' \\
&map_x(u, v) \leftarrow x"f_x + c_x \\
&map_y(u, v) \leftarrow y"f_y + c_y
\end{aligned}
$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In 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 a pixel domain, not a rotation matrix R in 3D space. R can be computed from H as

$$
R = \texttt{cameraMatrix}^{-1} \cdot H \cdot \texttt{cameraMatrix}
$$

where `cameraMatrix` can be chosen arbitrarily.

getDefaultNewCameraMatrix

Returns the default new camera matrix.

C++: Mat **getDefaultNewCameraMatrix**(InputArray **cameraMatrix**, Size **imgSize**=Size(), bool **center-PrincipalPoint**=false)

Python: cv2.**getDefaultNewCameraMatrix**(cameraMatrix[, imgsize[, centerPrincipalPoint]]) → retval

> **Parameters**
>
> > **cameraMatrix** – Input camera matrix.
> >
> > **imageSize** – Camera view image size in pixels.
> >
> > **centerPrincipalPoint** – Location of the principal point in the new camera matrix. The parameter indicates whether this location 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 `centerPrinicipalPoint=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 is important to move the principal points in both views to the same y-coordinate (which is required by most of stereo correspondence algorithms), and may be to the same x-coordinate too. So, you can form the new camera matrix for each view where the principal points are located at the center.

undistort

Transforms an image to compensate for lens distortion.

C++: void **undistort**(InputArray **src**, OutputArray **dst**, InputArray **cameraMatrix**, InputArray **distCoeffs**, InputArray **newCameraMatrix**=noArray())

Python: cv2.**undistort**(src, cameraMatrix, distCoeffs$\big[$, dst$\big[$, newCameraMatrix $\big]\big]$) $\rightarrow$ dst

C: void **cvUndistort2**(const CvArr* **src**, CvArr* **dst**, const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, const CvMat* **newCameraMatrix**=NULL)

Python: cv.**Undistort2**(src, dst, cameraMatrix, distCoeffs) $\rightarrow$ None

> **Parameters**
>
> > **src** – Input (distorted) image.
> >
> > **dst** – Output (corrected) image that has the same size and type as `src` .
> >
> > **cameraMatrix** – Input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
> >
> > **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
> >
> > **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 a different matrix.

The function transforms an 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 zeros (black color).

A 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 resolution 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

Computes the ideal point coordinates from the observed point coordinates.

C++: void **undistortPoints**(InputArray **src**, OutputArray **dst**, InputArray **cameraMatrix**, InputArray **distCoeffs**, InputArray **R**=noArray(), InputArray **P**=noArray())

C: void **cvUndistortPoints**(const CvMat* **src**, CvMat* **dst**, const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, const CvMat* **R**=NULL, const CvMat* **P**=NULL)

Python: cv.**UndistortPoints**(src, dst, cameraMatrix, distCoeffs, R=None, P=None) → None

> **Parameters**
>
> > **src** – Observed point coordinates, 1xN or Nx1 2-channel (CV_32FC2 or CV_64FC2).
> >
> > **dst** – Output ideal point coordinates after undistortion and reverse perspective transformation.
> >
> > **cameraMatrix** – Camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
> >
> > **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
> >
> > **R** – Rectification transformation in the 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** – New camera matrix (3x3) or 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 performs a reverse transformation to projectPoints() . In case of a 3D object, it does not reconstruct its 3D coordinates, but for a planar object, it does, 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 an 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 for both a stereo camera head or a monocular camera (when R is empty).

3.3 Miscellaneous Image Transformations

adaptiveThreshold

Applies an adaptive threshold to an array.

C++: void **adaptiveThreshold**(InputArray **src**, OutputArray **dst**, double **maxValue**, int **adaptiveMethod**, int **thresholdType**, int **blockSize**, double **C**)

Python: cv2.**adaptiveThreshold**(src, maxValue, adaptiveMethod, thresholdType, blockSize, C[, dst]) → dst

C: void **cvAdaptiveThreshold**(const CvArr* **src**, CvArr* **dst**, double **maxValue**, int **adaptiveMethod**=CV_ADAPTIVE_THRESH_MEAN_C, int **thresholdType**=CV_THRESH_BINARY, int **blockSize**=3, double **param1**=5)

Python: cv.**AdaptiveThreshold**(src, dst, maxValue, adaptiveMethod=CV_ADAPTIVE_THRESH_MEAN_C, thresholdType=CV_THRESH_BINARY, blockSize=3, param1=5) → None

> Parameters

>> **src** – Source 8-bit single-channel image.

>> **dst** – Destination image of the same size and the same type as src .

>> **maxValue** – Non-zero value assigned to the pixels for which the condition is satisfied. See the details below.

>> **adaptiveMethod** – Adaptive thresholding algorithm to use, ADAPTIVE_THRESH_MEAN_C or ADAPTIVE_THRESH_GAUSSIAN_C . See the details below.

>> **thresholdType** – Thresholding type that must be either THRESH_BINARY or THRESH_BINARY_INV .

>> **blockSize** – Size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on.

>> **C** – Constant subtracted from the mean or weighted mean (see the details below). Normally, it is positive but may be zero or negative as well.

The function transforms a grayscale image to a binary image according to the formulae:

- **THRESH_BINARY**

$$dst(x, y) = \begin{cases} \texttt{maxValue} & \text{if } src(x, y) > T(x, y) \\ 0 & \text{otherwise} \end{cases}$$

- **THRESH_BINARY_INV**

$$dst(x, y) = \begin{cases} 0 & \text{if } src(x, y) > T(x, y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

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 a mean of the blockSize × blockSize neighborhood of (x, y) minus C .

- For the method `ADAPTIVE_THRESH_GAUSSIAN_C` , the threshold value $T(x, y)$ is a weighted sum (cross-correlation with a Gaussian window) of the `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

Converts an image from one color space to another.

C++: void **cvtColor**(InputArray **src**, OutputArray **dst**, int **code**, int **dstCn**=0)

Python: cv2.**cvtColor**(src, code[, dst[, dstCn]]) → dst

C: void **cvCvtColor**(const CvArr* **src**, CvArr* **dst**, int **code**)

Python: cv.**CvtColor**(src, dst, code) → None

> **Parameters**
>
>> **src** – Source image: 8-bit unsigned, 16-bit unsigned (`CV_16UC...`), or single-precision floating-point.
>>
>> **dst** – Destination image of the same size and depth as `src` .
>>
>> **code** – Color space conversion code. See the description below.
>>
>> **dstCn** – Number of channels in the destination image. If the parameter is 0, the number of the channels is derived automatically from `src` and `code` .

The function converts an input image from one color space to another. In case of a transformation to-from RGB color space, the order of the channels should be specified explicitly (RGB or BGR). Note that the default color format in OpenCV is often referred to as RGB but it is actually BGR (the bytes are reversed). So the first byte in a standard (24-bit) color image will be an 8-bit Blue component, the second byte will be Green, and the third byte will be Red. The fourth, fifth, and sixth bytes would then be the second pixel (Blue, then Green, then Red), and so on.

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
- 0 to 1 for `CV_32F` images

In case of linear transformations, the range does not matter. But in case of a non-linear transformation, an input RGB image should be normalized to the proper value range to get the correct results, for example, for RGB → L*u*v* transformation. For example, if you have a 32-bit floating-point image directly converted from an 8-bit image without any scaling, then it will have the 0..255 value range instead of 0..1 assumed by the function. So, before calling `cvtColor` , you need first to scale the image down:

```
img *= 1./255;
cvtColor(img, img, CV_BGR2Luv);
```

If you use `cvtColor` with 8-bit images, the conversion will have some information lost. For many applications, this will not be noticeable but it is recommended to use 32-bit images in applications that need the full range of colors or that convert an image before an operation and then convert back.

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

$$\text{RGB[A] to Gray:} \quad Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

and

$$\text{Gray to RGB[A]:} \quad R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

The conversion from a RGB image to gray is done with:

```
cvtColor(src, bwsrc, CV_RGB2GRAY);
```

More advanced channel reordering can also be done with `mixChannels()`.

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB`):

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

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

X, Y and Z cover the whole value range (in case of floating-point images, Z may exceed 1).

- RGB $\leftrightarrow$ YCrCb JPEG (or YCC) (`CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB`)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

Y, Cr, and Cb cover the whole value range.

- **RGB $\leftrightarrow$ HSV** (`CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB`) In case of 8-bit and 16-bit images, R, G, and B are converted to the floating-point format and scaled to fit the 0 to 1 range.

$$V \leftarrow \max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - \min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/(V - \min(R,G,B)) & \text{if } V = R \\ 120 + 60(B - R)/(V - \min(R,G,B)) & \text{if } V = G \\ 240 + 60(R - G)/(V - \min(R,G,B)) & \text{if } V = B \end{cases}$$

If $H < 0$ then $H \leftarrow H + 360$. On output $0 \leq V \leq 1, 0 \leq S \leq 1, 0 \leq H \leq 360$.

The values are then converted to the destination data type:

- 8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 (\text{to fit to 0 to 255})$$

- 16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

- **32-bit images** H, S, and V are left as is

- **RGB** $\leftrightarrow$ **HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`).** In case of 8-bit and 16-bit images, R, G, and B are converted to the floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow \max(R,G,B)$$

$$V_{min} \leftarrow \min(R,G,B)$$

$$L \leftarrow \frac{V_{max} + V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max} - V_{min}}{V_{max} + V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max} - V_{min}}{2 - (V_{max} + V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V_{max} = R \\ 120 + 60(B - R)/S & \text{if } V_{max} = G \\ 240 + 60(R - G)/S & \text{if } V_{max} = B \end{cases}$$

If $H < 0$ then $H \leftarrow H + 360$. On output $0 \leq L \leq 1, 0 \leq S \leq 1, 0 \leq H \leq 360$.

The values are then converted to the destination data type:

- 8-bit images

$$V \leftarrow 255 \cdot V, S \leftarrow 255 \cdot S, H \leftarrow H/2 \ (\text{to fit to 0 to 255})$$

- 16-bit images (currently not supported)

$$V < -65535 \cdot V, S < -65535 \cdot S, H < -H$$

– **32-bit images** H, S, V are left as is

- **RGB $\leftrightarrow$ CIE L*a*b* (CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB).** In case of 8-bit and 16-bit images, R, G, and B are converted to the floating-point format and scaled to fit the 0 to 1 range.

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

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

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

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

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

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

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

$$\text{delta} = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

This outputs $0 \leq L \leq 100, -127 \leq a \leq 127, -127 \leq b \leq 127$. 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, and b are left as is

- **RGB $\leftrightarrow$ CIE L*u*v* (CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB).** In case of 8-bit and 16-bit images, R, G, and B are converted to the floating-point format and scaled to fit 0 to 1 range.

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

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

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

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

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

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

This outputs $0 \le L \le 100, -134 \le u \le 220, -140 \le v \le 122$.

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, and v are left as is

The above formulae for converting RGB to/from various color spaces have been taken from multiple sources on the 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 enables you to get color pictures from a single plane where R,G, and B pixels (sensors of a particular component) are interleaved as follows:
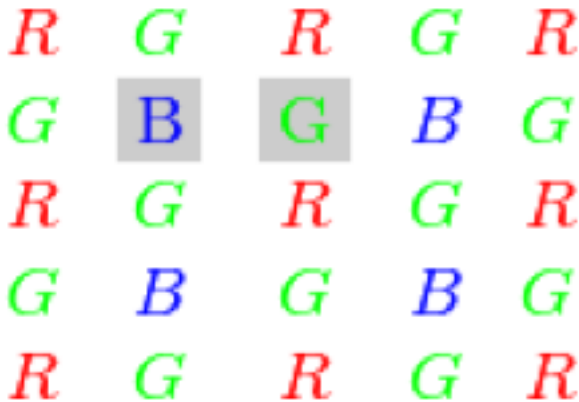
The output RGB components of a pixel are interpolated from 1, 2, or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants CV_Bayer $C_1 C_2$ 2BGR and CV_Bayer $C_1 C_2$ 2RGB indicate the particular pattern type. These are components from the second row, second and third columns, respectively. For example, the above pattern has a very popular "BG" type.

distanceTransform

Calculates the distance to the closest zero pixel for each pixel of the source image.

C++: void **distanceTransform**(InputArray **src**, OutputArray **dst**, int **distanceType**, int **maskSize**)

C++: void **distanceTransform**(InputArray **src**, OutputArray **dst**, OutputArray **labels**, int **distanceType**, int **maskSize**)

Python: cv2.**distanceTransform**(src, distanceType, maskSize[, dst[, labels]]) → dst, labels

C: void **cvDistTransform**(const CvArr* **src**, CvArr* **dst**, int **distanceType**=CV_DIST_L2, int **maskSize**=3, const float* **mask**=NULL, CvArr* **labels**=NULL)

Python: cv.**DistTransform**(src, dst, distanceType=CV_DIST_L2, maskSize=3, mask=None, labels=None) → None

>   **Parameters**

>>   **src** – 8-bit, single-channel (binary) source image.

>>   **dst** – Output image with calculated distances. It is a 32-bit floating-point, single-channel image of the same size as `src`.

>>   **distanceType** – Type of distance. It can be CV_DIST_L1, CV_DIST_L2 , or CV_DIST_C .

>>   **maskSize** – Size of the distance transform mask. It can be 3, 5, or CV_DIST_MASK_PRECISE (the latter option is only supported by the first function). In case of the CV_DIST_L1 or CV_DIST_C distance type, the parameter is forced to 3 because a 3×3 mask gives the same result as 5×5 or any larger aperture.

>>   **labels** – Optional output 2D array of labels (the discrete Voronoi diagram). It has the type CV_32SC1 and the same size as `src`. See the details below.

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 [Felzenszwalb04].

In other cases, the algorithm [Borgefors86] is used. This means that for a 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 5×5 mask). The overall distance is calculated as a sum of these basic distances. Since the distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (denoted as `a`), all the diagonal shifts must have the same cost (denoted as `b`), and all knight's moves must have the same cost (denoted as `c`). For the 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 a relative error (a 5×5 mask gives more accurate results). For a,``b`` , and c , OpenCV uses the values suggested in the original paper:

CV_DIST_C	(3×3)	**a = 1, b = 1**
CV_DIST_L1	(3×3)	a = 1, b = 2
CV_DIST_L2	(3×3)	a=0.955, b=1.3693
CV_DIST_L2	(5×5)	a=1, b=1.4, c=2.1969

Typically, for a fast, coarse distance estimation CV_DIST_L2, a 3×3 mask is used. For a more accurate distance estimation CV_DIST_L2 , a 5×5 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 also identifies 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 the Voronoi diagram for a binary image. Currently, the second variant can use only the approximate distance transform algorithm.

floodFill

Fills a connected component with the given color.

C++: int **floodFill**(InputOutputArray **image**, Point **seed**, Scalar **newVal**, Rect* **rect**=0, Scalar **loDiff**=Scalar(), Scalar **upDiff**=Scalar(), int **flags**=4)

C++: int **floodFill**(InputOutputArray **image**, InputOutputArray **mask**, Point **seed**, Scalar **newVal**, Rect* **rect**=0, Scalar **loDiff**=Scalar(), Scalar **upDiff**=Scalar(), int **flags**=4)

Python: cv2.**floodFill**(image, mask, seedPoint, newVal[, loDiff[, upDiff[, flags]]]) → retval, rect

C: void **cvFloodFill**(CvArr* **image**, CvPoint **seedPoint**, CvScalar **newVal**, CvScalar **loDiff**=cvScalarAll(0), CvScalar **upDiff**=cvScalarAll(0), CvConnectedComp* **comp**=NULL, int **flags**=4, CvArr* **mask**=NULL)

Python: cv.**FloodFill**(image, seedPoint, newVal, loDiff=(0, 0, 0, 0), upDiff=(0, 0, 0, 0), flags=4, mask=None) → comp

> **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 the details below.
> >
> > **mask** – (For the second function only) Operation mask that should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller. The function uses and updates the mask, so you take responsibility of initializing the mask content. Flood-filling cannot 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 does not overlap.
> >
> > ---
> >
> > **Note:** Since the mask is larger than the filled image, a pixel (x, y) in image corresponds to the pixel $(x + 1, y + 1)$ in the mask .
> >
> > ---
> >
> > **seed** – 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** – Optional output parameter set by the function to the minimum bounding rectangle of the repainted domain.
> >
> > **flags** – Operation flags. Lower bits contain a connectivity value, 4 (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 (that is, the range is floating).

– **FLOODFILL_MASK_ONLY** If set, the function does not change the image (`newVal` is ignored), but fills the mask. The flag can be used for the second variant only.

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:

-

$$\texttt{src}(x', y') - \texttt{loDiff} \le \texttt{src}(x, y) \le \texttt{src}(x', y') + \texttt{upDiff}$$

in case of a grayscale image and floating range

-

$$\texttt{src}(\texttt{seed.x}, \texttt{seed.y}) - \texttt{loDiff} \le \texttt{src}(x, y) \le \texttt{src}(\texttt{seed.x}, \texttt{seed.y}) + \texttt{upDiff}$$

in case of a grayscale image and fixed range

-

$$\texttt{src}(x', y')_r - \texttt{loDiff}_r \le \texttt{src}(x, y)_r \le \texttt{src}(x', y')_r + \texttt{upDiff}_r,$$

$$\texttt{src}(x', y')_g - \texttt{loDiff}_g \le \texttt{src}(x, y)_g \le \texttt{src}(x', y')_g + \texttt{upDiff}_g$$

and

$$\texttt{src}(x', y')_b - \texttt{loDiff}_b \le \texttt{src}(x, y)_b \le \texttt{src}(x', y')_b + \texttt{upDiff}_b$$

in case of a color image and floating range

-

$$\texttt{src}(\texttt{seed.x}, \texttt{seed.y})_r - \texttt{loDiff}_r \le \texttt{src}(x, y)_r \le \texttt{src}(\texttt{seed.x}, \texttt{seed.y})_r + \texttt{upDiff}_r,$$

$$\texttt{src}(\texttt{seed.x}, \texttt{seed.y})_g - \texttt{loDiff}_g \le \texttt{src}(x, y)_g \le \texttt{src}(\texttt{seed.x}, \texttt{seed.y})_g + \texttt{upDiff}_g$$

and

$$\texttt{src}(\texttt{seed.x}, \texttt{seed.y})_b - \texttt{loDiff}_b \le \texttt{src}(x, y)_b \le \texttt{src}(\texttt{seed.x}, \texttt{seed.y})_b + \texttt{upDiff}_b$$

in case of a color image and fixed range

where $\texttt{src}(x', y')$ 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 color/brightness of the pixel should be close enough to:

- Color/brightness of one of its neighbors that already belong to the connected component in case of a floating range.

- Color/brightness of the seed point in case of a fixed range.

Use these functions to 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, and so on. Various modes of the function are demonstrated in the `floodfill.cpp` sample.

See Also:

`findContours()`

---

inpaint

Restores the selected region in an image using the region neighborhood.

C++: void **inpaint**(InputArray **src**, InputArray **inpaintMask**, OutputArray **dst**, double **inpaintRadius**, int **flags**)

Python: cv2.**inpaint**(src, inpaintMask, inpaintRange, flags[, dst]) → dst

C: void **cvInpaint**(const CvArr* **src**, const CvArr* **mask**, CvArr* **dst**, double **inpaintRadius**, int **flags**)

Python: cv.**Inpaint**(src, mask, dst, inpaintRadius, flags) → None

> **Parameters**
>
> > **src** – Input 8-bit 1-channel or 3-channel image.
> >
> > **inpaintMask** – Inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.
> >
> > **dst** – Output image with the same size and type as `src`.
> >
> > **inpaintRadius** – Radius of a circlular neighborhood of each point inpainted that is considered by the algorithm.
> >
> > **flags** – Inpainting method that could be one of the following:
> >
> > – **INPAINT_NS** Navier-Stokes based method.
> >
> > – **INPAINT_TELEA** Method by Alexandru Telea [Telea04].

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video. See http://en.wikipedia.org/wiki/Inpainting for more details.

integral

Calculates the integral of an image.

C++: void **integral**(InputArray **image**, OutputArray **sum**, int **sdepth**=-1)

C++: void **integral**(InputArray **image**, OutputArray **sum**, OutputArray **sqsum**, int **sdepth**=-1)

C++: void **integral**(InputArray **image**, OutputArray **sum**, OutputArray **sqsum**, OutputArray **tilted**, int **sdepth**=-1)

Python: cv2.**integral**(src[, sum[, sdepth]]) → sum

Python: cv2.**integral2**(src[, sum[, sqsum[, sdepth]]]) → sum, sqsum

Python: cv2.**integral3**(src[, sum[, sqsum[, tilted[, sdepth]]]]) → sum, sqsum, tilted

C: void **cvIntegral**(const CvArr* **image**, CvArr* **sum**, CvArr* **sqsum**=NULL, CvArr* **tiltedSum**=NULL)

Python: cv.**Integral**(image, sum, sqsum=None, tiltedSum=None) → None

> **Parameters**
>
> > **image** – Source image as $W \times H$, 8-bit or floating-point (32f or 64f).
> >
> > **sum** – Integral image as $(W+1) \times (H+1)$, 32-bit integer or floating-point (32f or 64f).
> >
> > **sqsum** – Integral image for squared pixel values. It is $(W+1) \times (H+1)$, double-precision floating-point (64f) array.

tilted – Integral for the image rotated by 45 degrees. It is $(W+1) \times (H+1)$ array with the same data type as sum.

sdepth – Desired depth of the integral and the tilted integral images, CV_32S, CV_32F, or CV_64F.

The functions calculate one or more integral images for the source image as follows:

$$\text{sum}(X, Y) = \sum_{x<X, y<Y} \text{image}(x, y)$$

$$\text{sqsum}(X, Y) = \sum_{x<X, y<Y} \text{image}(x, y)^2$$

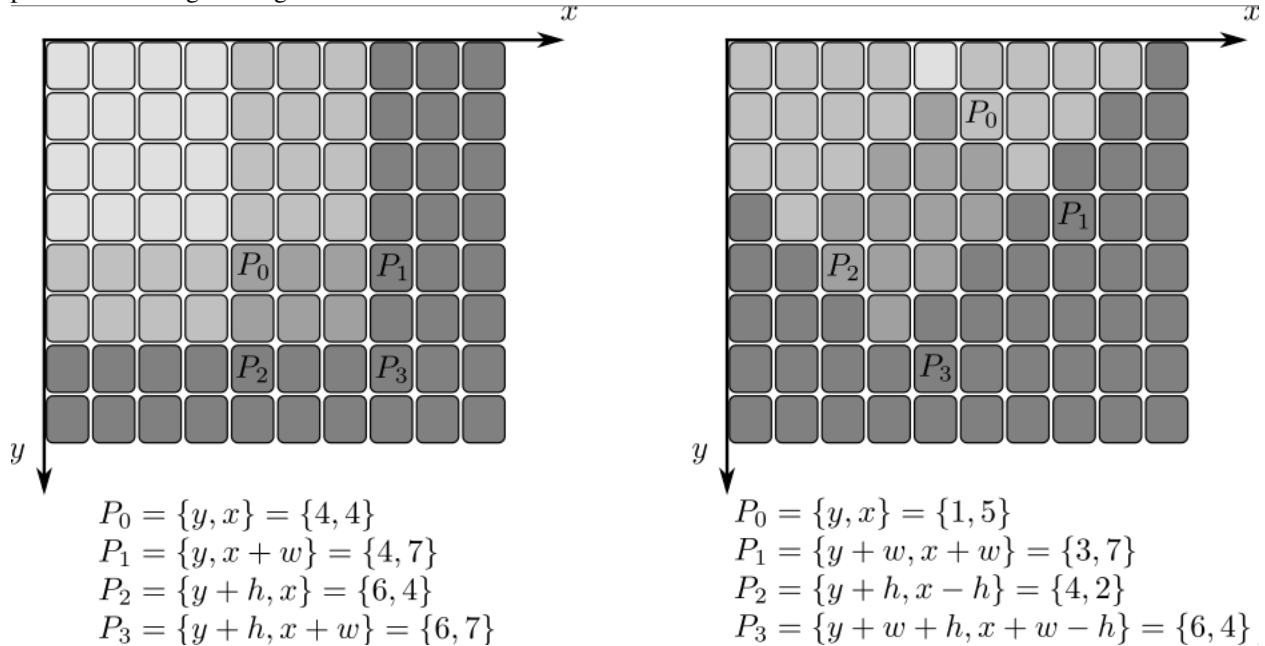$$\text{tilted}(X, Y) = \sum_{y<Y, abs(x-X+1) \leq Y-y-1} \text{image}(x, y)$$

Using these integral images, you can calculate sa um, 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 a variable window size, for example. In case of multi-channel images, sums for each channel are accumulated independently.

As a practical example, the next figure shows the calculation of the integral of a straight rectangle Rect(3,3,3,2) and of a tilted rectangle Rect(5,1,2,3) . The selected pixels in the original image are shown, as well as the relative pixels in the integral images sum and tilted .

$P_0 = \{y, x\} = \{4, 4\}$
$P_1 = \{y, x+w\} = \{4, 7\}$
$P_2 = \{y+h, x\} = \{6, 4\}$
$P_3 = \{y+h, x+w\} = \{6, 7\}$

$P_0 = \{y, x\} = \{1, 5\}$
$P_1 = \{y+w, x+w\} = \{3, 7\}$
$P_2 = \{y+h, x-h\} = \{4, 2\}$
$P_3 = \{y+w+h, x+w-h\} = \{6, 4\}$

threshold

Applies a fixed-level threshold to each array element.

C++: double **threshold**(InputArray **src**, OutputArray **dst**, double **thresh**, double **maxVal**, int **threshold-Type**)

Python: cv2.**threshold**(src, thresh, maxval, type[, dst]) → retval, dst

C: double **cvThreshold**(const CvArr* **src**, CvArr* **dst**, double **threshold**, double **maxValue**, int **threshold-Type**)

Python: cv.**Threshold**(src, dst, threshold, maxValue, thresholdType) → None

> **Parameters**
>> **src** – Source array (single-channel, 8-bit of 32-bit floating point).
>>
>> **dst** – Destination array of the same size and type as src .
>>
>> **thresh** – Threshold value.
>>
>> **maxVal** – Maximum value to use with the THRESH_BINARY and THRESH_BINARY_INV thresholding types.
>>
>> **thresholdType** – Thresholding type (see the details below).

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, that is, filtering out pixels with too small or too large values. There are several types of thresholding supported by the function. They 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 the Otsu's algorithm and uses it instead of the specified thresh . The function returns the computed threshold value. Currently, the Otsu's method is implemented only for 8-bit images.
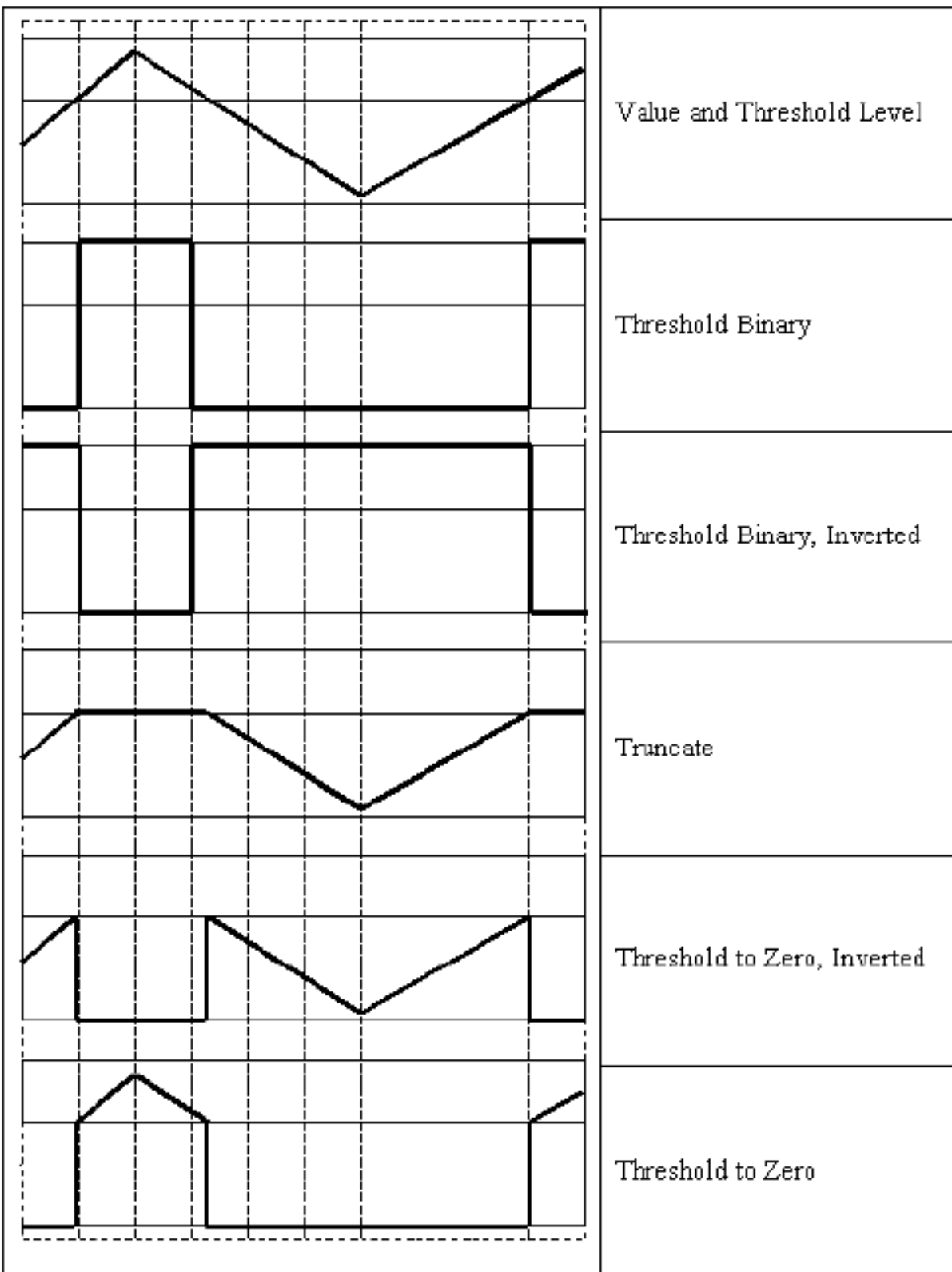
See Also:

adaptiveThreshold(), findContours(), compare(), min(), max()

watershed

Performs a marker-based image segmentation using the watershed algrorithm.

C++: void **watershed**(InputArray **image**, InputOutputArray **markers**)

C: void **cvWatershed**(const CvArr* **image**, CvArr* **markers**)

Python: cv2.**watershed**(image, markers) → None

> **Parameters**
>
> > **image** – Input 8-bit 3-channel image.
> >
> > **markers** – 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 [Meyer92].

Before passing the image to the function, you have to roughly outline the desired regions in the image markers with positive (>0) indices. So, every region is represented as one or more connected components with the pixel values 1, 2, 3, and so on. Such markers can be retrieved from a binary mask using findContours() and drawContours() (see the watershed.cpp demo). The markers are "seeds" of the future image regions. All the other pixels in markers , whose relation to the outlined regions is not known and should be defined by the algorithm, should be set to 0's. In the function output, each pixel in markers is set to a value of the "seed" components or to -1 at boundaries between the regions.

Visual demonstration and usage example of the function can be found in the OpenCV samples directory (see the watershed.cpp demo).

Note: Any two neighbor connected components are not necessarily separated by a watershed boundary (-1's pixels); for example, they can touch each other in the initial marker image passed to the function.

See Also:

findContours()

grabCut

Runs the GrabCut algorithm.

C++: void **grabCut**(InputArray **image**, InputOutputArray **mask**, Rect **rect**, InputOutputArray **bgdModel**, InputOutputArray **fgdModel**, int **iterCount**, int **mode**)

Python: cv2.**grabCut**(img, mask, rect, bgdModel, fgdModel, iterCount[, mode]) → None

> **Parameters**
>
> > **image** – Input 8-bit 3-channel image.
> >
> > **mask** – Input/output 8-bit single-channel mask. The mask is initialized by the function when mode is set to GC_INIT_WITH_RECT. Its elements may have one of following values:
> >
> > – **GC_BGD** defines an obvious background pixels.
> >
> > – **GC_FGD** defines an obvious foreground (object) pixel.

> – **GC_PR_BGD** defines a possible background pixel.

> – **GC_PR_BGD** defines a possible foreground pixel.

> **rect** – ROI containing a segmented object. The pixels outside of the ROI are marked as "obvious background". The parameter is only used when `mode==GC_INIT_WITH_RECT` .

> **bgdModel** – Temporary array for the background model. Do not modify it while you are processing the same image.

> **fgdModel** – Temporary arrays for the foreground model. Do not modify it while you are processing the same image.

> **iterCount** – Number of iterations the algorithm should make before returning the result. Note that the result can be refined with further calls with `mode==GC_INIT_WITH_MASK` or `mode==GC_EVAL` .

> **mode** – Operation mode that could be one of the following:

> – **GC_INIT_WITH_RECT** The function initializes the state and the mask using the provided rectangle. After that it runs `iterCount` iterations of the algorithm.

> – **GC_INIT_WITH_MASK** The function initializes the state using the provided mask. Note that `GC_INIT_WITH_RECT` and `GC_INIT_WITH_MASK` can be combined. Then, all the pixels outside of the ROI are automatically initialized with `GC_BGD` .

> – **GC_EVAL** The value means that the algorithm should just resume.

The function implements the GrabCut image segmentation algorithm. See the sample `grabcut.cpp` to learn how to use the function.

3.4 Histograms

calcHist

Calculates a histogram of a set of arrays.

C++: void **calcHist**(const Mat* **arrays**, int **narrays**, const int* **channels**, InputArray **mask**, OutputArray **hist**, int **dims**, const int* **histSize**, const float** **ranges**, bool **uniform**=true, bool **accumulate**=false)

C++: void **calcHist**(const Mat* **arrays**, int **narrays**, const int* **channels**, InputArray **mask**, SparseMat& **hist**, int **dims**, const int* **histSize**, const float** **ranges**, bool **uniform**=true, bool **accumulate**=false)

Python: cv2.**calcHist**(images, channels, mask, histSize, ranges[, hist[, accumulate]]) → hist

C: void **cvCalcHist**(IplImage** **image**, CvHistogram* **hist**, int **accumulate**=0, const CvArr* **mask**=NULL)

Python: cv.**CalcHist**(image, hist, accumulate=0, mask=None) → None

> **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** – Number of source arrays.

channels – List of the `dims` channels used to compute the histogram. 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`, and so on.

mask – Optional mask. If the matrix is not empty, it must be an 8-bit array of the same size as `arrays[i]` . The non-zero mask elements mark the array elements counted in the histogram.

hist – Output histogram, which is a dense or sparse `dims` -dimensional array.

dims – Histogram dimensionality that must be positive and not greater than `CV_MAX_DIMS` (equal to 32 in the current OpenCV version).

histSize – Array of histogram sizes in each dimension.

ranges – Array of the `dims` arrays of the histogram bin boundaries in each dimension. When the histogram is uniform (`uniform` =true), then for each dimension `i` it is enough to specify the lower (inclusive) boundary L_0 of the 0-th histogram bin and the upper (exclusive) boundary $U_{histSize[i]-1}$ for the last histogram bin `histSize[i]-1` . That is, in case of a 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: $L_0, U_0 = L_1, U_1 = L_2, ..., U_{histSize[i]-2} = L_{histSize[i]-1}, U_{histSize[i]-1}$. The array elements, that are not between L_0 and $U_{histSize[i]-1}$, are not counted in the histogram.

uniform – Flag indicatinfg whether the histogram is uniform or not (see above).

accumulate – Accumulation flag. If it is set, the histogram is not cleared in the beginning when it is allocated. This feature enables you to compute a single histogram from several sets of arrays, or to update the histogram in time.

The functions `calcHist` calculate the histogram of one or more arrays. The elements of a tuple used to increment a histogram bin are taken from the corresponding input arrays at the same location. The sample below shows how to compute a 2D Hue-Saturation histogram for a color image.

```cpp
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat src, hsv;
    if( argc != 2 || !(src=imread(argv[1], 1)).data )
        return -1;

    cvtColor(src, hsv, CV_BGR2HSV);

    // 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 };
    const 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_8UC3);

    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/maxVal);
            rectangle( histImg, Point(h*scale, s*scale),
                       Point( (h+1)*scale - 1, (s+1)*scale - 1),
                       Scalar::all(intensity),
                       CV_FILLED );
        }

    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "H-S Histogram", 1 );
    imshow( "H-S Histogram", histImg );
    waitKey();
}
```

calcBackProject

Calculates the back projection of a histogram.

C++: void **calcBackProject**(const Mat* **arrays**, int **narrays**, const int* **channels**, InputArray **hist**, OutputArray **backProject**, const float** **ranges**, double **scale**=1, bool **uniform**=true)

C++: void **calcBackProject**(const Mat* **arrays**, int **narrays**, const int* **channels**, const SparseMat& **hist**, OutputArray **backProject**, const float** **ranges**, double **scale**=1, bool **uniform**=true)

Python: cv2.**calcBackProject**(images, channels, hist, ranges[, dst[, scale]]) → dst

C: void **cvCalcBackProject**(IplImage** **image**, CvArr* **backProject**, const CvHistogram* **hist**)

Python: cv.**CalcBackProject**(image, backProject, hist) → None

> **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** – Number of source arrays.
> >
> > **channels** – The list of channels 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, and so on.

    hist – Input histogram that can be dense or sparse.

    backProject – Destination back projection aray that is a single-channel array of the same size and depth as arrays[0] .

    ranges – Array of arrays of the histogram bin boundaries in each dimension. See calcHist() .

    scale – Optional scale factor for the output back projection.

    uniform – Flag indicating 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. See how, for example, you can find and track a bright-colored object in a scene:

1. Before tracking, show the object to the camera so that it covers almost the whole frame. Calculate a hue histogram. The histogram may have strong maximums, corresponding to the dominant colors in the object.

2. When tracking, calculate a 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 make sense to suppress pixels with non-sufficient color saturation and too dark or too bright pixels.

3. Find connected components in the resulting picture and choose, for example, the largest component.

This is an approximate algorithm of the CamShift() color object tracker.

See Also:

calcHist()

compareHist

Compares two histograms.

C++: double **compareHist**(InputArray **H1**, InputArray **H2**, int **method**)

C++: double **compareHist**(const SparseMat& **H1**, const SparseMat& **H2**, int **method**)

Python: cv2.**compareHist**(H1, H2, method) → retval

C: double **cvCompareHist**(const CvHistogram* **hist1**, const CvHistogram* **hist2**, int **method**)

Python: cv.**CompareHist**(hist1, hist2, method) → float

    Parameters

        H1 – First compared histogram.

        H2 – Second compared histogram of the same size as H1 .

        method – Comparison method that could be 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 a 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. In such histograms, 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 `EMD()` function.

EMD

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

C++: float **EMD**(InputArray **signature1**, InputArray **signature2**, int **distType**, InputArray **cost**=noArray(), float* **lowerBound**=0, OutputArray **flow**=noArray())

C: float **cvCalcEMD2**(const CvArr* **signature1**, const CvArr* **signature2**, int **distType**, CvDistanceFunction **distFunc**=NULL, const CvArr* **cost**=NULL, CvArr* **flow**=NULL, float* **lowerBound**=NULL, void* **userdata**=NULL)

Python: cv.**CalcEMD2**(signature1, signature2, distType, distFunc=None, cost=None, flow=None, lowerBound=None, userdata=None) $\rightarrow$ float

      Parameters

> **signature1** – First signature, a `size1` × `dims` + 1 floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used.
>
> **signature2** – Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different. In this case an extra "dummy" point is added to either `signature1` or `signature2`.
>
> **distType** – Used metric. `CV_DIST_L1`, `CV_DIST_L2`, and `CV_DIST_C` stand for one of the standard metrics. `CV_DIST_USER` means that a pre-calculated cost matrix `cost` is used.
>
> **distFunc** – Custom distance function supported by the old interface. `CvDistanceFunction` is defined as:
>
> ```
> typedef float (CV_CDECL * CvDistanceFunction)( const float* a,
>                       const float* b, void* userdata );
> ```
>
> where `a` and `b` are point coordinates and `userdata` is the same as the last parameter.
>
> **cost** – User-defined `size1` × `size2` cost matrix. Also, if a cost matrix is used, lower boundary `lowerBound` cannot be calculated because it needs a metric function.
>
> **lowerBound** – Optional input/output parameter: lower boundary of a distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or if the signatures consist of weights only (the signature matrices have a single column). You **must** initialize `*lowerBound`. If the calculated distance between mass centers is greater or equal to `*lowerBound` (it means that the signatures are far enough), the function does not calculate EMD. In any case `*lowerBound` is set to the calculated distance between mass centers on return. Thus, if you want to calculate both distance between mass centers and EMD, `*lowerBound` should be set to 0.
>
> **flow** – Resultant `size1` × `size2` flow matrix: $flow_{i,j}$ is a flow from `i`-th point of `signature1` to `j`-th point of `signature2`.
>
> **userdata** – Optional pointer directly passed to the custom distance function.

The function computes the earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the applications described in [RubnerSept98] is multi-dimensional histogram comparison for image retrieval. EMD is a transportation problem that is solved using some modification of a simplex algorithm, thus the complexity is exponential in the worst case, though, on average it is much faster. In the case of a real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

equalizeHist

Equalizes the histogram of a grayscale image.

C++: void **equalizeHist**(InputArray **src**, OutputArray **dst**)

Python: cv2.**equalizeHist**(src[, dst]) → dst

C: void **cvEqualizeHist**(const CvArr* **src**, CvArr* **dst**)

> **Parameters**
>
> > **src** – Source 8-bit single channel image.
> >
> > **dst** – Destination image of the same size and type as `src`.

The function equalizes the histogram of the input image using the following algorithm:

1. Calculate the histogram H for `src`.

2. Normalize the histogram so that the sum of histogram bins is 255.

3. Compute the integral of the histogram:

$$H'_i = \sum_{0 \leq j < i} H(j)$$

4. Transform the image using H' as a look-up table: $dst(x, y) = H'(src(x, y))$

The algorithm normalizes the brightness and increases the contrast of the image.

Extra Histogram Functions (C API)

The rest of the section describes additional C functions operating on `CvHistogram`.

CalcBackProjectPatch

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

C: void **cvCalcBackProjectPatch**(IplImage** **images**, CvArr* **dst**, CvSize **patch_size**, CvHistogram* **hist**,
int **method**, double **factor**)

Python: cv.**CalcBackProjectPatch**(images, dst, patchSize, hist, method, factor) → None

> **Parameters**
>
>> **images** – Source images (though, you may pass CvMat** as well).
>>
>> **dst** – Destination image.
>>
>> **patch_size** – Size of the patch slid though the source image.
>>
>> **hist** – Histogram.
>>
>> **method** – Comparison method passed to `CompareHist()` (see the function description).
>>
>> **factor** – Normalization factor for histograms that affects the normalization scale of the destination image. Pass 1 if not sure.

The function calculates the back projection by comparing histograms of the source image patches with the given histogram. The function is similar to `matchTemplate()`, but instead of comparing the raster patch with all its possible positions within the search window, the function `CalcBackProjectPatch` compares histograms. See the algorithm diagram below:
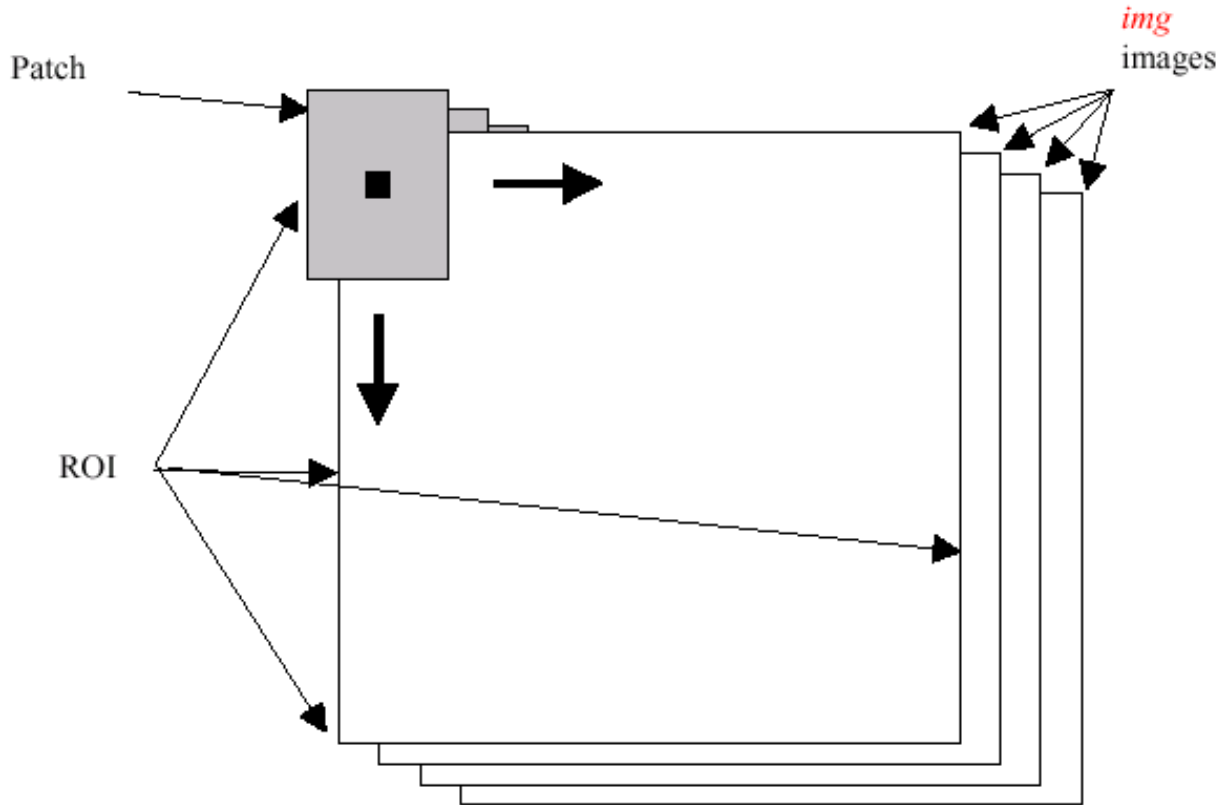
CalcProbDensity

Divides one histogram by another.

C: void **cvCalcProbDensity**(const CvHistogram* **hist1**, const CvHistogram* **hist2**, CvHistogram* **dsthist**, double **scale**=255)

Python: cv.**CalcProbDensity**(hist1, hist2, dsthist, scale=255) → None

> **Parameters**
>
> > **hist1** – First histogram (the divisor).
> >
> > **hist2** – Second histogram.
> >
> > **dsthist** – Destination histogram.
> >
> > **scale** – Scale factor for the destination histogram.

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

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

ClearHist

Clears the histogram.

C: void **cvClearHist**(CvHistogram* **hist**)

Python: cv.**ClearHist**(hist) $\rightarrow$ None

> **Parameters**
>
>> **hist** – Histogram.

The function sets all of the histogram bins to 0 in case of a dense histogram and removes all histogram bins in case of a sparse array.

CopyHist

Copies a histogram.

C: void **cvCopyHist**(const CvHistogram* **src**, CvHistogram** **dst**)

> **Parameters**
>
>> **src** – Source histogram.
>>
>> **dst** – Pointer to the destination histogram.

The function makes a copy of the histogram. If the second histogram pointer *dst is NULL, a new histogram of the same size as src is created. Otherwise, both histograms must have equal types and sizes. Then the function copies the bin values of the source histogram to the destination histogram and sets the same bin value ranges as in src.

CreateHist

Creates a histogram.

C: CvHistogram* **cvCreateHist**(int **dims**, int* **sizes**, int **type**, float** **ranges**=NULL, int **uniform**=1)

Python: cv.**CreateHist**(dims, type, ranges, uniform=1) $\rightarrow$ hist

> **Parameters**
>
>> **dims** – Number of histogram dimensions.
>>
>> **sizes** – Array of the histogram dimension sizes.
>>
>> **type** – Histogram representation format. CV_HIST_ARRAY means that the histogram data is represented as a multi-dimensional dense array CvMatND. CV_HIST_SPARSE means that histogram data is represented as a multi-dimensional sparse array CvSparseMat.
>>
>> **ranges** – Array of ranges for the histogram bins. Its meaning depends on the uniform parameter value. The ranges are used when the histogram is calculated or backprojected to determine which histogram bin corresponds to which value/tuple of values from the input image(s).
>>
>> **uniform** – Uniformity flag. If not zero, the histogram has evenly spaced bins and for every $0 <= i < cDims$ ranges[i] is an array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is then split into dims[i] equal parts to determine the i-th input tuple value ranges for every histogram bin. And if uniform=0 , then the i-th element of the ranges array contains dims[i]+1 elements: $lower_0, upper_0, lower_1, upper_1 = lower_2, ... upper_{dims[i]-1}$ where $lower_j$ and $upper_j$ are lower and upper boundaries of the i-th input tuple value for the j-th bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin are not counted by CalcHist() and filled with 0 by CalcBackProject().

The function creates a histogram of the specified size and returns a pointer to the created histogram. If the array ranges is 0, the histogram bin ranges must be specified later via the function SetHistBinRanges(). Though CalcHist()

and `CalcBackProject()` may process 8-bit images without setting bin ranges, they assume they are equally spaced in 0 to 255 bins.

GetHistValue*D

Returns a pointer to the histogram bin.

C: float **cvGetHistValue_1D**(hist **None**, idx0 **None**)

C: float **cvGetHistValue_2D**(hist **None**, idx0 **None**, idx1 **None**)

C: float **cvGetHistValue_3D**(hist **None**, idx0 **None**, idx1 **None**, idx2 **None**)

C: float **cvGetHistValue_nD**(hist **None**, idx **None**)

>   **Parameters**

>> **hist** – Histogram.

>> **idx0** – 0-th index.

>> **idx1** – 1-st index.

>> **idx2** – 2-nd index.

>> **idx** – Array of indices.

```
#define cvGetHistValue_1D( hist, idx0 )
    ((float*)(cvPtr1D( (hist)->bins, (idx0), 0 ))
#define cvGetHistValue_2D( hist, idx0, idx1 )
    ((float*)(cvPtr2D( (hist)->bins, (idx0), (idx1), 0 )))
#define cvGetHistValue_3D( hist, idx0, idx1, idx2 )
    ((float*)(cvPtr3D( (hist)->bins, (idx0), (idx1), (idx2), 0 )))
#define cvGetHistValue_nD( hist, idx )
    ((float*)(cvPtrND( (hist)->bins, (idx), 0 )))
```

The macros `GetHistValue` return a pointer to the specified bin of the 1D, 2D, 3D, or N-D histogram. In case of a sparse histogram, the function creates a new bin and sets it to 0, unless it exists already.

GetMinMaxHistValue

Finds the minimum and maximum histogram bins.

C: void **cvGetMinMaxHistValue**(const CvHistogram* **hist**, float* **min_value**, float* **max_value**, int* **min_idx**=NULL, int* **max_idx**=NULL)

Python: cv.**GetMinMaxHistValue**(hist)-> (minValue, maxValue, minIdx, maxIdx)

>   **Parameters**

>> **hist** – Histogram.

>> **min_value** – Pointer to the minimum value of the histogram.

>> **max_value** – Pointer to the maximum value of the histogram.

>> **min_idx** – Pointer to the array of coordinates for the minimum.

>> **max_idx** – Pointer to the array of coordinates for the maximum.

The function finds the minimum and maximum histogram bins and their positions. All of output arguments are optional. Among several extremas with the same value the ones with the minimum index (in the lexicographical order) are returned. In case of several maximums or minimums, the earliest in the lexicographical order (extrema locations) is returned.

MakeHistHeaderForArray

Makes a histogram out of an array.

C: CvHistogram* **cvMakeHistHeaderForArray**(int **dims**, int* **sizes**, CvHistogram* **hist**, float* **data**, float** **ranges**=NULL, int **uniform**=1)

>   **Parameters**
>
>>   **dims** – Number of the histogram dimensions.
>>
>>   **sizes** – Array of the histogram dimension sizes.
>>
>>   **hist** – Histogram header initialized by the function.
>>
>>   **data** – Array used to store histogram bins.
>>
>>   **ranges** – Histogram bin ranges. See `CreateHist()` for details.
>>
>>   **uniform** – Uniformity flag. See `CreateHist()` for details.

The function initializes the histogram, whose header and bins are allocated by the user. `ReleaseHist()` does not need to be called afterwards. Only dense histograms can be initialized this way. The function returns `hist`.

NormalizeHist

Normalizes the histogram.

C: void **cvNormalizeHist**(CvHistogram* **hist**, double **factor**)

Python: cv.**NormalizeHist**(hist, factor) → None

>   **Parameters**
>
>>   **hist** – Pointer to the histogram.
>>
>>   **factor** – Normalization factor.

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

QueryHistValue*D

Queries the value of the histogram bin.

C: float **QueryHistValue_1D**(CvHistogram **hist**, int **idx0**)

C: float **QueryHistValue_2D**(CvHistogram **hist**, int **idx0**, int **idx1**)

C: float **QueryHistValue_3D**(CvHistogram **hist**, int **idx0**, int **idx1**, int **idx2**)

C: float **QueryHistValue_nD**(CvHistogram **hist**, const int* **idx**)

Python: cv.**QueryHistValue_1D**(hist, idx0) → float

Python: cv.**QueryHistValue_2D**(hist, idx0, idx1) → float

Python: cv.**QueryHistValue_3D**(hist, idx0, idx1, idx2) → float

Python: cv.`QueryHistValueND`(hist, idx) → float

> **Parameters**
>
> > **hist** – Histogram.
> >
> > **idx0** – 0-th index.
> >
> > **idx1** – 1-st index.
> >
> > **idx2** – 2-nd index.
> >
> > **idx** – Array of indices.

The macros return the value of the specified bin of the 1D, 2D, 3D, or N-D histogram. In case of a sparse histogram, the function returns 0. If the bin is not present in the histogram, no new bin is created.

ReleaseHist

Releases the histogram.

C: void **cvReleaseHist**(CvHistogram** **hist**)

> **Parameters**
>
> > **hist** – Double pointer to the released histogram.

The function releases the histogram (header and the data). The pointer to the histogram is cleared by the function. If `*hist` pointer is already `NULL`, the function does nothing.

SetHistBinRanges

Sets the bounds of the histogram bins.

C: void **cvSetHistBinRanges**(CvHistogram* **hist**, float** **ranges**, int **uniform**=1)

> **Parameters**
>
> > **hist** – Histogram.
> >
> > **ranges** – Array of bin ranges arrays. See `CreateHist()` for details.
> >
> > **uniform** – Uniformity flag. See `CreateHist()` for details.

This is a standalone function for setting bin ranges in the histogram. For a more detailed description of the parameters `ranges` and `uniform`, see the `CalcHist()` function that can initialize the ranges as well. Ranges for the histogram bins must be set before the histogram is calculated or the backproject of the histogram is calculated.

ThreshHist

Thresholds the histogram.

C: void **cvThreshHist**(CvHistogram* **hist**, double **threshold**)

Python: cv.`ThreshHist`(hist, threshold) → None

> **Parameters**
>
> > **hist** – Pointer to the histogram.
> >
> > **threshold** – Threshold level.

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

CalcPGH

Calculates a pair-wise geometrical histogram for a contour.

C: void **cvCalcPGH**(const CvSeq* **contour**, CvHistogram* **hist**)

Python: cv.**CalcPGH**(contour, hist) → None

> **Parameters**
>
> > **contour** – Input contour. Currently, only integer point coordinates are allowed.
> >
> > **hist** – Calculated histogram. It must be two-dimensional.

The function calculates a 2D pair-wise geometrical histogram (PGH), described in [Iivarinen97] for the contour. The algorithm considers every pair of contour edges. The angle between the edges and the minimum/maximum distances are determined for every pair. To do this, each of the edges in turn is taken as the base, while the function loops through all the other edges. When the base edge and any other edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented (that is, the histogram is transposed relatively to the definition in the original paper). The histogram can be used for contour matching.

3.5 Structural Analysis and Shape Descriptors

moments

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

C++: Moments **moments**(InputArray **array**, bool **binaryImage**=false)

Python: cv2.**moments**(array[, binaryImage]) → retval

C: void **cvMoments**(const CvArr* **array**, CvMoments* **moments**, int **binary**=0)

Python: cv.**Moments**(array, binary=0) → moments

> **Parameters**
>
> > **array** – 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** – If it is true, all non-zero image pixels are treated as 1's. The parameter is used for images only.
> >
> > **moments** – Output moments.

The function computes moments, up to the 3rd order, of a vector shape or a rasterized shape. The results are returned in the structure `Moments` defined as:

```
class Moments
{
public:
    Moments();
    Moments(double m00, double m10, double m01, double m20, double m11,
            double m02, double m30, double m21, double m12, double m03 );
    Moments( const CvMoments& moments );
    operator CvMoments() const;
```

In case of a raster image, the spatial moments `Moments::m`$_{ji}$ are computed as:

$$m_{ji} = \sum_{x,y} \left(\texttt{array}(x,y) \cdot x^j \cdot y^i \right)$$

The central moments `Moments::mu`$_{ji}$ are computed as:

$$mu_{ji} = \sum_{x,y} \left(\texttt{array}(x,y) \cdot (x - \bar{x})^j \cdot (y - \bar{y})^i \right)$$

where $(\bar{x}, \bar{y})$ is the mass center:

$$\bar{x} = \frac{m_{10}}{m_{00}}, \quad \bar{y} = \frac{m_{01}}{m_{00}}$$

The normalized central moments `Moments::nu`$_{ij}$ are computed as:

$$nu_{ji} = \frac{mu_{ji}}{m_{00}^{(i+j)/2+1}}.$$

Note: $mu_{00} = m_{00}, nu_{00} = 1 \; nu_{10} = mu_{10} = mu_{01} = mu_{10} = 0$, hence the values are not stored.

The moments of a contour are defined in the same way but computed using the Green's formula (see http://en.wikipedia.org/wiki/Green_theorem). So, due to a limited raster resolution, the moments computed for a contour are slightly different from the moments computed for the same rasterized contour.

See Also:

`contourArea()`, `arcLength()`

HuMoments

Calculates seven Hu invariants.

C++: void **HuMoments**(const Moments& **moments**, double* **hu**)

Python: cv2.**HuMoments**(m) $\rightarrow$ hu

C: void **cvGetHuMoments**(const CvMoments* **moments**, CvHuMoments* **hu**)

Python: cv.**GetHuMoments**(moments) $\rightarrow$ hu

> **Parameters**
>
>> **moments** – Input moments computed with `moments()` .
>>
>> **hu** – Output Hu invariants.

The function calculates seven Hu invariants (introduced in [Hu62]; see also http://en.wikipedia.org/wiki/Image_moment) defined as:

$$hu[0] = \eta_{20} + \eta_{02}$$
$$hu[1] = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$
$$hu[2] = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$
$$hu[3] = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$
$$hu[4] = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$
$$hu[5] = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$
$$hu[6] = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

---

where η_{ji} stands for `Moments::nu`$_{ji}$.

These values are proved to be invariants to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. This invariance is proved with the assumption of infinite image resolution. In case of raster images, the computed Hu invariants for the original and transformed images are a bit different.

See Also:

`matchShapes()`

findContours

Finds contours in a binary image.

C++: void **findContours**(InputOutputArray **image**, OutputArrayOfArrays **contours**, OutputArray **hierar-chy**, int **mode**, int **method**, Point **offset**=Point())

C++: void **findContours**(InputOutputArray **image**, OutputArrayOfArrays **contours**, int **mode**, int **method**, Point **offset**=Point())

C: int **cvFindContours**(CvArr* **image**, CvMemStorage* **storage**, CvSeq** **firstContour**, int **headerSize**=sizeof(CvContour), int **mode**=CV_RETR_LIST, int **method**=CV_CHAIN_APPROX_SIMPLE, CvPoint **offset**=cvPoint(0, 0))

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

> **Parameters**
>
>> **image** – Source, an 8-bit single-channel image. Non-zero pixels are treated as 1's. Zero pixels remain 0's, so the image is treated as `binary` . You can use `compare()` , `inRange()` , `threshold()` , `adaptiveThreshold()` , `Canny()` , and others to create a binary image out of a grayscale or color one. The function modifies the `image` while extracting the contours.
>>
>> **contours** – Detected contours. Each contour is stored as a vector of points.
>>
>> **hiararchy** – Optional output vector containing information about the image topology. It has as many elements as the number of contours. For each contour `contours[i]` , the elements `hierarchy[i][0]` , `hiearchy[i][1]` , `hierarchy[i][2]` , and `hierarchy[i][3]` are set to 0-based indices in `contours` of the next and previous contours at the same hierarchical level: the first child contour and the parent contour, respectively. If for a contour `i` there are no next, previous, parent, or nested contours, the corresponding elements of `hierarchy[i]` will be negative.
>>
>> **mode** – Contour retrieval mode.
>>
>>> – **CV_RETR_EXTERNAL** retrieves only the extreme outer contours. It sets `hierarchy[i][2]=hierarchy[i][3]=-1` for all the contours.
>>>
>>> – **CV_RETR_LIST** retrieves all of the contours without establishing any hierarchical relationships.
>>>
>>> – **CV_RETR_CCOMP** retrieves all of the contours and organizes them into a two-level hierarchy. At the top level, there are external boundaries of the components. At the second level, there are boundaries of the holes. If there is another contour inside a hole of a connected component, it is still put at 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 shown in the OpenCV `contours.c` demo.
>>
>> **method** – Contour approximation method.

- **CV_CHAIN_APPROX_NONE** stores absolutely all the contour points. That is, any 2 subsequent points (x1,y1) and (x2,y2) of the contour will be either horizontal, vertical or diagonal neighbors, that is, max(abs(x1-x2),abs(y2-y1))==1.

- **CV_CHAIN_APPROX_SIMPLE** compresses horizontal, vertical, and diagonal segments and leaves only their end points. For example, an up-right rectangular contour is encoded with 4 points.

- **CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS** applies one of the flavors of the Teh-Chin chain approximation algorithm. See [TehChin89] for details.

offset – Optional offset by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context.

The function retrieves contours from the binary image using the algorithm [Suzuki85]. The contours are a useful tool for shape analysis and object detection and recognition. See `squares.c` in the OpenCV sample directory.

Note: Source `image` is modified by this function.

---

drawContours

Draws contours outlines or filled contours.

C++: void **drawContours**(InputOutputArray **image**, InputArrayOfArrays **contours**, int **contourIdx**, const Scalar& **color**, int **thickness**=1, int **lineType**=8, InputArray **hierarchy**=noArray(), int **maxLevel**=INT_MAX, Point **offset**=Point())

Python: cv2.**drawContours**(image, contours, contourIdx, color[, thickness[, lineType[, hierarchy[, maxLevel[, offset]]]]]) → None

C: void **cvDrawContours**(CvArr* **img**, CvSeq* **contour**, CvScalar **externalColor**, CvScalar **holeColor**, int **maxLevel**, int **thickness**=1, int **lineType**=8)

Python: cv.**DrawContours**(img, contour, externalColor, holeColor, maxLevel, thickness=1, lineType=8, offset=(0, 0)) → None

> **Parameters**
>
> > **image** – Destination image.
> >
> > **contours** – All the input contours. Each contour is stored as a point vector.
> >
> > **contourIdx** – Parameter indicating a contour to draw. If it is negative, all the contours are drawn.
> >
> > **color** – Color of the contours.
> >
> > **thickness** – Thickness of lines the contours are drawn with. If it is negative (for example, `thickness=CV_FILLED`), the contour interiors are drawn.
> >
> > **lineType** – Line connectivity. See `line()` for details.
> >
> > **hierarchy** – 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 it is 0, only the specified contour is drawn. If it is 1, the function draws the contour(s) and all the nested contours. If it is 2, the function draws the contours, all the nested contours, all the nested-to-nested contours, and so on. This parameter is only taken into account when there is `hierarchy` available.

---

offset – Optional contour shift parameter. Shift all the drawn contours by the specified $\text{offset} = (dx, dy)$.

The function draws contour outlines in the image if $\text{thickness} \geq 0$ or fills the area bounded by the contours if $\text{thickness} < 0$. The example below shows how to retrieve connected components from the binary image and label them:

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

using namespace cv;

int main( int argc, char** argv )
{
    Mat src;
    // the first command-line parameter must be a filename of the binary
    // (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, 8, hierarchy );
    }

    namedWindow( "Components", 1 );
    imshow( "Components", dst );
    waitKey(0);
}
```

approxPolyDP

Approximates a polygonal curve(s) with the specified precision.

C++: void **approxPolyDP**(InputArray **curve**, OutputArray **approxCurve**, double **epsilon**, bool **closed**)

Python: cv2.**approxPolyDP**(curve, epsilon, closed[, approxCurve]) → approxCurve

C: CvSeq* **cvApproxPoly**(const void* **curve**, int **headerSize**, CvMemStorage* **storage**, int **method**, double **epsilon**, int **recursive**=0)

Parameters

> > **curve** – Input vector of a 2D point stored in:
>
> > – `std::vector` or `Mat` (C++ interface)
> >
> > – `Nx2` numpy array (Python interface)
> >
> > – `CvSeq` or `` `CvMat` (C interface)
>
> > **approxCurve** – Result of the approximation. The type should match the type of the input curve. In case of C interface the approximated curve is stored in the memory storage and pointer to it is returned.
>
> > **epsilon** – Parameter specifying the approximation accuracy. This is the maximum distance between the original curve and its approximation.
>
> > **closed** – If true, the approximated curve is closed (its first and last vertices are connected). Otherwise, it is not closed.
>
> > **headerSize** – Header size of the approximated curve. Normally, `sizeof(CvContour)` is used.
>
> > **storage** – Memory storage where the approximated curve is stored.
>
> > **method** – Contour approximation algorithm. Only `CV_POLY_APPROX_DP` is supported.
>
> > **recursive** – Recursion flag. If it is non-zero and `curve` is `CvSeq*`, the function `cvApproxPoly` approximates all the contours accessible from `curve` by `h_next` and `v_next` links.

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 uses the Douglas-Peucker algorithm http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm

See http://code.ros.org/svn/opencv/trunk/opencv/samples/cpp/contours.cpp for the function usage model.

ApproxChains

Approximates Freeman chain(s) with a polygonal curve.

C: CvSeq* **cvApproxChains**(CvSeq* **chain**, CvMemStorage* **storage**, int **method**=CV_CHAIN_APPROX_SIMPLE, double **parameter**=0, int **minimalPerimeter**=0, int **recursive**=0)

Python: cv.**ApproxChains**(chain, storage, method=CV_CHAIN_APPROX_SIMPLE, parameter=0, minimalPerimeter=0, recursive=0) → contours

> **Parameters**
>
> > **chain** – Pointer to the approximated Freeman chain that can refer to other chains.
> >
> > **storage** – Storage location for the resulting polylines.
> >
> > **method** – Approximation method (see the description of the function `FindContours()`).
> >
> > **parameter** – Method parameter (not used now).
> >
> > **minimalPerimeter** – Approximates only those contours whose perimeters are not less than `minimal_perimeter` . Other chains are removed from the resulting structure.
> >
> > **recursive** – Recursion flag. If it is non-zero, the function approximates all chains that can be obtained from `chain` by using the `h_next` or `v_next` links. Otherwise, the single input chain is approximated.

This is a standalone contour approximation routine, not represented in the new interface. When `FindContours()` retrieves contours as Freeman chains, it calls the function to get approximated contours, represented as polygons.

---

arcLength

Calculates a contour perimeter or a curve length.

C++: double **arcLength**(InputArray **curve**, bool **closed**)

Python: cv2.**arcLength**(curve, closed) → retval

C: double **cvArcLength**(const void* **curve**, CvSlice **slice**=CV_WHOLE_SEQ, int **isClosed**=-1)

Python: cv.**ArcLength**(curve, slice=CV_WHOLE_SEQ, isClosed=-1) → double

> **Parameters**

>> **curve** – Input vector of 2D points, stored in `std::vector` or `Mat`.

>> **closed** – Flag indicating whether the curve is closed or not.

The function computes a curve length or a closed contour perimeter.

boundingRect

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

C++: Rect **boundingRect**(InputArray **points**)

Python: cv2.**boundingRect**(points) → retval

C: CvRect **cvBoundingRect**(CvArr* **points**, int **update**=0)

Python: cv.**BoundingRect**(points, update=0) → CvRect

> **Parameters**

>> **points** – Input 2D point set, stored in `std::vector` or `Mat`.

The function calculates and returns the minimal up-right bounding rectangle for the specified point set.

contourArea

Calculates a contour area.

C++: double **contourArea**(InputArray **contour**, bool **oriented**=false)

Python: cv2.**contourArea**(contour[, oriented]) → retval

C: double **cvContourArea**(const CvArr* **contour**, CvSlice **slice**=CV_WHOLE_SEQ)

Python: cv.**ContourArea**(contour, slice=CV_WHOLE_SEQ) → double

> **Parameters**

>> **contour** – Input vector of 2D points (contour vertices), stored in `std::vector` or `Mat`.

>> **orientation** – Oriented area flag. If it is true, the function returns a signed area value, depending on the contour orientation (clockwise or counter-clockwise). Using this feature you can determine orientation of a contour by taking the sign of an area. By default, the parameter is `false`, which means that the absolute value is returned.

The function computes a 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.

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

Finds the convex hull of a point set.

C++: void **convexHull**(InputArray **points**, OutputArray **hull**, bool **clockwise**=false, bool **returnPoints**=true)

Python: cv2.**convexHull**(points[, hull[, returnPoints[, clockwise]]]) → hull

C: CvSeq* **cvConvexHull2**(const CvArr* **input**, void* **storage**=NULL, int **orientation**=CV_CLOCKWISE, int **returnPoints**=0)

Python: cv.**ConvexHull2**(points, storage, orientation=CV_CLOCKWISE, returnPoints=0) → convexHull

> **Parameters**
>
> > **points** – Input 2D point set, stored in std::vector or Mat.
> >
> > **hull** – Output convex hull. It is either an integer vector of indices or vector of points. In the first case, the hull elements are 0-based indices of the convex hull points in the original array (since the set of convex hull points is a subset of the original point set). In the second case, hull elements aree the convex hull points themselves.
> >
> > **storage** – Output memory storage in the old API (cvConvexHull2 returns a sequence containing the convex hull points or their indices).
> >
> > **clockwise** – Orientation flag. If it is true, the output convex hull is oriented clockwise. Otherwise, it is oriented counter-clockwise. The usual screen coordinate system is assumed so that the origin is at the top-left corner, x axis is oriented to the right, and y axis is oriented downwards.
> >
> > **orientation** – Convex hull orientation parameter in the old API, CV_CLOCKWISE or CV_COUNTERCLOCKWISE.
> >
> > **returnPoints** – Operation flag. In case of a matrix, when the flag is true, the function returns convex hull points. Otherwise, it returns indices of the convex hull points. When the output array is std::vector, the flag is ignored, and the output depends on the type of the vector: std::vector<int> implies returnPoints=true, std::vector<Point> implies returnPoints=false.

The functions find the convex hull of a 2D point set using the Sklansky's algorithm [Sklansky82] that has *O(N logN)* complexity in the current implementation. See the OpenCV sample convexhull.cpp that demonstrates the usage of different function variants.

ConvexityDefects

Finds the convexity defects of a contour.

C: CvSeq* **cvConvexityDefects**(const CvArr* **contour**, const CvArr* **convexhull**, CvMemStorage* **storage**=NULL)

Python: cv.**ConvexityDefects**(contour, convexhull, storage) → convexityDefects

>   **Parameters**

>   >   **contour** – Input contour.

>   >   **convexhull** – Convex hull obtained using ConvexHull2() that should contain pointers or indices to the contour points, not the hull points themselves (the returnPoints parameter in ConvexHull2() should be zero).

>   >   **storage** – Container for the output sequence of convexity defects. If it is NULL, the contour or hull (in that order) storage is used.

The function finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect structures, where CvConvexityDetect is defined as:

```
struct CvConvexityDefect
{
    CvPoint* start; // point of the contour where the defect begins
    CvPoint* end; // point of the contour where the defect ends
    CvPoint* depth_point; // the farthest from the convex hull point within the defect
    float depth; // distance between the farthest point and the convex hull
};
```

The figure below displays convexity defects of a hand contour:
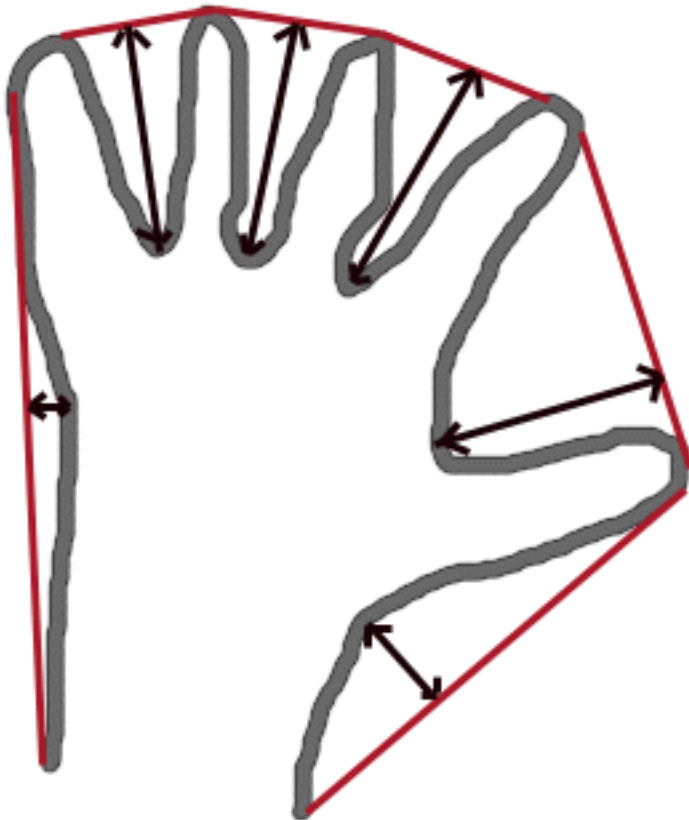
fitEllipse

Fits an ellipse around a set of 2D points.

C++: RotatedRect **fitEllipse**(InputArray **points**)

Python: cv2.**fitEllipse**(points) → retval

C: CvBox2D **cvFitEllipse2**(const CvArr* **points**)

Python: cv.**FitEllipse2**(points) → Box2D

> **Parameters**

> > **points** – Input 2D point set, stored in:

> > > – std::vector<> or Mat (C++ interface)

> > > – CvSeq* or CvMat* (C interface)

> > > – Nx2 numpy array (Python interface)

The function calculates the ellipse that fits (in a least-squares sense) a set of 2D points best of all. It returns the rotated rectangle in which the ellipse is inscribed. The algorithm [Fitzgibbon95] is used.

fitLine

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

C++: void **fitLine**(InputArray **points**, OutputArray **line**, int **distType**, double **param**, double **reps**, double **aeps**)

Python: cv2.**fitLine**(points, distType, param, reps, aeps) → line

C: void **cvFitLine**(const CvArr* **points**, int **distType**, double **param**, double **reps**, double **aeps**, float* **line**)

Python: cv.**FitLine**(points, distType, param, reps, aeps) → line

> **Parameters**

> > **points** – Input vector of 2D or 3D points, stored in std::vector<> or Mat.

> > **line** – Output line parameters. In case of 2D fitting, it should be a vector of 4 elements (like Vec4f) - (vx, vy, x0, y0), where (vx, vy) is a normalized vector collinear to the line and (x0, y0) is a point on the line. In case of 3D fitting, it should be a vector of 6 elements (like Vec6f) - (vx, vy, vz, x0, y0, z0), where (vx, vy, vz) is a normalized vector collinear to the line and (x0, y0, z0) is a point on the line.

> > **distType** – Distance used by the M-estimator (see the discussion below).

> > **param** – Numerical parameter (C) for some types of distances. If it is 0, an optimal value is chosen.

> > **reps** – Sufficient accuracy for the radius (distance between the coordinate origin and the line).

> > **aeps** – Sufficient accuracy for the angle. 0.01 would be a good default value for reps and aeps.

The function fitLine fits a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is a distance between the ith point, the line and $\rho(r)$ is a distance function, one of the following:

- 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

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

- distType=CV_DIST_FAIR

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

- distType=CV_DIST_WELSCH

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

- distType=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

The algorithm is based on the M-estimator (http://en.wikipedia.org/wiki/M-estimator) technique that iteratively fits the line using the weighted least-squares algorithm. After each iteration the weights w_i are adjusted to be inversely proportional to $\rho(r_i)$.

isContourConvex

Tests a contour convexity.

C++: bool **isContourConvex**(InputArray **contour**)

Python: cv2.**isContourConvex**(contour) $\rightarrow$ retval

C: int **cvCheckContourConvexity**(const CvArr* **contour**)

Python: cv.**CheckContourConvexity**(contour) $\rightarrow$ int

> **Parameters**

>> **contour** – Input vector of 2D points, stored in:

>>> – std::vector<> or Mat (C++ interface)

>>> – CvSeq* or CvMat* (C interface)

>>> – Nx2 numpy array (Python interface)

The function tests whether the input contour is convex or not. The contour must be simple, that is, without self-intersections. Otherwise, the function output is undefined.

minAreaRect

Finds a rotated rectangle of the minimum area enclosing the input 2D point set.

C++: RotatedRect **minAreaRect**(InputArray **points**)

Python: cv2.**minAreaRect**(points) → retval

C: CvBox2D **cvMinAreaRect2**(const CvArr* **points**, CvMemStorage* **storage**=NULL)

Python: cv.**MinAreaRect2**(points, storage=None) → CvBox2D

> **Parameters**
>
> > **points** – Input vector of 2D points, stored in:
> >
> > – `std::vector<>` or `Mat` (C++ interface)
> >
> > – `CvSeq*` or `CvMat*` (C interface)
> >
> > – Nx2 numpy array (Python interface)

The function calculates and returns the minimum-area bounding rectangle (possibly rotated) for a specified point set. See the OpenCV sample `minarea.cpp` .

minEnclosingCircle

Finds a circle of the minimum area enclosing a 2D point set.

C++: void **minEnclosingCircle**(InputArray **points**, Point2f& **center**, float& **radius**)

Python: cv2.**minEnclosingCircle**(points, center, radius) → None

C: int **cvMinEnclosingCircle**(const CvArr* **points**, CvPoint2D32f* **center**, float* **radius**)

Python: cv.**MinEnclosingCircle**(points)-> (int, center, radius)

> **Parameters**
>
> > **points** – Input vector of 2D points, stored in:
> >
> > – `std::vector<>` or `Mat` (C++ interface)
> >
> > – `CvSeq*` or `CvMat*` (C interface)
> >
> > – Nx2 numpy array (Python interface)
> >
> > **center** – Output center of the circle.
> >
> > **radius** – Output radius of the circle.

The function finds the minimal enclosing circle of a 2D point set using an iterative algorithm. See the OpenCV sample `minarea.cpp` .

matchShapes

Compares two shapes.

C++: double **matchShapes**(InputArray **object1**, InputArray **object2**, int **method**, double **parameter**=0)

Python: cv2.**matchShapes**(contour1, contour2, method, parameter) → retval

C: double **cvMatchShapes**(const void* **object1**, const void* **object2**, int **method**, double **parameter**=0)

Python: cv.**MatchShapes**(object1, object2, method, parameter=0) → None

>
> **Parameters**
>
> > **object1** – First contour or grayscale image.
> >
> > **object2** – Second contour or grayscale image.
> >
> > **method** – Comparison method: `CV_CONTOUR_MATCH_I1` , `CV_CONTOURS_MATCH_I2` or `CV_CONTOURS_MATCH_I3` (see the details below).
> >
> > **parameter** – Method-specific parameter (not supported now).

The function compares two shapes. All three implemented methods use the Hu invariants (see `HuMoments()`) as follows (A denotes `object1`,:math:B denotes `object2`):

- method=CV_CONTOUR_MATCH_I1

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

- method=CV_CONTOUR_MATCH_I2

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

- method=CV_CONTOUR_MATCH_I3

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

where

$$m_i^A = \text{sign}(h_i^A) \cdot \log h_i^A$$
$$m_i^B = \text{sign}(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B , respectively.

pointPolygonTest

Performs a point-in-contour test.

C++: double **pointPolygonTest**(InputArray **contour**, Point2f **pt**, bool **measureDist**)

Python: cv2.**pointPolygonTest**(contour, pt, measureDist) → retval

C: double **cvPointPolygonTest**(const CvArr* **contour**, CvPoint2D32f **pt**, int **measureDist**)

Python: cv.**PointPolygonTest**(contour, pt, measureDist) → double

> **Parameters**
>
> > **contour** – Input contour.
> >
> > **pt** – 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 a contour 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 is a signed distance between the point and the nearest contour edge.

See below a sample output of the function where each image pixel is tested against the contour.
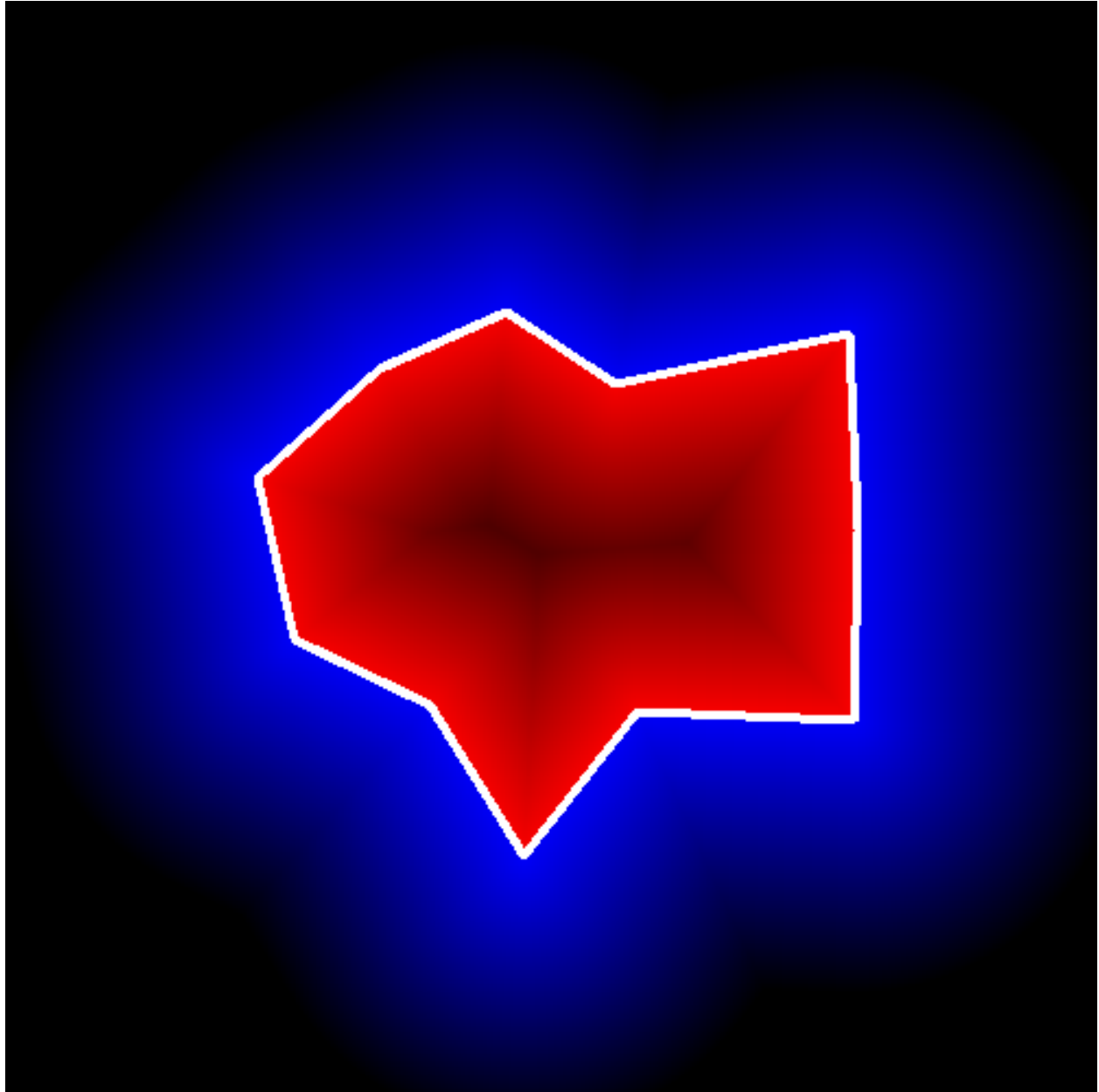
3.6 Planar Subdivisions (C API)

CvSubdiv2D

struct `CvSubdiv2D`
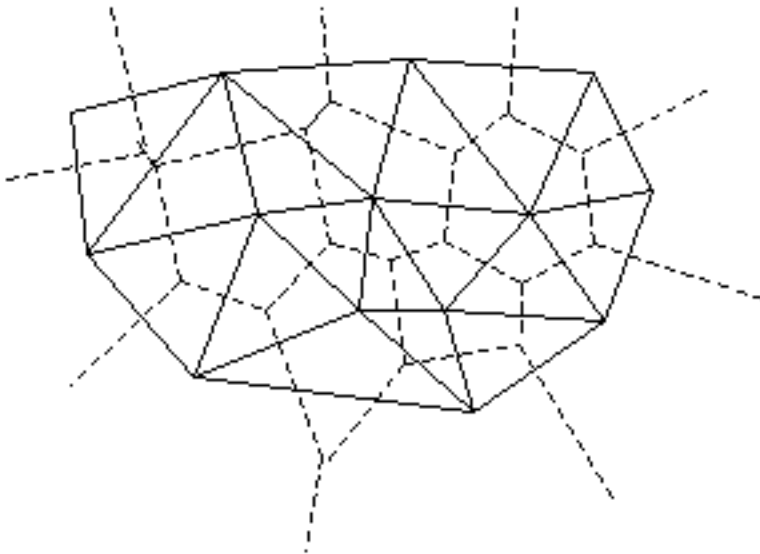
Planar subdivision.

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
CvSubdiv2D;
```

Planar subdivision is the subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on a 2D point set, where the points are linked together and form a planar graph, which, together with a few edges connecting the exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision, there is a dual subdivision in which facets and points (subdivision vertices) swap their roles. This means that a facet is treated as a vertex (called a virtual point below) of the dual subdivision and the original subdivision vertices become facets. In the figure below, the original subdivision is marked with solid lines and dual subdivision - with dotted lines.

OpenCV subdivides a plane into triangles using the Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case, the dual subdivision is a Voronoi diagram of the input 2D point set. The subdivisions can be used for the 3D piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG), and so forth.

CvQuadEdge2D

struct CvQuadEdge2D

Quad-edge of a planar subdivision.

```
/* one of edges within quad-edge, lower 2 bits is index (0..3)
   and upper bits are quad-edge pointer */
typedef long CvSubdiv2DEdge;

/* quad-edge structure fields */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```
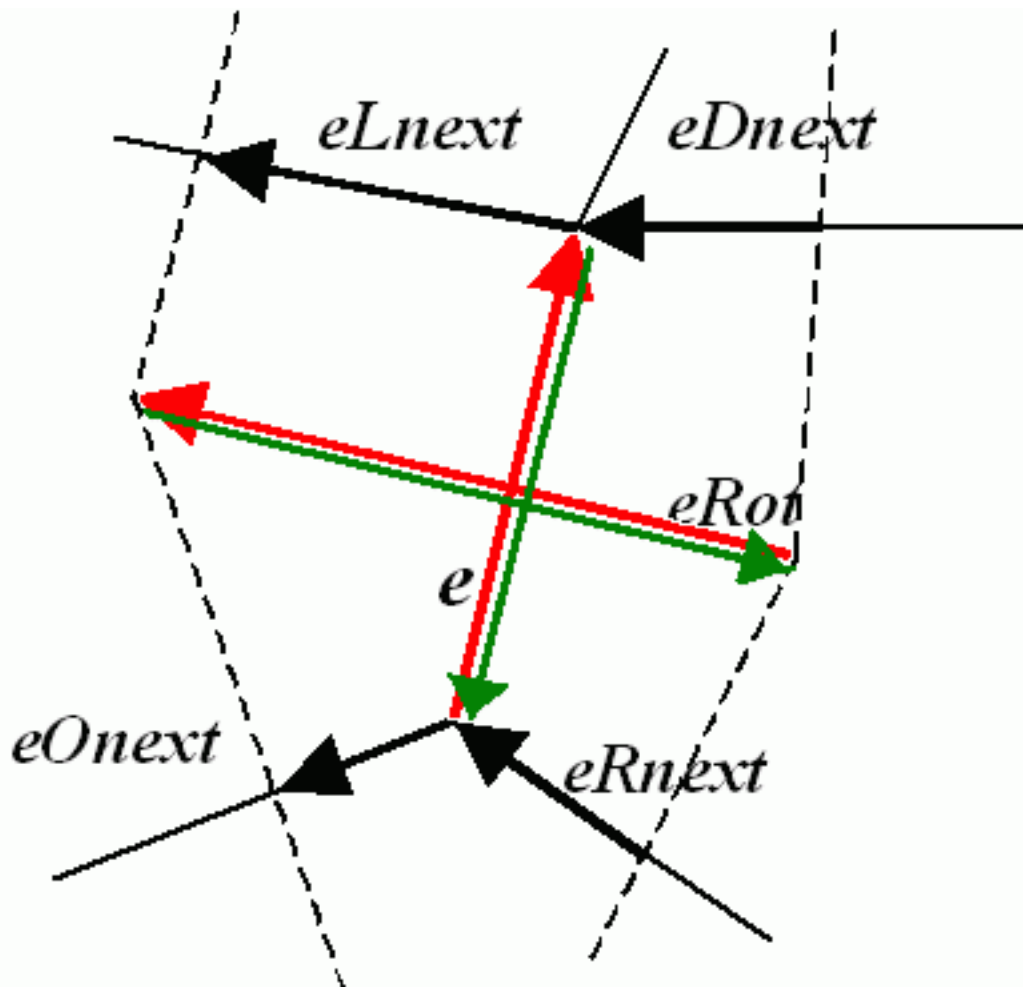
Quad-edge is a basic element of a subdivision containing four edges (e, eRot, reversed e, and reversed eRot):

CvSubdiv2DPoint

struct CvSubdiv2DPoint

Point of an original or dual subdivision.

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int             flags;      \
    CvSubdiv2DEdge first;       \
    CvPoint2D32f   pt;          \
    int id;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

- **id** This integer can be used to index auxillary data associated with each vertex of the planar subdivision.

CalcSubdivVoronoi2D

Calculates the coordinates of the Voronoi diagram cells.

C: void **cvCalcSubdivVoronoi2D**(CvSubdiv2D* **subdiv**)

Python: cv.**CalcSubdivVoronoi2D**(subdiv) → None

> **Parameters**
>
> > **subdiv** – Delaunay subdivision, in which all the points are already added.

The function calculates the coordinates of virtual points. All virtual points corresponding to a vertex of the original subdivision form (when connected together) a boundary of the Voronoi cell at that point.

ClearSubdivVoronoi2D

Removes all virtual points.

C: void **cvClearSubdivVoronoi2D**(CvSubdiv2D* **subdiv**)

Python: cv.**ClearSubdivVoronoi2D**(subdiv) → None

> **Parameters**
>
> > **subdiv** – Delaunay subdivision.

The function removes all of the virtual points. It is called internally in `CalcSubdivVoronoi2D()` if the subdivision was modified after the previous call to the function.

CreateSubdivDelaunay2D

Creates an empty Delaunay triangulation.

C: CvSubdiv2D* **cvCreateSubdivDelaunay2D**(CvRect **rect**, CvMemStorage* **storage**)

Python: cv.**CreateSubdivDelaunay2D**(rect, storage) → emptyDelaunayTriangulation

> **Parameters**
>
> > **rect** – Rectangle that includes all of the 2D points that are to be added to the subdivision.
> >
> > **storage** – Container for the subdivision.

The function creates an empty Delaunay subdivision where 2D points can be added using the function `SubdivDelaunay2DInsert()`. All of the points to be added must be within the specified rectangle, otherwise a runtime error is raised.

Note that the triangulation is a single large triangle that covers the given rectangle. Hence the three vertices of this triangle are outside the rectangle `rect`.

FindNearestPoint2D

Finds the subdivision vertex closest to the given point.

C: CvSubdiv2DPoint* **cvFindNearestPoint2D**(CvSubdiv2D* **subdiv**, CvPoint2D32f **pt**)

Python: cv.**FindNearestPoint2D**(subdiv, pt) → point

> **Parameters**
>
> > **subdiv** – Delaunay or another subdivision.
> >
> > **pt** – Input point.

The function is another function that locates the input point within the subdivision. It finds the subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using `Subdiv2DLocate()`) is used as a starting point. The function returns a pointer to the found subdivision vertex.

Subdiv2DEdgeDst

Returns the edge destination.

C: CvSubdiv2DPoint* **cvSubdiv2DEdgeDst**(CvSubdiv2DEdge **edge**)

Python: cv.**Subdiv2DEdgeDst**(edge) → point

> **Parameters**
>
> > **edge** – Subdivision edge (not a quad-edge).

The function returns the edge destination. The returned pointer may be NULL if the edge is from a dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using the function `CalcSubdivVoronoi2D()`.

Subdiv2DGetEdge

Returns one of the edges related to the given edge.

C: CvSubdiv2DEdge **cvSubdiv2DGetEdge**(CvSubdiv2DEdge **edge**, CvNextEdgeType **type**)

Python: cv.**Subdiv2DGetEdge**(edge, type) → CvSubdiv2DEdge

> **Parameters**
>
> > **edge** – Subdivision edge (not a quad-edge).
> >
> > **type** – Parameter specifying which of the related edges to return. The following values are possible:
> >
> > – **CV_NEXT_AROUND_ORG** next around the edge origin (`eOnext` on the picture below if `e` is the input edge)
> >
> > – **CV_NEXT_AROUND_DST** next around the edge vertex (`eDnext`)

- **CV_PREV_AROUND_ORG** previous around the edge origin (reversed `eRnext`)
- **CV_PREV_AROUND_DST** previous around the edge destination (reversed `eLnext`)
- **CV_NEXT_AROUND_LEFT** next around the left facet (`eLnext`)
- **CV_NEXT_AROUND_RIGHT** next around the right facet (`eRnext`)
- **CV_PREV_AROUND_LEFT** previous around the left facet (reversed `eOnext`)
- **CV_PREV_AROUND_RIGHT** previous around the right facet (reversed `eDnext`)
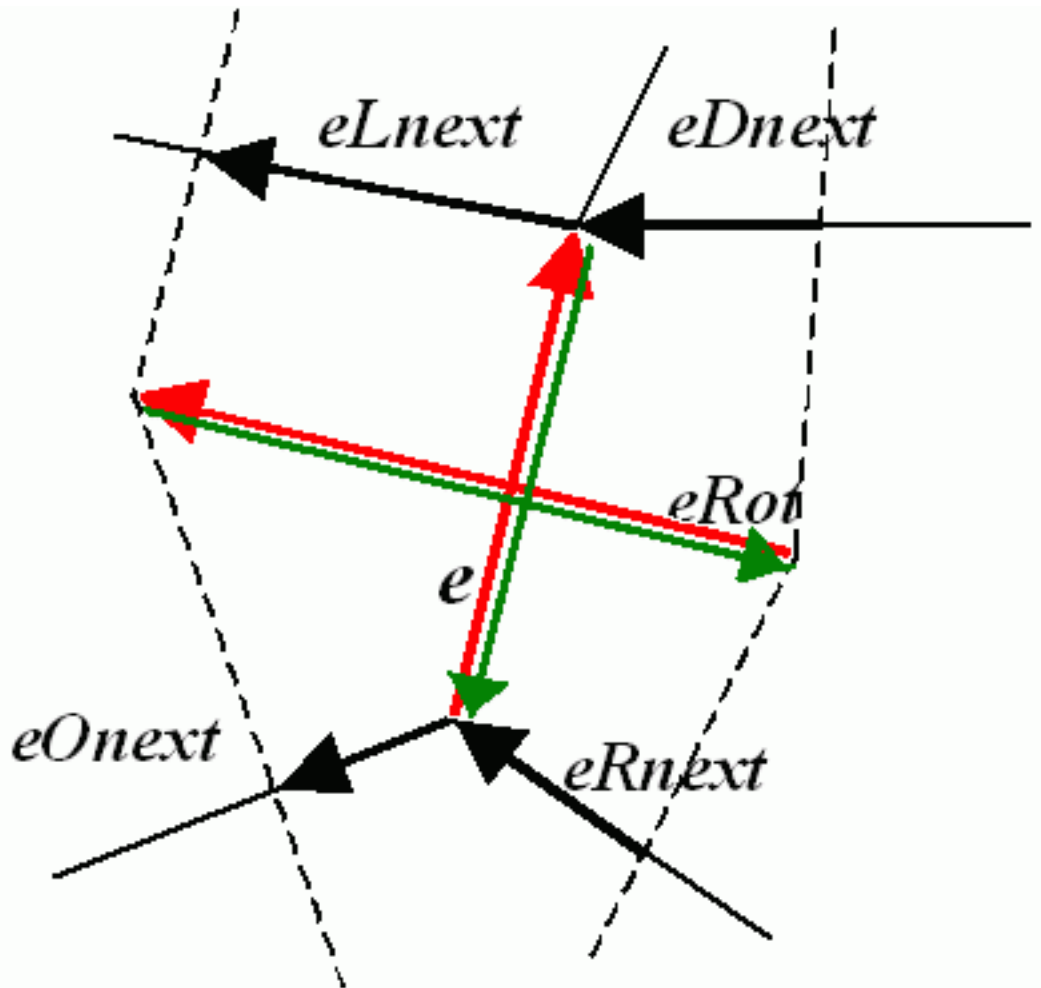
The function returns one of the edges related to the input edge.

Subdiv2DNextEdge

Returns next edge around the edge origin.

C: CvSubdiv2DEdge **cvSubdiv2DNextEdge**(CvSubdiv2DEdge **edge**)

Python: cv.**Subdiv2DNextEdge**(edge) → CvSubdiv2DEdge

> **Parameters**
>
> > **edge** – Subdivision edge (not a quad-edge).

The function returns the next edge around the edge origin: `eOnext` on the picture above if e is the input edge).

Subdiv2DLocate

Returns the location of a point within a Delaunay triangulation.

C: CvSubdiv2DPointLocation **cvSubdiv2DLocate**(CvSubdiv2D* **subdiv**, CvPoint2D32f **pt**, CvSubdiv2DEdge* **edge**, CvSubdiv2DPoint** **vertex**=NULL)

Python: cv.**Subdiv2DLocate**(subdiv, pt) -> (loc, where)

> **Parameters**
>
> > **subdiv** – Delaunay or another subdivision.
> >
> > **pt** – Point to locate.
> >
> > **edge** – Output edge that the point belongs to or is located to the right of it.
> >
> > **vertex** – Optional output vertex double pointer the input point coinsides with.

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

- The point falls into some facet. The function returns `CV_PTLOC_INSIDE` and *edge will contain one of edges of the facet.

- The point falls onto the edge. The function returns `CV_PTLOC_ON_EDGE` and *edge will contain this edge.

- The point coincides with one of the subdivision vertices. The function returns `CV_PTLOC_VERTEX` and *vertex will contain a pointer to the vertex.

- The point is outside the subdivsion reference rectangle. The function returns `CV_PTLOC_OUTSIDE_RECT` and no pointers are filled.

- One of input arguments is invalid. A runtime error is raised or, if silent or "parent" error processing mode is selected, `CV_PTLOC_ERROR` is returnd.

Subdiv2DRotateEdge

Returns another edge of the same quad-edge.

C: CvSubdiv2DEdge **cvSubdiv2DRotateEdge**(CvSubdiv2DEdge **edge**, int **rotate**)

Python: cv.**Subdiv2DRotateEdge**(edge, rotate) $\rightarrow$ CvSubdiv2DEdge

> **Parameters**
>
> > **edge** – Subdivision edge (not a quad-edge).
> >
> > **rotate** – Parameter specifying which of the edges of the same quad-edge as the input one to return. The following values are possible:
> >
> > - **0** the input edge (`e` on the picture below if `e` is the input edge)
> >
> > - **1** the rotated edge (`eRot`)
> >
> > - **2** the reversed edge (reversed `e` (in green))
> >
> > - **3** the reversed rotated edge (reversed `eRot` (in green))

The function returns one of the edges of the same quad-edge as the input edge.

SubdivDelaunay2DInsert

Inserts a single point into a Delaunay triangulation.

C: CvSubdiv2DPoint* **cvSubdivDelaunay2DInsert**(CvSubdiv2D* **subdiv**, CvPoint2D32f **pt**)

Python: cv.**SubdivDelaunay2DInsert**(subdiv, pt) → point

> **Parameters**
>
> > **subdiv** – Delaunay subdivision created by the function `CreateSubdivDelaunay2D()`.
> >
> > **pt** – Inserted point.

The function inserts a single point into a subdivision and modifies the subdivision topology appropriately. If a point with the same coordinates exists already, no new point is added. The function returns a pointer to the allocated point. No virtual point coordinates are calculated at this stage.

3.7 Motion Analysis and Object Tracking

accumulate

Adds an image to the accumulator.

C++: void **accumulate**(InputArray **src**, InputOutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**accumulate**(src, dst[, mask]) → None

C: void **cvAcc**(const CvArr* **src**, CvArr* **dst**, const CvArr* **mask**=NULL)

Python: cv.**Acc**(src, dst, mask=None) → None

> **Parameters**
>
> > **src** – Input image as 1- or 3-channel, 8-bit or 32-bit floating point.
> >
> > **dst** – 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` :

$$\mathtt{dst}(x, y) \leftarrow \mathtt{dst}(x, y) + \mathtt{src}(x, y) \quad \text{if} \quad \mathtt{mask}(x, y) \neq 0$$

The function supports multi-channel images. Each channel is processed independently.

The functions accumulate* can be used, for example, to collect statistics of a scene background viewed by a still camera and for the further foreground-background segmentation.

See Also:

`accumulateSquare()`, `accumulateProduct()`, `accumulateWeighted()`

accumulateSquare

Adds the square of a source image to the accumulator.

C++: void **accumulateSquare**(InputArray **src**, InputOutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**accumulateSquare**(src, dst[, mask]) → None

C: void **cvSquareAcc**(const CvArr* **src**, CvArr* **dst**, const CvArr* **mask**=NULL)

Python: cv.**SquareAcc**(src, dst, mask=None) → None

> **Parameters**

>> **src** – Input image as 1- or 3-channel, 8-bit or 32-bit floating point.

>> **dst** – 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 a power of 2, to the accumulator `dst` :

$$\mathtt{dst}(x,y) \leftarrow \mathtt{dst}(x,y) + \mathtt{src}(x,y)^2 \quad \text{if} \quad \mathtt{mask}(x,y) \neq 0$$

The function supports multi-channel images. Each channel is processed independently.

See Also:

accumulateSquare(), accumulateProduct(), accumulateWeighted()

accumulateProduct

Adds the per-element product of two input images to the accumulator.

C++: void **accumulateProduct**(InputArray **src1**, InputArray **src2**, InputOutputArray **dst**, InputArray **mask**=noArray())

Python: cv2.**accumulateProduct**(src1, src2, dst[, mask]) → None

C: void **cvMultiplyAcc**(const CvArr* **src1**, const CvArr* **src2**, CvArr* **dst**, const CvArr* **mask**=NULL)

Python: cv.**MultiplyAcc**(src1, src2, dst, mask=None) → None

> **Parameters**

>> **src1** – First input image, 1- or 3-channel, 8-bit or 32-bit floating point.

>> **src2** – 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 two images or their selected regions to the accumulator `dst` :

$$\mathtt{dst}(x,y) \leftarrow \mathtt{dst}(x,y) + \mathtt{src1}(x,y) \cdot \mathtt{src2}(x,y) \quad \text{if} \quad \mathtt{mask}(x,y) \neq 0$$

The function supports multi-channel images. Each channel is processed independently.

See Also:

accumulate(), accumulateSquare(), accumulateWeighted()

accumulateWeighted

Updates a running average.

C++: void **accumulateWeighted**(InputArray **src**, InputOutputArray **dst**, double **alpha**, InputArray **mask**=noArray())

3.7. Motion Analysis and Object Tracking

Python: cv2.**accumulateWeighted**(src, dst, alpha[, mask]) → None

C: void **cvRunningAvg**(const CvArr* **src**, CvArr* **dst**, double **alpha**, const CvArr* **mask**=NULL)

Python: cv.**RunningAvg**(src, dst, alpha, mask=None) → None

>   **Parameters**

>   > **src** – Input image as 1- or 3-channel, 8-bit or 32-bit floating point.

>   > **dst** – 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 a frame sequence:

$$\texttt{dst}(x,y) \leftarrow (1-\texttt{alpha}) \cdot \texttt{dst}(x,y) + \texttt{alpha} \cdot \texttt{src}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

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()

phaseCorrelate

The function is used to detect translational shifts that occur between two images. The operation takes advantage of the Fourier shift theorem for detecting the translational shift in the frequency domain. It can be used for fast image registration as well as motion esitimation. For more information please see http://en.wikipedia.org/wiki/Phase_correlation .

Calculates the cross-power spectrum of two supplied source arrays. The arrays are padded if needed with getOptimalDFTSize .

C++: Point2d **phaseCorrelate**(InputArray **src1**, InputArray **src2**, InputArray **window**=noArray())

>   **Parameters**

>   > **src1** – Source floating point array (CV_32FC1 or CV_64FC1)

>   > **src2** – Source floating point array (CV_32FC1 or CV_64FC1)

>   > **window** – Floating point array with windowing coefficients to reduce edge effects (optional).

>   > **result** – Detected phase shift (sub-pixel) between the two arrays.

The function performs the following equations

- First it applies a Hanning window (see http://en.wikipedia.org/wiki/Hann_function) to each image to remove possible edge effects. This window is cached until the array size changes to speed up processing time.

- Next it computes the forward DFTs of each source array: .. math:

  \mathbf{G}_a = \mathcal{F}\{src_1\}, \; \mathbf{G}_b = \mathcal{F}\{src_2\}

  where $\mathcal{F}$ is the forward DFT.

- It then computes the cross-power spectrum of each frequency domain array: .. math:

R = \frac{ \mathbf{G}_a \mathbf{G}_b^*}{|\mathbf{G}_a \mathbf{G}_b^*|}

- Next the cross-correlation is converted back into the time domain via the inverse DFT: .. math:

r = \mathcal{F}^{-1}\{R\}

- Finally, it computes the peak location and computes a 5x5 weighted centroid around the peak to achieve sub-pixel accuracy. .. math:

(\Delta x, \Delta y) = \texttt{weighted_centroid}\{\arg \max_{(x, y)}\{r\}\}

See Also:

dft(), getOptimalDFTSize(), idft(), mulSpectrums() createHanningWindow()

createHanningWindow

This function computes a Hanning window coefficients in two dimensions. See http://en.wikipedia.org/wiki/Hann_function and http://en.wikipedia.org/wiki/Window_function for more information.

C++: void **createHanningWindow**(OutputArray **dst**, Size **winSize**, int **type**)

>   **Parameters**

>> **dst** – Destination array to place Hann coefficients in

>> **winSize** – The window size specifications

>> **type** – Created array type

An example is shown below:

```
// create hanning window of size 100x100 and type CV_32F
Mat hann;
createHanningWindow(hann, Size(100, 100), CV_32F);
```

See Also:

phaseCorrelate()

3.8 Feature Detection

Canny

Finds edges in an image using the [Canny86] algorithm.

C++: void **Canny**(InputArray **image**, OutputArray **edges**, double **threshold1**, double **threshold2**, int **apertureSize**=3, bool **L2gradient**=false)

Python: cv2.**Canny**(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]]) → edges

C: void **cvCanny**(const CvArr* **image**, CvArr* **edges**, double **threshold1**, double **threshold2**, int **apertureSize**=3)

Python: cv.**Canny**(image, edges, threshold1, threshold2, apertureSize=3) → None

>   **Parameters**

>> **image** – Single-channel 8-bit input image.

edges – Output edge map. It has the same size and type as `image` .

threshold1 – First threshold for the hysteresis procedure.

threshold2 – Second threshold for the hysteresis procedure.

apertureSize – Aperture size for the `Sobel()` operator.

L2gradient – Flag indicating whether a 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 initial segments of strong edges. See http://en.wikipedia.org/wiki/Canny_edge_detector

cornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

C++: void **cornerEigenValsAndVecs**(InputArray **src**, OutputArray **dst**, int **blockSize**, int **apertureSize**, int **borderType**=BORDER_DEFAULT)

Python: cv2.**cornerEigenValsAndVecs**(src, blockSize, ksize$\big[$, dst$\big[$, borderType $\big]\big]$) $\rightarrow$ dst

C: void **cvCornerEigenValsAndVecs**(const CvArr* **image**, CvArr* **eigenvv**, int **blockSize**, int **aperture-Size**=3)

Python: cv.**CornerEigenValsAndVecs**(image, eigenvv, blockSize, apertureSize=3) $\rightarrow$ None

> **Parameters**

> > **src** – Input single-channel 8-bit or floating-point image.

> > **dst** – Image to store the results. It has the same size as `src` and the type `CV_32FC(6)` .

> > **blockSize** – Neighborhood size (see details below).

> > **apertureSize** – Aperture parameter for the `Sobel()` operator.

> > **boderType** – Pixel extrapolation method. See `borderInterpolate()` .

For every pixel p , the function `cornerEigenValsAndVecs` considers a `blockSize` $\times$ `blockSize` neigborhood $S(p)$. 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 the `Sobel()` operator.

After that, it finds eigenvectors and eigenvalues of M and stores them in the destination image as $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

- λ_1, λ_2 are the non-sorted eigenvalues of M

- x_1, y_1 are the eigenvectors corresponding to λ_1

- x_2, y_2 are the eigenvectors corresponding to λ_2

The output of the function can be used for robust edge or corner detection.

See Also:

`cornerMinEigenVal()`, `cornerHarris()`, `preCornerDetect()`

cornerHarris

Harris edge detector.

C++: void **cornerHarris**(InputArray **src**, OutputArray **dst**, int **blockSize**, int **apertureSize**, double **k**, int **borderType**=BORDER_DEFAULT)

Python: cv2.**cornerHarris**(src, blockSize, ksize, k[, dst[, borderType]]) → dst

C: void **cvCornerHarris**(const CvArr* **image**, CvArr* **harrisDst**, int **blockSize**, int **apertureSize**=3, double **k**=0.04)

Python: cv.**CornerHarris**(image, harrisDst, blockSize, apertureSize=3, k=0.04) → None

> **Parameters**
>
>> **src** – Input single-channel 8-bit or floating-point image.
>>
>> **dst** – Image to store the Harris detector responses. It has the type CV_32FC1 and the same size as src .
>>
>> **blockSize** – Neighborhood size (see the details on 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 2×2 gradient covariance matrix $M^{(x,y)}$ over a blockSize × blockSize neighborhood. Then, it computes the following characteristic:

$$\mathtt{dst}(x, y) = \det M^{(x,y)} - k \cdot \left(\operatorname{tr} M^{(x,y)} \right)^2$$

Corners in the image can be found as the local maxima of this response map.

cornerMinEigenVal

Calculates the minimal eigenvalue of gradient matrices for corner detection.

C++: void **cornerMinEigenVal**(InputArray **src**, OutputArray **dst**, int **blockSize**, int **apertureSize**=3, int **borderType**=BORDER_DEFAULT)

Python: cv2.**cornerMinEigenVal**(src, blockSize[, dst[, ksize[, borderType]]]) → dst

C: void **cvCornerMinEigenVal**(const CvArr* **image**, CvArr* **eigenval**, int **blockSize**, int **apertureSize**=3)

Python: cv.**CornerMinEigenVal**(image, eigenval, blockSize, apertureSize=3) → None

> **Parameters**
>
>> **src** – Input single-channel 8-bit or floating-point image.
>>
>> **dst** – Image to store the minimal eigenvalues. It has the type CV_32FC1 and the same size as src .
>>
>> **blockSize** – Neighborhood size (see the details on 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 covariance matrix of derivatives, that is, $\min(\lambda_1, \lambda_2)$ in terms of the formulae in the `cornerEigenValsAndVecs()` description.

cornerSubPix

Refines the corner locations.

C++: void **cornerSubPix**(InputArray **image**, InputOutputArray **corners**, Size **winSize**, Size **zeroZone**, TermCriteria **criteria**)

Python: cv2.**cornerSubPix**(image, corners, winSize, zeroZone, criteria) → None

C: void **cvFindCornerSubPix**(const CvArr* **image**, CvPoint2D32f* **corners**, int **count**, CvSize **winSize**, CvSize **zeroZone**, CvTermCriteria **criteria**)

Python: cv.**FindCornerSubPix**(image, corners, winSize, zeroZone, criteria) → corners

> **Parameters**
>
> > **image** – Input image.
> >
> > **corners** – Initial coordinates of the input corners and refined coordinates provided for 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 is 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 a size.
> >
> > **criteria** – Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after `criteria.maxCount` iterations or when the corner position moves by less than `criteria.epsilon` on some iteration.

The function iterates to find the sub-pixel accurate location of corners or radial saddle points, as shown on the figure below.
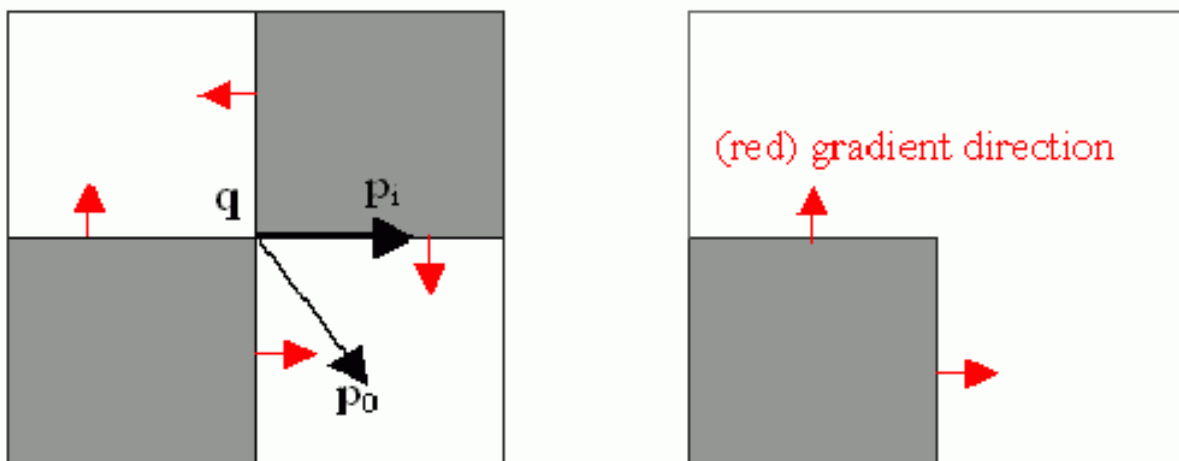
Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider

the expression:

$$\epsilon_i = DI_{p_i}{}^T \cdot (q - p_i)$$

where DI_{p_i} is an image gradient at one of the points p_i in a neighborhood of q. The value of q is to be found so that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot DI_{p_i}{}^T) - \sum_i (DI_{p_i} \cdot DI_{p_i}{}^T \cdot p_i)$$

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 stays within a set threshold.

goodFeaturesToTrack

Determines strong corners on an image.

C++: void **goodFeaturesToTrack**(InputArray **image**, OutputArray **corners**, int **maxCorners**, double **qualityLevel**, double **minDistance**, InputArray **mask**=noArray(), int **blockSize**=3, bool **useHarrisDetector**=false, double **k**=0.04)

Python: cv2.**goodFeaturesToTrack**(image, maxCorners, qualityLevel, minDistance[, corners[, mask[, blockSize[, useHarrisDetector[, k]]]]]) → corners

C: void **cvGoodFeaturesToTrack**(const CvArr* **image**, CvArr* **eigImage**, CvArr* **tempImage**, CvPoint2D32f* **corners**, int* **cornerCount**, double **qualityLevel**, double **minDistance**, const CvArr* **mask**=NULL, int **blockSize**=3, int **useHarris**=0, double **k**=0.04)

Python: cv.**GoodFeaturesToTrack**(image, eigImage, tempImage, cornerCount, qualityLevel, minDistance, mask=None, blockSize=3, useHarris=0, k=0.04) → corners

> **Parameters**

>> **image** – Input 8-bit or floating-point 32-bit, single-channel image.

>> **eigImage** – The parameter is ignored.

>> **tempImage** – The parameter is ignored.

>> **corners** – Output vector of detected corners.

>> **maxCorners** – Maximum number of corners to return. If there are more corners than are found, the strongest of them is returned.

>> **qualityLevel** – Parameter characterizing the minimal accepted quality of image corners. The parameter value is multiplied by the best corner quality measure, which is the minimal eigenvalue (see `cornerMinEigenVal()`) or the Harris function response (see `cornerHarris()`). The corners with the quality measure less than the product are rejected. For example, if the best corner has the quality measure = 1500, and the `qualityLevel=0.01` , then all the corners with the quality measure less than 15 are rejected.

>> **minDistance** – Minimum possible Euclidean distance between the returned corners.

>> **mask** – Optional region of interest. If the image is not empty (it needs to have the type `CV_8UC1` and the same size as `image`), it specifies the region in which the corners are detected.

> **blockSize** – Size of an average block for computing a derivative covariation matrix over each pixel neighborhood. See `cornerEigenValsAndVecs()` .
>
> **useHarrisDetector** – Parameter indicating whether to use a Harris detector (see `cornerHarris()`) or `cornerMinEigenVal()`.
>
> **k** – Free parameter of the Harris detector.

The function finds the most prominent corners in the image or in the specified image region, as described in [Shi94]:

1. Function calculates the corner quality measure at every source image pixel using the `cornerMinEigenVal()` or `cornerHarris()` .

2. Function performs a non-maximum suppression (the local maximums in *3 x 3* neighborhood are retained).

3. The corners with the minimal eigenvalue less than $qualityLevel \cdot \max_{x,y} qualityMeasureMap(x, y)$ are rejected.

4. The remaining corners are sorted by the quality measure in the descending order.

5. Function throws away each corner for which there is a stronger corner at a distance less than `maxDistance`.

The function can be used to initialize a point-based tracker of an object.

Note: If the function is called with different values A and B of the parameter `qualityLevel` , and A > {B}, the vector of returned corners with `qualityLevel=A` will be the prefix of the output vector with `qualityLevel=B` .

See Also:

`cornerMinEigenVal()`, `cornerHarris()`, `calcOpticalFlowPyrLK()`, `estimateRigidTransform()`, `PlanarObjectDetector`, `OneWayDescriptor`

HoughCircles

Finds circles in a grayscale image using the Hough transform.

C++: void **HoughCircles**(InputArray **image**, OutputArray **circles**, int **method**, double **dp**, double **minDist**, double **param1**=100, double **param2**=100, int **minRadius**=0, int **maxRadius**=0)

C: CvSeq* **cvHoughCircles**(CvArr* **image**, CvMemStorage* **circleStorage**, int **method**, double **dp**, double **minDist**, double **param1**=100, double **param2**=100, int **minRadius**=0, int **maxRadius**=0)

Python: cv2.**HoughCircles**(image, method, dp, minDist[, circles[, param1[, param2[, minRadius[, maxRadius]]]]]) → circles

> **Parameters**
>
> > **image** – 8-bit, single-channel, grayscale input image.
> >
> > **circles** – Output vector of found circles. Each vector is encoded as a 3-element floating-point vector $(x, y, radius)$.
> >
> > **circleStorage** – In C function this is a memory storage that will contain the output sequence of found circles.
> >
> > **method** – Detection method to use. Currently, the only implemented method is `CV_HOUGH_GRADIENT` , which is basically *21HT* , described in [Yuen90].
> >
> > **dp** – Inverse ratio of the accumulator resolution to the image resolution. For example, if `dp=1` , the accumulator has the same resolution as the input image. If `dp=2` , the accumulator has half as big width and height.

> > **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** – First method-specific parameter. In case of `CV_HOUGH_GRADIENT` , it is the higher threshold of the two passed to the `Canny()` edge detector (the lower one is twice smaller).
> >
> > **param2** – Second method-specific parameter. In case of `CV_HOUGH_GRADIENT` , it is the accumulator threshold for the circle centers at the 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 a modification of the Hough transform.

Example:

```cpp
#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, Size(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[i][1]));
        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, 8, 0 );
    }
    namedWindow( "circles", 1 );
    imshow( "circles", img );
    return 0;
}
```

Note: Usually the function detects the centers of circles well. However, it may fail to find correct radii. You can assist to 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 an additional procedure.

See Also:

`fitEllipse()`, `minEnclosingCircle()`

HoughLines

Finds lines in a binary image using the standard Hough transform.

C++: void **HoughLines**(InputArray **image**, OutputArray **lines**, double **rho**, double **theta**, int **threshold**, double **srn**=0, double **stn**=0)

Python: cv2.**HoughLines**(image, rho, theta, threshold[, lines[, srn[, stn]]]) → lines

C: CvSeq* **cvHoughLines2**(CvArr* **image**, void* **storage**, int **method**, double **rho**, double **theta**, int **threshold**, double **param1**=0, double **param2**=0)

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

> **Parameters**

>> **image** – 8-bit, single-channel binary source image. The image may be modified by the function.

>> **lines** – Output vector of lines. Each line is represented by a two-element vector (ρ, θ) . ρ is the distance from the coordinate origin $(0, 0)$ (top-left corner of the image). θ 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** – Accumulator threshold parameter. Only those lines are returned that get enough votes ($>$ `threshold`).

>> **srn** – For the multi-scale Hough transform, it is a divisor for the distance resolution `rho` . The coarse accumulator distance resolution is `rho` and the accurate accumulator resolution is `rho/srn` . If both `srn=0` and `stn=0` , the classical Hough transform is used. Otherwise, both these parameters should be positive.

>> **stn** – For the multi-scale Hough transform, it is a divisor for the distance resolution `theta`.

>> **method** – One of the following Hough transform variants:

>> – **CV_HOUGH_STANDARD** classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ) , where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of `CV_32FC2` type

>> – **CV_HOUGH_PROBABILISTIC** probabilistic Hough transform (more efficient in case if the picture contains a few long linear segments). It returns line segments rather than the whole line. Each segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of the `CV_32SC4` type.

>> – **CV_HOUGH_MULTI_SCALE** multi-scale variant of the classical Hough transform. The lines are encoded the same way as `CV_HOUGH_STANDARD`.

>> **param1** – First method-dependent parameter:

>> – For the classical Hough transform, it is not used (0).

>> – For the probabilistic Hough transform, it is the minimum line length.

>> – For the multi-scale Hough transform, it is `srn`.

>> **param2** – Second method-dependent parameter:

>> – For the classical Hough transform, it is not used (0).

>> – For the probabilistic Hough transform, it is the maximum gap between line segments lying on the same line to treat them as a single line segment (that is, to join them).

– For the multi-scale Hough transform, it is `stn`.

The function implements the standard or standard multi-scale Hough transform algorithm for line detection. See http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm for a good explanation of Hough transform. See also the example in `HoughLinesP()` description.

HoughLinesP

Finds line segments in a binary image using the probabilistic Hough transform.

C++: void **HoughLinesP**(InputArray **image**, OutputArray **lines**, double **rho**, double **theta**, int **threshold**, double **minLineLength**=0, double **maxLineGap**=0)

Python: cv2.**HoughLinesP**(image, rho, theta, threshold[, lines[, minLineLength[, maxLineGap]]]) → lines

> **Parameters**
>
> > **image** – 8-bit, single-channel binary source image. The image may be modified by the function.
> >
> > **lines** – 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 detected line segment.
> >
> > **rho** – Distance resolution of the accumulator in pixels.
> >
> > **theta** – Angle resolution of the accumulator in radians.
> >
> > **threshold** – Accumulator threshold parameter. Only those lines are returned that get enough votes (> `threshold`).
> >
> > **minLineLength** – Minimum line length. Line segments shorter than that are rejected.
> >
> > **maxLineGap** – Maximum allowed gap between points on the same line to link them.

The function implements the probabilistic Hough transform algorithm for line detection, described in [Matas00]. See the line detection example below:

```cpp
/* This is a standalone program. Pass an image name as the first parameter
of the program.  Switch between standard and probabilistic Hough transform
by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

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), 3, 8 );
    }
#endif
    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "Detected Lines", 1 );
    imshow( "Detected Lines", color_dst );

    waitKey(0);
    return 0;
}
```
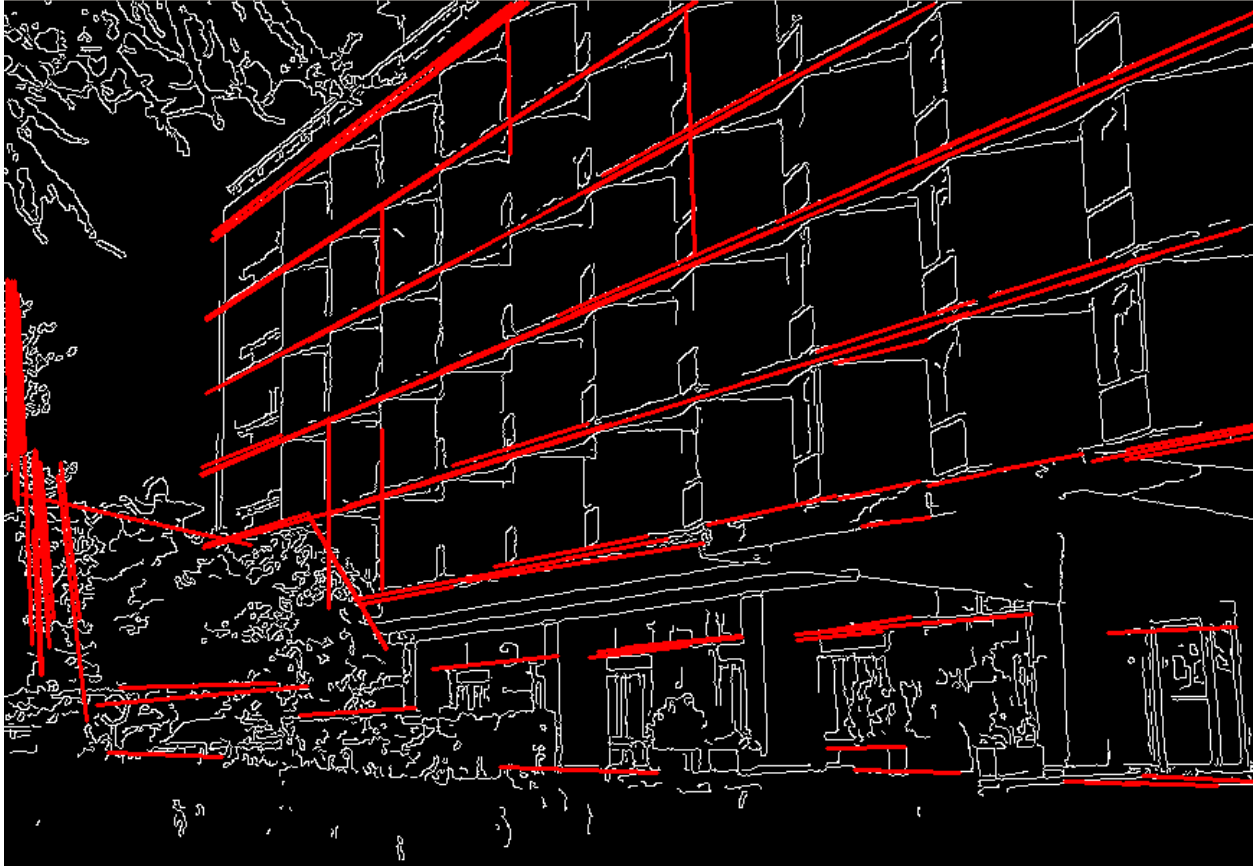
This is a sample picture the function parameters have been tuned for:

And this is the output of the above program in case of the probabilistic Hough transform:

preCornerDetect

Calculates a feature map for corner detection.

C++: void `preCornerDetect`(InputArray **src**, OutputArray **dst**, int **apertureSize**, int **border-Type**=BORDER_DEFAULT)

Python: cv2.`preCornerDetect`(src, ksize$\big[$, dst$\big[$, borderType $\big]\big]$) $\rightarrow$ dst

C: void `cvPreCornerDetect`(const CvArr* **image**, CvArr* **corners**, int **apertureSize**=3)

Python: cv.`PreCornerDetect`(image, corners, apertureSize=3) $\rightarrow$ None

> **Parameters**
>
> > **src** – Source single-channel 8-bit of floating-point image.
> >
> > **dst** – Output image that has the type `CV_32F` and the same size as `src` .
> >
> > **apertureSize** – Aperture size of the `Sobel()` .
> >
> > **borderType** – Pixel extrapolation method. See `borderInterpolate()` .

The function calculates the complex spatial derivative-based function of the source image

$$\mathtt{dst} = (D_x \mathtt{src})^2 \cdot D_{yy} \mathtt{src} + (D_y \mathtt{src})^2 \cdot D_{xx} \mathtt{src} - 2 D_x \mathtt{src} \cdot D_y \mathtt{src} \cdot D_{xy} \mathtt{src}$$

where D_x,:math:`D_y` are the first image derivatives, D_{xx},:math:`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;
```

3.9 Object Detection

matchTemplate

Compares a template against overlapped image regions.

C++: void **matchTemplate**(InputArray **image**, InputArray **temp**, OutputArray **result**, int **method**)

Python: cv2.**matchTemplate**(image, templ, method[, result]) → result

C: void **cvMatchTemplate**(const CvArr* **image**, const CvArr* **templ**, CvArr* **result**, int **method**)

Python: cv.**MatchTemplate**(image, templ, result, method) → None

> **Parameters**
>
> > **image** – Image where the search is running. It must be 8-bit or 32-bit floating-point.
> >
> > **templ** – Searched template. It must be not greater than the source image and have the same data type.
> >
> > **result** – Map of comparison results. It must be single-channel 32-bit floating-point. If image is $W \times H$ and templ is $w \times h$, then result is $(W - w + 1) \times (H - h + 1)$.
> >
> > **method** – Parameter specifying 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 in result. Here are the formulae for the available comparison methods (I denotes image, T template, R result). The summation is done over template and/or the image patch: $x' = 0...w - 1, y' = 0...h - 1$ * method=CV_TM_SQDIFF

$$R(x, y) = \sum_{x', y'} (T(x', y') - I(x + x', y + y'))^2$$

• method=CV_TM_SQDIFF_NORMED

$$R(x, y) = \frac{\sum_{x', y'} (T(x', y') - I(x + x', y + y'))^2}{\sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x + x', y + y')^2}}$$

• method=CV_TM_CCORR

$$R(x, y) = \sum_{x', y'} (T(x', y') \cdot I(x + x', y + y'))$$

• method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

- method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))$$

where

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

- method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

After the function finishes the comparison, the best matches can be found as global minimums (when `CV_TM_SQDIFF` was used) or maximums (when `CV_TM_CCORR` or `CV_TM_CCOEFF` was used) using the `minMaxLoc()` function. In 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.

HIGHGUI. HIGH-LEVEL GUI AND MEDIA I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt*, WinForms*, or Cocoa*) or without any UI at all, sometimes there it is required to try functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- Create and manipulate windows that can display images and "remember" their content (no need to handle repaint events from OS).

- Add trackbars to the windows, handle simple mouse events as well as keyboard commmands.

- Read and write images to/from disk or memory.

- Read video from camera or file and write video to a file.

4.1 User Interface

createTrackbar

Creates a trackbar and attaches it to the specified window.

C++: int `createTrackbar`(const string& **trackbarname**, const string& **winname**, int* **value**, int **count**, TrackbarCallback **onChange**=0, void* **userdata**=0)

C: int `cvCreateTrackbar`(const char* **trackbarName**, const char* **windowName**, int* **value**, int **count**, Cv-TrackbarCallback **onChange**)

Python: cv.`CreateTrackbar`(trackbarName, windowName, value, count, onChange) → None

> **Parameters**
>
> > **trackbarname** – Name of the created trackbar.
> >
> > **winname** – Name of the window that will be used as a parent of the created trackbar.
> >
> > **value** – Optional pointer to an integer variable whose value reflects the position of the slider. Upon creation, the slider position is defined by this variable.
> >
> > **count** – 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 the NULL pointer, no callbacks are called, but only `value` is updated.
> >
> > **userdata** – 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 slider or range control) with the specified name and range, assigns
a variable `value` to be a position syncronized with the trackbar and specifies the callback function `onChange` to be
called on the trackbar position change. The created trackbar is displayed in the specified window `winname`.

Note: **[Qt Backend Only]** `winname` can be empty (or NULL) if the trackbar should be attached to the control panel.

Clicking the label of each trackbar enables editing the trackbar values manually.

getTrackbarPos

Returns the trackbar position.

C++: int **getTrackbarPos**(const string& **trackbarname**, const string& **winname**)

Python: cv2.**getTrackbarPos**(trackbarname, winname) → retval

C: int **cvGetTrackbarPos**(const char* **trackbarName**, const char* **windowName**)

Python: cv.**GetTrackbarPos**(trackbarName, windowName) → None

> **Parameters**
>
> > **trackbarname** – Name of the trackbar.
> >
> > **winname** – Name of the window that is the parent of the trackbar.

The function returns the current position of the specified trackbar.

Note: **[Qt Backend Only]** `winname` can be empty (or NULL) if the trackbar is attached to the control panel.

---

imshow

Displays an image in the specified window.

C++: void **imshow**(const string& **winname**, InputArray **image**)

Python: cv2.**imshow**(winname, image) → None

C: void **cvShowImage**(const char* **winname**, const CvArr* **image**)

Python: cv.**ShowImage**(winname, image) → None

> **Parameters**
>
> > **winname** – Name of the window.
> >
> > **image** – Image to be shown.

The function `imshow` displays an image in the specified window. If the window was created with the
`CV_WINDOW_AUTOSIZE` flag, the image is shown with its original size. Otherwise, the image is scaled to fit the window.
The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.

- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].

- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

namedWindow

Creates a window.

C++: void **namedWindow**(const string& **winname**, int **flags**)

Python: cv2.**namedWindow**(winname[, flags]) → None

C: int **cvNamedWindow**(const char* **name**, int **flags**)

Python: cv.**NamedWindow**(name, flags=CV_WINDOW_AUTOSIZE) → None

>   **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 you cannot change the window size manually.

The function namedWindow creates a window that 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.

You can call destroyWindow() or destroyAllWindows() to close the window and de-allocate any associated memory usage. For a simple program, you do not really have to call these functions because all the resources and windows of the application are closed automatically by the operating system upon exit.

Note: Qt backend supports additional flags:

- **CV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE:** CV_WINDOW_NORMAL enables you to resize the window, whereas CV_WINDOW_AUTOSIZE adjusts automatically the window size to fit the displayed image (see imshow()), and you cannot change the window size manually.

- **CV_WINDOW_FREERATIO or CV_WINDOW_KEEPRATIO:** CV_WINDOW_FREERATIO adjusts the image with no respect to its ratio, whereas CV_WINDOW_KEEPRATIO keeps the image ratio.

- **CV_GUI_NORMAL or CV_GUI_EXPANDED:** CV_GUI_NORMAL is the old way to draw the window without statusbar and toolbar, whereas CV_GUI_EXPANDED is a new enhanced GUI.

By default, flags == CV_WINDOW_AUTOSIZE | CV_WINDOW_KEEPRATIO | CV_GUI_EXPANDED

---

destroyWindow

Destroys a window.

C++: void **destroyWindow**(const string& **winname**)

Python: cv2.**destroyWindow**(winname) → None

C: void **cvDestroyWindow**(const char* **name**)

Python: cv.**DestroyWindow**(name) → None

> **Parameters**

>> **winname** – Name of the window to be destroyed.

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

destroyAllWindows

Destroys all of the HighGUI windows.

C++: void **destroyAllWindows**()

Python: cv2.**destroyAllWindows**() → None

C: void **cvDestroyAllWindows**()

Python: cv.**DestroyAllWindows**() → None

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

MoveWindow

Moves window to the specified position

C: void **cvMoveWindow**(const char* **name**, int **x**, int **y**)

Python: cv.**MoveWindow**(name, x, y) → None

> **Parameters**

>> **name** – Window name

>> **x** – The new x-coordinate of the window

>> **y** – The new y-coordinate of the window

ResizeWindow

Resizes window to the specified size

C: void **cvResizeWindow**(const char* **name**, int **width**, int **height**)

Python: cv.**ResizeWindow**(name, width, height) → None

> **Parameters**

>> **name** – Window name

>> **width** – The new window width

>> **height** – The new window height

Note:

- The specified window size is for the image area. Toolbars are not counted.

- Only windows created without CV_WINDOW_AUTOSIZE flag can be resized.

SetMouseCallback

Sets mouse handler for the specified window

C: void **cvSetMouseCallback**(const char* **name**, CvMouseCallback **onMouse**, void* **param**=NULL)

Python: cv.**SetMouseCallback**(name, onMouse, param) → None

> **Parameters**
>
>> **name** – Window name
>>
>> **onMouse** – Mouse callback. See OpenCV samples, such as https://code.ros.org/svn/opencv/trunk/opencv/samples/cpp/ffilldemo.cpp, on how to specify and use the callback.
>>
>> **param** – The optional parameter passed to the callback.

setTrackbarPos

Sets the trackbar position.

C++: void **setTrackbarPos**(const string& **trackbarname**, const string& **winname**, int **pos**)

Python: cv2.**setTrackbarPos**(trackbarname, winname, pos) → None

C: void **cvSetTrackbarPos**(const char* **trackbarName**, const char* **windowName**, int **pos**)

Python: cv.**SetTrackbarPos**(trackbarName, windowName, pos) → None

> **Parameters**
>
>> **trackbarname** – Name of the trackbar.
>>
>> **winname** – Name of the window that is the parent of trackbar.
>>
>> **pos** – New position.

The function sets the position of the specified trackbar in the specified window.

Note: **[Qt Backend Only]** winname can be empty (or NULL) if the trackbar is attached to the control panel.

---

waitKey

Waits for a pressed key.

C++: int **waitKey**(int **delay**=0)

Python: cv2.**waitKey**([delay]) → retval

C: int **cvWaitKey**(int **delay**=0)

Python: cv.**WaitKey**(delay=0) → int

> **Parameters**
>
>> **delay** – Delay in milliseconds. 0 is the special value that means "forever".

The function waitKey waits for a key event infinitely (when delay ≤ 0) or for delay milliseconds, when it is positive. Since the OS has a minimum time between switching threads, the function will not wait exactly delay ms, it will wait at least delay ms, depending on what else is running on your computer at that time. It 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 an environment that takes care of event processing.

Note: The function only works if there is at least one HighGUI window created and the window is active. If there are several HighGUI windows, any of them can be active.

4.2 Reading and Writing Images and Video

imdecode

Reads an image from a buffer in memory.

C++: Mat **imdecode**(InputArray **buf**, int **flags**)

C: IplImage* **cvDecodeImage**(const CvMat* **buf**, int **iscolor**=CV_LOAD_IMAGE_COLOR)

C: CvMat* **cvDecodeImageM**(const CvMat* **buf**, int **iscolor**=CV_LOAD_IMAGE_COLOR)

Python: cv2.**imdecode**(buf, flags) → retval

> **Parameters**
>
> > **buf** – Input array or vector of bytes.
> >
> > **flags** – The same flags as in imread() .

The function reads an image from the specified buffer in the memory. If the buffer is too short or contains invalid data, the empty matrix/image is returned.

See imread() for the list of supported formats and flags description.

imencode

Encodes an image into a memory buffer.

C++: bool **imencode**(const string& **ext**, InputArray **img**, vector<uchar>& **buf**, const vector<int>& **params**=vector<int>())

C: CvMat* **cvEncodeImage**(const char* **ext**, const CvArr* **image**, const int* **params**=NULL)

Python: cv2.**imencode**(ext, img, buf[, params]) → retval

> **Parameters**
>
> > **ext** – File extension that defines the output format.
> >
> > **img** – Image to be written.
> >
> > **buf** – Output buffer resized to fit the compressed image.
> >
> > **params** – Format-specific parameters. See imwrite() .

The function compresses the image and stores it in the memory buffer that is resized to fit the result. See imwrite() for the list of supported formats and flags description.

Note: cvEncodeImage returns single-row matrix of type CV_8UC1 that contains encoded image as array of bytes.

---

imread

Loads an image from a file.

C++: Mat **imread**(const string& **filename**, int **flags**=1)

Python: cv2.**imread**(filename[, flags]) → retval

C: IplImage* **cvLoadImage**(const char* **filename**, int **flags**=CV_LOAD_IMAGE_COLOR)

C: CvMat* **cvLoadImageM**(const char* **filename**, int **flags**=CV_LOAD_IMAGE_COLOR)

Python: cv.**LoadImage**(filename, flags=CV_LOAD_IMAGE_COLOR) → None

Python: cv.**LoadImageM**(filename, flags=CV_LOAD_IMAGE_COLOR) → None

> **Parameters**
>
> > **filename** – Name of file to be loaded.
> >
> > **flags** – Flags specifying the color type of a loaded image:
> >
> > – **>0** Return a 3-channel color image
> >
> > – **=0** Return a grayscale image
> >
> > – **<0** Return the loaded image as is. Note that in the current implementation the alpha channel, if any, is stripped from the output image. For example, a 4-channel RGBA image is loaded as RGB if $flags \geq 0$.

The function imread loads an image from the specified file and returns it. If the image cannot be read (because of missing file, improper permissions, unsupported or invalid format), the function returns an empty matrix (Mat::data==NULL). Currently, the following file formats are supported:

- Windows bitmaps - *.bmp, *.dib (always supported)
- JPEG files - *.jpeg, *.jpg, *.jpe (see the *Notes* section)
- JPEG 2000 files - *.jp2 (see the *Notes* section)
- Portable Network Graphics - *.png (see the *Notes* section)
- Portable image format - *.pbm, *.pgm, *.ppm (always supported)
- Sun rasters - *.sr, *.ras (always supported)
- TIFF files - *.tiff, *.tif (see the *Notes* section)

Note:

- The function determines the type of an image by the content, not by the file extension.
- On Microsoft Windows* OS and MacOSX*, the codecs shipped with an OpenCV image (libjpeg, libpng, libtiff, and libjasper) are used by default. So, OpenCV can always read JPEGs, PNGs, and TIFFs. On MacOSX, there is also an option to use native MacOSX image readers. But beware that currently these native image loaders give images with different pixel values because of the color management embedded into MacOSX.
- On Linux*, BSD flavors and other Unix-like open-source operating systems, OpenCV looks for codecs supplied with an OS image. Install the relevant packages (do not forget the development files, for example, "libjpeg-dev", in Debian* and Ubuntu*) to get the codec support or turn on the OPENCV_BUILD_3RDPARTY_LIBS flag in CMake.

imwrite

Saves an image to a specified file.

C++: bool **imwrite**(const string& **filename**, InputArray **image**, const vector<int>& **params**=vector<int>())

Python: cv2.**imwrite**(filename, image[, params]) → retval

C: int **cvSaveImage**(const char* **filename**, const CvArr* **image**)

Python: cv.**SaveImage**(filename, image) → None

> **Parameters**

> > **filename** – Name of the file.

> > **image** – Image to be saved.

> > **params** – Format-specific save parameters encoded as pairs `paramId_1`, `paramValue_1`, `paramId_2`, `paramValue_2`, `...` . The following parameters are currently supported:

> > > – For JPEG, it can be a quality (`CV_IMWRITE_JPEG_QUALITY`) from 0 to 100 (the higher is the better). Default value is 95.

> > > – For PNG, it can be the compression level (`CV_IMWRITE_PNG_COMPRESSION`) from 0 to 9. A higher value means a smaller size and longer compression time. Default value is 3.

> > > – For PPM, PGM, or PBM, it can be a binary format flag (`CV_IMWRITE_PXM_BINARY`), 0 or 1. Default value is 1.

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 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 `VideoCapture`

Class for video capturing from video files or cameras. The class provides C++ API for capturing video from cameras or for reading video files. Here is how the class can be used:

```cpp
#include "opencv2/opencv.hpp"

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 VideoCapture destructor
    return 0;
}
```

Note: In C API the black-box structure `CvCapture` is used instead of `VideoCapture`.

VideoCapture::VideoCapture

VideoCapture constructors.

C++: VideoCapture::**VideoCapture**()

C++: VideoCapture::**VideoCapture**(const string& **filename**)

C++: VideoCapture::**VideoCapture**(int **device**)

Python: cv2.**VideoCapture**() → <VideoCapture object>

Python: cv2.**VideoCapture**(filename) → <VideoCapture object>

Python: cv2.**VideoCapture**(device) → <VideoCapture object>

C: CvCapture* **cvCaptureFromCAM**(int **device**)

Python: cv.**CaptureFromCAM**(device) → CvCapture

C: CvCapture* **cvCaptureFromFile**(const char* **filename**)

Python: cv.**CaptureFromFile**(filename) → CvCapture

> **Parameters**
>
>> **filename** – name of the opened video file
>>
>> **device** – id of the opened video capturing device (i.e. a camera index). If there is a single camera connected, just pass 0.

Note: In C API, when you finished working with video, release `CvCapture` structure with `cvReleaseCapture()`, or use `Ptr<CvCapture>` that calls `cvReleaseCapture()` automatically in the destructor.

VideoCapture::open

Open video file or a capturing device for video capturing

C++: bool VideoCapture::**open**(const string& **filename**)

C++: bool VideoCapture::**open**(int **device**)

Python: cv2.VideoCapture.**open**(filename) → successFlag

Python: cv2.VideoCapture.**open**(device) → successFlag

> **Parameters**
>
>> **filename** – name of the opened video file

> **device** – id of the opened video capturing device (i.e. a camera index).

The methods first call `VideoCapture::release()` to close the already opened file or camera.

VideoCapture::isOpened

Returns true if video capturing has been initialized already.

C++: bool VideoCapture::**isOpened**()

Python: cv2.VideoCapture.**isOpened**() → flag

If the previous call to `VideoCapture` constructor or `VideoCapture::open` succeeded, the method returns true.

VideoCapture::release

Closes video file or capturing device.

C++: void VideoCapture::**release**()

Python: cv2.VideoCapture.**release**()

C: void **cvReleaseCapture**(CvCapture** **capture**)

The methods are automatically called by subsequent `VideoCapture::open()` and by `VideoCapture` destructor.

The C function also deallocates memory and clears *`capture` pointer.

VideoCapture::grab

Grabs the next frame from video file or capturing device.

C++: bool VideoCapture::**grab**()

Python: cv2.VideoCapture.**grab**() → successFlag

C: int **cvGrabFrame**(CvCapture* **capture**)

Python: cv.**GrabFrame**(capture) → int

The methods/functions grab the next frame from video file or camera and return true (non-zero) in the case of success.

The primary use of the function is in multi-camera environments, especially when the cameras do not have hardware synchronization. That is, you call `VideoCapture::grab()` for each camera and after that call the slower method `VideoCapture::retrieve()` to decode and get frame from each camera. This way the overhead on demosaicing or motion jpeg decompression etc. is eliminated and the retrieved frames from different cameras will be closer in time.

Also, when a connected camera is multi-head (for example, a stereo camera or a Kinect device), the correct way of retrieving data from it is to call *VideoCapture::grab* first and then call `VideoCapture::retrieve()` one or more times with different values of the `channel` parameter. See https://code.ros.org/svn/opencv/trunk/opencv/samples/cpp/kinect_maps.cpp

VideoCapture::retrieve

Decodes and returns the grabbed video frame.

C++: bool VideoCapture::**retrieve**(Mat& **image**, int **channel**=0)

Python: cv2.VideoCapture.**retrieve**([image[, channel]]) → successFlag, image

---

C: IplImage* **cvRetrieveFrame**(CvCapture* **capture**)

Python: cv.**RetrieveFrame**(capture) → iplimage

The methods/functions decode and retruen the just grabbed frame. If no frames has been grabbed (camera has been disconnected, or there are no more frames in video file), the methods return false and the functions return NULL pointer.

Note: OpenCV 1.x functions cvRetrieveFrame and cv.RetrieveFrame return image stored inside the video capturing structure. It is not allowed to modify or release the image! You can copy the frame using cvCloneImage() and then do whatever you want with the copy.

VideoCapture::read

Grabs, decodes and returns the next video frame.

C++: VideoCapture& VideoCapture::**operator>>**(Mat& **image**)

C++: bool VideoCapture::**read**(Mat& **image**)

Python: cv2.VideoCapture.**read**([image]) → successFlag, image

C: IplImage* **cvQueryFrame**(CvCapture* **capture**)

Python: cv.**QueryFrame**(capture) → iplimage

The methods/functions combine VideoCapture::grab() and VideoCapture::retrieve() in one call. This is the most convenient method for reading video files or capturing data from decode and retruen the just grabbed frame. If no frames has been grabbed (camera has been disconnected, or there are no more frames in video file), the methods return false and the functions return NULL pointer.

Note: OpenCV 1.x functions cvRetrieveFrame and cv.RetrieveFrame return image stored inside the video capturing structure. It is not allowed to modify or release the image! You can copy the frame using cvCloneImage() and then do whatever you want with the copy.

VideoCapture::get

Returns the specified VideoCapture property

C++: double VideoCapture::**get**(int **propId**)

Python: cv2.VideoCapture.**get**(propId) → retval

C: double **cvGetCaptureProperty**(CvCapture* **capture**, int **propId**)

Python: cv.**GetCaptureProperty**(capture, propId) → double

    Parameters

        propId – Property identifier. It can be one of the following:

          – **CV_CAP_PROP_POS_MSEC** Current position of the video file in milliseconds or video capture timestamp.

          – **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next.

- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file: 0 - start of the film, 1 - end of the film.

- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream.

- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream.

- **CV_CAP_PROP_FPS** Frame rate.

- **CV_CAP_PROP_FOURCC** 4-character code of codec.

- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file.

- **CV_CAP_PROP_FORMAT** Format of the Mat objects returned by `retrieve()`.

- **CV_CAP_PROP_MODE** Backend-specific value indicating the current capture mode.

- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras).

- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras).

- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras).

- **CV_CAP_PROP_HUE** Hue of the image (only for cameras).

- **CV_CAP_PROP_GAIN** Gain of the image (only for cameras).

- **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras).

- **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB.

- **CV_CAP_PROP_WHITE_BALANCE** Currently not supported

- **CV_CAP_PROP_RECTIFICATION** Rectification flag for stereo cameras (note: only supported by DC1394 v 2.x backend currently)

Note: When querying a property that is not supported by the backend used by the `VideoCapture` class, value 0 is returned.

VideoCapture::set

Sets a property in the `VideoCapture`.

C++: bool VideoCapture::**set**(int **propertyId**, double **value**)

Python: cv2.VideoCapture.**set**(propId, value) → retval

C: int **cvSetCaptureProperty**(CvCapture* **capture**, int **propId**, double **value**)

Python: cv.**SetCaptureProperty**(capture, propId, value) → None

> **Parameters**
>
>> **propId** – Property identifier. It can be one of the following:
>>
>> - **CV_CAP_PROP_POS_MSEC** Current position of the video file in milliseconds.
>>
>> - **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next.
>>
>> - **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file: 0 - start of the film, 1 - end of the film.
>>
>> - **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream.
>>
>> - **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream.

> – **CV_CAP_PROP_FPS** Frame rate.
>
> – **CV_CAP_PROP_FOURCC** 4-character code of codec.
>
> – **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file.
>
> – **CV_CAP_PROP_FORMAT** Format of the Mat objects returned by `retrieve()`.
>
> – **CV_CAP_PROP_MODE** Backend-specific value indicating the current capture mode.
>
> – **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras).
>
> – **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras).
>
> – **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras).
>
> – **CV_CAP_PROP_HUE** Hue of the image (only for cameras).
>
> – **CV_CAP_PROP_GAIN** Gain of the image (only for cameras).
>
> – **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras).
>
> – **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB.
>
> – **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
>
> – **CV_CAP_PROP_RECTIFICATION** Rectification flag for stereo cameras (note: only supported by DC1394 v 2.x backend currently)
>
> **value** – Value of the property.

VideoWriter

class VideoWriter

Video writer class.

VideoWriter::VideoWriter

VideoWriter constructors

C++: VideoWriter::**VideoWriter**()

C++: VideoWriter::**VideoWriter**(const string& **filename**, int **fourcc**, double **fps**, Size **frameSize**, bool **isColor**=true)

Python: cv2.**VideoWriter**([filename, fourcc, fps, frameSize[, isColor]]) → <VideoWriter object>

C: CvVideoWriter* **cvCreateVideoWriter**(const char* **filename**, int **fourcc**, double **fps**, CvSize **frameSize**, int **isColor**=1)

Python: cv.**CreateVideoWriter**(filename, fourcc, fps, frameSize, isColor) → CvVideoWriter

Python: cv2.VideoWriter.**isOpened**() → retval

Python: cv2.VideoWriter.**open**(filename, fourcc, fps, frameSize[, isColor]) → retval

Python: cv2.VideoWriter.**write**(image) → None

> **Parameters**
>
> > **filename** – Name of the output video file.

> **fourcc** – 4-character code of codec used to compress the frames. For example, CV_FOURCC('P','I','M','1') is a MPEG-1 codec, CV_FOURCC('M','J','P','G') is a motion-jpeg codec etc.
>
> **fps** – Framerate of the created video stream.
>
> **frameSize** – Size of the video frames.
>
> **isColor** – If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only).

The constructors/functions initialize video writers. On Linux FFMPEG is used to write videos; on Windows FFMPEG or VFW is used; on MacOSX QTKit is used.

ReleaseVideoWriter

Releases the AVI writer.

C: void **cvReleaseVideoWriter**(CvVideoWriter** **writer**)

The function should be called after you finished using CvVideoWriter opened with CreateVideoWriter().

VideoWriter::open

Initializes or reinitializes video writer.

C++: bool VideoWriter::**open**(const string& **filename**, int **fourcc**, double **fps**, Size **frameSize**, bool **is-Color**=true)

Python: cv2.VideoWriter.**open**(filename, fourcc, fps, frameSize[, isColor]) → retval

The method opens video writer. Parameters are the same as in the constructor VideoWriter::VideoWriter().

VideoWriter::isOpened

Returns true if video writer has been successfully initialized.

C++: bool VideoWriter::**isOpened**()

Python: cv2.VideoWriter.**isOpened**() → retval

VideoWriter::write

Writes the next video frame

C++: VideoWriter& VideoWriter::**operator<<**(const Mat& **image**)

C++: void VideoWriter::**write**(const Mat& **image**)

Python: cv2.VideoWriter.**write**(image) → None

C: int **cvWriteFrame**(CvVideoWriter* **writer**, const IplImage* **image**)

Python: cv.**WriteFrame**(writer, image) → int
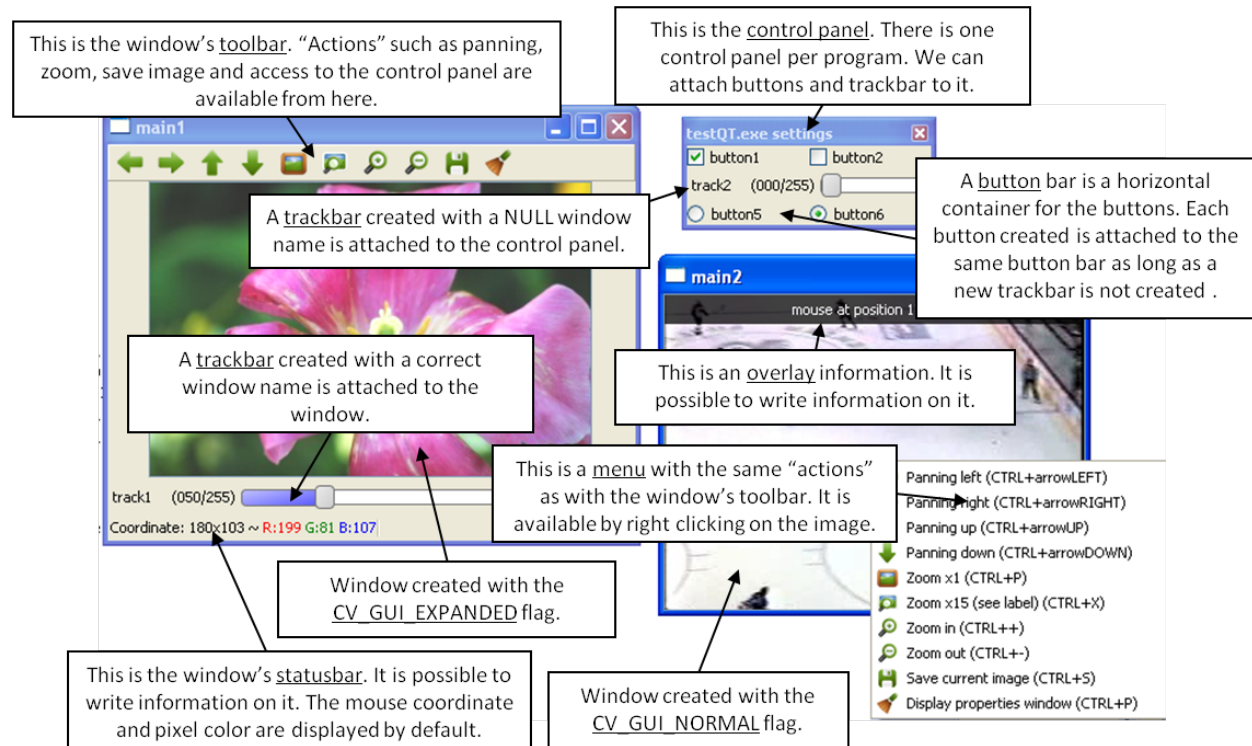
> **Parameters**
>
> > **writer** – Video writer structure (OpenCV 1.x API)
> >
> > **image** – The written frame

The functions/methods write the specified image to video file. It must have the same size as has been specified when opening the video writer.

4.3 Qt New Functions

This figure explains new functionality implemented with Qt* GUI. The new GUI provides a statusbar, a toolbar, and a control panel. The control panel can have trackbars and buttonbars attached to it. If you cannot see the control panel, press Ctrl+P or right-click any Qt window and select **Display properties window**.

- To attach a trackbar, the window name parameter must be NULL.

- To attach a buttonbar, a button must be created. If the last bar attached to the control panel is a buttonbar, the new button is added to the right of the last button. If the last bar attached to the control panel is a trackbar, or the control panel is empty, a new buttonbar is created. Then, a new button is attached to it.

See below the example used to generate the figure:

```
int main(int argc, char *argv[])
    int value = 50;
    int value2 = 0;

    cvNamedWindow("main1",CV_WINDOW_NORMAL);
    cvNamedWindow("main2",CV_WINDOW_AUTOSIZE | CV_GUI_NORMAL);

    cvCreateTrackbar( "track1", "main1", &value, 255,  NULL);//OK tested
    char* nameb1 = "button1";
    char* nameb2 = "button2";
    cvCreateButton(nameb1,callbackButton,nameb1,CV_CHECKBOX,1);

    cvCreateButton(nameb2,callbackButton,nameb2,CV_CHECKBOX,0);
    cvCreateTrackbar( "track2", NULL, &value2, 255, NULL);
```

```
    cvCreateButton("button5",callbackButton1,NULL,CV_RADIOBOX,0);
    cvCreateButton("button6",callbackButton2,NULL,CV_RADIOBOX,1);

    cvSetMouseCallback( "main2",on_mouse,NULL );

    IplImage* img1 = cvLoadImage("files/flower.jpg");
    IplImage* img2 = cvCreateImage(cvGetSize(img1),8,3);
    CvCapture* video = cvCaptureFromFile("files/hockey.avi");
    IplImage* img3 = cvCreateImage(cvGetSize(cvQueryFrame(video)),8,3);

    while(cvWaitKey(33) != 27)
    {
        cvAddS(img1,cvScalarAll(value),img2);
        cvAddS(cvQueryFrame(video),cvScalarAll(value2),img3);
        cvShowImage("main1",img2);
        cvShowImage("main2",img3);
    }

    cvDestroyAllWindows();
    cvReleaseImage(&img1);
    cvReleaseImage(&img2);
    cvReleaseImage(&img3);
    cvReleaseCapture(&video);
    return 0;
}
```

setWindowProperty

Changes parameters of a window dynamically.

C++: void **setWindowProperty**(const string& **name**, int **prop_id**, double **prop_value**)

Python: cv2.**setWindowProperty**(winname, prop_id, prop_value) → None

C: void **cvSetWindowProperty**(const char* **name**, int **propId**, double **propValue**)

> **Parameters**

>> **name** – Name of the window.

>> **prop_id** – Window property to edit. The following operation flags are available:

>>> – **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (`CV_WINDOW_NORMAL` or `CV_WINDOW_FULLSCREEN`).

>>> – **CV_WND_PROP_AUTOSIZE** Change if the window is resizable (`CV_WINDOW_NORMAL` or `CV_WINDOW_AUTOSIZE`).

>>> – **CV_WND_PROP_ASPECTRATIO** Change if the aspect ratio of the image is preserved (`CV_WINDOW_FREERATIO` or `CV_WINDOW_KEEPRATIO`).

>> **prop_value** – New value of the window property. The following operation flags are available:

>>> – **CV_WINDOW_NORMAL** Change the window to normal size or make the window resizable.

>>> – **CV_WINDOW_AUTOSIZE** Constrain the size by the displayed image. The window is not resizable.

>>> – **CV_WINDOW_FULLSCREEN** Change the window to fullscreen.

> > – **CV_WINDOW_FREERATIO** Make the window resizable without any ratio constraints.
> >
> > – **CV_WINDOW_KEEPRATIO** Make the window resizable, but preserve the proportions of the displayed image.

The function `setWindowProperty` enables changing properties of a window.

getWindowProperty

Provides parameters of a window.

C++: void **getWindowProperty**(const string& **name**, int **prop_id**)

Python: cv2.**getWindowProperty**(winname, prop_id) → retval

C: void **cvGetWindowProperty**(const char* **name**, int **propId**)

> **Parameters**
>
> > **name** – Name of the window.
> >
> > **prop_id** – Window property to retrive. The following operation flags are available:
> >
> > – **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (`CV_WINDOW_NORMAL` or `CV_WINDOW_FULLSCREEN`).
> >
> > – **CV_WND_PROP_AUTOSIZE** Change if the window is resizable (`CV_WINDOW_NORMAL` or `CV_WINDOW_AUTOSIZE`).
> >
> > – **CV_WND_PROP_ASPECTRATIO** Change if the aspect ratio of the image is preserved (`CV_WINDOW_FREERATIO` or `CV_WINDOW_KEEPRATIO`).

See `setWindowProperty()` to know the meaning of the returned values.

The function `getWindowProperty` returns properties of a window.

fontQt

Creates the font to draw a text on an image.

C++: CvFont **fontQt**(const string& **nameFont**, int **pointSize**=-1, Scalar **color**=Scalar::all(0), int **weight**=CV_FONT_NORMAL, int **style**=CV_STYLE_NORMAL, int **spacing**=0)

C: CvFont **cvFontQt**(const char* **nameFont**, int **pointSize**=-1, CvScalar **color**=cvScalarAll(0), int **weight**=CV_FONT_NORMAL, int **style**=CV_STYLE_NORMAL, int **spacing**=0)

> **Parameters**
>
> > **nameFont** – Name of the font. The name should match the name of a system font (such as *Times*). If the font is not found, a default one is used.
> >
> > **pointSize** – Size of the font. If not specified, equal zero or negative, the point size of the font is set to a system-dependent default value. Generally, this is 12 points.
> >
> > **color** – Color of the font in BGRA where A = 255 is fully transparent. Use the macro CV _ RGB for simplicity.
> >
> > **weight** – Font weight. The following operation flags are available:
> >
> > – **CV_FONT_LIGHT** Weight of 25
> >
> > – **CV_FONT_NORMAL** Weight of 50

– **CV_FONT_DEMIBOLD** Weight of 63

– **CV_FONT_BOLD** Weight of 75

– **CV_FONT_BLACK** Weight of 87

You can also specify a positive integer for better control.

style – Font style. The following operation flags are available:

– **CV_STYLE_NORMAL** Normal font

– **CV_STYLE_ITALIC** Italic font

– **CV_STYLE_OBLIQUE** Oblique font

spacing – Spacing between characters. It can be negative or positive.

The function `fontQt` creates a `CvFont` object. This `CvFont` is not compatible with `putText` .

A basic usage of this function is the following:

```
CvFont font = fontQt(''Times'');
addText( img1, ''Hello World !'', Point(50,50), font);
```

addText

Creates the font to draw a text on an image.

C++: void **addText**(const Mat& **img**, const string& **text**, Point **location**, CvFont* **font**)

C: void **cvAddText**(const CvArr* **img**, const char* **text**, CvPoint **location**, CvFont* **font**)

> **Parameters**
>
> > **img** – 8-bit 3-channel image where the text should be drawn.
> >
> > **text** – Text to write on an image.
> >
> > **location** – Point(x,y) where the text should start on an image.
> >
> > **font** – Font to use to draw a text.

The function `addText` draws *text* on an image *img* using a specific font *font* (see example `fontQt()`)

displayOverlay

Displays a text on a window image as an overlay for a specified duration.

C++: void **displayOverlay**(const string& **name**, const string& **text**, int **delay**)

C: void **cvDisplayOverlay**(const char* **name**, const char* **text**, int **delay**)

> **Parameters**
>
> > **name** – Name of the window.
> >
> > **text** – Overlay text to write on a window image.
> >
> > **delay** – The period (in milliseconds), during which the overlay text is displayed. If this function is called before the previous overlay text timed out, the timer is restarted and the text is updated. If this value is zero, the text never disappears.

The function `displayOverlay` displays useful information/tips on top of the window for a certain amount of time *delay*. The function does not modify the image, displayed in the window, that is, after the specified delay the original content of the window is restored.

displayStatusBar

Displays a text on the window statusbar during the specified period of time.

C++: void **displayStatusBar**(const string& **name**, const string& **text**, int **delay**)

C: void **cvDisplayStatusBar**(const char* **name**, const char* **text**, int **delayms**)

> **Parameters**
>
> > **name** – Name of the window.
> >
> > **text** – Text to write on the window statusbar.
> >
> > **delay** – Duration (in milliseconds) to display the text. If this function is called before the previous text timed out, the timer is restarted and the text is updated. If this value is zero, the text never disappears.

The function `displayOverlay` displays useful information/tips on top of the window for a certain amount of time *delay* . This information is displayed on the window statubar (the window must be created with the `CV_GUI_EXPANDED` flags).

createOpenGLCallback

Creates a callback function called to draw OpenGL on top the the image display by `windowname`.

C++: void **createOpenGLCallback**(const string& **window_name**, OpenGLCallback **callbackOpenGL**, void* **userdata**=NULL, double **angle**=-1, double **zmin**=-1, double **zmax**=-1)

C: void **cvCreateOpenGLCallback**(const char* **windowName**, CvOpenGLCallback **callbackOpenGL**, void* **userdata**=NULL, double **angle**=-1, double **zmin**=-1, double **zmax**=-1)

> **Parameters**
>
> > **window_name** – Name of the window.
> >
> > **callbackOpenGL** – Pointer to the function to be called every frame. This function should be prototyped as `void Foo(*void);` .
> >
> > **userdata** – Pointer passed to the callback function. *(Optional)*
> >
> > **angle** – Parameter specifying the field of a view angle, in degrees, in the y direction. Default value is 45 degrees. *(Optional)*
> >
> > **zmin** – Parameter specifying the distance from the viewer to the near clipping plane (always positive). Default value is 0.01. *(Optional)*
> >
> > **zmax** – Parameter specifying the distance from the viewer to the far clipping plane (always positive). Default value is 1000. *(Optional)*

The function `createOpenGLCallback` can be used to draw 3D data on the window. See the example of callback use below:

```
void on_opengl(void* param)
{
    glLoadIdentity();

    glTranslated(0.0, 0.0, -1.0);

    glRotatef( 55, 1, 0, 0 );
    glRotatef( 45, 0, 1, 0 );
    glRotatef( 0, 0, 0, 1 );

    static const int coords[6][4][3] = {
        { { +1, -1, -1 }, { -1, -1, -1 }, { -1, +1, -1 }, { +1, +1, -1 } },
        { { +1, +1, -1 }, { -1, +1, -1 }, { -1, +1, +1 }, { +1, +1, +1 } },
        { { +1, -1, +1 }, { +1, -1, -1 }, { +1, +1, -1 }, { +1, +1, +1 } },
        { { -1, -1, -1 }, { -1, -1, +1 }, { -1, +1, +1 }, { -1, +1, -1 } },
        { { +1, -1, +1 }, { -1, -1, +1 }, { -1, -1, -1 }, { +1, -1, -1 } },
        { { -1, -1, +1 }, { +1, -1, +1 }, { +1, +1, +1 }, { -1, +1, +1 } }
    };

    for (int i = 0; i < 6; ++i) {
            glColor3ub( i*20, 100+i*10, i*42 );
            glBegin(GL_QUADS);
            for (int j = 0; j < 4; ++j) {
                    glVertex3d(0.2 * coords[i][j][0], 0.2 * coords[i][j][1], 0.2 * coords[i][j][2]);
            }
            glEnd();
    }
}
```

saveWindowParameters

Saves parameters of the specified window.

C++: void **saveWindowParameters**(const string& **name**)

C: void **cvSaveWindowParameters**(const char* **name**)

> **Parameters**
>
> > **name** – Name of the window.

The function `saveWindowParameters` saves size, location, flags, trackbars value, zoom and panning location of the window `window_name` .

loadWindowParameters

Loads parameters of the specified window.

C++: void **loadWindowParameters**(const string& **name**)

C: void **cvLoadWindowParameters**(const char* **name**)

> **Parameters**
>
> > **name** – Name of the window.

The function `loadWindowParameters` loads size, location, flags, trackbars value, zoom and panning location of the window `window_name` .

createButton

Attaches a button to the control panel.

C++: `createButton`(const string& **button_name**=NULL, ButtonCallback **on_change**=NULL, void* **user-data**=NULL, int **button_type**=CV_PUSH_BUTTON, int **initial_button_state**=0)

C: `cvCreateButton`(const char* **buttonName**=NULL, CvButtonCallback **onChange**=NULL, void* **user-data**=NULL, int **buttonType**=CV_PUSH_BUTTON, int **initialButtonState**=0)

   Parameters

   button_name – Name of the button.

   on_change – Pointer to the function to be called every time the button changes its state. This function should be prototyped as `void Foo(int state,*void);` . *state* is the current state of the button. It could be -1 for a push button, 0 or 1 for a check/radio box button.

   userdata – Pointer passed to the callback function.

   button_type – Optional type of the button.

   – **CV_PUSH_BUTTON** Push button

   – **CV_CHECKBOX** Checkbox button

   – **CV_RADIOBOX** Radiobox button. The radiobox on the same buttonbar (same line) are exclusive, that is only one can be selected at a time.

   initial_button_state – Default state of the button. Use for checkbox and radiobox. Its value could be 0 or 1. *(Optional)*

The function `createButton` attaches a button to the control panel. Each button is added to a buttonbar to the right of the last button. A new buttonbar is created if nothing was attached to the control panel before, or if the last element attached to the control panel was a trackbar.

See below various examples of the `createButton` function call:

```
createButton(NULL,callbackButton);//create a push button "button 0", that will call callbackButton.
createButton("button2",callbackButton,NULL,CV_CHECKBOX,0);
createButton("button3",callbackButton,&value);
createButton("button5",callbackButton1,NULL,CV_RADIOBOX);
createButton("button6",callbackButton2,NULL,CV_PUSH_BUTTON,1);
```

VIDEO. VIDEO ANALYSIS

5.1 Motion Analysis and Object Tracking

calcOpticalFlowPyrLK

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

C++: void **calcOpticalFlowPyrLK**(InputArray **prevImg**, InputArray **nextImg**, InputArray **prevPts**, InputOutputArray **nextPts**, OutputArray **status**, OutputArray **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)

Python: cv2.**calcOpticalFlowPyrLK**(prevImg, nextImg, prevPts[, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, derivLambda[, flags]]]]]]]]) → nextPts, status, err

C: void **cvCalcOpticalFlowPyrLK**(const CvArr* **prev**, const CvArr* **curr**, CvArr* **prevPyr**, CvArr* **currPyr**, const CvPoint2D32f* **prevFeatures**, CvPoint2D32f* **currFeatures**, int **count**, CvSize **winSize**, int **level**, char* **status**, float* **trackError**, CvTermCriteria **criteria**, int **flags**)

Python: cv.**CalcOpticalFlowPyrLK**(prev, curr, prevPyr, currPyr, prevFeatures, winSize, level, criteria, flags, guesses=None) -> (currFeatures, status, trackError)

> Parameters
>
> > **prevImg** – First 8-bit single-channel or 3-channel input image.
> >
> > **nextImg** – Second input image of the same size and the same type as `prevImg` .
> >
> > **prevPts** – Vector of 2D points for which the flow needs to be found. The point coordinates must be single-precision floating-point numbers.
> >
> > **nextPts** – Output vector of 2D points (with single-precision floating-point coordinates) containing the calculated new positions of input features in the second image. When `OPTFLOW_USE_INITIAL_FLOW` flag is passed, the vector must have the same size as in the input.
> >
> > **status** – Output status vector. Each element of the vector is set to 1 if the flow for the corresponding features has been found. Otherwise, it is set to 0.
> >
> > **err** – Output vector that contains 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 set to 0, pyramids are not used (single level). If set to 1, two levels are used, and so on.

criteria – Parameter specifying 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 – Not used.

flags – Operation flags:

– **OPTFLOW_USE_INITIAL_FLOW** Use initial estimations stored in `nextPts` . If the flag is not set, then `prevPts` is copied to `nextPts` and is considered as the initial estimate.

The function implements a sparse iterative version of the Lucas-Kanade optical flow in pyramids. See [Bouguet00].

calcOpticalFlowFarneback

Computes a dense optical flow using the Gunnar Farneback's algorithm.

C++: void **calcOpticalFlowFarneback**(InputArray **prevImg**, InputArray **nextImg**, InputOutputArray **flow**, double **pyrScale**, int **levels**, int **winsize**, int **iterations**, int **polyN**, double **polySigma**, int **flags**)

C: void **cvCalcOpticalFlowFarneback**(const CvArr* **prevImg**, const CvArr* **nextImg**, CvArr* **flow**, double **pyrScale**, int **levels**, int **winsize**, int **iterations**, int **polyN**, double **polySigma**, int **flags**)

Python: cv2.**calcOpticalFlowFarneback**(prevImg, nextImg, pyr_scale, levels, winsize, iterations, poly_n, poly_sigma, flags[, flow]) → flow

> **Parameters**
>
> > **prevImg** – First 8-bit single-channel input image.
> >
> > **nextImg** – Second input image of the same size and the same type as `prevImg` .
> >
> > **flow** – Computed flow image that has the same size as `prevImg` and type `CV_32FC2` .
> >
> > **pyrScale** – Parameter specifying the image scale (<1) to build pyramids for each image. `pyrScale=0.5` means a classical pyramid, where each next layer is twice smaller than the previous one.
> >
> > **levels** – 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** – Averaging window size. Larger values increase the algorithm robustness to image noise and give more chances for fast motion detection, but yield more blurred motion field.
> >
> > **iterations** – Number of iterations the algorithm does at each pyramid level.
> >
> > **polyN** – Size of the pixel neighborhood used to find polynomial expansion in each pixel. 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 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** – Operation flags that can be a combination of the following:
> >
> > – **OPTFLOW_USE_INITIAL_FLOW** Use the input `flow` as an initial flow approximation.

– **OPTFLOW_FARNEBACK_GAUSSIAN** Use the Gaussian `winsize` × `winsize` filter instead of a box filter of the same size for optical flow estimation. Usually, this option gives z more accurate flow than with a box filter, at the cost of lower speed. Normally, `winsize` for a Gaussian window should be set to a larger value to achieve the same level of robustness.

The function finds an optical flow for each `prevImg` pixel using the [Farneback2003] alorithm so that

$$prevImg(y, x) \sim nextImg(y + flow(y, x)[1], x + flow(y, x)[0])$$

CalcOpticalFlowBM

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

Python: cv.**CalcOpticalFlowBM**(prev, curr, blockSize, shiftSize, maxRange, usePrevious, velx, vely) → None

> **Parameters**
>
> > **prev** – First image, 8-bit, single-channel
> >
> > **curr** – Second image, 8-bit, single-channel
> >
> > **blockSize** – Size of basic blocks that are compared
> >
> > **shiftSize** – Block coordinate increments
> >
> > **maxRange** – Size of the scanned neighborhood in pixels around the block
> >
> > **usePrevious** – Flag that specifies whether to use the input velocity as initial approximations or not.
> >
> > **velx** – Horizontal component of the optical flow of
> >
> > $$\left\lfloor \frac{prev\text{->}width - blockSize.width}{shiftSize.width} \right\rfloor \times \left\lfloor \frac{prev\text{->}height - blockSize.height}{shiftSize.height} \right\rfloor$$
> >
> > size, 32-bit floating-point, single-channel
> >
> > **vely** – Vertical component of the optical flow of the same size `velx` , 32-bit floating-point, single-channel

The function calculates the optical flow for overlapped blocks `blockSize.width x blockSize.height` pixels each, thus the velocity fields are smaller than the original images. For every block in `prev` the functions tries to find a similar block in `curr` in some neighborhood of the original block or shifted by (`velx(x0,y0)`, `vely(x0,y0)`) block as has been calculated by previous function call (if `usePrevious=1`)

CalcOpticalFlowHS

Calculates the optical flow for two images using Horn-Schunck algorithm.

Python: cv.**CalcOpticalFlowHS**(prev, curr, usePrevious, velx, vely, lambda, criteria) → None

> **Parameters**
>
> > **prev** – First image, 8-bit, single-channel
> >
> > **curr** – Second image, 8-bit, single-channel
> >
> > **usePrevious** – Flag that specifies whether to use the input velocity as initial approximations or not.

velx – Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely – Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

lambda – Smoothness weight. The larger it is, the smoother optical flow map you get.

criteria – Criteria of termination of velocity computing

The function computes the flow for every pixel of the first input image using the Horn and Schunck algorithm [Horn81]. The function is obsolete. To track sparse features, use `calcOpticalFlowPyrLK()`. To track all the pixels, use `calcOpticalFlowFarneback()`.

CalcOpticalFlowLK

Calculates the optical flow for two images using Lucas-Kanade algorithm.

C: void **cvCalcOpticalFlowLK**(const CvArr* **prev**, const CvArr* **curr**, CvSize **winSize**, CvArr* **velx**, CvArr* **vely**)

Python: cv.**CalcOpticalFlowLK**(prev, curr, winSize, velx, vely) → None

> **Parameters**

>> **prev** – First image, 8-bit, single-channel

>> **curr** – Second image, 8-bit, single-channel

>> **winSize** – Size of the averaging window used for grouping pixels

>> **velx** – Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

>> **vely** – Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

The function computes the flow for every pixel of the first input image using the Lucas and Kanade algorithm [Lucas81]. The function is obsolete. To track sparse features, use `calcOpticalFlowPyrLK()`. To track all the pixels, use `calcOpticalFlowFarneback()`.

estimateRigidTransform

Computes an optimal affine transformation between two 2D point sets.

C++: Mat **estimateRigidTransform**(InputArray **src**, InputArray **dst**, bool **fullAffine**)

Python: cv2.**estimateRigidTransform**(src, dst, fullAffine) → retval

> **Parameters**

>> **src** – First input 2D point set stored in `std::vector` or `Mat`, or an image stored in `Mat`.

>> **dst** – Second input 2D point set of the same size and the same type as A, or another image.

>> **fullAffine** – If true, the function finds an optimal affine transformation with no additional resrictions (6 degrees of freedom). Otherwise, the class of transformations to choose from is limited to combinations of translation, rotation, and uniform scaling (5 degrees of freedom).

The function finds an optimal affine transform *[A|b]* (a `2 x 3` floating-point matrix) that approximates best the affine transformation between:

- Two point sets

- Two raster images. In this case, the function first finds some features in the `src` image and finds the corresponding features in `dst` image. After that, the problem is reduced to the first case.

In case of point sets, the problem is formulated as follows: you need to find a 2x2 matrix A and 2x1 vector b so that:

$$[A^*|b^*] = \arg\min_{[A|b]} \sum_i \|\text{dst}[i] - A\text{src}[i]^T - b\|^2$$

where `src[i]` and `dst[i]` are the i-th points in `src` and `dst`, respectively

$[A|b]$ can be either arbitrary (when `fullAffine=true`) or have a form of

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ -a_{12} & a_{11} & b_2 \end{bmatrix}$$

when `fullAffine=false` .

See Also:

`getAffineTransform()`, `getPerspectiveTransform()`, `findHomography()`

updateMotionHistory

Updates the motion history image by a moving silhouette.

C++: void **updateMotionHistory**(InputArray **silhouette**, InputOutputArray **mhi**, double **timestamp**, double **duration**)

Python: cv2.**updateMotionHistory**(silhouette, mhi, timestamp, duration) → None

C: void **cvUpdateMotionHistory**(const CvArr* **silhouette**, CvArr* **mhi**, double **timestamp**, double **duration**)

Python: cv.**UpdateMotionHistory**(silhouette, mhi, timestamp, duration) → None

> **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 follows:

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

That is, MHI pixels where the motion occurs are set to the current `timestamp` , while the pixels where the motion happened last time a long time ago are cleared.

The function, together with `calcMotionGradient()` and `calcGlobalOrientation()` , implements a motion templates technique described in [Davis97] and [Bradski00]. See also the OpenCV sample `motempl.c` that demonstrates the use of all the motion template functions.

calcMotionGradient

Calculates a gradient orientation of a motion history image.

C++: void **calcMotionGradient**(InputArray **mhi**, OutputArray **mask**, OutputArray **orientation**, double **delta1**, double **delta2**, int **apertureSize**=3)

Python: cv2.**calcMotionGradient**(mhi, delta1, delta2[, mask[, orientation[, apertureSize]]]) → mask, orientation

C: void **cvCalcMotionGradient**(const CvArr* **mhi**, CvArr* **mask**, CvArr* **orientation**, double **delta1**, double **delta2**, int **apertureSize**=3)

Python: cv.**CalcMotionGradient**(mhi, mask, orientation, delta1, delta2, apertureSize=3) → None

> **Parameters**

>> **mhi** – Motion history single-channel floating-point image.

>> **mask** – Output mask image that has the type CV_8UC1 and the same size as mhi . Its non-zero elements mark pixels where the motion gradient data is correct.

>> **orientation** – Output motion gradient orientation image that has the same type and the same size as mhi . Each pixel of the image is a motion orientation, from 0 to 360 degrees.

>> **delta1** – Minimal (or maximal) allowed difference between mhi values within a pixel neighorhood.

>> **delta2** – Maximal (or minimal) allowed difference between mhi values within a pixel neighorhood. That is, the function finds the minimum ($m(x, y)$) and maximum ($M(x, y)$) mhi values over 3×3 neighborhood of each pixel and marks the motion orientation at (x, y) as valid only if

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

>> **apertureSize** – Aperture size of the Sobel() operator.

The function calculates a gradient orientation at each pixel (x, y) as:

$$\texttt{orientation}(x, y) = \arctan \frac{\texttt{dmhi}/\texttt{dy}}{\texttt{dmhi}/\texttt{dx}}$$

In fact, fastAtan2() 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

Calculates a global motion orientation in a selected region.

C++: double **calcGlobalOrientation**(InputArray **orientation**, InputArray **mask**, InputArray **mhi**, double **timestamp**, double **duration**)

Python: cv2.**calcGlobalOrientation**(orientation, mask, mhi, timestamp, duration) → retval

C: double **cvCalcGlobalOrientation**(const CvArr* **orientation**, const CvArr* **mask**, const CvArr* **mhi**, double **timestamp**, double **duration**)

Python: cv.**CalcGlobalOrientation**(orientation, mask, mhi, timestamp, duration) → float

> **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 a region whose direction needs to be calculated.
>
> **mhi** – Motion history image calculated by `updateMotionHistory()`.
>
> **timestamp** – Timestamp passed to `updateMotionHistory()`.
>
> **duration** – Maximum duration of a motion track in milliseconds, passed to `updateMotionHistory()`.

The function calculates an 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 a larger weight and the motion occurred in the past has a smaller weight, as recorded in `mhi`.

segmentMotion

Splits a motion history image into a few parts corresponding to separate independent motions (for example, left hand, right hand).

C++: void **segmentMotion**(InputArray **mhi**, OutputArray **segmask**, vector<Rect>& **boundingRects**, double **timestamp**, double **segThresh**)

Python: cv2.**segmentMotion**(mhi, timestamp, segThresh[, segmask]) → segmask, boundingRects

C: CvSeq* **cvSegmentMotion**(const CvArr* **mhi**, CvArr* **segMask**, CvMemStorage* **storage**, double **timestamp**, double **segThresh**)

Python: cv.**SegmentMotion**(mhi, segMask, storage, timestamp, segThresh) → None

> **Parameters**
>
>> **mhi** – Motion history image.
>>
>> **segmask** – Image where the found mask should be stored, single-channel, 32-bit floating-point.
>>
>> **boundingRects** – Vector containing ROIs of motion connected components.
>>
>> **timestamp** – Current time in milliseconds or other units.
>>
>> **segThresh** – Segmentation threshold that is recommended to be equal to the interval between motion history "steps" or greater.

The function finds all of the motion segments and marks them in `segmask` with individual values (1,2,...). It also computes a vector with ROIs of motion connected components. After that the motion direction for every component can be calculated with `calcGlobalOrientation()` using the extracted mask of the particular component.

CamShift

Finds an object center, size, and orientation.

C++: RotatedRect **CamShift**(InputArray **probImage**, Rect& **window**, TermCriteria **criteria**)

Python: cv2.**CamShift**(probImage, window, criteria) → retval, window

C: int **cvCamShift**(const CvArr* **probImage**, CvRect **window**, CvTermCriteria **criteria**, CvConnectedComp* **comp**, CvBox2D* **box**=NULL)

Python: cv.**CamShift**(probImage, window, criteria)-> (int, comp, box)

> **Parameters**
>
>> **probImage** – 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 algrorithm [Bradski98]. First, it finds an object center using `meanShift()` and then adjusts the window size and finds the optimal rotation. The function returns the rotated rectangle structure that includes the object position, size, and 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

Finds an object on a back projection image.

C++: int **meanShift**(InputArray **probImage**, Rect& **window**, TermCriteria **criteria**)

Python: cv2.**meanShift**(probImage, window, criteria) → retval, window

C: int **cvMeanShift**(const CvArr* **probImage**, CvRect **window**, CvTermCriteria **criteria**, CvConnected-Comp* **comp**)

Python: cv.**MeanShift**(probImage, window, criteria) → comp

>> **Parameters**

>>> **probImage** – Back projection of the object histogram. See `calcBackProject()` for details.

>>> **window** – Initial search window.

>>> **criteria** – Stop criteria for the iterative search algorithm.

The function implements the iterative object search algorithm. It takes the input back projection of an object 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. For example, you can do this by retrieving connected components with `findContours()` , throwing away contours with small area (`contourArea()`), and rendering the remaining contours with `drawContours()` .

KalmanFilter

class **KalmanFilter**
    Kalman filter class.

The class implements a standard Kalman filter http://en.wikipedia.org/wiki/Kalman_filter, [Welch95]. However, you can modify `transitionMatrix`, `controlMatrix`, and `measurementMatrix` to get an extended Kalman filter functionality. See the OpenCV sample `kalman.cpp` .

KalmanFilter::KalmanFilter

The constructors.

C++: KalmanFilter::**KalmanFilter**()

C++: KalmanFilter::**KalmanFilter**(int **dynamParams**, int **measureParams**, int **controlParams**=0, int **type**=CV_32F)

Python: cv2.**KalmanFilter**(dynamParams, measureParams[, controlParams[, type]]) → <KalmanFilter object>

C: CvKalman* **cvCreateKalman**(int **dynamParams**, int **measureParams**, int **controlParams**=0)

Python: cv.**CreateKalman**(dynamParams, measureParams, controlParams=0) → CvKalman
>   The full constructor.

>   **Parameters**

>   >   **dynamParams** – Dimensionality of the state.

>   >   **measureParams** – Dimensionality of the measurement.

>   >   **controlParams** – Dimensionality of the control vector.

>   >   **type** – Type of the created matrices that should be `CV_32F` or `CV_64F`.

Note: In C API when CvKalman* kalmanFilter structure is not needed anymore, it should be released with cvReleaseKalman(&kalmanFilter)

KalmanFilter::init

Re-initializes Kalman filter. The previous content is destroyed.

C++: void KalmanFilter::**init**(int **dynamParams**, int **measureParams**, int **controlParams**=0, int **type**=CV_32F)

>   **Parameters**

>   >   **dynamParams** – Dimensionalityensionality of the state.

>   >   **measureParams** – Dimensionality of the measurement.

>   >   **controlParams** – Dimensionality of the control vector.

>   >   **type** – Type of the created matrices that should be `CV_32F` or `CV_64F`.

KalmanFilter::predict

Computes a predicted state.

C++: const Mat& KalmanFilter::**predict**(const Mat& **control**=Mat())

Python: cv2.KalmanFilter.**predict**([control]) → retval

C: const CvMat* **cvKalmanPredict**(CvKalman* **kalman**, const CvMat* **control**=NULL)

Python: cv.**KalmanPredict**(kalman, control=None) → cvmat

>   **Parameters**

>   >   **control** – The optional input control

KalmanFilter::correct

Updates the predicted state from the measurement.

C++: const Mat& KalmanFilter::**correct**(const Mat& **measurement**)

Python: cv2.KalmanFilter.**correct**(measurement) → retval

C: const CvMat* **cvKalmanCorrect**(CvKalman* **kalman**, const CvMat* **measurement**)

Python: cv.**KalmanCorrect**(kalman, measurement) → cvmat

> **Parameters**
>
> > **control** – The measured system parameters

BackgroundSubtractor

class BackgroundSubtractor

Base class for background/foreground segmentation.

```cpp
class BackgroundSubtractor
{
public:
    virtual ~BackgroundSubtractor();
    virtual void operator()(InputArray image, OutputArray fgmask, double learningRate=0);
    virtual void getBackgroundImage(OutputArray backgroundImage) const;
};
```

The class is only used to define the common interface for the whole family of background/foreground segmentation algorithms.

BackgroundSubtractor::operator()

Computes a foreground mask.

C++: void BackgroundSubtractor::**operator()**(InputArray **image**, OutputArray **fgmask**, double **learningRate**=0)

Python: cv2.BackgroundSubtractor.**apply**(image[, fgmask[, learningRate]]) → fgmask

> **Parameters**
>
> > **image** – Next video frame.
> >
> > **fgmask** – The output foreground mask as an 8-bit binary image.

BackgroundSubtractor::getBackgroundImage

Computes a background image.

C++: void BackgroundSubtractor::**getBackgroundImage**(OutputArray **backgroundImage**) const

> **Parameters**
>
> > **backgroundImage** – The output background image.

Note: Sometimes the background image can be very blurry, as it contain the average background statistics.

BackgroundSubtractorMOG

BackgroundSubtractorMOG : public BackgroundSubtractor

Gaussian Mixture-based Backbround/Foreground Segmentation Algorithm.

The class implements the algorithm described in P. KadewTraKuPong and R. Bowden, *An improved adaptive background mixture model for real-time tracking with shadow detection*, Proc. 2nd European Workshp on Advanced Video-Based Surveillance Systems, 2001: http://personal.ee.surrey.ac.uk/Personal/R.Bowden/publications/avbs01/avbs01.pdf

BackgroundSubtractorMOG::BackgroundSubtractorMOG

The contructors

C++: BackgroundSubtractorMOG::**BackgroundSubtractorMOG**()

C++: BackgroundSubtractorMOG::**BackgroundSubtractorMOG**(int **history**, int **nmixtures**, double **backgroundRatio**, double **noiseSigma**=0)

Python: cv2.**BackgroundSubtractorMOG**(history, nmixtures, backgroundRatio[, noiseSigma]) → <BackgroundSubtractorMOG object>

> **Parameters**

>> **history** – Length of the history.

>> **nmixtures** – Number of Gaussian mixtures.

>> **backgroundRatio** – Background ratio.

>> **noiseSigma** – Noise strength.

Default constructor sets all parameters to default values.

BackgroundSubtractorMOG::operator()

Updates the background model and returns the foreground mask

C++: void BackgroundSubtractorMOG::**operator()**(InputArray **image**, OutputArray **fgmask**, double **learningRate**=0)

Parameters are the same as in BackgroundSubtractor::operator()

BackgroundSubtractorMOG2

Gaussian Mixture-based Backbround/Foreground Segmentation Algorithm.

BackgroundSubtractorMOG2 : public BackgroundSubtractor
> Here are important members of the class that control the algorithm, which you can set after constructing the class instance:

> int **nmixtures**
>> Maximum allowed number of mixture comonents. Actual number is determined dynamically per pixel.

> float **backgroundRatio**
>> Threshold defining whether the component is significant enough to be included into the background model (corresponds to TB=1-cf from the paper??which paper??). cf=0.1 => TB=0.9 is default. For alpha=0.001, it means that the mode should exist for approximately 105 frames before it is considered foreground.

float `varThresholdGen`

> Threshold for the squared Mahalanobis distance that helps decide when a sample is close to the existing components (corresponds to Tg). If it is not close to any component, a new component is generated. 3 sigma => Tg=3*3=9 is default. A smaller Tg value generates more components. A higher Tg value may result in a small number of components but they can grow too large.

float `fVarInit`

> Initial variance for the newly generated components. It affects the speed of adaptation. The parameter value is based on your estimate of the typical standard deviation from the images. OpenCV uses 15 as a reasonable value.

float `fVarMin`

> Parameter used to further control the variance.

float `fVarMax`

> Parameter used to further control the variance.

float `fCT`

> Complexity reduction parameter. This parameter defines the number of samples needed to accept to prove the component exists. CT=0.05 is a default value for all the samples. By setting CT=0 you get an algorithm very similar to the standard Stauffer&Grimson algorithm.

uchar `nShadowDetection`

> The value for marking shadow pixels in the output foreground mask. Default value is 127.

float `fTau`

> Shadow threshold. The shadow is detected if the pixel is a darker version of the background. Tau is a threshold defining how much darker the shadow can be. Tau= 0.5 means that if a pixel is more than twice darker then it is not shadow. See Prati,Mikic,Trivedi,Cucchiarra, *Detecting Moving Shadows...*, IEEE PAMI,2003.

The class implements the Gaussian mixture model background subtraction described in:

- Z.Zivkovic, *Improved adaptive Gausian mixture model for background subtraction*, International Conference Pattern Recognition, UK, August, 2004, http://www.zoranz.net/Publications/zivkovic2004ICPR.pdf. The code is very fast and performs also shadow detection. Number of Gausssian components is adapted per pixel.

- Z.Zivkovic, F. van der Heijden, *Efficient Adaptive Density Estimapion per Image Pixel for the Task of Background Subtraction*, Pattern Recognition Letters, vol. 27, no. 7, pages 773-780, 2006. The algorithm similar to the standard Stauffer&Grimson algorithm with additional selection of the number of the Gaussian components based on: Z.Zivkovic, F.van der Heijden, Recursive unsupervised learning of finite mixture models, IEEE Trans. on Pattern Analysis and Machine Intelligence, vol.26, no.5, pages 651-656, 2004.

BackgroundSubtractorMOG2::BackgroundSubtractorMOG2

The constructors.

C++: BackgroundSubtractorMOG2::**BackgroundSubtractorMOG2**()

C++: BackgroundSubtractorMOG2::**BackgroundSubtractorMOG2**(int **history**, float **varThreshold**, bool **bShadowDetection**=1)

> **Parameters**
>
> > **history** – Length of the history.
> >
> > **varThreshold** – Threshold on the squared Mahalanobis distance to decide whether it is well described by the background model (see Cthr??). This parameter does not affect the background update. A typical value could be 4 sigma, that is, varThreshold=4*4=16; (see Tb??).

> **bShadowDetection** – Parameter defining whether shadow detection should be enabled (`true` or `false`).

BackgroundSubtractorMOG2::operator()

Updates the background model and computes the foreground mask

C++: void BackgroundSubtractorMOG2::**operator()**(InputArray **image**, OutputArray **fgmask**, double **learningRate**=-1)

See `BackgroundSubtractor::operator()`.

BackgroundSubtractorMOG2::getBackgroundImage

Returns background image

C++: void BackgroundSubtractorMOG2::**getBackgroundImage**(OutputArray **backgroundImage**)

See `BackgroundSubtractor::getBackgroundImage()`.

CALIB3D. CAMERA CALIBRATION AND 3D RECONSTRUCTION

6.1 Camera Calibration and 3D Reconstruction

The functions in this section use a so-called pinhole camera model. In this model, 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 a camera matrix, or a matrix of intrinsic parameters
- (cx, cy) is a principal point that is usually at the image center
- fx, fy are the focal lengths expressed in pixel-related units

Thus, if an image from the camera is scaled by a 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. So, once estimated, it 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 a still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to a coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

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

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

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

k_1, k_2, k_3, k_4, k_5, and k_6 are radial distortion coefficients. p_1 and p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$$

vector. That is, if the vector contains four 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. If, for example, a camera has been calibrated on images of 320 x 240 resolution, absolutely the same distortion coefficients can be used for 640 x 480 images 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 do the following:

- 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 (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

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

C++: double **calibrateCamera**(InputArrayOfArrays **objectPoints**, InputArrayOfArrays **imagePoints**, Size **imageSize**, InputOutputArray **cameraMatrix**, InputOutputArray **distCoeffs**, OutputArrayOfArrays **rvecs**, OutputArrayOfArrays **tvecs**, int **flags**=0)

Python: cv2.**calibrateCamera**(objectPoints, imagePoints, imageSize[, cameraMatrix[, distCoeffs[, rvecs[, tvecs[, flags]]]]]) → retval, cameraMatrix, distCoeffs, rvecs, tvecs

C: double **cvCalibrateCamera2**(const CvMat* **objectPoints**, const CvMat* **imagePoints**, const CvMat* **pointCounts**, CvSize **imageSize**, CvMat* **cameraMatrix**, CvMat* **distCoeffs**, CvMat* **rvecs**=NULL, CvMat* **tvecs**=NULL, int **flags**=0)

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

      Parameters

objectPoints – In the new interface it is a vector of vectors of calibration pattern points in the calibration pattern coordinate space. The outer vector contains as many elements as the number of the pattern views. If the same calibration pattern is shown in each view and it is fully visible, 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 a pattern coordinate system, then, if the rig is planar, it may make sense to put the model to a XY coordinate plane so that Z-coordinate of each input object point is 0.

In the old interface all the vectors of object points from different views are concatenated together.

imagePoints – In the new interface it is a vector of vectors of the projections of calibration pattern points. `imagePoints.size()` and `objectPoints.size()` and `imagePoints[i].size()` must be equal to `objectPoints[i].size()` for each `i`.

In the old interface all the vectors of object points from different views are concatenated together.

pointCounts – In the old interface this is a vector of integers, containing as many elements, as the number of views of the calibration pattern. Each element is the number of points in each view. Usually, all the elements are the same and equal to the number of feature points on the calibration pattern.

imageSize – Size of the image used only to initialize the intrinsic camera matrix.

cameraMatrix – Output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx`, `fy`, `cx`, `cy` must be initialized before calling the function.

distCoeffs – Output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements.

rvecs – 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, that is, a real position of the calibration pattern in the k-th pattern view (k=0.. M -1).

tvecs – Output vector of translation vectors estimated for each pattern view.

flags – Different flags that may be zero or a combination of the following values:

- **CV_CALIB_USE_INTRINSIC_GUESS** `cameraMatrix` contains valid initial values of `fx`, `fy`, `cx`, `cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`imageSize` is used), and focal distances are computed in a least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate 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 a different location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.

- **CV_CALIB_FIX_ASPECT_RATIO** The functions considers only `fy` as a free parameter. The ratio `fx/fy` stays the same as in the input `cameraMatrix` . When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

– **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients (p_1, p_2) are set to zeros and stay zero.

– **CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6** The corresponding radial distortion coefficient is not changed during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used. Otherwise, it is set to 0.

– **CV_CALIB_RATIONAL_MODEL** Coefficients k4, k5, and k6 are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function computes and returns only 5 distortion coefficients.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The algorithm is based on [Zhang2000] and [BouguetMCT]. The coordinates of 3D object points and their corresponding 2D projections in each view must be specified. That may be achieved by using an object with a 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 zeros). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm performs the following steps:

1. Compute the initial intrinsic parameters (the option only available for planar calibration patterns) or read them from the input parameters. The distortion coefficients are all set to zeros initially unless some of `CV_CALIB_FIX_K?` are specified.

2. Estimate the initial camera pose as if the intrinsic parameters have been already known. This is done using `solvePnP()` .

3. Run the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error, that is, 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()` for details.

The function returns the final re-projection error.

Note: If you use a non-square (=non-NxN) grid and `findChessboardCorners()` for calibration, and `calibrateCamera` returns bad values (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 have probably used `patternSize=cvSize(rows,cols)` instead of using `patternSize=cvSize(cols,rows)` in `findChessboardCorners()` .

See Also:

`findChessboardCorners()`, `solvePnP()`, `initCameraMatrix2D()`, `stereoCalibrate()`, `undistort()`

calibrationMatrixValues

Computes useful camera characteristics from the camera matrix.

C++: void **calibrationMatrixValues**(InputArray **cameraMatrix**, Size **imageSize**, double **apertureWidth**, double **apertureHeight**, double& **fovx**, double& **fovy**, double& **focalLength**, Point2d& **principalPoint**, double& **aspectRatio**)

Python: cv2.**calibrationMatrixValues**(cameraMatrix, imageSize, apertureWidth, apertureHeight) → fovx, fovy, focalLength, principalPoint, aspectRatio

> **Parameters**
>
>> **cameraMatrix** – Input camera matrix that can be estimated by `calibrateCamera()` or `stereoCalibrate()`.
>>
>> **imageSize** – Input image size in pixels.
>>
>> **apertureWidth** – Physical width of the sensor.
>>
>> **apertureHeight** – Physical height of the sensor.
>>
>> **fovx** – Output field of view in degrees along the horizontal sensor axis.
>>
>> **fovy** – Output field of view in degrees along the vertical sensor axis.
>>
>> **focalLength** – Focal length of the lens in mm.
>>
>> **principalPoint** – Principal point in pixels.
>>
>> **aspectRatio** – f_y/f_x

The function computes various useful camera characteristics from the previously estimated camera matrix.

composeRT

Combines two rotation-and-shift transformations.

C++: void **composeRT**(InputArray **rvec1**, InputArray **tvec1**, InputArray **rvec2**, InputArray **tvec2**, OutputArray **rvec3**, OutputArray **tvec3**, OutputArray **dr3dr1**=noArray(), OutputArray **dr3dt1**=noArray(), OutputArray **dr3dr2**=noArray(), OutputArray **dr3dt2**=noArray(), OutputArray **dt3dr1**=noArray(), OutputArray **dt3dt1**=noArray(), OutputArray **dt3dr2**=noArray(), OutputArray **dt3dt2**=noArray())

Python: cv2.**composeRT**(rvec1, tvec1, rvec2, tvec2[, rvec3[, tvec3[, dr3dr1[, dr3dt1[, dr3dr2[, dr3dt2[, dt3dr1[, dt3dt1[, dt3dr2[, dt3dt2]]]]]]]]]]) → rvec3, tvec3, dr3dr1, dr3dt1, dr3dr2, dr3dt2, dt3dr1, dt3dt1, dt3dr2, dt3dt2

> **Parameters**
>
>> **rvec1** – First rotation vector.
>>
>> **tvec1** – First translation vector.
>>
>> **rvec2** – Second rotation vector.
>>
>> **tvec2** – Second translation vector.
>>
>> **rvec3** – Output rotation vector of the superposition.
>>
>> **tvec3** – Output translation vector of the superposition.
>>
>> **d*d*** – Optional output derivatives of `rvec3` or `tvec3` with regard to `rvec1`, `rvec2`, `tvec1` and `tvec2`, respectively.

The functions compute:

$$
\begin{aligned}
\texttt{rvec3} &= \mathrm{rodrigues}^{-1}\left(\mathrm{rodrigues}(\texttt{rvec2}) \cdot \mathrm{rodrigues}(\texttt{rvec1})\right) \\
\texttt{tvec3} &= \mathrm{rodrigues}(\texttt{rvec2}) \cdot \texttt{tvec1} + \texttt{tvec2}
\end{aligned}
,
$$

where rodrigues denotes a rotation vector to a rotation matrix transformation, and rodrigues^{-1} denotes the inverse transformation. See `Rodrigues()` for details.

Also, the functions can compute the derivatives of the output vectors with regards to 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 a matrix multiplication.

computeCorrespondEpilines

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

C++: void **computeCorrespondEpilines**(InputArray **points**, int **whichImage**, InputArray **F**, OutputArray **lines**)

C: void **cvComputeCorrespondEpilines**(const CvMat* **points**, int **whichImage**, const CvMat* **F**, CvMat* **lines**)

Python: cv.**ComputeCorrespondEpilines**(points, whichImage, F, lines) → None

> **Parameters**
>
> > **points** – 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** – Fundamental matrix that can be estimated using `findFundamentalMat()` or `stereoRectify()` .
> >
> > **lines** – Output vector of the epipolar lines corresponding to the points 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 (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 so that $a_i^2 + b_i^2 = 1$.

convertPointsToHomogeneous

Converts points from Euclidean to homogeneous space.

C++: void **convertPointsToHomogeneous**(InputArray **src**, OutputArray **dst**)

Python: cv2.**convertPointsToHomogeneous**(src[, dst]) → dst

> **Parameters**
>
> > **src** – Input vector of N-dimensional points.
> >
> > **dst** – Output vector of N+1-dimensional points.

The function converts points from Euclidean to homogeneous space by appending 1's to the tuple of point coordinates. That is, each point (x1, x2, ..., xn) is converted to (x1, x2, ..., xn, 1).

convertPointsFromHomogeneous

Converts points from homogeneous to Euclidean space.

C++: void **convertPointsFromHomogeneous**(InputArray **src**, OutputArray **dst**)

Python: cv2.**convertPointsFromHomogeneous**(src[, dst]) → dst

>   **Parameters**
>
>>   **src** – Input vector of N-dimensional points.
>>
>>   **dst** – Output vector of N-1-dimensional points.

The function converts points homogeneous to Euclidean space using perspective projection. That is, each point (x1, x2, ... x(n-1), xn) is converted to (x1/xn, x2/xn, ..., x(n-1)/xn). When xn=0, the output point coordinates will be (0,0,0,...).

convertPointsHomogeneous

Converts points to/from homogeneous coordinates.

C++: void **convertPointsHomogeneous**(InputArray **src**, OutputArray **dst**)

Python: cv2.**convertPointsHomogeneous**(src[, dst]) → dst

C: void **cvConvertPointsHomogeneous**(const CvMat* **src**, CvMat* **dst**)

Python: cv.**ConvertPointsHomogeneous**(src, dst) → None

>   **Parameters**
>
>>   **src** – Input array or vector of 2D, 3D, or 4D points.
>>
>>   **dst** – Output vector of 2D, 3D, or 4D points.

The function converts 2D or 3D points from/to homogeneous coordinates by calling either convertPointsToHomogeneous() or convertPointsFromHomogeneous().

Note: The function is obsolete. Use one of the previous two functions instead.

---

decomposeProjectionMatrix

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

C++: void **decomposeProjectionMatrix**(InputArray **projMatrix**, OutputArray **cameraMatrix**, OutputArray **rotMatrix**, OutputArray **transVect**, OutputArray **rotMatrixX**=noArray(), OutputArray **rotMatrixY**=noArray(), OutputArray **rotMatrixZ**=noArray(), OutputArray **eulerAngles**=noArray())

Python: cv2.**decomposeProjectionMatrix**(projMatrix[, cameraMatrix[, rotMatrix[, transVect[, rotMatrixX[, rotMatrixY[, rotMatrixZ[, eulerAngles]]]]]]]) → cameraMatrix, rotMatrix, transVect, rotMatrixX, rotMatrixY, rotMatrixZ, eulerAngles

C: void **cvDecomposeProjectionMatrix**(const CvMat* **projMatrix**, CvMat* **cameraMatrix**, CvMat* **rotMatrix**, CvMat* **transVect**, CvMat* **rotMatrX**=NULL, CvMat* **rotMatrY**=NULL, CvMat* **rotMatrZ**=NULL, CvPoint3D64f* **eulerAngles**=NULL)

Python: cv.**DecomposeProjectionMatrix**(projMatrix, cameraMatrix, rotMatrix, transVect, rotMa-
trX=None, rotMatrY=None, rotMatrZ=None) → eulerAngles

> **Parameters**
>
>> **projMatrix** – 3x4 input projection matrix P.
>>
>> **cameraMatrix** – Output 3x3 camera matrix K.
>>
>> **rotMatrix** – Output 3x3 external rotation matrix R.
>>
>> **transVect** – 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 three-element vector containing 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 a camera.

It optionally returns three rotation matrices, one for each axis, and three Euler angles that could be used in OpenGL.

The function is based on `RQDecomp3x3()` .

drawChessboardCorners

Renders the detected chessboard corners.

C++: void **drawChessboardCorners**(InputOutputArray **image**, Size **patternSize**, InputArray **corners**, bool
patternWasFound)

Python: cv2.**drawChessboardCorners**(image, patternSize, corners, patternWasFound) → None

C: void **cvDrawChessboardCorners**(CvArr* **image**, CvSize **patternSize**, CvPoint2D32f* **corners**, int
count, int **patternWasFound**)

Python: cv.**DrawChessboardCorners**(image, patternSize, corners, patternWasFound) → None

> **Parameters**
>
>> **image** – Destination image. It must be an 8-bit color image.
>>
>> **patternSize** – Number of inner corners per a chessboard row and column (`patternSize = cv::Size(points_per_row,points_per_column)`).
>>
>> **corners** – Array of detected corners, the output of `findChessboardCorners`.
>>
>> **patternWasFound** – Parameter indicating whether the complete board was found or not. The return value of `findChessboardCorners()` should be passed here.

The function draws individual chessboard corners detected either as red circles if the board was not found, or as colored corners connected with lines if the board was found.

findChessboardCorners

Finds the positions of internal corners of the chessboard.

C++: bool **findChessboardCorners**(InputArray **image**, Size **patternSize**, OutputArray **corners**, int
flags=CV_CALIB_CB_ADAPTIVE_THRESH+CV_CALIB_CB_NORMALIZE_IMAGE
)

Python: cv2.**findChessboardCorners**(image, patternSize[, corners[, flags]]) → retval, corners

C: int **cvFindChessboardCorners**(const void* **image**, CvSize **patternSize**, Cv-Point2D32f* **corners**, int* **cornerCount**=NULL, int **flags**=CV_CALIB_CB_ADAPTIVE_THRESH)

Python: cv.**FindChessboardCorners**(image, patternSize, flags=CV_CALIB_CB_ADAPTIVE_THRESH) → corners

> Parameters

>> **image** – Source chessboard view. It must be an 8-bit grayscale or color image.

>> **patternSize** – Number of inner corners per a chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows)).

>> **corners** – Output array of detected corners.

>> **flags** – Various operation flags that can be zero 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 extracted at the contour retrieval stage.

>>> – **CALIB_CB_FAST_CHECK** Run a fast check on the image that looks for chessboard corners, and shortcut the call if none is found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners are found and they are 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 detected coordinates are approximate, and to determine their positions more accurately, the function calls cornerSubPix(). You also may use the function cornerSubPix() with different parameters if returned coordinates are not accurate enough.

Sample usage of detecting and drawing chessboard corners:

```
Size patternsize(8,6); //interior number of corners
Mat gray = ....; //source image
vector<Point2f> corners; //this will be filled by the detected corners

//CALIB_CB_FAST_CHECK saves a lot of time on images
//that do not contain any chessboard corners
bool patternfound = findChessboardCorners(gray, patternsize, corners,
        CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE
        + CALIB_CB_FAST_CHECK);

if(patternfound)
  cornerSubPix(gray, corners, Size(11, 11), Size(-1, -1),
    TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

drawChessboardCorners(img, patternsize, Mat(corners), patternfound);
```

Note: The function requires white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environments. Otherwise, if there is no border and the background is dark, the

outer black squares cannot be segmented properly and so the square grouping and ordering algorithm fails.

findCirclesGrid

Finds the centers in the grid of circles.

C++: bool **findCirclesGrid**(InputArray **image**, Size **patternSize**, OutputArray **centers**, int **flags**=CALIB_CB_SYMMETRIC_GRID, const Ptr<FeatureDetector>& **blobDetector**=new SimpleBlobDetector())

Python: cv2.**findCirclesGridDefault**(image, patternSize[, centers[, flags]]) → centers

> **Parameters**
>
> > **image** – Grid view of source circles. It must be an 8-bit grayscale or color image.
> >
> > **patternSize** – Number of circles per a grid row and column (patternSize = Size(points_per_row, points_per_colum)).
> >
> > **centers** – Output array of detected centers.
> >
> > **flags** – Various operation flags that can be one of the following values:
> >
> > – **CALIB_CB_SYMMETRIC_GRID** Use symmetric pattern of circles.
> >
> > – **CALIB_CB_ASYMMETRIC_GRID** Use asymmetric pattern of circles.
> >
> > – **CALIB_CB_CLUSTERING** Use a special algorithm for grid detection. It is more robust to perspective distortions but much more sensitive to background clutter.
> >
> > **blobDetector** – FeatureDetector that finds blobs like dark circles on light background

The function attempts to determine whether the input image contains a grid of circles. If it is, the function locates centers of the circles. The function returns a non-zero value if all of the centers 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.

Sample usage of detecting and drawing the centers of circles:

```
Size patternsize(7,7); //number of centers
Mat gray = ....; //source image
vector<Point2f> centers; //this will be filled by the detected centers

bool patternfound = findCirclesGrid(gray, patternsize, centers);

drawChessboardCorners(img, patternsize, Mat(centers), patternfound);
```

Note: The function requires white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environments.

solvePnP

Finds an object pose from 3D-2D point correspondences.

C++: void **solvePnP**(InputArray **objectPoints**, InputArray **imagePoints**, InputArray **cameraMatrix**, InputArray **distCoeffs**, OutputArray **rvec**, OutputArray **tvec**, bool **useExtrinsicGuess**=false, int **flags**=CV_ITERATIVE)

Python: cv2.**solvePnP**(objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, useExtrinsicGuess[, flags]]]]) → rvec, tvec

C: void **cvFindExtrinsicCameraParams2**(const CvMat* **objectPoints**, const CvMat* **imagePoints**, const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, CvMat* **rvec**, CvMat* **tvec**, int **useExtrinsicGuess**=0)

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

> Parameters

>> **objectPoints** – Array of object points in the object coordinate space, 3xN/Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. `vector<Point3f>` can be also passed here.

>> **imagePoints** – Array of corresponding image points, 2xN/Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. `vector<Point2f>` can be also passed here.

>> **cameraMatrix** – Input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$.

>> **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

>> **rvec** – Output rotation vector (see `Rodrigues()`) that, together with `tvec` , brings points from the model coordinate system to the camera coordinate system.

>> **tvec** – Output translation vector.

>> **useExtrinsicGuess** – If true (1), the function uses the provided `rvec` and `tvec` values as initial approximations of the rotation and translation vectors, respectively, and further optimizes them.

>> **flags** – Method for solving a PnP problem:

>> – **CV_ITERATIVE** Iterative method is based on Levenberg-Marquardt optimization. In this case the function finds such a pose that minimizes reprojection error, that is the sum of squared distances between the observed projections `imagePoints` and the projected (using `projectPoints()`) `objectPoints` .

>> – **CV_P3P** Method is based on the paper of X.S. Gao, X.-R. Hou, J. Tang, H.-F. Chang "Complete Solution Classification for the Perspective-Three-Point Problem". In this case the function requires exactly four object and image points.

>> – **CV_EPNP** Method has been introduced by F.Moreno-Noguer, V.Lepetit and P.Fua in the paper "EPnP: Efficient Perspective-n-Point Camera Pose Estimation".

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.

solvePnPRansac

Finds an object pose from 3D-2D point correspondences using the RANSAC scheme.

C++: void **solvePnPRansac**(InputArray **objectPoints**, InputArray **imagePoints**, InputArray **cameraMatrix**, InputArray **distCoeffs**, OutputArray **rvec**, OutputArray **tvec**, bool **useExtrinsicGuess**=false, int **iterationsCount**=100, float **reprojectionError**=8.0, int **minInliersCount**=100, OutputArray **inliers**=noArray(), int **flags**=CV_ITERATIVE)

Python: cv2.**solvePnPRansac**(objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, use-ExtrinsicGuess[, iterationsCount[, reprojectionError[, minInliersCount[, inliers[, flags]]]]]]]]) → rvec, tvec, inliers

> Parameters

> > **objectPoints** – Array of object points in the object coordinate space, 3xN/Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3f> can be also passed here.

> > **imagePoints** – Array of corresponding image points, 2xN/Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2f> can be also passed here.

> > **cameraMatrix** – Input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$.

> > **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

> > **rvec** – Output rotation vector (see Rodrigues()) that, together with tvec , brings points from the model coordinate system to the camera coordinate system.

> > **tvec** – Output translation vector.

> > **useExtrinsicGuess** – If true (1), the function uses the provided rvec and tvec values as initial approximations of the rotation and translation vectors, respectively, and further optimizes them.

> > **iterationsCount** – Number of iterations.

> > **reprojectionError** – Inlier threshold value used by the RANSAC procedure. The parameter value is the maximum allowed distance between the observed and computed point projections to consider it an inlier.

> > **minInliersCount** – Number of inliers. If the algorithm at some stage finds more inliers than minInliersCount , it finishes.

> > **inliers** – Output vector that contains indices of inliers in objectPoints and imagePoints .

> > **flags** – Method for solving a PnP problem (see solvePnP()).

The function estimates an 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, that is, the sum of squared distances between the observed projections imagePoints and the projected (using projectPoints()) objectPoints. The use of RANSAC makes the function resistant to outliers.

findFundamentalMat

Calculates a fundamental matrix from the corresponding points in two images.

C++: Mat **findFundamentalMat**(InputArray **points1**, InputArray **points2**, int **method**=FM_RANSAC, double **param1**=3., double **param2**=0.99, OutputArray **mask**=noArray())

Python: cv2.**findFundamentalMat**(points1, points2[, method[, param1[, param2[, mask]]]]) → retval, mask

C: int **cvFindFundamentalMat**(const CvMat* **points1**, const CvMat* **points2**, CvMat* **fundamentalMatrix**, int **method**=CV_FM_RANSAC, double **param1**=1., double **param2**=0.99, CvMat* **status**=NULL)

Python: cv.**FindFundamentalMat**(points1, points2, fundamentalMatrix, method=CV_FM_RANSAC, param1=1., param2=0.99, status=None) → None

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 a 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 – Parameter used for RANSAC. It is the maximum distance from a point to an 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 – Parameter used for the RANSAC or LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.

status – 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 the 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 a 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 one matrix is found. But in case of the 7-point algorithm, the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to computeCorrespondEpilines() that finds the epipolar lines corresponding to the specified points. It can also be passed to stereoRectifyUncalibrated() to compute the rectification transformation.

```
// Example. Estimation of fundamental matrix using the RANSAC algorithm
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

Finds a perspective transformation between two planes.

C++: Mat **findHomography**(InputArray **srcPoints**, InputArray **dstPoints**, int **method**=0, double **ransacReprojThreshold**=3, OutputArray **mask**=noArray())

Python: cv2.**findHomography**(srcPoints, dstPoints[, method[, ransacReprojThreshold[, mask]]]) → retval, mask

C: void **cvFindHomography**(const CvMat* **srcPoints**, const CvMat* **dstPoints**, CvMat* **H**, int **method**=0, double **ransacReprojThreshold**=3, CvMat* **status**=NULL)

Python: cv.**FindHomography**(srcPoints, dstPoints, H, method, ransacReprojThreshold=3.0, status=None) → None

> **Parameters**
>
>> **srcPoints** – Coordinates of the points in the original plane, a matrix of the type `CV_32FC2` or `vector<Point2f>` .
>>
>> **dstPoints** – Coordinates of the points in the target plane, a matrix of the type `CV_32FC2` or a `vector<Point2f>` .
>>
>> **method** – Method used to computed a homography matrix. The following methods are possible:
>>
>> – **0** - a regular method using all the points
>>
>> – **CV_RANSAC** - RANSAC-based robust method
>>
>> – **CV_LMEDS** - Least-Median robust method
>>
>> **ransacReprojThreshold** – Maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if
>>
>> $$\|\texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(\texttt{H} * \texttt{srcPoints}_i)\| > \texttt{ransacReprojThreshold}$$
>>
>> then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range of 1 to 10.
>>
>> **status** – 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 an initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($\texttt{srcPoints}_i$,:math:*dstPoints_i*) fit the rigid perspective transformation (that is, there are some outliers), this initial estimate will be poor. In this case, you can use one of the two robust methods. Both methods, `RANSAC` and `LMeDS` , try many different random subsets of the corresponding point pairs (of four 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 case of a robust method) with the Levenberg-Marquardt method to reduce the re-projection error even more.

The method `RANSAC` can handle practically any ratio of outliers but it needs a 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 there are no outliers and the noise is rather small, use the default method (`method=0`).

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(), estimateRigidTransform(), warpPerspective(), perspectiveTransform()

estimateAffine3D

Computes an optimal affine transformation between two 3D point sets.

C++: int **estimateAffine3D**(InputArray **srcpt**, InputArray **dstpt**, OutputArray **out**, OutputArray **inliers**, double **ransacThreshold**=3.0, double **confidence**=0.99)

Python: cv2.**estimateAffine3D**(srcpt, dstpt[, out[, inliers[, ransacThreshold[, confidence]]]]) → retval, out, inliers

> **Parameters**
>
> > **srcpt** – First input 3D point set.
> >
> > **dstpt** – Second input 3D point set.
> >
> > **out** – Output 3D affine transformation matrix 3×4 .
> >
> > **inliers** – Output vector indicating which points are inliers.
> >
> > **ransacThreshold** – Maximum reprojection error in the RANSAC algorithm to consider a point as an inlier.
> >
> > **confidence** – Confidence level, between 0 and 1, for the estimated transformation. Anything between 0.95 and 0.99 is usually good enough. Values too close to 1 can slow down the estimation significantly. Values lower than 0.8-0.9 can result in an incorrectly estimated transformation.

The function estimates an optimal 3D affine transformation between two 3D point sets using the RANSAC algorithm.

filterSpeckles

Filters off small noise blobs (speckles) in the disparity map

C++: void **filterSpeckles**(InputOutputArray **img**, double **newVal**, int **maxSpeckleSize**, double **maxDiff**, InputOutputArray **buf**=noArray())

Python: cv2.**filterSpeckles**(img, newVal, maxSpeckleSize, maxDiff[, buf]) → None

> **Parameters**
>
> > **img** – The input 16-bit signed disparity image

newVal – The disparity value used to paint-off the speckles

maxSpeckleSize – The maximum speckle size to consider it a speckle. Larger blobs are not affected by the algorithm

maxDiff – Maximum difference between neighbor disparity pixels to put them into the same blob. Note that since StereoBM, StereoSGBM and may be other algorithms return a fixed-point disparity map, where disparity values are multiplied by 16, this scale factor should be taken into account when specifying this parameter value.

buf – The optional temporary buffer to avoid memory allocation within the function.

getOptimalNewCameraMatrix

Returns the new camera matrix based on the free scaling parameter.

C++: Mat **getOptimalNewCameraMatrix**(InputArray **cameraMatrix**, InputArray **distCoeffs**, Size **imageSize**, double **alpha**, Size **newImageSize**=Size(), Rect* **validPixROI**=0, bool **centerPrincipalPoint**=false)

Python: cv2.**getOptimalNewCameraMatrix**(cameraMatrix, distCoeffs, imageSize, alpha[, newImgSize[, centerPrincipalPoint]]) → retval, validPixROI

C: void **cvGetOptimalNewCameraMatrix**(const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, CvSize **imageSize**, double **alpha**, CvMat* **newCameraMatrix**, CvSize **newImageSize**=cvSize(0, 0), CvRect* **validPixROI**=0, int **centerPrincipalPoint**=0)

Python: cv.**GetOptimalNewCameraMatrix**(cameraMatrix, distCoeffs, imageSize, alpha, newCameraMatrix, newImageSize=(0, 0), validPixROI=0) → None

Parameters

cameraMatrix – Input camera matrix.

distCoeffs – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imageSize – Original image size.

alpha – Free scaling parameter between 0 (when all the pixels in the undistorted image are valid) and 1 (when all the source image pixels are retained in the undistorted image). See `stereoRectify()` for details.

newCameraMatrix – Output new camera matrix.

newImageSize – Image size after rectification. By default, it is set to `imageSize`.

validPixROI – Optional output rectangle that outlines all-good-pixels region in the undistorted image. See `roi1`, `roi2` description in `stereoRectify()`.

centerPrincipalPoint – Optional flag that indicates whether in the new camera matrix the principal point should be at the image center or not. By default, the principal point is chosen to best fit a subset of the source image (determined by `alpha`) to the corrected image.

The function computes and returns the optimal new camera matrix based on the free scaling parameter. By varying this parameter, you 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 is likely to 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 `newImageSize` should be passed to `initUndistortRectifyMap()` to produce the maps for `remap()`.

initCameraMatrix2D

Finds an initial camera matrix from 3D-2D point correspondences.

C++: Mat **initCameraMatrix2D**(InputArrayOfArrays **objectPoints**, InputArrayOfArrays **imagePoints**, Size **imageSize**, double **aspectRatio**=1.)

Python: cv2.**initCameraMatrix2D**(objectPoints, imagePoints, imageSize[, aspectRatio]) → retval

C: void **cvInitIntrinsicParams2D**(const CvMat* **objectPoints**, const CvMat* **imagePoints**, const CvMat* **pointCounts**, CvSize **imageSize**, CvMat* **cameraMatrix**, double **aspectRatio**=1.)

Python: cv.**InitIntrinsicParams2D**(objectPoints, imagePoints, pointCounts, imageSize, cameraMatrix, aspectRatio=1.) → None

> Parameters

> > **objectPoints** – Vector of vectors of the calibration pattern points in the calibration pattern coordinate space. In the old interface all the per-view vectors are concatenated. See calibrateCamera() for details.

> > **imagePoints** – Vector of vectors of the projections of the calibration pattern points. In the old interface all the per-view vectors are concatenated.

> > **npoints** – The integer vector of point counters for each view.

> > **imageSize** – 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 * $ aspectRatio.

The function estimates and returns an initial camera matrix for the camera calibration process. Currently, the function only supports planar calibration patterns, which are patterns where each object point has z-coordinate =0.

matMulDeriv

Computes partial derivatives of the matrix product for each multiplied matrix.

C++: void **matMulDeriv**(InputArray **A**, InputArray **B**, OutputArray **dABdA**, OutputArray **dABdB**)

Python: cv2.**matMulDeriv**(A, B[, dABdA[, dABdB]]) → dABdA, dABdB

> Parameters

> > **A** – First multiplied matrix.

> > **B** – Second multiplied matrix.

> > **dABdA** – First output derivative matrix d(A*B)/dA of size A.rows*B.cols × $A.rows * A.cols$.

> > **dABdB** – Second output derivative matrix d(A*B)/dB of size A.rows*B.cols × $B.rows * B.cols$.

The function computes partial derivatives of the elements of the matrix product $A * B$ with regard to the elements of each of the two input matrices. The function is used to compute the Jacobian matrices in stereoCalibrate() but can also be used in any other similar optimization function.

projectPoints

Projects 3D points to an image plane.

C++: void **projectPoints**(InputArray **objectPoints**, InputArray **rvec**, InputArray **tvec**, InputArray **cameraMatrix**, InputArray **distCoeffs**, OutputArray **imagePoints**, OutputArray **jacobian**=noArray(), double **aspectRatio**=0)

Python: cv2.**projectPoints**(objectPoints, rvec, tvec, cameraMatrix, distCoeffs[, imagePoints[, jacobian[, aspectRatio]]]) → imagePoints, jacobian

C: void **cvProjectPoints2**(const CvMat* **objectPoints**, const CvMat* **rvec**, const CvMat* **tvec**, const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, CvMat* **imagePoints**, CvMat* **dpdrot**=NULL, CvMat* **dpdt**=NULL, CvMat* **dpdf**=NULL, CvMat* **dpdc**=NULL, CvMat* **dpddist**=NULL)

Python: cv.**ProjectPoints2**(objectPoints, rvec, tvec, cameraMatrix, distCoeffs, imagePoints, dpdrot=None, dpdt=None, dpdf=None, dpdc=None, dpddist=None) → None

> **Parameters**
>
> > **objectPoints** – Array of object points, 3xN/Nx3 1-channel or 1xN/Nx1 3-channel (or vector<Point3f>), where N is the number of points in the view.
> >
> > **rvec** – Rotation vector. See Rodrigues() for details.
> >
> > **tvec** – Translation vector.
> >
> > **cameraMatrix** – Camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
> >
> > **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
> >
> > **imagePoints** – Output array of image points, 2xN/Nx2 1-channel or 1xN/Nx1 2-channel, or vector<Point2f> .
> >
> > **jacobian** – Optional output 2Nx(10+<numDistCoeffs>) jacobian matrix of derivatives of image points with respect to components of the rotation vector, translation vector, focal lengths, coordinates of the principal point and the distortion coefficients. In the old interface different components of the jacobian are returned via different output parameters.
> >
> > **aspectRatio** – Optional "fixed aspect ratio" parameter. If the parameter is not 0, the function assumes that the aspect ratio (*fx/fy*) is fixed and correspondingly adjusts the jacobian matrix.

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 be used to compute a re-projection error given the current intrinsic and extrinsic parameters.

Note: By setting rvec=tvec=(0,0,0) or by setting cameraMatrix to a 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function. This means that 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.

---

reprojectImageTo3D

Reprojects a disparity image to 3D space.

C++: void **reprojectImageTo3D**(InputArray **disparity**, OutputArray **_3dImage**, InputArray **Q**, bool **handle-MissingValues**=false, int **depth**=-1)

Python: cv2.**reprojectImageTo3D**(disparity, Q[, _3dImage[, handleMissingValues[, ddepth]]]) → _3dImage

C: void **cvReprojectImageTo3D**(const CvArr* **disparity**, CvArr* **_3dImage**, const CvMat* **Q**, int **handle-MissingValues**=0)

Python: cv.**ReprojectImageTo3D**(disparity, _3dImage, Q, handleMissingValues=0) → None

> **Parameters**
>
>> **disparity** – Input single-channel 8-bit unsigned, 16-bit signed, 32-bit signed or 32-bit floating-point disparity image.
>>
>> **_3dImage** – Output 3-channel floating-point image of the same size as `disparity` . Each element of `_3dImage(x,y)` contains 3D coordinates of the point `(x,y)` computed from the disparity map.
>>
>> **Q** – 4×4 perspective transformation matrix that can be obtained with `stereoRectify()`.
>>
>> **handleMissingValues** – Indicates, whether the function should handle missing values (i.e. points where the disparity was not computed). If `handleMissingValues=true`, then pixels with the minimal disparity that corresponds to the outliers (see `StereoBM::operator()`) are transformed to 3D points with a very large Z value (currently set to 10000).
>>
>> **ddepth** – The optional output array depth. If it is `-1`, the output image will have `CV_32F` depth. `ddepth` can also be set to `CV_16S`, `CV_32S` or `CV_32F`.

The function transforms a single-channel disparity map to a 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` andthe corresponding disparity `d=disparity(x,y)` , it computes:

$$[X\ Y\ Z\ W]^\mathsf{T} = \mathtt{Q} * [\mathtt{x}\ \mathtt{y}\ \mathtt{disparity(x,y)}\ 1]^\mathsf{T}$$
$$\mathtt{\_3dImage}(\mathtt{x},\mathtt{y}) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be an arbitrary 4×4 matrix (for example, the one computed by `stereoRectify()`). To reproject a sparse set of points {(x,y,d),...} to 3D space, use `perspectiveTransform()` .

RQDecomp3x3

Computes an RQ decomposition of 3x3 matrices.

C++: Vec3d **RQDecomp3x3**(InputArray **M**, OutputArray **R**, OutputArray **Q**, OutputArray **Qx**=noArray(), OutputArray **Qy**=noArray(), OutputArray **Qz**=noArray())

Python: cv2.**RQDecomp3x3**(src[, mtxR[, mtxQ[, Qx[, Qy[, Qz]]]]]) → retval, mtxR, mtxQ, Qx, Qy, Qz

C: void **cvRQDecomp3x3**(const CvMat* **M**, CvMat* **R**, CvMat* **Q**, CvMat* **Qx**=NULL, CvMat* **Qy**=NULL, CvMat* **Qz**=NULL, CvPoint3D64f* **eulerAngles**=NULL)

Python: cv.**RQDecomp3x3**(M, R, Q, Qx=None, Qy=None, Qz=None) → eulerAngles

> **Parameters**
>
>> **M** – 3x3 input matrix.
>>
>> **R** – Output 3x3 upper-triangular matrix.
>>
>> **Q** – Output 3x3 orthogonal matrix.
>>
>> **Qx** – Optional output 3x3 rotation matrix around x-axis.
>>
>> **Qy** – Optional output 3x3 rotation matrix around y-axis.

Qz – Optional output 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

Converts a rotation matrix to a rotation vector or vice versa.

C++: void **Rodrigues**(InputArray **src**, OutputArray **dst**, OutputArray **jacobian**=noArray())

Python: cv2.**Rodrigues**(src[, dst[, jacobian]]) → dst, jacobian

C: int **cvRodrigues2**(const CvMat* **src**, CvMat* **dst**, CvMat* **jacobian**=0)

Python: cv.**Rodrigues2**(src, dst, jacobian=0) → None

> **Parameters**

>> **src** – Input rotation vector (3x1 or 1x3) or rotation matrix (3x3).

>> **dst** – Output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively.

>> **jacobian** – Optional output Jacobian matrix, 3x9 or 9x3, which is a matrix of partial derivatives of the output array components with respect to the input array components.

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta)rr^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can be also 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

class StereoBM

Class for computing stereo correspondence using the block matching algorithm.

```
// Block matching stereo correspondence algorithm class StereoBM
{
    enum { NORMALIZED_RESPONSE = CV_STEREO_BM_NORMALIZED_RESPONSE,
        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 searched for.
    // SADWindowSize is the size of averaging window used to match pixel blocks
    //    (larger values mean better robustness to noise, but yield blurry disparity maps)
    StereoBM(int preset, int ndisparities=0, int SADWindowSize=21);
    // separate initialization function
    void init(int preset, int ndisparities=0, int SADWindowSize=21);
    // computes the disparity for the two rectified 8-bit single-channel images.
    // the disparity will be 16-bit signed (fixed-point) or 32-bit floating-point image of the same size as left.
    void operator()( InputArray left, InputArray right, OutputArray disparity, int disptype=CV_16S );

    Ptr<CvStereoBMState> state;
};
```

The class is a C++ wrapper for the associated functions. In particular, StereoBM::operator() is the wrapper for cvFindStereoCorrespondenceBM().

StereoBM::StereoBM

The constructors.

C++: StereoBM::**StereoBM**()

C++: StereoBM::**StereoBM**(int **preset**, int **ndisparities**=0, int **SADWindowSize**=21)

Python: cv2.StereoBM.**StereoBM**(preset[, ndisparities[, SADWindowSize]]) → <StereoBM object>

C: CvStereoBMState* **cvCreateStereoBMState**(int **preset**=CV_STEREO_BM_BASIC, int **ndisparities**=0)

Python: cv.**CreateStereoBMState**(preset=CV_STEREO_BM_BASIC, ndisparities=0) → StereoBMState

> **Parameters**
>
> > **preset** – specifies the whole set of algorithm parameters, one of:
> >
> > – BASIC_PRESET - parameters suitable for general cameras
> >
> > – FISH_EYE_PRESET - parameters suitable for wide-angle cameras
> >
> > – NARROW_PRESET - parameters suitable for narrow-angle cameras
> >
> > After constructing the class, you can override any parameters set by the preset.
> >
> > **ndisparities** – the disparity search range. For each pixel algorithm will find the best disparity from 0 (default minimum disparity) to ndisparities. The search range can then be shifted by changing the minimum disparity.
> >
> > **SADWindowSize** – the linear size of the blocks compared by the algorithm. The size should be odd (as the block is centered at the current pixel). Larger block size implies smoother, though less accurate disparity map. Smaller block size gives more detailed disparity map, but there is higher chance for algorithm to find a wrong correspondence.

The constructors initialize StereoBM state. You can then call StereoBM::operator() to compute disparity for a specific stereo pair.

Note: In the C API you need to deallocate CvStereoBM state when it is not needed anymore using cvReleaseStereoBMState(&stereobm).

StereoBM::operator()

Computes disparity using the BM algorithm for a rectified stereo pair.

C++: void StereoBM::**operator()**(InputArray **left**, InputArray **right**, OutputArray **disp**, int **disptype**=CV_16S)

Python: cv2.StereoBM.**compute**(left, right[, disparity[, disptype]]) → disparity

C: void **cvFindStereoCorrespondenceBM**(const CvArr* **left**, const CvArr* **right**, CvArr* **disparity**, CvStereoBMState* **state**)

Python: cv.**FindStereoCorrespondenceBM**(left, right, disparity, state) → None

> **Parameters**
>
>> **left** – Left 8-bit single-channel or 3-channel image.
>>
>> **right** – Right image of the same size and the same type as the left one.
>>
>> **disp** – Output disparity map. It has the same size as the input images. When disptype==CV_16S, the map is a 16-bit signed single-channel image, containing disparity values scaled by 16. To get the true disparity values from such fixed-point representation, you will need to divide each disp element by 16. If disptype==CV_32F, the disparity map will already contain the real disparity values on output.
>>
>> **disptype** – Type of the output disparity map, CV_16S (default) or CV_32F.
>>
>> **state** – The pre-initialized CvStereoBMState structure in the case of the old API.

The method executes the BM algorithm on a rectified stereo pair. See the stereo_match.cpp OpenCV sample on how to prepare images and call the method. Note that the method is not constant, thus you should not use the same StereoBM instance from within different threads simultaneously.

StereoSGBM

class **StereoSGBM**

Class for computing stereo correspondence using the semi-global block matching algorithm.

```
class StereoSGBM
{
    StereoSGBM();
    StereoSGBM(int minDisparity, int numDisparities, int SADWindowSize,
               int P1=0, int P2=0, int disp12MaxDiff=0,
               int preFilterCap=0, int uniquenessRatio=0,
               int speckleWindowSize=0, int speckleRange=0,
               bool fullDP=false);
    virtual ~StereoSGBM();

    virtual void operator()(InputArray left, InputArray right, OutputArray disp);

    int minDisparity;
    int numberOfDisparities;
    int SADWindowSize;
    int preFilterCap;
    int uniquenessRatio;
    int P1, P2;
    int speckleWindowSize;
    int speckleRange;
    int disp12MaxDiff;
```

```
    bool fullDP;

    ...
};
```

The class implements the modified H. Hirschmuller algorithm HH08 that differs from the original one as follows:

- By default, the algorithm is single-pass, which means that you consider only 5 directions instead of 8. Set `fullDP=true` to run the full variant of the algorithm but beware that it may consume a lot of memory.

- The algorithm matches blocks, not individual pixels. Though, setting `SADWindowSize=1` reduces the blocks to single pixels.

- Mutual information cost function is not implemented. Instead, a simpler Birchfield-Tomasi sub-pixel metric from BT96 is used. Though, the color images are supported as well.

- Some pre- and post- processing steps from K. Konolige algorithm `StereoBM::operator()` are included, for example: pre-filtering (`CV_STEREO_BM_XSOBEL` type) and post-filtering (uniqueness check, quadratic interpolation and speckle filtering).

StereoSGBM::StereoSGBM

C++: `StereoSGBM::`**`StereoSGBM`**`()`

C++: `StereoSGBM::`**`StereoSGBM`**`(int **minDisparity**, int **numDisparities**, int **SADWindowSize**, int **P1**=0, int **P2**=0, int **disp12MaxDiff**=0, int **preFilterCap**=0, int **uniquenessRatio**=0, int **speckleWindowSize**=0, int **speckleRange**=0, bool **fullDP**=false)`

Python: `cv2.StereoSGBM.`**`StereoSGBM`**`(minDisparity, numDisparities, SADWindowSize[, P1[, P2[, disp12MaxDiff[, preFilterCap[, uniquenessRatio[, speckleWindowSize[, speckleRange[, fullDP ]]]]]]]])` → `<StereoSGBM object>`

Initializes `StereoSGBM` and sets parameters to custom values.??

> **Parameters**
>
>> **minDisparity** – Minimum possible disparity value. Normally, it is zero but sometimes rectification algorithms can shift images, so this parameter needs to be adjusted accordingly.
>>
>> **numDisparities** – Maximum disparity minus minimum disparity. The value is always greater than zero. In the current implementation, this parameter must be divisible by 16.
>>
>> **SADWindowSize** – Matched block size. It must be an odd number `>=1` . Normally, it should be somewhere in the `3..11` range.
>>
>> **P1** – The first parameter controlling the disparity smoothness. See below.
>>
>> **P2** – The second parameter controlling the disparity smoothness. The larger the values are, the smoother the disparity is. `P1` is the penalty on the disparity change by plus or minus 1 between neighbor pixels. `P2` is the penalty on the disparity change by more than 1 between neighbor pixels. The algorithm requires `P2 > P1` . See `stereo_match.cpp` sample where some reasonably good `P1` and `P2` values are shown (like `8*number_of_image_channels*SADWindowSize*SADWindowSize` and `32*number_of_image_channels*SADWindowSize*SADWindowSize` , respectively).
>>
>> **disp12MaxDiff** – Maximum allowed difference (in integer pixel units) in the left-right disparity check. Set it to a non-positive value to disable the check.

> **preFilterCap** – Truncation value for the prefiltered image pixels. The algorithm first computes x-derivative at each pixel and clips its value by [`-preFilterCap, preFilterCap`] interval. The result values are passed to the Birchfield-Tomasi pixel cost function.
>
> **uniquenessRatio** – Margin in percentage by which the best (minimum) computed cost function value should "win" the second best value to consider the found match correct. Normally, a value within the 5-15 range is good enough.
>
> **speckleWindowSize** – Maximum size of smooth disparity regions to consider their noise speckles and invalidate. Set it to 0 to disable speckle filtering. Otherwise, set it somewhere in the 50-200 range.
>
> **speckleRange** – Maximum disparity variation within each connected component. If you do speckle filtering, set the parameter to a positive value, multiple of 16. Normally, 16 or 32 is good enough.
>
> **fullDP** – Set it to `true` to run the full-scale two-pass dynamic programming algorithm. It will consume O(W*H*numDisparities) bytes, which is large for 640x480 stereo and huge for HD-size pictures. By default, it is set to `false` .

The first constructor initializes `StereoSGBM` with all the default parameters. So, you only have to set `StereoSGBM::numberOfDisparities` at minimum. The second constructor enables you to set each parameter to a custom value.

StereoSGBM::operator ()

C++: void StereoSGBM::**operator()**(InputArray **left**, InputArray **right**, OutputArray **disp**)

Python: cv2.StereoSGBM.**compute**(left, right[, disp]) → disp

> Computes disparity using the SGBM algorithm for a rectified stereo pair.

> **Parameters**

> > **left** – Left 8-bit single-channel or 3-channel image.

> > **right** – Right image of the same size and the same type as the left one.

> > **disp** – Output disparity map. It is a 16-bit signed single-channel image of the same size as the input image. It contains disparity values scaled by 16. So, to get the floating-point disparity map, you need to divide each `disp` element by 16.

The method executes the SGBM algorithm on a rectified stereo pair. See `stereo_match.cpp` OpenCV sample on how to prepare images and call the method.

Note: The method is not constant, so you should not use the same `StereoSGBM` instance from different threads simultaneously.

---

stereoCalibrate

Calibrates the stereo camera.

C++: double **stereoCalibrate**(InputArrayOfArrays **objectPoints**, InputArrayOfArrays **imagePoints1**, InputArrayOfArrays **imagePoints2**, InputOutputArray **cameraMatrix1**, InputOutputArray **distCoeffs1**, InputOutputArray **cameraMatrix2**, InputOutputArray **distCoeffs2**, Size **imageSize**, OutputArray **R**, OutputArray **T**, OutputArray **E**, OutputArray **F**, TermCriteria **term_crit**=TermCriteria(TermCriteria::COUNT+ TermCriteria::EPS, 30, 1e-6), int **flags**=CALIB_FIX_INTRINSIC)

Python: cv2.**stereoCalibrate**(*objectPoints, imagePoints1, imagePoints2, imageSize[, cameraMatrix1[, distCoeffs1[, cameraMatrix2[, distCoeffs2[, R[, T[, E[, F[, criteria[, flags]]]]]]]]]*) → retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F

C: double **cvStereoCalibrate**(const CvMat* **objectPoints**, const CvMat* **imagePoints1**, const CvMat* **imagePoints2**, const CvMat* **pointCounts**, CvMat* **cameraMatrix1**, CvMat* **distCoeffs1**, CvMat* **cameraMatrix2**, CvMat* **distCoeffs2**, CvSize **imageSize**, CvMat* **R**, CvMat* **T**, CvMat* **E**=0, CvMat* **F**=0, CvTermCriteria **termCrit**=cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 30, 1e-6), int **flags**=CV_CALIB_FIX_INTRINSIC)

Python: cv.**StereoCalibrate**(objectPoints, imagePoints1, imagePoints2, pointCounts, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T, E=None, F=None, termCrit=(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 30, 1e-6), flags=CV_CALIB_FIX_INTRINSIC) → None

> **Parameters**

>> **objectPoints** – Vector of vectors of the calibration pattern points.

>> **imagePoints1** – Vector of vectors of the projections of the calibration pattern points, observed by the first camera.

>> **imagePoints2** – Vector of vectors of the projections of the calibration pattern points, observed by the second camera.

>> **cameraMatrix1** – Input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of `CV_CALIB_USE_INTRINSIC_GUESS` , `CV_CALIB_FIX_ASPECT_RATIO` , `CV_CALIB_FIX_INTRINSIC` , or `CV_CALIB_FIX_FOCAL_LENGTH` are specified, some or all of the matrix components must be initialized. See the flags description for details.

>> **distCoeffs1** – Input/output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. The output vector length depends on the flags.

>> **cameraMatrix2** – Input/output second camera matrix. The parameter is similar to `cameraMatrix1` .

>> **distCoeffs2** – Input/output lens distortion coefficients for the second camera. The parameter is similar to `distCoeffs1` .

>> **imageSize** – Size of the image used only to initialize intrinsic camera matrix.

>> **R** – Output rotation matrix between the 1st and the 2nd camera coordinate systems.

>> **T** – Output translation vector between the coordinate systems of the cameras.

>> **E** – Output essential matrix.

>> **F** – Output fundamental matrix.

>> **term_crit** – Termination criteria for the iterative optimization algorithm.

6.1. Camera Calibration and 3D Reconstruction 363

flags – Different flags that may be zero or a combination of the following values:

- **CV_CALIB_FIX_INTRINSIC** Fix `cameraMatrix?` and `distCoeffs?` so that only `R`, `T`, `E`, and `F` matrices are estimated.

- **CV_CALIB_USE_INTRINSIC_GUESS** Optimize some or all of the intrinsic parameters according to the specified flags. Initial values are provided by the user.

- **CV_CALIB_FIX_PRINCIPAL_POINT** Fix the principal points during the optimization.

- **CV_CALIB_FIX_FOCAL_LENGTH** Fix $f_x^{(j)}$ and $f_y^{(j)}$.

- **CV_CALIB_FIX_ASPECT_RATIO** Optimize $f_y^{(j)}$. Fix the ratio $f_x^{(j)}/f_y^{(j)}$.

- **CV_CALIB_SAME_FOCAL_LENGTH** Enforce $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$.

- **CV_CALIB_ZERO_TANGENT_DIST** Set tangential distortion coefficients for each camera to zeros and fix there.

- **CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6** Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used. Otherwise, it is set to 0.

- **CV_CALIB_RATIONAL_MODEL** Enable coefficients k4, k5, and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function computes and returns only 5 distortion coefficients.

The function estimates transformation between two cameras making a stereo pair. If you have a stereo camera where the relative position and orientation of two cameras is fixed, and if you computed poses of an object relative to the first camera and to the second camera, (R1, T1) and (R2, T2), respectively (this can be done with [solvePnP()](#)), then those poses definitely relate to each other. This means that, given (R_1,:math:`T_1`), it should be possible to compute (R_2,:math:`T_2`). You only need to know the position and orientation of the second camera relative to the first camera. This is what the described function does. It computes (R,:math:`T`) so that:

$$R_2 = R * R_1 \quad T_2 = R * T_1 + T,$$

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 the function can also compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E\, cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform a full calibration of each of two cameras. However, due to the high dimensionality of the parameter space and noise in the input data, the function can diverge from the correct solution. If the intrinsic parameters can be estimated with high accuracy for each of the cameras individually (for example, using [calibrateCamera()](#)), you are 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, for example, pass `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which is usually a reasonable assumption.

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

Computes rectification transforms for each head of a calibrated stereo camera.

C++: void **stereoRectify**(InputArray **cameraMatrix1**, InputArray **distCoeffs1**, InputArray **cameraMatrix2**, InputArray **distCoeffs2**, Size **imageSize**, InputArray **R**, InputArray **T**, OutputArray **R1**, OutputArray **R2**, OutputArray **P1**, OutputArray **P2**, OutputArray **Q**, int **flags**=CALIB_ZERO_DISPARITY, double **alpha**, Size **newImageSize**=Size(), Rect* **roi1**=0, Rect* **roi2**=0)

C: void **cvStereoRectify**(const CvMat* **cameraMatrix1**, const CvMat* **cameraMatrix2**, const CvMat* **distCoeffs1**, const CvMat* **distCoeffs2**, CvSize **imageSize**, const CvMat* **R**, const CvMat* **T**, CvMat* **R1**, CvMat* **R2**, CvMat* **P1**, CvMat* **P2**, CvMat* **Q**=0, int **flags**=CV_CALIB_ZERO_DISPARITY, double **alpha**=-1, CvSize **newImageSize**=cvSize(0, 0), CvRect* **roi1**=0, CvRect* **roi2**=0)

Python: cv.**StereoRectify**(cameraMatrix1, cameraMatrix2, distCoeffs1, distCoeffs2, imageSize, R, T, R1, R2, P1, P2, Q=None, flags=CV_CALIB_ZERO_DISPARITY, alpha=-1, newImageSize=(0, 0))-> (roi1, roi2)

> Parameters

> > **cameraMatrix1** – First camera matrix.

> > **cameraMatrix2** – Second camera matrix.

> > **distCoeffs1** – First camera distortion parameters.

> > **distCoeffs2** – Second camera distortion parameters.

> > **imageSize** – Size of the image used for stereo calibration.

> > **R** – Rotation matrix between the coordinate systems of the first and the second cameras.

> > **T** – Translation vector between coordinate systems of the cameras.

> > **R1** – Output 3x3 rectification transform (rotation matrix) for the first camera.

> > **R2** – Output 3x3 rectification transform (rotation matrix) for the second camera.

> > **P1** – Output 3x4 projection matrix in the new (rectified) coordinate systems for the first camera.

> > **P2** – Output 3x4 projection matrix in the new (rectified) coordinate systems for the second camera.

> > **Q** – Output 4×4 disparity-to-depth mapping matrix (see `reprojectImageTo3D()`).

> > **flags** – Operation flags that may be zero 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 the horizontal or vertical direction (depending on the orientation of epipolar lines) to maximize the useful image area.

> > **alpha** – Free scaling parameter. If it is -1 or absent, the function performs the default scaling. Otherwise, the parameter should be between 0 and 1. `alpha=0` means that the rectified images are zoomed and shifted so that only valid pixels are visible (no black areas after rectification). `alpha=1` means that the rectified image is decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images (no source image pixels are lost). Obviously, any intermediate value yields an intermediate result between those two extreme cases.

> > **newImageSize** – New image resolution after rectification. The same size should be passed to `initUndistortRectifyMap()` (see the `stereo_calib.cpp` sample in OpenCV samples

directory). When (0,0) is passed (default), it is set to the original `imageSize` . Setting it to larger value can help you preserve details in the original image, especially when there is a big radial distortion.

roi1 –

roi2 – Optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0` , the ROIs cover the whole images. Otherwise, they are likely to 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, this makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. The function takes the matrices computed by `stereoCalibrate()` as input. As output, it provides two rotation matrices and also two projection matrices in the new coordinates. The function distinguishes the following two cases:

1. **Horizontal stereo**: the first and the second camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). In the rectified images, the corresponding epipolar lines in the left and right cameras are horizontal and have the same y-coordinate. P1 and P2 look like:

$$
\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}
$$

$$
\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} ,
$$

where T_x is a horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. **Vertical stereo**: the first and the second camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). The epipolar lines in the rectified images are vertical and have the same x-coordinate. P1 and P2 look like:

$$
\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}
$$

$$
\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix} ,
$$

where T_y is a vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first three 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.

See below the screenshot from the `stereo_calib.cpp` sample. Some red horizontal lines pass through the corresponding image regions. This means that the images are well rectified, which is what most stereo correspondence algorithms rely on. The green rectangles are `roi1` and `roi2` . You see that their interiors are all valid pixels.
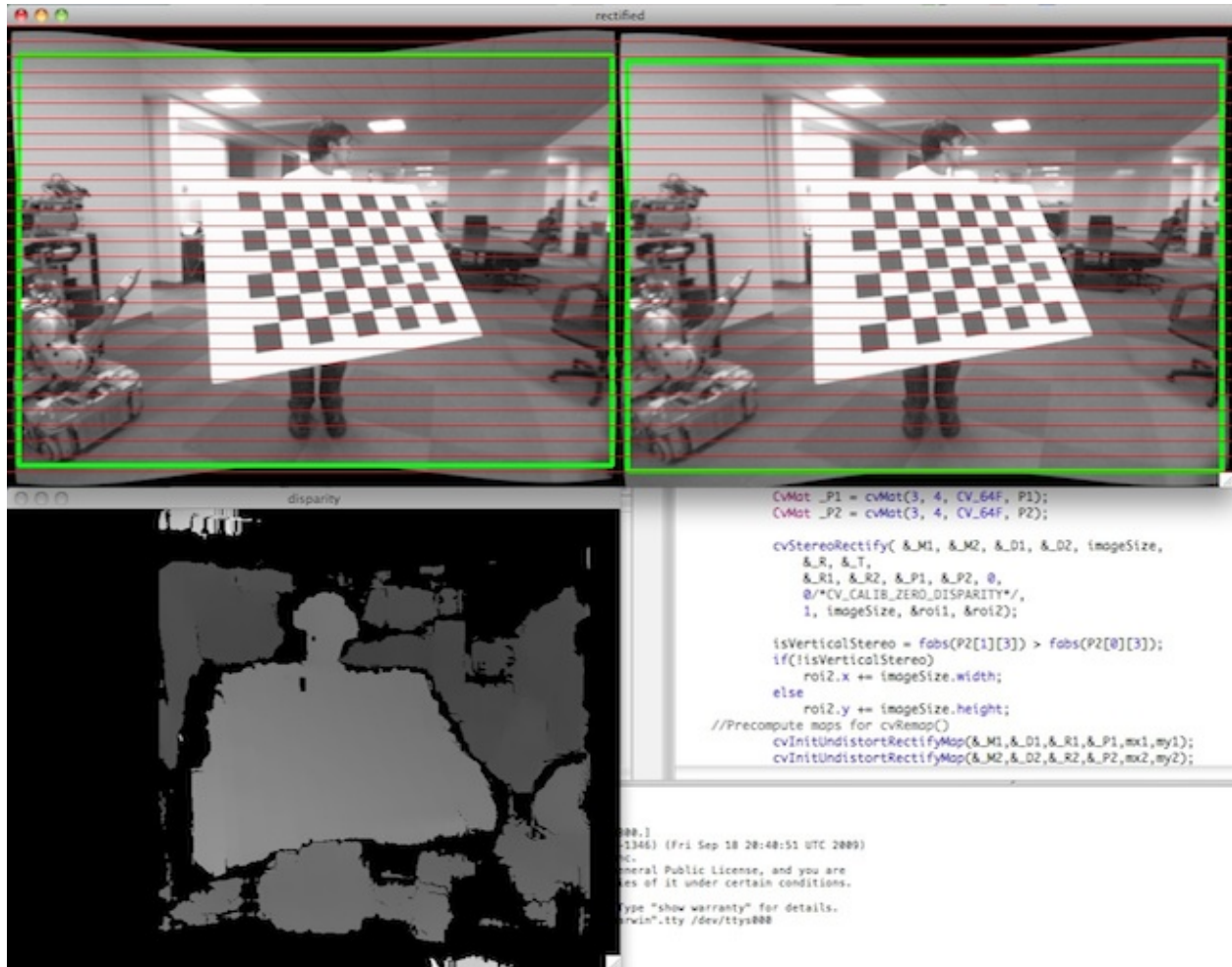
stereoRectifyUncalibrated

Computes a rectification transform for an uncalibrated stereo camera.

C++: bool **stereoRectifyUncalibrated**(InputArray **points1**, InputArray **points2**, InputArray **F**, Size **img-Size**, OutputArray **H1**, OutputArray **H2**, double **threshold**=5)

Python: cv2.**stereoRectifyUncalibrated**(points1, points2, F, imgSize[, H1[, H2[, threshold]]]) → retval, H1, H2

C: void **cvStereoRectifyUncalibrated**(const CvMat* **points1**, const CvMat* **points2**, const CvMat* **F**, CvSize **imageSize**, CvMat* **H1**, CvMat* **H2**, double **threshold**=5)

Python: cv.**StereoRectifyUncalibrated**(points1, points2, F, imageSize, H1, H2, threshold=5) → None

> **Parameters**
>
> > **points1** – Array of feature points in the first image.
> >
> > **points2** – The corresponding points in the second image. The same formats as in findFundamentalMat() are supported.
> >
> > **F** – Input fundamental matrix. It can be computed from the same set of point pairs using findFundamentalMat() .

imageSize – Size of the image.

H1 – Output rectification homography matrix for the first image.

H2 – Output rectification homography matrix for the second image.

threshold – Optional threshold used to filter out the outliers. If the parameter is greater than zero, all the point pairs that do not comply with the epipolar geometry (that is, the points for which $|\texttt{points2[i]}^{\top} * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise,all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in the space, which explains the suffix "uncalibrated". Another related difference from `stereoRectify()` is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations encoded by the homography matrices `H1` and `H2` . The function implements the algorithm [Hartley99].

Note: 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 a significant distortion, it would be better to correct it 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()` . Then, the images can be corrected using `undistort()` , or just the point coordinates can be corrected with `undistortPoints()` .

FEATURES2D. 2D FEATURES FRAMEWORK

7.1 Feature Detection and Description

FAST

Detects corners using the FAST algorithm

C++: void **FAST**(const Mat& **image**, vector<KeyPoint>& **keypoints**, int **threshold**, bool **nonmaxSupres-sion**=true)

        Parameters

                image – Image where keypoints (corners) are detected.

                keypoints – Keypoints detected on the image.

                threshold – Threshold on difference between intensity of the central pixel and pixels on a circle around this pixel. See the algorithm description below.

                nonmaxSupression – If it is true, non-maximum supression is applied to detected corners (keypoints).

Detects corners using the FAST algorithm by E. Rosten (*Machine Learning for High-speed Corner Detection*, 2006).

MSER

class MSER

Maximally stable extremal region extractor.

```
class MSER : public CvMSERParams
{
public:
    // default constructor
    MSER();
    // constructor that initializes all the algorithm parameters
    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 MSERs,
    // each encoded as a contour (vector<Point>, see findContours)
```

```
    // the optional mask marks the area where MSERs are searched for
    void operator()( const Mat& image, vector<vector<Point> >& msers, const Mat& mask ) const;
};
```

The class encapsulates all the parameters of the MSER extraction algorithm (see http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions). Also see http://opencv.willowgarage.com/wiki/documentation/cpp/features2d/MSER for usefull comments and parameters description.

StarDetector

class StarDetector

Class implementing the `Star` keypoint detector, a modified version of the `CenSurE` keypoint detector described in [Agrawal08].

StarDetector::StarDetector

The Star Detector constructor

C++: StarDetector::**StarDetector**()

C++: StarDetector::**StarDetector**(int **maxSize**, int **responseThreshold**, int **lineThresholdProjected**, int **lineThresholdBinarized**, int **suppressNonmaxSize**)

Python: cv2.**StarDetector**(maxSize, responseThreshold, lineThresholdProjected, lineThresholdBinarized, suppressNonmaxSize) → <StarDetector object>

> **Parameters**

>> **maxSize** – maximum size of the features. The following values are supported: 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128. In the case of a different value the result is undefined.

>> **responseThreshold** – threshold for the approximated laplacian, used to eliminate weak features. The larger it is, the less features will be retrieved

>> **lineThresholdProjected** – another threshold for the laplacian to eliminate edges

>> **lineThresholdBinarized** – yet another threshold for the feature size to eliminate edges. The larger the 2nd threshold, the more points you get.

StarDetector::operator()

Finds keypoints in an image

C++: void StarDetector::**operator()**(const Mat& **image**, vector<KeyPoint>& **keypoints**)

Python: cv2.StarDetector.**detect**(image) → keypoints

C: CvSeq* **cvGetStarKeypoints**(const CvArr* **image**, CvMemStorage* **storage**, CvStarDetectorParams **params**=cvStarDetectorParams())

Python: cv.**GetStarKeypoints**(image, storage, params) → keypoints

> **Parameters**

>> **image** – The input 8-bit grayscale image

>> **keypoints** – The output vector of keypoints

> > **storage** – The memory storage used to store the keypoints (OpenCV 1.x API only)
> >
> > **params** – The algorithm parameters stored in `CvStarDetectorParams` (OpenCV 1.x API only)

SIFT

class SIFT

Class for extracting keypoints and computing descriptors using the Scale Invariant Feature Transform (SIFT) approach.

```cpp
class CV_EXPORTS SIFT
{
public:
    struct CommonParams
    {
        static const int DEFAULT_NOCTAVES = 4;
        static const int DEFAULT_NOCTAVE_LAYERS = 3;
        static const int DEFAULT_FIRST_OCTAVE = -1;
        enum{ FIRST_ANGLE = 0, AVERAGE_ANGLE = 1 };

        CommonParams();
        CommonParams( int _nOctaves, int _nOctaveLayers, int _firstOctave,
                                        int _angleMode );
        int nOctaves, nOctaveLayers, firstOctave;
        int angleMode;
    };

    struct DetectorParams
    {
        static double GET_DEFAULT_THRESHOLD()
          { return 0.04 / SIFT::CommonParams::DEFAULT_NOCTAVE_LAYERS / 2.0; }
        static double GET_DEFAULT_EDGE_THRESHOLD() { return 10.0; }

        DetectorParams();
        DetectorParams( double _threshold, double _edgeThreshold );
        double threshold, edgeThreshold;
    };

    struct DescriptorParams
    {
        static double GET_DEFAULT_MAGNIFICATION() { return 3.0; }
        static const bool DEFAULT_IS_NORMALIZE = true;
        static const int DESCRIPTOR_SIZE = 128;

        DescriptorParams();
        DescriptorParams( double _magnification, bool _isNormalize,
                                        bool _recalculateAngles );
        double magnification;
        bool isNormalize;
        bool recalculateAngles;
    };

    SIFT();
    //! sift-detector constructor
    SIFT( double _threshold, double _edgeThreshold,
        int _nOctaves=CommonParams::DEFAULT_NOCTAVES,
        int _nOctaveLayers=CommonParams::DEFAULT_NOCTAVE_LAYERS,
```

```
            int _firstOctave=CommonParams::DEFAULT_FIRST_OCTAVE,
            int _angleMode=CommonParams::FIRST_ANGLE );
    //! sift-descriptor constructor
    SIFT( double _magnification, bool _isNormalize=true,
            bool _recalculateAngles = true,
            int _nOctaves=CommonParams::DEFAULT_NOCTAVES,
            int _nOctaveLayers=CommonParams::DEFAULT_NOCTAVE_LAYERS,
            int _firstOctave=CommonParams::DEFAULT_FIRST_OCTAVE,
            int _angleMode=CommonParams::FIRST_ANGLE );
    SIFT( const CommonParams& _commParams,
            const DetectorParams& _detectorParams = DetectorParams(),
            const DescriptorParams& _descriptorParams = DescriptorParams() );


    //! returns the descriptor size in floats (128)
    int descriptorSize() const { return DescriptorParams::DESCRIPTOR_SIZE; }
    //! finds the keypoints using the SIFT algorithm
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints) const;
    //! finds the keypoints and computes descriptors for them using SIFT algorithm.
    //! Optionally it can compute descriptors for the user-provided keypoints
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints,
                    Mat& descriptors,
                    bool useProvidedKeypoints=false) const;


    CommonParams getCommonParams () const { return commParams; }
    DetectorParams getDetectorParams () const { return detectorParams; }
    DescriptorParams getDescriptorParams () const { return descriptorParams; }
protected:
    ...
};
```

SURF

class SURF

Class for extracting Speeded Up Robust Features from an image [Bay06]. The class is derived from `CvSURFParams` structure, which specifies the algorithm parameters:

   int **extended**

   - 0 means that the basic descriptors (64 elements each) shall be computed

   - 1 means that the extended descriptors (128 elements each) shall be computed

   int **upright**

   - 0 means that detector computes orientation of each feature.

   - 1 means that the orientation is not computed (which is much, much faster). For example, if you match images from a stereo pair, or do image stitching, the matched features likely have very similar angles, and you can speed up feature extraction by setting `upright=1`.

   double **hessianThreshold**
   Threshold for the keypoint detector. Only features, whose hessian is larger than `hessianThreshold` are retained by the detector. Therefore, the larger the value, the less keypoints you will get. A good default value could be from 300 to 500, depending from the image contrast.

int **nOctaves**

> The number of a gaussian pyramid octaves that the detector uses. It is set to 4 by default. If you want to get very large features, use the larger value. If you want just small features, decrease it.

int **nOctaveLayers**

> The number of images within each octave of a gaussian pyramid. It is set to 2 by default.

SURF::SURF

The SURF extractor constructors.

C++: SURF::**SURF**()

C++: SURF::**SURF**(double **hessianThreshold**, int **nOctaves**=4, int **nOctaveLayers**=2, bool **extended**=false, bool **upright**=false)

Python: cv2.**SURF**(_hessianThreshold[, _nOctaves[, _nOctaveLayers[, _extended[, _upright]]]]) → <SURF object>

> **Parameters**
>
> > **hessianThreshold** – Threshold for hessian keypoint detector used in SURF.
> >
> > **nOctaves** – Number of pyramid octaves the keypoint detector will use.
> >
> > **nOctaveLayers** – Number of octave layers within each octave.
> >
> > **extended** – Extended descriptor flag (true - use extended 128-element descriptors; false - use 64-element descriptors).
> >
> > **upright** – Up-right or rotated features flag (true - do not compute orientation of features; false - compute orientation).

SURF::operator()

Detects keypoints and computes SURF descriptors for them.

C++: void SURF::**operator()**(const Mat& **image**, const Mat& **mask**, vector<KeyPoint>& **keypoints**)

C++: void SURF::**operator()**(const Mat& **image**, const Mat& **mask**, vector<KeyPoint>& **keypoints**, vector<float>& **descriptors**, bool **useProvidedKeypoints**=false)

Python: cv2.SURF.**detect**(img, mask) → keypoints

Python: cv2.SURF.**detect**(img, mask[, useProvidedKeypoints]) → keypoints, descriptors

C: void **cvExtractSURF**(const CvArr* **image**, const CvArr* **mask**, CvSeq** **keypoints**, CvSeq** **descriptors**, CvMemStorage* **storage**, CvSURFParams **params**)

Python: cv.**ExtractSURF**(image, mask, storage, params)-> (keypoints, descriptors)

> **Parameters**
>
> > **image** – Input 8-bit grayscale image
> >
> > **mask** – Optional input mask that marks the regions where we should detect features.
> >
> > **keypoints** – The input/output vector of keypoints
> >
> > **descriptors** – The output concatenated vectors of descriptors. Each descriptor is 64- or 128-element vector, as returned by SURF::descriptorSize(). So the total size of descriptors will be keypoints.size()*descriptorSize().

> > **useProvidedKeypoints** – Boolean flag. If it is true, the keypoint detector is not run. Instead, the provided vector of keypoints is used and the algorithm just computes their descriptors.
> >
> > **storage** – Memory storage for the output keypoints and descriptors in OpenCV 1.x API.
> >
> > **params** – SURF algorithm parameters in OpenCV 1.x API.

ORB

class ORB

Class for extracting ORB features and descriptors from an image.

```
class ORB
{
public:
    /** The patch sizes that can be used (only one right now) */
    struct CommonParams
    {
        enum { DEFAULT_N_LEVELS = 3, DEFAULT_FIRST_LEVEL = 0};

        /** default constructor */
        CommonParams(float scale_factor = 1.2f, unsigned int n_levels = DEFAULT_N_LEVELS,
            int edge_threshold = 31, unsigned int first_level = DEFAULT_FIRST_LEVEL);
        void read(const FileNode& fn);
        void write(FileStorage& fs) const;

        /** Coefficient by which we divide the dimensions from one scale pyramid level to the next */
        float scale_factor_;
        /** The number of levels in the scale pyramid */
        unsigned int n_levels_;
        /** The level at which the image is given
         * if 1, that means we will also look at the image scale_factor_ times bigger
         */
        unsigned int first_level_;
        /** How far from the boundary the points should be */
        int edge_threshold_;
    };

    // c:function::default constructor
    ORB();
    // constructor that initializes all the algorithm parameters
    ORB( const CommonParams detector_params );
    // returns the number of elements in each descriptor (32 bytes)
    int descriptorSize() const;
    // detects keypoints using ORB
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints) const;
    // detects ORB keypoints and computes the ORB descriptors for them;
    // output vector "descriptors" stores elements of descriptors and has size
    // equal descriptorSize()*keypoints.size() as each descriptor is
    // descriptorSize() elements of this vector.
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints,
                    cv::Mat& descriptors,
                    bool useProvidedKeypoints=false) const;
};
```

The class implements ORB.

RandomizedTree

class RandomizedTree

Class containing a base structure for RTreeClassifier.

```cpp
class CV_EXPORTS RandomizedTree
{
public:
        friend class RTreeClassifier;

        RandomizedTree();
        ~RandomizedTree();

        void train(std::vector<BaseKeypoint> const& base_set,
                RNG &rng, int depth, int views,
                size_t reduced_num_dim, int num_quant_bits);
        void train(std::vector<BaseKeypoint> const& base_set,
                RNG &rng, PatchGenerator &make_patch, int depth,
                int views, size_t reduced_num_dim, int num_quant_bits);

        // next two functions are EXPERIMENTAL
        //(do not use unless you know exactly what you do)
        static void quantizeVector(float *vec, int dim, int N, float bnds[2],
                int clamp_mode=0);
        static void quantizeVector(float *src, int dim, int N, float bnds[2],
                uchar *dst);

        // patch_data must be a 32x32 array (no row padding)
        float* getPosterior(uchar* patch_data);
        const float* getPosterior(uchar* patch_data) const;
        uchar* getPosterior2(uchar* patch_data);

        void read(const char* file_name, int num_quant_bits);
        void read(std::istream &is, int num_quant_bits);
        void write(const char* file_name) const;
        void write(std::ostream &os) const;

        int classes() { return classes_; }
        int depth() { return depth_; }

        void discardFloatPosteriors() { freePosteriors(1); }

        inline void applyQuantization(int num_quant_bits)
                { makePosteriors2(num_quant_bits); }

private:
        int classes_;
        int depth_;
        int num_leaves_;
        std::vector<RTreeNode> nodes_;
        float **posteriors_;        // 16-byte aligned posteriors
        uchar **posteriors2_;     // 16-byte aligned posteriors
        std::vector<int> leaf_counts_;

        void createNodes(int num_nodes, RNG &rng);
        void allocPosteriorsAligned(int num_leaves, int num_classes);
        void freePosteriors(int which);
                // which: 1=posteriors_, 2=posteriors2_, 3=both
```

```
        void init(int classes, int depth, RNG &rng);
        void addExample(int class_id, uchar* patch_data);
        void finalize(size_t reduced_num_dim, int num_quant_bits);
        int getIndex(uchar* patch_data) const;
        inline float* getPosteriorByIndex(int index);
        inline uchar* getPosteriorByIndex2(int index);
        inline const float* getPosteriorByIndex(int index) const;
        void convertPosteriorsToChar();
        void makePosteriors2(int num_quant_bits);
        void compressLeaves(size_t reduced_num_dim);
        void estimateQuantPercForPosteriors(float perc[2]);
};
```

RandomizedTree::train

Trains a randomized tree using an input set of keypoints.

C++: void **train**(std::vector<BaseKeypoint> const& **base_set**, RNG& **rng**, PatchGenerator& **make_patch**, int **depth**, int **views**, size_t **reduced_num_dim**, int **num_quant_bits**)

C++: void **train**(std::vector<BaseKeypoint> const& **base_set**, RNG& **rng**, PatchGenerator& **make_patch**, int **depth**, int **views**, size_t **reduced_num_dim**, int **num_quant_bits**)

>   **Parameters**

>   >   **base_set** – Vector of the BaseKeypoint type. It contains image keypoints used for training.

>   >   **rng** – Random-number generator used for training.

>   >   **make_patch** – Patch generator used for training.

>   >   **depth** – Maximum tree depth.

>   >   **views** – Number of random views of each keypoint neighborhood to generate.

>   >   **reduced_num_dim** – Number of dimensions used in the compressed signature.

>   >   **num_quant_bits** – Number of bits used for quantization.

RandomizedTree::read

Reads a pre-saved randomized tree from a file or stream.

C++: **read**(const char* **file_name**, int **num_quant_bits**)

C++: **read**(std::istream& **is**, int **num_quant_bits**)

>   **Parameters**

>   >   **file_name** – Name of the file that contains randomized tree data.

>   >   **is** – Input stream associated with the file that contains randomized tree data.

>   >   **num_quant_bits** – Number of bits used for quantization.

RandomizedTree::write

Writes the current randomized tree to a file or stream.

C++: void **write**(const char* **file_name**) const

C++: void **write**(std::ostream& **os**) const

> **Parameters**

>> **file_name** – Name of the file where randomized tree data is stored.

>> **is** – Output stream associated with the file where randomized tree data is stored.

RandomizedTree::applyQuantization

C++: void **applyQuantization**(int **num_quant_bits**)

> Applies quantization to the current randomized tree.

>> **Parameters**

>>> **num_quant_bits** – Number of bits used for quantization.

RTreeNode

class RTreeNode

Class containing a base structure for `RandomizedTree`.

```cpp
struct RTreeNode
{
        short offset1, offset2;

        RTreeNode() {}

        RTreeNode(uchar x1, uchar y1, uchar x2, uchar y2)
                : offset1(y1*PATCH_SIZE + x1),
                offset2(y2*PATCH_SIZE + x2)
        {}

        //! Left child on 0, right child on 1
        inline bool operator() (uchar* patch_data) const
        {
                return patch_data[offset1] > patch_data[offset2];
        }
};
```

RTreeClassifier

class RTreeClassifier

Class containing `RTreeClassifier`. It represents the Calonder descriptor originally introduced by Michael Calonder.

```cpp
class CV_EXPORTS RTreeClassifier
{
public:
        static const int DEFAULT_TREES = 48;
        static const size_t DEFAULT_NUM_QUANT_BITS = 4;

        RTreeClassifier();

        void train(std::vector<BaseKeypoint> const& base_set,
                RNG &rng,
```

```
                int num_trees = RTreeClassifier::DEFAULT_TREES,
                int depth = DEFAULT_DEPTH,
                int views = DEFAULT_VIEWS,
                size_t reduced_num_dim = DEFAULT_REDUCED_NUM_DIM,
                int num_quant_bits = DEFAULT_NUM_QUANT_BITS,
                        bool print_status = true);
        void train(std::vector<BaseKeypoint> const& base_set,
                RNG &rng,
                PatchGenerator &make_patch,
                int num_trees = RTreeClassifier::DEFAULT_TREES,
                int depth = DEFAULT_DEPTH,
                int views = DEFAULT_VIEWS,
                size_t reduced_num_dim = DEFAULT_REDUCED_NUM_DIM,
                int num_quant_bits = DEFAULT_NUM_QUANT_BITS,
                 bool print_status = true);

        // sig must point to a memory block of at least
        //classes()*sizeof(float|uchar) bytes
        void getSignature(IplImage *patch, uchar *sig);
        void getSignature(IplImage *patch, float *sig);
        void getSparseSignature(IplImage *patch, float *sig,
                float thresh);

        static int countNonZeroElements(float *vec, int n, double tol=1e-10);
        static inline void safeSignatureAlloc(uchar **sig, int num_sig=1,
                        int sig_len=176);
        static inline uchar* safeSignatureAlloc(int num_sig=1,
                        int sig_len=176);

        inline int classes() { return classes_; }
        inline int original_num_classes()
                { return original_num_classes_; }

        void setQuantization(int num_quant_bits);
        void discardFloatPosteriors();

        void read(const char* file_name);
        void read(std::istream &is);
        void write(const char* file_name) const;
        void write(std::ostream &os) const;

        std::vector<RandomizedTree> trees_;

private:
        int classes_;
        int num_quant_bits_;
        uchar **posteriors_;
        ushort *ptemp_;
        int original_num_classes_;
        bool keep_floats_;
};
```

RTreeClassifier::train

Trains a randomized tree classifier using an input set of keypoints.

C++: void **train**(vector<BaseKeypoint> const& **base_set**, RNG& **rng**, int **num_trees**=RTreeClassifier::DEFAULT_TREES, int **depth**=DEFAULT_DEPTH, int **views**=DEFAULT_VIEWS, size_t **reduced_num_dim**=DEFAULT_REDUCED_NUM_DIM, int **num_quant_bits**=DEFAULT_NUM_QUANT_BITS, bool **print_status**=true)

C++: void **train**(vector<BaseKeypoint> const& **base_set**, RNG& **rng**, PatchGenerator& **make_patch**, int **num_trees**=RTreeClassifier::DEFAULT_TREES, int **depth**=DEFAULT_DEPTH, int **views**=DEFAULT_VIEWS, size_t **reduced_num_dim**=DEFAULT_REDUCED_NUM_DIM, int **num_quant_bits**=DEFAULT_NUM_QUANT_BITS, bool **print_status**=true)

> **Parameters**

>> **base_set** – Vector of the `BaseKeypoint` type. It contains image keypoints used for training.

>> **rng** – Random-number generator used for training.

>> **make_patch** – Patch generator used for training.

>> **num_trees** – Number of randomized trees used in `RTreeClassificator`.

>> **depth** – Maximum tree depth.

>> **views** – Number of random views of each keypoint neighborhood to generate.

>> **reduced_num_dim** – Number of dimensions used in the compressed signature.

>> **num_quant_bits** – Number of bits used for quantization.

>> **print_status** – Current status of training printed on the console.

RTreeClassifier::getSignature

Returns a signature for an image patch.

C++: void **getSignature**(IplImage* **patch**, uchar* **sig**)

C++: void **getSignature**(IplImage* **patch**, float* **sig**)

> **Parameters**

>> **patch** – Image patch to calculate the signature for.

>> **sig** – Output signature (array dimension is `reduced_num_dim`) .

RTreeClassifier::getSparseSignature

Returns a sparse signature for an image patch

C++: void **getSparseSignature**(IplImage* **patch**, float* **sig**, float **thresh**)

> **Parameters**

>> **patch** – Image patch to calculate the signature for.

>> **sig** – Output signature (array dimension is `reduced_num_dim`) .

>> **thresh** – Threshold used for compressing the signature.

Returns a signature for an image patch similarly to `getSignature` but uses a threshold for removing all signature elements below the threshold so that the signature is compressed.

RTreeClassifier::countNonZeroElements

Returns the number of non-zero elements in an input array.

C++: **static** int **countNonZeroElements**(float* **vec**, int **n**, double **tol**=1e-10)

> **Parameters**
>
>> **vec** – Input vector containing float elements.
>>
>> **n** – Input vector size.
>>
>> **tol** – Threshold used for counting elements. All elements less than `tol` are considered as zero elements.

RTreeClassifier::read

Reads a pre-saved `RTreeClassifier` from a file or stream.

C++: **read**(const char* **file_name**)

C++: **read**(std::istream& **is**)

> **Parameters**
>
>> **file_name** – Name of the file that contains randomized tree data.
>>
>> **is** – Input stream associated with the file that contains randomized tree data.

RTreeClassifier::write

Writes the current `RTreeClassifier` to a file or stream.

C++: void **write**(const char* **file_name**) const

C++: void **write**(std::ostream& **os**) const

> **Parameters**
>
>> **file_name** – Name of the file where randomized tree data is stored.
>>
>> **os** – Output stream associated with the file where randomized tree data is stored.

RTreeClassifier::setQuantization

Applies quantization to the current randomized tree.

C++: void **setQuantization**(int **num_quant_bits**)

> **Parameters**
>
>> **num_quant_bits** – Number of bits used for quantization.

The example below demonstrates the usage of `RTreeClassifier` for matching the features. The features are extracted from the test and train images with SURF. Output is `best_corr` and `best_corr_idx` arrays that keep the best probabilities and corresponding features indices for every train feature.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq *objectKeypoints = 0, *objectDescriptors = 0;
CvSeq *imageKeypoints = 0, *imageDescriptors = 0;
CvSURFParams params = cvSURFParams(500, 1);
```

```
cvExtractSURF( test_image, 0, &imageKeypoints, &imageDescriptors,
                storage, params );
cvExtractSURF( train_image, 0, &objectKeypoints, &objectDescriptors,
                storage, params );

RTreeClassifier detector;
int patch_width = PATCH_SIZE;
iint patch_height = PATCH_SIZE;
vector<BaseKeypoint> base_set;
int i=0;
CvSURFPoint* point;
for (i=0;i<(n_points > 0 ? n_points : objectKeypoints->total);i++)
{
        point=(CvSURFPoint*)cvGetSeqElem(objectKeypoints,i);
        base_set.push_back(
                BaseKeypoint(point->pt.x,point->pt.y,train_image));
}

        //Detector training
 RNG rng( cvGetTickCount() );
PatchGenerator gen(0,255,2,false,0.7,1.3,-CV_PI/3,CV_PI/3,
                        -CV_PI/3,CV_PI/3);

printf("RTree Classifier training...n");
detector.train(base_set,rng,gen,24,DEFAULT_DEPTH,2000,
        (int)base_set.size(), detector.DEFAULT_NUM_QUANT_BITS);
printf("Donen");

float* signature = new float[detector.original_num_classes()];
float* best_corr;
int* best_corr_idx;
if (imageKeypoints->total > 0)
{
        best_corr = new float[imageKeypoints->total];
        best_corr_idx = new int[imageKeypoints->total];
}

for(i=0; i < imageKeypoints->total; i++)
{
        point=(CvSURFPoint*)cvGetSeqElem(imageKeypoints,i);
        int part_idx = -1;
        float prob = 0.0f;

        CvRect roi = cvRect((int)(point->pt.x) - patch_width/2,
                (int)(point->pt.y) - patch_height/2,
                 patch_width, patch_height);
        cvSetImageROI(test_image, roi);
        roi = cvGetImageROI(test_image);
        if(roi.width != patch_width || roi.height != patch_height)
        {
                best_corr_idx[i] = part_idx;
                best_corr[i] = prob;
        }
        else
        {
                cvSetImageROI(test_image, roi);
                IplImage* roi_image =
                        cvCreateImage(cvSize(roi.width, roi.height),
```

```
                test_image->depth, test_image->nChannels);
        cvCopy(test_image,roi_image);

        detector.getSignature(roi_image, signature);
        for (int j = 0; j< detector.original_num_classes();j++)
        {
                if (prob < signature[j])
                {
                        part_idx = j;
                        prob = signature[j];
                }
        }

        best_corr_idx[i] = part_idx;
        best_corr[i] = prob;

        if (roi_image)
                cvReleaseImage(&roi_image);
    }
    cvResetImageROI(test_image);
}
```

7.2 Common Interfaces of Feature Detectors

Feature detectors in OpenCV have wrappers with a common interface that enables you to easily switch between different algorithms solving the same problem. All objects that implement keypoint detectors inherit the `FeatureDetector` interface.

KeyPoint

class KeyPoint

Data structure for salient point detectors.

> Point2f **pt**
> > coordinates of the keypoint

> float **size**
> > diameter of the meaningful keypoint neighborhood

> float **angle**
> > computed orientation of the keypoint (-1 if not applicable)

> float **response**
> > the response by which the most strong keypoints have been selected. Can be used for further sorting or subsampling

> int **octave**
> > octave (pyramid layer) from which the keypoint has been extracted

> int **class_id**
> > object id that can be used to clustered keypoints by an object they belong to

KeyPoint::KeyPoint

The keypoint constructors

C++: KeyPoint::**KeyPoint**()

C++: KeyPoint::**KeyPoint**(Point2f **_pt**, float **_size**, float **_angle**=-1, float **_response**=0, int **_octave**=0, int **_class_id**=-1)

C++: KeyPoint::**KeyPoint**(float **x**, float **y**, float **_size**, float **_angle**=-1, float **_response**=0, int **_octave**=0, int **_class_id**=-1)

Python: cv2.**KeyPoint**(x, y, _size[, _angle[, _response[, _octave[, _class_id]]]]) → <KeyPoint object>

> **Parameters**
>
> > **x** – x-coordinate of the keypoint
> >
> > **y** – y-coordinate of the keypoint
> >
> > **_pt** – x & y coordinates of the keypoint
> >
> > **_size** – keypoint diameter
> >
> > **_angle** – keypoint orientation
> >
> > **_response** – keypoint detector response on the keypoint (that is, strength of the keypoint)
> >
> > **_octave** – pyramid octave in which the keypoint has been detected
> >
> > **_class_id** – object id

FeatureDetector

class **FeatureDetector**

Abstract base class for 2D image feature detectors.

```
class CV_EXPORTS FeatureDetector
{
public:
    virtual ~FeatureDetector();

    void detect( const Mat& image, vector<KeyPoint>& keypoints,
                 const Mat& mask=Mat() ) const;

    void detect( const vector<Mat>& images,
                 vector<vector<KeyPoint> >& keypoints,
                 const vector<Mat>& masks=vector<Mat>() ) const;

    virtual void read(const FileNode&);
    virtual void write(FileStorage&) const;

    static Ptr<FeatureDetector> create( const string& detectorType );

protected:
...
};
```

FeatureDetector::detect

Detects keypoints in an image (first variant) or image set (second variant).

C++: void FeatureDetector::**detect**(const Mat& **image**, vector<KeyPoint>& **keypoints**, const Mat& **mask**=Mat()) const

C++: void FeatureDetector::**detect**(const vector<Mat>& **images**, vector<vector<KeyPoint>>& **keypoints**, const vector<Mat>& **masks**=vector<Mat>()) const

> **Parameters**

>> **image** – Image.

>> **images** – Image set.

>> **keypoints** – The detected keypoints. In the second variant of the method keypoints[i] is a set of keypoints detected in images[i] .

>> **mask** – Mask specifying where to look for keypoints (optional). It must be a 8-bit integer matrix with non-zero values in the region of interest.

>> **masks** – Masks for each input image specifying where to look for keypoints (optional). masks[i] is a mask for images[i].

FeatureDetector::read

Reads a feature detector object from a file node.

C++: void FeatureDetector::**read**(const FileNode& **fn**)

> **Parameters**

>> **fn** – File node from which the detector is read.

FeatureDetector::write

Writes a feature detector object to a file storage.

C++: void FeatureDetector::**write**(FileStorage& **fs**) const

> **Parameters**

>> **fs** – File storage where the detector is written.

FeatureDetector::create

Creates a feature detector by its name.

C++: Ptr<FeatureDetector> FeatureDetector::**create**(const string& **detectorType**)

> **Parameters**

>> **detectorType** – Feature detector type.

The following detector types are supported:

- "FAST" – `FastFeatureDetector`

- "STAR" – `StarFeatureDetector`

- "SIFT" – `SiftFeatureDetector`

- "SURF" – SurfFeatureDetector

- "ORB" – OrbFeatureDetector

- "MSER" – MserFeatureDetector

- "GFTT" – GoodFeaturesToTrackDetector

- "HARRIS" – GoodFeaturesToTrackDetector with Harris detector enabled

- "Dense" – DenseFeatureDetector

- "SimpleBlob" – SimpleBlobDetector

Also a combined format is supported: feature detector adapter name ("Grid" – GridAdaptedFeatureDetector, "Pyramid" – PyramidAdaptedFeatureDetector) + feature detector name (see above), for example: "GridFAST", "PyramidSTAR" .

FastFeatureDetector

class **FastFeatureDetector**

Wrapping class for feature detection using the FAST() method.

```
class FastFeatureDetector : public FeatureDetector
{
public:
    FastFeatureDetector( int threshold=1, bool nonmaxSuppression=true );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

GoodFeaturesToTrackDetector

class **GoodFeaturesToTrackDetector**

Wrapping class for feature detection using the goodFeaturesToTrack() function.

```
class GoodFeaturesToTrackDetector : public FeatureDetector
{
public:
    class Params
    {
    public:
        Params( int maxCorners=1000, double qualityLevel=0.01,
                double minDistance=1., int blockSize=3,
                bool useHarrisDetector=false, double k=0.04 );
        void read( const FileNode& fn );
        void write( FileStorage& fs ) const;

        int maxCorners;
        double qualityLevel;
        double minDistance;
        int blockSize;
        bool useHarrisDetector;
        double k;
    };
```

```
    GoodFeaturesToTrackDetector( const GoodFeaturesToTrackDetector::Params& params=
                                        GoodFeaturesToTrackDetector::Params() );
    GoodFeaturesToTrackDetector( int maxCorners, double qualityLevel,
                                 double minDistance, int blockSize=3,
                                 bool useHarrisDetector=false, double k=0.04 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

MserFeatureDetector

class MserFeatureDetector

Wrapping class for feature detection using the MSER class.

```
class MserFeatureDetector : public FeatureDetector
{
public:
    MserFeatureDetector( CvMSERParams params=cvMSERParams() );
    MserFeatureDetector( int delta, int minArea, int maxArea,
                         double maxVariation, double minDiversity,
                         int maxEvolution, double areaThreshold,
                         double minMargin, int edgeBlurSize );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

StarFeatureDetector

class StarFeatureDetector

Wrapping class for feature detection using the StarDetector class.

```
class StarFeatureDetector : public FeatureDetector
{
public:
    StarFeatureDetector( int maxSize=16, int responseThreshold=30,
                         int lineThresholdProjected = 10,
                         int lineThresholdBinarized=8, int suppressNonmaxSize=5 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

SiftFeatureDetector

class SiftFeatureDetector

Wrapping class for feature detection using the SIFT class.

```
class SiftFeatureDetector : public FeatureDetector
{
public:
    SiftFeatureDetector(
        const SIFT::DetectorParams& detectorParams=SIFT::DetectorParams(),
        const SIFT::CommonParams& commonParams=SIFT::CommonParams() );
    SiftFeatureDetector( double threshold, double edgeThreshold,
                        int nOctaves=SIFT::CommonParams::DEFAULT_NOCTAVES,
                        int nOctaveLayers=SIFT::CommonParams::DEFAULT_NOCTAVE_LAYERS,
                        int firstOctave=SIFT::CommonParams::DEFAULT_FIRST_OCTAVE,
                        int angleMode=SIFT::CommonParams::FIRST_ANGLE );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

SurfFeatureDetector

class SurfFeatureDetector

Wrapping class for feature detection using the SURF class.

```
class SurfFeatureDetector : public FeatureDetector
{
public:
    SurfFeatureDetector( double hessianThreshold = 400., int octaves = 3,
                        int octaveLayers = 4 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

OrbFeatureDetector

class OrbFeatureDetector

Wrapping class for feature detection using the ORB class.

```
class OrbFeatureDetector : public FeatureDetector
{
public:
    OrbFeatureDetector( size_t n_features );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

SimpleBlobDetector

class SimpleBlobDetector

Class for extracting blobs from an image.

```
class SimpleBlobDetector : public FeatureDetector
{
public:
struct Params
{
    Params();
    float thresholdStep;
    float minThreshold;
    float maxThreshold;
    size_t minRepeatability;
    float minDistBetweenBlobs;

    bool filterByColor;
    uchar blobColor;

    bool filterByArea;
    float minArea, maxArea;

    bool filterByCircularity;
    float minCircularity, maxCircularity;

    bool filterByInertia;
    float minInertiaRatio, maxInertiaRatio;

    bool filterByConvexity;
    float minConvexity, maxConvexity;
};

SimpleBlobDetector(const SimpleBlobDetector::Params &parameters = SimpleBlobDetector::Params());

protected:
    ...
};
```

The class implements a simple algorithm for extracting blobs from an image:

1. Convert the source image to binary images by applying thresholding with several thresholds from `minThreshold` (inclusive) to `maxThreshold` (exclusive) with distance `thresholdStep` between neighboring thresholds.

2. Extract connected components from every binary image by `findContours()` and calculate their centers.

3. Group centers from several binary images by their coordinates. Close centers form one group that corresponds to one blob, which is controlled by the `minDistBetweenBlobs` parameter.

4. From the groups, estimate final centers of blobs and their radiuses and return as locations and sizes of keypoints.

This class performs several filtrations of returned blobs. You should set `filterBy*` to true/false to turn on/off corresponding filtration. Available filtrations:

- **By color**. This filter compares the intensity of a binary image at the center of a blob to `blobColor`. If they differ, the blob is filtered out. Use `blobColor = 0` to extract dark blobs and `blobColor = 255` to extract light blobs.

- **By area**. Extracted blobs have an area between `minArea` (inclusive) and `maxArea` (exclusive).

- **By circularity**. Extracted blobs have circularity ($\frac{4*\pi*Area}{perimeter*perimeter}$) between `minCircularity` (inclusive) and `maxCircularity` (exclusive).

- **By ratio of the minimum inertia to maximum inertia**. Extracted blobs have this ratio between `minInertiaRatio` (inclusive) and `maxInertiaRatio` (exclusive).

- **By convexity**. Extracted blobs have convexity (area / area of blob convex hull) between `minConvexity` (inclusive) and `maxConvexity` (exclusive).

Default values of parameters are tuned to extract dark circular blobs.

GridAdaptedFeatureDetector

class `GridAdaptedFeatureDetector`

Class adapting a detector to partition the source image into a grid and detect points in each cell.

```cpp
class GridAdaptedFeatureDetector : public FeatureDetector
{
public:
    /*
     * detector          Detector that will be adapted.
     * maxTotalKeypoints  Maximum count of keypoints detected on the image.
     *                    Only the strongest keypoints will be kept.
     * gridRows          Grid row count.
     * gridCols          Grid column count.
     */
    GridAdaptedFeatureDetector( const Ptr<FeatureDetector>& detector,
                                int maxTotalKeypoints, int gridRows=4,
                                int gridCols=4 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

PyramidAdaptedFeatureDetector

class `PyramidAdaptedFeatureDetector`

Class adapting a detector to detect points over multiple levels of a Gaussian pyramid. Consider using this class for detectors that are not inherently scaled.

```cpp
class PyramidAdaptedFeatureDetector : public FeatureDetector
{
public:
    PyramidAdaptedFeatureDetector( const Ptr<FeatureDetector>& detector,
                                   int levels=2 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

DynamicAdaptedFeatureDetector

class `DynamicAdaptedFeatureDetector`

Adaptively adjusting detector that iteratively detects features until the desired number is found.

```
class DynamicAdaptedFeatureDetector: public FeatureDetector
{
public:
    DynamicAdaptedFeatureDetector( const Ptr<AdjusterAdapter>& adjuster,
        int min_features=400, int max_features=500, int max_iters=5 );
    ...
};
```

If the detector is persisted, it "remembers" the parameters used for the last detection. In this case, the detector may be used for consistent numbers of keypoints in a set of temporally related images, such as video streams or panorama series.

`DynamicAdaptedFeatureDetector` uses another detector, such as FAST or SURF, to do the dirty work, with the help of `AdjusterAdapter`. If the detected number of features is not large enough, `AdjusterAdapter` adjusts the detection parameters so that the next detection results in a bigger or smaller number of features. This is repeated until either the number of desired features are found or the parameters are maxed out.

Adapters can be easily implemented for any detector via the `AdjusterAdapter` interface.

Beware that this is not thread-safe since the adjustment of parameters requires modification of the feature detector class instance.

Example of creating `DynamicAdaptedFeatureDetector`:

```
//sample usage:
//will create a detector that attempts to find
//100 - 110 FAST Keypoints, and will at most run
//FAST feature detection 10 times until that
//number of keypoints are found
Ptr<FeatureDetector> detector(new DynamicAdaptedFeatureDetector (100, 110, 10,
                              new FastAdjuster(20,true)));
```

DynamicAdaptedFeatureDetector::DynamicAdaptedFeatureDetector

The constructor

C++: DynamicAdaptedFeatureDetector::**DynamicAdaptedFeatureDetector**(const
Ptr<AdjusterAdapter>&
adjuster, int
min_features, int
max_features, int
max_iters)

> **Parameters**
>
> > **adjuster** – `AdjusterAdapter` that detects features and adjusts parameters.
> >
> > **min_features** – Minimum desired number of features.
> >
> > **max_features** – Maximum desired number of features.
> >
> > **max_iters** – Maximum number of times to try adjusting the feature detector parameters. For `FastAdjuster`, this number can be high, but with `Star` or `Surf` many iterations can be time-comsuming. At each iteration the detector is rerun.

AdjusterAdapter

class **AdjusterAdapter**

Class providing an interface for adjusting parameters of a feature detector. This interface is used by `DynamicAdaptedFeatureDetector` . It is a wrapper for `FeatureDetector` that enables adjusting parameters after feature detection.

```cpp
class AdjusterAdapter: public FeatureDetector
{
public:
    virtual ~AdjusterAdapter() {}
    virtual void tooFew(int min, int n_detected) = 0;
    virtual void tooMany(int max, int n_detected) = 0;
    virtual bool good() const = 0;
    virtual Ptr<AdjusterAdapter> clone() const = 0;
    static Ptr<AdjusterAdapter> create( const string& detectorType );
};
```

See `FastAdjuster`, `StarAdjuster`, and `SurfAdjuster` for concrete implementations.

AdjusterAdapter::tooFew

Adjusts the detector parameters to detect more features.

C++: void `AdjusterAdapter::`**`tooFew`**(int **min**, int **n_detected**)

> **Parameters**
>
> > **min** – Minimum desired number of features.
> >
> > **n_detected** – Number of features detected during the latest run.

Example:

```cpp
void FastAdjuster::tooFew(int min, int n_detected)
{
        thresh_--;
}
```

AdjusterAdapter::tooMany

Adjusts the detector parameters to detect less features.

C++: void `AdjusterAdapter::`**`tooMany`**(int **max**, int **n_detected**)

> **Parameters**
>
> > **max** – Maximum desired number of features.
> >
> > **n_detected** – Number of features detected during the latest run.

Example:

```cpp
void FastAdjuster::tooMany(int min, int n_detected)
{
        thresh_++;
}
```

AdjusterAdapter::good

Returns false if the detector parameters cannot be adjusted any more.

C++: bool AdjusterAdapter::**good**() const

Example:

```
bool FastAdjuster::good() const
{
        return (thresh_ > 1) && (thresh_ < 200);
}
```

AdjusterAdapter::create

Creates an adjuster adapter by name

C++: Ptr<AdjusterAdapter> AdjusterAdapter::**create**(const string& **detectorType**)
Creates an adjuster adapter by name detectorType. The detector name is the same as in FeatureDetector::create(), but now supports "FAST", "STAR", and "SURF" only.

FastAdjuster

class **FastAdjuster**

AdjusterAdapter for FastFeatureDetector. This class decreases or increases the threshold value by 1.

```
class FastAdjuster FastAdjuster: public AdjusterAdapter
{
public:
        FastAdjuster(int init_thresh = 20, bool nonmax = true);
        ...
};
```

StarAdjuster

class **StarAdjuster**

AdjusterAdapter for StarFeatureDetector. This class adjusts the responseThreshhold of StarFeatureDetector.

```
class StarAdjuster: public AdjusterAdapter
{
        StarAdjuster(double initial_thresh = 30.0);
        ...
};
```

SurfAdjuster

class **SurfAdjuster**

AdjusterAdapter for SurfFeatureDetector. This class adjusts the hessianThreshold of SurfFeatureDetector.

```
class SurfAdjuster: public SurfAdjuster
{
        SurfAdjuster();
        ...
};
```

7.3 Common Interfaces of Descriptor Extractors

Extractors of keypoint descriptors in OpenCV have wrappers with a common interface that enables you to easily switch between different algorithms solving the same problem. This section is devoted to computing descriptors represented as vectors in a multidimensional space. All objects that implement the `vector` descriptor extractors inherit the `DescriptorExtractor` interface.

DescriptorExtractor

class `DescriptorExtractor`

Abstract base class for computing descriptors for image keypoints.

```cpp
class CV_EXPORTS DescriptorExtractor
{
public:
    virtual ~DescriptorExtractor();

    void compute( const Mat& image, vector<KeyPoint>& keypoints,
                  Mat& descriptors ) const;
    void compute( const vector<Mat>& images, vector<vector<KeyPoint> >& keypoints,
                  vector<Mat>& descriptors ) const;

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;

    virtual int descriptorSize() const = 0;
    virtual int descriptorType() const = 0;

    static Ptr<DescriptorExtractor> create( const string& descriptorExtractorType );

protected:
    ...
};
```

In this interface, a keypoint descriptor can be represented as a dense, fixed-dimension vector of a basic type. Most descriptors follow this pattern as it simplifies computing distances between descriptors. Therefore, a collection of descriptors is represented as `Mat`, where each row is a keypoint descriptor.

DescriptorExtractor::compute

Computes the descriptors for a set of keypoints detected in an image (first variant) or image set (second variant).

C++: void DescriptorExtractor::**compute**(const Mat& **image**, vector<KeyPoint>& **keypoints**, Mat& **descriptors**) const

C++: void DescriptorExtractor::**compute**(const vector<Mat>& **images**, vector<vector<KeyPoint>>& **keypoints**, vector<Mat>& **descriptors**) const

> **Parameters**
>
> > **image** – Image.
> >
> > **images** – Image set.
> >
> > **keypoints** – Input collection of keypoints. Keypoints for which a descriptor cannot be computed are removed. Sometimes new keypoints can be added, for example: SIFT duplicates keypoint with several dominant orientations (for each orientation).

> **descriptors** – Computed descriptors. In the second variant of the method `descriptors[i]` are descriptors computed for a `keypoints[i]'`. Row ''j is the `keypoints` (or `keypoints[i]`) is the descriptor for keypoint j-th keypoint.

DescriptorExtractor::read

Reads the object of a descriptor extractor from a file node.

C++: void DescriptorExtractor::**read**(const FileNode& **fn**)

> **Parameters**
>
> > **fn** – File node from which the detector is read.

DescriptorExtractor::write

Writes the object of a descriptor extractor to a file storage.

C++: void DescriptorExtractor::**write**(FileStorage& **fs**) const

> **Parameters**
>
> > **fs** – File storage where the detector is written.

DescriptorExtractor::create

Creates a descriptor extractor by name.

C++: Ptr<DescriptorExtractor> DescriptorExtractor::**create**(const string& **descriptorExtractorType**)

> **Parameters**
>
> > **descriptorExtractorType** – Descriptor extractor type.

The current implementation supports the following types of a descriptor extractor:

- `"SIFT"` – `SiftDescriptorExtractor`

- `"SURF"` – `SurfDescriptorExtractor`

- `"ORB"` – `OrbDescriptorExtractor`

- `"BRIEF"` – `BriefDescriptorExtractor`

A combined format is also supported: descriptor extractor adapter name (`"Opponent"` – `OpponentColorDescriptorExtractor`) + descriptor extractor name (see above), for example: `"OpponentSIFT"` .

SiftDescriptorExtractor

class **SiftDescriptorExtractor**

Wrapping class for computing descriptors by using the SIFT class.

```
class SiftDescriptorExtractor : public DescriptorExtractor
{
public:
    SiftDescriptorExtractor(
        const SIFT::DescriptorParams& descriptorParams=SIFT::DescriptorParams(),
        const SIFT::CommonParams& commonParams=SIFT::CommonParams() );
```

```
    SiftDescriptorExtractor( double magnification, bool isNormalize=true,
        bool recalculateAngles=true, int nOctaves=SIFT::CommonParams::DEFAULT_NOCTAVES,
        int nOctaveLayers=SIFT::CommonParams::DEFAULT_NOCTAVE_LAYERS,
        int firstOctave=SIFT::CommonParams::DEFAULT_FIRST_OCTAVE,
        int angleMode=SIFT::CommonParams::FIRST_ANGLE );

    virtual void read (const FileNode &fn);
    virtual void write (FileStorage &fs) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

SurfDescriptorExtractor

class `SurfDescriptorExtractor`

Wrapping class for computing descriptors by using the SURF class.

```
class SurfDescriptorExtractor : public DescriptorExtractor
{
public:
    SurfDescriptorExtractor( int nOctaves=4,
                             int nOctaveLayers=2, bool extended=false );

    virtual void read (const FileNode &fn);
    virtual void write (FileStorage &fs) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

OrbDescriptorExtractor

class `OrbDescriptorExtractor`

Wrapping class for computing descriptors by using the ORB class.

```
template<typename T>
class ORbDescriptorExtractor : public DescriptorExtractor
{
public:
    OrbDescriptorExtractor( ORB::PatchSize patch_size );

    virtual void read( const FileNode &fn );
    virtual void write( FileStorage &fs ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

CalonderDescriptorExtractor

class `CalonderDescriptorExtractor`

Wrapping class for computing descriptors by using the RTreeClassifier class.

```cpp
template<typename T>
class CalonderDescriptorExtractor : public DescriptorExtractor
{
public:
    CalonderDescriptorExtractor( const string& classifierFile );

    virtual void read( const FileNode &fn );
    virtual void write( FileStorage &fs ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

OpponentColorDescriptorExtractor

class `OpponentColorDescriptorExtractor`

Class adapting a descriptor extractor to compute descriptors in the Opponent Color Space (refer to Van de Sande et al., CGIV 2008 *Color Descriptors for Object Category Recognition*). Input RGB image is transformed in the Opponent Color Space. Then, an unadapted descriptor extractor (set in the constructor) computes descriptors on each of three channels and concatenates them into a single color descriptor.

```cpp
class OpponentColorDescriptorExtractor : public DescriptorExtractor
{
public:
    OpponentColorDescriptorExtractor( const Ptr<DescriptorExtractor>& dextractor );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
};
```

BriefDescriptorExtractor

class `BriefDescriptorExtractor`

Class for computing BRIEF descriptors described in a paper of Calonder M., Lepetit V., Strecha C., Fua P. *BRIEF: Binary Robust Independent Elementary Features* , 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.

```cpp
class BriefDescriptorExtractor : public DescriptorExtractor
{
public:
    static const int PATCH_SIZE = 48;
    static const int KERNEL_SIZE = 9;
```

```
    // bytes is a length of descriptor in bytes. It can be equal 16, 32 or 64 bytes.
    BriefDescriptorExtractor( int bytes = 32 );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
};
```

7.4 Common Interfaces of Descriptor Matchers

Matchers of keypoint descriptors in OpenCV have wrappers with a common interface that enables you to easily switch between different algorithms solving the same problem. This section is devoted to matching descriptors that are represented as vectors in a multidimensional space. All objects that implement vector descriptor matchers inherit the DescriptorMatcher interface.

DMatch

class **DMatch**

Class for matching keypoint descriptors: query descriptor index, train descriptor index, train image index, and distance between descriptors.

```
struct DMatch
{
    DMatch() : queryIdx(-1), trainIdx(-1), imgIdx(-1),
               distance(std::numeric_limits<float>::max()) {}
    DMatch( int _queryIdx, int _trainIdx, float _distance ) :
            queryIdx(_queryIdx), trainIdx(_trainIdx), imgIdx(-1),
            distance(_distance) {}
    DMatch( int _queryIdx, int _trainIdx, int _imgIdx, float _distance ) :
            queryIdx(_queryIdx), trainIdx(_trainIdx), imgIdx(_imgIdx),
            distance(_distance) {}

    int queryIdx; // query descriptor index
    int trainIdx; // train descriptor index
    int imgIdx;   // train image index

    float distance;

    // less is better
    bool operator<( const DMatch &m ) const;
};
```

DescriptorMatcher

class **DescriptorMatcher**

Abstract base class for matching keypoint descriptors. It has two groups of match methods: for matching descriptors of an image with another image or with an image set.

```
class DescriptorMatcher
{
public:
    virtual ~DescriptorMatcher();

    virtual void add( const vector<Mat>& descriptors );

    const vector<Mat>& getTrainDescriptors() const;
    virtual void clear();
    bool empty() const;
    virtual bool isMaskSupported() const = 0;

    virtual void train();

    /*
     * Group of methods to match descriptors from an image pair.
     */
    void match( const Mat& queryDescriptors, const Mat& trainDescriptors,
                vector<DMatch>& matches, const Mat& mask=Mat() ) const;
    void knnMatch( const Mat& queryDescriptors, const Mat& trainDescriptors,
                   vector<vector<DMatch> >& matches, int k,
                   const Mat& mask=Mat(), bool compactResult=false ) const;
    void radiusMatch( const Mat& queryDescriptors, const Mat& trainDescriptors,
                      vector<vector<DMatch> >& matches, float maxDistance,
                      const Mat& mask=Mat(), bool compactResult=false ) const;
    /*
     * Group of methods to match descriptors from one image to an image set.
     */
    void match( const Mat& queryDescriptors, vector<DMatch>& matches,
                const vector<Mat>& masks=vector<Mat>() );
    void knnMatch( const Mat& queryDescriptors, vector<vector<DMatch> >& matches,
                   int k, const vector<Mat>& masks=vector<Mat>(),
                   bool compactResult=false );
    void radiusMatch( const Mat& queryDescriptors, vector<vector<DMatch> >& matches,
                      float maxDistance, const vector<Mat>& masks=vector<Mat>(),
                      bool compactResult=false );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;

    virtual Ptr<DescriptorMatcher> clone( bool emptyTrainData=false ) const = 0;

    static Ptr<DescriptorMatcher> create( const string& descriptorMatcherType );

protected:
    vector<Mat> trainDescCollection;
    ...
};
```

DescriptorMatcher::add

Adds descriptors to train a descriptor collection. If the collection trainDescCollectionis is not empty, the new descriptors are added to existing train descriptors.

C++: void DescriptorMatcher::**add**(const vector<Mat>& **descriptors**)

   Parameters

> **descriptors** – Descriptors to add. Each `descriptors[i]` is a set of descriptors from the same train image.

DescriptorMatcher::getTrainDescriptors

Returns a constant link to the train descriptor collection `trainDescCollection` .

C++: const vector<Mat>& DescriptorMatcher::**getTrainDescriptors**() const

DescriptorMatcher::clear

Clears the train descriptor collection.

C++: void DescriptorMatcher::**clear**()

DescriptorMatcher::empty

Returns true if there are no train descriptors in the collection.

C++: bool DescriptorMatcher::**empty**() const

DescriptorMatcher::isMaskSupported

Returns true if the descriptor matcher supports masking permissible matches.

C++: bool DescriptorMatcher::**isMaskSupported**()

DescriptorMatcher::train

Trains a descriptor matcher

C++: void DescriptorMatcher::**train**()

Trains a descriptor matcher (for example, the flann index). In all methods to match, the method `train()` is run every time before matching. Some descriptor matchers (for example, `BruteForceMatcher`) have an empty implementation of this method. Other matchers really train their inner structures (for example, `FlannBasedMatcher` trains `flann::Index`).

DescriptorMatcher::match

Finds the best match for each descriptor from a query set.

C++: void DescriptorMatcher::**match**(const Mat& **queryDescriptors**, const Mat& **trainDescriptors**, vector<DMatch>& **matches**, const Mat& **mask**=Mat()) const

C++: void DescriptorMatcher::**match**(const Mat& **queryDescriptors**, vector<DMatch>& **matches**, const vector<Mat>& **masks**=vector<Mat>())

> **Parameters**

>> **queryDescriptors** – Query set of descriptors.

>> **trainDescriptors** – Train set of descriptors. This set is not added to the train descriptors collection stored in the class object.

> **matches** – Matches. If a query descriptor is masked out in `mask` , no match is added for this
> descriptor. So, `matches` size may be smaller than the query descriptors count.
>
> **mask** – Mask specifying permissible matches between an input query and train matrices of
> descriptors.
>
> **masks** – Set of masks. Each `masks[i]` specifies permissible matches be-
> tween the input query descriptors and stored train descriptors from the i-th image
> `trainDescCollection[i]`.

In the first variant of this method, the train descriptors are passed as an input argument. In the second variant of the
method, train descriptors collection that was set by `DescriptorMatcher::add` is used. Optional mask (or masks) can
be passed to specify which query and training descriptors can be matched. Namely, `queryDescriptors[i]` can be
matched with `trainDescriptors[j]` only if `mask.at<uchar>(i,j)` is non-zero.

DescriptorMatcher::knnMatch

Finds the k best matches for each descriptor from a query set.

C++: void `DescriptorMatcher::knnMatch`(const Mat& **queryDescriptors**, const Mat& **trainDescrip-**
tors, vector<vector<DMatch>>& **matches**, int **k**, const Mat&
mask=Mat(), bool **compactResult**=false) const

C++: void `DescriptorMatcher::knnMatch`(const Mat& **queryDescriptors**, vector<vector<DMatch>>&
matches, int **k**, const vector<Mat>& **masks**=vector<Mat>(),
bool **compactResult**=false)

> **Parameters**
>
>> **queryDescriptors** – Query set of descriptors.
>>
>> **trainDescriptors** – Train set of descriptors. This set is not added to the train descriptors
>> collection stored in the class object.
>>
>> **mask** – Mask specifying permissible matches between an input query and train matrices of
>> descriptors.
>>
>> **masks** – Set of masks. Each `masks[i]` specifies permissible matches be-
>> tween the input query descriptors and stored train descriptors from the i-th image
>> `trainDescCollection[i]`.
>>
>> **matches** – Matches. Each `matches[i]` is k or less matches for the same query descriptor.
>>
>> **k** – Count of best matches found per each query descriptor or less if a query descriptor has
>> less than k possible matches in total.
>>
>> **compactResult** – Parameter used when the mask (or masks) is not empty. If
>> `compactResult` is false, the `matches` vector has the same size as `queryDescriptors`
>> rows. If `compactResult` is true, the `matches` vector does not contain matches for fully
>> masked-out query descriptors.

These extended variants of `DescriptorMatcher::match()` methods find several best matches for each query descrip-
tor. The matches are returned in the distance increasing order. See `DescriptorMatcher::match()` for the details
about query and train descriptors.

DescriptorMatcher::radiusMatch

For each query descriptor, finds the training descriptors not farther than the specified distance.

C++: void DescriptorMatcher::**radiusMatch**(const Mat& **queryDescriptors**, const Mat& **trainDescriptors**, vector<vector<DMatch>>& **matches**, float **maxDistance**, const Mat& **mask**=Mat(), bool **compactResult**=false) const

C++: void DescriptorMatcher::**radiusMatch**(const Mat& **queryDescriptors**, vector<vector<DMatch>>& **matches**, float **maxDistance**, const vector<Mat>& **masks**=vector<Mat>(), bool **compactResult**=false)

> **Parameters**
>
>> **queryDescriptors** – Query set of descriptors.
>>
>> **trainDescriptors** – Train set of descriptors. This set is not added to the train descriptors collection stored in the class object.
>>
>> **mask** – Mask specifying permissible matches between an input query and train matrices of descriptors.
>>
>> **masks** – Set of masks. Each masks[i] specifies permissible matches between the input query descriptors and stored train descriptors from the i-th image trainDescCollection[i].
>>
>> **matches** – Found matches.
>>
>> **compactResult** – Parameter used when the mask (or masks) is not empty. If compactResult is false, the matches vector has the same size as queryDescriptors rows. If compactResult is true, the matches vector does not contain matches for fully masked-out query descriptors.
>>
>> **maxDistance** – Threshold for the distance between matched descriptors.

For each query descriptor, the methods find such training descriptors that the distance between the query descriptor and the training descriptor is equal or smaller than maxDistance. Found matches are returned in the distance increasing order.

DescriptorMatcher::clone

Clones the matcher.

C++: Ptr<DescriptorMatcher> DescriptorMatcher::**clone**(bool **emptyTrainData**) const

> **Parameters**
>
>> **emptyTrainData** – If emptyTrainData is false, the method creates a deep copy of the object, that is, copies both parameters and train data. If emptyTrainData is true, the method creates an object copy with the current parameters but with empty train data.

DescriptorMatcher::create

Creates a descriptor matcher of a given type with the default parameters (using default constructor).

C++: Ptr<DescriptorMatcher> DescriptorMatcher::**create**(const string& **descriptorMatcherType**)

> **Parameters**
>
>> **descriptorMatcherType** – Descriptor matcher type. Now the following matcher types are supported:
>>
>> – BruteForce (it uses L2)

– BruteForce-L1

– BruteForce-Hamming

– BruteForce-HammingLUT

– FlannBased

BruteForceMatcher

class BruteForceMatcher

Brute-force descriptor matcher. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. This descriptor matcher supports masking permissible matches of descriptor sets.

```cpp
template<class Distance>
class BruteForceMatcher : public DescriptorMatcher
{
public:
    BruteForceMatcher( Distance d = Distance() );
    virtual ~BruteForceMatcher();

    virtual bool isMaskSupported() const;
    virtual Ptr<DescriptorMatcher> clone( bool emptyTrainData=false ) const;
protected:
    ...
}
```

For efficiency, `BruteForceMatcher` is used as a template parameterized with the distance type. For float descriptors, `L2<float>` is a common choice. The following distances are supported:

```cpp
template<typename T>
struct Accumulator
{
    typedef T Type;
};

template<> struct Accumulator<unsigned char>  { typedef unsigned int Type; };
template<> struct Accumulator<unsigned short> { typedef unsigned int Type; };
template<> struct Accumulator<char>   { typedef int Type; };
template<> struct Accumulator<short>  { typedef int Type; };

/*
 * Euclidean distance functor
 */
template<class T>
struct L2
{
    typedef T ValueType;
    typedef typename Accumulator<T>::Type ResultType;

    ResultType operator()( const T* a, const T* b, int size ) const;
};

/*
 * Squared Euclidean distance functor
 */
template<class T>
struct SL2
```

```
{
    typedef T ValueType;
    typedef typename Accumulator<T>::Type ResultType;

    ResultType operator()( const T* a, const T* b, int size ) const;
};
// Note: in case of SL2 distance a parameter maxDistance in the method DescriptorMatcher::radiusMatch
// is a squared maximum distance in L2.

/*
 * Manhattan distance (city block distance) functor
 */
template<class T>
struct CV_EXPORTS L1
{
    typedef T ValueType;
    typedef typename Accumulator<T>::Type ResultType;

    ResultType operator()( const T* a, const T* b, int size ) const;
};

/*
 * Hamming distance functor
 */
struct HammingLUT
{
    typedef unsigned char ValueType;
    typedef int ResultType;

    ResultType operator()( const unsigned char* a, const unsigned char* b,
                           int size ) const;
    ...
};

struct Hamming
{
    typedef unsigned char ValueType;
    typedef int ResultType;

    ResultType operator()( const unsigned char* a, const unsigned char* b,
                           int size ) const;
};
```

FlannBasedMatcher

class **FlannBasedMatcher**

Flann-based descriptor matcher. This matcher trains `flann::Index_` on a train descriptor collection and calls its nearest search methods to find the best matches. So, this matcher may be faster when matching a large train collection than the brute force matcher. `FlannBasedMatcher` does not support masking permissible matches of descriptor sets because `flann::Index` does not support this.

```
class FlannBasedMatcher : public DescriptorMatcher
{
public:
    FlannBasedMatcher(
      const Ptr<flann::IndexParams>& indexParams=new flann::KDTreeIndexParams(),
```

```
      const Ptr<flann::SearchParams>& searchParams=new flann::SearchParams() );

    virtual void add( const vector<Mat>& descriptors );
    virtual void clear();

    virtual void train();
    virtual bool isMaskSupported() const;

    virtual Ptr<DescriptorMatcher> clone( bool emptyTrainData=false ) const;
protected:
    ...
};
```

7.5 Common Interfaces of Generic Descriptor Matchers

Matchers of keypoint descriptors in OpenCV have wrappers with a common interface that enables you to easily switch between different algorithms solving the same problem. This section is devoted to matching descriptors that cannot be represented as vectors in a multidimensional space. GenericDescriptorMatcher is a more generic interface for descriptors. It does not make any assumptions about descriptor representation. Every descriptor with the DescriptorExtractor interface has a wrapper with the GenericDescriptorMatcher interface (see VectorDescriptorMatcher). There are descriptors such as the One-way descriptor and Ferns that have the GenericDescriptorMatcher interface implemented but do not support DescriptorExtractor.

GenericDescriptorMatcher

class GenericDescriptorMatcher

Abstract interface for extracting and matching a keypoint descriptor. There are also DescriptorExtractor and DescriptorMatcher for these purposes but their interfaces are intended for descriptors represented as vectors in a multidimensional space. GenericDescriptorMatcher is a more generic interface for descriptors. DescriptorMatcher and GenericDescriptorMatcher have two groups of match methods: for matching keypoints of an image with another image or with an image set.

```
class GenericDescriptorMatcher
{
public:
    GenericDescriptorMatcher();
    virtual ~GenericDescriptorMatcher();

    virtual void add( const vector<Mat>& images,
                      vector<vector<KeyPoint> >& keypoints );

    const vector<Mat>& getTrainImages() const;
    const vector<vector<KeyPoint> >& getTrainKeypoints() const;
    virtual void clear();

    virtual void train() = 0;

    virtual bool isMaskSupported() = 0;

    void classify( const Mat& queryImage,
                   vector<KeyPoint>& queryKeypoints,
                   const Mat& trainImage,
                   vector<KeyPoint>& trainKeypoints ) const;
```

```
    void classify( const Mat& queryImage,
                   vector<KeyPoint>& queryKeypoints );

    /*
     * Group of methods to match keypoints from an image pair.
     */
    void match( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
                vector<DMatch>& matches, const Mat& mask=Mat() ) const;
    void knnMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                   const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
                   vector<vector<DMatch> >& matches, int k,
                   const Mat& mask=Mat(), bool compactResult=false ) const;
    void radiusMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                      const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
                      vector<vector<DMatch> >& matches, float maxDistance,
                      const Mat& mask=Mat(), bool compactResult=false ) const;
    /*
     * Group of methods to match keypoints from one image to an image set.
     */
    void match( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                vector<DMatch>& matches, const vector<Mat>& masks=vector<Mat>() );
    void knnMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                   vector<vector<DMatch> >& matches, int k,
                   const vector<Mat>& masks=vector<Mat>(), bool compactResult=false );
    void radiusMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                      vector<vector<DMatch> >& matches, float maxDistance,
                      const vector<Mat>& masks=vector<Mat>(), bool compactResult=false );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const = 0;

protected:
    ...
};
```

GenericDescriptorMatcher::add

Adds images and their keypoints to the training collection stored in the class instance.

C++: void GenericDescriptorMatcher::**add**(const vector<Mat>& **images**, vector<vector<KeyPoint>>& **keypoints**)

> **Parameters**
>
> > **images** – Image collection.
> >
> > **keypoints** – Point collection. It is assumed that keypoints[i] are keypoints detected in the image images[i] .

GenericDescriptorMatcher::getTrainImages

Returns a train image collection.

C++: const vector<Mat>& GenericDescriptorMatcher::**getTrainImages**() const

---

GenericDescriptorMatcher::getTrainKeypoints

Returns a train keypoints collection.

C++: const vector<vector<KeyPoint>>& GenericDescriptorMatcher::**getTrainKeypoints**() const

GenericDescriptorMatcher::clear

Clears a train collection (images and keypoints).

C++: void GenericDescriptorMatcher::**clear**()

GenericDescriptorMatcher::train

Trains descriptor matcher

C++: void GenericDescriptorMatcher::**train**()

Prepares descriptor matcher, for example, creates a tree-based structure, to extract descriptors or to optimize descriptors matching.

GenericDescriptorMatcher::isMaskSupported

Returns `true` if a generic descriptor matcher supports masking permissible matches.

C++: void GenericDescriptorMatcher::**isMaskSupported**()

GenericDescriptorMatcher::classify

Classifies keypoints from a query set.

C++: void GenericDescriptorMatcher::**classify**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, const Mat& **trainImage**, vector<KeyPoint>& **trainKeypoints**) const

C++: void GenericDescriptorMatcher::**classify**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**)

>   **Parameters**
>
>>   **queryImage** – Query image.
>>
>>   **queryKeypoints** – Keypoints from a query image.
>>
>>   **trainImage** – Train image.
>>
>>   **trainKeypoints** – Keypoints from a train image.

The method classifies each keypoint from a query set. The first variant of the method takes a train image and its keypoints as an input argument. The second variant uses the internally stored training collection that can be built using the `GenericDescriptorMatcher::add` method.

The methods do the following:

1. Call the `GenericDescriptorMatcher::match` method to find correspondence between the query set and the training set.

2. Set the `class_id` field of each keypoint from the query set to `class_id` of the corresponding keypoint from the training set.

GenericDescriptorMatcher::match

Finds the best match in the training set for each keypoint from the query set.

C++: void GenericDescriptorMatcher::**match**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, const Mat& **trainImage**, vector<KeyPoint>& **trainKeypoints**, vector<DMatch>& **matches**, const Mat& **mask**=Mat()) const

C++: void GenericDescriptorMatcher::**match**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, vector<DMatch>& **matches**, const vector<Mat>& **masks**=vector<Mat>())

> Parameters

>> **queryImage** – Query image.

>> **queryKeypoints** – Keypoints detected in `queryImage` .

>> **trainImage** – Train image. It is not added to a train image collection stored in the class object.

>> **trainKeypoints** – Keypoints detected in `trainImage` . They are not added to a train points collection stored in the class object.

>> **matches** – Matches. If a query descriptor (keypoint) is masked out in `mask` , match is added for this descriptor. So, `matches` size may be smaller than the query keypoints count.

>> **mask** – Mask specifying permissible matches between an input query and train keypoints.

>> **masks** – Set of masks. Each `masks[i]` specifies permissible matches between input query keypoints and stored train keypoints from the i-th image.

The methods find the best match for each query keypoint. In the first variant of the method, a train image and its keypoints are the input arguments. In the second variant, query keypoints are matched to the internally stored training collection that can be built using the `GenericDescriptorMatcher::add` method. Optional mask (or masks) can be passed to specify which query and training descriptors can be matched. Namely, `queryKeypoints[i]` can be matched with `trainKeypoints[j]` only if `mask.at<uchar>(i,j)` is non-zero.

GenericDescriptorMatcher::knnMatch

Finds the `k` best matches for each query keypoint.

C++: void GenericDescriptorMatcher::**knnMatch**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, const Mat& **trainImage**, vector<KeyPoint>& **trainKeypoints**, vector<vector<DMatch>>& **matches**, int **k**, const Mat& **mask**=Mat(), bool **compactResult**=false) const

C++: void GenericDescriptorMatcher::**knnMatch**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, vector<vector<DMatch>>& **matches**, int **k**, const vector<Mat>& **masks**=vector<Mat>(), bool **compactResult**=false)

The methods are extended variants of `GenericDescriptorMatch::match`. The parameters are similar, and the the semantics is similar to `DescriptorMatcher::knnMatch`. But this class does not require explicitly computed keypoint descriptors.

GenericDescriptorMatcher::radiusMatch

For each query keypoint, finds the training keypoints not farther than the specified distance.

C++: void GenericDescriptorMatcher::**radiusMatch**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, const Mat& **trainImage**, vector<KeyPoint>& **trainKeypoints**, vector<vector<DMatch>>& **matches**, float **maxDistance**, const Mat& **mask**=Mat(), bool **compactResult**=false) const

C++: void GenericDescriptorMatcher::**radiusMatch**(const Mat& **queryImage**, vector<KeyPoint>& **queryKeypoints**, vector<vector<DMatch>>& **matches**, float **maxDistance**, const vector<Mat>& **masks**=vector<Mat>(), bool **compactResult**=false)

The methods are similar to `DescriptorMatcher::radius`. But this class does not require explicitly computed keypoint descriptors.

GenericDescriptorMatcher::read

Reads a matcher object from a file node.

C++: void GenericDescriptorMatcher::**read**(const FileNode& **fn**)

GenericDescriptorMatcher::write

Writes a match object to a file storage.

C++: void GenericDescriptorMatcher::**write**(FileStorage& **fs**) const

GenericDescriptorMatcher::clone

Clones the matcher.

C++: Ptr<GenericDescriptorMatcher> GenericDescriptorMatcher::**clone**(bool **emptyTrainData**) const

> **Parameters**
>
> > **emptyTrainData** – If `emptyTrainData` is false, the method creates a deep copy of the object, that is, copies both parameters and train data. If `emptyTrainData` is true, the method creates an object copy with the current parameters but with empty train data.

OneWayDescriptorMatcher

class `OneWayDescriptorMatcher`

Wrapping class for computing, matching, and classifying descriptors using the `OneWayDescriptorBase` class.

```
class OneWayDescriptorMatcher : public GenericDescriptorMatcher
{
public:
    class Params
    {
```

```cpp
public:
    static const int POSE_COUNT = 500;
    static const int PATCH_WIDTH = 24;
    static const int PATCH_HEIGHT = 24;
    static float GET_MIN_SCALE() { return 0.7f; }
    static float GET_MAX_SCALE() { return 1.5f; }
    static float GET_STEP_SCALE() { return 1.2f; }

    Params( int poseCount = POSE_COUNT,
            Size patchSize = Size(PATCH_WIDTH, PATCH_HEIGHT),
            string pcaFilename = string(),
            string trainPath = string(), string trainImagesList = string(),
            float minScale = GET_MIN_SCALE(), float maxScale = GET_MAX_SCALE(),
            float stepScale = GET_STEP_SCALE() );

    int poseCount;
    Size patchSize;
    string pcaFilename;
    string trainPath;
    string trainImagesList;

    float minScale, maxScale, stepScale;
};

OneWayDescriptorMatcher( const Params& params=Params() );
virtual ~OneWayDescriptorMatcher();

void initialize( const Params& params, const Ptr<OneWayDescriptorBase>& base=Ptr<OneWayDescriptorBase>() );

// Clears keypoints stored in collection and OneWayDescriptorBase
virtual void clear();

virtual void train();

virtual bool isMaskSupported();

virtual void read( const FileNode &fn );
virtual void write( FileStorage& fs ) const;

virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const;
protected:
    ...
};
```

FernDescriptorMatcher

class FernDescriptorMatcher

Wrapping class for computing, matching, and classifying descriptors using the `FernClassifier` class.

```cpp
class FernDescriptorMatcher : public GenericDescriptorMatcher
{
public:
    class Params
    {
    public:
        Params( int nclasses=0,
```

```
                int patchSize=FernClassifier::PATCH_SIZE,
                int signatureSize=FernClassifier::DEFAULT_SIGNATURE_SIZE,
                int nstructs=FernClassifier::DEFAULT_STRUCTS,
                int structSize=FernClassifier::DEFAULT_STRUCT_SIZE,
                int nviews=FernClassifier::DEFAULT_VIEWS,
                int compressionMethod=FernClassifier::COMPRESSION_NONE,
                const PatchGenerator& patchGenerator=PatchGenerator() );

        Params( const string& filename );

        int nclasses;
        int patchSize;
        int signatureSize;
        int nstructs;
        int structSize;
        int nviews;
        int compressionMethod;
        PatchGenerator patchGenerator;

        string filename;
    };

    FernDescriptorMatcher( const Params& params=Params() );
    virtual ~FernDescriptorMatcher();

    virtual void clear();

    virtual void train();

    virtual bool isMaskSupported();

    virtual void read( const FileNode &fn );
    virtual void write( FileStorage& fs ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const;

protected:
        ...
};
```

VectorDescriptorMatcher

class VectorDescriptorMatcher

Class used for matching descriptors that can be described as vectors in a finite-dimensional space.

```
class CV_EXPORTS VectorDescriptorMatcher : public GenericDescriptorMatcher
{
public:
    VectorDescriptorMatcher( const Ptr<DescriptorExtractor>& extractor, const Ptr<DescriptorMatcher>& matcher );
    virtual ~VectorDescriptorMatcher();

    virtual void add( const vector<Mat>& imgCollection,
                      vector<vector<KeyPoint> >& pointCollection );
    virtual void clear();
    virtual void train();
    virtual bool isMaskSupported();
```

```
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const;

protected:
    ...
};
```

Example:

```
VectorDescriptorMatcher matcher( new SurfDescriptorExtractor,
                                 new BruteForceMatcher<L2<float> > );
```

7.6 Drawing Function of Keypoints and Matches

drawMatches

Draws the found matches of keypoints from two images.

C++: void **drawMatches**(const Mat& **img1**, const vector<KeyPoint>& **keypoints1**, const Mat& **img2**, const vector<KeyPoint>& **keypoints2**, const vector<DMatch>& **matches1to2**, Mat& **outImg**, const Scalar& **matchColor**=Scalar::all(-1), const Scalar& **singlePointColor**=Scalar::all(-1), const vector<char>& **matchesMask**=vector<char>(), int **flags**=DrawMatchesFlags::DEFAULT)

C++: void **drawMatches**(const Mat& **img1**, const vector<KeyPoint>& **keypoints1**, const Mat& **img2**, const vector<KeyPoint>& **keypoints2**, const vector<vector<DMatch>>& **matches1to2**, Mat& **outImg**, const Scalar& **matchColor**=Scalar::all(-1), const Scalar& **singlePointColor**=Scalar::all(-1), const vector<vector<char>>& **matchesMask**=vector<vector<char> >(), int **flags**=DrawMatchesFlags::DEFAULT)

   Parameters

   img1 – First source image.

   keypoints1 – Keypoints from the first source image.

   img2 – Second source image.

   keypoints2 – Keypoints from the second source image.

   matches – Matches from the first image to the second one, which means that keypoints1[i] has a corresponding point in keypoints2[matches[i]] .

   outImg – Output image. Its content depends on the flags value defining what is drawn in the output image. See possible flags bit values below.

   matchColor – Color of matches (lines and connected keypoints). If matchColor==Scalar::all(-1) , the color is generated randomly.

   singlePointColor – Color of single keypoints (circles), which means that keypoints do not have the matches. If singlePointColor==Scalar::all(-1) , the color is generated randomly.

   matchesMask – Mask determining which matches are drawn. If the mask is empty, all matches are drawn.

> **flags** – Flags setting drawing features. Possible `flags` bit values are defined by `DrawMatchesFlags`.

This function draws matches of keypoints from two images in the output image. Match is a line connecting two keypoints (circles). The structure `DrawMatchesFlags` is defined as follows:

```
struct DrawMatchesFlags
{
    enum
    {
        DEFAULT = 0, // Output image matrix will be created (Mat::create),
                     // i.e. existing memory of output image may be reused.
                     // Two source images, matches, and single keypoints
                     // will be drawn.
                     // For each keypoint, only the center point will be
                     // drawn (without a circle around the keypoint with the
                     // keypoint size and orientation).
        DRAW_OVER_OUTIMG = 1, // Output image matrix will not be
                     // created (using Mat::create). Matches will be drawn
                     // on existing content of output image.
        NOT_DRAW_SINGLE_POINTS = 2, // Single keypoints will not be drawn.
        DRAW_RICH_KEYPOINTS = 4 // For each keypoint, the circle around
                     // keypoint with keypoint size and orientation will
                     // be drawn.
    };
};
```

drawKeypoints

Draws keypoints.

C++: void **drawKeypoints**(const Mat& **image**, const vector<KeyPoint>& **keypoints**, Mat& **outImg**, const Scalar& **color**=Scalar::all(-1), int **flags**=DrawMatchesFlags::DEFAULT)

> **Parameters**
>
> > **image** – Source image.
> >
> > **keypoints** – Keypoints from the source image.
> >
> > **outImg** – Output image. Its content depends on the `flags` value defining what is drawn in the output image. See possible `flags` bit values below.
> >
> > **color** – Color of keypoints.
> >
> > **flags** – Flags setting drawing features. Possible `flags` bit values are defined by `DrawMatchesFlags`. See details above in `drawMatches()`.

7.7 Object Categorization

This section describes approaches based on local 2D features and used to categorize objects.

BOWTrainer

class **BOWTrainer**

Abstract base class for training the *bag of visual words* vocabulary from a set of descriptors. For details, see, for example, *Visual Categorization with Bags of Keypoints* by Gabriella Csurka, Christopher R. Dance, Lixin Fan, Jutta Willamowski, Cedric Bray, 2004.

```cpp
class BOWTrainer
{
public:
    BOWTrainer(){}
    virtual ~BOWTrainer(){}

    void add( const Mat& descriptors );
    const vector<Mat>& getDescriptors() const;
    int descripotorsCount() const;

    virtual void clear();

    virtual Mat cluster() const = 0;
    virtual Mat cluster( const Mat& descriptors ) const = 0;

protected:
    ...
};
```

BOWTrainer::add

Adds descriptors to a training set.

C++: void BOWTrainer::**add**(const Mat& **descriptors**)

> **Parameters**
>
> > **descriptors** – Descriptors to add to a training set. Each row of the `descriptors` matrix is a descriptor.

The training set is clustered using `clustermethod` to construct the vocabulary.

BOWTrainer::getDescriptors

Returns a training set of descriptors.

C++: const vector<Mat>& BOWTrainer::**getDescriptors**() const

BOWTrainer::descripotorsCount

Returns the count of all descriptors stored in the training set.

C++: const vector<Mat>& BOWTrainer::**descripotorsCount**() const

BOWTrainer::cluster

Clusters train descriptors.

C++: Mat BOWTrainer::**cluster**() const

C++: Mat BOWTrainer::**cluster**(const Mat& **descriptors**) const

---

> **Parameters**
>> **descriptors** – Descriptors to cluster. Each row of the `descriptors` matrix is a descriptor. Descriptors are not added to the inner train descriptor set.

The vocabulary consists of cluster centers. So, this method returns the vocabulary. In the first variant of the method, train descriptors stored in the object are clustered. In the second variant, input descriptors are clustered.

BOWKMeansTrainer

class **BOWKMeansTrainer**

`kmeans()` -based class to train visual vocabulary using the *bag of visual words* approach.

```
class BOWKMeansTrainer : public BOWTrainer
{
public:
    BOWKMeansTrainer( int clusterCount, const TermCriteria& termcrit=TermCriteria(),
                      int attempts=3, int flags=KMEANS_PP_CENTERS );
    virtual ~BOWKMeansTrainer(){}

    // Returns trained vocabulary (i.e. cluster centers).
    virtual Mat cluster() const;
    virtual Mat cluster( const Mat& descriptors ) const;

protected:
    ...
};
```

BOWKMeansTrainer::BOWKMeansTrainer

The constructor.

C++: BOWKMeansTrainer::**BOWKMeansTrainer**(int **clusterCount**, const TermCriteria& **termcrit**=TermCriteria(), int **attempts**=3, int **flags**=KMEANS_PP_CENTERS)

See `kmeans()` function parameters.

BOWImgDescriptorExtractor

class **BOWImgDescriptorExtractor**

Class to compute an image descriptor using the *bag of visual words*. Such a computation consists of the following steps:

1. Compute descriptors for a given image and its keypoints set.

2. Find the nearest visual words from the vocabulary for each keypoint descriptor.

3. Compute the bag-of-words image descriptor as is a normalized histogram of vocabulary words encountered in the image. The `i`-th bin of the histogram is a frequency of `i`-th word of the vocabulary in the given image.

The class declaration is the following:

```
class BOWImgDescriptorExtractor
{
public:
    BOWImgDescriptorExtractor( const Ptr<DescriptorExtractor>& dextractor,
```

```
                                const Ptr<DescriptorMatcher>& dmatcher );
    virtual ~BOWImgDescriptorExtractor(){}

    void setVocabulary( const Mat& vocabulary );
    const Mat& getVocabulary() const;
    void compute( const Mat& image, vector<KeyPoint>& keypoints,
                  Mat& imgDescriptor,
                  vector<vector<int> >* pointIdxsOfClusters=0,
                  Mat* descriptors=0 );
    int descriptorSize() const;
    int descriptorType() const;

protected:
    ...
};
```

BOWImgDescriptorExtractor::BOWImgDescriptorExtractor

The constructor.

C++: BOWImgDescriptorExtractor::**BOWImgDescriptorExtractor**(const   Ptr<DescriptorExtractor>&
                                                            dextractor,   const
                                                            Ptr<DescriptorMatcher>&
                                                            dmatcher)

>   **Parameters**

>>   **dextractor** – Descriptor extractor that is used to compute descriptors for an input image and
>>   its keypoints.

>>   **dmatcher** – Descriptor matcher that is used to find the nearest word of the trained vocabu-
>>   lary for each keypoint descriptor of the image.

BOWImgDescriptorExtractor::setVocabulary

Sets a visual vocabulary.

C++: void BOWImgDescriptorExtractor::**setVocabulary**(const Mat& **vocabulary**)

>   **Parameters**

>>   **vocabulary** – Vocabulary (can be trained using the inheritor of `BOWTrainer`). Each row of
>>   the vocabulary is a visual word (cluster center).

BOWImgDescriptorExtractor::getVocabulary

Returns the set vocabulary.

C++: const Mat& BOWImgDescriptorExtractor::**getVocabulary**() const

BOWImgDescriptorExtractor::compute

Computes an image descriptor using the set visual vocabulary.

C++: void BOWImgDescriptorExtractor::**compute**(const Mat& **image**, vector<KeyPoint>& **keypoints**, Mat& **imgDescriptor**, vector<vector<int>>* **pointIdxsOfClusters**=0, Mat* **descriptors**=0)

> **Parameters**
>
>> **image** – Image, for which the descriptor is computed.
>>
>> **keypoints** – Keypoints detected in the input image.
>>
>> **imgDescriptor** – Computed output image descriptor.
>>
>> **pointIdxsOfClusters** – Indices of keypoints that belong to the cluster. This means that `pointIdxsOfClusters[i]` are keypoint indices that belong to the `i` -th cluster (word of vocabulary) returned if it is non-zero.
>>
>> **descriptors** – Descriptors of the image keypoints that are returned if they are non-zero.

BOWImgDescriptorExtractor::descriptorSize

Returns an image discriptor size if the vocabulary is set. Otherwise, it returns 0.

C++: int BOWImgDescriptorExtractor::**descriptorSize**() const

BOWImgDescriptorExtractor::descriptorType

Returns an image descriptor type.

C++: int BOWImgDescriptorExtractor::**descriptorType**() const

OBJDETECT. OBJECT DETECTION

8.1 Cascade Classification

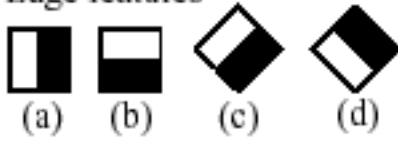### Haar Feature-based Cascade Classifier for Object Detection

The object detector described below has been initially proposed by Paul Viola [Viola01] and improved by Rainer Lienhart [Lienhart02].

First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundred sample views of a particular object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.
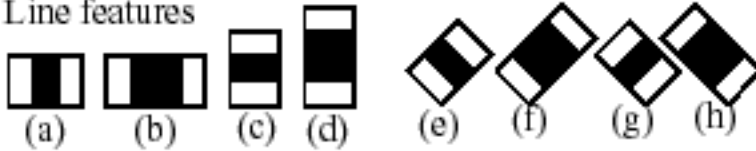
After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a "1" if the region is likely to show the object (i.e., face/car), and "0" otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word "cascade" in the classifier name means that the resultant classifier consists of several simpler classifiers (*stages*) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word "boosted" means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different `boosting` techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifers, and are calculated as described below. The current algorithm uses the following Haar-like features:
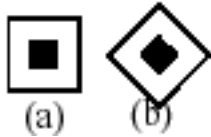
The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in the case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and the `integral()` description).

To see the object detector at work, have a look at the facedetect demo: https://code.ros.org/svn/opencv/trunk/opencv/samples/cpp/facedetect.cpp

The following reference is for the detection part only. There is a separate application called `opencv_traincascade` that can train a cascade of boosted classifiers from a set of samples.

Note: In the new C++ interface it is also possible to use LBP (local binary pattern) features in addition to Haar-like features.

FeatureEvaluator

class FeatureEvaluator

Base class for computing feature values in cascade classifiers.

```cpp
class CV_EXPORTS FeatureEvaluator
{
public:
    enum { HAAR = 0, LBP = 1 }; // supported feature types
    virtual ~FeatureEvaluator(); // destructor
    virtual bool read(const FileNode& node);
    virtual Ptr<FeatureEvaluator> clone() const;
    virtual int getFeatureType() const;

    virtual bool setImage(const Mat& img, Size origWinSize);
    virtual bool setWindow(Point p);

    virtual double calcOrd(int featureIdx) const;
```

```
    virtual int calcCat(int featureIdx) const;

    static Ptr<FeatureEvaluator> create(int type);
};
```

FeatureEvaluator::read

Reads parameters of features from the `FileStorage` node.

C++: bool FeatureEvaluator::**read**(const FileNode& **node**)

>   **Parameters**

>>   **node** – File node from which the feature parameters are read.

FeatureEvaluator::clone

Returns a full copy of the feature evaluator.

C++: Ptr<FeatureEvaluator> FeatureEvaluator::**clone**() const

FeatureEvaluator::getFeatureType

Returns the feature type (HAAR or LBP for now).

C++: int FeatureEvaluator::**getFeatureType**() const

FeatureEvaluator::setImage

Assigns an image to feature evaluator.

C++: bool FeatureEvaluator::**setImage**(const Mat& **img**, Size **origWinSize**)

>   **Parameters**

>>   **img** – Matrix of the type `CV_8UC1` containing an image where the features are computed.

>>   **origWinSize** – Size of training images.

The method assigns an image, where the features will be computed, to the feature evaluator.

FeatureEvaluator::setWindow

Assigns a window in the current image where the features will be computed.

C++: bool FeatureEvaluator::**setWindow**(Point **p**)

>   **Parameters**

>>   **p** – Upper left point of the window where the features are computed. Size of the window is equal to the size of training images.

FeatureEvaluator::calcOrd

Computes the value of an ordered (numerical) feature.

C++: double FeatureEvaluator::**calcOrd**(int **featureIdx**) const

> **Parameters**
>
> > **featureIdx** – Index of the feature whose value is computed.

The function returns the computed value of an ordered feature.

FeatureEvaluator::calcCat

Computes the value of a categorical feature.

C++: int FeatureEvaluator::**calcCat**(int **featureIdx**) const

> **Parameters**
>
> > **featureIdx** – Index of the feature whose value is computed.

The function returns the computed label of a categorical feature, which is the value from [0,... (number of categories - 1)].

FeatureEvaluator::create

Constructs the feature evaluator.

C++: **static** Ptr<FeatureEvaluator> FeatureEvaluator::**create**(int **type**)

> **Parameters**
>
> > **type** – Type of features evaluated by cascade (HAAR or LBP for now).

CascadeClassifier

class **CascadeClassifier**

Cascade classifier class for object detection.

CascadeClassifier::CascadeClassifier

Loads a classifier from a file.

C++: CascadeClassifier::**CascadeClassifier**(const string& **filename**)

Python: cv2.**CascadeClassifier**(filename) → <CascadeClassifier object>

> **Parameters**
>
> > **filename** – Name of the file from which the classifier is loaded.

CascadeClassifier::empty

Checks whether the classifier has been loaded.

C++: bool CascadeClassifier::**empty**() const

Python: cv2.CascadeClassifier.**empty**() → retval

CascadeClassifier::load

Loads a classifier from a file.

C++: bool CascadeClassifier::**load**(const string& **filename**)

Python: cv2.CascadeClassifier.**load**(filename) → retval

> **Parameters**
>
> > **filename** – Name of the file from which the classifier is loaded. The file may contain an old HAAR classifier trained by the haartraining application or a new cascade classifier trained by the traincascade application.

CascadeClassifier::read

Reads a classifier from a FileStorage node.

C++: bool CascadeClassifier::**read**(const FileNode& **node**)

Note: The file may contain a new cascade classifier (trained traincascade application) only.

CascadeClassifier::detectMultiScale

Detects objects of different sizes in the input image. The detected objects are returned as a list of rectangles.

C++: void CascadeClassifier::**detectMultiScale**(const Mat& **image**, vector<Rect>& **objects**, double **scaleFactor**=1.1, int **minNeighbors**=3, int **flags**=0, Size **minSize**=Size(), Size **maxSize**=Size())

Python: cv2.CascadeClassifier.**detectMultiScale**(image[, scaleFactor[, minNeighbors[, flags[, minSize[, maxSize]]]]]) → objects

Python: cv2.CascadeClassifier.**detectMultiScale**(image, rejectLevels, levelWeights[, scaleFactor[, minNeighbors[, flags[, minSize[, maxSize[, outputRejectLevels]]]]]]) → objects

C: CvSeq* **cvHaarDetectObjects**(const CvArr* **image**, CvHaarClassifierCascade* **cascade**, CvMemStorage* **storage**, double **scaleFactor**=1.1, int **minNeighbors**=3, int **flags**=0, CvSize **minSize**=cvSize(0, 0), CvSize **maxSize**=cvSize(0, 0))

Python: cv.**HaarDetectObjects**(image, cascade, storage, scaleFactor=1.1, minNeighbors=3, flags=0, minSize=(0, 0)) → detectedObjects

> **Parameters**
>
> > **cascade** – Haar classifier cascade (OpenCV 1.x API only). It can be loaded from XML or YAML file using Load(). When the cascade is not needed anymore, release it using cvReleaseHaarClassifierCascade(&cascade).

image – Matrix of the type `CV_8U` containing an image where objects are detected.

objects – Vector of rectangles where each rectangle contains the detected object.

scaleFactor – Parameter specifying how much the image size is reduced at each image scale.

minNeighbors – Parameter specifying how many neighbors each candiate rectangle should have to retain it.

flags – Parameter with the same meaning for an old cascade as in the function `cvHaarDetectObjects`. It is not used for a new cascade.

minSize – Minimum possible object size. Objects smaller than that are ignored.

maxSize – Maximum possible object size. Objects larger than that are ignored.

CascadeClassifier::setImage

Sets an image for detection.

C++: bool CascadeClassifier::**setImage**(Ptr<FeatureEvaluator>& **feval**, const Mat& **image**)

C: void **cvSetImagesForHaarClassifierCascade**(CvHaarClassifierCascade* **cascade**, const CvArr* **sum**, const CvArr* **sqsum**, const CvArr* **tiltedSum**, double **scale**)

    Parameters

        cascade – Haar classifier cascade (OpenCV 1.x API only). See `CascadeClassifier::detectMultiScale()` for more information.

        feval – Pointer to the feature evaluator used for computing features.

        image – Matrix of the type `CV_8UC1` containing an image where the features are computed.

The function is automatically called by `CascadeClassifier::detectMultiScale()` at every image scale. But if you want to test various locations manually using `CascadeClassifier::runAt()`, you need to call the function before, so that the integral images are computed.

Note: in the old API you need to supply integral images (that can be obtained using `Integral()`) instead of the original image.

CascadeClassifier::runAt

Runs the detector at the specified point.

C++: int CascadeClassifier::**runAt**(Ptr<FeatureEvaluator>& **feval**, Point **pt**)

C: int **cvRunHaarClassifierCascade**(CvHaarClassifierCascade* **cascade**, CvPoint **pt**, int **startStage**=0)

    Parameters

        cascade – Haar classifier cascade (OpenCV 1.x API only). See `CascadeClassifier::detectMultiScale()` for more information.

        feval – Feature evaluator used for computing features.

        pt – Upper left point of the window where the features are computed. Size of the window is equal to the size of training images.

The function returns 1 if the cascade classifier detects an object in the given location. Otherwise, it returns negated index of the stage at which the candidate has been rejected.

Use `CascadeClassifier::setImage()` to set the image for the detector to work with.

groupRectangles

Groups the object candidate rectangles.

C++: void **groupRectangles**(vector<Rect>& **rectList**, int **groupThreshold**, double **eps**=0.2)

Python: cv2.**groupRectangles**(rectList, groupThreshold[, eps]) → None

Python: cv2.**groupRectangles**(rectList, groupThreshold[, eps]) → weights

Python: cv2.**groupRectangles**(rectList, groupThreshold, eps, weights, levelWeights) → None

> **Parameters**

>> **rectList** – Input/output vector of rectangles. Output vector includes retained and grouped rectangles.

>> **groupThreshold** – Minimum possible number of rectangles minus 1. The threshold is used in a group of rectangles to retain it.

>> **eps** – Relative difference between sides of the rectangles to merge them into a group.

The function is a wrapper for the generic function `partition()` . It clusters all the input rectangles using the rectangle equivalence criteria that combines rectangles with similar sizes and similar locations. The similarity is defined by `eps`. When `eps=0` , no clustering is done at all. If $eps \rightarrow +\inf$, all the rectangles are put in one cluster. Then, the small clusters containing less than or equal to `groupThreshold` rectangles are rejected. In each other cluster, the average rectangle is computed and put into the output rectangle list.

8.2 Latent SVM

Discriminatively Trained Part Based Models for Object Detection

The object detector described below has been initially proposed by P.F. Felzenszwalb in [Felzenszwalb2010]. It is based on a Dalal-Triggs detector that uses a single filter on histogram of oriented gradients (HOG) features to represent an object category. This detector uses a sliding window approach, where a filter is applied at all positions and scales of an image. The first innovation is enriching the Dalal-Triggs model using a star-structured part-based model defined by a "root" filter (analogous to the Dalal-Triggs filter) plus a set of parts filters and associated deformation models. The score of one of star models at a particular position and scale within an image is the score of the root filter at the given location plus the sum over parts of the maximum, over placements of that part, of the part filter score on its location minus a deformation cost easuring the deviation of the part from its ideal location relative to the root. Both root and part filter scores are defined by the dot product between a filter (a set of weights) and a subwindow of a feature pyramid computed from the input image. Another improvement is a representation of the class of models by a mixture of star models. The score of a mixture model at a particular position and scale is the maximum over components, of the score of that component model at the given location.

In OpenCV there are C implementation of Latent SVM and C++ wrapper of it. C version is the structure `CvObjectDetection` and a set of functions working with this structure (see `cvLoadLatentSvmDetector()`, `cvReleaseLatentSvmDetector()`, `cvLatentSvmDetectObjects()`). C++ version is the class `LatentSvmDetector` and has slightly different functionality in contrast with C version - it supports loading and detection of several models.

There are two examples of Latent SVM usage: `samples/c/latentsvmdetect.cpp` and `samples/cpp/latentsvm_multidetect.cpp`.

CvLSVMFilterPosition

struct `CvLSVMFilterPosition`

Structure describes the position of the filter in the feature pyramid.

> unsigned int **l**
>> level in the feature pyramid
>
> unsigned int **x**
>> x-coordinate in level l
>
> unsigned int **y**
>> y-coordinate in level l

CvLSVMFilterObject

struct `CvLSVMFilterObject`

Description of the filter, which corresponds to the part of the object.

> CvLSVMFilterPosition **V**
>> ideal (penalty = 0) position of the partial filter from the root filter position (V_i in the paper)
>
> float **`fineFunction[4]`**
>> vector describes penalty function (d_i in the paper) pf[0] * x + pf[1] * y + pf[2] * x^2 + pf[3] * y^2
>
> **`int sizeX, sizeY`**
>> Rectangular map (sizeX x sizeY), every cell stores feature vector (dimension = p)
>
> int **`numFeatures`**
>> number of features
>
> float* **H**
>> matrix of feature vectors to set and get feature vectors (i,j) used formula H[(j * sizeX + i) * p + k], where k - component of feature vector in cell (i, j)

CvLatentSvmDetector

struct `CvLatentSvmDetector`

Structure contains internal representation of trained Latent SVM detector.

> int **`num_filters`**
>> total number of filters (root plus part) in model
>
> int **`num_components`**
>> number of components in model
>
> int* **`num_part_filters`**
>> array containing number of part filters for each component
>
> CvLSVMFilterObject** **`filters`**
>> root and part filters for all model components

float* **b**
    biases for all model components

float **score_threshold**
    confidence level threshold

CvObjectDetection

struct CvObjectDetection

Structure contains the bounding box and confidence level for detected object.

CvRect **rect**
    bounding box for a detected object

float **score**
    confidence level

cvLoadLatentSvmDetector

Loads trained detector from a file.

C++: CvLatentSvmDetector* **cvLoadLatentSvmDetector**(const char* **filename**)

    Parameters

        filename – Name of the file containing the description of a trained detector

cvReleaseLatentSvmDetector

Release memory allocated for CvLatentSvmDetector structure.

C++: void **cvReleaseLatentSvmDetector**(CvLatentSvmDetector** **detector**)

    Parameters

        detector – CvLatentSvmDetector structure to be released

cvLatentSvmDetectObjects

Find rectangular regions in the given image that are likely to contain objects and corresponding confidence levels.

C++: CvSeq* **cvLatentSvmDetectObjects**(IplImage* **image**, CvLatentSvmDetector* **detector**, CvMem-Storage* **storage**, float **overlap_threshold**, int **numThreads**)

    Parameters

        image – image

        detector – LatentSVM detector in internal representation

        storage – Memory storage to store the resultant sequence of the object candidate rectangles

        overlap_threshold – Threshold for the non-maximum suppression algorithm

        numThreads – Number of threads used in parallel version of the algorithm

LatentSvmDetector

class LatentSvmDetector

This is a C++ wrapping class of Latent SVM. It contains internal representation of several trained Latent SVM detectors (models) and a set of methods to load the detectors and detect objects using them.

LatentSvmDetector::ObjectDetection

class LatentSvmDetector::**ObjectDetection**

Structure contains the detection information.

> Rect **rect**
> > bounding box for a detected object
>
> float **score**
> > confidence level
>
> int **classID**
> > class (model or detector) ID that detect an object

LatentSvmDetector::LatentSvmDetector

Two types of constructors.

C++: LatentSvmDetector::**LatentSvmDetector**()

C++: LatentSvmDetector::**LatentSvmDetector**(const vector<string>& **filenames**, const vector<string>& **classNames**=vector<string>())

> **Parameters**
>
> > **filenames** – A set of filenames storing the trained detectors (models). Each file contains one model. See examples of such files here /opencv_extra/testdata/cv/latentsvmdetector/models_VOC2007/.
> >
> > **classNames** – A set of trained models names. If it's empty then the name of each model will be constructed from the name of file containing the model. E.g. the model stored in "/home/user/cat.xml" will get the name "cat".

LatentSvmDetector::~LatentSvmDetector

Destructor.

C++: LatentSvmDetector::**~LatentSvmDetector**()

LatentSvmDetector::~clear

Clear all trained models and their names stored in an class object.

C++: void LatentSvmDetector::**clear**()

LatentSvmDetector::load

Load the trained models from given `.xml` files and return `true` if at least one model was loaded.

C++: bool LatentSvmDetector::**load**(const vector<string>& **filenames**, const vector<string>& **classNames**)

> **Parameters**
>
> > **filenames** – A set of filenames storing the trained detectors (models). Each file contains one model. See examples of such files here /opencv_extra/testdata/cv/latentsvmdetector/models_VOC2007/.
> >
> > **classNames** – A set of trained models names. If it's empty then the name of each model will be constructed from the name of file containing the model. E.g. the model stored in "/home/user/cat.xml" will get the name "cat".

LatentSvmDetector::detect

Find rectangular regions in the given image that are likely to contain objects of loaded classes (models) and corresponding confidence levels.

C++: void LatentSvmDetector::**detect**(const Mat& **image**, vector<ObjectDetection>& **objectDetections**, float **overlapThreshold**=0.5, int **numThreads**=-1)

> **Parameters**
>
> > **image** – An image.
> >
> > **objectDetections** – The detections: rectangulars, scores and class IDs.
> >
> > **overlapThreshold** – Threshold for the non-maximum suppression algorithm.
> >
> > **numThreads** – Number of threads used in parallel version of the algorithm.

LatentSvmDetector::getClassNames

Return the class (model) names that were passed in constructor or method `load` or extructed from models filenames in those methods.

C++: const vector<string>& LatentSvmDetector::**getClassNames**() const

LatentSvmDetector::getClassCount

Return a count of loaded models (classes).

C++: size_t **getClassCount**() const

ML. MACHINE LEARNING

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression, and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different sets of features (like an ability to handle missing measurements or categorical input variables), 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.

9.1 Statistical Models

CvStatModel

class `CvStatModel`

Base class for statistical models in ML.

```
class CvStatModel
{
public:
    /* CvStatModel(); */
    /* CvStatModel( const Mat& train_data ... ); */

    virtual ~CvStatModel();

    virtual void clear()=0;

    /* virtual bool train( const Mat& train_data, [int tflag,] ..., const
        Mat& responses, ...,
     [const Mat& var_idx,] ..., [const Mat& sample_idx,] ...
     [const Mat& var_type,] ..., [const Mat& missing_mask,]
        <misc_training_alg_params> ... )=0;
      */

    /* virtual float predict( const Mat& sample ... ) const=0; */

    virtual void save( const char* filename, const char* name=0 )=0;
    virtual void load( const char* filename, const char* name=0 )=0;

    virtual void write( CvFileStorage* storage, const char* name )=0;
    virtual void read( CvFileStorage* storage, CvFileNode* node )=0;
};
```

In this declaration, some methods are commented off. 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 part of the base class.

CvStatModel::CvStatModel

The default constuctor.

C++: `CvStatModel::`**`CvStatModel`**`()`

Each statistical model class in ML has a default constructor without parameters. This constructor is useful for a two-stage model construction, when the default constructor is followed by `CvStatModel::train()` or `CvStatModel::load()`.

CvStatModel::CvStatModel(...)

The training constructor.

`CvStatModel::CvStatModel( const Mat& train_data ... )`

Most ML classes provide a single-step constructor and train constructors. This constructor is equivalent to the default constructor, followed by the `CvStatModel::train()` method with the parameters that are passed to the constructor.

CvStatModel::~CvStatModel

The virtual destructor.

C++: `CvStatModel::`**`~CvStatModel`**`()`

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 `CvStatModel::clear()` that deallocates all the memory.

CvStatModel::clear

Deallocates memory and resets the model state.

C++: `void CvStatModel::`**`clear`**`()`

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 `CvStatModel::train()` methods of the derived classes, from the methods `CvStatModel::load()`, `CvStatModel::read()`, or even explicitly by the user.

CvStatModel::save

Saves the model to a file.

C++: void CvStatModel::**save**(const char* **filename**, const char* **name**=0)

Python: cv2.StatModel.**save**(filename[, name]) → None

The method save saves the complete model state to the specified XML or YAML file with the specified name or default name (which depends on a particular class). *Data persistence* functionality from CxCore is used.

CvStatModel::load

Loads the model from a file.

C++: void CvStatModel::**load**(const char* **filename**, const char* **name**=0)

Python: cv2.StatModel.**load**(filename[, name]) → None

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 CvStatModel::clear().

CvStatModel::write

Writes the model to the file storage.

C++: void CvStatModel::**write**(CvFileStorage* **storage**, const char* **name**)

The method write stores the complete model state in the file storage with the specified name or default name (which depends on the particular class). The method is called by CvStatModel::save().

CvStatModel::read

Reads the model from the file storage.

C++: void CvStatModel::**read**(CvFileStorage* **storage**, CvFileNode* **node**)

The method read restores the complete model state from the specified node of the file storage. Use the function GetFileNodeByName() to locate the node.

The previous model state is cleared by CvStatModel::clear().

CvStatModel::train

Trains the model.

bool CvStatModel::train(const Mat& train_data, [int tflag,] ..., const Mat& responses, ...,   [const Ma

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, that is, 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 the tflag parameter that specifies the orientation as follows:

- tflag=CV_ROW_SAMPLE The feature vectors are stored as rows.
- tflag=CV_COL_SAMPLE The feature vectors are stored as columns.

The `train_data` must have the `CV_32FC1` (32-bit floating-point, single-channel) format. Responses are usually stored in a 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 a separate parameter or as the last element of the `var_type` vector:

- `CV_VAR_CATEGORICAL` The output values are discrete class labels.

- `CV_VAR_ORDERED(=CV_VAR_NUMERICAL)` The output values are ordered. This means that two different values can be compared as numbers, and this is a regression problem.

Types of input variables can be also specified using `var_type`. Most algorithms can handle only ordered input variables.

Many ML models may be trained on a selected feature subset, and/or on a selected sample subset of the training set. To make it easier for you, the method `train` usually includes the `var_idx` and `sample_idx` parameters. The former parameter identifies variables (features) of interest, and the latter one identifies samples of interest. Both vectors are either integer (`CV_32SC1`) vectors (lists of 0-based indices) or 8-bit (`CV_8UC1`) masks of active variables/samples. You 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 of 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 `CvStatModel::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

Predicts the response for a sample.

```
float CvStatModel::predict( const Mat& sample[, <prediction_params>] ) const
```

The method is used to predict the response for a new sample. In case of a classification, the method returns the class label. In case of a regression, the method returns 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.

9.2 Normal Bayes Classifier

This simple classification model assumes 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.

CvNormalBayesClassifier

class `CvNormalBayesClassifier`

Bayes classifier for normally distributed data.

CvNormalBayesClassifier::CvNormalBayesClassifier

Default and training constructors.

C++: CvNormalBayesClassifier::**CvNormalBayesClassifier**()

C++: CvNormalBayesClassifier::**CvNormalBayesClassifier**(const Mat& **trainData**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat())

C++: CvNormalBayesClassifier::**CvNormalBayesClassifier**(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0)

Python: cv2.**NormalBayesClassifier**(trainData, responses[, varIdx[, sampleIdx]]) → <Normal-BayesClassifier object>

The constructors follow conventions of `CvStatModel::CvStatModel()`. See `CvStatModel::train()` for parameters descriptions.

CvNormalBayesClassifier::train

Trains the model.

C++: bool CvNormalBayesClassifier::**train**(const Mat& **trainData**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), bool **update**=false)

C++: bool CvNormalBayesClassifier::**train**(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, bool **update**=false)

Python: cv2.NormalBayesClassifier.**train**(trainData, responses[, varIdx[, sampleIdx[, update]]]) → retval

> **Parameters**

> > **update** – Identifies whether the model should be trained from scratch (`update=false`) or should be updated using the new training data (`update=true`).

The method trains the Normal Bayes classifier. It follows the conventions of the generic `CvStatModel::train()` approach with the following limitations:

- Only `CV_ROW_SAMPLE` data layout is supported.

- Input variables are all ordered.

- Output variable is categorical , which means that elements of `responses` must be integer numbers, though the vector may have the `CV_32FC1` type.

- Missing measurements are not supported.

CvNormalBayesClassifier::predict

Predicts the response for sample(s).

C++: float CvNormalBayesClassifier::**predict**(const Mat& **samples**, Mat* **results**=0) const

C++: float CvNormalBayesClassifier::**predict**(const CvMat* **samples**, CvMat* **results**=0) const

Python: cv2.NormalBayesClassifier.**predict**(samples) → retval, results

The method estimates the most probable classes for input vectors. Input vectors (one or more) are stored as rows of the matrix samples. In 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.

9.3 K-Nearest Neighbors

The algorithm caches all 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, and so on. 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

class CvKNearest

The class implements K-Nearest Neighbors model as described in the beginning of this section.

CvKNearest::CvKNearest

Default and training constructors.

C++: CvKNearest::**CvKNearest**()

C++: CvKNearest::**CvKNearest**(const Mat& **trainData**, const Mat& **responses**, const Mat& **sampleIdx**=Mat(), bool **isRegression**=false, int **max_k**=32)

C++: CvKNearest::**CvKNearest**(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **sampleIdx**=0, bool **isRegression**=false, int **max_k**=32)

See CvKNearest::train() for additional parameters descriptions.

CvKNearest::train

Trains the model.

C++: bool CvKNearest::**train**(const Mat& **trainData**, const Mat& **responses**, const Mat& **sampleIdx**=Mat(), bool **isRegression**=false, int **maxK**=32, bool **updateBase**=false)

C++: bool CvKNearest::**train**(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **sampleIdx**=0, bool **is_regression**=false, int **maxK**=32, bool **updateBase**=false)

Python: cv2.KNearest.**train**(trainData, responses[, sampleIdx[, isRegression[, maxK[, updateBase]]]]) → retval

> **Parameters**

>> **isRegression** – Type of the problem: true for regression and false for classification.

>> **maxK** – Number of maximum neighbors that may be passed to the method CvKNearest::find_nearest().

> **updateBase** – 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 `maxK` must not be larger than the original value.

The method trains the K-Nearest model. It follows the conventions of the generic `CvStatModel::train()` approach with the following limitations:

- Only `CV_ROW_SAMPLE` data layout is supported.

- Input variables are all ordered.

- Output variables can be either categorical (`is_regression=false`) or ordered (`is_regression=true`).

- Variable subsets (`var_idx`) and missing measurements are not supported.

CvKNearest::find_nearest

Finds the neighbors and predicts responses for input vectors.

C++: float CvKNearest::**find_nearest**(const Mat& **samples**, int **k**, Mat* **results**=0, const float** **neighbors**=0, Mat* **neighborResponses**=0, Mat* **dist**=0) const

C++: float CvKNearest::**find_nearest**(const Mat& **samples**, int **k**, Mat& **results**, Mat& **neighborResponses**, Mat& **dists**) const

C++: float CvKNearest::**find_nearest**(const CvMat* **samples**, int **k**, CvMat* **results**=0, const float** **neighbors**=0, CvMat* **neighborResponses**=0, CvMat* **dist**=0) const

Python: cv2.KNearest.**find_nearest**(samples, k[, results[, neighborResponses[, dists]]]) → retval, results, neighborResponses, dists

> **Parameters**

>> **samples** – Input samples stored by rows. It is a single-precision floating-point matrix of $number\_of\_samples \times number\_of\_features$ size.

>> **k** – Number of used nearest neighbors. It must satisfy constraint: $k \leq$ CvKNearest::get_max_k().

>> **results** – Vector with results of prediction (regression or classification) for each input sample. It is a single-precision floating-point vector with `number_of_samples` elements.

>> **neighbors** – Optional output pointers to the neighbor vectors themselves. It is an array of `k*samples->rows` pointers.

>> **neighborResponses** – Optional output values for corresponding `neighbors`. It is a single-precision floating-point matrix of $number\_of\_samples \times k$ size.

>> **dist** – Optional output distances from the input vectors to the corresponding `neighbors`. It is a single-precision floating-point matrix of $number\_of\_samples \times k$ size.

For each input vector (a row of the matrix `samples`), the method finds the k nearest neighbors. In case of regression, the predicted result is a mean value of the particular vector's neighbor responses. In case of classification, the class is determined by voting.

For each input vector, the neighbors are sorted by their distances to the vector.

In case of C++ interface you can use output pointers to empty matrices and the function will allocate memory itself.

If only a single input vector is passed, all output matrices are optional and the predicted value is returned by the method.

CvKNearest::get_max_k

Returns the number of maximum neighbors that may be passed to the method `CvKNearest::find_nearest()`.

C++: int CvKNearest::**get_max_k**() const

CvKNearest::get_var_count

Returns the number of used features (variables count).

C++: int CvKNearest::**get_var_count**() const

CvKNearest::get_sample_count

Returns the total number of train samples.

C++: int CvKNearest::**get_sample_count**() const

CvKNearest::is_regression

Returns type of the problem: `true` for regression and `false` for classification.

C++: bool CvKNearest::**is_regression**() const

The sample below (currently using the obsolete `CvMat` structures) demonstrates the use of the k-nearest classifier for 2D point classification:

```
#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(200,200), cvScalar(50,50) );

    cvGetRows( trainData, &trainData2, train_sample_count/2, train_sample_count );
    cvRandArr( &rng_state, &trainData2, CV_RAND_NORMAL, cvScalar(300,300), cvScalar(50,50) );

    cvGetRows( trainClasses, &trainClasses1, 0, train_sample_count/2 );
    cvSet( &trainClasses1, cvScalar(1) );
```

```
cvGetRows( trainClasses, &trainClasses2, train_sample_count/2, train_sample_count );
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;

        // estimate the response and get the neighbors' labels
        response = knn.find_nearest(&sample,K,0,0,nearests,0);

        // compute the number of neighbors representing the majority
        for( k = 0, accuracy = 0; k < K; k++ )
        {
            if( nearests->data.fl[k] == response)
                accuracy++;
        }
        // highlight the pixel depending on the accuracy (or confidence)
        cvSet2D( img, i, j, response == 1 ?
            (accuracy > 5 ? CV_RGB(180,0,0) : CV_RGB(180,120,0)) :
            (accuracy > 5 ? CV_RGB(0,180,0) : CV_RGB(120,120,0)) );
    }
}

// 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;
}
```

9.4 Support Vector Machines

Originally, support vector machines (SVM) was a technique for building an optimal binary (2-class) classifier. Later the technique was extended to regression and clustering problems. SVM is a partial case of kernel-based methods. It maps feature vectors into a higher-dimensional space using a kernel function and builds an optimal linear discriminat-

ing function in this space or an optimal hyper-plane that fits into the training data. In 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, which means that the margin between the separating hyper-plane and the nearest feature vectors from both classes (in case of 2-class classifier) is maximal. The feature vectors that are the closest to the hyper-plane are called *support vectors*, which means that the position of other vectors does not affect the hyper-plane (the decision function).

SVM implementation in OpenCV is based on [LibSVM].

CvParamGrid

class `CvParamGrid`

The structure represents the logarithmic grid range of statmodel parameters. It is used for optimizing statmodel accuracy by varying model parameters, the accuracy estimate being computed by cross-validation.

double CvParamGrid::**min_val**
    Minimum value of the statmodel parameter.

double CvParamGrid::**max_val**
    Maximum value of the statmodel parameter.

double CvParamGrid::**step**
    Logarithmic step for iterating the statmodel parameter.

The grid determines the following iteration sequence of the statmodel parameter values:

$$(\text{min\_val}, \text{min\_val} * \text{step}, \text{min\_val} * \text{step}^2, \ldots, \text{min\_val} * \text{step}^n),$$

where n is the maximal index satisfying

$$\text{min\_val} * \text{step}^n < \text{max\_val}$$

The grid is logarithmic, so `step` must always be greater then 1.

CvParamGrid::CvParamGrid

The constructors.

C++: CvParamGrid::**CvParamGrid**()

C++: CvParamGrid::**CvParamGrid**(double **min_val**, double **max_val**, double **log_step**)

The full constructor initializes corresponding members. The default constructor creates a dummy grid:

```
CvParamGrid::CvParamGrid()
{
    min_val = max_val = step = 0;
}
```

CvParamGrid::check

Checks validness of the grid.

C++: bool CvParamGrid::**check**()

Returns `true` if the grid is valid and `false` otherwise. The grid is valid if and only if:

- Lower bound of the grid is less then the upper one.

- Lower bound of the grid is positive.

- Grid step is greater then 1.

CvSVMParams

class **CvSVMParams**

SVM training parameters.

The structure must be initialized and passed to the training method of `CvSVM`.

CvSVMParams::CvSVMParams

The constructors.

C++: `CvSVMParams::`**`CvSVMParams()`**

C++: `CvSVMParams::`**`CvSVMParams`**`(`int **svm_type**, int **kernel_type**, double **degree**, double **gamma**, double **coef0**, double **Cvalue**, double **nu**, double **p**, CvMat* **class_weights**, CvTermCriteria **term_crit**`)`

   Parameters

   svm_type – Type of a SVM formulation. Possible values are:

   – **CvSVM::C_SVC** C-Support Vector Classification. n-class classification ($n \geq 2$), allows imperfect separation of classes with penalty multiplier `C` for outliers.

   – **CvSVM::NU_SVC** ν-Support Vector Classification. n-class classification with possible imperfect separation. Parameter ν (in the range 0..1, the larger the value, the smoother the decision boundary) is used instead of `C`.

   – **CvSVM::ONE_CLASS** Distribution Estimation (One-class SVM). All the training data are from the same class, SVM builds a boundary that separates the class from the rest of the feature space.

   – **CvSVM::EPS_SVR** ϵ-Support Vector Regression. The distance between feature vectors from the training set and the fitting hyper-plane must be less than `p`. For outliers the penalty multiplier `C` is used.

   – **CvSVM::NU_SVR** ν-Support Vector Regression. ν is used instead of `p`.

   See [LibSVM] for details.

   kernel_type – Type of a SVM kernel. Possible values are:

   – **CvSVM::LINEAR** Linear kernel. No mapping is done, linear discrimination (or regression) is done in the original feature space. It is the fastest option. $K(x_i, x_j) = x_i^\mathsf{T} x_j$.

   – **CvSVM::POLY** Polynomial kernel: $K(x_i, x_j) = (\gamma x_i^\mathsf{T} x_j + coef0)^{degree}, \gamma > 0$.

   – **CvSVM::RBF** Radial basis function (RBF), a good choice in most cases. $K(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}, \gamma > 0$.

   – **CvSVM::SIGMOID** Sigmoid kernel: $K(x_i, x_j) = \tanh(\gamma x_i^\mathsf{T} x_j + coef0)$.

   degree – Parameter `degree` of a kernel function (POLY).

   gamma – Parameter γ of a kernel function (POLY / RBF / SIGMOID).

> **coef0** – Parameter `coef0` of a kernel function (POLY / SIGMOID).
>
> **Cvalue** – Parameter `C` of a SVM optimiazation problem (C_SVC / EPS_SVR / NU_SVR).
>
> **nu** – Parameter ν of a SVM optimization problem (NU_SVC / ONE_CLASS / NU_SVR).
>
> **p** – Parameter ϵ of a SVM optimization problem (EPS_SVR).
>
> **class_weights** – Optional weights in the C_SVC problem , assigned to particular classes. They are multiplied by `C` so the parameter `C` of class `#i` becomes $class\_weights_i * C$. Thus these weights affect the misclassification penalty for different classes. The larger weight, the larger penalty on misclassification of data from the corresponding class.
>
> **term_crit** – Termination criteria of the iterative SVM training procedure which solves a partial case of constrained quadratic optimization problem. You can specify tolerance and/or the maximum number of iterations.

The default constructor initialize the structure with following values:

```
CvSVMParams::CvSVMParams() :
    svm_type(CvSVM::C_SVC), kernel_type(CvSVM::RBF), degree(0),
    gamma(1), coef0(0), C(1), nu(0), p(0), class_weights(0)
{
    term_crit = cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 1000, FLT_EPSILON );
}
```

CvSVM

class CvSVM

Support Vector Machines.

CvSVM::CvSVM

Default and training constructors.

C++: `CvSVM::CvSVM()`

C++: `CvSVM::CvSVM`(const Mat& **trainData**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), CvSVMParams **params**=CvSVMParams())

C++: `CvSVM::CvSVM`(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, CvSVMParams **params**=CvSVMParams())

Python: `cv2.SVM`(trainData, responses[, varIdx[, sampleIdx[, params]]]) → <SVM object>

The constructors follow conventions of `CvStatModel::CvStatModel()`. See `CvStatModel::train()` for parameters descriptions.

CvSVM::train

Trains an SVM.

C++: bool `CvSVM::train`(const Mat& **trainData**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), CvSVMParams **params**=CvSVMParams())

C++: bool `CvSVM::train`(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, CvSVMParams **params**=CvSVMParams())

Python: `cv2.SVM.train`(trainData, responses[, varIdx[, sampleIdx[, params]]]) → retval

The method trains the SVM model. It follows the conventions of the generic `CvStatModel::train()` approach with the following limitations:

- Only the `CV_ROW_SAMPLE` data layout is supported.

- Input variables are all ordered.

- 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 the `CvSVMParams` structure.

CvSVM::train_auto

Trains an SVM with optimal parameters.

C++: bool CvSVM::**train_auto**(const Mat& **trainData**, const Mat& **responses**, const Mat& **varIdx**, const Mat& **sampleIdx**, CvSVMParams **params**, int **k_fold**=10, CvParamGrid **Cgrid**=CvSVM::get_default_grid(CvSVM::C), CvParamGrid **gammaGrid**=CvSVM::get_default_grid(CvSVM::GAMMA), CvParamGrid **pGrid**=CvSVM::get_default_grid(CvSVM::P), CvParamGrid **nuGrid**=CvSVM::get_default_grid(CvSVM::NU), CvParamGrid **coeffGrid**=CvSVM::get_default_grid(CvSVM::COEF), CvParamGrid **degreeGrid**=CvSVM::get_default_grid(CvSVM::DEGREE), bool **balanced**=false)

C++: bool CvSVM::**train_auto**(const CvMat* **trainData**, const CvMat* **responses**, const CvMat* **varIdx**, const CvMat* **sampleIdx**, CvSVMParams **params**, int **kfold**=10, CvParamGrid **Cgrid**=get_default_grid(CvSVM::C), CvParamGrid **gammaGrid**=get_default_grid(CvSVM::GAMMA), CvParamGrid **pGrid**=get_default_grid(CvSVM::P), CvParamGrid **nuGrid**=get_default_grid(CvSVM::NU), CvParamGrid **coeffGrid**=get_default_grid(CvSVM::COEF), CvParamGrid **degreeGrid**=get_default_grid(CvSVM::DEGREE), bool **balanced**=false)

Python: cv2.SVM.**train_auto**(trainData, responses, varIdx, sampleIdx, params[, k_fold[, Cgrid[, gammaGrid[, pGrid[, nuGrid[, coeffGrid[, degreeGrid[, balanced]]]]]]]]) → retval

Parameters

k_fold – Cross-validation parameter. The training set is divided into `k_fold` subsets. One subset is used to train the model, the others form the test set. So, the SVM algorithm is executed `k_fold` times.

*Grid – Iteration grid for the corresponding SVM parameter.

balanced – If `true` and the problem is 2-class classification then the method creates more balanced cross-validation subsets that is proportions between classes in subsets are close to such proportion in the whole train dataset.

The method trains the SVM model automatically by choosing the optimal parameters C, gamma, p, nu, coef0, degree from `CvSVMParams`. Parameters are considered optimal when the cross-validation estimate of the test set error is minimal.

If there is no need to optimize a parameter, the corresponding grid step should be set to any value less than or equal to 1. For example, to avoid optimization in gamma, set `gamma_grid.step = 0`, `gamma_grid.min_val`, `gamma_grid.max_val` as arbitrary numbers. In this case, the value `params.gamma` is taken for gamma.

And, finally, if the optimization in a parameter is required but the corresponding grid is unknown, you may call the function `CvSVM::get_default_grid()`. To generate a grid, for example, for `gamma`, call `CvSVM::get_default_grid(CvSVM::GAMMA)`.

This function works for the 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 parameters specified in `params` is executed.

CvSVM::predict

Predicts the response for input sample(s).

C++: float CvSVM::**predict**(const Mat& **sample**, bool **returnDFVal**=false) const

C++: float CvSVM::**predict**(const CvMat* **sample**, bool **returnDFVal**=false) const

C++: float CvSVM::**predict**(const CvMat* **samples**, CvMat* **results**) const

Python: cv2.SVM.**predict**(sample[, returnDFVal]) → retval

> **Parameters**
>
>> **sample(s)** – Input sample(s) for prediction.
>>
>> **returnDFVal** – Specifies a type of the return value. If `true` and the problem is 2-class classification then the method returns the decision function value that is signed distance to the margin, else the function returns a class label (classification) or estimated function value (regression).
>>
>> **results** – Output prediction responses for corresponding samples.

If you pass one sample then prediction result is returned. If you want to get responses for several samples then you should pass the `results` matrix where prediction results will be stored.

CvSVM::get_default_grid

Generates a grid for SVM parameters.

C++: CvParamGrid CvSVM::**get_default_grid**(int **param_id**)

> **Parameters**
>
>> **param_id** – SVM parameters IDs that must be one of the following:
>>
>> – **CvSVM::C**
>>
>> – **CvSVM::GAMMA**
>>
>> – **CvSVM::P**
>>
>> – **CvSVM::NU**
>>
>> – **CvSVM::COEF**
>>
>> – **CvSVM::DEGREE**
>>
>> The grid is 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

Returns the current SVM parameters.

C++: CvSVMParams CvSVM::**get_params**() const

This function may be used to get the optimal parameters obtained while automatically training CvSVM::train_auto().

CvSVM::get_support_vector

Retrieves a number of support vectors and the particular vector.

C++: int CvSVM::**get_support_vector_count**() const

C++: const float* CvSVM::**get_support_vector**(int **i**) const

Python: cv2.SVM.**get_support_vector_count**() → nsupportVectors

> **Parameters**
>
> > **i** – Index of the particular support vector.

The methods can be used to retrieve a set of support vectors.

CvSVM::get_var_count

Returns the number of used features (variables count).

C++: int CvSVM::**get_var_count**() const

Python: cv2.SVM.**get_var_count**() → nvars

9.5 Decision Trees

The ML classes discussed in this section implement Classification and Regression Tree algorithms described in [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 (tree where each non-leaf node has two child nodes). It can be used either for classification or for regression. For classification, each tree leaf is marked with a class label; multiple leaves may have the same label. For regression, a constant is also assigned to each tree leaf, 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 (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 following variables are possible:

- **Ordered variables.** The variable value is compared with a threshold that is also stored in the node. If the value is less than the threshold, the procedure goes to the left. Otherwise, it goes to the right. For example, if the weight is less than 1 kilogram, the procedure goes to the left, else to the right.

- **Categorical variables.** A discrete variable value is tested to see whether it belongs to a certain subset of values (also stored in the node) from a limited set of values the variable could take. If it does, the procedure goes to the left. Otherwise, it goes to the right. For example, if the color is green or red, go to the left, else to the right.

So, 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 the 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 to one or more other variables with nearly the same results.

Training Decision Trees

The tree is built recursively, starting from the root node. All training data (feature vectors and responses) is used to split the root node. In each node the optimum decision rule (the best "primary" split) is found based on some criteria. In machine learning, `gini` "purity" criteria are used for classification, and sum of squared errors is used for regression. Then, if necessary, the surrogate splits are found. They resemble the results of the primary split on the training data. All the data is divided using the primary and the surrogate splits (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 (that is, stop splitting the node further) in one of the following cases:

- Depth of the constructed tree branch has reached the specified maximum value.

- Number of training samples in the node is less than the specified threshold when it is not statistically representative to split the node further.

- All the samples in the node belong to the same class or, in case of regression, the variation is too small.

- The best found split 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. Usually tree ensembles build trees that are small enough and use their own protection schemes against overfitting.

Variable Importance

Besides the prediction that is an obvious use of decision trees, the tree can be also used for various data analyses. One of the key properties of the constructed decision tree algorithms is an ability to compute the 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 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.

CvDTreeSplit

class `CvDTreeSplit`

The structure represents a possible decision tree node split. It has public members:

int **var_idx**
    Index of variable on which the split is created.

int **inversed**
    If it is not null then inverse split rule is used that is left and right branches are exchanged in the rule expressions below.

float **quality**
    The split quality, a positive number. It is used to choose the best primary split, then to choose and sort the surrogate splits. After the tree is constructed, it is also used to compute variable importance.

CvDTreeSplit* **next**
    Pointer to the next split in the node list of splits.

int[] **subset**
    Bit array indicating the value subset in case of split on a categorical variable. The rule is:

```
if var_value in subset
  then next_node <- left
  else next_node <- right
```

float ord::**c**
    The threshold value in case of split on an ordered variable. The rule is:

```
if var_value < ord.c
  then next_node<-left
  else next_node<-right
```

int ord::**split_point**
    Used internally by the training algorithm.

CvDTreeNode

class **CvDTreeNode**

The structure represents a node in a decision tree. It has public members:

int **class_idx**
    Class index normalized to 0..class_count-1 range and assigned to the node. It is used internally in classification trees and tree ensembles.

int **Tn**
    Tree index in a ordered sequence of pruned trees. The indices are used during and after the pruning procedure. The root node has the maximum value Tn of the whole tree, child nodes have Tn less than or equal to the parent's Tn, and nodes with $Tn \leq CvDTree :: pruned\_tree\_idx$ are not used at prediction stage (the corresponding branches are considered as cut-off), even if they have not been physically deleted from the tree at the pruning stage.

double **value**
    Value at the node: a class label in case of classification or estimated function value in case of regression.

CvDTreeNode* **parent**
    Pointer to the parent node.

CvDTreeNode* **left**
    Pointer to the left child node.

CvDTreeNode* **right**
    Pointer to the right child node.

CvDTreeSplit* `split`
>   Pointer to the first (primary) split in the node list of splits.

int `sample_count`
>   The number of samples that fall into the node at the training stage. It is used to resolve the difficult cases - when the variable for the primary split is missing and all the variables for other surrogate splits are missing too. In this case the sample is directed to the left if `left->sample_count > right->sample_count` and to the right otherwise.

int `depth`
>   Depth of the node. The root node depth is 0, the child nodes depth is the parent's depth + 1.

Other numerous fields of `CvDTreeNode` are used internally at the training stage.

CvDTreeParams

class `CvDTreeParams`

The structure contains all the decision tree training parameters. You can initialize it by default constructor and then override any parameters directly before training, or the structure may be fully initialized using the advanced variant of the constructor.

CvDTreeParams::CvDTreeParams

The constructors.

C++: `CvDTreeParams::`**`CvDTreeParams`**`()`

C++: `CvDTreeParams::`**`CvDTreeParams`**`(int` **max_depth**, int **min_sample_count**, float **regression_accuracy**, bool **use_surrogates**, int **max_categories**, int **cv_folds**, bool **use_1se_rule**, bool **truncate_pruned_tree**, const float* **priors**`)`

>   **Parameters**

>>   **max_depth** – The maximum possible depth of the tree. That is the training algorithms attempts to split a node while its depth is less than `max_depth`. The actual depth may be smaller if the other termination criteria are met (see the outline of the training procedure in the beginning of the section), and/or if the tree is pruned.

>>   **min_sample_count** – If the number of samples in a node is less than this parameter then the node will not be splitted.

>>   **regression_accuracy** – Termination criteria for regression trees. If all absolute differences between an estimated value in a node and values of train samples in this node are less than this parameter then the node will not be splitted.

>>   **use_surrogates** – If true then surrogate splits will be built. These splits allow to work with missing data and compute variable importance correctly.

>>   **max_categories** – Cluster possible values of a categorical variable into K $\leq$ `max_categories` clusters to find a suboptimal split. If a discrete variable, on which the training procedure tries to make a split, takes more than `max_categories` values, the precise best subset estimation may take a very long time because the algorithm is exponential. Instead, many decision trees engines (including ML) try to find sub-optimal split in this case by clustering all the samples into `max_categories` clusters that is some categories are merged together. The clustering is applied only in n>2-class classification problems for categorical variables with N > `max_categories` possible values. In case of regression and

2-class classification the optimal split can be found efficiently without employing clustering, thus the parameter is not used in these cases.

cv_folds – If `cv_folds > 1` then prune a tree with K-fold cross-validation where K is equal to `cv_folds`.

use_1se_rule – If true then a pruning will be harsher. This will make a tree more compact and more resistant to the training data noise but a bit less accurate.

truncate_pruned_tree – If true then pruned branches are physically removed from the tree. Otherwise they are retained and it is possible to get results from the original unpruned (or pruned less aggressively) tree by decreasing `CvDTree::pruned_tree_idx` parameter.

priors – The array of a priori class probabilities, sorted by the class label value. The parameter can be used to tune the decision tree preferences toward a certain class. For example, if you want to detect some rare anomaly occurrence, the training base will likely contain much more normal cases than anomalies, so a very good classification performance will be achieved just by considering every case as normal. To avoid this, the priors can be specified, where the anomaly probability is artificially increased (up to 0.5 or even greater), so the weight of the misclassified anomalies becomes much bigger, and the tree is adjusted properly. You can also think about this parameter as weights of prediction categories which determine relative weights that you give to misclassification. That is, if the weight of the first category is 1 and the weight of the second category is 10, then each mistake in predicting the second category is equivalent to making 10 mistakes in predicting the first category.

The default constructor initializes all the parameters with the default values tuned for the standalone classification tree:

```
CvDTreeParams() : max_categories(10), max_depth(INT_MAX), min_sample_count(10),
    cv_folds(10), use_surrogates(true), use_1se_rule(true),
    truncate_pruned_tree(true), regression_accuracy(0.01f), priors(0)
{}
```

CvDTreeTrainData

class `CvDTreeTrainData`

Decision tree training data and shared data for tree ensembles. The structure is mostly used internally for storing both standalone trees and tree ensembles efficiently. Basically, it contains the following types of information:

1. Training parameters, an instance of `CvDTreeParams`.

2. Training data preprocessed to find the best splits more efficiently. For tree ensembles, this preprocessed data is reused by all trees. Additionally, the training data characteristics shared by all trees in the ensemble are stored here: variable types, the number of classes, a class label compression map, and so on.

3. Buffers, memory storages for tree nodes, splits, and other elements of the constructed trees.

There are two ways of using this structure. In simple cases (for example, a standalone tree or the ready-to-use "black box" tree ensemble from machine learning, like *Random Trees* or *Boosting*), there is no need to care or even to know about the structure. You just construct the needed statistical model, train it, and use it. The `CvDTreeTrainData` structure is 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:

1. 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`.

2. One or more trees are trained using this data (see the special form of the method `CvDTree::train()`).

3. The structure is released as soon as all the trees using it are released.

CvDTree

class CvDTree

The class implements a decision tree as described in the beginning of this section.

CvDTree::train

Trains a decision tree.

C++: bool CvDTree::**train**(const Mat& **train_data**, int **tflag**, const Mat& **responses**, const Mat& **var_idx**=Mat(), const Mat& **sample_idx**=Mat(), const Mat& **var_type**=Mat(), const Mat& **missing_mask**=Mat(), CvDTreeParams **params**=CvDTreeParams())

C++: bool CvDTree::**train**(const CvMat* **trainData**, int **tflag**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, const CvMat* **varType**=0, const CvMat* **missingDataMask**=0, CvDTreeParams **params**=CvDTreeParams())

C++: bool CvDTree::**train**(CvMLData* **trainData**, CvDTreeParams **params**=CvDTreeParams())

C++: bool CvDTree::**train**(CvDTreeTrainData* **trainData**, const CvMat* **subsampleIdx**)

Python: cv2.DTree.**train**(trainData, tflag, responses[, varIdx[, sampleIdx[, varType[, missingDataMask[, params]]]]]) → retval

There are four `train` methods in `CvDTree`:

- The **first two** methods follow 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, and so on. The last parameter contains all of the necessary training parameters (see the `CvDTreeParams` description).

- The **third** method uses `CvMLData` to pass training data to a decision tree.

- The **last** method `train` is mostly used for building tree ensembles. It takes the pre-constructed `CvDTreeTrainData` instance and an optional subset of the training set. The indices in `subsampleIdx` are counted relatively to the `_sample_idx`, passed to the `CvDTreeTrainData` constructor. For example, if `_sample_idx=[1, 5, 7, 100]`, then `subsampleIdx=[0,3]` means that the samples `[1, 100]` of the original training set are used.

CvDTree::predict

Returns the leaf node of a decision tree corresponding to the input vector.

C++: CvDTreeNode* CvDTree::**predict**(const Mat& **sample**, const Mat& **missingDataMask**=Mat(), bool **preprocessedInput**=false) const

C++: CvDTreeNode* CvDTree::**predict**(const CvMat* **sample**, const CvMat* **missingDataMask**=0, bool **preprocessedInput**=false) const

Python: cv2.DTree.**predict**(sample[, missingDataMask[, preprocessedInput]]) → retval

> **Parameters**
>
> > **sample** – Sample for prediction.
> >
> > **missingDataMask** – Optional input missing measurement mask.

> **preprocessedInput** – This 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 $num\_of\_categories_i - 1$ ranges since 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.

The method traverses the decision tree and returns the reached leaf node as 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`.

CvDTree::calc_error

Returns error of the decision tree.

C++: float CvDTree::**calc_error**(CvMLData* **trainData**, int **type**, std::vector<float>* **resp**=0)

> **Parameters**
>
> > **data** – Data for the decision tree.
> >
> > **type** – Type of error. Possible values are:
> >
> > – **CV_TRAIN_ERROR** Error on train samples.
> >
> > – **CV_TEST_ERROR** Erron on test samples.
> >
> > **resp** – If it is not null then size of this vector will be set to the number of samples and each element will be set to result of prediction on the corresponding sample.

The method calculates error of the decision tree. In case of classification it is the percentage of incorrectly classified samples and in case of regression it is the mean of squared errors on samples.

CvDTree::getVarImportance

Returns the variable importance array.

C++: Mat CvDTree::**getVarImportance**()

C++: const CvMat* CvDTree::**get_var_importance**()

Python: cv2.DTree.**getVarImportance**() → importanceVector

CvDTree::get_root

Returns the root of the decision tree.

C++: const CvDTreeNode* CvDTree::**get_root**() const

CvDTree::get_pruned_tree_idx

Returns the `CvDTree::pruned_tree_idx` parameter.

C++: int CvDTree::**get_pruned_tree_idx**() const

The parameter `DTree::pruned_tree_idx` is used to prune a decision tree. See the `CvDTreeNode::Tn` parameter.

CvDTree::get_data

Returns used train data of the decision tree.

C++: const CvDTreeTrainData* CvDTree::**get_data**() const

Example: building a tree for classifying mushrooms. See the `mushroom.cpp` sample that demonstrates how to build and use the decision tree.

9.6 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 that provides 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. However, many of them smartly combine results to a strong classifier that 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 $(x_i, y_i)1N$ 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 to the learning task at hand. The desired two-class output is encoded as -1 and +1.

Different variants of boosting are known as Discrete Adaboost, Real AdaBoost, LogitBoost, and Gentle AdaBoost [FHT98]. All of them are very similar in their overall structure. Therefore, this chapter focuses only on the standard two-class Discrete AdaBoost algorithm, outlined below. Initially the same weight is assigned to each sample (step 2). Then, 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 that 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).

Two-class Discrete AdaBoost Algorithm

1. Set N examples $(x_i, y_i)1N$ with $x_i \in R^K, y_i \in -1, +1$.

2. Assign weights as $w_i = 1/N, i = 1, ..., N$.

3. Repeat for $m = 1, 2, ..., M$:

    3.1. Fit the classifier $f_m(x) \in -1, 1$, using weights w_i on the training data.

    3.2. Compute $err_m = E_w[1_{(y \neq f_m(x))}], c_m = log((1 - err_m)/err_m)$.

    3.3. Set $w_i \Leftarrow w_i exp[c_m 1_{(y_i \neq f_m(x_i))}], i = 1, 2, ..., N$, and renormalize so that $\Sigma i w_i = 1$.

4. Classify new samples x using the formula: $sign(\Sigma m = 1 M c_m f_m(x))$.

Note: Similar to the classical boosting methods, the current implementation supports two-class classifiers only. For $M > 2$ classes, there is the **AdaBoost.MH** algorithm (described in [FHT98]) that reduces the problem to the two-class problem, yet with a much larger training set.

To reduce computation time for boosted models without substantially losing accuracy, the influence trimming technique can 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 a very low relative weight have a small impact on the weak classifier training. 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].

CvBoostParams

class **CvBoostParams**
    Boosting training parameters.

The structure is derived from `CvDTreeParams` but not all of the decision tree parameters are supported. In particular, cross-validation is not supported.

All parameters are public. You can initialize them by a constructor and then override some of them directly if you want.

CvBoostParams::CvBoostParams

The constructors.

C++: `CvBoostParams::`**`CvBoostParams`**`()`

C++: `CvBoostParams::`**`CvBoostParams`**`(int` **`boost_type`**`, int` **`weak_count`**`, double` **`weight_trim_rate`**`, int`
                                    `max_depth``, bool` **`use_surrogates`**`, const float*` **`priors`**`)`

>   **Parameters**

>   **boost_type** – Type of the boosting algorithm. Possible values are:

>   – **CvBoost::DISCRETE** Discrete AbaBoost.

>   – **CvBoost::REAL** Real AdaBoost. It is a technique that utilizes confidence-rated predictions and works well with categorical data.

>   – **CvBoost::LOGIT** LogitBoost. It can produce good regression fits.

>   – **CvBoost::GENTLE** Gentle AdaBoost. It puts less weight on outlier data points and for that reason is often good with regression data.

>   Gentle AdaBoost and Real AdaBoost are often the preferable choices.

>   **weak_count** – The number of weak classifiers.

>   **weight_trim_rate** – A threshold between 0 and 1 used to save computational time. Samples with summary weight $\leq 1 - \mathtt{weight\_trim\_rate}$ do not participate in the *next* iteration of training. Set this parameter to 0 to turn off this functionality.

See `CvDTreeParams::CvDTreeParams()` for description of other parameters.

Also there is one structure member that you can set directly:

int **split_criteria**
    Splitting criteria used to choose optimal splits during a weak tree construction. Possible values are:

>   • **CvBoost::DEFAULT** Use the default for the particular boosting method, see below.

>   • **CvBoost::GINI** Use Gini index. This is default option for Real AdaBoost; may be also used for Discrete AdaBoost.

- **CvBoost::MISCLASS** Use misclassification rate. This is default option for Discrete AdaBoost; may be also used for Real AdaBoost.

- **CvBoost::SQERR** Use least squares criteria. This is default and the only option for LogitBoost and Gentle AdaBoost.

Default parameters are:

```
CvBoostParams::CvBoostParams()
{
    boost_type = CvBoost::REAL;
    weak_count = 100;
    weight_trim_rate = 0.95;
    cv_folds = 0;
    max_depth = 1;
}
```

CvBoostTree

class **CvBoostTree**

The weak tree 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: In case of LogitBoost and Gentle AdaBoost, each weak predictor is a regression tree, rather than a classification tree. Even in case of Discrete AdaBoost and Real AdaBoost, the `CvBoostTree::predict` return value (`CvDTreeNode::value`) is not an output class label. A negative value "votes" for class #0, a positive value - for class #1. The votes are weighted. The weight of each individual tree may be increased or decreased using the method `CvBoostTree::scale`.

CvBoost

class **CvBoost**

Boosted tree classifier derived from `CvStatModel`.

CvBoost::CvBoost

Default and training constructors.

C++: CvBoost::**CvBoost**()

C++: CvBoost::**CvBoost**(const Mat& **trainData**, int **tflag**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), const Mat& **varType**=Mat(), const Mat& **missingDataMask**=Mat(), CvBoostParams **params**=CvBoostParams())

C++: CvBoost::**CvBoost**(const CvMat* **trainData**, int **tflag**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, const CvMat* **varType**=0, const CvMat* **missingDataMask**=0, CvBoostParams **params**=CvBoostParams())

Python: cv2.**Boost**(trainData, tflag, responses[, varIdx[, sampleIdx[, varType[, missingDataMask[, params]]]]]) → <Boost object>

The constructors follow conventions of `CvStatModel::CvStatModel()`. See `CvStatModel::train()` for parameters descriptions.

CvBoost::train

Trains a boosted tree classifier.

C++: bool CvBoost::**train**(const Mat& **trainData**, int **tflag**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), const Mat& **var-Type**=Mat(), const Mat& **missingDataMask**=Mat(), CvBoostParams **params**=CvBoostParams(), bool **update**=false)

C++: bool CvBoost::**train**(const CvMat* **trainData**, int **tflag**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, const CvMat* **varType**=0, const Cv-Mat* **missingDataMask**=0, CvBoostParams **params**=CvBoostParams(), bool **update**=false)

C++: bool CvBoost::**train**(CvMLData* **data**, CvBoostParams **params**=CvBoostParams(), bool **up-date**=false)

Python: cv2.Boost.**train**(trainData, tflag, responses[, varIdx[, sampleIdx[, varType[, missingDataMask[, params[, update]]]]]]) → retval

> **Parameters**
>
>> **update** – Specifies whether the classifier needs to be updated (`true`, the new weak tree classifiers added to the existing ensemble) or the classifier needs to be rebuilt from scratch (`false`).

The train method follows the common template of `CvStatModel::train()`. The responses must be categorical, which means that boosted trees cannot be built for regression, and there should be two classes.

CvBoost::predict

Predicts a response for an input sample.

C++: float CvBoost::**predict**(const Mat& **sample**, const Mat& **missing**=Mat(), const Range& **slice**=Range::all(), bool **rawMode**=false, bool **returnSum**=false) const

C++: float CvBoost::**predict**(const CvMat* **sample**, const CvMat* **missing**=0, CvMat* **weak_responses**=0, CvSlice **slice**=CV_WHOLE_SEQ, bool **raw_mode**=false, bool **return_sum**=false) const

Python: cv2.Boost.**predict**(sample[, missing[, slice[, rawMode[, returnSum]]]]) → retval

> **Parameters**
>
>> **sample** – Input sample.
>>
>> **missing** – Optional mask of missing measurements. To handle missing measurements, the weak classifiers must include surrogate splits (see `CvDTreeParams::use_surrogates`).
>>
>> **weak_responses** – Optional output parameter, a floating-point vector with responses of each individual weak classifier. The number of elements in the vector must be equal to the slice length.
>>
>> **slice** – Continuous subset of the sequence of weak classifiers to be used for prediction. By default, all the weak classifiers are used.
>>
>> **raw_mode** – Normally, it should be set to `false`.
>>
>> **return_sum** – If `true` then return sum of votes instead of the class label.

The method runs the sample through the trees in the ensemble and returns the output class label based on the weighted voting.

CvBoost::prune

Removes the specified weak classifiers.

C++: void CvBoost::**prune**(CvSlice **slice**)

Python: cv2.Boost.**prune**(slice) → None

> **Parameters**
>
> > **slice** – Continuous subset of the sequence of weak classifiers to be removed.

The method removes the specified weak classifiers from the sequence.

Note: Do not confuse this method with the pruning of individual decision trees, which is currently not supported.

CvBoost::calc_error

Returns error of the boosted tree classifier.

C++: float CvBoost::**calc_error**(CvMLData* **_data**, int **type**, std::vector<float>* **resp**=0)

The method is identical to `CvDTree::calc_error()` but uses the boosted tree classifier as predictor.

CvBoost::get_weak_predictors

Returns the sequence of weak tree classifiers.

C++: CvSeq* CvBoost::**get_weak_predictors**()

The method returns the sequence of weak classifiers. Each element of the sequence is a pointer to the `CvBoostTree` class or to some of its derivatives.

CvBoost::get_params

Returns current parameters of the boosted tree classifier.

C++: const CvBoostParams& CvBoost::**get_params**() const

CvBoost::get_data

Returns used train data of the boosted tree classifier.

C++: const CvDTreeTrainData* CvBoost::**get_data**() const

9.7 Gradient Boosted Trees

Gradient Boosted Trees (GBT) is a generalized boosting algorithm introduced by Jerome Friedman: http://www.salfordsystems.com/doc/GreedyFuncApproxSS.pdf . In contrast to the AdaBoost.M1 algorithm, GBT can deal with both multiclass classification and regression problems. Moreover, it can use any differential loss function, some popular ones are implemented. Decision trees (`CvDTree`) usage as base learners allows to process ordered and categorical variables.

Training the GBT model

Gradient Boosted Trees model represents an ensemble of single regression trees built in a greedy fashion. Training procedure is an iterative proccess similar to the numerical optimization via the gradient descent method. Summary loss on the training set depends only on the current model predictions for the thaining samples, in other words $\sum_{i=1}^{N} L(y_i, F(x_i)) \equiv \mathcal{L}(F(x_1), F(x_2), ..., F(x_N)) \equiv \mathcal{L}(F)$. And the $\mathcal{L}(F)$ gradient can be computed as follows:

$$grad(\mathcal{L}(F)) = \left(\frac{\partial L(y_1, F(x_1))}{\partial F(x_1)}, \frac{\partial L(y_2, F(x_2))}{\partial F(x_2)}, ..., \frac{\partial L(y_N, F(x_N))}{\partial F(x_N)} \right).$$

At every training step, a single regression tree is built to predict an antigradient vector components. Step length is computed corresponding to the loss function and separately for every region determined by the tree leaf. It can be eliminated by changing values of the leaves directly.

See below the main scheme of the training proccess:

1. Find the best constant model.

2. For i in $[1, M]$:

    (a) Compute the antigradient.

    (b) Grow a regression tree to predict antigradient components.

    (c) Change values in the tree leaves.

    (d) Add the tree to the model.

The following loss functions are implemented for regression problems:

- Squared loss (`CvGBTrees::SQUARED_LOSS`): $L(y, f(x)) = \frac{1}{2}(y - f(x))^2$

- Absolute loss (`CvGBTrees::ABSOLUTE_LOSS`): $L(y, f(x)) = |y - f(x)|$

- Huber loss (`CvGBTrees::HUBER_LOSS`): $L(y, f(x)) = \begin{cases} \delta \cdot \left(|y - f(x)| - \dfrac{\delta}{2} \right) & : |y - f(x)| > \delta \\ \dfrac{1}{2} \cdot (y - f(x))^2 & : |y - f(x)| \leq \delta \end{cases}$,

    where δ is the α-quantile estimation of the $|y - f(x)|$. In the current implementation $\alpha = 0.2$.

The following loss functions are implemented for classification problems:

- Deviance or cross-entropy loss (`CvGBTrees::DEVIANCE_LOSS`): K functions are built, one function for each output class, and $L(y, f_1(x), ..., f_K(x)) = -\sum_{k=0}^{K} 1(y = k) \ln p_k(x)$, where $p_k(x) = \dfrac{\exp f_k(x)}{\sum_{i=1}^{K} \exp f_i(x)}$ is the estimation of the probability of $y = k$.

As a result, you get the following model:

$$f(x) = f_0 + \nu \cdot \sum_{i=1}^{M} T_i(x),$$

where f_0 is the initial guess (the best constant model) and ν is a regularization parameter from the interval $(0, 1]$, futher called *shrinkage*.

Predicting with the GBT Model

To get the GBT model prediciton, you need to compute the sum of responses of all the trees in the ensemble. For regression problems, it is the answer. For classification problems, the result is $\arg\max_{i=1..K}(f_i(x))$.

CvGBTreesParams

class `CvGBTreesParams`

GBT training parameters.

The structure contains parameters for each sigle decision tree in the ensemble, as well as the whole model characteristics. The structure is derived from `CvDTreeParams` but not all of the decision tree parameters are supported: cross-validation, pruning, and class priorities are not used.

CvGBTreesParams::CvGBTreesParams

C++: `CvGBTreesParams::`**`CvGBTreesParams`**`()`

C++: `CvGBTreesParams::`**`CvGBTreesParams`**`(int **loss_function_type**, int **weak_count**, float **shrinkage**, float **subsample_portion**, int **max_depth**, bool **use_surrogates**)`

> **Parameters**
>
> > **loss_function_type** – Type of the loss function used for training (see *Training the GBT model*). It must be one of the following types: `CvGBTrees::SQUARED_LOSS`, `CvGBTrees::ABSOLUTE_LOSS`, `CvGBTrees::HUBER_LOSS`, `CvGBTrees::DEVIANCE_LOSS`. The first three types are used for regression problems, and the last one for classification.
> >
> > **weak_count** – Count of boosting algorithm iterations. `weak_count*K` is the total count of trees in the GBT model, where `K` is the output classes count (equal to one in case of a regression).
> >
> > **shrinkage** – Regularization parameter (see *Training the GBT model*).
> >
> > **subsample_portion** – Portion of the whole training set used for each algorithm iteration. Subset is generated randomly. For more information see http://www.salfordsystems.com/doc/StochasticBoostingSS.pdf.
> >
> > **max_depth** – Maximal depth of each decision tree in the ensemble (see `CvDTree`).
> >
> > **use_surrogates** – If `true`, surrogate splits are built (see `CvDTree`).

By default the following constructor is used:

```
CvGBTreesParams(CvGBTrees::SQUARED_LOSS, 200, 0.8f, 0.01f, 3, false)
    : CvDTreeParams( 3, 10, 0, false, 10, 0, false, false, 0 )
```

CvGBTrees

class `CvGBTrees`

The class implements the Gradient boosted tree model as described in the beginning of this section.

CvGBTrees::CvGBTrees

Default and training constructors.

C++: `CvGBTrees::CvGBTrees()`

C++: `CvGBTrees::CvGBTrees`(const Mat& **trainData**, int **tflag**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), const Mat& **varType**=Mat(), const Mat& **missingDataMask**=Mat(), CvGBTreesParams **params**=CvGBTreesParams())

C++: `CvGBTrees::CvGBTrees`(const CvMat* **trainData**, int **tflag**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, const CvMat* **varType**=0, const CvMat* **missingDataMask**=0, CvGBTreesParams **params**=CvGBTreesParams())

Python: `cv2.GBTrees`([trainData, tflag, responses[, varIdx[, sampleIdx[, varType[, missingDataMask[, params]]]]]]) → <GBTrees object>

The constructors follow conventions of `CvStatModel::CvStatModel()`. See `CvStatModel::train()` for parameters descriptions.

CvGBTrees::train

Trains a Gradient boosted tree model.

C++: bool `CvGBTrees::train`(const Mat& **trainData**, int **tflag**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), const Mat& **varType**=Mat(), const Mat& **missingDataMask**=Mat(), CvGBTreesParams **params**=CvGBTreesParams(), bool **update**=false)

C++: bool `CvGBTrees::train`(const CvMat* **trainData**, int **tflag**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, const CvMat* **varType**=0, const CvMat* **missingDataMask**=0, CvGBTreesParams **params**=CvGBTreesParams(), bool **update**=false)

C++: bool `CvGBTrees::train`(CvMLData* **data**, CvGBTreesParams **params**=CvGBTreesParams(), bool **update**=false)

Python: `cv2.GBTrees.train`(trainData, tflag, responses[, varIdx[, sampleIdx[, varType[, missingDataMask[, params[, update]]]]]]) → retval

The first train method follows the common template (see `CvStatModel::train()`). Both tflag values (`CV_ROW_SAMPLE`, `CV_COL_SAMPLE`) are supported. `trainData` must be of the `CV_32F` type. `responses` must be a matrix of type `CV_32S` or `CV_32F`. In both cases it is converted into the `CV_32F` matrix inside the training procedure. `varIdx` and `sampleIdx` must be a list of indices (`CV_32S`) or a mask (`CV_8U` or `CV_8S`). `update` is a dummy parameter.

The second form of `CvGBTrees::train()` function uses `CvMLData` as a data set container. `update` is still a dummy parameter.

All parameters specific to the GBT model are passed into the training function as a `CvGBTreesParams` structure.

CvGBTrees::predict

Predicts a response for an input sample.

C++: float `CvGBTrees::predict`(const Mat& **sample**, const Mat& **missing**=Mat(), const Range& **slice**=Range::all(), int **k**=-1) const

C++: float CvGBTrees::**predict**(const CvMat* **sample**, const CvMat* **missing**=0, CvMat* **weakResponses**=0, CvSlice **slice**=CV_WHOLE_SEQ, int **k**=-1) const

Python: cv2.GBTrees.**predict**(sample[, missing[, slice[, k]]]) → retval

> **Parameters**

> > **sample** – Input feature vector that has the same format as every training set element. If not all the variables were actualy used during training, sample contains forged values at the appropriate places.

> > **missing** – Missing values mask, which is a dimentional matrix of the same size as sample having the CV_8U type. 1 corresponds to the missing value in the same position in the sample vector. If there are no missing values in the feature vector, an empty matrix can be passed instead of the missing mask.

> > **weak_responses** – Matrix used to obtain predictions of all the trees. The matrix has K rows, where K is the count of output classes (1 for the regression case). The matrix has as many columns as the slice length.

> > **slice** – Parameter defining the part of the ensemble used for prediction. If slice = Range::all(), all trees are used. Use this parameter to get predictions of the GBT models with different ensemble sizes learning only one model.

> > **k** – Number of tree ensembles built in case of the classification problem (see *Training the GBT model*). Use this parameter to change the ouput to sum of the trees' predictions in the k-th ensemble only. To get the total GBT model prediction, k value must be -1. For regression problems, k is also equal to -1.

The method predicts the response corresponding to the given sample (see *Predicting with the GBT Model*). The result is either the class label or the estimated function value. The CvGBTrees::predict() method enables using the parallel version of the GBT model prediction if the OpenCV is built with the TBB library. In this case, predictions of single trees are computed in a parallel fashion.

CvGBTrees::clear

Clears the model.

C++: void CvGBTrees::**clear**()

Python: cv2.GBTrees.**clear**() → None

The function deletes the data set information and all the weak models and sets all internal variables to the initial state. The function is called in CvGBTrees::train() and in the destructor.

CvGBTrees::calc_error

Calculates a training or testing error.

C++: float CvGBTrees::**calc_error**(CvMLData* **_data**, int **type**, std::vector<float>* **resp**=0)

> **Parameters**

> > **_data** – Data set.

> > **type** – Parameter defining the error that should be computed: train (CV_TRAIN_ERROR) or test (CV_TEST_ERROR).

> > **resp** – If non-zero, a vector of predictions on the corresponding data set is returned.

If the `CvMLData` data is used to store the data set, `CvGBTrees::calc_error()` can be used to get a training/testing error easily and (optionally) all predictions on the training/testing set. If the Intel* TBB* library is used, the error is computed in a parallel way, namely, predictions for different samples are computed at the same time. In case of a regression problem, a mean squared error is returned. For classifications, the result is a misclassification error in percent.

9.8 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 recieved the majority of "votes". In case of a 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 different training sets. These sets are generated from the original training set using the bootstrap procedure: for each training set, you 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 trained tree, not all the variables are used to find the best split, but 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 $\sqrt{number\_of\_variables}$ by default. None of the built trees 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 follows:

1. Get a prediction for each vector, which is oob relative to the i-th tree, using the very i-th tree.

2. After all the trees have been trained, for each vector that has ever been oob, find the class-*winner* for it (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.

3. Compute the classification error estimate as a ratio of the number of misclassified oob vectors to all the vectors in the original data. In case of regression, the oob-error is computed as the squared error for oob vectors difference divided by the total number of vectors.

For the random trees usage example, please, see letter_recog.cpp sample in OpenCV distribution.

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

CvRTParams

class `CvRTParams`
    Training parameters of random trees.

The set of training parameters for the forest is a 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.

CvRTParams::CvRTParams:

The constructors.

C++: CvRTParams::**CvRTParams**()

C++: CvRTParams::**CvRTParams**(int **max_depth**, int **min_sample_count**, float **regression_accuracy**, bool **use_surrogates**, int **max_categories**, const float* **priors**, bool **calc_var_importance**, int **nactive_vars**, int **max_num_of_trees_in_the_forest**, float **forest_accuracy**, int **termcrit_type**)

> **Parameters**
>
> > **calc_var_importance** – If true then variable importance will be calculated and then it can be retrieved by `CvRTrees::get_var_importance()`.
> >
> > **nactive_vars** – The size of the randomly selected subset of features at each tree node and that are used to find the best split(s). If you set it to 0 then the size will be set to the square root of the total number of features.
> >
> > **max_num_of_trees_in_the_forest** – The maximum number of trees in the forest (suprise, suprise).
> >
> > **forest_accuracy** – Sufficient accuracy (OOB error).
> >
> > **termcrit_type** – The type of the termination criteria:
> >
> > – **CV_TERMCRIT_ITER** Terminate learning by the `max_num_of_trees_in_the_forest`;
> >
> > – **CV_TERMCRIT_EPS** Terminate learning by the `forest_accuracy`;
> >
> > – **CV_TERMCRIT_ITER | CV_TERMCRIT_EPS** Use both termination criterias.

For meaning of other parameters see `CvDTreeParams::CvDTreeParams()`.

The default constructor sets all parameters to default values which are different from default values of `CvDTreeParams`:

```
CvRTParams::CvRTParams() : CvDTreeParams( 5, 10, 0, false, 10, 0, false, false, 0 ),
    calc_var_importance(false), nactive_vars(0)
{
    term_crit = cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 50, 0.1 );
}
```

CvRTrees

class CvRTrees
> The class implements the random forest predictor as described in the beginning of this section.

CvRTrees::train

Trains the Random Trees model.

C++: bool CvRTrees::**train**(const Mat& **trainData**, int **tflag**, const Mat& **responses**, const Mat& **varIdx**=Mat(), const Mat& **sampleIdx**=Mat(), const Mat& **varType**=Mat(), const Mat& **missingDataMask**=Mat(), CvRTParams **params**=CvRTParams())

C++: bool CvRTrees::**train**(const CvMat* **trainData**, int **tflag**, const CvMat* **responses**, const CvMat* **varIdx**=0, const CvMat* **sampleIdx**=0, const CvMat* **varType**=0, const CvMat* **missingDataMask**=0, CvRTParams **params**=CvRTParams())

C++: bool CvRTrees::**train**(CvMLData* **data**, CvRTParams **params**=CvRTParams())

Python: cv2.RTrees.**train**(trainData, tflag, responses[, varIdx[, sampleIdx[, varType[, missingDataMask[, params]]]]]) → retval

The method `CvRTrees::train()` is very similar to the method `CvDTree::train()` and follows the generic method CvStatModel::train() conventions. All the parameters specific to the algorithm training are passed as a `CvRTParams` instance. The estimate of the training error (`oob-error`) is stored in the protected class member `oob_error`.

CvRTrees::predict

Predicts the output for an input sample.

C++: double CvRTrees::**predict**(const Mat& **sample**, const Mat& **missing**=Mat()) const

C++: float CvRTrees::**predict**(const CvMat* **sample**, const CvMat* **missing**=0) const

Python: cv2.RTrees.**predict**(sample[, missing]) → retval

> **Parameters**
>
> > **sample** – Sample for classification.
> >
> > **missing** – Optional missing measurement mask of the 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::predict_prob

Returns a fuzzy-predicted class label.

C++: float CvRTrees::**predict_prob**(const cv::Mat& **sample**, const cv::Mat& **missing**=cv::Mat()) const

C++: float CvRTrees::**predict_prob**(const CvMat* **sample**, const CvMat* **missing**=0) const

Python: cv2.RTrees.**predict_prob**(sample[, missing]) → retval

> **Parameters**
>
> > **sample** – Sample for classification.
> >
> > **missing** – Optional missing measurement mask of the sample.

The function works for binary classification problems only. It returns the number between 0 and 1. This number represents probability or confidence of the sample belonging to the second class. It is calculated as the proportion of decision trees that classified the sample to the second class.

CvRTrees::getVarImportance

Returns the variable importance array.

C++: Mat CvRTrees::**getVarImportance**()

C++: const CvMat* CvRTrees::**get_var_importance**()

Python: cv2.RTrees.**getVarImportance**() → importanceVector

The method returns the variable importance vector, computed at the training stage when CvRTParams::calc_var_importance is set to true. If this flag was set to false, the NULL pointer is returned. This differs from the decision trees where variable importance can be computed anytime after the training.

CvRTrees::get_proximity

Retrieves the proximity measure between two training samples.

C++: float CvRTrees::**get_proximity**(const CvMat* **sample1**, const CvMat* **sample2**, const CvMat* **missing1**=0, const CvMat* **missing2**=0) const

> **Parameters**
>
> > **sample_1** – The first sample.
> >
> > **sample_2** – The second sample.
> >
> > **missing1** – Optional missing measurement mask of the first sample.
> >
> > **missing2** – Optional missing measurement mask of the second sample.

The method returns proximity measure between any two samples. This is a ratio of those trees in the ensemble, in which the samples fall into the same leaf node, to the total number of the trees.

CvRTrees::calc_error

Returns error of the random forest.

C++: float CvRTrees::**calc_error**(CvMLData* **data**, int **type**, std::vector<float>* **resp**=0)

The method is identical to CvDTree::calc_error() but uses the random forest as predictor.

CvRTrees::get_train_error

Returns the train error.

C++: float CvRTrees::**get_train_error**()

The method works for classification problems only. It returns the proportion of incorrectly classified train samples.

CvRTrees::get_rng

Returns the state of the used random number generator.

C++: CvRNG* CvRTrees::**get_rng**()

CvRTrees::get_tree_count

Returns the number of trees in the constructed random forest.

C++: int CvRTrees::**get_tree_count**() const

CvRTrees::get_tree

Returns the specific decision tree in the constructed random forest.

C++: CvForestTree* CvRTrees::**get_tree**(int **i**) const

> **Parameters**
>
> > **i** – Index of the decision tree.

9.9 Extremely randomized trees

Extremely randomized trees have been introduced by Pierre Geurts, Damien Ernst and Louis Wehenkel in the article "Extremely randomized trees", 2006 [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7485&rep=rep1&type=pdf]. The algorithm of growing Extremely randomized trees is similar to *Random Trees* (Random Forest), but there are two differences:

1. Extremely randomized trees don't apply the bagging procedure to constract the training samples for each tree. The same input training set is used to train all trees.

2. Extremely randomized trees pick a node split very extremely (both a variable index and variable spliting value are chosen randomly), whereas Random Forest finds the best split (optimal one by variable index and variable spliting value) among random subset of variables.

CvERTrees

class CvERTrees
> The class implements the Extremely randomized trees algorithm. `CvERTrees` is inherited from `CvRTrees` and has the same interface, so see description of `CvRTrees` class to get detailes. To set the training parameters of Extremely randomized trees the same class `CvRTParams` is used.

9.10 Expectation Maximization

The Expectation Maximization(EM) 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 N feature vectors { $x_1, x_2, ..., x_N$ } from a d-dimensional Euclidean space drawn from a Gaussian mixture:

$$p(x; a_k, S_k, \pi_k) = \sum_{k=1}^{m} \pi_k p_k(x), \quad \pi_k \geq 0, \quad \sum_{k=1}^{m} \pi_k = 1,$$

$$p_k(x) = \varphi(x; a_k, S_k) = \frac{1}{(2\pi)^{d/2} \mid S_k \mid^{1/2}} exp \left\{ -\frac{1}{2}(x - a_k)^{\mathsf{T}} S_k^{-1}(x - a_k) \right\},$$

where m is the number of mixtures, p_k is the normal distribution density with the mean a_k and covariance matrix S_k, π_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 all the mixture parameters, that is, a_k, S_k and π_k :

$$L(x, \theta) = \log p(x, \theta) = \sum_{i=1}^{N} \log \left(\sum_{k=1}^{m} \pi_k p_k(x) \right) \to \max_{\theta \in \Theta},$$

$$\Theta = \left\{ (a_k, S_k, \pi_k) : a_k \in \mathbb{R}^d, S_k = S_k^T > 0, S_k \in \mathbb{R}^{d \times d}, \pi_k \geq 0, \sum_{k=1}^{m} \pi_k = 1 \right\}.$$

The EM algorithm is an iterative procedure. Each iteration includes two steps. At the first step (Expectation step or E-step), you 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 (including macnine learning) the `kmeans()` algorithm is used for that purpose.

One of the main problems of the EM algorithm is a large number of parameters to estimate. The majority of the parameters reside 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 an identity matrix and μ_k is a mixture-dependent "scale" parameter. So, a robust computation scheme could start with 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

class `CvEMParams`

Parameters of the EM algorithm. All parameters are public. You can initialize them by a constructor and then override some of them directly if you want.

CvEMParams::CvEMParams

The constructors

C++: `CvEMParams::`**`CvEMParams`**`()`

C++: `CvEMParams::`**`CvEMParams`**(int **nclusters**, int **cov_mat_type**=CvEM::COV_MAT_DIAGONAL, int **start_step**=CvEM::START_AUTO_STEP, CvTermCriteria **term_crit**=cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 100, FLT_EPSILON), const CvMat* **probs**=0, const CvMat* **weights**=0, const CvMat* **means**=0, const CvMat** **covs**=0)

> **Parameters**

>> **nclusters** – The number of mixture components in the gaussian mixture model. Some of EM implementation could determine the optimal number of mixtures within a specified value range, but that is not the case in ML yet.

>> **cov_mat_type** – Constraint on covariance matrices which defines type of matrices. Possible values are:

>>> – **CvEM::COV_MAT_SPHERICAL** A scaled identity matrix $\mu_k * I$. There is the only parameter μ_k to be estimated for earch matrix. The option may be used in special cases, when the constraint is relevant, or as a first step in the optimization (for example in case when the data is preprocessed with PCA). The results of such preliminary estimation may be passed again to the optimization procedure, this time with `cov_mat_type=CvEM::COV_MAT_DIAGONAL`.

>>> – **CvEM::COV_MAT_DIAGONAL** A diagonal matrix with positive diagonal elements. The number of free parameters is d for each matrix. This is most commonly used option yielding good estimation results.

>>> – **CvEM::COV_MAT_GENERIC** A symmetric positively defined matrix. The number of free parameters in each matrix is about $d^2/2$. It is not recommended to use this option, unless there is pretty accurate initial estimation of the parameters and/or a huge number of training samples.

>> **start_step** – The start step of the EM algorithm:

>>> – **CvEM::START_E_STEP** Start with Expectation step. You need to provide means a_k of mixture components to use this option. Optionally you can pass weights π_k and covariance matrices S_k of mixture components.

>>> – **CvEM::START_M_STEP** Start with Maximization step. You need to provide initial probabilites $p_{i,k}$ to use this option.

>>> – **CvEM::START_AUTO_STEP** Start with Expectation step. You need not provide any parameters because they will be estimated by the k-means algorithm.

>> **term_crit** – The termination criteria of the EM algorithm. The EM algorithm can be terminated by the number of iterations `term_crit.max_iter` (number of M-steps) or when relative change of likelihood logarithm is less than `term_crit.epsilon`.

>> **probs** – Initial probabilities $p_{i,k}$ of sample i to belong to mixture component k. It is a floating-point matrix of $nsamples \times nclusters$ size. It is used and must be not NULL only when `start_step=CvEM::START_M_STEP`.

>> **weights** – Initial weights π_k of mixture components. It is a floating-point vector with $nclusters$ elements. It is used (if not NULL) only when `start_step=CvEM::START_E_STEP`.

>> **means** – Initial means a_k of mixture components. It is a floating-point matrix of $nclusters \times dims$ size. It is used used and must be not NULL only when `start_step=CvEM::START_E_STEP`.

> **covs** – Initial covariance matrices S_k of mixture components. Each of covariance matrices is a valid square floating-point matrix of $dims \times dims$ size. It is used (if not NULL) only when `start_step=CvEM::START_E_STEP`.

The default constructor represents a rough rule-of-the-thumb:

```
CvEMParams() : nclusters(10), cov_mat_type(1/*CvEM::COV_MAT_DIAGONAL*/),
    start_step(0/*CvEM::START_AUTO_STEP*/), probs(0), weights(0), means(0), covs(0)
{
    term_crit=cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 100, FLT_EPSILON );
}
```

With another contstructor it is possible to override a variety of parameters from a single number of mixtures (the only essential problem-dependent parameter) to initial values for the mixture parameters.

CvEM

class CvEM

> The class implements the EM algorithm as described in the beginning of this section.

CvEM::train

Estimates the Gaussian mixture parameters from a sample set.

C++: void CvEM::**train**(const Mat& **samples**, const Mat& **sample_idx**=Mat(), CvEMParams **params**=CvEMParams(), Mat* **labels**=0)

C++: bool CvEM::**train**(const CvMat* **samples**, const CvMat* **sampleIdx**=0, CvEMParams **params**=CvEMParams(), CvMat* **labels**=0)

Python: cv2.EM.**train**(samples[, sampleIdx[, params]]) → retval, labels

> **Parameters**
>
> > **samples** – Samples from which the Gaussian mixture model will be estimated.
> >
> > **sample_idx** – Mask of samples to use. All samples are used by default.
> >
> > **params** – Parameters of the EM algorithm.
> >
> > **labels** – The optional output "class label" for each sample: $labels_i = \arg \max_k(p_{i,k}), i = 1..N$ (indices of the most probable mixture component for each sample).

Unlike many of the ML models, EM is an unsupervised learning algorithm and it does not take responses (class labels or function values) as input. Instead, it computes the *Maximum Likelihood Estimate* of the Gaussian mixture parameters from an input sample set, stores all the parameters inside the structure: $p_{i,k}$ in `probs`, a_k in `means`, S_k in `covs[k]`, π_k in `weights`, and optionally computes the output "class label" for each sample: $labels_i = \arg \max_k(p_{i,k}), i = 1..N$ (indices of the most probable mixture component for each sample).

The trained model can be used further for prediction, just like any other classifier. The trained model is similar to the `CvNormalBayesClassifier`.

For an example of clustering random samples of the multi-Gaussian distribution using EM, see `em.cpp` sample in the OpenCV distribution.

CvEM::predict

Returns a mixture component index of a sample.

C++: float CvEM::**predict**(const Mat& **sample**, Mat* **probs**=0) const

C++: float CvEM::**predict**(const CvMat* **sample**, CvMat* **probs**) const

Python: cv2.EM.**predict**(sample) $\rightarrow$ retval, probs

> **Parameters**
>
> > **sample** – A sample for classification.
> >
> > **probs** – If it is not null then the method will write posterior probabilities of each component given the sample data to this parameter.

CvEM::getNClusters

Returns the number of mixture components M in the gaussian mixture model.

C++: int CvEM::**getNClusters**() const

C++: int CvEM::**get_nclusters**() const

Python: cv2.EM.**getNClusters**() $\rightarrow$ retval

CvEM::getMeans

Returns mixture means a_k.

C++: Mat CvEM::**getMeans**() const

C++: const CvMat* CvEM::**get_means**() const

Python: cv2.EM.**getMeans**() $\rightarrow$ means

CvEM::getCovs

Returns mixture covariance matrices S_k.

C++: void CvEM::**getCovs**(std::vector<cv::Mat>& **covs**) const

C++: const CvMat** CvEM::**get_covs**() const

Python: cv2.EM.**getCovs**($\big[$covs$\big]$) $\rightarrow$ covs

CvEM::getWeights

Returns mixture weights π_k.

C++: Mat CvEM::**getWeights**() const

C++: const CvMat* CvEM::**get_weights**() const

Python: cv2.EM.**getWeights**() $\rightarrow$ weights

CvEM::getProbs

Returns vectors of probabilities for each training sample.

C++: Mat CvEM::**getProbs**() const

C++: const CvMat* CvEM::**get_probs**() const

Python: cv2.EM.**getProbs**() → probs

For each training sample i (that have been passed to the constructor or to CvEM::train()) returns probabilites $p_{i,k}$ to belong to a mixture component k.

CvEM::getLikelihood

Returns logarithm of likelihood.

C++: double CvEM::**getLikelihood**() const

C++: double CvEM::**get_log_likelihood**() const

Python: cv2.EM.**getLikelihood**() → likelihood

CvEM::getLikelihoodDelta

Returns difference between logarithm of likelihood on the last iteration and logarithm of likelihood on the previous iteration.

C++: double CvEM::**getLikelihoodDelta**() const

C++: double CvEM::**get_log_likelihood_delta**() const

Python: cv2.EM.**getLikelihoodDelta**() → likelihood delta

CvEM::write_params

Writes used parameters of the EM algorithm to a file storage.

C++: void CvEM::**write_params**(CvFileStorage* **fs**) const

> **Parameters**
>
> > **fs** – A file storage where parameters will be written.

CvEM::read_params

Reads parameters of the EM algorithm.

C++: void CvEM::**read_params**(CvFileStorage* **fs**, CvFileNode* **node**)

> **Parameters**
>
> > **fs** – A file storage with parameters of the EM algorithm.
> >
> > **node** – The parent map. If it is NULL, the function searches a node with parameters in all the top-level nodes (streams), starting with the first one.

The function reads EM parameters from the specified file storage node. For example of clustering random samples of multi-Gaussian distribution using EM see em.cpp sample in OpenCV distribution.

9.11 Neural Networks

ML implements feed-forward artificial neural networks or, 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 directionally linked with the neurons from the previous and the next layer. The example below represents a 3-layer perceptron with three inputs, two outputs, and the hidden layer including five neurons:
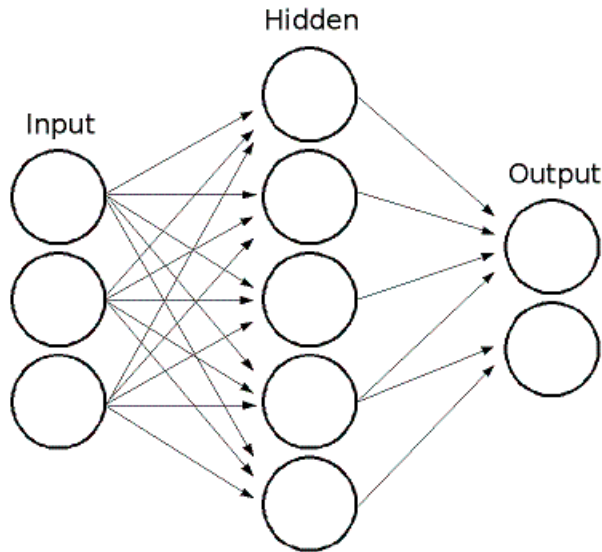
All the neurons in MLP are similar. Each of them has several input links (it takes the output values from several neurons in the previous layer as input) and several output links (it passes the response to several neurons in the next layer). The values retrieved from the previous layer are summed up with certain weights, individual for each neuron, plus the bias term. The sum is transformed using the activation function f that may be also different for different neurons.
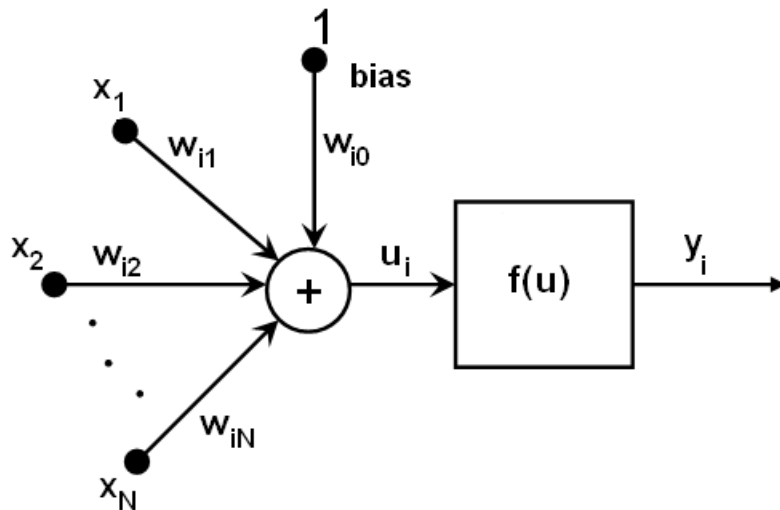

In other words, given the outputs x_j of the layer n , the outputs y_i of the layer $n + 1$ are computed as:

$$u_i = \sum_j (w_{i,j}^{n+1} * x_j) + w_{i,bias}^{n+1}$$

$$y_i = f(u_i)$$

Different activation functions may be used. ML implements three standard functions:

- Identity function (`CvANN_MLP::IDENTITY`): $f(x) = x$

- Symmetrical sigmoid (`CvANN_MLP::SIGMOID_SYM`): $f(x) = \beta * (1 - e^{-\alpha x})/(1 + e^{-\alpha x})$, which is the default choice for MLP. The standard sigmoid with $\beta = 1, \alpha = 1$ is shown below:
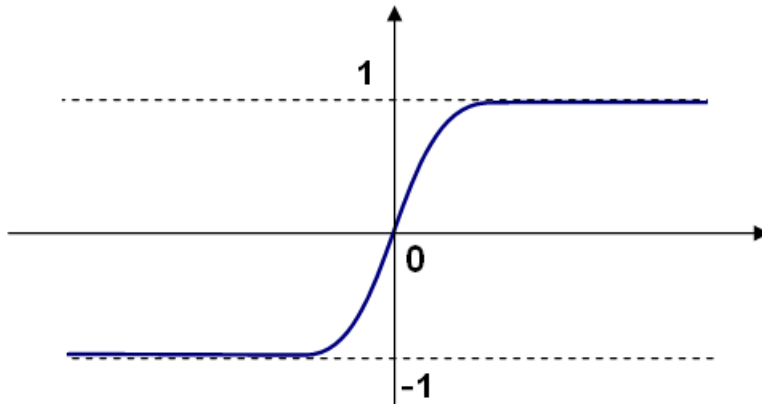
- Gaussian function (`CvANN_MLP::GAUSSIAN`): $f(x) = \beta e^{-\alpha x * x}$, which is not completely supported at the moment.

In ML, all the neurons have the same activation functions, with the same free parameters (α, β) that are specified by user and are not altered by the training algorithms.

So, the whole trained network works as follows:

1. Take the feature vector as input. The vector size is equal to the size of the input layer.

2. Pass values as input to the first hidden layer.

3. Compute outputs of the hidden layer using the weights and the activation functions.

4. Pass outputs further downstream until you compute the output layer.

So, to compute the network, you need 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 enable the network to give the desired response to the provided input vectors.

The larger the network size (the number of hidden layers and their sizes) is, the more the potential network flexibility is. The error on the training set could be made arbitrarily small. But at the same time the learned network also "learns" the noise present in the training set, so the error on the test set usually starts increasing after the network size reaches a limit. Besides, the larger networks are trained much longer than the smaller ones, so it is reasonable to pre-process the data, using `PCA::operator()` or similar technique, and train a smaller network on only essential features.

Another MPL feature is an inability to handle categorical data as is. However, there is a workaround. If a certain feature in the input or output (in 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 a binary tuple of M elements, where the i -th element is 1 if and only if the feature is equal to the i -th value out of M possible. It increases the size of the input/output layer but speeds up the training algorithm convergence and at the same time enables "fuzzy" values of such variables, that is, a tuple of probabilities instead of a fixed value.

ML implements two algorithms for training MLP's. The first algorithm is a classical random sequential back-propagation algorithm. The second (default) one is a batch RPROP algorithm.

CvANN_MLP_TrainParams

class `CvANN_MLP_TrainParams`

Parameters of the MLP training algorithm. You can initialize the structure by a constructor or the individual parameters can be adjusted after the structure is created.

The back-propagation algorithm parameters:

double **`bp_dw_scale`**
> Strength of the weight gradient term. The recommended value is about 0.1.

double **`bp_moment_scale`**
> Strength of the momentum term (the difference between weights on the 2 previous iterations). This parameter provides some inertia to smooth the random fluctuations of the weights. It can vary from 0 (the feature is disabled) to 1 and beyond. The value 0.1 or so is good enough

The RPROP algorithm parameters (see [RPROP93] for details):

double **`rp_dw0`**
> Initial value Δ_0 of update-values Δ_{ij}.

double **`rp_dw_plus`**
> Increase factor η^+. It must be >1.

double **`rp_dw_minus`**
> Decrease factor η^-. It must be <1.

double **`rp_dw_min`**
> Update-values lower limit Δ_{min}. It must be positive.

double **`rp_dw_max`**
> Update-values upper limit Δ_{max}. It must be >1.

CvANN_MLP_TrainParams::CvANN_MLP_TrainParams

The constructors.

C++: CvANN_MLP_TrainParams::**`CvANN_MLP_TrainParams`**()

C++: CvANN_MLP_TrainParams::**`CvANN_MLP_TrainParams`**(CvTermCriteria **term_crit**, int **train_method**, double **param1**, double **param2**=0)

> **Parameters**

>> **term_crit** – Termination criteria of the training algorithm. You can specify the maximum number of iterations (`max_iter`) and/or how much the error could change between the iterations to make the algorithm continue (`epsilon`).

>> **train_method** – Training method of the MLP. Possible values are:

>> – **CvANN_MLP_TrainParams::BACKPROP** The back-propagation algorithm.

>> – **CvANN_MLP_TrainParams::RPROP** The RPROP algorithm.

>> **param1** – Parameter of the training method. It is `rp_dw0` for RPROP and `bp_dw_scale` for BACKPROP.

>> **param2** – Parameter of the training method. It is `rp_dw_min` for RPROP and `bp_moment_scale` for BACKPROP.

By default the RPROP algorithm is used:

```
CvANN_MLP_TrainParams::CvANN_MLP_TrainParams()
{
    term_crit = cvTermCriteria( CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 1000, 0.01 );
    train_method = RPROP;
    bp_dw_scale = bp_moment_scale = 0.1;
    rp_dw0 = 0.1; rp_dw_plus = 1.2; rp_dw_minus = 0.5;
    rp_dw_min = FLT_EPSILON; rp_dw_max = 50.;
}
```

CvANN_MLP

class **CvANN_MLP**

MLP model.

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 `CvANN_MLP::create()`. All the weights are set to zeros. Then, the network is trained using a set of input and output vectors. The training procedure can be repeated more than once, that is, the weights can be adjusted based on the new training data.

CvANN_MLP::CvANN_MLP

The constructors.

C++: `CvANN_MLP::`**`CvANN_MLP`**`()`

C++: `CvANN_MLP::`**`CvANN_MLP`**(const CvMat* **layerSizes**, int **activateFunc**=CvANN_MLP::SIGMOID_SYM, double **fparam1**=0, double **fparam2**=0)

Python: `cv2.`**`ANN_MLP`**(layerSizes$[$, activateFunc$[$, fparam1$[$, fparam2$]]]$) → <ANN_MLP object>

The advanced constructor allows to create MLP with the specified topology. See `CvANN_MLP::create()` for details.

CvANN_MLP::create

Constructs MLP with the specified topology.

C++: void `CvANN_MLP::`**`create`**(const Mat& **layerSizes**, int **activateFunc**=CvANN_MLP::SIGMOID_SYM, double **fparam1**=0, double **fparam2**=0)

C++: void `CvANN_MLP::`**`create`**(const CvMat* **layerSizes**, int **activateFunc**=CvANN_MLP::SIGMOID_SYM, double **fparam1**=0, double **fparam2**=0)

Python: `cv2.ANN_MLP.`**`create`**(layerSizes$[$, activateFunc$[$, fparam1$[$, fparam2$]]]$) → None

> **Parameters**

>> **layerSizes** – Integer vector specifying the number of neurons in each layer including the input and output layers.

>> **activateFunc** – Parameter specifying the activation function for each neuron: one of `CvANN_MLP::IDENTITY`, `CvANN_MLP::SIGMOID_SYM`, and `CvANN_MLP::GAUSSIAN`.

>> **fparam1/fparam2** – Free parameters of the activation function, α and β, respectively. See the formulas in the introduction section.

The method creates an MLP network with the specified topology and assigns the same activation function to all the neurons.

CvANN_MLP::train

Trains/updates MLP.

C++: int CvANN_MLP::**train**(const Mat& **inputs**, const Mat& **outputs**, const Mat& **sampleWeights**, const Mat& **sampleIdx**=Mat(), CvANN_MLP_TrainParams **params**=CvANN_MLP_TrainParams(), int **flags**=0)

C++: int CvANN_MLP::**train**(const CvMat* **inputs**, const CvMat* **outputs**, const CvMat* **sampleWeights**, const CvMat* **sampleIdx**=0, CvANN_MLP_TrainParams **params**=CvANN_MLP_TrainParams(), int **flags**=0)

Python: cv2.ANN_MLP.**train**(inputs, outputs, sampleWeights[, sampleIdx[, params[, flags]]]) → niterations

> **Parameters**
>
>> **inputs** – Floating-point matrix of input vectors, one vector per row.
>>
>> **outputs** – Floating-point matrix of the corresponding output vectors, one vector per row.
>>
>> **sampleWeights** – (RPROP only) Optional floating-point vector of weights for each sample. Some samples may be more important than others for training. You may want to raise the weight of certain classes to find the right balance between hit-rate and false-alarm rate, and so on.
>>
>> **sampleIdx** – Optional integer vector indicating the samples (rows of `inputs` and `outputs`) that are taken into account.
>>
>> **params** – Training parameters. See the `CvANN_MLP_TrainParams` description.
>>
>> **flags** – Various parameters to control the training algorithm. A combination of the following parameters is possible:
>>
>>> – **UPDATE_WEIGHTS** Algorithm updates the network weights, rather than computes them from scratch. In the latter case the weights are initialized using the 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 equal to 1. If the network is assumed to be updated frequently, the new training data could be much different from original one. In this case, you 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 feature independently, by transforming it to the certain range depending on the used activation function.

This method applies the specified training algorithm to computing/adjusting the network weights. It returns the number of done iterations.

CvANN_MLP::predict

Predicts responses for input samples.

C++: float CvANN_MLP::**predict**(const Mat& **inputs**, Mat& **outputs**) const

C++: float CvANN_MLP::**predict**(const CvMat* **inputs**, CvMat* **outputs**) const

Python: `cv2.ANN_MLP.`**`predict`**`(inputs, outputs)` → retval

> **Parameters**

>> **inputs** – Input samples.

>> **outputs** – Predicted responses for corresponding samples.

The method returns a dummy value which should be ignored.

CvANN_MLP::get_layer_count

Returns the number of layers in the MLP.

C++: int CvANN_MLP::**`get_layer_count`**()

CvANN_MLP::get_layer_sizes

Returns numbers of neurons in each layer of the MLP.

C++: const CvMat* CvANN_MLP::**`get_layer_sizes`**()

The method returns the integer vector specifying the number of neurons in each layer including the input and output layers of the MLP.

CvANN_MLP::get_weights

Returns neurons weights of the particular layer.

C++: double* CvANN_MLP::**`get_weights`**(int **layer**)

> **Parameters**

>> **layer** – Index of the particular layer.

9.12 MLData

For the machine learning algorithms, the data set is often stored in a file of the `.csv`-like format. The file contains a table of predictor and response values where each row of the table corresponds to a sample. Missing values are supported. The UC Irvine Machine Learning Repository (http://archive.ics.uci.edu/ml/) provides many data sets stored in such a format to the machine learning community. The class `MLData` is implemented to easily load the data for training one of the OpenCV machine learning algorithms. For float values, only the `'.'` separator is supported.

CvMLData

class **CvMLData**

Class for loading the data from a `.csv` file.

```
class CV_EXPORTS CvMLData
{
public:
    CvMLData();
    virtual ~CvMLData();
```

```
    int read_csv(const char* filename);

    const CvMat* get_values() const;
    const CvMat* get_responses();
    const CvMat* get_missing() const;

    void set_response_idx( int idx );
    int get_response_idx() const;


    void set_train_test_split( const CvTrainTestSplit * spl);
    const CvMat* get_train_sample_idx() const;
    const CvMat* get_test_sample_idx() const;
    void mix_train_and_test_idx();

    const CvMat* get_var_idx();
    void chahge_var_idx( int vi, bool state );

    const CvMat* get_var_types();
    void set_var_types( const char* str );

    int get_var_type( int var_idx ) const;
    void change_var_type( int var_idx, int type);

    void set_delimiter( char ch );
    char get_delimiter() const;

    void set_miss_ch( char ch );
    char get_miss_ch() const;

    const std::map<std::string, int>& get_class_labels_map() const;

protected:
    ...
};
```

CvMLData::read_csv

Reads the data set from a `.csv`-like `filename` file and stores all read values in a matrix.

C++: int CvMLData::**read_csv**(const char* **filename**)

> **Parameters**
>
> > **filename** – The input file name

While reading the data, the method tries to define the type of variables (predictors and responses): ordered or categorical. If a value of the variable is not numerical (except for the label for a missing value), the type of the variable is set to `CV_VAR_CATEGORICAL`. If all existing values of the variable are numerical, the type of the variable is set to `CV_VAR_ORDERED`. So, the default definition of variables types works correctly for all cases except the case of a categorical variable with numerical class labeles. In this case, the type `CV_VAR_ORDERED` is set. You should change the type to `CV_VAR_CATEGORICAL` using the method `CvMLData::change_var_type()`. For categorical variables, a common map is built to convert a string class label to the numerical class label. Use `CvMLData::get_class_labels_map()` to obtain this map.

Also, when reading the data, the method constructs the mask of missing values. For example, values are egual to '*?*'.

CvMLData::get_values

Returns a pointer to the matrix of predictors and response values

C++: const CvMat* CvMLData::**get_values**() const

The method returns a pointer to the matrix of predictor and response `values` or `0` if the data has not been loaded from the file yet.

The row count of this matrix equals the sample count. The column count equals predictors + 1 for the response (if exists) count. This means that each row of the matrix contains values of one sample predictor and response. The matrix type is `CV_32FC1`.

CvMLData::get_responses

Returns a pointer to the matrix of response values

C++: const CvMat* CvMLData::**get_responses**()

The method returns a pointer to the matrix of response values or throws an exception if the data has not been loaded from the file yet.

This is a single-column matrix of the type `CV_32FC1`. Its row count is equal to the sample count, one column and .

CvMLData::get_missing

Returns a pointer to the mask matrix of missing values

C++: const CvMat* CvMLData::**get_missing**() const

The method returns a pointer to the mask matrix of missing values or throws an exception if the data has not been loaded from the file yet.

This matrix has the same size as the `values` matrix (see `CvMLData::get_values()`) and the type `CV_8UC1`.

CvMLData::set_response_idx

Specifies index of response column in the data matrix

C++: void CvMLData::**set_response_idx**(int **idx**)

The method sets the index of a response column in the `values` matrix (see `CvMLData::get_values()`) or throws an exception if the data has not been loaded from the file yet.

The old response columns become predictors. If `idx < 0`, there is no response.

CvMLData::get_response_idx

Returns index of the response column in the loaded data matrix

C++: int CvMLData::**get_response_idx**() const

The method returns the index of a response column in the `values` matrix (see `CvMLData::get_values()`) or throws an exception if the data has not been loaded from the file yet.

If `idx < 0`, there is no response.

CvMLData::set_train_test_split

Divides the read data set into two disjoint training and test subsets.

C++: void CvMLData::**set_train_test_split**(const CvTrainTestSplit* **spl**)

This method sets parameters for such a split using spl (see CvTrainTestSplit) or throws an exception if the data has not been loaded from the file yet.

CvMLData::get_train_sample_idx

Returns the matrix of sample indices for a training subset

C++: const CvMat* CvMLData::**get_train_sample_idx**() const

The method returns the matrix of sample indices for a training subset. This is a single-row matrix of the type CV_32SC1. If data split is not set, the method returns 0. If the data has not been loaded from the file yet, an exception is thrown.

CvMLData::get_test_sample_idx

Returns the matrix of sample indices for a testing subset

C++: const CvMat* CvMLData::**get_test_sample_idx**() const

CvMLData::mix_train_and_test_idx

Mixes the indices of training and test samples

C++: void CvMLData::**mix_train_and_test_idx**()

The method shuffles the indices of training and test samples preserving sizes of training and test subsets if the data split is set by CvMLData::get_values(). If the data has not been loaded from the file yet, an exception is thrown.

CvMLData::get_var_idx

Returns the indices of the active variables in the data matrix

C++: const CvMat* CvMLData::**get_var_idx**()

The method returns the indices of variables (columns) used in the values matrix (see CvMLData::get_values()).

It returns 0 if the used subset is not set. It throws an exception if the data has not been loaded from the file yet. Returned matrix is a single-row matrix of the type CV_32SC1. Its column count is equal to the size of the used variable subset.

CvMLData::chahge_var_idx

Enables or disables particular variable in the loaded data

C++: void CvMLData::**chahge_var_idx**(int **vi**, bool **state**)

By default, after reading the data set all variables in the values matrix (see CvMLData::get_values()) are used. But you may want to use only a subset of variables and include/exclude (depending on state value) a variable with the vi index from the used subset. If the data has not been loaded from the file yet, an exception is thrown.

CvMLData::get_var_types

Returns a matrix of the variable types.

C++: const CvMat* CvMLData::`get_var_types`()

The function returns a single-row matrix of the type `CV_8UC1`, where each element is set to either `CV_VAR_ORDERED` or `CV_VAR_CATEGORICAL`. The number of columns is equal to the number of variables. If data has not been loaded from file yet an exception is thrown.

CvMLData::set_var_types

Sets the variables types in the loaded data.

C++: void CvMLData::`set_var_types`(const char* **str**)

In the string, a variable type is followed by a list of variables indices. For example: `"ord[0-17],cat[18]"`, `"ord[0,2,4,10-12], cat[1,3,5-9,13,14]"`, `"cat"` (all variables are categorical), `"ord"` (all variables are ordered).

CvMLData::get_var_type

Returns type of the specified variable

C++: int CvMLData::`get_var_type`(int **var_idx**) const

The method returns the type of a variable by the index `var_idx` (`CV_VAR_ORDERED` or `CV_VAR_CATEGORICAL`).

CvMLData::change_var_type

Changes type of the specified variable

C++: void CvMLData::`change_var_type`(int **var_idx**, int **type**)

The method changes type of variable with index `var_idx` from existing type to `type` (`CV_VAR_ORDERED` or `CV_VAR_CATEGORICAL`).

CvMLData::set_delimiter

Sets the delimiter in the file used to separate input numbers

C++: void CvMLData::`set_delimiter`(char **ch**)

The method sets the delimiter for variables in a file. For example: `','` (default), `';'`, `' '` (space), or other characters. The floating-point separator `'.'` is not allowed.

CvMLData::get_delimiter

Returns the currently used delimiter character.

C++: char CvMLData::`get_delimiter`() const

CvMLData::set_miss_ch

Sets the character used to specify missing values

C++: void CvMLData::**set_miss_ch**(char **ch**)

The method sets the character used to specify missing values. For example: '?' (default), '-'. The floating-point separator '.' is not allowed.

CvMLData::get_miss_ch

Returns the currently used missing value character.

C++: char CvMLData::**get_miss_ch**() const

CvMLData::get_class_labels_map

Returns a map that converts strings to labels.

C++: const std::map<std::string, int>& CvMLData::**get_class_labels_map**() const

The method returns a map that converts string class labels to the numerical class labels. It can be used to get an original class label as in a file.

CvTrainTestSplit

class CvTrainTestSplit

Structure setting the split of a data set read by `CvMLData`.

```
struct CvTrainTestSplit
{
    CvTrainTestSplit();
    CvTrainTestSplit( int train_sample_count, bool mix = true);
    CvTrainTestSplit( float train_sample_portion, bool mix = true);

    union
    {
        int count;
        float portion;
    } train_sample_part;
    int train_sample_part_mode;

    bool mix;
};
```

There are two ways to construct a split:

- Set the training sample count (subset size) `train_sample_count`. Other existing samples are located in a test subset.

- Set a training sample portion in `[0,..1]`. The flag `mix` is used to mix training and test samples indices when the split is set. Otherwise, the data set is split in the storing order: the first part of samples of a given size is a training subset, the second part is a test subset.

FLANN. CLUSTERING AND SEARCH IN MULTI-DIMENSIONAL SPACES

10.1 Fast Approximate Nearest Neighbor Search

This section documents OpenCV's interface to the 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 [Muja2009] .

flann::Index_

class `flann::`**`Index_`**

The FLANN nearest neighbor index class. This class is templated with the type of elements for which the index is built.

flann::Index_<T>::Index_

Constructs a nearest neighbor search index for a given dataset.

C++: `flann::Index_<T>::`**`Index_`**`(const Mat&` **features**`, const IndexParams&` **params**`)`

> **Parameters**
>
>> **features** – Matrix of containing the features(points) to index. The size of the matrix is `num_features x feature_dimensionality` and the data type of the elements in the matrix must coincide with the type of the index.
>>
>> **params** – Structure containing the index parameters. The type of index that will be constructed depends on the type of this parameter. See the description.

The method constructs a fast search structure from a set of features using the specified algorithm with specified parameters, as defined by `params`. `params` is a reference to one of the following class `IndexParams` descendants:

- **LinearIndexParams** When passing an object of this type, the index will perform a linear, brute-force search.

  ```
  struct LinearIndexParams : public IndexParams
  {
  };
  ```

- **KDTreeIndexParams** When passing an object of this type the index constructed will consist of 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 values are in the range [1..16]

• **KMeansIndexParams** When passing an object of this type the index constructed will be a hierarchical k-means tree.

```
struct KMeansIndexParams : public IndexParams
{
    KMeansIndexParams(
        int branching = 32,
        int iterations = 11,
        flann_centers_init_t centers_init = CENTERS_RANDOM,
        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 in the k-means clustering stage when building the k-means tree. A value of -1 used here means that the k-means clustering should be iterated until convergence

– **centers_init** The algorithm to use for selecting the initial centers when performing a k-means clustering step. The possible values are CENTERS_RANDOM (picks the initial cluster centers randomly), CENTERS_GONZALES (picks the initial centers using Gonzales' algorithm) and CENTERS_KMEANSPP (picks the initial centers using the algorithm suggested in arthur_kmeanspp_2007)

– **cb_index** This parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When cb_index is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

• **CompositeIndexParams** When using a parameters object of this type the index created combines the randomized kd-trees and the hierarchical k-means tree.

```
struct CompositeIndexParams : public IndexParams
{
    CompositeIndexParams(
        int trees = 4,
        int branching = 32,
        int iterations = 11,
        flann_centers_init_t centers_init = CENTERS_RANDOM,
        float cb_index = 0.2 );
};
```

• **LshIndexParams** When using a parameters object of this type the index created uses multi-probe LSH (by Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search by Qin Lv, William Josephson, Zhe Wang, Moses Charikar, Kai Li., Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB). Vienna, Austria. September 2007)

```
struct LshIndexParams : public IndexParams
{
    LshIndexParams(
        unsigned int table_number,
        unsigned int key_size,
        unsigned int multi_probe_level );
};
```

– **table_number** the number of hash tables to use (between 10 and 30 usually).

- **key_size** the size of the hash key in bits (between 10 and 20 usually).

- **multi_probe_level** the number of bits to shift to check for neighboring buckets (0 is regular LSH, 2 is recommended).

- **AutotunedIndexParams** When passing an object of this type the index created is automatically tuned to offer the best performance, by choosing the optimal index type (randomized kd-trees, hierarchical kmeans, linear) and parameters for the dataset provided.

```cpp
struct AutotunedIndexParams : public IndexParams
{
    AutotunedIndexParams(
        float target_precision = 0.9,
        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 percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

- **build_weight** Specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times.

- **memory_weight** Is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

- **sample_fraction** Is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm on the full dataset gives the most accurate results, but for very large datasets can take longer than desired. In such case using just a fraction of the data helps speeding up this algorithm while still giving good approximations of the optimum parameters.

- **SavedIndexParams** This object type is used for loading a previously saved index from the disk.

```cpp
struct SavedIndexParams : public IndexParams
{
    SavedIndexParams( std::string filename );
};
```

- **filename** The filename in which the index was saved.

flann::Index_<T>::knnSearch

Performs a K-nearest neighbor search for a given query point using the index.

C++: void flann::Index_<T>::**knnSearch**(const vector<T>& **query**, vector<int>& **indices**, vector<float>& **dists**, int **knn**, const SearchParams& **params**)

C++: void flann::Index_<T>::**knnSearch**(const Mat& **queries**, Mat& **indices**, Mat& **dists**, int **knn**, const SearchParams& **params**)

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_<T>::radiusSearch

Performs a radius nearest neighbor search for a given query point.

C++: int flann::Index_<T>::**radiusSearch**(const vector<T>& **query**, vector<int>& **indices**, vector<float>& **dists**, float **radius**, const SearchParams& **params**)

C++: int flann::Index_<T>::**radiusSearch**(const Mat& **query**, Mat& **indices**, Mat& **dists**, float **radius**, const SearchParams& **params**)

> **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_<T>::save

Saves the index to a file.

C++: void flann::Index_<T>::**save**(std::string **filename**)

> **Parameters**
>
> > **filename** – The file to save the index to

flann::Index_<T>::getIndexParameters

Returns the index paramreters.

C++: const IndexParams* flann::Index_<T>::**getIndexParameters**()

The method is useful in the case of auto-tuned indices, when the parameters are chosen during the index construction. Then, the method can be used to retrieve the actual parameter values.

10.2 Clustering

flann::hierarchicalClustering<ET,DT>

Clusters features using hierarchical k-means algorithm.

C++: int flann::**hierarchicalClustering<ET, DT>**(const Mat& **features**, Mat& **centers**, const KMeansIndexParams& **params**)

> **Parameters**
>
> > **features** – The points to be clustered. The matrix must have elements of type ET.
> >
> > **centers** – The centers of the clusters obtained. The matrix must have type DT. 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 method clusters the given feature vectors by constructing a hierarchical k-means tree and choosing a cut in the tree that minimizes the cluster's variance. It returns the number of clusters found.

GPU. GPU-ACCELERATED COMPUTER VISION

11.1 GPU Module Introduction

General Information

The OpenCV GPU module is a set of classes and functions to utilize GPU computational capabilities. It is implemented using NVIDIA* CUDA* Runtime API and supports only NVIDIA GPUs. The OpenCV GPU module includes utility functions, low-level vision primitives, and high-level algorithms. The utility functions and low-level primitives provide a powerful infrastructure for developing fast vision algorithms taking advantage of GPU whereas the high-level functionality includes some state-of-the-art algorithms (such as stereo correspondence, face and people detectors, and others) ready to be used by the application developers.

The GPU module is designed as a host-level API. This means that if you have pre-compiled OpenCV GPU binaries, you are not required to have the CUDA Toolkit installed or write any extra code to make use of the GPU.

The GPU module depends on the CUDA Toolkit and NVIDIA Performance Primitives library (NPP). Make sure you have the latest versions of this software installed. You can download two libraries for all supported platforms from the NVIDIA site. To compile the OpenCV GPU module, you need a compiler compatible with the CUDA Runtime Toolkit.

The OpenCV GPU module is designed for ease of use and does not require any knowledge of CUDA. Though, such a knowledge will certainly be useful to handle non-trivial cases or achieve the highest performance. It is helpful to understand the cost of various operations, what the GPU does, what the preferred data formats are, and so on. The GPU module is an effective instrument for quick implementation of GPU-accelerated computer vision algorithms. However, if your algorithm involves many simple operations, then, for the best possible performance, you may still need to write your own kernels to avoid extra write and read operations on the intermediate results.

To enable CUDA support, configure OpenCV using `CMake` with `WITH_CUDA=ON` . When the flag is set and if CUDA is installed, the full-featured OpenCV GPU module is built. Otherwise, the module is still built but at runtime all functions from the module throw `Exception` with `CV_GpuNotSupported` error code, except for `gpu::getCudaEnabledDeviceCount()`. The latter function returns zero GPU count in this case. Building OpenCV without CUDA support does not perform device code compilation, so it does not require the CUDA Toolkit installed. Therefore, using the `gpu::getCudaEnabledDeviceCount()` function, you can implement a high-level algorithm that will detect GPU presence at runtime and choose an appropriate implementation (CPU or GPU) accordingly.

Compilation for Different NVIDIA* Platforms

NVIDIA* compiler enables generating binary code (cubin and fatbin) and intermediate code (PTX). Binary code often implies a specific GPU architecture and generation, so the compatibility with other GPUs is not guaranteed. PTX is targeted for a virtual platform that is defined entirely by the set of capabilities or features. Depending on the selected virtual platform, some of the instructions are emulated or disabled, even if the real hardware supports all the features.

At the first call, the PTX code is compiled to binary code for the particular GPU using a JIT compiler. When the target GPU has a compute capability (CC) lower than the PTX code, JIT fails. By default, the OpenCV GPU module includes:

- Binaries for compute capabilities 1.3 and 2.0 (controlled by `CUDA_ARCH_BIN` in `CMake`)

- PTX code for compute capabilities 1.1 and 1.3 (controlled by `CUDA_ARCH_PTX` in `CMake`)

This means that for devices with CC 1.3 and 2.0 binary images are ready to run. For all newer platforms, the PTX code for 1.3 is JIT'ed to a binary image. For devices with CC 1.1 and 1.2, the PTX for 1.1 is JIT'ed. For devices with CC 1.0, no code is available and the functions throw `Exception()`. For platforms where JIT compilation is performed first, the run is slow.

On a GPU with CC 1.0, you can still compile the GPU module and most of the functions will run flawlessly. To achieve this, add "1.0" to the list of binaries, for example, `CUDA_ARCH_BIN="1.0 1.3 2.0"` . The functions that cannot be run on CC 1.0 GPUs throw an exception.

You can always determine at runtime whether the OpenCV GPU-built binaries (or PTX code) are compatible with your GPU. The function `gpu::DeviceInfo::isCompatible()` returns the compatibility status (true/false).

Threading and Multi-threading

The OpenCV GPU module follows the CUDA Runtime API conventions regarding the multi-threaded programming. This means that for the first API call a CUDA context is created implicitly, attached to the current CPU thread and then is used as the "current" context of the thread. All further operations, such as a memory allocation, GPU code compilation, are associated with the context and the thread. Since any other thread is not attached to the context, memory (and other resources) allocated in the first thread cannot be accessed by another thread. Instead, for this other thread CUDA creates another context associated with it. In short, by default, different threads do not share resources. But you can remove this limitation by using the CUDA Driver API (version 3.1 or later). You can retrieve context reference for one thread, attach it to another thread, and make it "current" for that thread. As a result, the threads can share memory and other resources. It is also possible to create a context explicitly before calling any GPU code and attach it to all the threads you want to share the resources with.

It is also possible to create the context explicitly using the CUDA Driver API, attach, and set the "current" context for all necessary threads. The CUDA Runtime API (and OpenCV functions, respectively) picks it up.

Utilizing Multiple GPUs

In the current version, each of the OpenCV GPU algorithms can use only a single GPU. So, to utilize multiple GPUs, you have to manually distribute the work between GPUs. Consider the following ways of utilizing multiple GPUs:

- If you use only synchronous functions, create several CPU threads (one per each GPU). From within each thread, create a CUDA context for the corresponding GPU using `gpu::setDevice()` or Driver API. Each of the threads will use the associated GPU.

- If you use asynchronous functions, you can use the Driver API to create several CUDA contexts associated with different GPUs but attached to one CPU thread. Within the thread you can switch from one GPU to another by making the corresponding context "current". With non-blocking GPU calls, managing algorithm is clear.

While developing algorithms for multiple GPUs, note a data passing overhead. For primitive functions and small images, it can be significant, which may eliminate all the advantages of having multiple GPUs. But for high-level algorithms, consider using multi-GPU acceleration. For example, the Stereo Block Matching algorithm has been successfully parallelized using the following algorithm:

1. Split each image of the stereo pair into two horizontal overlapping stripes.

2. Process each pair of stripes (from the left and right images) on a separate Fermi* GPU.

3. Merge the results into a single disparity map.

With this algorithm, a dual GPU gave a 180 % performance increase comparing to the single Fermi GPU. For a source code example, see https://code.ros.org/svn/opencv/trunk/opencv/samples/gpu/.

11.2 Initalization and Information

gpu::getCudaEnabledDeviceCount

Returns the number of installed CUDA-enabled devices.

C++: int gpu::**getCudaEnabledDeviceCount**()

Use this function before any other GPU functions calls. If OpenCV is compiled without GPU support, this function returns 0.

gpu::setDevice

Sets a device and initializes it for the current thread.

C++: void gpu::**setDevice**(int **device**)

> **Parameters**

> > **device** – System index of a GPU device starting with 0.

If the call of this function is omitted, a default device is initialized at the fist GPU usage.

gpu::getDevice

Returns the current device index set by gpu::setDevice() or initialized by default.

C++: int gpu::**getDevice**()

gpu::resetDevice

Explicitly destroys and cleans up all resources associated with the current device in the current process.

C++: void gpu::**resetDevice**()

Any subsequent API call to this device will reinitialize the device.

gpu::FeatureSet

class gpu::**FeatureSet**

Class providing GPU computing features.

```
enum FeatureSet
{
    FEATURE_SET_COMPUTE_10,
    FEATURE_SET_COMPUTE_11,
    FEATURE_SET_COMPUTE_12,
    FEATURE_SET_COMPUTE_13,
    FEATURE_SET_COMPUTE_20,
    FEATURE_SET_COMPUTE_21,
    GLOBAL_ATOMICS,
    SHARED_ATOMICS,
    NATIVE_DOUBLE
};
```

gpu::TargetArchs

class gpu::**TargetArchs**

Class providing a set of static methods to check what NVIDIA* card architecture the GPU module was built for.

The following method checks whether the module was built with the support of the given feature:

> **C++: static** bool gpu::TargetArchs::**builtWith**(FeatureSet **feature**)

>> **Parameters**

>>> **feature** – Feature to be checked. See gpu::FeatureSet.

There is a set of methods to check whether the module contains intermediate (PTX) or binary GPU code for the given architecture(s):

> **C++: static** bool gpu::TargetArchs::**has**(int **major**, int **minor**)

> **C++: static** bool gpu::TargetArchs::**hasPtx**(int **major**, int **minor**)

> **C++: static** bool gpu::TargetArchs::**hasBin**(int **major**, int **minor**)

> **C++: static** bool gpu::TargetArchs::**hasEqualOrLessPtx**(int **major**, int **minor**)

> **C++: static** bool gpu::TargetArchs::**hasEqualOrGreater**(int **major**, int **minor**)

> **C++: static** bool gpu::TargetArchs::**hasEqualOrGreaterPtx**(int **major**, int **minor**)

> **C++: static** bool gpu::TargetArchs::**hasEqualOrGreaterBin**(int **major**, int **minor**)

>> **Parameters**

>>> **major** – Major compute capability version.

>>> **minor** – Minor compute capability version.

According to the CUDA C Programming Guide Version 3.2: "PTX code produced for some specific compute capability can always be compiled to binary code of greater or equal compute capability".

gpu::DeviceInfo

class gpu::**DeviceInfo**

Class providing functionality for querying the specified GPU properties.

```cpp
class CV_EXPORTS DeviceInfo
{
public:
    DeviceInfo();
    DeviceInfo(int device_id);

    string name() const;

    int majorVersion() const;
    int minorVersion() const;

    int multiProcessorCount() const;

    size_t freeMemory() const;
    size_t totalMemory() const;

    bool supports(FeatureSet feature) const;
    bool isCompatible() const;

    int deviceID() const;
};
```

gpu::DeviceInfo::DeviceInfo

The constructors.

C++: gpu::DeviceInfo::**DeviceInfo**()

C++: gpu::DeviceInfo::**DeviceInfo**(int **device_id**)

> Parameters

> > **device_id** – System index of the GPU device starting with 0.

Constructs the DeviceInfo object for the specified device. If device_id parameter is missed, it constructs an object for the current device.

gpu::DeviceInfo::name

Returns the device name.

C++: string gpu::DeviceInfo::**name**()

gpu::DeviceInfo::majorVersion

Returns the major compute capability version.

C++: int gpu::DeviceInfo::**majorVersion**()

gpu::DeviceInfo::minorVersion

Returns the minor compute capability version.

C++: int gpu::DeviceInfo::**minorVersion**()

gpu::DeviceInfo::multiProcessorCount

Returns the number of streaming multiprocessors.

C++: int gpu::DeviceInfo::**multiProcessorCount**()

gpu::DeviceInfo::freeMemory

Returns the amount of free memory in bytes.

C++: size_t gpu::DeviceInfo::**freeMemory**()

gpu::DeviceInfo::totalMemory

Returns the amount of total memory in bytes.

C++: size_t gpu::DeviceInfo::**totalMemory**()

gpu::DeviceInfo::supports

Provides information on GPU feature support.

C++: bool gpu::DeviceInfo::**supports**(FeatureSet **feature**)

>   **Parameters**

>   **feature** – Feature to be checked. See `gpu::FeatureSet`.

This function returns `true` if the device has the specified GPU feature. Otherwise, it returns `false` .

gpu::DeviceInfo::isCompatible

Checks the GPU module and device compatibility.

C++: bool gpu::DeviceInfo::**isCompatible**()

This function returns `true` if the GPU module can be run on the specified device. Otherwise, it returns `false` .

gpu::DeviceInfo::deviceID

Returns system index of the GPU device starting with 0.

C++: int gpu::DeviceInfo::**deviceID**()

11.3 Data Structures

gpu::DevMem2D_

class gpu::**DevMem2D_**

Lightweight class encapsulating pitched memory on a GPU and passed to nvcc-compiled code (CUDA kernels). Typically, it is used internally by OpenCV and by users who write device code. You can call its members from both host and device code.

```
template <typename T> struct DevMem2D_
{
    int cols;
    int rows;
    T* data;
    size_t step;

    DevMem2D_() : cols(0), rows(0), data(0), step(0){};
    DevMem2D_(int rows, int cols, T *data, size_t step);

    template <typename U>
    explicit DevMem2D_(const DevMem2D_<U>& d);

    typedef T elem_type;
    enum { elem_size = sizeof(elem_type) };

    __CV_GPU_HOST_DEVICE__ size_t elemSize() const;

    /* returns pointer to the beginning of the given image row */
    __CV_GPU_HOST_DEVICE__ T* ptr(int y = 0);
    __CV_GPU_HOST_DEVICE__ const T* ptr(int y = 0) const;
};

typedef DevMem2D_<unsigned char> DevMem2D;
typedef DevMem2D_<float> DevMem2Df;
typedef DevMem2D_<int> DevMem2Di;
```

gpu::PtrStep_

class gpu::**PtrStep_**

Structure similar to gpu::DevMem2D_ but containing only a pointer and row step. Width and height fields are excluded due to performance reasons. The structure is intended for internal use or for users who write device code.

```
template<typename T> struct PtrStep_
{
        T* data;
        size_t step;

        PtrStep_();
        PtrStep_(const DevMem2D_<T>& mem);

        typedef T elem_type;
        enum { elem_size = sizeof(elem_type) };

        __CV_GPU_HOST_DEVICE__ size_t elemSize() const;
```

```
        __CV_GPU_HOST_DEVICE__ T* ptr(int y = 0);
        __CV_GPU_HOST_DEVICE__ const T* ptr(int y = 0) const;
};


typedef PtrStep_<unsigned char> PtrStep;
typedef PtrStep_<float> PtrStepf;
typedef PtrStep_<int> PtrStepi;
```

gpu::PtrElemStep_

class gpu::**PtrElemStep_**

Structure similar to gpu::DevMem2D_ but containing only a pointer and a row step in elements. Width and height fields are excluded due to performance reasons. This class can only be constructed if sizeof(T) is a multiple of 256. The structure is intended for internal use or for users who write device code.

```
template<typename T> struct PtrElemStep_ : public PtrStep_<T>
{
        PtrElemStep_(const DevMem2D_<T>& mem);
        __CV_GPU_HOST_DEVICE__ T* ptr(int y = 0);
        __CV_GPU_HOST_DEVICE__ const T* ptr(int y = 0) const;
};
```

gpu::GpuMat

class gpu::**GpuMat**

Base storage class for GPU memory with reference counting. Its interface matches the Mat interface with the following limitations:

  • no arbitrary dimensions support (only 2D)

  • no functions that return references to their data (because references on GPU are not valid for CPU)

  • no expression templates technique support

Beware that the latter limitation may lead to overloaded matrix operators that cause memory allocations. The GpuMat class is convertible to gpu::DevMem2D_ and gpu::PtrStep_ so it can be passed directly to the kernel.

Note: In contrast with Mat, in most cases GpuMat::isContinuous() == false . This means that rows are aligned to a size depending on the hardware. Single-row GpuMat is always a continuous matrix.

```
class CV_EXPORTS GpuMat
{
public:
        //! default constructor
        GpuMat();

        GpuMat(int rows, int cols, int type);
        GpuMat(Size size, int type);


        .....


        //! builds GpuMat from Mat. Blocks uploading to device.
        explicit GpuMat (const Mat& m);
```

```
        //! returns lightweight DevMem2D_ structure for passing
        //to nvcc-compiled code. Contains size, data ptr and step.
        template <class T> operator DevMem2D_<T>() const;
        template <class T> operator PtrStep_<T>() const;

        //! blocks uploading data to GpuMat.
        void upload(const cv::Mat& m);
        void upload(const CudaMem& m, Stream& stream);

        //! downloads data from device to host memory. Blocking calls.
        void download(cv::Mat& m) const;

        //! download async
        void download(CudaMem& m, Stream& stream) const;
};
```

Note: You are not recommended to leave static or global `GpuMat` variables allocated, that is, to rely on its destructor. The destruction order of such variables and CUDA context is undefined. GPU memory release function returns error if the CUDA context has been destroyed before.

See Also:

`Mat`

gpu::createContinuous

Creates a continuous matrix in the GPU memory.

C++: void gpu::**createContinuous**(int **rows**, int **cols**, int **type**, GpuMat& **m**)

C++: GpuMat gpu::**createContinuous**(int **rows**, int **cols**, int **type**)

C++: void gpu::**createContinuous**(Size **size**, int **type**, GpuMat& **m**)

C++: GpuMat gpu::**createContinuous**(Size **size**, int **type**)

> **Parameters**
>
> > **rows** – Row count.
> >
> > **cols** – Column count.
> >
> > **type** – Type of the matrix.
> >
> > **m** – Destination matrix. This parameter changes only if it has a proper type and area (rows $\times$ cols).

Matrix is called continuous if its elements are stored continuously, that is, without gaps at the end of each row.

gpu::ensureSizeIsEnough

Ensures that the size of a matrix is big enough and the matrix has a proper type.

C++: void gpu::**ensureSizeIsEnough**(int **rows**, int **cols**, int **type**, GpuMat& **m**)

C++: void gpu::**ensureSizeIsEnough**(Size **size**, int **type**, GpuMat& **m**)

> **Parameters**
>
> > **rows** – Minimum desired number of rows.

> **cols** – Minimum desired number of columns.
>
> **size** – Rows and coumns passed as a structure.
>
> **type** – Desired matrix type.
>
> **m** – Destination matrix.

The function does not reallocate memory if the matrix has proper attributes already.

gpu::registerPageLocked

Page-locks the memory of matrix and maps it for the device(s).

C++: void gpu::`registerPageLocked`(Mat& **m**)

> **Parameters**
>
> > **m** – Input matrix.

gpu::unregisterPageLocked

Unmaps the memory of matrix and makes it pageable again.

C++: void gpu::`unregisterPageLocked`(Mat& **m**)

> **Parameters**
>
> > **m** – Input matrix.

gpu::CudaMem

class gpu::`CudaMem`

Class with reference counting wrapping special memory type allocation functions from CUDA. Its interface is also `Mat()`-like but with additional memory type parameters.

- **ALLOC_PAGE_LOCKED** sets a page locked memory type used commonly for fast and asynchronous uploading/downloading data from/to GPU.

- **ALLOC_ZEROCOPY** specifies a zero copy memory allocation that enables mapping the host memory to GPU address space, if supported.

- **ALLOC_WRITE_COMBINED** sets the write combined buffer that is not cached by CPU. Such buffers are used to supply GPU with data when GPU only reads it. The advantage is a better CPU cache utilization.

Note: Allocation size of such memory types is usually limited. For more details, see *CUDA 2.2 Pinned Memory APIs* document or *CUDA C Programming Guide*.

```
class CV_EXPORTS CudaMem
{
public:
        enum  { ALLOC_PAGE_LOCKED = 1, ALLOC_ZEROCOPY = 2,
                ALLOC_WRITE_COMBINED = 4 };

        CudaMem(Size size, int type, int alloc_type = ALLOC_PAGE_LOCKED);

        //! creates from cv::Mat with coping data
```

```
        explicit CudaMem(const Mat& m, int alloc_type = ALLOC_PAGE_LOCKED);

        ......

        void create(Size size, int type, int alloc_type = ALLOC_PAGE_LOCKED);

        //! returns matrix header with disabled ref. counting for CudaMem data.
        Mat createMatHeader() const;
        operator Mat() const;

        //! maps host memory into device address space
        GpuMat createGpuMatHeader() const;
        operator GpuMat() const;

        //if host memory can be mapped to gpu address space;
        static bool canMapHostMemory();

        int alloc_type;
};
```

gpu::CudaMem::createMatHeader

Creates a header without reference counting to gpu::CudaMem data.

C++: Mat gpu::CudaMem::**createMatHeader**() const

gpu::CudaMem::createGpuMatHeader

Maps CPU memory to GPU address space and creates the gpu::GpuMat header without reference counting for it.

C++: GpuMat gpu::CudaMem::**createGpuMatHeader**() const

This can be done only if memory was allocated with the ALLOC_ZEROCOPY flag and if it is supported by the hardware. Laptops often share video and CPU memory, so address spaces can be mapped, which eliminates an extra copy.

gpu::CudaMem::canMapHostMemory

Returns true if the current hardware supports address space mapping and ALLOC_ZEROCOPY memory allocation.

C++: **static** bool gpu::CudaMem::**canMapHostMemory**()

gpu::Stream

class gpu::**Stream**

This class encapsulates a queue of asynchronous calls. Some functions have overloads with the additional gpu::Stream parameter. The overloads do initialization work (allocate output buffers, upload constants, and so on), start the GPU kernel, and return before results are ready. You can check whether all operations are complete via gpu::Stream::queryIfComplete(). You can asynchronously upload/download data from/to page-locked buffers, using the gpu::CudaMem or Mat header that points to a region of gpu::CudaMem.

Note: Currently, you may face problems if an operation is enqueued twice with different data. Some functions use the constant GPU memory, and next call may update the memory before the previous one has been finished.

But calling different operations asynchronously is safe because each operation has its own constant buffer. Memory copy/upload/download/set operations to the buffers you hold are also safe.

```
class CV_EXPORTS Stream
{
public:
        Stream();
        ~Stream();

        Stream(const Stream&);
        Stream& operator=(const Stream&);

        bool queryIfComplete();
        void waitForCompletion();

        //! downloads asynchronously.
        // Warning! cv::Mat must point to page locked memory
                (i.e. to CudaMem data or to its subMat)
        void enqueueDownload(const GpuMat& src, CudaMem& dst);
        void enqueueDownload(const GpuMat& src, Mat& dst);

        //! uploads asynchronously.
        // Warning! cv::Mat must point to page locked memory
                (i.e. to CudaMem data or to its ROI)
        void enqueueUpload(const CudaMem& src, GpuMat& dst);
        void enqueueUpload(const Mat& src, GpuMat& dst);

        void enqueueCopy(const GpuMat& src, GpuMat& dst);

        void enqueueMemSet(const GpuMat& src, Scalar val);
        void enqueueMemSet(const GpuMat& src, Scalar val, const GpuMat& mask);

        // converts matrix type, ex from float to uchar depending on type
        void enqueueConvert(const GpuMat& src, GpuMat& dst, int type,
                double a = 1, double b = 0);
};
```

gpu::Stream::queryIfComplete

Returns `true` if the current stream queue is finished. Otherwise, it returns false.

C++: bool gpu::Stream::**queryIfComplete**()

gpu::Stream::waitForCompletion

Blocks the current CPU thread until all operations in the stream are complete.

C++: void gpu::Stream::**waitForCompletion**()

gpu::StreamAccessor

class gpu::**StreamAccessor**

Class that enables getting cudaStream_t from gpu::Stream and is declared in stream_accessor.hpp because it is the only public header that depends on the CUDA Runtime API. Including it brings a dependency to your code.

```
struct StreamAccessor
{
    CV_EXPORTS static cudaStream_t getStream(const Stream& stream);
};
```

11.4 Operations on Matrices

gpu::gemm

Performs generalized matrix multiplication.

C++: void **gemm**(const GpuMat& **src1**, const GpuMat& **src2**, double **alpha**, const GpuMat& **src3**, double **beta**, GpuMat& **dst**, int **flags**=0, Stream& **stream**=Stream::Null())

>   **Parameters**

>>   **src1** – First multiplied input matrix that should have `CV_32FC1` , `CV_64FC1` , `CV_32FC2` , or `CV_64FC2` type.

>>   **src2** – Second multiplied input matrix of the same type as `src1` .

>>   **alpha** – Weight of the matrix product.

>>   **src3** – Third optional delta matrix added to the matrix product. It should have the same type as `src1` and `src2` .

>>   **beta** – Weight of `src3` .

>>   **dst** – Destination matrix. It has 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`

>>   **stream** – Stream for the asynchronous version.

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

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

See Also:

gemm()

gpu::transpose

Transposes a matrix.

C++: void gpu::**transpose**(const GpuMat& **src**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

>   **Parameters**

>>   **src** – Source matrix. 1-, 4-, 8-byte element sizes are supported for now (CV_8UC1, CV_8UC4, CV_16UC2, CV_32FC1, etc).

> **dst** – Destination matrix.
>
> **stream** – Stream for the asynchronous version.

See Also:

transpose()

gpu::flip

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

C++: void gpu::**flip**(const GpuMat& **src**, GpuMat& **dst**, int **flipCode**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source matrix. Only `CV_8UC1` and `CV_8UC4` matrices are supported for now.
> >
> > **dst** – Destination matrix.
> >
> > **flipCode** – Flip mode for the source:
> >
> > – `0` Flips around x-axis.
> >
> > – `>0` Flips around y-axis.
> >
> > – `<0` Flips around both axes.
> >
> > **stream** – Stream for the asynchronous version.

See Also:

flip()

gpu::LUT

Transforms the source matrix into the destination matrix using the given look-up table: `dst(I) = lut(src(I))`

C++: void gpu::**LUT**(const GpuMat& **src**, const Mat& **lut**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source matrix. `CV_8UC1` and `CV_8UC3` matrices are supported for now.
> >
> > **lut** – Look-up table of 256 elements. It is a continuous `CV_8U` matrix.
> >
> > **dst** – Destination matrix with the same depth as `lut` and the same number of channels as `src`.
> >
> > **stream** – Stream for the asynchronous version.

See Also:

LUT()

gpu::merge

Makes a multi-channel matrix out of several single-channel matrices.

C++: void gpu::**merge**(const GpuMat* **src**, size_t **n**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

C++: void gpu::**merge**(const vector<GpuMat>& **src**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**

>   **src** – Array/vector of source matrices.

>   **n** – Number of source matrices.

>   **dst** – Destination matrix.

>   **stream** – Stream for the asynchronous version.

See Also:

merge()

gpu::split

Copies each plane of a multi-channel matrix into an array.

C++: void gpu::**split**(const GpuMat& **src**, GpuMat* **dst**, Stream& **stream**=Stream::Null())

C++: void gpu::**split**(const GpuMat& **src**, vector<GpuMat>& **dst**, Stream& **stream**=Stream::Null())

>   **Parameters**

>   **src** – Source matrix.

>   **dst** – Destination array/vector of single-channel matrices.

>   **stream** – Stream for the asynchronous version.

See Also:

split()

gpu::magnitude

Computes magnitudes of complex matrix elements.

C++: void gpu::**magnitude**(const GpuMat& **xy**, GpuMat& **magnitude**, Stream& **stream**=Stream::Null())

C++: void gpu::**magnitude**(const GpuMat& **x**, const GpuMat& **y**, GpuMat& **magnitude**, Stream& **stream**=Stream::Null())

>   **Parameters**

>   **xy** – Source complex matrix in the interleaved format (CV_32FC2).

>   **x** – Source matrix containing real components (CV_32FC1).

>   **y** – Source matrix containing imaginary components (CV_32FC1).

>   **magnitude** – Destination matrix of float magnitudes (CV_32FC1).

>   **stream** – Stream for the asynchronous version.

See Also:

magnitude()

gpu::magnitudeSqr

Computes squared magnitudes of complex matrix elements.

C++: void gpu::**magnitudeSqr**(const GpuMat& **xy**, GpuMat& **magnitude**, Stream& **stream**=Stream::Null())

C++: void gpu::**magnitudeSqr**(const GpuMat& **x**, const GpuMat& **y**, GpuMat& **magnitude**, Stream& **stream**=Stream::Null())

> Parameters

>> **xy** – Source complex matrix in the interleaved format (`CV_32FC2`).

>> **x** – Source matrix containing real components (`CV_32FC1`).

>> **y** – Source matrix containing imaginary components (`CV_32FC1`).

>> **magnitude** – Destination matrix of float magnitude squares (`CV_32FC1`).

>> **stream** – Stream for the asynchronous version.

gpu::phase

Computes polar angles of complex matrix elements.

C++: void gpu::**phase**(const GpuMat& **x**, const GpuMat& **y**, GpuMat& **angle**, bool **angleInDegrees**=false, Stream& **stream**=Stream::Null())

> Parameters

>> **x** – Source matrix containing real components (`CV_32FC1`).

>> **y** – Source matrix containing imaginary components (`CV_32FC1`).

>> **angle** – Destionation matrix of angles (`CV_32FC1`).

>> **angleInDegress** – Flag for angles that must be evaluated in degress.

>> **stream** – Stream for the asynchronous version.

See Also:

phase()

gpu::cartToPolar

Converts Cartesian coordinates into polar.

C++: void gpu::**cartToPolar**(const GpuMat& **x**, const GpuMat& **y**, GpuMat& **magnitude**, GpuMat& **angle**, bool **angleInDegrees**=false, Stream& **stream**=Stream::Null())

> Parameters

>> **x** – Source matrix containing real components (`CV_32FC1`).

>> **y** – Source matrix containing imaginary components (`CV_32FC1`).

>> **magnitude** – Destination matrix of float magnitudes (`CV_32FC1`).

>> **angle** – Destionation matrix of angles (`CV_32FC1`).

>> **angleInDegress** – Flag for angles that must be evaluated in degress.

>> **stream** – Stream for the asynchronous version.

See Also:

cartToPolar()

gpu::polarToCart

Converts polar coordinates into Cartesian.

C++: void gpu::**polarToCart**(const GpuMat& **magnitude**, const GpuMat& **angle**, GpuMat& **x**, GpuMat& **y**, bool **angleInDegrees**=false, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **magnitude** – Source matrix containing magnitudes (`CV_32FC1`).
>>
>> **angle** – Source matrix containing angles (`CV_32FC1`).
>>
>> **x** – Destination matrix of real components (`CV_32FC1`).
>>
>> **y** – Destination matrix of imaginary components (`CV_32FC1`).
>>
>> **angleInDegress** – Flag that indicates angles in degress.
>>
>> **stream** – Stream for the asynchronous version.

See Also:

`polarToCart()`

11.5 Per-element Operations

gpu::add

Computes a matrix-matrix or matrix-scalar sum.

C++: void gpu::**add**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), int **dtype**=-1, Stream& **stream**=Stream::Null())

C++: void gpu::**add**(const GpuMat& **src1**, const Scalar& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), int **dtype**=-1, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src1** – First source matrix.
>>
>> **src2** – Second source matrix or a scalar to be added to `src1` . Matrix should have the same size and type as `src1` .
>>
>> **dst** – Destination matrix that has the same size and number of channels as the input array(s). The depth is defined by `dtype` or `src1` depth.
>>
>> **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.
>>
>> **dtype** – Optional depth of the output array.
>>
>> **stream** – Stream for the asynchronous version.

See Also:

`add()`

gpu::subtract

Computes a matrix-matrix or matrix-scalar difference.

C++: void gpu::**subtract**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), int **dtype**=-1, Stream& **stream**=Stream::Null())

C++: void gpu::**subtract**(const GpuMat& **src1**, const Scalar& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), int **dtype**=-1, Stream& **stream**=Stream::Null())

> Parameters

>> **src1** – First source matrix.

>> **src2** – Second source matrix or a scalar to be added to `src1` . Matrix should have the same size and type as `src1` .

>> **dst** – Destination matrix that has the same size and number of channels as the input array(s). The depth is defined by `dtype` or `src1` depth.

>> **mask** – Optional operation mask, 8-bit single channel array, that specifies elements of the destination array to be changed.

>> **dtype** – Optional depth of the output array.

>> **stream** – Stream for the asynchronous version.

See Also:

subtract()

gpu::multiply

Computes a matrix-matrix or matrix-scalar per-element product.

C++: void gpu::**multiply**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, double **scale**=1, int **dtype**=-1, Stream& **stream**=Stream::Null())

C++: void gpu::**multiply**(const GpuMat& **src1**, const Scalar& **src2**, GpuMat& **dst**, double **scale**=1, int **dtype**=-1, Stream& **stream**=Stream::Null())

> Parameters

>> **src1** – First source matrix.

>> **src2** – Second source matrix or a scalar to be multiplied by `src1` elements.

>> **dst** – Destination matrix that has the same size and number of channels as the input array(s). The depth is defined by `dtype` or `src1` depth.

>> **scale** – Optional scale factor.

>> **dtype** – Optional depth of the output array.

>> **stream** – Stream for the asynchronous version.

See Also:

multiply()

gpu::divide

Computes a matrix-matrix or matrix-scalar division.

C++: void gpu::**divide**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, double **scale**=1, int **dtype**=-1, Stream& **stream**=Stream::Null())

C++: void gpu::**divide**(const GpuMat& **src1**, const Scalar& **src2**, GpuMat& **dst**, double **scale**=1, int **dtype**=-1, Stream& **stream**=Stream::Null())

C++: void gpu::**divide**(double **src1**, const GpuMat& **src2**, GpuMat& **dst**, int **dtype**=-1, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src1** – First source matrix or a scalar.
> >
> > **src2** – Second source matrix or a scalar. The `src1` elements are divided by it.
> >
> > **dst** – Destination matrix that has the same size and number of channels as the input array(s). The depth is defined by `dtype` or `src1` depth.
> >
> > **scale** – Optional scale factor.
> >
> > **dtype** – Optional depth of the output array.
> >
> > **stream** – Stream for the asynchronous version.

This function, in contrast to `divide()`, uses a round-down rounding mode.

See Also:

`divide()`

addWeighted

Computes the weighted sum of two arrays.

void gpu::addWeighted(const GpuMat& src1, double alpha, const GpuMat& src2, double beta, double gamma, Gpu

> **Parameters**
>
> > **src1** – First source array.
> >
> > **alpha** – Weight for the first array elements.
> >
> > **src2** – Second source array of the same size and channel number as `src1` .
> >
> > **beta** – Weight for the second array elements.
> >
> > **dst** – Destination array that has the same size and number of channels as the input arrays.
> >
> > **gamma** – Scalar added to each sum.
> >
> > **dtype** – Optional depth of the destination array. When both input arrays have the same depth, `dtype` can be set to `-1`, which will be equivalent to `src1.depth()`.
> >
> > **stream** – Stream for the asynchronous version.

The function `addWeighted` calculates 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 a multi-dimensional index of array elements. In case of multi-channel arrays, each channel is processed independently.

See Also:

addWeighted()

gpu::exp

Computes an exponent of each matrix element.

C++: void gpu::**exp**(const GpuMat& **src**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source matrix. CV_32FC1 matrixes are supported for now.
>>
>> **dst** – Destination matrix with the same size and type as src .
>>
>> **stream** – Stream for the asynchronous version.

See Also:

exp()

gpu::pow

Raises every matrix element to a power.

C++: void gpu::**pow**(const GpuMat& **src**, double **power**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source matrix. Supports all type, except CV_64F depth.
>>
>> **power** – Exponent of power.
>>
>> **dst** – Destination matrix with the same size and type as src .
>>
>> **stream** – Stream for the asynchronous version.

The function pow raises every element of the input matrix to p :

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

See Also:

pow()

gpu::log

Computes a natural logarithm of absolute value of each matrix element.

C++: void gpu::**log**(const GpuMat& **src**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source matrix. CV_32FC1 matrixes are supported for now.
>>
>> **dst** – Destination matrix with the same size and type as src .
>>
>> **stream** – Stream for the asynchronous version.

See Also:

log()

gpu::absdiff

Computes per-element absolute difference of two matrices (or of a matrix and scalar).

C++: void gpu::**absdiff**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

C++: void gpu::**absdiff**(const GpuMat& **src1**, const Scalar& **src2**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

>   Parameters

>   **src1** – First source matrix.

>   **src2** – Second source matrix or a scalar to be added to `src1` .

>   **dst** – Destination matrix with the same size and type as `src1` .

>   **stream** – Stream for the asynchronous version.

See Also:

absdiff()

gpu::compare

Compares elements of two matrices.

C++: void gpu::**compare**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, int **cmpop**, Stream& **stream**=Stream::Null())

>   Parameters

>   **src1** – First source matrix.

>   **src2** – Second source matrix with the same size and type as `src1` .

>   **dst** – Destination matrix with the same size as `src1` and the `CV_8UC1` type.

>   **cmpop** – Flag specifying the relation between the elements to be checked:

>   – **CMP_EQ:** `src1(.)  == src2(.)`

>   – **CMP_GT:** `src1(.)  < src2(.)`

>   – **CMP_GE:** `src1(.)  <= src2(.)`

>   – **CMP_LT:** `src1(.)  < src2(.)`

>   – **CMP_LE:** `src1(.)  <= src2(.)`

>   – **CMP_NE:** `src1(.)  != src2(.)`

>   **stream** – Stream for the asynchronous version.

See Also:

compare()

gpu::bitwise_not

Performs a per-element bitwise inversion.

C++: void gpu::**bitwise_not**(const GpuMat& **src**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source matrix.
> >
> > **dst** – Destination matrix with the same size and type as src .
> >
> > **mask** – Optional operation mask. 8-bit single channel image.
> >
> > **stream** – Stream for the asynchronous version.

gpu::bitwise_or

Performs a per-element bitwise disjunction of two matrices.

C++: void gpu::**bitwise_or**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src1** – First source matrix.
> >
> > **src2** – Second source matrix with the same size and type as src1 .
> >
> > **dst** – Destination matrix with the same size and type as src1 .
> >
> > **mask** – Optional operation mask. 8-bit single channel image.
> >
> > **stream** – Stream for the asynchronous version.

gpu::bitwise_and

Performs a per-element bitwise conjunction of two matrices.

C++: void gpu::**bitwise_and**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src1** – First source matrix.
> >
> > **src2** – Second source matrix with the same size and type as src1 .
> >
> > **dst** – Destination matrix with the same size and type as src1 .
> >
> > **mask** – Optional operation mask. 8-bit single channel image.
> >
> > **stream** – Stream for the asynchronous version.

gpu::bitwise_xor

Performs a per-element bitwise exclusive or operation of two matrices.

C++: void gpu::**bitwise_xor**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, const GpuMat& **mask**=GpuMat(), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src1** – First source matrix.
> >
> > **src2** – Second source matrix with the same size and type as src1 .
> >
> > **dst** – Destination matrix with the same size and type as src1 .
> >
> > **mask** – Optional operation mask. 8-bit single channel image.

> **stream** – Stream for the asynchronous version.

gpu::min

Computes the per-element minimum of two matrices (or a matrix and a scalar).

C++: void gpu::**min**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

C++: void gpu::**min**(const GpuMat& **src1**, double **src2**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**

>> **src1** – First source matrix.

>> **src2** – Second source matrix or a scalar to compare `src1` elements with.

>> **dst** – Destination matrix with the same size and type as `src1` .

>> **stream** – Stream for the asynchronous version.

See Also:

`min()`

gpu::max

Computes the per-element maximum of two matrices (or a matrix and a scalar).

C++: void gpu::**max**(const GpuMat& **src1**, const GpuMat& **src2**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

C++: void gpu::**max**(const GpuMat& **src1**, double **src2**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

> **Parameters**

>> **src1** – First source matrix.

>> **src2** – Second source matrix or a scalar to compare `src1` elements with.

>> **dst** – Destination matrix with the same size and type as `src1` .

>> **stream** – Stream for the asynchronous version.

See Also:

`max()`

11.6 Image Processing

gpu::meanShiftFiltering

Performs mean-shift filtering for each point of the source image.

C++: void gpu::**meanShiftFiltering**(const GpuMat& **src**, GpuMat& **dst**, int **sp**, int **sr**, TermCriteria **criteria**=TermCriteria(TermCriteria::MAX_ITER + TermCriteria::EPS, 5, 1))

> **Parameters**

>> **src** – Source image. Only `CV_8UC4` images are supported for now.

>> **dst** – Destination image containing the color of mapped points. It has the same size and type as `src` .

>> **sp** – Spatial window radius.

>> **sr** – Color window radius.

>> **criteria** – Termination criteria. See `TermCriteria`.

It maps each point of the source image into another point. As a result, you have a new color and new position of each point.

gpu::meanShiftProc

Performs a mean-shift procedure and stores information about processed points (their colors and positions) in two images.

C++: void gpu::**meanShiftProc**(const GpuMat& **src**, GpuMat& **dstr**, GpuMat& **dstsp**, int **sp**, int **sr**, TermCriteria **criteria**=TermCriteria(TermCriteria::MAX_ITER + TermCriteria::EPS, 5, 1))

> **Parameters**

>> **src** – Source image. Only `CV_8UC4` images are supported for now.

>> **dstr** – Destination image containing the color of mapped points. The size and type is the same as `src` .

>> **dstsp** – Destination image containing the position of mapped points. The size is the same as `src` size. The type is `CV_16SC2` .

>> **sp** – Spatial window radius.

>> **sr** – Color window radius.

>> **criteria** – Termination criteria. See `TermCriteria`.

See Also:

`gpu::meanShiftFiltering()`

gpu::meanShiftSegmentation

Performs a mean-shift segmentation of the source image and eliminates small segments.

C++: void gpu::**meanShiftSegmentation**(const GpuMat& **src**, Mat& **dst**, int **sp**, int **sr**, int **minsize**, TermCriteria **criteria**=TermCriteria(TermCriteria::MAX_ITER + TermCriteria::EPS, 5, 1))

> **Parameters**

>> **src** – Source image. Only `CV_8UC4` images are supported for now.

>> **dst** – Segmented image with the same size and type as `src` .

>> **sp** – Spatial window radius.

>> **sr** – Color window radius.

>> **minsize** – Minimum segment size. Smaller segements are merged.

>> **criteria** – Termination criteria. See `TermCriteria`.

gpu::integral

Computes an integral image and a squared integral image.

C++: void gpu::**integral**(const GpuMat& **src**, GpuMat& **sum**, Stream& **stream**=Stream::Null())

C++: void gpu::**integral**(const GpuMat& **src**, GpuMat& **sum**, GpuMat& **sqsum**, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source image. Only `CV_8UC1` images are supported for now.
>>
>> **sum** – Integral image containing 32-bit unsigned integer values packed into `CV_32SC1` .
>>
>> **sqsum** – Squared integral image of the `CV_32FC1` type.
>>
>> **stream** – Stream for the asynchronous version.

See Also:

integral()

gpu::sqrIntegral

Computes a squared integral image.

C++: void gpu::**sqrIntegral**(const GpuMat& **src**, GpuMat& **sqsum**, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source image. Only `CV_8UC1` images are supported for now.
>>
>> **sqsum** – Squared integral image containing 64-bit unsigned integer values packed into `CV_64FC1` .
>>
>> **stream** – Stream for the asynchronous version.

gpu::columnSum

Computes a vertical (column) sum.

C++: void gpu::**columnSum**(const GpuMat& **src**, GpuMat& **sum**)

> **Parameters**
>
>> **src** – Source image. Only `CV_32FC1` images are supported for now.
>>
>> **sum** – Destination image of the `CV_32FC1` type.

gpu::cornerHarris

Computes the Harris cornerness criteria at each image pixel.

C++: void gpu::**cornerHarris**(const GpuMat& **src**, GpuMat& **dst**, int **blockSize**, int **ksize**, double **k**, int **borderType**=BORDER_REFLECT101)

> **Parameters**
>
>> **src** – Source image. Only `CV_8UC1` and `CV_32FC1` images are supported for now.
>>
>> **dst** – Destination image containing cornerness values. It has the same size as `src` and `CV_32FC1` type.

> **blockSize** – Neighborhood size.
>
> **ksize** – Aperture parameter for the Sobel operator.
>
> **k** – Harris detector free parameter.
>
> **borderType** – Pixel extrapolation method. Only `BORDER_REFLECT101` and `BORDER_REPLICATE` are supported for now.

See Also:

cornerHarris()

gpu::cornerMinEigenVal

Computes the minimum eigen value of a 2x2 derivative covariation matrix at each pixel (the cornerness criteria).

C++: void gpu::**cornerMinEigenVal**(const GpuMat& **src**, GpuMat& **dst**, int **blockSize**, int **ksize**, int **bor-derType**=BORDER_REFLECT101)

>> **Parameters**
>>
>>> **src** – Source image. Only `CV_8UC1` and `CV_32FC1` images are supported for now.
>>>
>>> **dst** – Destination image containing cornerness values. The size is the same. The type is `CV_32FC1` .
>>>
>>> **blockSize** – Neighborhood size.
>>>
>>> **ksize** – Aperture parameter for the Sobel operator.
>>>
>>> **k** – Harris detector free parameter.
>>>
>>> **borderType** – Pixel extrapolation method. Only `BORDER_REFLECT101` and `BORDER_REPLICATE` are supported for now.

See Also:

cornerMinEigenVal()

gpu::mulSpectrums

Performs a per-element multiplication of two Fourier spectrums.

C++: void gpu::**mulSpectrums**(const GpuMat& **a**, const GpuMat& **b**, GpuMat& **c**, int **flags**, bool **conjB**=false)

>> **Parameters**
>>
>>> **a** – First spectrum.
>>>
>>> **b** – Second spectrum with the same size and type as `a` .
>>>
>>> **c** – Destination spectrum.
>>>
>>> **flags** – Mock parameter used for CPU/GPU interfaces similarity.
>>>
>>> **conjB** – Optional flag to specify if the second spectrum needs to be conjugated before the multiplication.
>>
>> Only full (not packed) `CV_32FC2` complex spectrums in the interleaved format are supported for now.

See Also:

mulSpectrums()

gpu::mulAndScaleSpectrums

Performs a per-element multiplication of two Fourier spectrums and scales the result.

C++: void gpu::**mulAndScaleSpectrums**(const GpuMat& **a**, const GpuMat& **b**, GpuMat& **c**, int **flags**, float **scale**, bool **conjB**=false)

>   **Parameters**

>>   **a** – First spectrum.

>>   **b** – Second spectrum with the same size and type as a .

>>   **c** – Destination spectrum.

>>   **flags** – Mock parameter used for CPU/GPU interfaces similarity.

>>   **scale** – Scale constant.

>>   **conjB** – Optional flag to specify if the second spectrum needs to be conjugated before the multiplication.

>   Only full (not packed) CV_32FC2 complex spectrums in the interleaved format are supported for now.

See Also:

mulSpectrums()

gpu::dft

Performs a forward or inverse discrete Fourier transform (1D or 2D) of the floating point matrix.

C++: void gpu::**dft**(const GpuMat& **src**, GpuMat& **dst**, Size **dft_size**, int **flags**=0)

>   **Parameters**

>>   **src** – Source matrix (real or complex).

>>   **dst** – Destination matrix (real or complex).

>>   **dft_size** – Size of a discrete Fourier transform.

>>   **flags** – Optional flags:

>>   – **DFT_ROWS** transforms each individual row of the source matrix.

>>   – **DFT_SCALE** scales the result: divide it by the number of elements in the transform (obtained from dft_size).

>>   – **DFT_INVERSE** inverts DFT. Use for complex-complex cases (real-complex and complex-real cases are always forward and inverse, respectively).

>>   – **DFT_REAL_OUTPUT** specifies the output as real. The source matrix is the result of real-complex transform, so the destination matrix must be real.

Use to handle real matrices (CV32FC1) and complex matrices in the interleaved format (CV32FC2).

The source matrix should be continuous, otherwise reallocation and data copying is performed. The function chooses an operation mode depending on the flags, size, and channel count of the source matrix:

-   If the source matrix is complex and the output is not specified as real, the destination matrix is complex and has the dft_size size and CV_32FC2 type. The destination matrix contains a full result of the DFT (forward or inverse).

- If the source matrix is complex and the output is specified as real, the function assumes that its input is the result of the forward transform (see the next item). The destionation matrix has the `dft_size` size and `CV_32FC1` type. It contains the result of the inverse DFT.

- If the source matrix is real (its type is `CV_32FC1`), forward DFT is performed. The result of the DFT is packed into complex (`CV_32FC2`) matrix. So, the width of the destination matrix is `dft_size.width / 2 + 1`. But if the source is a single column, the height is reduced instead of the width.

See Also:

dft()

gpu::convolve

Computes a convolution (or cross-correlation) of two images.

C++: void gpu::**convolve**(const GpuMat& **image**, const GpuMat& **templ**, GpuMat& **result**, bool **ccorr**=false)

C++: void gpu::**convolve**(const GpuMat& **image**, const GpuMat& **templ**, GpuMat& **result**, bool **ccorr**, ConvolveBuf& **buf**)

        Parameters

            image – Source image. Only `CV_32FC1` images are supported for now.

            templ – Template image. The size is not greater than the `image` size. The type is the same as `image` .

            result – Result image. The size and type is the same as `image` .

            ccorr – Flags to evaluate cross-correlation instead of convolution.

            buf – Optional buffer to avoid extra memory allocations (for many calls with the same sizes).

gpu::ConvolveBuf

class gpu::**ConvolveBuf**

Class providing a memory buffer for the gpu::convolve() function.

```
struct CV_EXPORTS ConvolveBuf
{
    ConvolveBuf() {}
    ConvolveBuf(Size image_size, Size templ_size)
        { create(image_size, templ_size); }
    void create(Size image_size, Size templ_size);

private:
    // Hidden
};
```

gpu::ConvolveBuf::ConvolveBuf

The constructors.

C++: ConvolveBuf::**ConvolveBuf**()

Constructs an empty buffer that is properly resized after the first call of the gpu::convolve() function.

C++: ConvolveBuf::**ConvolveBuf**(Size **image_size**, Size **templ_size**)

Constructs a buffer for the gpu::convolve() function with respective arguments.

gpu::matchTemplate

Computes a proximity map for a raster template and an image where the template is searched for.

C++: void gpu::**matchTemplate**(const GpuMat& **image**, const GpuMat& **templ**, GpuMat& **result**, int **method**)

> **Parameters**
>
> > **image** – Source image. CV_32F and CV_8U depth images (1..4 channels) are supported for now.
> >
> > **templ** – Template image with the size and type the same as image .
> >
> > **result** – Map containing comparison results (CV_32FC1). If image is *W x H* and templ is *w x h*, then result must be *W-w+1 x H-h+1*.
> >
> > **method** – Specifies the way to compare the template with the image.
>
> The following methods are supported for the CV_8U depth images for now:
>
> - CV_TM_SQDIFF
> - CV_TM_SQDIFF_NORMED
> - CV_TM_CCORR
> - CV_TM_CCORR_NORMED
> - CV_TM_CCOEFF
> - CV_TM_CCOEFF_NORMED
>
> The following methods are supported for the CV_32F images for now:
>
> - CV_TM_SQDIFF
> - CV_TM_CCORR

See Also:

matchTemplate()

gpu::remap

Applies a generic geometrical transformation to an image.

void gpu::remap(const GpuMat& src, GpuMat& dst, const GpuMat& xmap, const GpuMat& ymap, int interpolation,

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image with the size the same as xmap and the type the same as src .
> >
> > **xmap** – X values. Only CV_32FC1 type is supported.
> >
> > **ymap** – Y values. Only CV_32FC1 type is supported.
> >
> > **interpolation** – Interpolation method (see resize()). INTER_NEAREST , INTER_LINEAR and INTER_CUBIC are supported for now.

> **borderMode** – Pixel extrapolation method (see `borderInterpolate()`). `BORDER_REFLECT101` , `BORDER_REPLICATE` , `BORDER_CONSTANT` , `BORDER_REFLECT` and `BORDER_WRAP` are supported for now.
>
> **borderValue** – Value used in case of a constant border. By default, it is 0.
>
> **stream** – Stream for the asynchronous version.

The function transforms the source image using the specified map:

$$dst(x, y) = src(xmap(x, y), ymap(x, y))$$

Values of pixels with non-integer coordinates are computed using the bilinear interpolation.

See Also:

`remap()`

gpu::cvtColor

Converts an image from one color space to another.

C++: void gpu::**cvtColor**(const GpuMat& **src**, GpuMat& **dst**, int **code**, int **dcn**=0, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image with `CV_8U` , `CV_16U` , or `CV_32F` depth and 1, 3, or 4 channels.
> >
> > **dst** – Destination image with the same size and depth as `src` .
> >
> > **code** – Color space conversion code. For details, see `cvtColor()` . Conversion to/from Luv and Bayer color spaces is not supported.
> >
> > **dcn** – Number of channels in the destination image. If the parameter is 0, the number of the channels is derived automatically from `src` and the `code` .
> >
> > **stream** – Stream for the asynchronous version.

3-channel color spaces (like `HSV`, `XYZ`, and so on) can be stored in a 4-channel image for better perfomance.

See Also:

`cvtColor()`

gpu::threshold

Applies a fixed-level threshold to each array element.

C++: double gpu::**threshold**(const GpuMat& **src**, GpuMat& **dst**, double **thresh**, double **maxval**, int **type**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source array (single-channel).
> >
> > **dst** – Destination array with the same size and type as `src` .
> >
> > **thresh** – Threshold value.
> >
> > **maxVal** – Maximum value to use with `THRESH_BINARY` and `THRESH_BINARY_INV` threshold types.

> **thresholdType** – Threshold type. For details, see `threshold()`. The THRESH_OTSU threshold type is not supported.
>
> **stream** – Stream for the asynchronous version.

See Also:

`threshold()`

gpu::resize

Resizes an image.

C++: void gpu::**resize**(const GpuMat& **src**, GpuMat& **dst**, Size **dsize**, double **fx**=0, double **fy**=0, int **interpolation**=INTER_LINEAR, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image with the same type as `src`. The size is `dsize` (when it is non-zero) or the size is computed from `src.size()`, `fx`, and `fy`.
> >
> > **dsize** – Destination image size. If it is zero, it is computed as:
> >
> > $$\text{dsize} = \text{Size(round(fx*src.cols), round(fy*src.rows))}$$
> >
> > Either `dsize` or both `fx` and `fy` must be non-zero.
> >
> > **fx** – Scale factor along the horizontal axis. If it is zero, it is computed as:
> >
> > $$\text{(double)dsize.width/src.cols}$$
> >
> > **fy** – Scale factor along the vertical axis. If it is zero, it is computed as:
> >
> > $$\text{(double)dsize.height/src.rows}$$
> >
> > **interpolation** – Interpolation method. INTER_NEAREST , INTER_LINEAR and INTER_CUBIC are supported for now.
> >
> > **stream** – Stream for the asynchronous version.

See Also:

`resize()`

gpu::warpAffine

Applies an affine transformation to an image.

C++: void gpu::**warpAffine**(const GpuMat& **src**, GpuMat& **dst**, const Mat& **M**, Size **dsize**, int **flags**=INTER_LINEAR, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image. CV_8U , CV_16U , CV_32S , or CV_32F depth and 1, 3, or 4 channels are supported.
> >
> > **dst** – Destination image with the same type as `src`. The size is `dsize`.

> **M** – *2x3* transformation matrix.
>
> **dsize** – Size of the destination image.
>
> **flags** – Combination of interpolation methods (see `resize()`) and the optional flag WARP_INVERSE_MAP specifying that M is an inverse transformation (`dst=>src`). Only `INTER_NEAREST` , `INTER_LINEAR` , and `INTER_CUBIC` interpolation methods are supported.
>
> **stream** – Stream for the asynchronous version.

See Also:

`warpAffine()`

gpu::warpPerspective

Applies a perspective transformation to an image.

C++: void gpu::**warpPerspective**(const GpuMat& **src**, GpuMat& **dst**, const Mat& **M**, Size **dsize**, int **flags**=INTER_LINEAR, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source image. `CV_8U` , `CV_16U` , `CV_32S` , or `CV_32F` depth and 1, 3, or 4 channels are supported.
>>
>> **dst** – Destination image with the same type as `src` . The size is `dsize` .
>>
>> **M** – *3x3* transformation matrix.
>>
>> **dsize** – Size of the destination image.
>>
>> **flags** – Combination of interpolation methods (see `resize()`) and the optional flag WARP_INVERSE_MAP specifying that M is the inverse transformation (`dst => src`). Only `INTER_NEAREST` , `INTER_LINEAR` , and `INTER_CUBIC` interpolation methods are supported.
>>
>> **stream** – Stream for the asynchronous version.

See Also:

`warpPerspective()`

gpu::rotate

Rotates an image around the origin (0,0) and then shifts it.

C++: void gpu::**rotate**(const GpuMat& **src**, GpuMat& **dst**, Size **dsize**, double **angle**, double **xShift**=0, double **yShift**=0, int **interpolation**=INTER_LINEAR, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source image. `CV_8UC1` and `CV_8UC4` types are supported.
>>
>> **dst** – Destination image with the same type as `src` . The size is `dsize` .
>>
>> **dsize** – Size of the destination image.
>>
>> **angle** – Angle of rotation in degrees.
>>
>> **xShift** – Shift along the horizontal axis.
>>
>> **yShift** – Shift along the vertical axis.

> **interpolation** – Interpolation method. Only `INTER_NEAREST` , `INTER_LINEAR` , and `INTER_CUBIC` are supported.
>
> **stream** – Stream for the asynchronous version.

See Also:

gpu::warpAffine()

gpu::copyMakeBorder

Forms a border around an image.

C++: void gpu::**copyMakeBorder**(const GpuMat& **src**, GpuMat& **dst**, int **top**, int **bottom**, int **left**, int **right**, int **borderType**, const Scalar& **value**=Scalar(), Stream& **stream**=Stream::Null())

>   **Parameters**
>
> > **src** – Source image. `CV_8UC1` , `CV_8UC4` , `CV_32SC1` , and `CV_32FC1` types are supported.
> >
> > **dst** – Destination image with the same type as `src`. The size is `Size(src.cols+left+right, src.rows+top+bottom)` .
> >
> > **top** –
> >
> > **bottom** –
> >
> > **left** –
> >
> > **right** – Number of pixels in each direction from the source image rectangle to extrapolate. For example: `top=1, bottom=1, left=1, right=1` mean that 1 pixel-wide border needs to be built.
> >
> > **borderType** – Border type. See borderInterpolate() for details. `BORDER_REFLECT101` , `BORDER_REPLICATE` , `BORDER_CONSTANT` , `BORDER_REFLECT` and `BORDER_WRAP` are supported for now.
> >
> > **value** – Border value.
> >
> > **stream** – Stream for the asynchronous version.

See Also:

copyMakeBorder()

gpu::rectStdDev

Computes a standard deviation of integral images.

C++: void gpu::**rectStdDev**(const GpuMat& **src**, const GpuMat& **sqr**, GpuMat& **dst**, const Rect& **rect**, Stream& **stream**=Stream::Null())

>   **Parameters**
>
> > **src** – Source image. Only the `CV_32SC1` type is supported.
> >
> > **sqr** – Squared source image. Only the `CV_32FC1` type is supported.
> >
> > **dst** – Destination image with the same type and size as `src` .
> >
> > **rect** – Rectangular window.
> >
> > **stream** – Stream for the asynchronous version.

gpu::evenLevels

Computes levels with even distribution.

C++: void gpu::**evenLevels**(GpuMat& **levels**, int **nLevels**, int **lowerLevel**, int **upperLevel**)

>   **Parameters**

>>   **levels** – Destination array. `levels` has 1 row, nLevels columns, and the `CV_32SC1` type.

>>   **nLevels** – Number of computed levels. `nLevels` must be at least 2.

>>   **lowerLevel** – Lower boundary value of the lowest level.

>>   **upperLevel** – Upper boundary value of the greatest level.

gpu::histEven

Calculates a histogram with evenly distributed bins.

C++: void gpu::**histEven**(const GpuMat& **src**, GpuMat& **hist**, int **histSize**, int **lowerLevel**, int **upperLevel**, Stream& **stream**=Stream::Null())

C++: void gpu::**histEven**(const GpuMat& **src**, GpuMat& **hist**, GpuMat& **buf**, int **histSize**, int **lowerLevel**, int **upperLevel**, Stream& **stream**=Stream::Null())

C++: void gpu::**histEven**(const GpuMat& **src**, GpuMat* **hist**, int* **histSize**, int* **lowerLevel**, int* **upperLevel**, Stream& **stream**=Stream::Null())

C++: void gpu::**histEven**(const GpuMat& **src**, GpuMat* **hist**, GpuMat& **buf**, int* **histSize**, int* **lowerLevel**, int* **upperLevel**, Stream& **stream**=Stream::Null())

>   **Parameters**

>>   **src** – Source image. `CV_8U`, `CV_16U`, or `CV_16S` depth and 1 or 4 channels are supported. For a four-channel image, all channels are processed separately.

>>   **hist** – Destination histogram with one row, `histSize` columns, and the `CV_32S` type.

>>   **histSize** – Size of the histogram.

>>   **lowerLevel** – Lower boundary of lowest-level bin.

>>   **upperLevel** – Upper boundary of highest-level bin.

>>   **buf** – Optional buffer to avoid extra memory allocations (for many calls with the same sizes).

>>   **stream** – Stream for the asynchronous version.

gpu::histRange

Calculates a histogram with bins determined by the `levels` array.

C++: void gpu::**histRange**(const GpuMat& **src**, GpuMat& **hist**, const GpuMat& **levels**, Stream& **stream**=Stream::Null())

C++: void gpu::**histRange**(const GpuMat& **src**, GpuMat& **hist**, const GpuMat& **levels**, GpuMat& **buf**, Stream& **stream**=Stream::Null())

C++: void gpu::**histRange**(const GpuMat& **src**, GpuMat* **hist**, const GpuMat* **levels**, Stream& **stream**=Stream::Null())

C++: void gpu::**histRange**(const GpuMat& **src**, GpuMat* **hist**, const GpuMat* **levels**, GpuMat& **buf**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image. `CV_8U` , `CV_16U` , or `CV_16S` depth and 1 or 4 channels are supported. For a four-channel image, all channels are processed separately.
> >
> > **hist** – Destination histogram with one row, (`levels.cols-1`) columns, and the `CV_32SC1` type.
> >
> > **levels** – Number of levels in the histogram.
> >
> > **buf** – Optional buffer to avoid extra memory allocations (for many calls with the same sizes).
> >
> > **stream** – Stream for the asynchronous version.

gpu::calcHist

Calculates histogram for one channel 8-bit image.

C++: void gpu::**calcHist**(const GpuMat& **src**, GpuMat& **hist**, Stream& **stream**=Stream::Null())

C++: void gpu::**calcHist**(const GpuMat& **src**, GpuMat& **hist**, GpuMat& **buf**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image.
> >
> > **hist** – Destination histogram with one row, 256 columns, and the `CV_32SC1` type.
> >
> > **buf** – Optional buffer to avoid extra memory allocations (for many calls with the same sizes).
> >
> > **stream** – Stream for the asynchronous version.

gpu::equalizeHist

Equalizes the histogram of a grayscale image.

C++: void gpu::**equalizeHist**(const GpuMat& **src**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

C++: void gpu::**equalizeHist**(const GpuMat& **src**, GpuMat& **dst**, GpuMat& **hist**, Stream& **stream**=Stream::Null())

C++: void gpu::**equalizeHist**(const GpuMat& **src**, GpuMat& **dst**, GpuMat& **hist**, GpuMat& **buf**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image.
> >
> > **dst** – Destination image.
> >
> > **hist** – Destination histogram with one row, 256 columns, and the `CV_32SC1` type.
> >
> > **buf** – Optional buffer to avoid extra memory allocations (for many calls with the same sizes).
> >
> > **stream** – Stream for the asynchronous version.

See Also:

equalizeHist()

gpu::buildWarpPlaneMaps

Builds plane warping maps.

C++: void gpu::**buildWarpPlaneMaps**(Size **src_size**, Rect **dst_roi**, const Mat& **R**, double **f**, double **s**, double **dist**, GpuMat& **map_x**, GpuMat& **map_y**, Stream& **stream**=Stream::Null())

       Parameters

              stream – Stream for the asynchronous version.

gpu::buildWarpCylindricalMaps

Builds cylindrical warping maps.

C++: void gpu::**buildWarpCylindricalMaps**(Size **src_size**, Rect **dst_roi**, const Mat& **R**, double **f**, double **s**, GpuMat& **map_x**, GpuMat& **map_y**, Stream& **stream**=Stream::Null())

       Parameters

              stream – Stream for the asynchronous version.

gpu::buildWarpSphericalMaps

Builds spherical warping maps.

C++: void gpu::**buildWarpSphericalMaps**(Size **src_size**, Rect **dst_roi**, const Mat& **R**, double **f**, double **s**, GpuMat& **map_x**, GpuMat& **map_y**, Stream& **stream**=Stream::Null())

       Parameters

              stream – Stream for the asynchronous version.

gpu::pyrDown

Smoothes an image and downsamples it.

C++: void gpu::**pyrDown**(const GpuMat& **src**, GpuMat& **dst**, int **borderType**=BORDER_DEFAULT, Stream& **stream**=Stream::Null())

       Parameters

              src – Source image.

              dst – Destination image. Will have `Size((src.cols+1)/2, (src.rows+1)/2)` size and the same type as `src` .

              borderType – Pixel extrapolation method (see `borderInterpolate()`). `BORDER_REFLECT101` , `BORDER_REPLICATE` , `BORDER_CONSTANT` , `BORDER_REFLECT` and `BORDER_WRAP` are supported for now.

              stream – Stream for the asynchronous version.

See Also:

`pyrDown()`

gpu::pyrUp

Upsamples an image and then smoothes it.

C++: void gpu::**pyrUp**(const GpuMat& **src**, GpuMat& **dst**, int **borderType**=BORDER_DEFAULT, Stream& **stream**=Stream::Null())

>   **Parameters**

>> **src** – Source image.

>> **dst** – Destination image. Will have `Size(src.cols*2, src.rows*2)` size and the same type as `src`.

>> **borderType** – Pixel extrapolation method (see `borderInterpolate()`). `BORDER_REFLECT101` , `BORDER_REPLICATE` , `BORDER_CONSTANT` , `BORDER_REFLECT` and `BORDER_WRAP` are supported for now.

>> **stream** – Stream for the asynchronous version.

See Also:

`pyrUp()`

gpu::blendLinear

Performs linear blending of two images.

C++: void gpu::**blendLinear**(const GpuMat& **img1**, const GpuMat& **img2**, const GpuMat& **weights1**, const GpuMat& **weights2**, GpuMat& **result**, Stream& **stream**=Stream::Null())

>   **Parameters**

>> **img1** – First image. Supports only `CV_8U` and `CV_32F` depth.

>> **img1** – Second image. Must have the same size and the same type as `img1` .

>> **weights1** – Weights for first image. Must have tha same size as `img1` . Supports only `CV_32F` type.

>> **weights2** – Weights for second image. Must have tha same size as `img2` . Supports only `CV_32F` type.

>> **result** – Destination image.

>> **stream** – Stream for the asynchronous version.

gpu::Canny

Finds edges in an image using the [Canny86] algorithm.

C++: void gpu::**Canny**(const GpuMat& **image**, GpuMat& **edges**, double **low_thresh**, double **high_thresh**, int **apperture_size**=3, bool **L2gradient**=false)

C++: void gpu::**Canny**(const GpuMat& **image**, CannyBuf& **buf**, GpuMat& **edges**, double **low_thresh**, double **high_thresh**, int **apperture_size**=3, bool **L2gradient**=false)

C++: void gpu::**Canny**(const GpuMat& **dx**, const GpuMat& **dy**, GpuMat& **edges**, double **low_thresh**, double **high_thresh**, bool **L2gradient**=false)

C++: void gpu::**Canny**(const GpuMat& **dx**, const GpuMat& **dy**, CannyBuf& **buf**, GpuMat& **edges**, double **low_thresh**, double **high_thresh**, bool **L2gradient**=false)

Parameters

> **image** – Single-channel 8-bit input image.
>
> **dx** – First derivative of image in the vertical direction. Support only `CV_32S` type.
>
> **dy** – First derivative of image in the horizontal direction. Support only `CV_32S` type.
>
> **edges** – Output edge map. It has the same size and type as `image` .
>
> **threshold1** – First threshold for the hysteresis procedure.
>
> **threshold2** – Second threshold for the hysteresis procedure.
>
> **apertureSize** – Aperture size for the `Sobel()` operator.
>
> **L2gradient** – Flag indicating whether a 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`).
>
> **buf** – Optional buffer to avoid extra memory allocations (for many calls with the same sizes).

See Also:

`Canny()`

11.7 Matrix Reductions

gpu::meanStdDev

Computes a mean value and a standard deviation of matrix elements.

C++: void gpu::**meanStdDev**(const GpuMat& **mtx**, Scalar& **mean**, Scalar& **stddev**)

> **Parameters**
>
> > **mtx** – Source matrix. `CV_8UC1` matrices are supported for now.
> >
> > **mean** – Mean value.
> >
> > **stddev** – Standard deviation value.

See Also:

`meanStdDev()`

gpu::norm

Returns the norm of a matrix (or difference of two matrices).

C++: double gpu::**norm**(const GpuMat& **src1**, int **normType**=NORM_L2)

C++: double gpu::**norm**(const GpuMat& **src1**, int **normType**, GpuMat& **buf**)

C++: double gpu::**norm**(const GpuMat& **src1**, const GpuMat& **src2**, int **normType**=NORM_L2)

> **Parameters**
>
> > **src1** – Source matrix. Any matrices except 64F are supported.
> >
> > **src2** – Second source matrix (if any) with the same size and type as `src1`.
> >
> > **normType** – Norm type. `NORM_L1` , `NORM_L2` , and `NORM_INF` are supported for now.

> **buf** – Optional buffer to avoid extra memory allocations. It is resized automatically.

See Also:

norm()

gpu::sum

Returns the sum of matrix elements.

C++: Scalar gpu::**sum**(const GpuMat& **src**)

C++: Scalar gpu::**sum**(const GpuMat& **src**, GpuMat& **buf**)

> **Parameters**
>
>> **src** – Source image of any depth except for CV_64F .
>>
>> **buf** – Optional buffer to avoid extra memory allocations. It is resized automatically.

See Also:

sum()

gpu::absSum

Returns the sum of absolute values for matrix elements.

C++: Scalar gpu::**absSum**(const GpuMat& **src**)

C++: Scalar gpu::**absSum**(const GpuMat& **src**, GpuMat& **buf**)

> **Parameters**
>
>> **src** – Source image of any depth except for CV_64F .
>>
>> **buf** – Optional buffer to avoid extra memory allocations. It is resized automatically.

gpu::sqrSum

Returns the squared sum of matrix elements.

C++: Scalar gpu::**sqrSum**(const GpuMat& **src**)

C++: Scalar gpu::**sqrSum**(const GpuMat& **src**, GpuMat& **buf**)

> **Parameters**
>
>> **src** – Source image of any depth except for CV_64F .
>>
>> **buf** – Optional buffer to avoid extra memory allocations. It is resized automatically.

gpu::minMax

Finds global minimum and maximum matrix elements and returns their values.

C++: void gpu::**minMax**(const GpuMat& **src**, double* **minVal**, double* **maxVal**=0, const GpuMat& **mask**=GpuMat())

C++: void gpu::**minMax**(const GpuMat& **src**, double* **minVal**, double* **maxVal**, const GpuMat& **mask**, GpuMat& **buf**)

> **Parameters**

>> **src** – Single-channel source image.

>> **minVal** – Pointer to the returned minimum value. Use `NULL` if not required.

>> **maxVal** – Pointer to the returned maximum value. Use `NULL` if not required.

>> **mask** – Optional mask to select a sub-matrix.

>> **buf** – Optional buffer to avoid extra memory allocations. It is resized automatically.

The function does not work with `CV_64F` images on GPUs with the compute capability < 1.3.

See Also:

minMaxLoc()

gpu::minMaxLoc

Finds global minimum and maximum matrix elements and returns their values with locations.

C++: void gpu::**minMaxLoc**(const GpuMat& **src**, double* **minVal**, double* **maxVal**=0, Point* **minLoc**=0, Point* **maxLoc**=0, const GpuMat& **mask**=GpuMat())

C++: void gpu::**minMaxLoc**(const GpuMat& **src**, double* **minVal**, double* **maxVal**, Point* **minLoc**, Point* **maxLoc**, const GpuMat& **mask**, GpuMat& **valbuf**, GpuMat& **locbuf**)

> **Parameters**

>> **src** – Single-channel source image.

>> **minVal** – Pointer to the returned minimum value. Use `NULL` if not required.

>> **maxVal** – Pointer to the returned maximum value. Use `NULL` if not required.

>> **minValLoc** – Pointer to the returned minimum location. Use `NULL` if not required.

>> **maxValLoc** – Pointer to the returned maximum location. Use `NULL` if not required.

>> **mask** – Optional mask to select a sub-matrix.

>> **valbuf** – Optional values buffer to avoid extra memory allocations. It is resized automatically.

>> **locbuf** – Optional locations buffer to avoid extra memory allocations. It is resized automatically.

> The function does not work with `CV_64F` images on GPU with the compute capability < 1.3.

See Also:

minMaxLoc()

gpu::countNonZero

Counts non-zero matrix elements.

C++: int gpu::**countNonZero**(const GpuMat& **src**)

C++: int gpu::**countNonZero**(const GpuMat& **src**, GpuMat& **buf**)

> **Parameters**

>> **src** – Single-channel source image.

> > > **buf** – Optional buffer to avoid extra memory allocations. It is resized automatically.

The function does not work with `CV_64F` images on GPUs with the compute capability < 1.3.

See Also:

`countNonZero()`

gpu::reduce

Reduces a matrix to a vector.

C++: void gpu::**reduce**(const GpuMat& **mtx**, GpuMat& **vec**, int **dim**, int **reduceOp**, int **dtype**=-1, Stream& **stream**=Stream::Null())

> > **Parameters**

> > > **mtx** – Source 2D matrix.

> > > **vec** – Destination vector. Its size and type is defined by `dim` and `dtype` parameters.

> > > **dim** – Dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row. 1 means that the matrix is reduced to a single column.

> > > **reduceOp** – Reduction operation that could be one of the following:

> > > – **CV_REDUCE_SUM** The output is the sum of all rows/columns of the matrix.

> > > – **CV_REDUCE_AVG** The output is the mean vector of all rows/columns of the matrix.

> > > – **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all rows/columns of the matrix.

> > > – **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all rows/columns of the matrix.

> > > **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 the 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 a raster image. In case of `CV_REDUCE_SUM` and `CV_REDUCE_AVG`, the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

See Also:

`reduce()`

11.8 Object Detection

gpu::HOGDescriptor

class gpu::**HOGDescriptor**

The class implements Histogram of Oriented Gradients ([Dalal2005]) object detector.

```
struct CV_EXPORTS HOGDescriptor
{
    enum { DEFAULT_WIN_SIGMA = -1 };
    enum { DEFAULT_NLEVELS = 64 };
    enum { DESCR_FORMAT_ROW_BY_ROW, DESCR_FORMAT_COL_BY_COL };

    HOGDescriptor(Size win_size=Size(64, 128), Size block_size=Size(16, 16),
                  Size block_stride=Size(8, 8), Size cell_size=Size(8, 8),
                  int nbins=9, double win_sigma=DEFAULT_WIN_SIGMA,
                  double threshold_L2hys=0.2, bool gamma_correction=true,
                  int nlevels=DEFAULT_NLEVELS);

    size_t getDescriptorSize() const;
    size_t getBlockHistogramSize() const;

    void setSVMDetector(const vector<float>& detector);

    static vector<float> getDefaultPeopleDetector();
    static vector<float> getPeopleDetector48x96();
    static vector<float> getPeopleDetector64x128();

    void detect(const GpuMat& img, vector<Point>& found_locations,
                double hit_threshold=0, Size win_stride=Size(),
                Size padding=Size());

    void detectMultiScale(const GpuMat& img, vector<Rect>& found_locations,
                          double hit_threshold=0, Size win_stride=Size(),
                          Size padding=Size(), double scale0=1.05,
                          int group_threshold=2);

    void getDescriptors(const GpuMat& img, Size win_stride,
                        GpuMat& descriptors,
                        int descr_format=DESCR_FORMAT_COL_BY_COL);

    Size win_size;
    Size block_size;
    Size block_stride;
    Size cell_size;
    int nbins;
    double win_sigma;
    double threshold_L2hys;
    bool gamma_correction;
    int nlevels;

private:
    // Hidden
}
```

Interfaces of all methods are kept similar to the CPU HOG descriptor and detector analogues as much as possible.

gpu::HOGDescriptor::HOGDescriptor

Creates the HOG descriptor and detector.

C++: `gpu::HOGDescriptor::`**`HOGDescriptor`**(Size **win_size**=Size(64, 128), Size **block_size**=Size(16, 16), Size **block_stride**=Size(8, 8), Size **cell_size**=Size(8, 8), int **nbins**=9, double **win_sigma**=DEFAULT_WIN_SIGMA, double **threshold_L2hys**=0.2, bool **gamma_correction**=true, int **nlevels**=DEFAULT_NLEVELS)

>   Parameters

>>   **win_size** – Detection window size. Align to block size and block stride.

>>   **block_size** – Block size in pixels. Align to cell size. Only (16,16) is supported for now.

>>   **block_stride** – Block stride. It must be a multiple of cell size.

>>   **cell_size** – Cell size. Only (8, 8) is supported for now.

>>   **nbins** – Number of bins. Only 9 bins per cell are supported for now.

>>   **win_sigma** – Gaussian smoothing window parameter.

>>   **threshold_L2Hys** – L2-Hys normalization method shrinkage.

>>   **gamma_correction** – Flag to specify whether the gamma correction preprocessing is required or not.

>>   **nlevels** – Maximum number of detection window increases.

gpu::HOGDescriptor::getDescriptorSize

Returns the number of coefficients required for the classification.

C++: size_t gpu::HOGDescriptor::**getDescriptorSize**() const

gpu::HOGDescriptor::getBlockHistogramSize

Returns the block histogram size.

C++: size_t gpu::HOGDescriptor::**getBlockHistogramSize**() const

gpu::HOGDescriptor::setSVMDetector

Sets coefficients for the linear SVM classifier.

C++: void gpu::HOGDescriptor::**setSVMDetector**(const vector<float>& **detector**)

gpu::HOGDescriptor::getDefaultPeopleDetector

Returns coefficients of the classifier trained for people detection (for default window size).

C++: **static** vector<float> gpu::HOGDescriptor::**getDefaultPeopleDetector**()

gpu::HOGDescriptor::getPeopleDetector48x96

Returns coefficients of the classifier trained for people detection (for 48x96 windows).

C++: **static** vector<float> gpu::HOGDescriptor::**getPeopleDetector48x96**()

gpu::HOGDescriptor::getPeopleDetector64x128

Returns coefficients of the classifier trained for people detection (for 64x128 windows).

C++: **static** vector<float> gpu::HOGDescriptor::**getPeopleDetector64x128**()

gpu::HOGDescriptor::detect

Performs object detection without a multi-scale window.

C++: void gpu::HOGDescriptor::**detect**(const GpuMat& **img**, vector<Point>& **found_locations**, double **hit_threshold**=0, Size **win_stride**=Size(), Size **padding**=Size())

> **Parameters**
>
> > **img** – Source image. `CV_8UC1` and `CV_8UC4` types are supported for now.
> >
> > **found_locations** – Left-top corner points of detected objects boundaries.
> >
> > **hit_threshold** – Threshold for the distance between features and SVM classifying plane. Usually it is 0 and should be specfied in the detector coefficients (as the last free coefficient). But if the free coefficient is omitted (which is allowed), you can specify it manually here.
> >
> > **win_stride** – Window stride. It must be a multiple of block stride.
> >
> > **padding** – Mock parameter to keep the CPU interface compatibility. It must be (0,0).

gpu::HOGDescriptor::detectMultiScale

Performs object detection with a multi-scale window.

C++: void gpu::HOGDescriptor::**detectMultiScale**(const GpuMat& **img**, vector<Rect>& **found_locations**, double **hit_threshold**=0, Size **win_stride**=Size(), Size **padding**=Size(), double **scale0**=1.05, int **group_threshold**=2)

> **Parameters**
>
> > **img** – Source image. See `gpu::HOGDescriptor::detect()` for type limitations.
> >
> > **found_locations** – Detected objects boundaries.
> >
> > **hit_threshold** – Threshold for the distance between features and SVM classifying plane. See `gpu::HOGDescriptor::detect()` for details.
> >
> > **win_stride** – Window stride. It must be a multiple of block stride.
> >
> > **padding** – Mock parameter to keep the CPU interface compatibility. It must be (0,0).
> >
> > **scale0** – Coefficient of the detection window increase.
> >
> > **group_threshold** – Coefficient to regulate the similarity threshold. When detected, some objects can be covered by many rectangles. 0 means not to perform grouping. See `groupRectangles()` .

gpu::HOGDescriptor::getDescriptors

Returns block descriptors computed for the whole image.

C++: void gpu::HOGDescriptor::**getDescriptors**(const GpuMat& **img**, Size **win_stride**, GpuMat& **descriptors**, int **descr_format**=DESCR_FORMAT_COL_BY_COL)

> **Parameters**
>
>> **img** – Source image. See `gpu::HOGDescriptor::detect()` for type limitations.
>>
>> **win_stride** – Window stride. It must be a multiple of block stride.
>>
>> **descriptors** – 2D array of descriptors.
>>
>> **descr_format** – Descriptor storage format:
>>
>>> – **DESCR_FORMAT_ROW_BY_ROW** - Row-major order.
>>>
>>> – **DESCR_FORMAT_COL_BY_COL** - Column-major order.

The function is mainly used to learn the classifier.

gpu::CascadeClassifier_GPU

class gpu::**CascadeClassifier_GPU**

Cascade classifier class used for object detection.

```cpp
class CV_EXPORTS CascadeClassifier_GPU
{
public:
        CascadeClassifier_GPU();
        CascadeClassifier_GPU(const string& filename);
        ~CascadeClassifier_GPU();

        bool empty() const;
        bool load(const string& filename);
        void release();

        /* Returns number of detected objects */
        int detectMultiScale( const GpuMat& image, GpuMat& objectsBuf, double scaleFactor=1.2, int minNeighbors=4, Siz

        /* Finds only the largest object. Special mode if training is required.*/
        bool findLargestObject;

        /* Draws rectangles in input image */
        bool visualizeInPlace;

        Size getClassifierSize() const;
};
```

gpu::CascadeClassifier_GPU::CascadeClassifier_GPU

Loads the classifier from a file.

C++: gpu::**CascadeClassifier_GPU**(const string& **filename**)

> **Parameters**
>
>> **filename** – Name of the file from which the classifier is loaded. Only the old `haar` classifier (trained by the `haar` training application) and NVIDIA's `nvbin` are supported.

gpu::CascadeClassifier_GPU::empty

Checks whether the classifier is loaded or not.

C++: bool gpu::CascadeClassifier_GPU::**empty**() const

gpu::CascadeClassifier_GPU::load

Loads the classifier from a file. The previous content is destroyed.

C++: bool gpu::CascadeClassifier_GPU::**load**(const string& **filename**)

> **Parameters**
>
>> **filename** – Name of the file from which the classifier is loaded. Only the old haar classifier (trained by the haar training application) and NVIDIA's nvbin are supported.

gpu::CascadeClassifier_GPU::release

Destroys the loaded classifier.

C++: void gpu::CascadeClassifier_GPU::**release**()

gpu::CascadeClassifier_GPU::detectMultiScale

Detects objects of different sizes in the input image.

C++: int gpu::CascadeClassifier_GPU::**detectMultiScale**(const GpuMat& **image**, GpuMat& **objects-Buf**, double **scaleFactor**=1.2, int **min-Neighbors**=4, Size **minSize**=Size())

> **Parameters**
>
>> **image** – Matrix of type CV_8U containing an image where objects should be detected.
>>
>> **objects** – Buffer to store detected objects (rectangles). If it is empty, it is allocated with the default size. If not empty, the function searches not more than N objects, where N = sizeof(objectsBufer's data)/sizeof(cv::Rect).
>>
>> **scaleFactor** – Value to specify how much the image size is reduced at each image scale.
>>
>> **minNeighbors** – Value to specify how many neighbours each candidate rectangle has to retain.
>>
>> **minSize** – Minimum possible object size. Objects smaller than that are ignored.

The detected objects are returned as a list of rectangles.

The function returns the number of detected objects, so you can retrieve them as in the following example:

```
gpu::CascadeClassifier_GPU cascade_gpu(...);

Mat image_cpu = imread(...)
GpuMat image_gpu(image_cpu);

GpuMat objbuf;
int detections_number = cascade_gpu.detectMultiScale( image_gpu,
        objbuf, 1.2, minNeighbors);
```

```
Mat obj_host;
// download only detected number of rectangles
objbuf.colRange(0, detections_number).download(obj_host);

Rect* faces = obj_host.ptr<Rect>();
for(int i = 0; i < detections_num; ++i)
   cv::rectangle(image_cpu, faces[i], Scalar(255));

imshow("Faces", image_cpu);
```

See Also:

CascadeClassifier::detectMultiScale()

11.9 Feature Detection and Description

gpu::SURF_GPU

class gpu::**SURF_GPU**

Class used for extracting Speeded Up Robust Features (SURF) from an image.

```
class SURF_GPU : public CvSURFParams
{
public:
    enum KeypointLayout
    {
        SF_X = 0,
        SF_Y,
        SF_LAPLACIAN,
        SF_SIZE,
        SF_DIR,
        SF_HESSIAN,
        SF_FEATURE_STRIDE
    };

    //! the default constructor
    SURF_GPU();
    //! the full constructor taking all the necessary parameters
    explicit SURF_GPU(double _hessianThreshold, int _nOctaves=4,
        int _nOctaveLayers=2, bool _extended=false, float _keypointsRatio=0.01f);

    //! returns the descriptor size in float's (64 or 128)
    int descriptorSize() const;

    //! upload host keypoints to device memory
    void uploadKeypoints(const vector<KeyPoint>& keypoints,
        GpuMat& keypointsGPU);
    //! download keypoints from device to host memory
    void downloadKeypoints(const GpuMat& keypointsGPU,
        vector<KeyPoint>& keypoints);

    //! download descriptors from device to host memory
    void downloadDescriptors(const GpuMat& descriptorsGPU,
        vector<float>& descriptors);
```

```
    void operator()(const GpuMat& img, const GpuMat& mask,
        GpuMat& keypoints);

    void operator()(const GpuMat& img, const GpuMat& mask,
        GpuMat& keypoints, GpuMat& descriptors,
        bool useProvidedKeypoints = false,
        bool calcOrientation = true);

    void operator()(const GpuMat& img, const GpuMat& mask,
        std::vector<KeyPoint>& keypoints);

    void operator()(const GpuMat& img, const GpuMat& mask,
        std::vector<KeyPoint>& keypoints, GpuMat& descriptors,
        bool useProvidedKeypoints = false,
        bool calcOrientation = true);

    void operator()(const GpuMat& img, const GpuMat& mask,
        std::vector<KeyPoint>& keypoints,
        std::vector<float>& descriptors,
        bool useProvidedKeypoints = false,
        bool calcOrientation = true);

    void releaseMemory();

    //! max keypoints = keypointsRatio * img.size().area()
    float keypointsRatio;

    GpuMat sum, mask1, maskSum, intBuffer;

    GpuMat det, trace;

    GpuMat maxPosBuffer;
};
```

The class `SURF_GPU` implements Speeded Up Robust Features descriptor. There is a fast multi-scale Hessian keypoint detector that can be used to find the keypoints (which is the default option). But the descriptors can also be computed for the user-specified keypoints. Only 8-bit grayscale images are supported.

The class `SURF_GPU` can store results in the GPU and CPU memory. It provides functions to convert results between CPU and GPU version (`uploadKeypoints`, `downloadKeypoints`, `downloadDescriptors`). The format of CPU results is the same as `SURF` results. GPU results are stored in `GpuMat`. The `keypoints` matrix is $nFeatures \times 6$ matrix with the `CV_32FC1` type.

- `keypoints.ptr<float>(SF_X)[i]` contains x coordinate of the i-th feature.

- `keypoints.ptr<float>(SF_Y)[i]` contains y coordinate of the i-th feature.

- `keypoints.ptr<float>(SF_LAPLACIAN)[i]` contains the laplacian sign of the i-th feature.

- `keypoints.ptr<float>(SF_SIZE)[i]` contains the size of the i-th feature.

- `keypoints.ptr<float>(SF_DIR)[i]` contain orientation of the i-th feature.

- `keypoints.ptr<float>(SF_HESSIAN)[i]` contains the response of the i-th feature.

The `descriptors` matrix is $nFeatures \times$ `descriptorSize` matrix with the `CV_32FC1` type.

The class `SURF_GPU` uses some buffers and provides access to it. All buffers can be safely released between function calls.

See Also:

SURF

gpu::FAST_GPU

class gpu::**FAST_GPU**

Class used for corner detection using the FAST algorithm.

```cpp
class FAST_GPU
{
public:
    enum
    {
        LOCATION_ROW = 0,
        RESPONSE_ROW,
        ROWS_COUNT
    };

    // all features have same size
    static const int FEATURE_SIZE = 7;

    explicit FAST_GPU(int threshold, bool nonmaxSupression = true,
                      double keypointsRatio = 0.05);

    void operator ()(const GpuMat& image, const GpuMat& mask, GpuMat& keypoints);
    void operator ()(const GpuMat& image, const GpuMat& mask,
                     std::vector<KeyPoint>& keypoints);

    void downloadKeypoints(const GpuMat& d_keypoints,
                           std::vector<KeyPoint>& keypoints);

    void convertKeypoints(const Mat& h_keypoints,
                          std::vector<KeyPoint>& keypoints);

    void release();

    bool nonmaxSupression;

    int threshold;

    double keypointsRatio;

    int calcKeyPointsLocation(const GpuMat& image, const GpuMat& mask);

    int getKeyPoints(GpuMat& keypoints);
};
```

The class FAST_GPU implements FAST corner detection algorithm.

See Also:

FAST()

gpu::FAST_GPU::FAST_GPU

Constructor.

C++: `gpu::FAST_GPU::`**`FAST_GPU`**`(int` **threshold**, bool **nonmaxSupression**=true, double **keypointsRatio**=0.05`)`

> Parameters

>> **threshold** – Threshold on difference between intensity of the central pixel and pixels on a circle around this pixel.

>> **nonmaxSupression** – If it is true, non-maximum supression is applied to detected corners (keypoints).

>> **keypointsRatio** – Inner buffer size for keypoints store is determined as (keypointsRatio * image_width * image_height).

gpu::FAST_GPU::operator ()

Finds the keypoints using FAST detector.

C++: void `gpu::FAST_GPU::`**`operator()`**`(const GpuMat&` **image**, const GpuMat& **mask**, GpuMat& **keypoints**`)`

C++: void `gpu::FAST_GPU::`**`operator()`**`(const GpuMat&` **image**, const GpuMat& **mask**, std::vector<KeyPoint>& **keypoints**`)`

> Parameters

>> **image** – Image where keypoints (corners) are detected. Only 8-bit grayscale images are supported.

>> **mask** – Optional input mask that marks the regions where we should detect features.

>> **keypoints** – The output vector of keypoints. Can be stored both in CPU and GPU memory. For GPU memory keypoints.ptr<Vec2s>(LOCATION_ROW)[i] will contain location of i'th point and keypoints.ptr<float>(RESPONSE_ROW)[i] will contaion response of i'th point (if non-maximum supression is applied).

gpu::FAST_GPU::downloadKeypoints

Download keypoints from GPU to CPU memory.

C++: void `gpu::FAST_GPU::`**`downloadKeypoints`**`(const GpuMat&` **d_keypoints**, std::vector<KeyPoint>& **keypoints**`)`

gpu::FAST_GPU::convertKeypoints

Converts keypoints from GPU representation to vector of `KeyPoint`.

C++: void `gpu::FAST_GPU::`**`convertKeypoints`**`(const Mat&` **h_keypoints**, std::vector<KeyPoint>& **keypoints**`)`

gpu::FAST_GPU::release

Releases inner buffer memory.

C++: void `gpu::FAST_GPU::`**`release`**`()`

gpu::FAST_GPU::calcKeyPointsLocation

Find keypoints and compute it's response if `nonmaxSupression` is true.

The function returns count of detected keypoints.

gpu::FAST_GPU::getKeyPoints

Gets final array of keypoints.

C++: int gpu::FAST_GPU::**getKeyPoints**(GpuMat& **keypoints**)

> **Parameters**
>
> > **keypoints** – The output vector of keypoints.

The function performs nonmax supression if needed and returns final count of keypoints.

gpu::ORB_GPU

class gpu::**ORB_GPU**

Class for extracting ORB features and descriptors from an image.

```
class ORB_GPU
{
public:
    enum
    {
        X_ROW = 0,
        Y_ROW,
        RESPONSE_ROW,
        ANGLE_ROW,
        OCTAVE_ROW,
        SIZE_ROW,
        ROWS_COUNT
    };

    enum
    {
        DEFAULT_FAST_THRESHOLD = 20
    };

    explicit ORB_GPU(size_t n_features = 500,
            const ORB::CommonParams& detector_params = ORB::CommonParams());

    void operator()(const GpuMat& image, const GpuMat& mask,
                    std::vector<KeyPoint>& keypoints);
    void operator()(const GpuMat& image, const GpuMat& mask, GpuMat& keypoints);

    void operator()(const GpuMat& image, const GpuMat& mask,
                    std::vector<KeyPoint>& keypoints, GpuMat& descriptors);
    void operator()(const GpuMat& image, const GpuMat& mask,
                    GpuMat& keypoints, GpuMat& descriptors);

    void downloadKeyPoints(GpuMat& d_keypoints, std::vector<KeyPoint>& keypoints);

    void convertKeyPoints(Mat& d_keypoints, std::vector<KeyPoint>& keypoints);
```

```
    int descriptorSize() const;

    void setParams(size_t n_features, const ORB::CommonParams& detector_params);
    void setFastParams(int threshold, bool nonmaxSupression = true);

    void release();

    bool blurForDescriptor;
};
```

The class implements ORB feature detection and description algorithm.

gpu::ORB_GPU::ORB_GPU

Constructor.

C++: gpu::ORB_GPU::**ORB_GPU**(size_t **n_features**=500, const ORB::CommonParams& **detector_params**=ORB::CommonParams())

gpu::ORB_GPU::operator()

Detects keypoints and computes descriptors for them.

C++: void gpu::ORB_GPU::**operator()**(const GpuMat& **image**, const GpuMat& **mask**, std::vector<KeyPoint>& **keypoints**)

C++: void gpu::ORB_GPU::**operator()**(const GpuMat& **image**, const GpuMat& **mask**, GpuMat& **keypoints**)

C++: void gpu::ORB_GPU::**operator()**(const GpuMat& **image**, const GpuMat& **mask**, std::vector<KeyPoint>& **keypoints**, GpuMat& **descriptors**)

C++: void gpu::ORB_GPU::**operator()**(const GpuMat& **image**, const GpuMat& **mask**, GpuMat& **keypoints**, GpuMat& **descriptors**)

>   **Parameters**

>   >   **image** – Input 8-bit grayscale image.

>   >   **mask** – Optional input mask that marks the regions where we should detect features.

>   >   **keypoints** – The input/output vector of keypoints. Can be stored both in CPU and GPU memory. For GPU memory * keypoints.ptr<float>(X_ROW)[i] contains x coordinate of the i'th feature. * keypoints.ptr<float>(Y_ROW)[i] contains y coordinate of the i'th feature. * keypoints.ptr<float>(RESPONSE_ROW)[i] contains the response of the i'th feature. * keypoints.ptr<float>(ANGLE_ROW)[i] contains orientation of the i'th feature. * keypoints.ptr<float>(OCTAVE_ROW)[i] contains the octave of the i'th feature. * keypoints.ptr<float>(SIZE_ROW)[i] contains the size of the i'th feature.

>   >   **descriptors** – Computed descriptors. if blurForDescriptor is true, image will be blurred before descriptors calculation.

gpu::ORB_GPU::downloadKeypoints

Download keypoints from GPU to CPU memory.

C++: void gpu::ORB_GPU::**downloadKeypoints**(const GpuMat& **d_keypoints**, std::vector<KeyPoint>& **keypoints**)

gpu::ORB_GPU::convertKeypoints

Converts keypoints from GPU representation to vector of `KeyPoint`.

C++: void gpu::ORB_GPU::**convertKeypoints**(const Mat& **h_keypoints**, std::vector<KeyPoint>& **key-points**)

gpu::ORB_GPU::release

Releases inner buffer memory.

C++: void gpu::ORB_GPU::**release**()

gpu::BruteForceMatcher_GPU

class gpu::**BruteForceMatcher_GPU**

Brute-force descriptor matcher. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. This descriptor matcher supports masking permissible matches between descriptor sets.

```cpp
template<class Distance>
class BruteForceMatcher_GPU
{
public:
    // Add descriptors to train descriptor collection.
    void add(const std::vector<GpuMat>& descCollection);

    // Get train descriptors collection.
    const std::vector<GpuMat>& getTrainDescriptors() const;

    // Clear train descriptors collection.
    void clear();

    // Return true if there are no train descriptors in collection.
    bool empty() const;

    // Return true if the matcher supports mask in match methods.
    bool isMaskSupported() const;

    void matchSingle(const GpuMat& query, const GpuMat& train,
        GpuMat& trainIdx, GpuMat& distance,
        const GpuMat& mask = GpuMat(), Stream& stream = Stream::Null());

    static void matchDownload(const GpuMat& trainIdx,
        const GpuMat& distance, std::vector<DMatch>& matches);
    static void matchConvert(const Mat& trainIdx,
        const Mat& distance, std::vector<DMatch>& matches);

    void match(const GpuMat& query, const GpuMat& train,
        std::vector<DMatch>& matches, const GpuMat& mask = GpuMat());

    void makeGpuCollection(GpuMat& trainCollection, GpuMat& maskCollection,
        const vector<GpuMat>& masks = std::vector<GpuMat>());

    void matchCollection(const GpuMat& query, const GpuMat& trainCollection,
        GpuMat& trainIdx, GpuMat& imgIdx, GpuMat& distance,
        const GpuMat& maskCollection, Stream& stream = Stream::Null());
```

```cpp
static void matchDownload(const GpuMat& trainIdx, GpuMat& imgIdx,
    const GpuMat& distance, std::vector<DMatch>& matches);
static void matchConvert(const Mat& trainIdx, const Mat& imgIdx,
    const Mat& distance, std::vector<DMatch>& matches);

void match(const GpuMat& query, std::vector<DMatch>& matches,
    const std::vector<GpuMat>& masks = std::vector<GpuMat>());

void knnMatchSingle(const GpuMat& query, const GpuMat& train,
    GpuMat& trainIdx, GpuMat& distance, GpuMat& allDist, int k,
    const GpuMat& mask = GpuMat(), Stream& stream = Stream::Null());

static void knnMatchDownload(const GpuMat& trainIdx, const GpuMat& distance,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);
static void knnMatchConvert(const Mat& trainIdx, const Mat& distance,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);

void knnMatch(const GpuMat& query, const GpuMat& train,
    std::vector< std::vector<DMatch> >& matches, int k,
    const GpuMat& mask = GpuMat(), bool compactResult = false);

void knnMatch2Collection(const GpuMat& query, const GpuMat& trainCollection,
    GpuMat& trainIdx, GpuMat& imgIdx, GpuMat& distance,
    const GpuMat& maskCollection = GpuMat(), Stream& stream = Stream::Null());

static void knnMatch2Download(const GpuMat& trainIdx, const GpuMat& imgIdx, const GpuMat& distance,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);
static void knnMatch2Convert(const Mat& trainIdx, const Mat& imgIdx, const Mat& distance,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);

void knnMatch(const GpuMat& query, std::vector< std::vector<DMatch> >& matches, int k,
    const std::vector<GpuMat>& masks = std::vector<GpuMat>(),
    bool compactResult = false);

void radiusMatchSingle(const GpuMat& query, const GpuMat& train,
    GpuMat& trainIdx, GpuMat& distance, GpuMat& nMatches, float maxDistance,
    const GpuMat& mask = GpuMat(), Stream& stream = Stream::Null());

static void radiusMatchDownload(const GpuMat& trainIdx, const GpuMat& distance, const GpuMat& nMatches,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);
static void radiusMatchConvert(const Mat& trainIdx, const Mat& distance, const Mat& nMatches,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);

void radiusMatch(const GpuMat& query, const GpuMat& train,
    std::vector< std::vector<DMatch> >& matches, float maxDistance,
    const GpuMat& mask = GpuMat(), bool compactResult = false);

void radiusMatchCollection(const GpuMat& query, GpuMat& trainIdx, GpuMat& imgIdx, GpuMat& distance, GpuMat& nMatch
    const std::vector<GpuMat>& masks = std::vector<GpuMat>(), Stream& stream = Stream::Null());

static void radiusMatchDownload(const GpuMat& trainIdx, const GpuMat& imgIdx, const GpuMat& distance, const GpuMat
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);
static void radiusMatchConvert(const Mat& trainIdx, const Mat& imgIdx, const Mat& distance, const Mat& nMatches,
    std::vector< std::vector<DMatch> >& matches, bool compactResult = false);

void radiusMatch(const GpuMat& query, std::vector< std::vector<DMatch> >& matches, float maxDistance,
    const std::vector<GpuMat>& masks = std::vector<GpuMat>(), bool compactResult = false);
```

```
private:
    std::vector<GpuMat> trainDescCollection;
};
```

The class `BruteForceMatcher_GPU` has an interface similar to the class `DescriptorMatcher`. It has two groups of
`match` methods: for matching descriptors of one image with another image or with an image set. Also, all functions
have an alternative to save results either to the GPU memory or to the CPU memory. The `Distance` template parameter
is kept for CPU/GPU interfaces similarity. `BruteForceMatcher_GPU` supports only the `L1<float>`, `L2<float>`, and
`Hamming` distance types.

See Also:

`DescriptorMatcher`, `BruteForceMatcher`

gpu::BruteForceMatcher_GPU::match

Finds the best match for each descriptor from a query set with train descriptors.

C++: void gpu::BruteForceMatcher_GPU::**match**(const GpuMat& **query**, const GpuMat& **train**,
std::vector<DMatch>& **matches**, const GpuMat&
mask=GpuMat())

C++: void gpu::BruteForceMatcher_GPU::**matchSingle**(const GpuMat& **query**, const GpuMat& **train**,
GpuMat& **trainIdx**, GpuMat& **distance**,
const GpuMat& **mask**=GpuMat(), Stream&
stream=Stream::Null())

C++: void gpu::BruteForceMatcher_GPU::**match**(const GpuMat& **query**, std::vector<DMatch>&
matches, const std::vector<GpuMat>&
masks=std::vector<GpuMat>())

C++: void gpu::BruteForceMatcher_GPU::**matchCollection**(const GpuMat& **query**, const GpuMat&
trainCollection, GpuMat& **trainIdx**,
GpuMat& **imgIdx**, GpuMat& **dis-
tance**, const GpuMat& **masks**, Stream&
stream=Stream::Null())

See Also:

`DescriptorMatcher::match()`

gpu::BruteForceMatcher_GPU::makeGpuCollection

Performs a GPU collection of train descriptors and masks in a suitable format for the
`gpu::BruteForceMatcher_GPU::matchCollection()` function.

C++: void gpu::BruteForceMatcher_GPU::**makeGpuCollection**(GpuMat& **trainCollection**,
GpuMat& **maskCollection**,
const vector<GpuMat>&
masks=std::vector<GpuMat>())

gpu::BruteForceMatcher_GPU::matchDownload

Downloads matrices obtained via `gpu::BruteForceMatcher_GPU::matchSingle()` or
`gpu::BruteForceMatcher_GPU::matchCollection()` to vector with `DMatch`.

C++: void gpu::BruteForceMatcher_GPU::**matchDownload**(const GpuMat& **trainIdx**, const GpuMat& **distance**, std::vector<DMatch>& **matches**)

C++: void gpu::BruteForceMatcher_GPU::**matchDownload**(const GpuMat& **trainIdx**, GpuMat& **imgIdx**, const GpuMat& **distance**, std::vector<DMatch>& **matches**)

gpu::BruteForceMatcher_GPU::matchConvert

Converts matrices obtained via `gpu::BruteForceMatcher_GPU::matchSingle()` or `gpu::BruteForceMatcher_GPU::matchCollection()` to vector with `DMatch`.

C++: void gpu::BruteForceMatcher_GPU::**matchConvert**(const Mat& **trainIdx**, const Mat& **distance**, std::vector<DMatch>& **matches**)

C++: void gpu::BruteForceMatcher_GPU::**matchConvert**(const Mat& **trainIdx**, const Mat& **imgIdx**, const Mat& **distance**, std::vector<DMatch>& **matches**)

gpu::BruteForceMatcher_GPU::knnMatch

Finds the k best matches for each descriptor from a query set with train descriptors.

C++: void gpu::BruteForceMatcher_GPU::**knnMatch**(const GpuMat& **query**, const GpuMat& **train**, std::vector<std::vector<DMatch>>& **matches**, int **k**, const GpuMat& **mask**=GpuMat(), bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**knnMatchSingle**(const GpuMat& **query**, const GpuMat& **train**, GpuMat& **trainIdx**, GpuMat& **distance**, GpuMat& **allDist**, int **k**, const GpuMat& **mask**=GpuMat(), Stream& **stream**=Stream::Null())

C++: void gpu::BruteForceMatcher_GPU::**knnMatch**(const GpuMat& **query**, std::vector<std::vector<DMatch>>& **matches**, int **k**, const std::vector<GpuMat>& **masks**=std::vector<GpuMat>(), bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**knnMatch2Collection**(const GpuMat& **query**, const GpuMat& **trainCollection**, GpuMat& **trainIdx**, GpuMat& **imgIdx**, GpuMat& **distance**, const GpuMat& **maskCollection**=GpuMat(), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **query** – Query set of descriptors.
> >
> > **train** – Training set of descriptors. It is not be added to train descriptors collection stored in the class object.
> >
> > **k** – Number of the best matches per each query descriptor (or less if it is not possible).
> >
> > **mask** – Mask specifying permissible matches between the input query and train matrices of descriptors.

compactResult – If `compactResult` is `true`, the `matches` vector does not contain matches for fully masked-out query descriptors.

stream – Stream for the asynchronous version.

The function returns detected k (or less if not possible) matches in the increasing order by distance.

The third variant of the method stores the results in GPU memory.

See Also:

`DescriptorMatcher::knnMatch()`

gpu::BruteForceMatcher_GPU::knnMatchDownload

Downloads matrices obtained via `gpu::BruteForceMatcher_GPU::knnMatchSingle()` or `gpu::BruteForceMatcher_GPU::knnMatch2Collection()` to vector with `DMatch`.

C++: void gpu::BruteForceMatcher_GPU::**knnMatchDownload**(const GpuMat& **trainIdx**, const GpuMat& **distance**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**knnMatch2Download**(const GpuMat& **trainIdx**, const GpuMat& **imgIdx**, const GpuMat& **distance**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

If `compactResult` is `true`, the `matches` vector does not contain matches for fully masked-out query descriptors.

gpu::BruteForceMatcher_GPU::knnMatchConvert

Converts matrices obtained via `gpu::BruteForceMatcher_GPU::knnMatchSingle()` or `gpu::BruteForceMatcher_GPU::knnMatch2Collection()` to CPU vector with `DMatch`.

C++: void gpu::BruteForceMatcher_GPU::**knnMatchConvert**(const Mat& **trainIdx**, const Mat& **distance**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**knnMatch2Convert**(const Mat& **trainIdx**, const Mat& **imgIdx**, const Mat& **distance**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

If `compactResult` is `true`, the `matches` vector does not contain matches for fully masked-out query descriptors.

gpu::BruteForceMatcher_GPU::radiusMatch

For each query descriptor, finds the best matches with a distance less than a given threshold.

C++: void gpu::BruteForceMatcher_GPU::**radiusMatch**(const GpuMat& **query**, const GpuMat& **train**, std::vector<std::vector<DMatch>>& **matches**, float **maxDistance**, const GpuMat& **mask**=GpuMat(), bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**radiusMatchSingle**(const GpuMat& **query**, const GpuMat& **train**, GpuMat& **trainIdx**, GpuMat& **distance**, GpuMat& **nMatches**, float **maxDistance**, const GpuMat& **mask**=GpuMat(), Stream& **stream**=Stream::Null())

C++: void gpu::BruteForceMatcher_GPU::**radiusMatch**(const GpuMat& **query**, std::vector<std::vector<DMatch>>& **matches**, float **maxDistance**, const std::vector<GpuMat>& **masks**=std::vector<GpuMat>(), bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**radiusMatchCollection**(const GpuMat& **query**, GpuMat& **trainIdx**, GpuMat& **imgIdx**, GpuMat& **distance**, GpuMat& **nMatches**, float **maxDistance**, const std::vector<GpuMat>& **masks**=std::vector<GpuMat>(), Stream& **stream**=Stream::Null())

>   Parameters
>
>>   **query** – Query set of descriptors.
>>
>>   **train** – Training set of descriptors. It is not added to train descriptors collection stored in the class object.
>>
>>   **maxDistance** – Distance threshold.
>>
>>   **mask** – Mask specifying permissible matches between the input query and train matrices of descriptors.
>>
>>   **compactResult** – If `compactResult` is `true` , the `matches` vector does not contain matches for fully masked-out query descriptors.
>>
>>   **stream** – Stream for the asynchronous version.

The function returns detected matches in the increasing order by distance.

The methods work only on devices with the compute capability $>= 1.1$.

The third variant of the method stores the results in GPU memory and does not store the points by the distance.

See Also:

`DescriptorMatcher::radiusMatch()`

gpu::BruteForceMatcher_GPU::radiusMatchDownload

Downloads matrices obtained via `gpu::BruteForceMatcher_GPU::radiusMatchSingle()` or `gpu::BruteForceMatcher_GPU::radiusMatchCollection()` to vector with `DMatch`.

C++: void gpu::BruteForceMatcher_GPU::**radiusMatchDownload**(const GpuMat& **trainIdx**, const GpuMat& **distance**, const GpuMat& **nMatches**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

`void gpu::BruteForceMatcher_GPU::radiusMatchDownload(const GpuMat& trainIdx, const GpuMat& imgIdx, const G`

If `compactResult` is `true` , the `matches` vector does not contain matches for fully masked-out query descriptors.

gpu::BruteForceMatcher_GPU::radiusMatchConvert

Converts matrices obtained via `gpu::BruteForceMatcher_GPU::radiusMatchSingle()` or `gpu::BruteForceMatcher_GPU::radiusMatchCollection()` to vector with `DMatch`.

C++: void gpu::BruteForceMatcher_GPU::**radiusMatchConvert**(const Mat& **trainIdx**, const Mat& **distance**, const Mat& **nMatches**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

C++: void gpu::BruteForceMatcher_GPU::**radiusMatchConvert**(const Mat& **trainIdx**, const Mat& **imgIdx**, const Mat& **distance**, const Mat& **nMatches**, std::vector<std::vector<DMatch>>& **matches**, bool **compactResult**=false)

If `compactResult` is `true` , the `matches` vector does not contain matches for fully masked-out query descriptors.

11.10 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images.

gpu::BaseRowFilter_GPU

class gpu::**BaseRowFilter_GPU**

Base class for linear or non-linear filters that processes rows of 2D arrays. Such filters are used for the "horizontal" filtering passes in separable filters.

```cpp
class BaseRowFilter_GPU
{
public:
    BaseRowFilter_GPU(int ksize_, int anchor_);
    virtual ~BaseRowFilter_GPU() {}
    virtual void operator()(const GpuMat& src, GpuMat& dst, Stream& stream = Stream::Null()) = 0;
    int ksize, anchor;
};
```

Note: This class does not allocate memory for a destination image. Usually this class is used inside `gpu::FilterEngine_GPU`.

---

gpu::BaseColumnFilter_GPU

class gpu::**BaseColumnFilter_GPU**

Base class for linear or non-linear filters that processes columns of 2D arrays. Such filters are used for the "vertical" filtering passes in separable filters.

```
class BaseColumnFilter_GPU
{
public:
    BaseColumnFilter_GPU(int ksize_, int anchor_);
    virtual ~BaseColumnFilter_GPU() {}
    virtual void operator()(const GpuMat& src, GpuMat& dst, Stream& stream = Stream::Null()) = 0;
    int ksize, anchor;
};
```

Note: This class does not allocate memory for a destination image. Usually this class is used inside
gpu::FilterEngine_GPU.

gpu::BaseFilter_GPU

class gpu::**BaseFilter_GPU**

Base class for non-separable 2D filters.

```
class CV_EXPORTS BaseFilter_GPU
{
public:
    BaseFilter_GPU(const Size& ksize_, const Point& anchor_);
    virtual ~BaseFilter_GPU() {}
    virtual void operator()(const GpuMat& src, GpuMat& dst, Stream& stream = Stream::Null()) = 0;
    Size ksize;
    Point anchor;
};
```

Note: This class does not allocate memory for a destination image. Usually this class is used inside
gpu::FilterEngine_GPU.

gpu::FilterEngine_GPU

class gpu::**FilterEngine_GPU**

Base class for the Filter Engine.

```
class CV_EXPORTS FilterEngine_GPU
{
public:
    virtual ~FilterEngine_GPU() {}

    virtual void apply(const GpuMat& src, GpuMat& dst,
                       Rect roi = Rect(0,0,-1,-1), Stream& stream = Stream::Null()) = 0;
};
```

The class can be used to apply an arbitrary filtering operation to an image. It contains all the necessary intermediate buffers. Pointers to the initialized FilterEngine_GPU instances are returned by various create*Filter_GPU functions (see below), and they are used inside high-level functions such as gpu::filter2D(), gpu::erode(), gpu::Sobel() , and others.

By using FilterEngine_GPU instead of functions you can avoid unnecessary memory allocation for intermediate buffers and get better performance:

```
while (...)
{
    gpu::GpuMat src = getImg();
    gpu::GpuMat dst;
    // Allocate and release buffers at each iterations
    gpu::GaussianBlur(src, dst, ksize, sigma1);
}

// Allocate buffers only once
cv::Ptr<gpu::FilterEngine_GPU> filter =
    gpu::createGaussianFilter_GPU(CV_8UC4, ksize, sigma1);
while (...)
{
    gpu::GpuMat src = getImg();
    gpu::GpuMat dst;
    filter->apply(src, dst, cv::Rect(0, 0, src.cols, src.rows));
}
// Release buffers only once
filter.release();
```

FilterEngine_GPU can process a rectangular sub-region of an image. By default, if roi == Rect(0,0,-1,-1) , FilterEngine_GPU processes the inner region of an image (Rect(anchor.x, anchor.y, src_size.width - ksize.width, src_size.height - ksize.height)) because some filters do not check whether indices are outside the image for better perfomance. See below to understand which filters support processing the whole image and which do not and identify image type limitations.

Note: The GPU filters do not support the in-place mode.

See Also:

gpu::BaseRowFilter_GPU,   gpu::BaseColumnFilter_GPU,   gpu::BaseFilter_GPU,
gpu::createFilter2D_GPU(),   gpu::createSeparableFilter_GPU(),   gpu::createBoxFilter_GPU(),
gpu::createMorphologyFilter_GPU(), gpu::createLinearFilter_GPU(), gpu::createSeparableLinearFilter_GPU(),
gpu::createDerivFilter_GPU(), gpu::createGaussianFilter_GPU()

gpu::createFilter2D_GPU

Creates a non-separable filter engine with the specified filter.

C++: Ptr<FilterEngine_GPU> gpu::**createFilter2D_GPU**(const Ptr<BaseFilter_GPU>& **filter2D**, int **srcType**, int **dstType**)

>   **Parameters**

>>   **filter2D** – Non-separable 2D filter.

>>   **srcType** – Input image type. It must be supported by filter2D .

>>   **dstType** – Output image type. It must be supported by filter2D .

Usually this function is used inside such high-level functions as gpu::createLinearFilter_GPU(), gpu::createBoxFilter_GPU().

gpu::createSeparableFilter_GPU

Creates a separable filter engine with the specified filters.

C++: Ptr<FilterEngine_GPU> gpu::**createSeparableFilter_GPU**(const Ptr<BaseRowFilter_GPU>& **rowFilter**, const Ptr<BaseColumnFilter_GPU>& **columnFilter**, int **srcType**, int **bufType**, int **dstType**)

> **Parameters**
>
> > **rowFilter** – "Horizontal" 1D filter.
> >
> > **columnFilter** – "Vertical" 1D filter.
> >
> > **srcType** – Input image type. It must be supported by `rowFilter`.
> >
> > **bufType** – Buffer image type. It must be supported by `rowFilter` and `columnFilter`.
> >
> > **dstType** – Output image type. It must be supported by `columnFilter`.

Usually this function is used inside such high-level functions as `gpu::createSeparableLinearFilter_GPU()`.

gpu::getRowSumFilter_GPU

Creates a horizontal 1D box filter.

C++: Ptr<BaseRowFilter_GPU> gpu::**getRowSumFilter_GPU**(int **srcType**, int **sumType**, int **ksize**, int **anchor**=-1)

> **Parameters**
>
> > **srcType** – Input image type. Only `CV_8UC1` type is supported for now.
> >
> > **sumType** – Output image type. Only `CV_32FC1` type is supported for now.
> >
> > **ksize** – Kernel size.
> >
> > **anchor** – Anchor point. The default value (-1) means that the anchor is at the kernel center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

gpu::getColumnSumFilter_GPU

Creates a vertical 1D box filter.

C++: Ptr<BaseColumnFilter_GPU> gpu::**getColumnSumFilter_GPU**(int **sumType**, int **dstType**, int **ksize**, int **anchor**=-1)

> **Parameters**
>
> > **sumType** – Input image type. Only `CV_8UC1` type is supported for now.
> >
> > **dstType** – Output image type. Only `CV_32FC1` type is supported for now.
> >
> > **ksize** – Kernel size.
> >
> > **anchor** – Anchor point. The default value (-1) means that the anchor is at the kernel center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

gpu::createBoxFilter_GPU

Creates a normalized 2D box filter.

C++: Ptr<FilterEngine_GPU> gpu::**createBoxFilter_GPU**(int **srcType**, int **dstType**, const Size& **ksize**, const Point& **anchor**=Point(-1,-1))

C++: Ptr<BaseFilter_GPU> **getBoxFilter_GPU**(int **srcType**, int **dstType**, const Size& **ksize**, Point **anchor**=Point(-1, -1))

> **Parameters**
>
> > **srcType** – Input image type supporting `CV_8UC1` and `CV_8UC4` .
> >
> > **dstType** – Output image type. It supports only the same values as the source type.
> >
> > **ksize** – Kernel size.
> >
> > **anchor** – Anchor point. The default value `Point(-1, -1)` means that the anchor is at the kernel center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

boxFilter()

gpu::boxFilter

Smooths the image using the normalized box filter.

C++: void gpu::**boxFilter**(const GpuMat& **src**, GpuMat& **dst**, int **ddepth**, Size **ksize**, Point **anchor**=Point(-1,-1), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Input image. `CV_8UC1` and `CV_8UC4` source types are supported.
> >
> > **dst** – Output image type. The size and type is the same as `src` .
> >
> > **ddepth** – Output image depth. If -1, the output image has the same depth as the input one. The only values allowed here are `CV_8U` and -1.
> >
> > **ksize** – Kernel size.
> >
> > **anchor** – Anchor point. The default value `Point(-1, -1)` means that the anchor is at the kernel center.
> >
> > **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

boxFilter()

gpu::blur

Acts as a synonym for the normalized box filter.

C++: void gpu::**blur**(const GpuMat& **src**, GpuMat& **dst**, Size **ksize**, Point **anchor**=Point(-1,-1), Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Input image. `CV_8UC1` and `CV_8UC4` source types are supported.
> >
> > **dst** – Output image type with the same size and type as `src` .
> >
> > **ksize** – Kernel size.
> >
> > **anchor** – Anchor point. The default value Point(-1, -1) means that the anchor is at the kernel center.
> >
> > **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

`blur()`, `gpu::boxFilter()`

gpu::createMorphologyFilter_GPU

Creates a 2D morphological filter.

C++: Ptr<FilterEngine_GPU> gpu::**createMorphologyFilter_GPU**(int **op**, int **type**, const Mat& **kernel**, const Point& **anchor**=Point(-1,-1), int **iterations**=1)

C++: Ptr<BaseFilter_GPU> **getMorphologyFilter_GPU**(int **op**, int **type**, const Mat& **kernel**, const Size& **ksize**, Point **anchor**=Point(-1,-1))

> **Parameters**
>
> > **op** – Morphology operation id. Only `MORPH_ERODE` and `MORPH_DILATE` are supported.
> >
> > **type** – Input/output image type. Only `CV_8UC1` and `CV_8UC4` are supported.
> >
> > **kernel** – 2D 8-bit structuring element for the morphological operation.
> >
> > **size** – Size of a horizontal or vertical structuring element used for separable morphological operations.
> >
> > **anchor** – Anchor position within the structuring element. Negative values mean that the anchor is at the center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

`createMorphologyFilter()`

gpu::erode

Erodes an image by using a specific structuring element.

C++: void gpu::**erode**(const GpuMat& **src**, GpuMat& **dst**, const Mat& **kernel**, Point **anchor**=Point(-1, -1), int **iterations**=1, Stream& **stream**=Stream::Null())

> **Parameters**
>> **src** – Source image. Only `CV_8UC1` and `CV_8UC4` types are supported.
>>
>> **dst** – Destination image with the same size and type as `src` .
>>
>> **kernel** – Structuring element used for erosion. If `kernel=Mat()`, a 3x3 rectangular structuring element is used.
>>
>> **anchor** – Position of an anchor within the element. The default value (`-1, -1`) means that the anchor is at the element center.
>>
>> **iterations** – Number of times erosion to be applied.
>>
>> **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

erode()

gpu::dilate

Dilates an image by using a specific structuring element.

C++: void gpu::**dilate**(const GpuMat& **src**, GpuMat& **dst**, const Mat& **kernel**, Point **anchor**=Point(-1, -1), int **iterations**=1, Stream& **stream**=Stream::Null())

> **Parameters**
>> **src** – Source image. `CV_8UC1` and `CV_8UC4` source types are supported.
>>
>> **dst** – Destination image with the same size and type as `src`.
>>
>> **kernel** – Structuring element used for dilation. If `kernel=Mat()`, a 3x3 rectangular structuring element is used.
>>
>> **anchor** – Position of an anchor within the element. The default value (`-1, -1`) means that the anchor is at the element center.
>>
>> **iterations** – Number of times dilation to be applied.
>>
>> **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

dilate()

gpu::morphologyEx

Applies an advanced morphological operation to an image.

C++: void gpu::**morphologyEx**(const GpuMat& **src**, GpuMat& **dst**, int **op**, const Mat& **kernel**, Point **anchor**=Point(-1, -1), int **iterations**=1, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image. `CV_8UC1` and `CV_8UC4` source types are supported.
> >
> > **dst** – Destination image with the same size and type as `src` .
> >
> > **op** – Type of morphological operation. The following types are possible:
> >
> > – **MORPH_OPEN** opening
> >
> > – **MORPH_CLOSE** closing
> >
> > – **MORPH_GRADIENT** morphological gradient
> >
> > – **MORPH_TOPHAT** "top hat"
> >
> > – **MORPH_BLACKHAT** "black hat"
> >
> > **kernel** – Structuring element.
> >
> > **anchor** – Position of an anchor within the element. The default value `Point(-1, -1)` means that the anchor is at the element center.
> >
> > **iterations** – Number of times erosion and dilation to be applied.
> >
> > **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

morphologyEx()

gpu::createLinearFilter_GPU

Creates a non-separable linear filter.

C++: Ptr<FilterEngine_GPU> gpu::**createLinearFilter_GPU**(int **srcType**, int **dstType**, const Mat& **kernel**, const Point& **anchor**=Point(-1,-1))

C++: Ptr<BaseFilter_GPU> gpu::**getLinearFilter_GPU**(int **srcType**, int **dstType**, const Mat& **kernel**, const Size& **ksize**, Point **anchor**=Point(-1, -1))

> **Parameters**
>
> > **srcType** – Input image type. `CV_8UC1` and `CV_8UC4` types are supported.
> >
> > **dstType** – Output image type. The same type as `src` is supported.
> >
> > **kernel** – 2D array of filter coefficients. Floating-point coefficients will be converted to fixed-point representation before the actual processing.
> >
> > **ksize** – Kernel size.
> >
> > **anchor** – Anchor point. The default value Point(-1, -1) means that the anchor is at the kernel center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

`createLinearFilter()`

gpu::filter2D

Applies the non-separable 2D linear filter to an image.

C++: void gpu::**filter2D**(const GpuMat& **src**, GpuMat& **dst**, int **ddepth**, const Mat& **kernel**, Point **anchor**=Point(-1,-1), Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source image. `CV_8UC1` and `CV_8UC4` source types are supported.
>>
>> **dst** – Destination image. The size and the number of channels is the same as `src` .
>>
>> **ddepth** – Desired depth of the destination image. If it is negative, it is the same as `src.depth()` . It supports only the same depth as the source image depth.
>>
>> **kernel** – 2D array of filter coefficients. This filter works with integers kernels. If `kernel` has a `float` or `double` type, it uses fixed-point arithmetic.
>>
>> **anchor** – Anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor resides within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center.
>>
>> **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

`filter2D()`

gpu::Laplacian

Applies the Laplacian operator to an image.

C++: void gpu::**Laplacian**(const GpuMat& **src**, GpuMat& **dst**, int **ddepth**, int **ksize**=1, double **scale**=1, Stream& **stream**=Stream::Null())

> **Parameters**
>
>> **src** – Source image. `CV_8UC1` and `CV_8UC4` source types are supported.
>>
>> **dst** – Destination image. The size and number of channels is the same as `src` .
>>
>> **ddepth** – Desired depth of the destination image. It supports only the same depth as the source image depth.
>>
>> **ksize** – Aperture size used to compute the second-derivative filters (see `getDerivKernels()`). It must be positive and odd. Only `ksize = 1` and `ksize = 3` are supported.

> **scale** – Optional scale factor for the computed Laplacian values. By default, no scaling is applied (see `getDerivKernels()`).
>
> **stream** – Stream for the asynchronous version.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

See Also:

`Laplacian()`, `gpu::filter2D()`

gpu::getLinearRowFilter_GPU

Creates a primitive row filter with the specified kernel.

C++: Ptr<BaseRowFilter_GPU> gpu::**getLinearRowFilter_GPU**(int **srcType**, int **bufType**, const Mat& **rowKernel**, int **anchor**=-1, int **border-Type**=BORDER_CONSTANT)

> Parameters
>
> > **srcType** – Source array type. Only `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` source types are supported.
> >
> > **bufType** – Intermediate buffer type with as many channels as `srcType` .
> >
> > **rowKernel** – Filter coefficients. Support kernels with `size <= 16` .
> >
> > **anchor** – Anchor position within the kernel. Negative values mean that the anchor is positioned at the aperture center.
> >
> > **borderType** – Pixel extrapolation method. For details, see `borderInterpolate()`. For details on limitations, see below.

There are two versions of the algorithm: NPP and OpenCV.

- NPP version is called when `srcType == CV_8UC1` or `srcType == CV_8UC4` and `bufType == srcType` . Otherwise, the OpenCV version is called. NPP supports only `BORDER_CONSTANT` border type and does not check indices outside the image.

- OpenCV version supports only `CV_32F` buffer depth and `BORDER_REFLECT101` , `BORDER_REPLICATE` , and `BORDER_CONSTANT` border types. It checks indices outside the image.

See Also:

`createSeparableLinearFilter()` .

gpu::getLinearColumnFilter_GPU

Creates a primitive column filter with the specified kernel.

C++: Ptr<BaseColumnFilter_GPU> gpu::**getLinearColumnFilter_GPU**(int **bufType**, int **dstType**, const Mat& **columnKernel**, int **anchor**=-1, int **border-Type**=BORDER_CONSTANT)

> Parameters
>
> > **bufType** – Inermediate buffer type with as many channels as `dstType` .

> > > **dstType** – Destination array type. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` destination types are supported.
> > >
> > > **columnKernel** – Filter coefficients. Support kernels with `size <= 16` .
> > >
> > > **anchor** – Anchor position within the kernel. Negative values mean that the anchor is positioned at the aperture center.
> > >
> > > **borderType** – Pixel extrapolation method. For details, see `borderInterpolate()` . For details on limitations, see below.

There are two versions of the algorithm: NPP and OpenCV.

- NPP version is called when `dstType == CV_8UC1` or `dstType == CV_8UC4` and `bufType == dstType` . Otherwise, the OpenCV version is called. NPP supports only `BORDER_CONSTANT` border type and does not check indices outside the image.

- OpenCV version supports only `CV_32F` buffer depth and `BORDER_REFLECT101` , `BORDER_REPLICATE` , and `BORDER_CONSTANT` border types. It checks indices outside image.

See Also:

`gpu::getLinearRowFilter_GPU()`, `createSeparableLinearFilter()`

gpu::createSeparableLinearFilter_GPU

Creates a separable linear filter engine.

C++: Ptr<FilterEngine_GPU> gpu::**createSeparableLinearFilter_GPU**(int **srcType**, int **dstType**, const Mat& **rowKernel**, const Mat& **columnKernel**, const Point& **anchor**=Point(-1,-1), int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1)

> > Parameters
>
> > > **srcType** – Source array type. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` source types are supported.
> > >
> > > **dstType** – Destination array type. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` destination types are supported.
> > >
> > > **rowKernel** – Horizontal filter coefficients. Support kernels with `size <= 16` .
> > >
> > > **columnKernel** – Vertical filter coefficients. Support kernels with `size <= 16` .
> > >
> > > **anchor** – Anchor position within the kernel. Negative values mean that anchor is positioned at the aperture center.
> > >
> > > **rowBorderType** – Pixel extrapolation method in the vertical direction For details, see `borderInterpolate()`. For details on limitations, see `gpu::getLinearRowFilter_GPU()`, cpp:ocv:func:*gpu::getLinearColumnFilter_GPU*.
> > >
> > > **columnBorderType** – Pixel extrapolation method in the horizontal direction.

See Also:

`gpu::getLinearRowFilter_GPU()`, `gpu::getLinearColumnFilter_GPU()`, `createSeparableLinearFilter()`

---

gpu::sepFilter2D

Applies a separable 2D linear filter to an image.

C++: void gpu::**sepFilter2D**(const GpuMat& **src**, GpuMat& **dst**, int **ddepth**, const Mat& **kernelX**, const Mat& **kernelY**, Point **anchor**=Point(-1,-1), int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1, Stream& **stream**=Stream::Null())

> Parameters

>> **src** – Source image. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_32SC1` , `CV_32FC1` source types are supported.

>> **dst** – Destination image with the same size and number of channels as `src` .

>> **ddepth** – Destination image depth. `CV_8U` , `CV_16S` , `CV_32S` , and `CV_32F` are supported.

>> **kernelX** – Horizontal filter coefficients.

>> **kernelY** – Vertical filter coefficients.

>> **anchor** – Anchor position within the kernel. The default value (`-1, 1`) means that the anchor is at the kernel center.

>> **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.

>> **columnBorderType** – Pixel extrapolation method in the horizontal direction.

>> **stream** – Stream for the asynchronous version.

See Also:

`gpu::createSeparableLinearFilter_GPU()`, `sepFilter2D()`

gpu::createDerivFilter_GPU

Creates a filter engine for the generalized Sobel operator.

C++: Ptr<FilterEngine_GPU> gpu::**createDerivFilter_GPU**(int **srcType**, int **dstType**, int **dx**, int **dy**, int **ksize**, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1)

> Parameters

>> **srcType** – Source image type. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` source types are supported.

>> **dstType** – Destination image type with as many channels as `srcType` , `CV_8U` , `CV_16S` , `CV_32S` , and `CV_32F` depths are supported.

>> **dx** – Derivative order in respect of x.

>> **dy** – Derivative order in respect of y.

>> **ksize** – Aperture size. See `getDerivKernels()` for details.

>> **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.

>> **columnBorderType** – Pixel extrapolation method in the horizontal direction.

See Also:

`gpu::createSeparableLinearFilter_GPU()`, `createDerivFilter()`

gpu::Sobel

Applies the generalized Sobel operator to an image.

C++: void gpu::**Sobel**(const GpuMat& **src**, GpuMat& **dst**, int **ddepth**, int **dx**, int **dy**, int **ksize**=3, double **scale**=1, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` source types are supported.
> >
> > **dst** – Destination image with the same size and number of channels as source image.
> >
> > **ddepth** – Destination image depth. `CV_8U` , `CV_16S` , `CV_32S` , and `CV_32F` are supported.
> >
> > **dx** – Derivative order in respect of x.
> >
> > **dy** – Derivative order in respect of y.
> >
> > **ksize** – Size of the extended Sobel kernel. Possible valies are 1, 3, 5 or 7.
> >
> > **scale** – Optional scale factor for the computed derivative values. By default, no scaling is applied. For details, see `getDerivKernels()` .
> >
> > **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.
> >
> > **columnBorderType** – Pixel extrapolation method in the horizontal direction.
> >
> > **stream** – Stream for the asynchronous version.

See Also:

`gpu::createSeparableLinearFilter_GPU()`, `Sobel()`

gpu::Scharr

Calculates the first x- or y- image derivative using the Scharr operator.

C++: void gpu::**Scharr**(const GpuMat& **src**, GpuMat& **dst**, int **ddepth**, int **dx**, int **dy**, double **scale**=1, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` source types are supported.
> >
> > **dst** – Destination image with the same size and number of channels as `src` has.
> >
> > **ddepth** – Destination image depth. `CV_8U` , `CV_16S` , `CV_32S` , and `CV_32F` are supported.
> >
> > **xorder** – Order of the derivative x.
> >
> > **yorder** – Order of the derivative y.
> >
> > **scale** – Optional scale factor for the computed derivative values. By default, no scaling is applied. See `getDerivKernels()` for details.

> **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.
>
> **columnBorderType** – Pixel extrapolation method in the horizontal direction.
>
> **stream** – Stream for the asynchronous version.

See Also:

`gpu::createSeparableLinearFilter_GPU()`, `Scharr()`

gpu::createGaussianFilter_GPU

Creates a Gaussian filter engine.

C++: Ptr<FilterEngine_GPU> gpu::**createGaussianFilter_GPU**(int **type**, Size **ksize**, double **sigmaX**, double **sigmaY**=0, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1)

> **Parameters**
>
> > **type** – Source and destination image type. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` are supported.
> >
> > **ksize** – Aperture size. See `getGaussianKernel()` for details.
> >
> > **sigmaX** – Gaussian sigma in the horizontal direction. See `getGaussianKernel()` for details.
> >
> > **sigmaY** – Gaussian sigma in the vertical direction. If 0, then `sigmaY` ← `sigmaX` .
> >
> > **rowBorderType** – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.
> >
> > **columnBorderType** – Pixel extrapolation method in the horizontal direction.

See Also:

`gpu::createSeparableLinearFilter_GPU()`, `createGaussianFilter()`

gpu::GaussianBlur

Smooths an image using the Gaussian filter.

C++: void gpu::**GaussianBlur**(const GpuMat& **src**, GpuMat& **dst**, Size **ksize**, double **sigmaX**, double **sigmaY**=0, int **rowBorderType**=BORDER_DEFAULT, int **columnBorderType**=-1, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **src** – Source image. `CV_8UC1` , `CV_8UC4` , `CV_16SC1` , `CV_16SC2` , `CV_16SC3` , `CV_32SC1` , `CV_32FC1` source types are supported.
> >
> > **dst** – Destination image with the same size and type as `src` .
> >
> > **ksize** – Gaussian kernel size. `ksize.width` and `ksize.height` can differ but they both must be positive and odd. If they are zeros, they are computed from `sigmaX` and `sigmaY` .
> >
> > **sigmaX** – Gaussian kernel standard deviation in X direction.

sigmaY – Gaussian kernel standard deviation in 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()` for details. To fully control the result regardless of possible future modification of all this semantics, you are recommended to specify all of `ksize`, `sigmaX`, and `sigmaY`.

rowBorderType – Pixel extrapolation method in the vertical direction. For details, see `borderInterpolate()`.

columnBorderType – Pixel extrapolation method in the horizontal direction.

stream – Stream for the asynchronous version.

See Also:

`gpu::createGaussianFilter_GPU()`, `GaussianBlur()`

gpu::getMaxFilter_GPU

Creates the maximum filter.

C++: Ptr<BaseFilter_GPU> gpu::**getMaxFilter_GPU**(int **srcType**, int **dstType**, const Size& **ksize**, Point **anchor**=Point(-1,-1))

>   **Parameters**

>>   **srcType** – Input image type. Only `CV_8UC1` and `CV_8UC4` are supported.

>>   **dstType** – Output image type. It supports only the same type as the source type.

>>   **ksize** – Kernel size.

>>   **anchor** – Anchor point. The default value (-1) means that the anchor is at the kernel center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

gpu::getMinFilter_GPU

Creates the minimum filter.

C++: Ptr<BaseFilter_GPU> gpu::**getMinFilter_GPU**(int **srcType**, int **dstType**, const Size& **ksize**, Point **anchor**=Point(-1,-1))

>   **Parameters**

>>   **srcType** – Input image type. Only `CV_8UC1` and `CV_8UC4` are supported.

>>   **dstType** – Output image type. It supports only the same type as the source type.

>>   **ksize** – Kernel size.

>>   **anchor** – Anchor point. The default value (-1) means that the anchor is at the kernel center.

Note: This filter does not check out-of-border accesses, so only a proper sub-matrix of a bigger matrix has to be passed to it.

11.11 Camera Calibration and 3D Reconstruction

gpu::StereoBM_GPU

class gpu::**StereoBM_GPU**

Class computing stereo correspondence (disparity map) using the block matching algorithm.

```cpp
class StereoBM_GPU
{
public:
    enum { BASIC_PRESET = 0, PREFILTER_XSOBEL = 1 };

    enum { DEFAULT_NDISP = 64, DEFAULT_WINSZ = 19 };

    StereoBM_GPU();
    StereoBM_GPU(int preset, int ndisparities = DEFAULT_NDISP,
                 int winSize = DEFAULT_WINSZ);

    void operator() (const GpuMat& left, const GpuMat& right,
                     GpuMat& disparity, Stream& stream = Stream::Null());

    static bool checkIfGpuCallReasonable();

    int preset;
    int ndisp;
    int winSize;

    float avergeTexThreshold;

    ...
};
```

The class also performs pre- and post-filtering steps: Sobel pre-filtering (if PREFILTER_XSOBEL flag is set) and low textureness filtering (if averageTexThreshols > 0). If avergeTexThreshold = 0 , low textureness filtering is disabled. Otherwise, the disparity is set to 0 in each point (x, y) , where for the left image

$$\sum HorizontalGradiensInWindow(x, y, winSize) < (winSize \cdot winSize) \cdot avergeTexThreshold$$

This means that the input left image is low textured.

gpu::StereoBM_GPU::StereoBM_GPU

Enables gpu::StereoBM_GPU constructors.

C++: gpu::StereoBM_GPU::**StereoBM_GPU**()

C++: gpu::StereoBM_GPU::**StereoBM_GPU**(int **preset**, int **ndisparities**=DEFAULT_NDISP, int **win-Size**=DEFAULT_WINSZ)

> **Parameters**
>
>> **preset** – Parameter presetting:
>>
>> – **BASIC_PRESET** Basic mode without pre-processing.
>>
>> – **PREFILTER_XSOBEL** Sobel pre-filtering mode.

> > **ndisparities** – Number of disparities. It must be a multiple of 8 and less or equal to 256.
>
> > **winSize** – Block size.

gpu::StereoBM_GPU::operator ()

Enables the stereo correspondence operator that finds the disparity for the specified rectified stereo pair.

C++: void gpu::StereoBM_GPU::**operator()**(const GpuMat& **left**, const GpuMat& **right**, GpuMat& **disparity**, Stream& **stream**=Stream::Null())

> > **Parameters**

> > > **left** – Left image. Only `CV_8UC1` type is supported.

> > > **right** – Right image with the same size and the same type as the left one.

> > > **disparity** – Output disparity map. It is a `CV_8UC1` image with the same size as the input images.

> > > **stream** – Stream for the asynchronous version.

gpu::StereoBM_GPU::checkIfGpuCallReasonable

Uses a heuristic method to estimate whether the current GPU is faster than the CPU in this algorithm. It queries the currently active device.

C++: bool gpu::StereoBM_GPU::**checkIfGpuCallReasonable**()

gpu::StereoBeliefPropagation

class gpu::**StereoBeliefPropagation**

Class computing stereo correspondence using the belief propagation algorithm.

```
class StereoBeliefPropagation
{
public:
    enum { DEFAULT_NDISP  = 64 };
    enum { DEFAULT_ITERS  = 5  };
    enum { DEFAULT_LEVELS = 5  };

    static void estimateRecommendedParams(int width, int height,
        int& ndisp, int& iters, int& levels);

    explicit StereoBeliefPropagation(int ndisp = DEFAULT_NDISP,
        int iters  = DEFAULT_ITERS,
        int levels = DEFAULT_LEVELS,
        int msg_type = CV_32F);
    StereoBeliefPropagation(int ndisp, int iters, int levels,
        float max_data_term, float data_weight,
        float max_disc_term, float disc_single_jump,
        int msg_type = CV_32F);

    void operator()(const GpuMat& left, const GpuMat& right,
                    GpuMat& disparity, Stream& stream = Stream::Null());
    void operator()(const GpuMat& data, GpuMat& disparity, Stream& stream = Stream::Null());
```

```
    int ndisp;

    int iters;
    int levels;

    float max_data_term;
    float data_weight;
    float max_disc_term;
    float disc_single_jump;

    int msg_type;


    ...
};
```

The class implements algorithm described in [Felzenszwalb2006] . It can compute own data cost (using a truncated linear model) or use a user-provided data cost.

Note: StereoBeliefPropagation requires a lot of memory for message storage:

$$width\_step \cdot height \cdot ndisp \cdot 4 \cdot (1 + 0.25)$$

and for data cost storage:

$$width\_step \cdot height \cdot ndisp \cdot (1 + 0.25 + 0.0625 + \cdots + \frac{1}{4^{levels}})$$

width_step is the number of bytes in a line including padding.

---

gpu::StereoBeliefPropagation::StereoBeliefPropagation

Enables the gpu::StereoBeliefPropagation constructors.

C++: gpu::StereoBeliefPropagation::**StereoBeliefPropagation**(int **ndisp**=DEFAULT_NDISP, int **iters**=DEFAULT_ITERS, int **levels**=DEFAULT_LEVELS, int **msg_type**=CV_32F)

C++: gpu::StereoBeliefPropagation::**StereoBeliefPropagation**(int **ndisp**, int **iters**, int **levels**, float **max_data_term**, float **data_weight**, float **max_disc_term**, float **disc_single_jump**, int **msg_type**=CV_32F)

> **Parameters**
>
> > **ndisp** – Number of disparities.
> >
> > **iters** – Number of BP iterations on each level.
> >
> > **levels** – Number of levels.
> >
> > **max_data_term** – Threshold for data cost truncation.
> >
> > **data_weight** – Data weight.
> >
> > **max_disc_term** – Threshold for discontinuity truncation.
> >
> > **disc_single_jump** – Discontinuity single jump.

> **msg_type** – Type for messages. `CV_16SC1` and `CV_32FC1` types are supported.

`StereoBeliefPropagation` uses a truncated linear model for the data cost and discontinuity terms:

$$\mathrm{DataCost} = data\_weight \cdot \min(|I_2 - I_1|, max\_data\_term)$$

$$\mathrm{DiscTerm} = \min(disc\_single\_jump \cdot |f_1 - f_2|, max\_disc\_term)$$

For more details, see [Felzenszwalb2006].

By default, `gpu::StereoBeliefPropagation` uses floating-point arithmetics and the `CV_32FC1` type for messages. But it can also use fixed-point arithmetics and the `CV_16SC1` message type for better performance. To avoid an overflow in this case, the parameters must satisfy the following requirement:

$$10 \cdot 2^{levels-1} \cdot max\_data\_term < SHRT\_MAX$$

gpu::StereoBeliefPropagation::estimateRecommendedParams

Uses a heuristic method to compute the recommended parameters (`ndisp`, `iters` and `levels`) for the specified image size (`width` and `height`).

C++: void gpu::StereoBeliefPropagation::**estimateRecommendedParams**(int **width**, int **height**, int& **ndisp**, int& **iters**, int& **levels**)

gpu::StereoBeliefPropagation::operator ()

Enables the stereo correspondence operator that finds the disparity for the specified rectified stereo pair or data cost.

C++: void gpu::StereoBeliefPropagation::**operator()**(const GpuMat& **left**, const GpuMat& **right**, GpuMat& **disparity**, Stream& **stream**=Stream::Null())

C++: void gpu::StereoBeliefPropagation::**operator()**(const GpuMat& **data**, GpuMat& **disparity**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **left** – Left image. `CV_8UC1` , `CV_8UC3` and `CV_8UC4` types are supported.
> >
> > **right** – Right image with the same size and the same type as the left one.
> >
> > **data** – User-specified data cost, a matrix of `msg_type` type and `Size(<image columns>*ndisp, <image rows>)` size.
> >
> > **disparity** – Output disparity map. If `disparity` is empty, the output type is `CV_16SC1` . Otherwise, the type is retained.
> >
> > **stream** – Stream for the asynchronous version.

gpu::StereoConstantSpaceBP

class gpu::**StereoConstantSpaceBP**

Class computing stereo correspondence using the constant space belief propagation algorithm.

```cpp
class StereoConstantSpaceBP
{
public:
    enum { DEFAULT_NDISP    = 128 };
    enum { DEFAULT_ITERS    = 8   };
    enum { DEFAULT_LEVELS   = 4   };
    enum { DEFAULT_NR_PLANE = 4   };

    static void estimateRecommendedParams(int width, int height,
        int& ndisp, int& iters, int& levels, int& nr_plane);

    explicit StereoConstantSpaceBP(int ndisp = DEFAULT_NDISP,
        int iters   = DEFAULT_ITERS,
        int levels  = DEFAULT_LEVELS,
        int nr_plane = DEFAULT_NR_PLANE,
        int msg_type = CV_32F);
    StereoConstantSpaceBP(int ndisp, int iters, int levels, int nr_plane,
        float max_data_term, float data_weight,
        float max_disc_term, float disc_single_jump,
        int min_disp_th = 0,
        int msg_type = CV_32F);

    void operator()(const GpuMat& left, const GpuMat& right,
                    GpuMat& disparity, Stream& stream = Stream::Null());

    int ndisp;

    int iters;
    int levels;

    int nr_plane;

    float max_data_term;
    float data_weight;
    float max_disc_term;
    float disc_single_jump;

    int min_disp_th;

    int msg_type;

    bool use_local_init_data_cost;

    ...
};
```

The class implements algorithm described in [Yang2010]. `StereoConstantSpaceBP` supports both local minimum and global minimum data cost initialization algoritihms. For more details, see the paper mentioned above. By default, a local algorithm is used. To enable a global algorithm, set `use_local_init_data_cost` to `false` .

gpu::StereoConstantSpaceBP::StereoConstantSpaceBP

Enables the `gpu::StereoConstantSpaceBP` constructors.

C++: `gpu::StereoConstantSpaceBP::`**`StereoConstantSpaceBP`**(int ndisp=DEFAULT_NDISP, int **iters**=DEFAULT_ITERS, int **levels**=DEFAULT_LEVELS, int **nr_plane**=DEFAULT_NR_PLANE, int **msg_type**=CV_32F)

C++: `StereoConstantSpaceBP::`**`StereoConstantSpaceBP`**(int **ndisp**, int **iters**, int **levels**, int **nr_plane**, float **max_data_term**, float **data_weight**, float **max_disc_term**, float **disc_single_jump**, int **min_disp_th**=0, int **msg_type**=CV_32F)

> **Parameters**
>
> > **ndisp** – Number of disparities.
> >
> > **iters** – Number of BP iterations on each level.
> >
> > **levels** – Number of levels.
> >
> > **nr_plane** – Number of disparity levels on the first level.
> >
> > **max_data_term** – Truncation of data cost.
> >
> > **data_weight** – Data weight.
> >
> > **max_disc_term** – Truncation of discontinuity.
> >
> > **disc_single_jump** – Discontinuity single jump.
> >
> > **min_disp_th** – Minimal disparity threshold.
> >
> > **msg_type** – Type for messages. `CV_16SC1` and `CV_32FC1` types are supported.

`StereoConstantSpaceBP` uses a truncated linear model for the data cost and discontinuity terms:

$$DataCost = data\_weight \cdot \min(|I_2 - I_1|, max\_data\_term)$$

$$DiscTerm = \min(disc\_single\_jump \cdot |f_1 - f_2|, max\_disc\_term)$$

For more details, see [Yang2010].

By default, `StereoConstantSpaceBP` uses floating-point arithmetics and the `CV_32FC1` type for messages. But it can also use fixed-point arithmetics and the `CV_16SC1` message type for better perfomance. To avoid an overflow in this case, the parameters must satisfy the following requirement:

$$10 \cdot 2^{levels-1} \cdot max\_data\_term < SHRT\_MAX$$

gpu::StereoConstantSpaceBP::estimateRecommendedParams

Uses a heuristic method to compute parameters (ndisp, iters, levelsand nrplane) for the specified image size (widthand height).

C++: void `gpu::StereoConstantSpaceBP::`**`estimateRecommendedParams`**(int **width**, int **height**, int& **ndisp**, int& **iters**, int& **levels**, int& **nr_plane**)

gpu::StereoConstantSpaceBP::operator ()

Enables the stereo correspondence operator that finds the disparity for the specified rectified stereo pair.

C++: void gpu::StereoConstantSpaceBP::**operator()**(const GpuMat& **left**, const GpuMat& **right**, GpuMat& **disparity**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **left** – Left image. `CV_8UC1` , `CV_8UC3` and `CV_8UC4` types are supported.
> >
> > **right** – Right image with the same size and the same type as the left one.
> >
> > **disparity** – Output disparity map. If `disparity` is empty, the output type is `CV_16SC1` . Otherwise, the output type is `disparity.type()` .
> >
> > **stream** – Stream for the asynchronous version.

gpu::DisparityBilateralFilter

class gpu::**DisparityBilateralFilter**

Class refinining a disparity map using joint bilateral filtering.

```cpp
class CV_EXPORTS DisparityBilateralFilter
{
public:
    enum { DEFAULT_NDISP  = 64 };
    enum { DEFAULT_RADIUS = 3 };
    enum { DEFAULT_ITERS  = 1 };

    explicit DisparityBilateralFilter(int ndisp = DEFAULT_NDISP,
        int radius = DEFAULT_RADIUS, int iters = DEFAULT_ITERS);

    DisparityBilateralFilter(int ndisp, int radius, int iters,
        float edge_threshold, float max_disc_threshold,
        float sigma_range);

    void operator()(const GpuMat& disparity, const GpuMat& image,
                    GpuMat& dst, Stream& stream = Stream::Null());

    ...
};
```

The class implements [Yang2010] algorithm.

gpu::DisparityBilateralFilter::DisparityBilateralFilter

Enables the gpu::DisparityBilateralFilter constructors.

C++: gpu::DisparityBilateralFilter::**DisparityBilateralFilter**(int **ndisp**=DEFAULT_NDISP, int **radius**=DEFAULT_RADIUS, int **iters**=DEFAULT_ITERS)

C++: gpu::DisparityBilateralFilter::**DisparityBilateralFilter**(int **ndisp**, int **radius**, int **iters**, float **edge_threshold**, float **max_disc_threshold**, float **sigma_range**)

Parameters

ndisp – Number of disparities.

radius – Filter radius.

iters – Number of iterations.

edge_threshold – Threshold for edges.

max_disc_threshold – Constant to reject outliers.

sigma_range – Filter range.

gpu::DisparityBilateralFilter::operator ()

Refines a disparity map using joint bilateral filtering.

C++: void gpu::DisparityBilateralFilter::**operator()**(const GpuMat& **disparity**, const GpuMat& **image**, GpuMat& **dst**, Stream& **stream**=Stream::Null())

Parameters

disparity – Input disparity map. `CV_8UC1` and `CV_16SC1` types are supported.

image – Input image. `CV_8UC1` and `CV_8UC3` types are supported.

dst – Destination disparity map. It has the same size and type as `disparity` .

stream – Stream for the asynchronous version.

gpu::drawColorDisp

Colors a disparity image.

C++: void gpu::**drawColorDisp**(const GpuMat& **src_disp**, GpuMat& **dst_disp**, int **ndisp**, Stream& **stream**=Stream::Null())

Parameters

src_disp – Source disparity image. `CV_8UC1` and `CV_16SC1` types are supported.

dst_disp – Output disparity image. It has the same size as `src_disp` . The type is `CV_8UC4` in BGRA format (alpha = 255).

ndisp – Number of disparities.

stream – Stream for the asynchronous version.

This function draws a colored disparity map by converting disparity values from `[0..ndisp)` interval first to HSV color space (where different disparity values correspond to different hues) and then converting the pixels to RGB for visualization.

gpu::reprojectImageTo3D

Reprojects a disparity image to 3D space.

C++: void gpu::**reprojectImageTo3D**(const GpuMat& **disp**, GpuMat& **xyzw**, const Mat& **Q**, Stream& **stream**=Stream::Null())

Parameters

disp – Input disparity image. `CV_8U` and `CV_16S` types are supported.

xyzw – Output 4-channel floating-point image of the same size as `disp` . Each element of `xyzw(x,y)` contains 3D coordinates `(x,y,z,1)` of the point `(x,y)` , computed from the disparity map.

Q – 4×4 perspective transformation matrix that can be obtained via `stereoRectify()` .

stream – Stream for the asynchronous version.

See Also:

`reprojectImageTo3D()`

gpu::solvePnPRansac

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

C++: void gpu::**solvePnPRansac**(const Mat& **object**, const Mat& **image**, const Mat& **camera_mat**, const Mat& **dist_coef**, Mat& **rvec**, Mat& **tvec**, bool **use_extrinsic_guess**=false, int **num_iters**=100, float **max_dist**=8.0, int **min_inlier_count**=100, vector<int>* **inliers**=NULL)

    Parameters

        object – Single-row matrix of object points.

        image – Single-row matrix of image points.

        camera_mat – 3x3 matrix of intrinsic camera parameters.

        dist_coef – Distortion coefficients. See `undistortPoints()` for details.

        rvec – Output 3D rotation vector.

        tvec – Output 3D translation vector.

        use_extrinsic_guess – Flag to indicate that the function must use `rvec` and `tvec` as an initial transformation guess. It is not supported for now.

        num_iters – Maximum number of RANSAC iterations.

        max_dist – Euclidean distance threshold to detect whether point is inlier or not.

        min_inlier_count – Flag to indicate that the function must stop if greater or equal number of inliers is achieved. It is not supported for now.

        inliers – Output vector of inlier indices.

See Also:

`solvePnPRansac()`

11.12 Video Analysis

gpu::BroxOpticalFlow

class gpu::**BroxOpticalFlow**

Class computing the optical flow for two images using Brox et al Optical Flow algorithm ([Brox2004]).

```
class BroxOpticalFlow
{
public:
    BroxOpticalFlow(float alpha_, float gamma_, float scale_factor_, int inner_iterations_, int outer_iterations_, int

    //! Compute optical flow
    //! frame0 - source frame (supports only CV_32FC1 type)
    //! frame1 - frame to track (with the same size and type as frame0)
    //! u      - flow horizontal component (along x axis)
    //! v      - flow vertical component (along y axis)
    void operator ()(const GpuMat& frame0, const GpuMat& frame1, GpuMat& u, GpuMat& v, Stream& stream = Stream::Null()

    //! flow smoothness
    float alpha;

    //! gradient constancy importance
    float gamma;

    //! pyramid scale factor
    float scale_factor;

    //! number of lagged non-linearity iterations (inner loop)
    int inner_iterations;

    //! number of warping iterations (number of pyramid levels)
    int outer_iterations;

    //! number of linear system solver iterations
    int solver_iterations;

    GpuMat buf;
};
```

gpu::interpolateFrames

Interpolate frames (images) using provided optical flow (displacement field).

C++: void gpu::**interpolateFrames**(const GpuMat& **frame0**, const GpuMat& **frame1**, const GpuMat& **fu**, const GpuMat& **fv**, const GpuMat& **bu**, const GpuMat& **bv**, float **pos**, GpuMat& **newFrame**, GpuMat& **buf**, Stream& **stream**=Stream::Null())

> **Parameters**
>
> > **frame0** – First frame (32-bit floating point images, single channel).
> >
> > **frame1** – Second frame. Must have the same type and size as `frame0` .
> >
> > **fu** – Forward horizontal displacement.
> >
> > **fv** – Forward vertical displacement.
> >
> > **bu** – Backward horizontal displacement.
> >
> > **bv** – Backward vertical displacement.
> >
> > **pos** – New frame position.
> >
> > **newFrame** – Output image.

buf – Temporary buffer, will have width x 6*height size, CV_32FC1 type and contain 6 GpuMat: occlusion masks for first frame, occlusion masks for second, interpolated forward horizontal flow, interpolated forward vertical flow, interpolated backward horizontal flow, interpolated backward vertical flow.

stream – Stream for the asynchronous version.

BIBLIOGRAPHY

[Arthur2007] Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding, Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, 2007

[Borgefors86] Borgefors, Gunilla, *Distance transformations in digital images*. Comput. Vision Graph. Image Process. 34 3, pp 344–371 (1986)

[Felzenszwalb04] Felzenszwalb, Pedro F. and Huttenlocher, Daniel P. *Distance Transforms of Sampled Functions*, TR2004-1963, TR2004-1963 (2004)

[Meyer92] Meyer, F. *Color Image Segmentation*, ICIP92, 1992

[Telea04] Alexandru Telea, *An Image Inpainting Technique Based on the Fast Marching Method*. Journal of Graphics, GPU, and Game Tools 9 1, pp 23-34 (2004)

[RubnerSept98]   25. Rubner. C. Tomasi, L.J. Guibas. *The Earth Mover's Distance as a Metric for Image Retrieval*. Technical Report STAN-CS-TN-98-86, Department of Computer Science, Stanford University, September 1998.

[Iivarinen97] Jukka Iivarinen, Markus Peura, Jaakko Srel, and Ari Visa. *Comparison of Combined Shape Descriptors for Irregular Objects*, 8th British Machine Vision Conference, BMVC'97. http://www.cis.hut.fi/research/IA/paper/publications/bmvc97/bmvc97.html

[Fitzgibbon95] Andrew W. Fitzgibbon, R.B.Fisher. *A Buyer's Guide to Conic Fitting*. Proc.5th British Machine Vision Conference, Birmingham, pp. 513-522, 1995.

[Hu62]   13. Hu. *Visual Pattern Recognition by Moment Invariants*, IRE Transactions on Information Theory, 8:2, pp. 179-187, 1962.

[Sklansky82] Sklansky, J., *Finding the Convex Hull of a Simple Polygon*. PRL 1 $number, pp 79-83 (1982)

[Suzuki85] Suzuki, S. and Abe, K., *Topological Structural Analysis of Digitized Binary Images by Border Following*. CVGIP 30 1, pp 32-46 (1985)

[TehChin89] Teh, C.H. and Chin, R.T., *On the Detection of Dominant Points on Digital Curve*. PAMI 11 8, pp 859-872 (1989)

[Canny86]   10. Canny. *A Computational Approach to Edge Detection*, IEEE Trans. on Pattern Analysis and Machine Intelligence, 8(6), pp. 679-698 (1986).

[Matas00] Matas, J. and Galambos, C. and Kittler, J.V., *Robust Detection of Lines Using the Progressive Probabilistic Hough Transform*. CVIU 78 1, pp 119-137 (2000)

[Shi94]   10. Shi and C. Tomasi. *Good Features to Track*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 593-600, June 1994.

[Yuen90] Yuen, H. K. and Princen, J. and Illingworth, J. and Kittler, J., *Comparative study of Hough transform methods for circle finding*. Image Vision Comput. 8 1, pp 71–77 (1990)

[Bouguet00] Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker.

[Bradski98] Bradski, G.R. "Computer Vision Face Tracking for Use in a Perceptual User Interface", Intel, 1998

[Bradski00] Davis, J.W. and Bradski, G.R. "Motion Segmentation and Pose Recognition with Motion History Gradients", WACV00, 2000

[Davis97] Davis, J.W. and Bobick, A.F. "The Representation and Recognition of Action Using Temporal Templates", CVPR97, 1997

[Farneback2003] Gunnar Farneback, Two-frame motion estimation based on polynomial expansion, Lecture Notes in Computer Science, 2003, (2749), , 363-370.

[Horn81] Berthold K.P. Horn and Brian G. Schunck. Determining Optical Flow. Artificial Intelligence, 17, pp. 185-203, 1981.

[Lucas81] Lucas, B., and Kanade, T. An Iterative Image Registration Technique with an Application to Stereo Vision, Proc. of 7th International Joint Conference on Artificial Intelligence (IJCAI), pp. 674-679.

[Welch95] Greg Welch and Gary Bishop "An Introduction to the Kalman Filter", 1995

[BouguetMCT] J.Y.Bouguet. MATLAB calibration tool. http://www.vision.caltech.edu/bouguetj/calib_doc/

[Hartley99] Hartley, R.I., Theory and Practice of Projective Rectification. IJCV 35 2, pp 115-127 (1999)

[Zhang2000] 26. Zhang. A Flexible New Technique for Camera Calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(11):1330-1334, 2000.

[Agrawal08] Agrawal, M. and Konolige, K. and Blas, M.R. "CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching", ECCV08, 2008

[Bay06] Bay, H. and Tuytelaars, T. and Van Gool, L. "SURF: Speeded Up Robust Features", 9th European Conference on Computer Vision, 2006

[Viola01] Paul Viola and Michael J. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. IEEE CVPR, 2001. The paper is available online at http://www.ai.mit.edu/people/viola/

[Lienhart02] Rainer Lienhart and Jochen Maydt. An Extended Set of Haar-like Features for Rapid Object Detection. IEEE ICIP 2002, Vol. 1, pp. 900-903, Sep. 2002. This paper, as well as the extended technical report, can be retrieved at http://www.lienhart.de/Publications/publications.html

[Felzenszwalb2010] Felzenszwalb, P. F. and Girshick, R. B. and McAllester, D. and Ramanan, D. *Object Detection with Discriminatively Trained Part Based Models*. PAMI, vol. 32, no. 9, pp. 1627-1645, September 2010

[Fukunaga90] 11. Fukunaga. *Introduction to Statistical Pattern Recognition*. second ed., New York: Academic Press, 1990.

[Burges98] 3. 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] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. (http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf)

[Breiman84] Breiman, L., Friedman, J. Olshen, R. and Stone, C. (1984), *Classification and Regression Trees*, Wadsworth.

[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.

[BackPropWikipedia] http://en.wikipedia.org/wiki/Backpropagation. Wikipedia article about the back-propagation algorithm.

[LeCun98] 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.

[RPROP93] 13. Riedmiller and H. Braun, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*, Proc. ICNN, San Francisco (1993).

[Dalal2005] Navneet Dalal and Bill Triggs. *Histogram of oriented gradients for human detection*. 2005.

[Felzenszwalb2006] Pedro F. Felzenszwalb algorithm [Pedro F. Felzenszwalb and Daniel P. Huttenlocher. *Efficient belief propagation for early vision*. International Journal of Computer Vision, 70(1), October 2006

[Yang2010] 17. Yang, L. Wang, and N. Ahuja. *A constant-space belief propagation algorithm for stereo matching*. In CVPR, 2010.

[Brox2004] 20. Brox, A. Bruhn, N. Papenberg, J. Weickert. *High accuracy optical flow estimation based on a theory for warping*. ECCV 2004.