Note: Comma-separated initializers and probably some other operations may require additional explicit `Mat()` or `Mat_<T>()` constructor 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 lowConstrastMask = 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.

roi – Region of interest.

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` refer-

ence 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:

```
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()` 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 **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 data type 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 data type. `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 as 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.

adjustROI forces the adjusted ROI to be inside of the parent matrix that is boundaries of the adjusted ROI are constrained by boundaries of the parent matrix. For example, if the submatrix A is located in the first row of a parent matrix and you called A.adjustROI(2, 2, 2, 2) then A will not be increased in the upward direction.

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:

```
// 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.

```
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 elements.

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:

```
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` .

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