Image Filtering

Image processing

Enumerations

enum	**cv::MorphShapes** {
	cv::MORPH_RECT = 0,
	cv::MORPH_CROSS = 1,
	cv::MORPH_ELLIPSE = 2
	}
	shape of the structuring element More...

enum	**cv::MorphTypes** {
	cv::MORPH_ERODE = 0,
	cv::MORPH_DILATE = 1,
	cv::MORPH_OPEN = 2,
	cv::MORPH_CLOSE = 3,
	cv::MORPH_GRADIENT = 4,
	cv::MORPH_TOPHAT = 5,
	cv::MORPH_BLACKHAT = 6,
	cv::MORPH_HITMISS = 7
	}
	type of morphological operation More...

Functions

void	**cv::bilateralFilter** (**InputArray** src, **OutputArray** dst, int d, double sigmaColor, double sigmaSpace, int borderType=**BORDER_DEFAULT**) Applies the bilateral filter to an image. More...
void	**cv::blur** (**InputArray** src, **OutputArray** dst, **Size** ksize, **Point** anchor=**Point**(-1,-1), int borderType=**BORDER_DEFAULT**) Blurs an image using the normalized box filter. More...
void	**cv::boxFilter** (**InputArray** src, **OutputArray** dst, int ddepth, **Size** ksize, **Point** anchor=**Point**(-1,-1), bool **normalize**=true, int borderType=**BORDER_DEFAULT**) Blurs an image using the box filter. More...
void	**cv::buildPyramid** (**InputArray** src, **OutputArrayOfArrays** dst, int maxlevel, int borderType=**BORDER_DEFAULT**) Constructs the Gaussian pyramid for an image. More...
void	**cv::dilate** (**InputArray** src, **OutputArray** dst, **InputArray** kernel, **Point** anchor=**Point**(-1,-1), int iterations=1, int borderType=**BORDER_CONSTANT**, const **Scalar** &borderValue=**morphologyDefaultBorderValue**()) Dilates an image by using a specific structuring element. More...
void	**cv::erode** (**InputArray** src, **OutputArray** dst, **InputArray** kernel, **Point** anchor=**Point**(-1,-1), int iterations=1, int borderType=**BORDER_CONSTANT**, const **Scalar** &borderValue=**morphologyDefaultBorderValue**()) Erodes an image by using a specific structuring element. More...
void	**cv::filter2D** (**InputArray** src, **OutputArray** dst, int ddepth, **InputArray** kernel, **Point** anchor=**Point**(-1,-1), double delta=0, int borderType=**BORDER_DEFAULT**) Convolves an image with the kernel. More...
void	**cv::GaussianBlur** (**InputArray** src, **OutputArray** dst, **Size** ksize, double sigmaX, double sigmaY=0, int borderType=**BORDER_DEFAULT**) Blurs an image using a Gaussian filter. More...
void	**cv::getDerivKernels** (**OutputArray** kx, **OutputArray** ky, int dx, int dy, int ksize, bool **normalize**=false, int ktype=**CV_32F**) Returns filter coefficients for computing spatial image derivatives. More...
Mat	**cv::getGaborKernel** (**Size** ksize, double sigma, double theta, double lambd, double gamma, double psi=**CV_PI** *0.5, int ktype=**CV_64F**) Returns Gabor filter coefficients. More...
Mat	**cv::getGaussianKernel** (int ksize, double sigma, int ktype=**CV_64F**) Returns Gaussian filter coefficients. More...
Mat	**cv::getStructuringElement** (int shape, **Size** ksize, **Point** anchor=**Point**(-1,-1)) Returns a structuring element of the specified size and shape for morphological operations. More...
void	**cv::Laplacian** (**InputArray** src, **OutputArray** dst, int ddepth, int ksize=1, double scale=1, double delta=0, int borderType=**BORDER_DEFAULT**) Calculates the Laplacian of an image. More...
void	**cv::medianBlur** (**InputArray** src, **OutputArray** dst, int ksize) Blurs an image using the median filter. More...
static **Scalar**	**cv::morphologyDefaultBorderValue** () returns "magic" border value for erosion and dilation. It is automatically transformed to **Scalar::all**(-DBL_MAX) for dilation. More...

void	cv::morphologyEx (InputArray src, OutputArray dst, int op, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar &borderValue=morphologyDefaultBorderValue()) Performs advanced morphological transformations. More...
void	cv::pyrDown (InputArray src, OutputArray dst, const Size &dstsize=Size(), int borderType=BORDER_DEFAULT) Blurs an image and downsamples it. More...
void	cv::pyrMeanShiftFiltering (InputArray src, OutputArray dst, double sp, double sr, int maxLevel=1, TermCriteria termcrit=TermCriteria(TermCriteria::MAX_ITER+TermCriteria::EPS, 5, 1)) Performs initial step of meanshift segmentation of an image. More...
void	cv::pyrUp (InputArray src, OutputArray dst, const Size &dstsize=Size(), int borderType=BORDER_DEFAULT) Upsamples an image and then blurs it. More...
void	cv::Scharr (InputArray src, OutputArray dst, int ddepth, int dx, int dy, double scale=1, double delta=0, int borderType=BORDER_DEFAULT) Calculates the first x- or y- image derivative using Scharr operator. More...
void	cv::sepFilter2D (InputArray src, OutputArray dst, int ddepth, InputArray kernelX, InputArray kernelY, Point anchor=Point(-1,-1), double delta=0, int borderType=BORDER_DEFAULT) Applies a separable linear filter to an image. More...
void	cv::Sobel (InputArray src, OutputArray dst, int ddepth, int dx, int dy, int ksize=3, double scale=1, double delta=0, int borderType=BORDER_DEFAULT) Calculates the first, second, third, or mixed image derivatives using an extended Sobel operator. More...
void	cv::spatialGradient (InputArray src, OutputArray dx, OutputArray dy, int ksize=3, int borderType=BORDER_DEFAULT) Calculates the first order image derivative in both x and y using a Sobel operator. More...
void	cv::sqrBoxFilter (InputArray _src, OutputArray _dst, int ddepth, Size ksize, Point anchor=Point(-1, -1), bool normalize=true, int borderType=BORDER_DEFAULT) Calculates the normalized sum of squares of the pixel values overlapping the filter. More...

Detailed Description

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 BorderTypes

Depth combinations

Input depth (src.depth())	Output depth (ddepth)
CV_8U	-1/CV_16S/CV_32F/CV_64F
CV_16U/CV_16S	-1/CV_32F/CV_64F
CV_32F	-1/CV_32F/CV_64F
CV_64F	-1/CV_64F

Note

when ddepth=-1, the output image will have the same depth as the source.

Enumeration Type Documentation

§ MorphShapes

enum **cv::MorphShapes**

shape of the structuring element

Enumerator	
MORPH_RECT Python: cv.MORPH_RECT	a rectangular structuring element: $$E_{ij} = 1$$
MORPH_CROSS Python: cv.MORPH_CROSS	a cross-shaped structuring element: $$E_{ij} = \begin{cases} 1 & \text{if i=\texttt{anchor.y} or j=\texttt{anchor.x}} \\ 0 & \text{otherwise} \end{cases}$$
MORPH_ELLIPSE Python: cv.MORPH_ELLIPSE	an elliptic structuring element, that is, a filled ellipse inscribed into the rectangle Rect(0, 0, esize.width, 0.esize.height)

§ MorphTypes

enum **cv::MorphTypes**

type of morphological operation

Enumerator	
MORPH_ERODE Python: cv.MORPH_ERODE	see **erode**
MORPH_DILATE Python: cv.MORPH_DILATE	see **dilate**
MORPH_OPEN Python: cv.MORPH_OPEN	an opening operation $$\texttt{dst} = \mathrm{open}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\mathrm{erode}(\texttt{src}, \texttt{element}))$$
MORPH_CLOSE Python: cv.MORPH_CLOSE	a closing operation $$\texttt{dst} = \mathrm{close}(\texttt{src}, \texttt{element}) = \mathrm{erode}(\mathrm{dilate}(\texttt{src}, \texttt{element}))$$
MORPH_GRADIENT Python: cv.MORPH_GRADIENT	a morphological gradient $$\texttt{dst} = \mathrm{morph\_grad}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\texttt{src}, \texttt{element}) - \mathrm{erode}(\texttt{src}, \texttt{element})$$
MORPH_TOPHAT Python: cv.MORPH_TOPHAT	"top hat" $$\texttt{dst} = \mathrm{tophat}(\texttt{src}, \texttt{element}) = \texttt{src} - \mathrm{open}(\texttt{src}, \texttt{element})$$
MORPH_BLACKHAT Python: cv.MORPH_BLACKHAT	"black hat" $$\texttt{dst} = \mathrm{blackhat}(\texttt{src}, \texttt{element}) = \mathrm{close}(\texttt{src}, \texttt{element}) - \texttt{src}$$
MORPH_HITMISS Python: cv.MORPH_HITMISS	"hit or miss" .- Only supported for CV_8UC1 binary images. A tutorial can be found in the documentation

Function Documentation

§ bilateralFilter()

```
void cv::bilateralFilter ( InputArray   src,
                           OutputArray  dst,
                           int          d,
                           double       sigmaColor,
                           double       sigmaSpace,
                           int          borderType = BORDER_DEFAULT
                         )
```

Python:

```
dst = cv.bilateralFilter( src, d, sigmaColor, sigmaSpace[, dst[, borderType]] )
```

Applies the bilateral filter to an image.

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.

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.
borderType	border mode used to extrapolate pixels outside of the image, see **BorderTypes**

Examples:

samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp.

§ blur()

```
void cv::blur ( InputArray   src,
                OutputArray  dst,
                Size         ksize,
                Point        anchor = Point(-1,-1),
                int          borderType = BORDER_DEFAULT
              )
```

Python:

```
dst = cv.blur( src, ksize[, dst[, anchor[, borderType]]] )
```

Blurs an image using the normalized box filter.

The function smooths an image using the kernel:

$$\texttt{K} = \frac{1}{\texttt{ksize.width*ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \hdotsfor{6} \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

The call `blur(src, dst, ksize, anchor, borderType)` is equivalent to `boxFilter(src, dst, src.type(), anchor, true, borderType)`.

Parameters

src	input image; it can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output image of the same size and type as src.
ksize	blurring kernel size.
anchor	anchor point; 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, see **BorderTypes**

See also

boxFilter, **bilateralFilter**, **GaussianBlur**, **medianBlur**

Examples:

samples/cpp/edge.cpp, **samples/cpp/laplace.cpp**, and **samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp**.

§ boxFilter()

```
void cv::boxFilter ( InputArray   src,
                     OutputArray  dst,
                     int          ddepth,
                     Size         ksize,
                     Point        anchor = Point(-1,-1),
                     bool         normalize = true,
                     int          borderType = BORDER_DEFAULT
                   )
```

Python:

```
dst = cv.boxFilter( src, ddepth, ksize[, dst[, anchor[, normalize[, borderType]]]] )
```

Blurs an image using the box filter.

The function smooths an image using the kernel:

$$\texttt{K} = \alpha \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \hdotsfor{6} \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

where

$$\alpha = \begin{cases} \frac{1}{\texttt{ksize.width*ksize.height}} & \text{when \texttt{normalize=true}} \\ 1 & \text{otherwise} \end{cases}$$

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**.

Parameters

src	input image.
dst	output image of the same size and type as src.
ddepth	the output image depth (-1 to use **src.depth()**).
ksize	blurring kernel size.
anchor	anchor point; 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, see **BorderTypes**

See also

blur, **bilateralFilter**, **GaussianBlur**, **medianBlur**, **integral**

§ buildPyramid()

```
void cv::buildPyramid ( InputArray          src,
                        OutputArrayOfArrays dst,
                        int                 maxlevel,
                        int                 borderType = BORDER_DEFAULT
                      )
```

Constructs the Gaussian pyramid for an image.

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`.

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.
borderType	Pixel extrapolation method, see **BorderTypes** (**BORDER_CONSTANT** isn't supported)

§ dilate()

```
void cv::dilate ( InputArray        src,
                  OutputArray       dst,
                  InputArray        kernel,
                  Point             anchor = Point(-1,-1),
                  int               iterations = 1,
                  int               borderType = BORDER_CONSTANT,
                  const Scalar &    borderValue = morphologyDefaultBorderValue()
                )
```

Python:

```
dst = cv.dilate( src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]] )
```

Dilates an image by using a specific structuring element.

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:

$$\texttt{dst}(x,y) = \max_{(x',y'):\,\texttt{element}(x',y')\ne0} \texttt{src}(x+x',y+y')$$

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.

Parameters

src	input image; the number of channels can be arbitrary, but the depth should be one of CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output image of the same size and type as src.
kernel	structuring element used for dilation; if elemenat=Mat(), a 3 x 3 rectangular structuring element is used. Kernel can be created using **getStructuringElement**
anchor	position of the anchor within the element; 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 **BorderTypes**
borderValue	border value in case of a constant border

See also

erode, **morphologyEx**, **getStructuringElement**

Examples:

samples/cpp/segment_objects.cpp, **samples/cpp/squares.cpp**, **samples/cpp/stitching_detailed.cpp**, **samples/cpp/tutorial_code/ImgProc/Morphology_1.cpp**, and **samples/tapi/squares.cpp**.

§ erode()

```
void cv::erode ( InputArray        src,
                 OutputArray       dst,
                 InputArray        kernel,
                 Point             anchor = Point(-1,-1),
                 int               iterations = 1,
                 int               borderType = BORDER_CONSTANT,
                 const Scalar &    borderValue = morphologyDefaultBorderValue()
               )
```

Python:

dst = cv.erode(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])

Erodes an image by using a specific structuring element.

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:

$$\texttt{dst} (x,y) = \min _{(x',y'): \, \texttt{element} (x',y') \ne0 } \texttt{src} (x+x',y+y')$$

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.

Parameters

src	input image; the number of channels can be arbitrary, but the depth should be one of CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output image of the same size and type as src.
kernel	structuring element used for erosion; if `element=Mat()`, a `3 x 3` rectangular structuring element is used. Kernel can be created using **getStructuringElement**.
anchor	position of the anchor within the element; 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 **BorderTypes**
borderValue	border value in case of a constant border

See also

dilate, **morphologyEx**, **getStructuringElement**

Examples:

samples/cpp/segment_objects.cpp, and **samples/cpp/tutorial_code/ImgProc/Morphology_1.cpp**.

§ filter2D()

```
void cv::filter2D ( InputArray  src,
                    OutputArray dst,
                    int         ddepth,
                    InputArray  kernel,
                    Point       anchor = Point(-1,-1),
                    double      delta = 0,
                    int         borderType = BORDER_DEFAULT
                  )
```

Python:

```
dst = cv.filter2D( src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]] )
```

Convolves an image with the kernel.

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:

$$\texttt{dst}(x,y) = \sum_{\substack{0 \leq x' < \texttt{kernel.cols},\\ 0 \leq y' < \texttt{kernel.rows}}} \texttt{kernel}(x',y') * \texttt{src}(x+x'-\texttt{anchor.x}, y+y'-\texttt{anchor.y})$$

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 for small kernels.

Parameters

src	input image.
dst	output image of the same size and the same number of channels as src.
ddepth	desired depth of the destination image, see **combinations**
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; 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 **BorderTypes**

See also

sepFilter2D, **dft**, **matchTemplate**

§ GaussianBlur()

```
void cv::GaussianBlur ( InputArray   src,
                        OutputArray  dst,
                        Size         ksize,
                        double       sigmaX,
                        double       sigmaY = 0,
                        int          borderType = BORDER_DEFAULT
                      )
```

Python:

```
dst = cv.GaussianBlur( src, ksize, sigmaX[, dst[, sigmaY[, borderType]]] )
```

Blurs an image using a Gaussian filter.

The function convolves the source image with the specified Gaussian kernel. In-place filtering is supported.

Parameters

src	input image; the image can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output 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 **BorderTypes**

See also

sepFilter2D, **filter2D**, **blur**, **boxFilter**, **bilateralFilter**, **medianBlur**

Examples:

samples/cpp/laplace.cpp, **samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp**, and **samples/cpp/tutorial_code/ImgTrans/Sobel_Demo.cpp**.

§ getDerivKernels()

```
void cv::getDerivKernels ( OutputArray kx,
                           OutputArray ky,
                           int         dx,
                           int         dy,
                           int         ksize,
                           bool        normalize = false,
                           int         ktype = CV_32F
                         )
```

Python:

```
kx, ky = cv.getDerivKernels( dx, dy, ksize[, kx[, ky[, normalize[, ktype]]]] )
```

Returns filter coefficients for computing spatial image derivatives.

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

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 $= 2^{ksize*2-dx-dy-2}$. 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 .

§ getGaborKernel()

```
Mat cv::getGaborKernel ( Size   ksize,
                         double sigma,
                         double theta,
                         double lambd,
                         double gamma,
                         double psi = CV_PI *0.5,
                         int    ktype = CV_64F
                       )
```

Python:

```
retval = cv.getGaborKernel( ksize, sigma, theta, lambd, gamma[, psi[, ktype]] )
```

Returns Gabor filter coefficients.

For more details about gabor filter equations and parameters, see: Gabor Filter.

Parameters

ksize	Size of the filter returned.
sigma	Standard deviation of the gaussian envelope.
theta	Orientation of the normal to the parallel stripes of a Gabor function.
lambd	Wavelength of the sinusoidal factor.
gamma	Spatial aspect ratio.
psi	Phase offset.
ktype	Type of filter coefficients. It can be CV_32F or CV_64F .

§ getGaussianKernel()

```
Mat cv::getGaussianKernel ( int    ksize,
                            double sigma,
                            int    ktype = CV_64F
                          )
```

Python:

retval = cv.getGaussianKernel(ksize, sigma[, ktype])

Returns Gaussian filter coefficients.

The function computes and returns the $\texttt{ksize} \times 1$ 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. 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.

Parameters

- **ksize** Aperture size. It should be odd ($\texttt{ksize} \mod 2 = 1$) 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 .

See also

sepFilter2D, **getDerivKernels**, **getStructuringElement**, **GaussianBlur**

§ getStructuringElement()

```
Mat cv::getStructuringElement ( int   shape,
                                Size  ksize,
                                Point anchor = Point(-1,-1)
                              )
```

Python:

retval = cv.getStructuringElement(shape, ksize[, anchor])

Returns a structuring element of the specified size and shape for morphological operations.

The function constructs and returns the structuring element that can be further passed to **erode**, **dilate** or **morphologyEx**. But you can also construct an arbitrary binary mask yourself and use it as the structuring element.

Parameters

- **shape** Element shape that could be one of **MorphShapes**
- **ksize** Size 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.

Examples:

samples/cpp/tutorial_code/ImgProc/Morphology_1.cpp, and **samples/cpp/tutorial_code/ImgProc/Morphology_2.cpp**.

§ Laplacian()

```
void cv::Laplacian ( InputArray   src,
                     OutputArray  dst,
                     int          ddepth,
                     int          ksize = 1,
                     double       scale = 1,
                     double       delta = 0,
                     int          borderType = BORDER_DEFAULT
                   )
```

Python:

```
dst = cv.Laplacian( src, ddepth[, dst[, ksize[, scale[, delta[, borderType]]]]] )
```

Calculates the Laplacian of an image.

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst} = \Delta \texttt{src} = \frac{\partial^2 \texttt{src}}{\partial x^2} + \frac{\partial^2 \texttt{src}}{\partial y^2}$$

This is done when `ksize > 1`. When `ksize == 1`, the Laplacian is computed by filtering the image with the following 3×3 aperture:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

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 **BorderTypes**

See also

Sobel, **Scharr**

Examples:

samples/cpp/laplace.cpp.

§ medianBlur()

```
void cv::medianBlur ( InputArray   src,
                      OutputArray  dst,
                      int          ksize
                    )
```

Python:

```
dst = cv.medianBlur( src, ksize[, dst] )
```

Blurs an image using the median filter.

The function smoothes an image using the median filter with the $\texttt{ksize} \times \texttt{ksize}$ aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

Note

The median filter uses **BORDER_REPLICATE** internally to cope with border pixels, see **BorderTypes**

Parameters

- **src** input 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 ...

See also

bilateralFilter, **blur**, **boxFilter**, **GaussianBlur**

Examples:

samples/cpp/laplace.cpp, **samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp**, and **samples/cpp/tutorial_code/ImgTrans/houghcircles.cpp**.

§ morphologyDefaultBorderValue()

```
static Scalar cv::morphologyDefaultBorderValue ( )
```

inline static

returns "magic" border value for erosion and dilation. It is automatically transformed to **Scalar::all**(-DBL_MAX) for dilation.

§ morphologyEx()

```
void cv::morphologyEx ( InputArray      src,
                        OutputArray     dst,
                        int             op,
                        InputArray      kernel,
                        Point           anchor = Point(-1,-1),
                        int             iterations = 1,
                        int             borderType = BORDER_CONSTANT,
                        const Scalar &  borderValue = morphologyDefaultBorderValue()
                      )
```

Python:

dst = cv.morphologyEx(src, op, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])

Performs advanced morphological transformations.

The function **cv::morphologyEx** can perform advanced morphological transformations using an erosion and dilation as basic operations.

Any of the operations can be done in-place. In case of multi-channel images, each channel is processed independently.

Parameters

src	Source image. The number of channels can be arbitrary. The depth should be one of CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	Destination image of the same size and type as source image.
op	Type of a morphological operation, see **MorphTypes**
kernel	Structuring element. It can be created using **getStructuringElement**.
anchor	Anchor position with the kernel. Negative values mean that the anchor is at the kernel center.
iterations	Number of times erosion and dilation are applied.
borderType	Pixel extrapolation method, see **BorderTypes**
borderValue	Border value in case of a constant border. The default value has a special meaning.

See also

dilate, **erode**, **getStructuringElement**

Note

The number of iterations is the number of times erosion or dilatation operation will be applied. For instance, an opening operation (**MORPH_OPEN**) with two iterations is equivalent to apply successively: erode -> erode -> dilate -> dilate (and not erode -> dilate -> erode -> dilate).

Examples:

samples/cpp/tutorial_code/ImgProc/Morphology_2.cpp.

§ pyrDown()

```
void cv::pyrDown ( InputArray   src,
                   OutputArray  dst,
                   const Size & dstsize = Size(),
                   int          borderType = BORDER_DEFAULT
                 )
```

Python:

dst = cv.pyrDown(src[, dst[, dstsize[, borderType]]])

Blurs an image and downsamples it.

By default, size of the output image is computed as `Size((src.cols+1)/2, (src.rows+1)/2)`, but in any case, the following conditions should be satisfied:

$$|\texttt{dstsize.width} * 2 - src.cols| \leq 2$$
$$|\texttt{dstsize.height} * 2 - src.rows| \leq 2$$

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.

Parameters

src	input image.
dst	output image; it has the specified size and the same type as src.
dstsize	size of the output image.
borderType	Pixel extrapolation method, see **BorderTypes** (**BORDER_CONSTANT** isn't supported)

Examples:

samples/cpp/squares.cpp, **samples/cpp/tutorial_code/ImgProc/Pyramids/Pyramids.cpp**, and **samples/tapi/squares.cpp**.

§ pyrMeanShiftFiltering()

```
void cv::pyrMeanShiftFiltering ( InputArray   src,
                                 OutputArray  dst,
                                 double       sp,
                                 double       sr,
                                 int          maxLevel = 1,
                                 TermCriteria termcrit = TermCriteria(TermCriteria::MAX_ITER+TermCriteria::EPS, 5, 1)
                               )
```

Python:

```
dst = cv.pyrMeanShiftFiltering( src, sp, sr[, dst[, maxLevel[, termcrit]]] )
```

Performs initial step of meanshift segmentation of an image.

The function implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X,Y) neighborhood in the joint space-color hyperspace is considered:

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

where (R,G,B) and (r,g,b) are the vectors of color components at (X,Y) and (x,y), respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value (X',Y') and average color vector (R',G',B') are found and they act as the neighborhood center on the next iteration:

$$(X,Y)\;(X',Y'), (R,G,B)\;(R',G',B').$$

After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration):

$$I(X,Y) <- (R*,G*,B*)$$

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).

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.

§ pyrUp()

```
void cv::pyrUp ( InputArray   src,
                 OutputArray  dst,
                 const Size & dstsize = Size(),
                 int          borderType = BORDER_DEFAULT
               )
```

Python:

dst = cv.pyrUp(src[, dst[, dstsize[, borderType]]])

Upsamples an image and then blurs it.

By default, size of the output image is computed as `Size(src.cols\*2, (src.rows\*2)`, but in any case, the following conditions should be satisfied:

$$|\texttt{dstsize.width} - src.cols * 2| \leq (\texttt{dstsize.width} \mod 2)$$
$$|\texttt{dstsize.height} - src.rows * 2| \leq (\texttt{dstsize.height} \mod 2)$$

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.

Parameters

src	input image.
dst	output image. It has the specified size and the same type as src .
dstsize	size of the output image.
borderType	Pixel extrapolation method, see BorderTypes (only BORDER_DEFAULT is supported)

Examples:

samples/cpp/squares.cpp, samples/cpp/tutorial_code/ImgProc/Pyramids/Pyramids.cpp, and samples/tapi/squares.cpp.

§ Scharr()

```
void cv::Scharr ( InputArray  src,
                  OutputArray dst,
                  int         ddepth,
                  int         dx,
                  int         dy,
                  double      scale = 1,
                  double      delta = 0,
                  int         borderType = BORDER_DEFAULT
                )
```

Python:

```
dst = cv.Scharr( src, ddepth, dx, dy[, dst[, scale[, delta[, borderType]]]] )
```

Calculates the first x- or y- image derivative using Scharr operator.

The function computes the first x- or y- spatial image derivative using the Scharr operator. The call

$$\texttt{Scharr(src, dst, ddepth, dx, dy, scale, delta, borderType)}$$

is equivalent to

$$\texttt{Sobel(src, dst, ddepth, dx, dy, CV\_SCHARR, scale, delta, borderType)}.$$

Parameters

src	input image.
dst	output image of the same size and the same number of channels as src.
ddepth	output image depth, see **combinations**
dx	order of the derivative x.
dy	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 **BorderTypes**

See also

cartToPolar

Examples:

samples/cpp/edge.cpp.

§ sepFilter2D()

```
void cv::sepFilter2D ( InputArray   src,
                       OutputArray  dst,
                       int          ddepth,
                       InputArray   kernelX,
                       InputArray   kernelY,
                       Point        anchor = Point(-1,-1),
                       double       delta = 0,
                       int          borderType = BORDER_DEFAULT
                     )
```

Python:

dst = cv.sepFilter2D(src, ddepth, kernelX, kernelY[, dst[, anchor[, delta[, borderType]]]])

Applies a separable linear filter to an image.

The function applies a separable linear filter to the image. That is, first, every row of src is filtered with the 1D kernel kernelX. Then, every column of the result is filtered with the 1D kernel kernelY. The final result shifted by delta is stored in dst .

Parameters

src	Source image.
dst	Destination image of the same size and the same number of channels as src .
ddepth	Destination image depth, see **combinations**
kernelX	Coefficients for filtering each row.
kernelY	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 **BorderTypes**

See also

filter2D, **Sobel**, **GaussianBlur**, **boxFilter**, **blur**

§ Sobel()

```
void cv::Sobel ( InputArray  src,
                 OutputArray dst,
                 int         ddepth,
                 int         dx,
                 int         dy,
                 int         ksize = 3,
                 double      scale = 1,
                 double      delta = 0,
                 int         borderType = BORDER_DEFAULT
               )
```

Python:

```
dst = cv.Sobel( src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]] )
```

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

In all cases except one, the $\texttt{ksize} \times \texttt{ksize}$ separable kernel is used to calculate the derivative. When $\texttt{ksize = 1}$, 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:

$$\texttt{dst} = \frac{\partial^{xorder+yorder} \texttt{src}}{\partial x^{xorder} \partial y^{yorder}}$$

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:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Parameters

src	input image.
dst	output image of the same size and the same number of channels as src .
ddepth	output image depth, see **combinations**; in the case of 8-bit input images it will result in truncated derivatives.
dx	order of the derivative x.
dy	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 **BorderTypes**

See also

Scharr, **Laplacian**, **sepFilter2D**, **filter2D**, **GaussianBlur**, **cartToPolar**

Examples:

samples/cpp/tutorial_code/ImgTrans/Sobel_Demo.cpp.

§ spatialGradient()

void cv::spatialGradient (**InputArray** src,
OutputArray dx,
OutputArray dy,
int ksize = 3,
int borderType = `BORDER_DEFAULT`
)

Python:
dx, dy = cv.spatialGradient(src[, dx[, dy[, ksize[, borderType]]]])

Calculates the first order image derivative in both x and y using a Sobel operator.

Equivalent to calling:

```
Sobel( src, dx, CV_16SC1, 1, 0, 3 );
Sobel( src, dy, CV_16SC1, 0, 1, 3 );
```

Parameters

src	input image.
dx	output image with first-order derivative in x.
dy	output image with first-order derivative in y.
ksize	size of Sobel kernel. It must be 3.
borderType	pixel extrapolation method, see **BorderTypes**

See also
Sobel

§ sqrBoxFilter()

void cv::sqrBoxFilter (**InputArray** _src,
OutputArray _dst,
int ddepth,
Size ksize,
Point anchor = `Point(-1, -1)`,
bool normalize = `true`,
int borderType = `BORDER_DEFAULT`
)

Python:
_dst = cv.sqrBoxFilter(_src, ddepth, ksize[, _dst[, anchor[, normalize[, borderType]]]])

Calculates the normalized sum of squares of the pixel values overlapping the filter.

For every pixel (x, y) in the source image, the function calculates the sum of squares of those neighboring pixel values which overlap the filter placed over the pixel (x, y).

The unnormalized square box filter can be useful in computing local image statistics such as the the local variance and standard deviation around the neighborhood of a pixel.

Parameters

_src	input image
_dst	output image of the same size and type as _src
ddepth	the output image depth (-1 to use **src.depth()**)
ksize	kernel size
anchor	kernel anchor point. The default value of Point(-1, -1) denotes that the anchor is at the kernel center.
normalize	flag, specifying whether the kernel is to be normalized by it's area or not.
borderType	border mode used to extrapolate pixels outside of the image, see **BorderTypes**

See also
boxFilter