Image Filtering - Spatial Filtering

Import required packages for image filtering.

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```python
ref = cv2.imread('/content/0a285c8b-1c31-48d4-89f2-af8b9edc36f6___RS_HL
5759.JPG')
plt.imshow(ref), plt.grid(False)

# while learning how to perform spatial filtering,
# you can also note how to apply different python commands.
plt.title('The original image')
plt.xticks([])
plt.yticks([])
plt.show()
```


There are two key factors in applying a filter on an image in digital image processing; **1)** the kernal type (and size), and **2)** the padding method (padding is the extrapolation procedure which describes the kernel output over the borders of the image).

There are different methods for defining the kernel and introducing the padding method in OpenCV. Unlike MATLAB that the kernel parameters are defined very easily, in OpenCV you have more control over the parameters and should take a longer way to reach the goal.

1-Averaging filter

Below, you see how to define a very simple averaging kernel and apply it on your images.

```
# Defining a kernel using numpy.
kernel_5 = np.ones((5,5),np.float32)/25
kernel_3 = np.ones((3,3),np.float32)/9

# Convolves an image with the kernel.
# -1 means that the center of the kernel is located on the center pixel.
# compare two kernel sizes.
filtered_5 = cv2.filter2D(ref,-1,kernel_5)
filtered_3 = cv2.filter2D(ref,-1,kernel_3)

# plot the results in two subplots.
fig=plt.figure(figsize=(14, 14), dpi= 80, facecolor='w', edgecolor='k')

plt.subplot(121), plt.imshow(filtered_3), plt.title('3-by-3 filter')
plt.grid(False)
plt.xticks([])
plt.yticks([])

plt.subplot(122), plt.imshow(filtered_5), plt.title('5-by-5 filter')
plt.grid(False)
plt.xticks([])
plt.yticks([])

plt.show()
```


The complete command for performing 2D spatial filter over images in OpenCV is cv2.filter2D with the following list of parameters. Some of the parameters are not necessarily used.

cv2.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])

src – input image.

ddepth – desired depth of the destination image; if it is negative, it will be the same as src.depth(); the following combinations of src.depth() and ddepth are supported:

src.depth() = CV_8U, ddepth = -1/CV_16S/CV_32F/CV_64F

src.depth() = CV_16U/CV_16S, ddepth = -1/CV_32F/CV_64F

src.depth() = CV_32F, ddepth = -1/CV_32F/CV_64F

src.depth() = CV_64F, ddepth = -1/CV_64F

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

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.

The anchor can be replaced by a single -1, indicating that the center of the kernel is on its middle pixel.

delta – optional value added to the filtered pixels before storing them in dst.

borderType – pixel extrapolation method (see borderInterpolate() for details).

Another simple way to apply a simple averaging filter is to use cv2.blur() function. The function can be applied as below:

In [5]:

```
# you can check the docs for further information.
blurred = cv2.blur(ref, (5, 5), -1)

plt.imshow(blurred), plt.grid(False), plt.xticks([]), plt.yticks([]),
plt.show()
```


Out[5]:

```
(,
 None,
 ([], ),
 ([], ),
 None)
```

2-Padding

As we mentioned earlier, one of the key factors in applying a filter over an image is to understand how to extrapolate the border values of the image while some parts of the kernel are dropped outside of the image border. "*borderType*" is the parameter which handles this functionality of kernels. Normally, the function is not called directly. It is used inside FilterEngine and copyMakeBorder() to compute tables for quick extrapolation. It means that we should first create the bordered (padded) image and then apply the filter over that image instead of the original image.

Various border types, image boundaries, are denoted with '|'

- BORDER_REPLICATE: *aaaaaa|abcdefgh|hhhhhhh*
- BORDER_REFLECT: *fedcba|abcdefgh|hgfedcb*
- BORDER_REFLECT_101: *gfedcb|abcdefgh|gfedcba*
- BORDER_WRAP: *cdefgh|abcdefgh|abcdefg*
- BORDER_CONSTANT: *iiiiii|abcdefgh|iiiiiii* with some specified 'i'

To create a bordered image we use the following command and args. Using this command you can insert different border sizes on each side of the image.

cv2.copyMakeBorder(src, top, bottom, left, right, borderType[, dst[, value]])

src – Source image.

Size(src.cols+left+right, src.rows+top+bottom)

top, bottom, left, right – Parameter specifying how many pixels in each direction from the source image rectangle to extrapolate. For example, top=1, bottom=1, left=1, right=1 mean that 1 pixel-wide border needs to be built.

borderType – Border type. One of the border types declared above.

value – Border value if borderType==BORDER_CONSTANT .

In [6]:

```
top = 10; bottom = 5; left = 20; right = 5
const = 100
img2 = cv2.copyMakeBorder(ref, top, bottom, left, right,
                          cv2.BORDER_WRAP)
img3 = cv2.copyMakeBorder(ref, top, bottom, left, right,
                          cv2.BORDER_REFLECT)
img4 = cv2.copyMakeBorder(ref, top, bottom, left, right,
                          cv2.BORDER_REPLICATE)
img5 = cv2.copyMakeBorder(ref, top, bottom, left, right,
                          cv2.BORDER_CONSTANT, const)

# Display the images
fig=plt.figure(figsize=(14, 14), dpi= 80, facecolor='w', edgecolor='k')

plt.subplot(221), plt.imshow(img2), plt.grid(False)
plt.xticks([]), plt.yticks([]), plt.title('wrap')

plt.subplot(222), plt.imshow(img3), plt.grid(False)
plt.xticks([]), plt.yticks([]), plt.title('reflect')

plt.subplot(223), plt.imshow(img4), plt.grid(False)
plt.xticks([]), plt.yticks([]), plt.title('replicate')

plt.subplot(224), plt.imshow(img5), plt.grid(False)
```
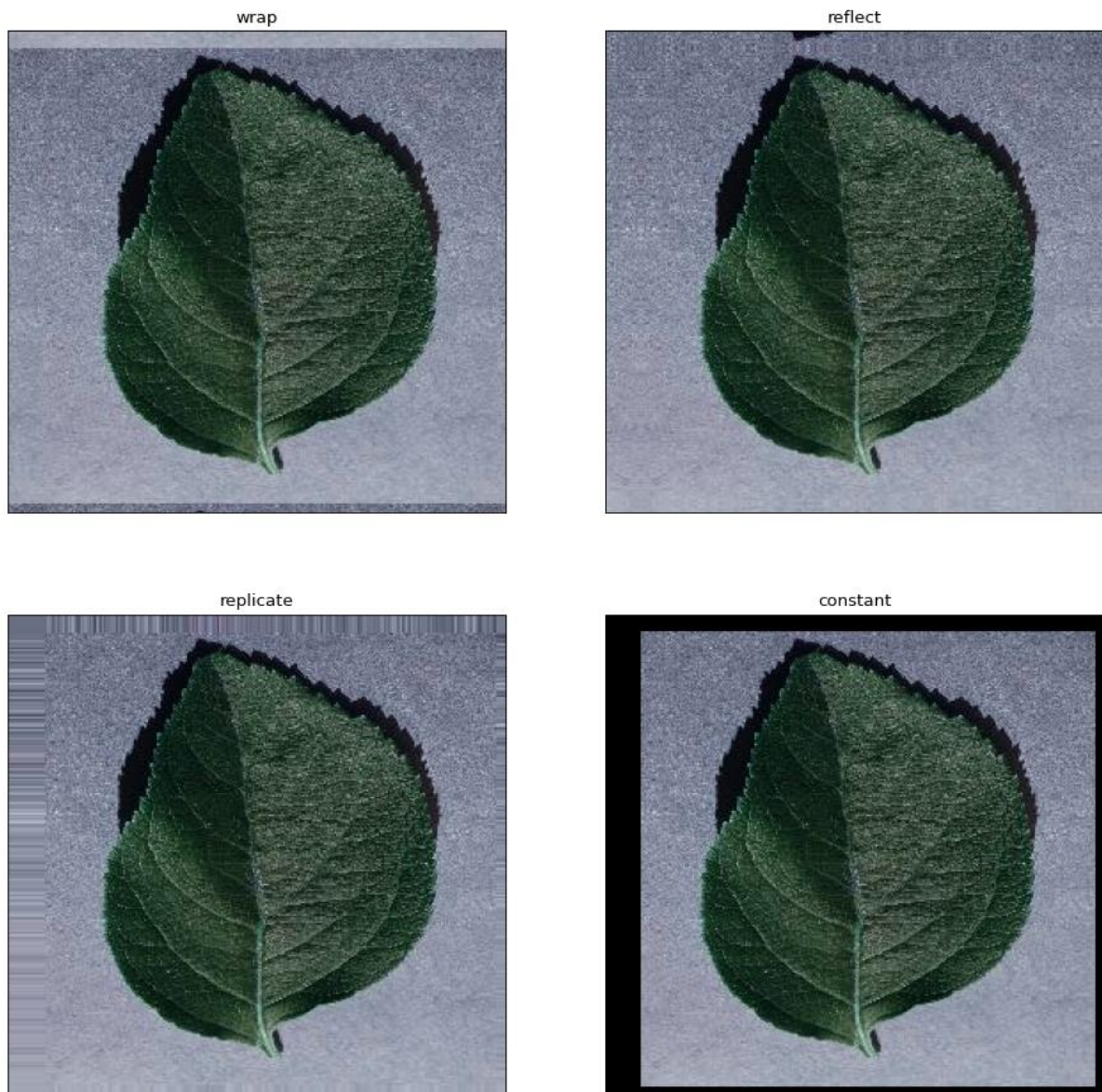
```
plt.xticks([]), plt.yticks([]), plt.title('constant')

plt.show()
```


3-User-defined kernels

One can apply user-defined kernels just like the way we applied in section 1. For example, consider applying the following kernel (which is a vertical gradient filter) on the image.

$[10-110-110-1]$

```
# vertical gradient kernel
# define a random kernel
vertical_gd = np.array([[1, 0, -1], [1, 0, -1], [1, 0, -1]])

# apply it.
filter_v = cv2.filter2D(ref[:, :, 2], -1, vertical_gd)
```

```
# show in a different colormap.
plt.imshow(filter_v, cmap='gray'), plt.grid(False)
plt.xticks([]), plt.yticks([])
```

```
((([], ),
  ([], ))
```


4-Median filtering

Median filter is a non-linear filter which is very useful in removing or decreasing the salt-and-pepper noise. The *medianBlur* function in OpenCV is designed to perform median filtering.

cv2.medianBlur(src, ksize[, dst])

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.

ksize – aperture linear size; it must be odd and greater than 1, for example: 3, 5, 7 ...

In order to better understand the effect of median filtering, we first add some noise to the image.

```
s_and_p = np.random.rand(ref.shape[0], ref.shape[1])

# if we consider 5% salt and pepper noise, we'd like to have
# 2.5% salt and 2.5% pepper. thus:
salt = s_and_p > .975
pepper = s_and_p < .025


# in order to add some noise, we should turn off black (pepper) locations
and
# turn on white (white) locations.
channel_2 = np.atleast_1d(ref[:, :, 1])
noisy = np.zeros_like(channel_2)
```

```
for i in range(channel_2.shape[0]*channel_2.shape[1]):
  if salt.ravel()[i] == 1:
    noisy.ravel()[i] = 255
  elif pepper.ravel()[i] == 1:
    noisy.ravel()[i] = 0
  else:
    noisy.ravel()[i] = channel_2.ravel()[i]

# apply median filter with size 3
Med = cv2.medianBlur(noisy, 3)

# Display the results
fig=plt.figure(figsize=(14, 14), dpi= 80, facecolor='w', edgecolor='k')
plt.subplot(121), plt.xticks([]), plt.yticks([])
plt.imshow(noisy, cmap='gray'), plt.grid(False)
plt.subplot(122), plt.xticks([]), plt.yticks([])
plt.imshow(Med, cmap='gray'), plt.grid(False)
plt.show()
```
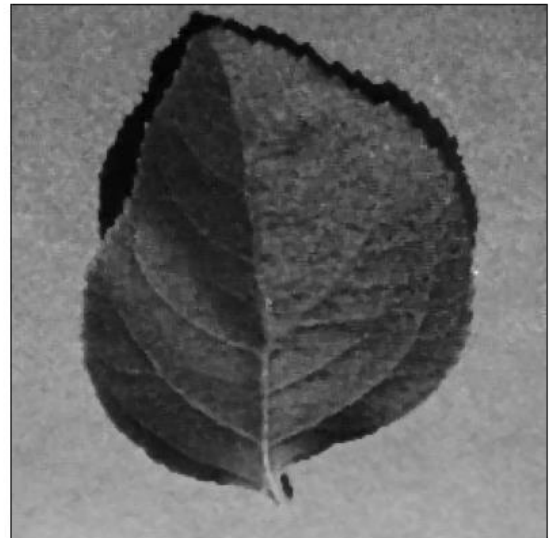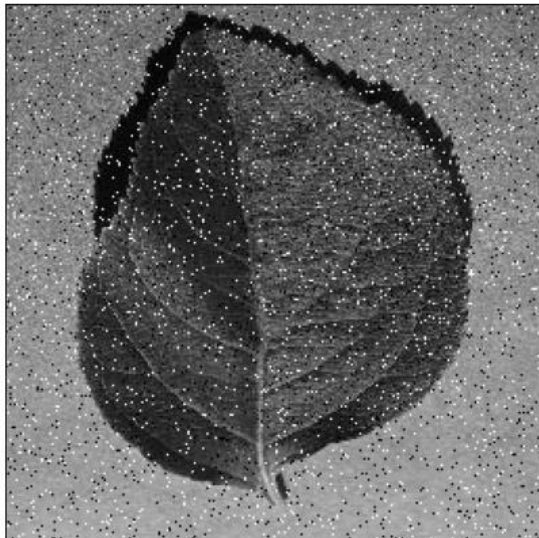

5- Gaussian filtering

There are multiple ways to perform a gaussian filtering over an image:

- using cv2.GaussianBlur() function,
- and, using cv2.sepFilter2D() after creating two separate horizontal and vertical kernels and applying them.

But, first of all, we need to have a noisy image; specifically with gaussian noise added to it. To create such an image we use Numpy to create random gaussian noise and then add it to the image.

```
# Creating random normal (gaussian) noise with pre-defined mean and std.
# The noisy image should be the size of the reference image.
mean = 0
sigma = 20.0
gauss_noise = np.random.normal(mean, sigma, (ref.shape[0], ref.shape[1]))
```

```python
# Convert RGB image to Grayscale image using cvtColor()
gray = cv2.cvtColor(ref, cv2.COLOR_BGR2GRAY)

# Add gaussian noise to the image
g_noisy = gray + gauss_noise # Gaussian noisy image

# Showing gray image, noise image, and noisy image
fig=plt.figure(figsize=(14, 14), dpi= 80, facecolor='w', edgecolor='k')
plt.subplot(131), plt.xticks([]), plt.yticks([])
plt.imshow(gray, cmap='gray'), plt.grid(False)
plt.subplot(132), plt.xticks([]), plt.yticks([])
plt.imshow(gauss_noise, cmap='gray'), plt.grid(False)
plt.subplot(133), plt.xticks([]), plt.yticks([])
plt.imshow(g_noisy, cmap='gray'), plt.grid(False)
```

(, None)
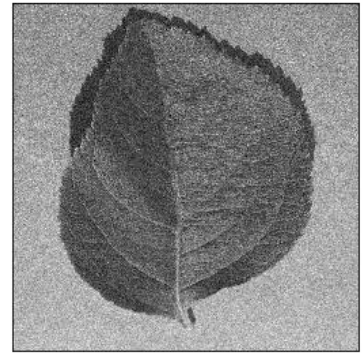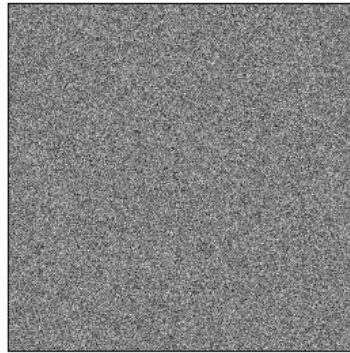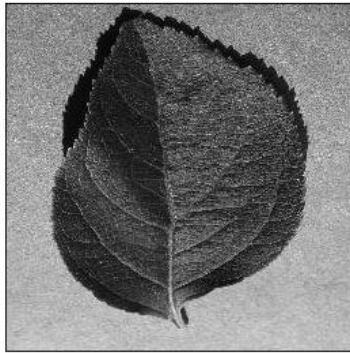
Now, we use simple cv2.GaussianBlur()to reduce gaussian noise in g_noisy image created above.

cv2.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]])

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.

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.
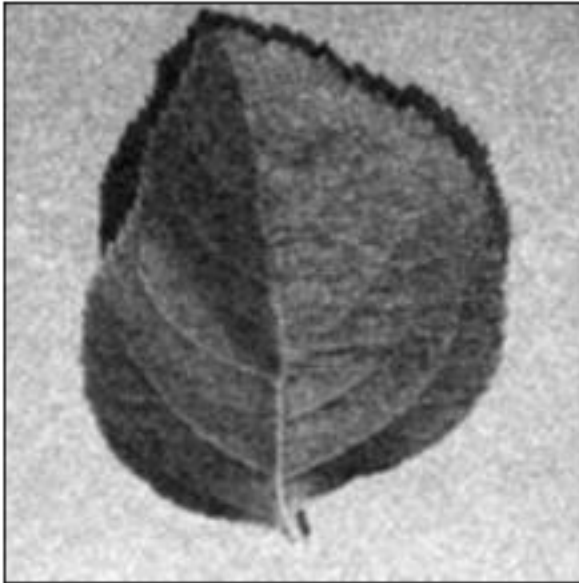
borderType – pixel extrapolation method.

```python
g_filtered = cv2.GaussianBlur(g_noisy, (3, 3), 20, 20)

# Display the result
plt.imshow(g_filtered, cmap='gray'), plt.grid(False)
plt.xticks([]), plt.yticks([])
```

```
(([], ),
 ([], ))
```


In order to use cv2.sepFilter2D() function, we should create a gaussian kernel. This is done using cv2.getGaussianKernel(). It creates 1-dimensional gaussian coefficients.

cv2.getGaussianKernel(ksize, sigma[, ktype])

ksize – Aperture size. It should be odd 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 .

```python
# Create a single gaussian kernel
g_kernel = cv2.getGaussianKernel(3, 20)
print(g_kernel)

# Apply two separate kernels over the image.
g_filtered_2 = cv2.sepFilter2D(g_noisy, -1, g_kernel, g_kernel)

# Displaying the results.
fig=plt.figure(figsize=(14, 14), dpi= 80, facecolor='w', edgecolor='k')
plt.subplot(121), plt.xticks([]), plt.yticks([]), plt.title('first method')
plt.imshow(g_filtered, cmap='gray'), plt.grid(False)
plt.subplot(122), plt.xticks([]), plt.yticks([]), plt.title('second
method')
plt.imshow(g_filtered_2, cmap='gray'), plt.grid(False)
```
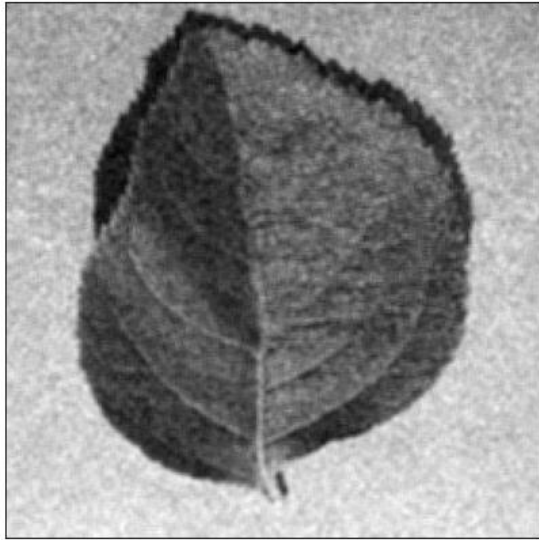
```
[[0.33319442]
 [0.33361117]
 [0.33319442]]
```
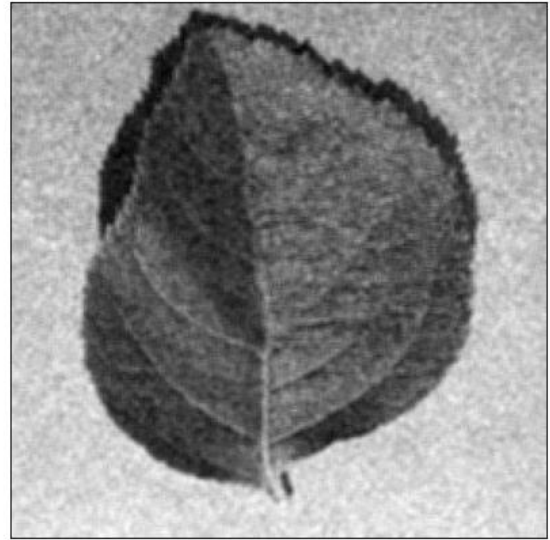
```
(, None)
```

first method   second method

Convolution and Padding described in this image