**2D Convolution ( Image Filtering )**

As in one-dimensional signals, images also can be filtered with various low-pass filters (LPF), high-pass filters (HPF), etc. LPF helps in removing noise, blurring images, etc. HPF filters help in finding edges in images.

OpenCV provides a function [**cv.filter2D()**](https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04) to convolve a kernel with an image. As an example, we will try an averaging filter on an image. A 5x5 averaging filter kernel will look like the below:

K=125⎡⎣⎢⎢⎢⎢⎢⎢1111111111111111111111111⎤⎦⎥⎥⎥⎥⎥⎥

The operation works like this: keep this kernel above a pixel, add all the 25 pixels below this kernel, take the average, and replace the central pixel with the new average value. This operation is continued for all the pixels in the image. Try this code and check the result:

import numpy as np

import cv2 as cv

from matplotlib import pyplot as plt

img = [cv.imread](https://docs.opencv.org/master/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bbd3a56)('opencv\_logo.png')

kernel = np.ones((5,5),np.float32)/25

dst = [cv.filter2D](https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04)(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')

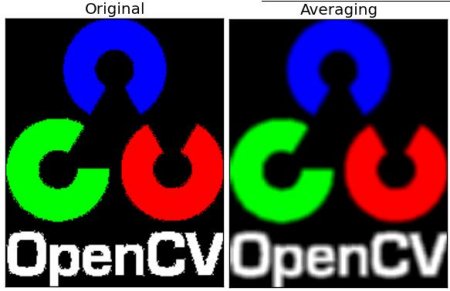
plt.xticks([]), plt.yticks([])

plt.subplot(122),plt.imshow(dst),plt.title('Averaging')

plt.xticks([]), plt.yticks([])

plt.show()

Result:



**image**

[◆](https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga27c049795ce870216ddfb366086b5a04)filter2D()

|  |  |  |  |
| --- | --- | --- | --- |
| void cv::filter2D | ( | [**InputArray**](https://docs.opencv.org/master/dc/d84/group__core__basic.html#ga353a9de602fe76c709e12074a6f362ba) | src, |
|  |  | [**OutputArray**](https://docs.opencv.org/master/dc/d84/group__core__basic.html#gaad17fda1d0f0d1ee069aebb1df2913c0) | dst, |
|  |  | int | ddepth, |
|  |  | [**InputArray**](https://docs.opencv.org/master/dc/d84/group__core__basic.html#ga353a9de602fe76c709e12074a6f362ba) | kernel, |
|  |  | [**Point**](https://docs.opencv.org/master/dc/d84/group__core__basic.html#ga1e83eafb2d26b3c93f09e8338bcab192) | anchor = [**Point**](https://docs.opencv.org/master/dc/d84/group__core__basic.html#ga1e83eafb2d26b3c93f09e8338bcab192)(-1,-1), |
|  |  | double | delta = 0, |
|  |  | int | borderType = [**BORDER\_DEFAULT**](https://docs.opencv.org/master/d2/de8/group__core__array.html#gga209f2f4869e304c82d07739337eae7c5afe14c13a4ea8b8e3b3ef399013dbae01) |
|  | ) |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Python:** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | dst | = | cv.filter2D( | src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]] | ) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

#include <[**opencv2/imgproc.hpp**](https://docs.opencv.org/master/d1/d4f/imgproc_2include_2opencv2_2imgproc_8hpp.html)>

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:

dst(x,y)=∑0≤y′<kernel.rows0≤x′<kernel.cols,kernel(x′,y′)∗src(x+x′−anchor.x,y+y′−anchor.y)

That is, the kernel is not mirrored around the anchor point. If you need a real convolution, flip the kernel using [**flip**](https://docs.opencv.org/master/d2/de8/group__core__array.html#gaca7be533e3dac7feb70fc60635adf441) 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(sort of claity) of the destination image, see [**combinations**](https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#filter_depths) |
| **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](https://docs.opencv.org/master/d2/de8/group__core__array.html" \l "ga209f2f4869e304c82d07739337eae7c5)**(**Extrapolation** is an estimation of a value based on extending a known sequence of values or facts beyond the area that is certainly known.) |

* Smoothing, also called blurring, is a simple and frequently used image processing operation.
* There are many reasons for smoothing. In this tutorial we will focus on smoothing in order to reduce noise (other uses will be seen in the following tutorials).
* To perform a smoothing operation we will apply a filter to our image. The most common type of filters are linear, in which an output pixel’s value (i.e. g(i,j)) is determined as a weighted sum of input pixel values (i.e. f(i+k,j+l)) :

g(i,j) = \sum_{k,l} f(i+k, j+l) h(k,l)

h(k,l) is called the kernel, which is nothing more than the coefficients of the filter.

It helps to visualize a filter as a window of coefficients sliding across the image.

* There are many kind of filters, here we will mention the most used:

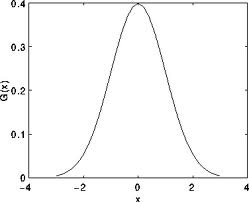
### Normalized Box Filter

* This filter is the simplest of all! Each output pixel is the mean of its kernel neighbors ( all of them contribute with equal weights)
* The kernel is below:

K = \dfrac{1}{K_{width} \cdot K_{height}} \begin{bmatrix}
    1 & 1 & 1 & ... & 1 \\
    1 & 1 & 1 & ... & 1 \\
    . & . & . & ... & 1 \\
    . & . & . & ... & 1 \\
    1 & 1 & 1 & ... & 1
   \end{bmatrix}

### Gaussian Filter

* Probably the most useful filter (although not the fastest). Gaussian filtering is done by convolving each point in the input array with a Gaussian kernel and then summing them all to produce the output array.
* Just to make the picture clearer, remember how a 1D Gaussian kernel look like?



Assuming that an image is 1D, you can notice that the pixel located in the middle would have the biggest weight. The weight of its neighbors decreases as the spatial distance between them and the center pixel increases.

**Note**

Remember that a 2D Gaussian can be represented as :

G_{0}(x, y) = A  e^{ \dfrac{ -(x - \mu_{x})^{2} }{ 2\sigma^{2}_{x} } +  \dfrac{ -(y - \mu_{y})^{2} }{ 2\sigma^{2}_{y} } }

where \mu is the mean (the peak) and \sigma represents the standard deviation (per each of the variables x and y)

### Median Filter

The median filter run through each element of the signal (in this case the image) and replace each pixel with the **median** of its neighboring pixels (located in a square neighborhood around the evaluated pixel).

### Bilateral Filter

* So far, we have explained some filters which main goal is to smooth an input image. However, sometimes the filters do not only dissolve the noise, but also smooth away the edges. To avoid this (at certain extent at least), we can use a bilateral filter.
* In an analogous way as the Gaussian filter, the bilateral filter also considers the neighboring pixels with weights assigned to each of them. These weights have two components, the first of which is the same weighting used by the Gaussian filter. The second component takes into account the difference in intensity between the neighboring pixels and the evaluated one.
* For a more detailed explanation you can check [this link](http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html)

## Code

* **What does this program do?**
  + Loads an image
  + Applies 4 different kinds of filters (explained in Theory) and show the filtered images sequentially
* **Downloadable code**: Click [here](https://github.com/opencv/opencv/tree/master/samples/cpp/tutorial_code/ImgProc/Smoothing.cpp)
* **Code at glance:**

#include *"opencv2/imgproc/imgproc.hpp"*

#include *"opencv2/highgui/highgui.hpp"*

**using** **namespace** std;

**using** **namespace** cv;

*/// Global Variables*

int DELAY\_CAPTION = 1500;

int DELAY\_BLUR = 100;

int MAX\_KERNEL\_LENGTH = 31;

Mat src; Mat dst;

char window\_name[] = "Filter Demo 1";

*/// Function headers*

int display\_caption( char\* caption );

int display\_dst( int delay );

*/\*\**

*\* function main*

*\*/*

int main( int argc, char\*\* argv )

{

namedWindow( window\_name, CV\_WINDOW\_AUTOSIZE );

*/// Load the source image*

src = imread( "../images/lena.jpg", 1 );

**if**( display\_caption( "Original Image" ) != 0 ) { **return** 0; }

dst = src.clone();

**if**( display\_dst( DELAY\_CAPTION ) != 0 ) { **return** 0; }

*/// Applying Homogeneous blur*

**if**( display\_caption( "Homogeneous Blur" ) != 0 ) { **return** 0; }

**for** ( int i = 1; i < MAX\_KERNEL\_LENGTH; i = i + 2 )

{ blur( src, dst, Size( i, i ), Point(-1,-1) );

**if**( display\_dst( DELAY\_BLUR ) != 0 ) { **return** 0; } }

*/// Applying Gaussian blur*

**if**( display\_caption( "Gaussian Blur" ) != 0 ) { **return** 0; }

**for** ( int i = 1; i < MAX\_KERNEL\_LENGTH; i = i + 2 )

{ GaussianBlur( src, dst, Size( i, i ), 0, 0 );

**if**( display\_dst( DELAY\_BLUR ) != 0 ) { **return** 0; } }

*/// Applying Median blur*

**if**( display\_caption( "Median Blur" ) != 0 ) { **return** 0; }

**for** ( int i = 1; i < MAX\_KERNEL\_LENGTH; i = i + 2 )

{ medianBlur ( src, dst, i );

**if**( display\_dst( DELAY\_BLUR ) != 0 ) { **return** 0; } }

*/// Applying Bilateral Filter*

**if**( display\_caption( "Bilateral Blur" ) != 0 ) { **return** 0; }

**for** ( int i = 1; i < MAX\_KERNEL\_LENGTH; i = i + 2 )

{ bilateralFilter ( src, dst, i, i\*2, i/2 );

**if**( display\_dst( DELAY\_BLUR ) != 0 ) { **return** 0; } }

*/// Wait until user press a key*

display\_caption( "End: Press a key!" );

waitKey(0);

**return** 0;

}

int display\_caption( char\* caption )

{

dst = Mat::zeros( src.size(), src.type() );

putText( dst, caption,

Point( src.cols/4, src.rows/2),

CV\_FONT\_HERSHEY\_COMPLEX, 1, Scalar(255, 255, 255) );

imshow( window\_name, dst );

int c = waitKey( DELAY\_CAPTION );

**if**( c >= 0 ) { **return** -1; }

**return** 0;

}

int display\_dst( int delay )

{

imshow( window\_name, dst );

int c = waitKey ( delay );

**if**( c >= 0 ) { **return** -1; }

**return** 0;

}

## Explanation

## 1. boxFilter

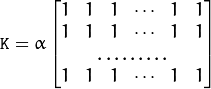
Blurs an image using the box filter.

Let’s check the OpenCV functions that involve only the smoothing procedure, since the rest is already known by now

**Python:**cv2.**boxFilter**(src, ddepth, ksize[, dst[, anchor[, normalize[, borderType]]]]) → dst

|  |  |
| --- | --- |
| **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. |

The function smoothes an image using the kernel:



where

\alpha = \fork{\frac{1}{\texttt{ksize.width*ksize.height}}}{when \texttt{normalize=true}}{1}{otherwise}

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()**](https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html#void%20integral(InputArray%20src,%20OutputArray%20sum,%20int%20sdepth)) .

.

1. **Normalized Block Filter:**

OpenCV offers the function [blur](http://docs.opencv.org/modules/imgproc/doc/filtering.html?highlight=blur#blur) to perform smoothing with this filter.

**Python:**cv2.**blur**(src, ksize[, dst[, anchor[, borderType]]]) → dst

|  |  |
| --- | --- |
| **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. |

The function smoothes an image using the kernel:



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

1. **Gaussian Filter:**

**Python:**cv2.**GaussianBlur**(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]]) → dst

|  |  |
| --- | --- |
| **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()](https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html" \l "Mat%20getGaussianKernel(int%20ksize,%20double%20sigma,%20int%20ktype)" \o "Mat getGaussianKernel(int ksize, double sigma, int ktype))** for details); to fully control the result regardless of possible future modifications of all this semantics, it is recommended to specify all of ksize, sigmaX, and sigmaY. * **borderType** – pixel extrapolation method (see **[borderInterpolate()](https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html" \l "int%20borderInterpolate(int%20p,%20int%20len,%20int%20borderType)" \o "int borderInterpolate(int p, int len, int borderType))** for details). |

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

It is performed by the function [GaussianBlur](http://docs.opencv.org/modules/imgproc/doc/filtering.html?highlight=gaussianblur" \l "gaussianblur) :

1. **Median Filter:**

**Python:**cv2.**medianBlur**(src, ksize[, dst]) → dst

|  |  |
| --- | --- |
| **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 ... |

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.

1. **Bilateral Filter**

**Python:**cv2.**bilateralFilter**(src, d, sigmaColor, sigmaSpace[, dst[, borderType]]) → dst

|  |  |
| --- | --- |
| **Parameters:** | * **src** – Source 8-bit or floating-point, 1-channel or 3-channel image. * **dst** – Destination image of the same size and type as src . * **d** – Diameter of each pixel neighborhood that is used during filtering. If it is non-positive, it is computed from sigmaSpace . * **sigmaColor** – Filter sigma in the color space. A larger value of the parameter means that farther colors within the pixel neighborhood (see sigmaSpace ) will be mixed together, resulting in larger areas of semi-equal color. * **sigmaSpace** – Filter sigma in the coordinate space. A larger value of the parameter means that farther pixels will influence each other as long as their colors are close enough (see sigmaColor ). When d>0 , it specifies the neighborhood size regardless of sigmaSpace . Otherwise, d is proportional to sigmaSpace . |

The function applies bilateral filtering to the input image, as described in <http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html> bilateralFilter can reduce unwanted noise very well while keeping edges fairly sharp. However, it is very slow compared to most filters.

*Sigma values*: For simplicity, you can set the 2 sigma values to be the same. If they are small (< 10), the filter will not have much effect, whereas if they are large (> 150), they will have a very strong effect, making the image look “cartoonish”.

*Filter size*: Large filters (d > 5) are very slow, so it is recommended to use d=5 for real-time applications, and perhaps d=9 for offline applications that need heavy noise filtering.

1. **Results**

* The code opens an image (in this case lena.jpg) and display it under the effects of the 4 filters explained.
* Here is a snapshot of the image smoothed using medianBlur:

