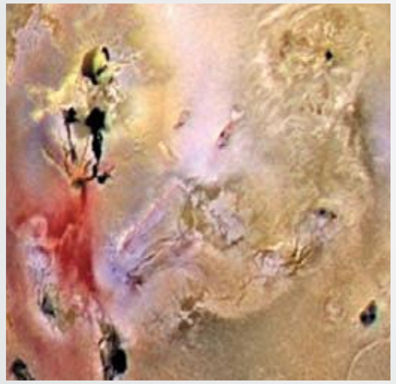


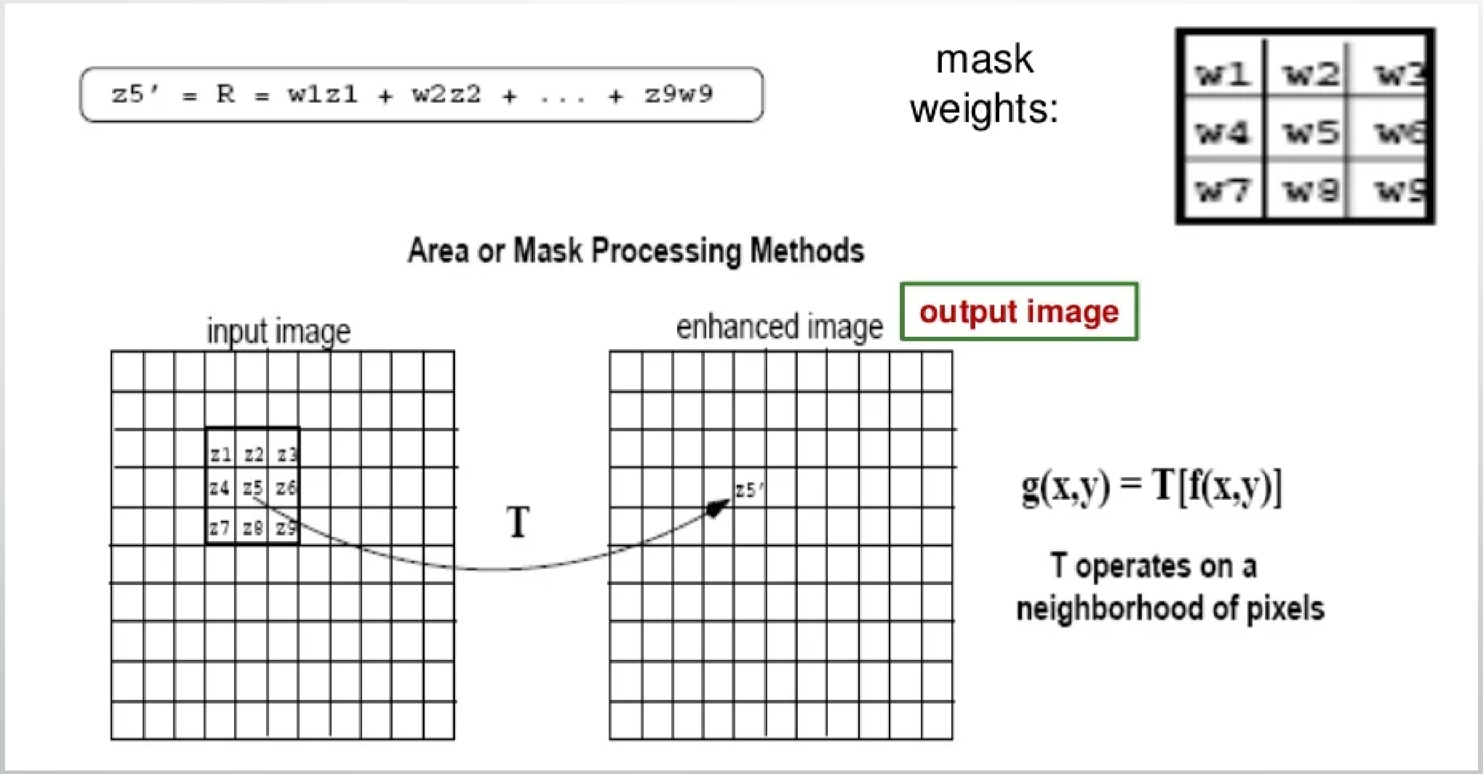
CS 314

IMAGE PROCESSING PRACTICAL

07 – Spatial Filtering

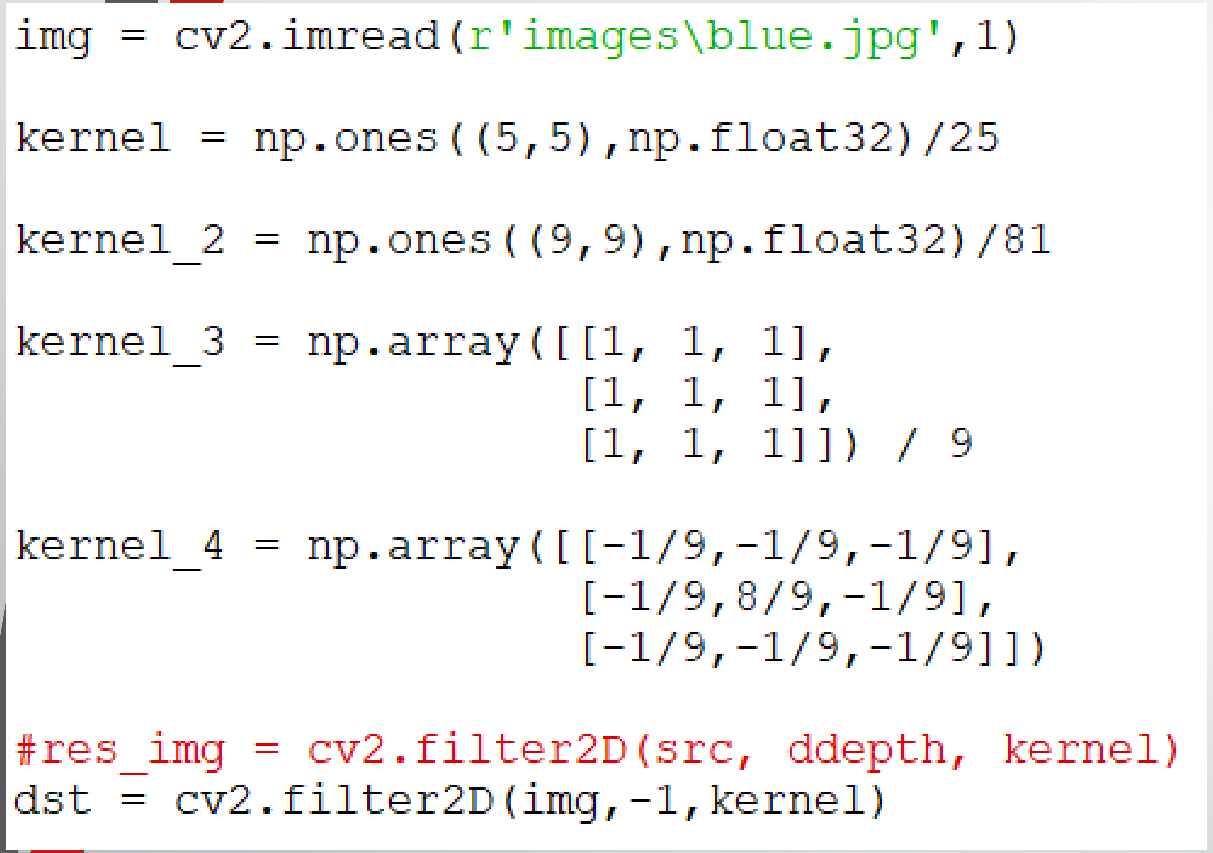
Spatial Filtering / 2D Convolution

Spatial Filtering technique is used directly on pixels of an image. Mask is usually considered to be added in size so that it has a specific center pixel. This mask is moved on the image such that the center of the mask traverses all image pixels.



Concept of Spatial Filtering

Creating custom filters



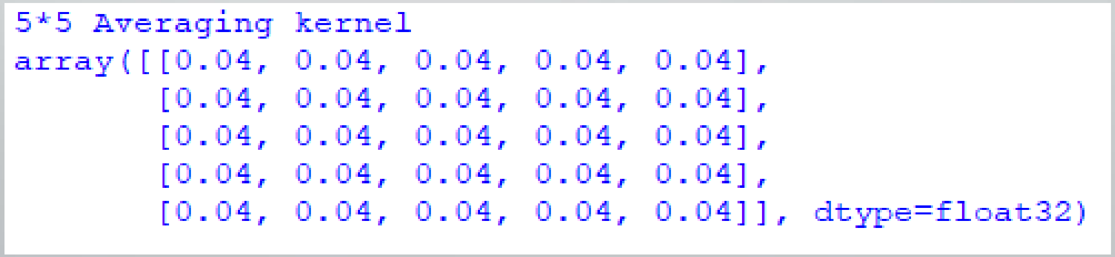
```
img = cv2.imread(r'images\blue.jpg',1)

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

kernel_2 = np.ones((9,9),np.float32)/81

kernel_3 = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 1]]) / 9

kernel_4 = np.array([[-1/9,-1/9,-1/9],
                     [-1/9,8/9,-1/9],
                     [-1/9,-1/9,-1/9]])

#res_img = cv2.filter2D(src, ddepth, kernel)
dst = cv2.filter2D(img,-1,kernel)
```

res_img = cv2.filter2D(src, ddepth, kernel)

src: The source image on which to apply the filter.

ddepth: desirable depth of destination image. Value -1 represents that the resulting image will have same depth as the source image.

kernel: kernel is the filter matrix applied on the image



```
5*5 Averaging kernel
array([[0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04]], dtype=float32)
```

Low Pass Filters

(Image Smoothing / De-noising)

Image Blurring (Image Smoothing)

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise (de-noising). This removes high frequency content (e.g: noise, edges) from the image resulting in edges being blurred when this is filter is applied.

OpenCV provides mainly four types of blurring techniques.

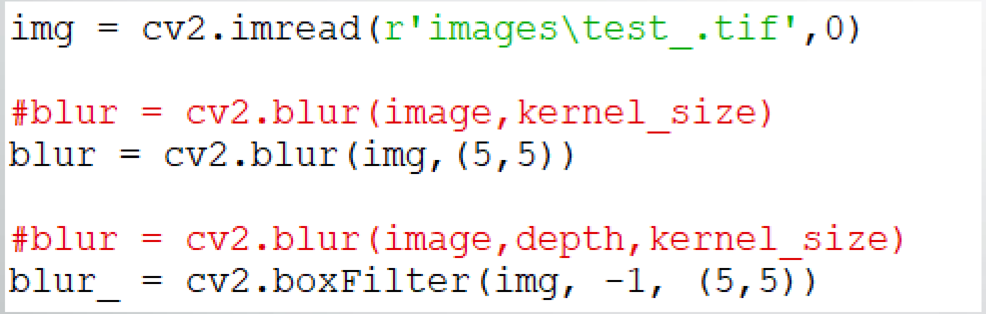
1. Averaging
2. Gaussian Filtering
3. Median Filtering
4. Bilateral Filtering

Averaging

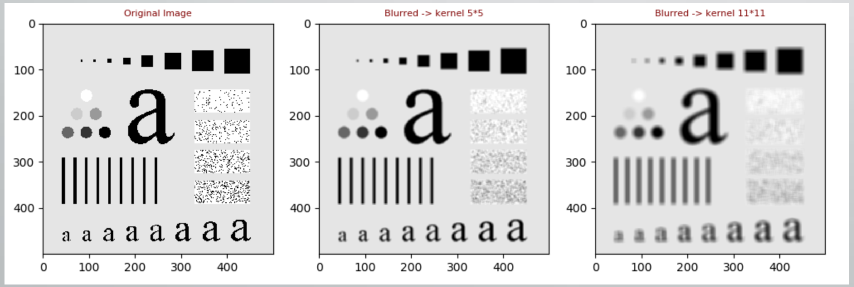
This is done by convolving the image with a normalized box filter. It simply takes the average of all the pixels under kernel area and replaces the central element with this average. This is done by the function **`cv2.blur()`** or **`cv2.boxFilter()`**. We should specify the width and height of kernel.

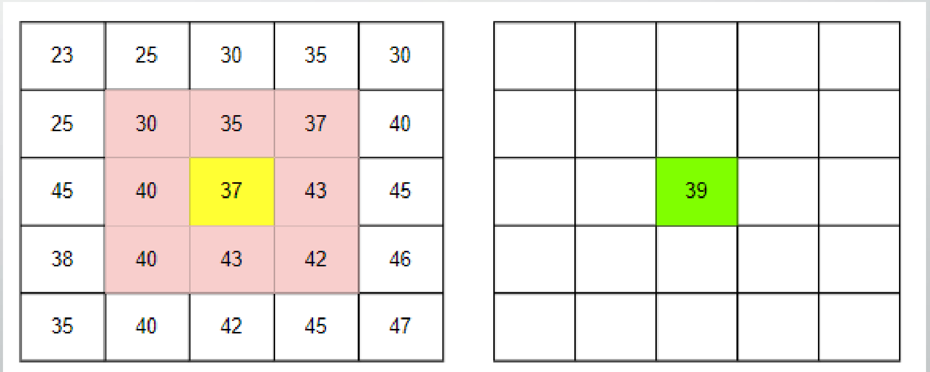
A 3x3 normalized box filter (K) would look like this:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



```
img = cv2.imread(r'images\test_.tif',0)

#blur = cv2.blur(image,kernel_size)
blur = cv2.blur(img,(5,5))

#blur = cv2.blur(image,depth,kernel_size)
blur_ = cv2.boxFilter(img, -1, (5,5))
```





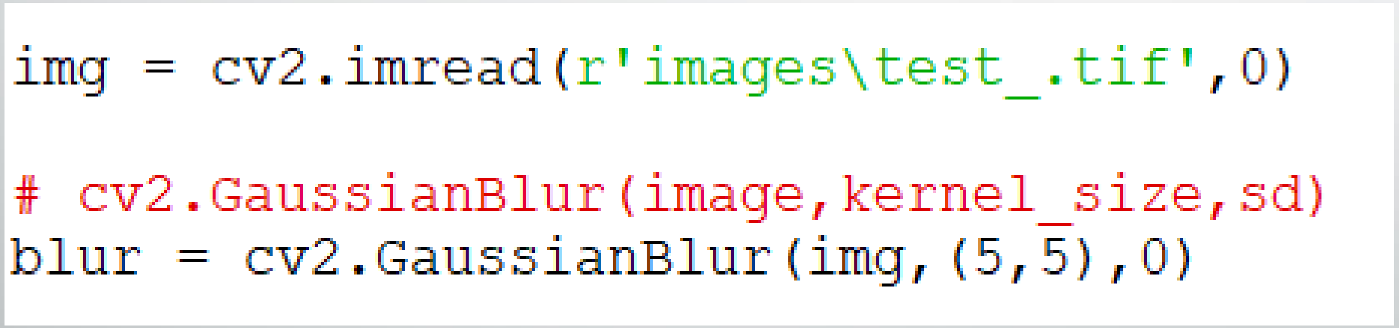
23	25	30	35	30
25	30	35	37	40
45	40	37	43	45
38	40	43	42	46
35	40	42	45	47

		39		

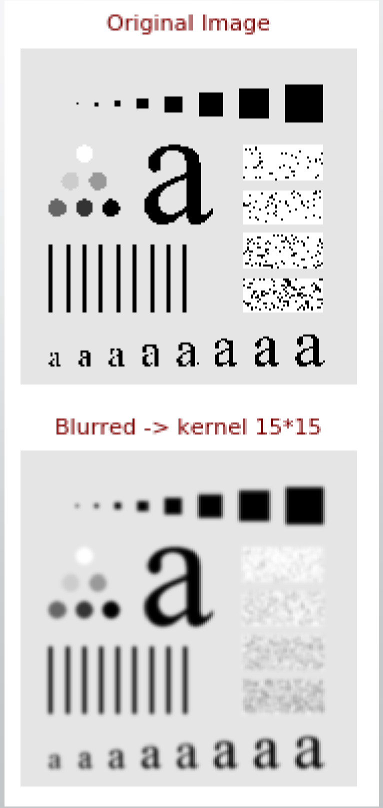
Value = (30 + 35+ 37+ 40 + 37+ 43+ 40+ 43+ 42) /9 = 38.55 = 39

Gaussian Filtering

In this approach, instead of a box filter consisting of equal filter coefficients, a Gaussian kernel is used. It is done with the function, **`cv2.GaussianBlur()`**. We should specify the width and height of the kernel which should be positive and odd. Gaussian filtering is highly effective in removing Gaussian noise from the image.



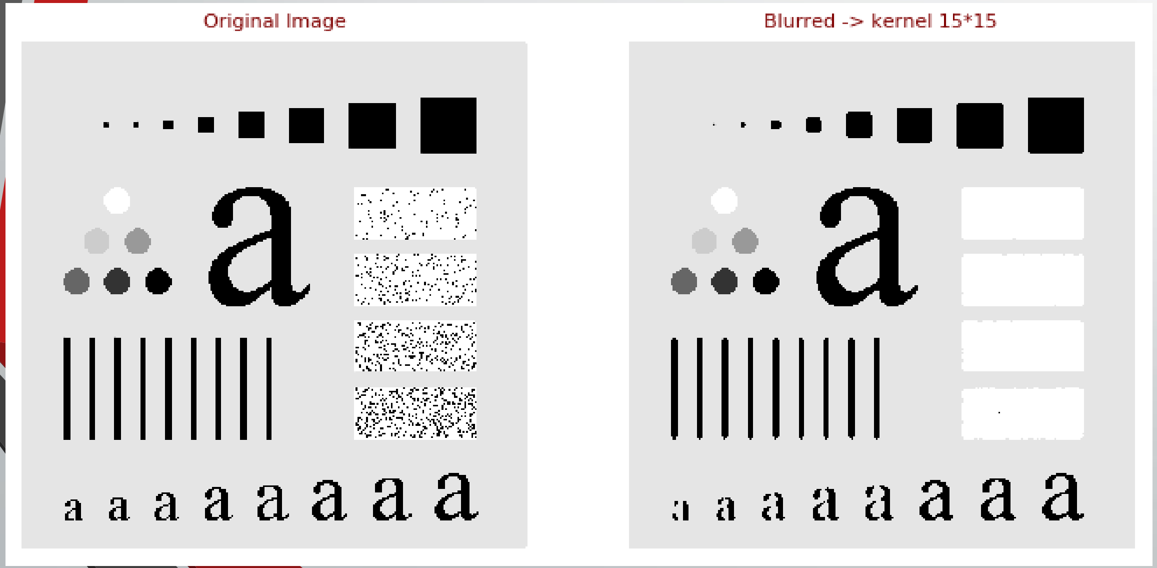
```
img = cv2.imread(r'images\test_.tif',0)

# cv2.GaussianBlur(image,kernel_size,sd)
blur = cv2.GaussianBlur(img,(5,5),0)
```

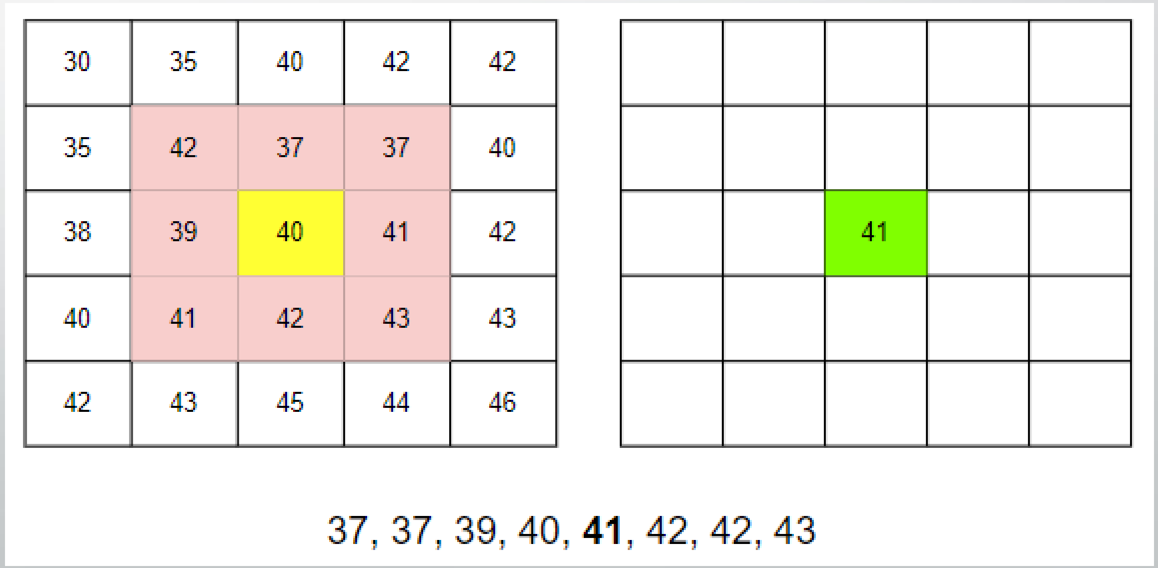


Median Filtering

Here, the function **`cv2.medianBlur()`** computes the median of all the pixels under the kernel window and the central pixel is replaced with this median value. This is highly effective in removing salt-and-pepper noise.

```
img = cv2.imread(r'images\test_pattern_blurring_orig.tif',0)

# cv2.medianBlur(image,kernel_size)
blur = cv2.medianBlur(img,5)
```





30	35	40	42	42
35	42	37	37	40
38	39	40	41	42
40	41	42	43	43
42	43	45	44	46

		41		

37, 37, 39, 40, **41**, 42, 42, 43

Bilateral Filtering

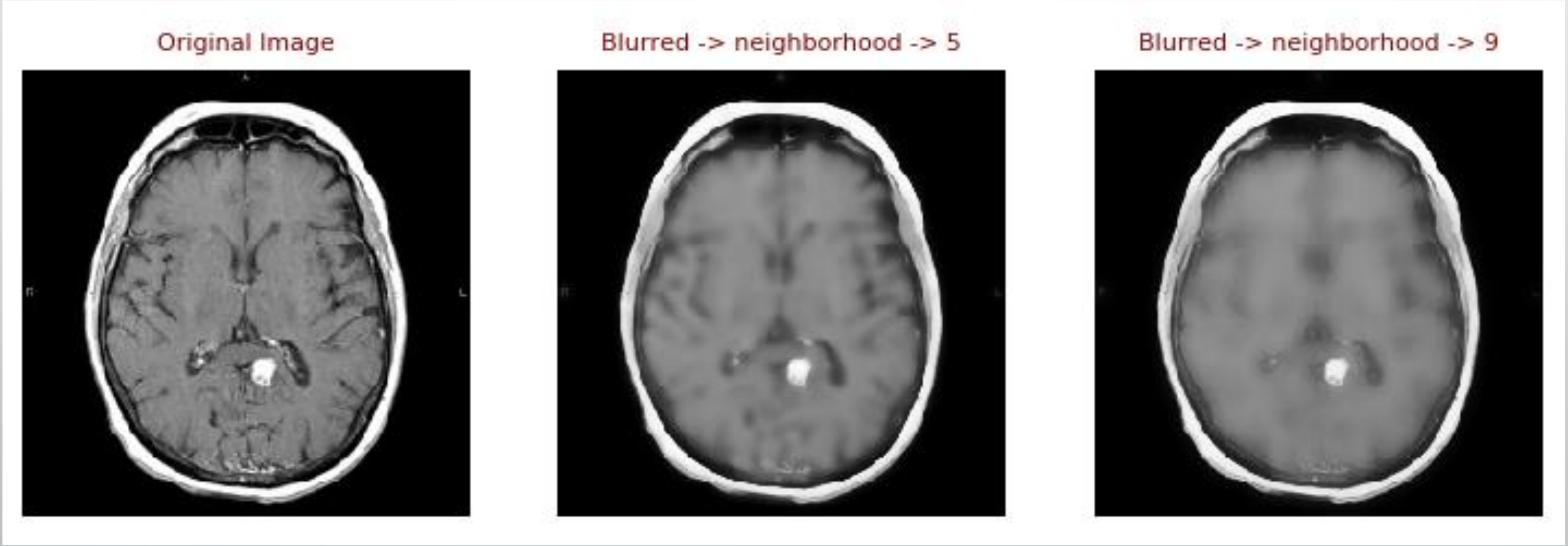
`cv2.bilateralFilter()` is highly effective at noise removal while preserving edges. Bilateral filter replaces the intensity of each pixel with a weighted average of intensity values from nearby pixels.

```
blur = cv2.bilateralFilter(img, d, sigmaColor, sigmaSpace)
```

d	**-**	Diameter of each pixel neighborhood that is used during filtering.
sigmaColor	**-**	Filter sigma in the color space. A larger value of the parameter means that farther colors within the pixel neighborhood 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.
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".

Bilateral Filtering

```
img = cv2.imread(r'images\img_mri_brain_tumor.jpg',0)

# cv2.bilateralFilter(image,neighborhood,sigmaColor,sigmaSpace)
blur = cv2.bilateralFilter(img, 9, 75,75)
```

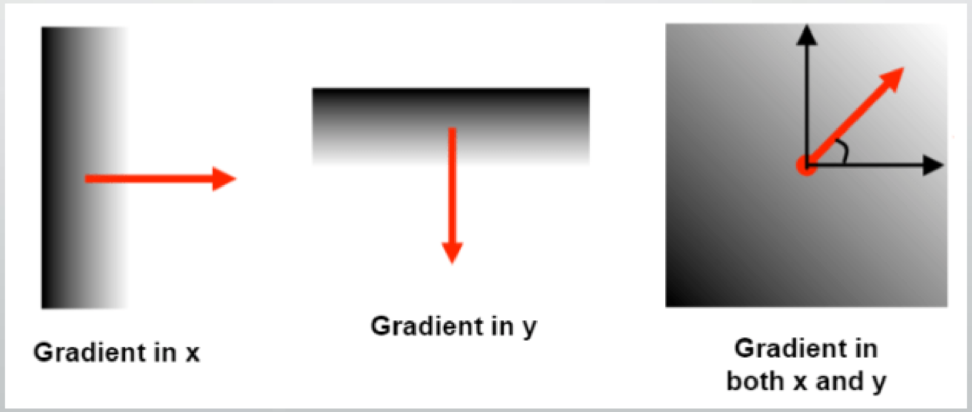


High Pass Filters
(Image Gradients & Sharpening)

High Pass Filtering

A **high pass filter** tends to retain the high frequency information within an image while reducing the low frequency information. The kernel of the high pass filter is designed to increase the brightness of the center pixel relative to neighboring pixels

An **image gradient** is a directional change in the intensity or color in an image. The gradient of the image is one of the fundamental building blocks in image processing. For example, the Canny edge detector uses image gradient for edge detection.



Sobel Filters

-1	0	+1
-2	0	+2
-1	0	+1

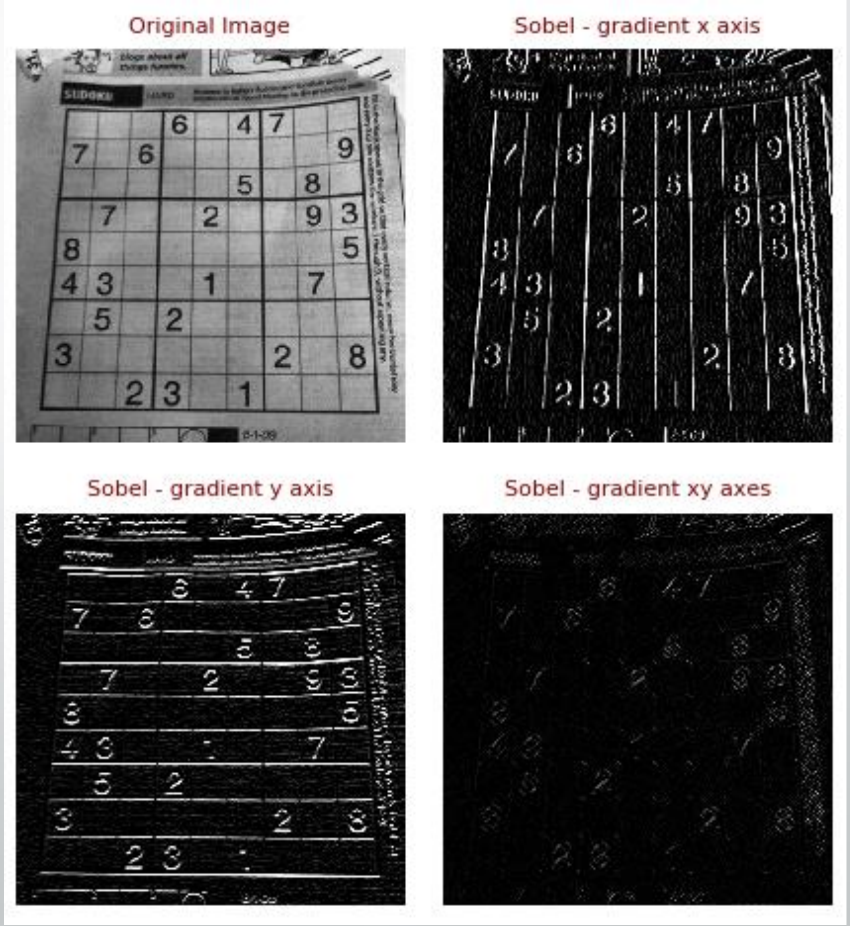
Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

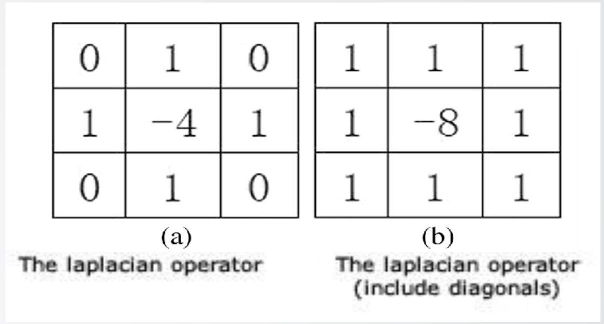
cv2.Sobel(image, CV2_dtype, gradient_xAxis, gradient_yAxis, ksize=3)

```
img = cv2.imread(r'images\sudoku-original.jpg',0)

#cv2.Sobel(image, CV2_dtype, xAxis, yAxis, ksize=3)
#vertical -> xAxis =1
grad_x = cv2.Sobel(img, cv2.CV_8U, 1, 0, ksize=3)
grad_x2 = cv2.Sobel(img, cv2.CV_32F, 1, 0, ksize=3)
grad_x3 = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)

#horizontal -> yAxis =1
grad_y = cv2.Sobel(img, cv2.CV_8U, 0,1, ksize=3)
grad_y2 = cv2.Sobel(img, cv2.CV_32F, 0, 1, ksize=3)
grad_y3 = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)

#both axes
grad_xy = cv2.Sobel(img, cv2.CV_8U, 1,1, ksize=3)

#vertical + horizontal
grad_xy = grad_x+grad_y
```



Laplacian Filters

A Laplacian filter is an **edge detector** which determines if a change in adjacent pixel values is from an edge or continuous progression.



0	1	0
1	−4	1
0	1	0

(a)

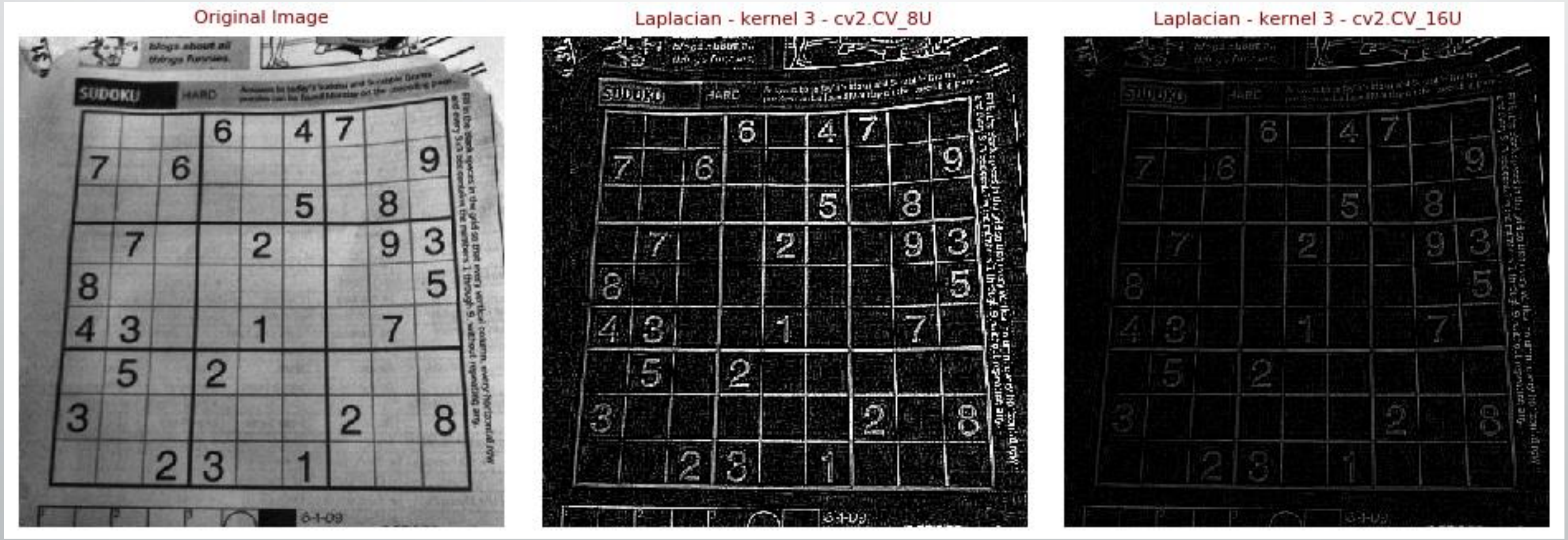
The laplacian operator

1	1	1
1	−8	1
1	1	1

(b)

The laplacian operator (include diagonals)

```
# cv2.Laplacian(src_gray, CV2_dtype, ksize=kernel_size)
laplacian = cv2.Laplacian(img,cv2.CV_8U,ksize=3)
laplacian_ = cv2.Laplacian(img,cv2.CV_16U,ksize=3)
```


Original Image
Laplacian - kernel 3 - cv2.CV_8U
Laplacian - kernel 3 - cv2.CV_16U

Unsharp Masking

Unsharp Masking

Steps

1. Blur the original image. We know by smoothing an image we suppress most of the high-frequency components.

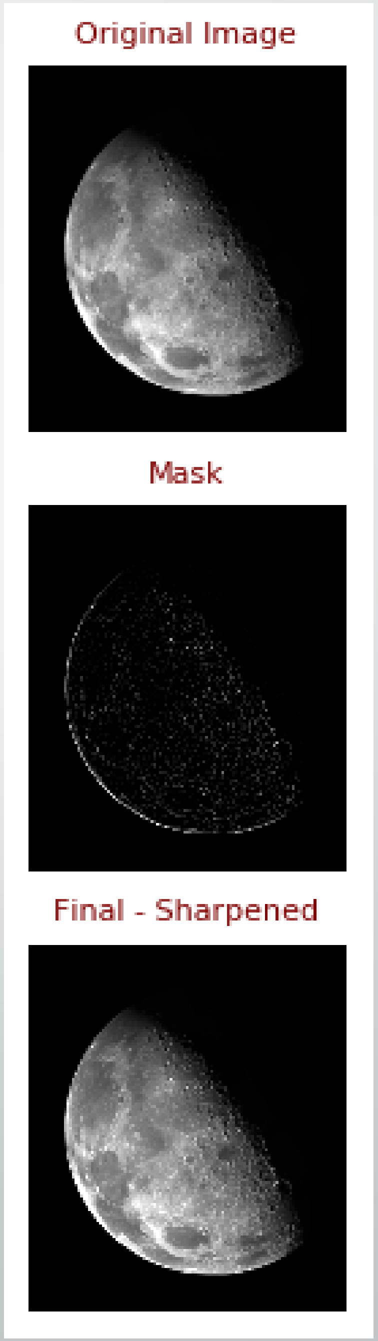
2. Subtract this smoothed image from the original image (the resulting difference is known as a mask).

3. Add this mask back to the original image [use **`addWeighted()`**]

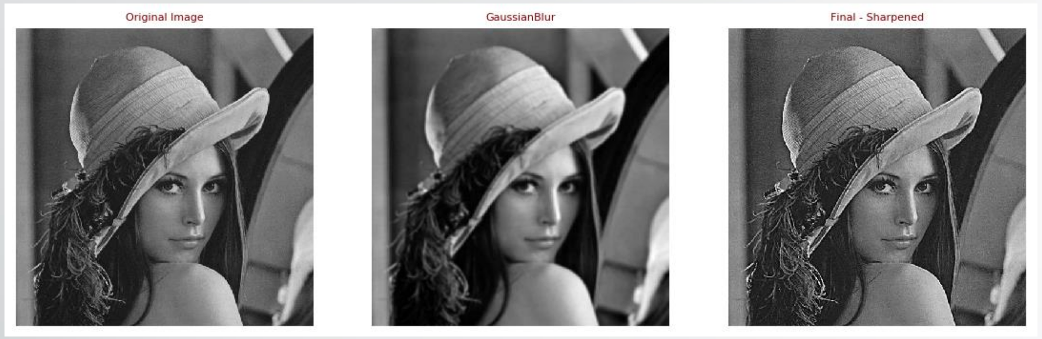
Ref: https://stackoverflow.com/questions/32454613/python-unsharp-mask

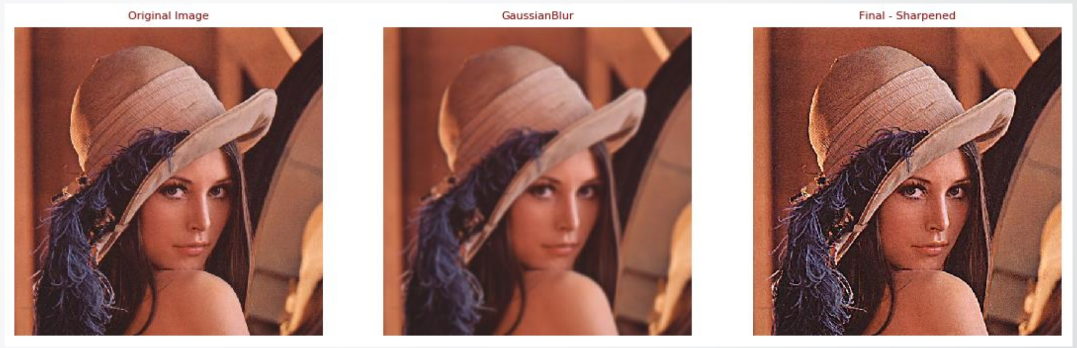
Unsharp Masking

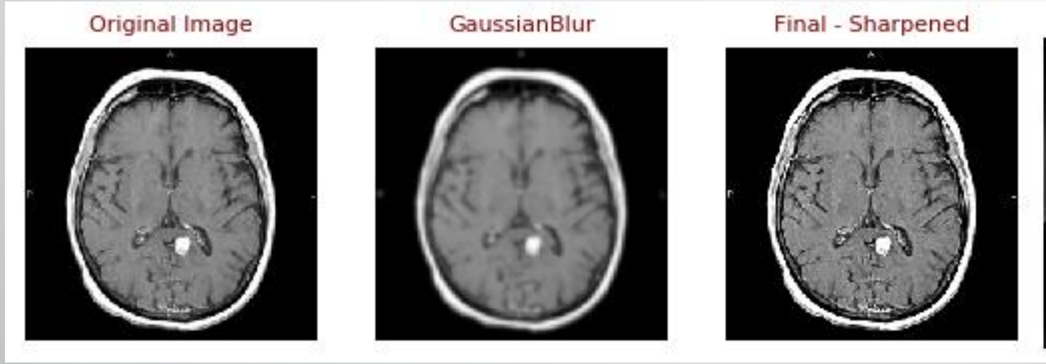
```
img = cv2.imread(r'images\blurry_moon.tif',0)

### Method 01
# Blur the image
gauss = cv2.GaussianBlur(img, (11,11), 0)
# Apply Unsharp masking
sharpened_image = cv2.addWeighted(img, 2, gauss, -1, 0)

### Method 02
# Blur the image - GaussianBlur
gauss = cv2.GaussianBlur(img, (13,13), 0)
# create the mask (laplacian)
mask = cv2.subtract(img,gauss)
# add mask and the orignal
sharpened_image = cv2.add(img, cv2.multiply(mask,0.95))
```

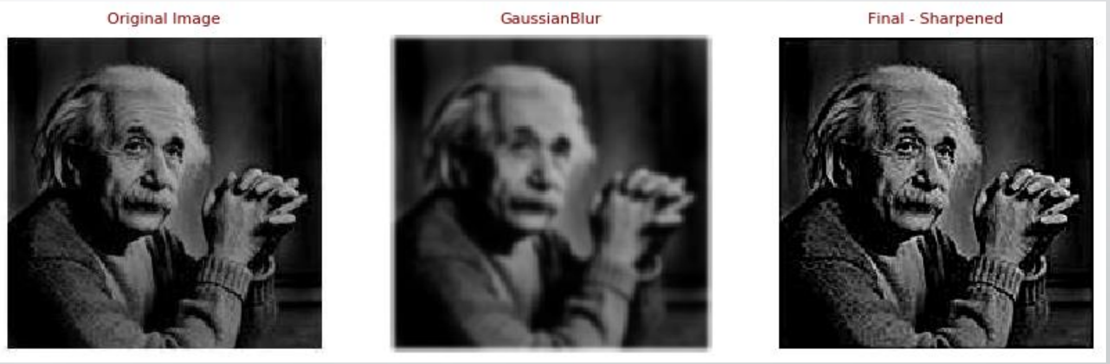


Unsharp Masking









– END –