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\leq 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 thecvCreateMat 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:
  2. void process\_video\_frame(**const** unsigned char\* pixels,
  3. int width, int height, int step)
  4. {
  5. Mat img(height, width, CV\_8UC3, pixels, step);
  6. GaussianBlur(img, img, Size(7,7), 1.5, 1.5);
  7. }
  8. Quickly initialize small matrices and/or get a super-fast element access.
  9. double m[3][3] = {{a, b, c}, {d, e, f}, {g, h, i}};
  10. 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\_**](http://docs.opencv.org/modules/core/doc/basic_structures.html#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() .

**Note**

* An example demonstrating the serial out capabilities of cv::Mat can be found at opencv\_source\_code/samples/cpp/cout\_mat.cpp

Mat::row

Creates a matrix header for the specified matrix row.

**C++:**Mat Mat::**row**(int **y**) const

|  |  |
| --- | --- |
| **Parameters:** | * **y** – 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:

**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:

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()**](http://docs.opencv.org/modules/core/doc/basic_structures.html#void Mat::copyTo(OutputArray m) const) method:

Mat A;

...

*// works, but looks a bit obscure.*

A.row(i) = A.row(j) + 0;

*// this is a bit longer, 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 **x**) const

|  |  |
| --- | --- |
| **Parameters:** | * **x** – 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()**](http://docs.opencv.org/modules/core/doc/basic_structures.html#Mat Mat::row(int y) const) description.

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\* **sz**, int **type**)[¶](http://docs.opencv.org/modules/core/doc/basic_structures.html#static MatExpr Mat::zeros(int ndims, const int* sz, int type))

|  |  |
| --- | --- |
| **Parameters:** | * **ndims** – Array dimensionality. * **rows** – Number of rows. * **cols** – Number of columns. * **size** – Alternative to the matrix size specification Size(cols, rows) . * **sz** – 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::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[¶](http://docs.opencv.org/modules/core/doc/basic_structures.html" \l "template<typename T> const T& Mat::at(int i, int j) const" \o "Permalink to this definition)

**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);

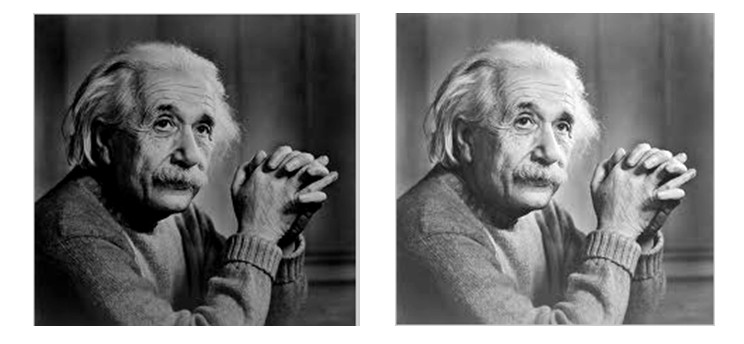
**for**(int i = 0; i < H.rows; i++)

**for**(int j = 0; j < H.cols; j++)

H.at<double>(i,j)=1./(i+j+1);

Brightness

Brightness is an attribute of visual perception in which a source appears to be radiating or reflecting light. In other words, brightness is the perception elicited by the luminance of a visual target. This is a subjective attribute/property of an object being observed.

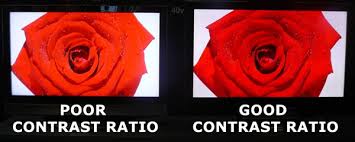


New (R,G,B)=Old(R+Delta, G+Delta, B+Delta)

Contrast

Contrast in visual perception is the difference in appearance of two or more parts of a field seen simultaneously or successively (hence: brightness contrast, lightness contrast, color contrast, simultaneous contrast, successive contrast, etc.).

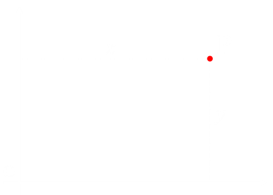
Contrast in physics is a quantity intended to correlate with the perceived brightness contrast, usually defined by one of a number of formulae (see below) which involve e.g. the luminance of the stimuli considered, for example: ΔL/L near the luminance threshold (known as Weber contrast), or LH/LL for much higher luminance.



New (R,G,B)=Old(R\*Alpha, G\*Alpha, B\*Alpha)

Rotation

Each pixel has a coordinate pair (x,y) describing its position on two orthogonal axes from defined origin O. It is around this origin we’re going to rotate our image.



What we need to do is take the RGB values at every (x,y) location, rotate it as needed, and then write these values in the new location.

|  |  |
| --- | --- |
| To complete the rotation we need to write the RGB value for each pixel in the new location.  The *(x,y)* coordinates relative to the axes are the same; what we have done is rotate the coordinate frame of reference by an angle *θ*. | http://www.datagenetics.com/blog/august32013/g22.png |

|  |  |
| --- | --- |
| http://www.datagenetics.com/blog/august32013/g32.png | Using some High School geometry we can work out the relationship between the source coordinates *(x,y)* and the destination coordinates*(x\*,y\*)*  http://www.datagenetics.com/blog/august32013/2d.png  It is convenient to write in a matrix format as shown above. You might remember this from school. |

### You spin me right round …

|  |  |
| --- | --- |
| http://www.datagenetics.com/blog/august32013/nbs.png | Here is our source image. By translating the coordinate frame we can place the origin at the centroid, and it is around this we will rotate it.  http://www.datagenetics.com/blog/august32013/eq.png  The matrix expands out as above.  We can loop over the image for each *(x,y)* coordinate and find its new destination. |

|  |  |
| --- | --- |
| http://www.datagenetics.com/blog/august32013/out.png | Here is the result with a rotation of *θ* = 15°  It's certainly worked. The image has been rotated, but what are all those dots?  This is a problem called *aliasing.*  It's the same reason that we see jagged staircases on lines drawn on low resolution screens at angles other than horizontal and vertical. Raster screens are digital, and pixel boundaries are at quantized locations. |

|  |  |
| --- | --- |
| Multiplying by *sines* and *cosines* on the integer coordinates of the source image gives real number results, and these have to be rounded back to integers again to be plotted. Sometimes this number rounding means the same destination location is addressed more than once, and sometimes certain pixels are missed completely. When the pixels are missed, the background shows through. This is why there are holes. | http://www.datagenetics.com/blog/august32013/alias.png |

**Aliasing**

The aliasing problem gets worse when angles are closesr to the diagonals. Here are a few examples of images at different rotations:

|  |  |
| --- | --- |
| 0°  http://www.datagenetics.com/blog/august32013/t1.png | 35°  http://www.datagenetics.com/blog/august32013/t2.png |
|  | |
| 45°  http://www.datagenetics.com/blog/august32013/t3.png | 170°  http://www.datagenetics.com/blog/august32013/t4.png |
|  |  |

What can we do about this? There are a variety of solutions. One of them is to oversample the source image. We can pretend that each of the original source pixels is really a grid of n x n smaller pixels (all of the same color), and calculate the destination coordinates of each of these subpixels and plot these.

A more refined way is called Area Mapping. For this, you invert the problem, and for each destination pixel, you find which four partial source pixels that it was created from. The color for the destination is calculated by the area-weighted average of the four source pixels (The source pixels that contribute more to the destination pixel have a greater influence on its color). This algorithm not only ensures that there are no gaps in the destination, but also appropriately averages the colors (ensuring both a smoother image and also keeping the average brightness of the rotated image constant).

However, there is a more elegant method, and this is the method that was used many years ago when computing power (and memory) were at a premium and every processor cycle worth its weight in gold. It is called the three shear rotation method. It's so clever that it's worth sharing in full detail.

Histogram

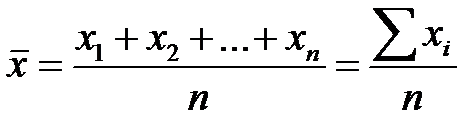
An "image histogram" is a type of [histogram](http://en.wikipedia.org/wiki/Histogram) that acts as a [graphical representation](http://en.wikipedia.org/wiki/Graphical_representation) of the [tonal](http://en.wikipedia.org/wiki/Lightness_(color)) distribution in a [digital image](http://en.wikipedia.org/wiki/Digital_image).[[1]](http://en.wikipedia.org/wiki/Image_histogram#cite_note-sutton-1) It plots the number of [pixels](http://en.wikipedia.org/wiki/Pixels) for each tonal value. By looking at the histogram for a specific image a viewer will be able to judge the entire tonal distribution at a glance.

Image histograms are present on many modern [digital cameras](http://en.wikipedia.org/wiki/Digital_cameras). Photographers can use them as an aid to show the distribution of tones captured, and whether image detail has been lost to blown-out highlights or blacked-out shadows.[[2]](http://en.wikipedia.org/wiki/Image_histogram#cite_note-2)

The [horizontal axis](http://en.wikipedia.org/wiki/Horizontal_axis) of the [graph](http://en.wikipedia.org/wiki/Graphics) represents the tonal variations, while the [vertical axis](http://en.wikipedia.org/wiki/Vertical_axis) represents the number of pixels in that particular tone.[[1]](http://en.wikipedia.org/wiki/Image_histogram#cite_note-sutton-1) The left side of the horizontal axis represents the black and dark areas, the middle represents medium grey and the right hand side represents light and pure white areas. The vertical axis represents the size of the area that is captured in each one of these zones. Thus, the histogram for a very dark image will have the majority of its data points on the left side and center of the graph. Conversely, the histogram for a very bright image with few dark areas and/or shadows will have most of its data points on the right side and center of the graph.

Basically it is the count of pixels having the same r,g,b intensities.

Mean :



Standard Deviation:



