

Assignment #3

Derivative and Gaussian based Convolution Filters

In the previous assignment, we examined simple convolution filters that can be designed to do low pass, high pass filtering. The high pass filtered image can be combined with the original image to accomplish sharpening. We also derived first derivative based filters using the Taylor series expansion around a given pixel to approximate the first derivatives in the X and Y directions (G_x and G_y). Combining the G_x and G_y led to the Sobel filter. In this assignment, first we will take a look at the second derivative approximation around a given pixel. Again, we will use the Taylor series to approximate the second derivative of the image in terms of the values of the neighboring pixels. The second derivative based filter is referred to as the Laplacian filter, and often results in better change or edge detection.

Development of kernel for the Laplacian Filter:

Taylor series expansion in two dimensions up to the second derivative is given by:

$$P(x \pm \Delta x, y \pm \Delta y) = P(x, y) \pm \Delta x \frac{dP}{dx} \pm \Delta y \frac{dP}{dy} + 0.5(\Delta x)^2 \frac{d^2P}{dx^2} + 0.5(\Delta y)^2 \frac{d^2P}{dy^2}$$

Where $P(x, y)$ is the value of a pixel and x and y are the horizontal and vertical coordinates of the pixel. $P(x \pm \Delta x, y \pm \Delta y)$ is some pixel in the local neighborhood of the center pixel $P(x, y)$ and $(\Delta x, \Delta y)$ are the integer offsets of the neighborhood pixel from the center pixel. In a 3×3 neighborhood, the offsets are ± 1 as shown below.

$$(\pm \Delta x, \pm \Delta y) = \begin{bmatrix} (-1, +1) & (0, +1) & (+1, +1) \\ (-1, 0) & (0, 0) & (+1, 0) \\ (-1, -1) & (0, -1) & (+1, -1) \end{bmatrix}$$

In a 5×5 neighborhood, the closest pixel will have offsets of ± 1 and the outer pixels in the neighborhood will have offsets of ± 2 .

The terms $\frac{dP}{dx}$ and $\frac{dP}{dy}$ are the x and y first derivative filtered images and the terms $\frac{d^2P}{dx^2}$ and $\frac{d^2P}{dy^2}$ are the x and y second derivative filtered images.

The definitions for the Gradient and Laplacian filtered images are as follows:

$$\text{Gradient (Magnitude) Filtered Image} = \sqrt{\left(\frac{dP}{dx}\right)^2 + \left(\frac{dP}{dy}\right)^2}$$

$$\text{Laplacian Filtered Image} = \frac{d^2P}{dx^2} + \frac{d^2P}{dy^2}$$

To come up with the Laplacian kernel, we can take a look at a 3x3 pixel window and do the Taylor series expansion for each of the pixel left, bottom, right and top (shown in bold below) with respect to the center pixel, we will come up with the following four equations.

$$(\pm\Delta x, \pm\Delta y) = \begin{bmatrix} (-1, +1) & (\mathbf{0}, +1) & (+1, +1) \\ (-\mathbf{1}, \mathbf{0}) & (0, 0) & (+\mathbf{1}, \mathbf{0}) \\ (-1, -1) & (\mathbf{0}, -1) & (+1, -1) \end{bmatrix}$$

$$P(x-1, y) = P(x, y) - \frac{dP}{dx} + 0.5 \frac{d^2P}{dx^2} \quad (1)$$

$$P(x, y-1) = P(x, y) - \frac{dP}{dy} + 0.5 \frac{d^2P}{dy^2} \quad (2)$$

$$P(x+1, y) = P(x, y) + \frac{dP}{dx} + 0.5 \frac{d^2P}{dx^2} \quad (3)$$

$$P(x, y+1) = P(x, y) + \frac{dP}{dy} + 0.5 \frac{d^2P}{dy^2} \quad (4)$$

If we add these four equations, we notice that the first derivatives cancel out and we are left with

$$\text{Sum of Four neighbors} = 4P(x, y) + \frac{d^2P}{dx^2} + \frac{d^2P}{dy^2}$$

Or:

$$\text{Laplacian Filtered Image} = \frac{d^2P}{dx^2} + \frac{d^2P}{dy^2} = \text{four neighbors} - 4P(x, y)$$

Which results in a kernel of

$$\text{Laplacian Filter} = \frac{d^2}{dx^2} + \frac{d^2}{dy^2} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\text{4-neighbor Laplacian Filter} = \frac{d^2}{dx^2} + \frac{d^2}{dy^2} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Similarly an eight neighbor Laplacian filter can be developed to yield a kernel of:

$$\text{8-neighbor Laplacian Filter} = \frac{d^2}{dx^2} + \frac{d^2}{dy^2} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

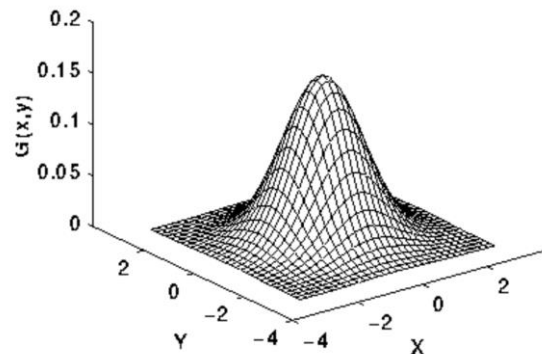
Problem #1: Derive the 8-neighbor Laplacian filter kernel. Show all the equations.

Problem #2: Write a Python function that returns the 2-d Gaussian kernel with specified standard deviation and kernel size. Then test the Gaussian kernel convolution on an image with different values of standard deviation.

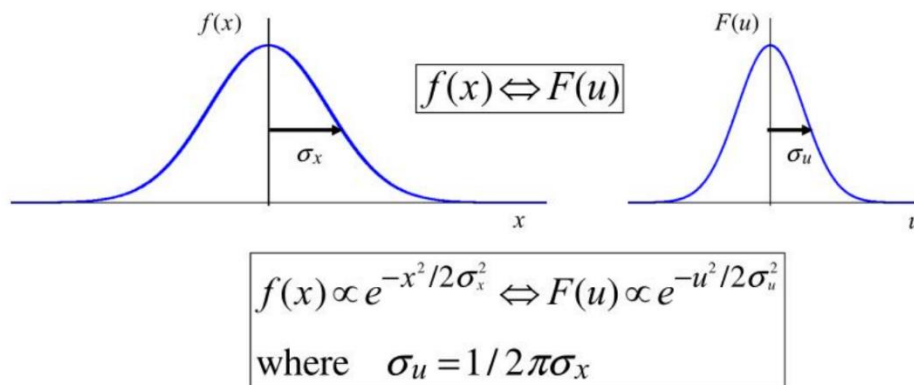
Partial Solution: 2-d Gaussian is defined as:

$$G_{\sigma}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

The 2-d Gaussian function appears as:



Note that a Gaussian kernel behaves as a low pass filter. This is because, the Fourier transform of a Gaussian function is a Gaussian itself with the standard deviation getting inverted in the frequency domain as shown for a 1-d signal below.



Notes:

- The Inverse Relationship – narrow function in the spatial domain results in a wide function in the Fourier Domain.

Thus if you choose a high standard deviation in the pixel domain, it will suppress more high frequencies. You will verify this by writing the Python program to determine the Gaussian kernel and then doing the convolution of it with an image by choosing different values of standard deviation.

Create a Python application called AdvancedConvolutionFilters. Add a python file to the project called Utils with the following code in it.

```
import numpy as np
import math

def compute_gaussian_kernel(kernel_size, sigma):
    kernel = np.zeros((kernel_size, kernel_size), dtype=float)
    for x in range(-kernel_size//2+1, kernel_size//2+1):
        for y in range(-kernel_size//2+1, kernel_size//2+1):
            kernel[x+kernel_size//2, y+kernel_size//2] =
(1/(2*math.pi*sigma**2))*math.exp(-((x**2+y**2)/(2*sigma**2)))
    kernel = kernel/np.min(kernel)
    return kernel, np.sum(kernel)
```

Note that the sum of the values in the kernel should equal 1 (or 0) so that it does not affect the scale (contrast) of the image. The sum of the values of the kernel is returned by the above function so that we can divide each kernel value by it to accomplish sum of kernel values being 1.

Add a file called MyConvolution.py with the following code in it:

```
import numpy as np

class MyConvolution(object):
    def convolve(self, img: np.array, kernel: np.array) -> np.array:
        # kernel is assumed to be square
        output_size = (img.shape[0]-kernel.shape[0]+1, img.shape[1]-
kernel.shape[0]+1)

        output_img = np.zeros((output_size[0],
output_size[1], img.shape[2]), dtype=img.dtype)
        kernel_size = kernel.shape[0] # kernel size

        for i in range(output_size[0]):
            for j in range(output_size[1]):
                for k in range(img.shape[2]): # RGB
                    mat = img[i:i+kernel_size, j:j+kernel_size, k] # values at
current kernel location
                    mat = mat.reshape((kernel_size, kernel_size))
                    # do element-wise multiplication and add the result
                    output_img[i, j, k] = np.clip(np.sum(np.multiply(mat,
kernel)), 0, 255)

        return output_img
```

Type the following code in Advanced ConvolutionFilters.py to test the convolution with the Gaussian kernel.

```

import sys
import cv2
from MyConvolution import MyConvolution
import numpy as np
import Utils

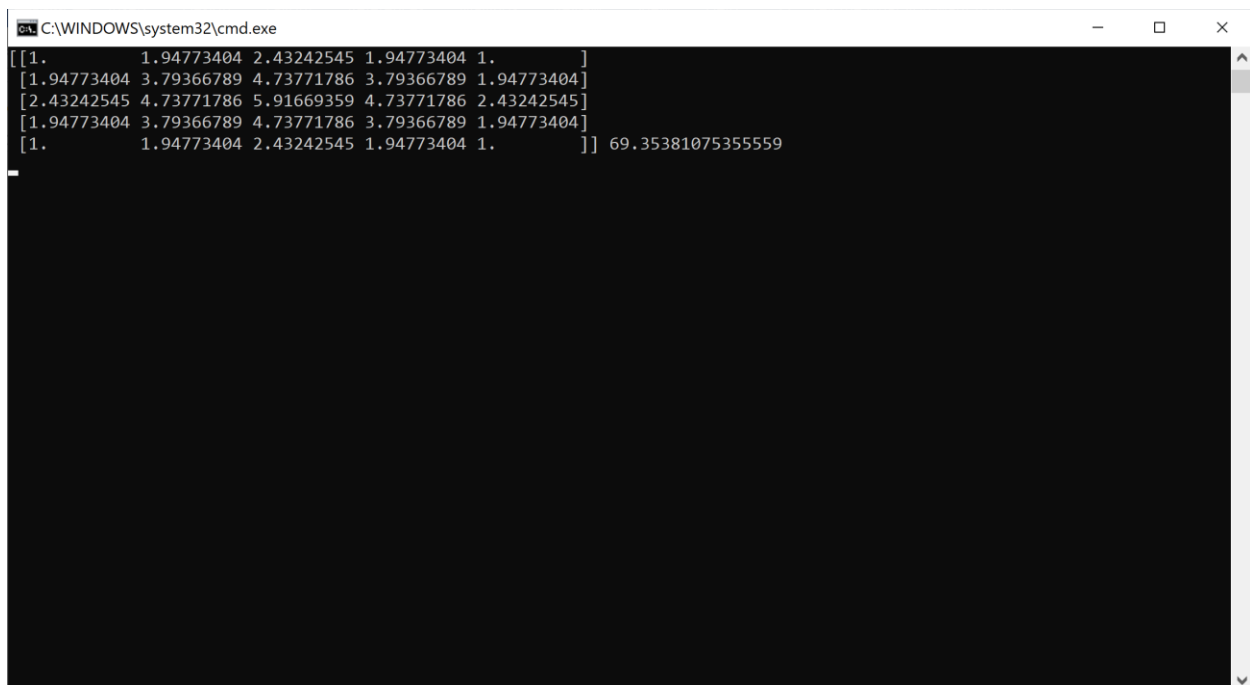
def main():
    g,denom = Utils.compute_gaussian_kernel(5,1.5)
    print(g, denom)
    kernel = g/denom
    image_filename = 'd:/temp/orig3.jpg'
    img = cv2.imread(image_filename)
    if img is None:
        print('Could not open or find the image: ', image_filename)
        exit(0)

    myconv = MyConvolution()
    conv_img = myconv.convolve(img,kernel)
    cv2.imshow('Original Image', img)
    cv2.imshow('Convolved Image', conv_img)
    cv2.waitKey()

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

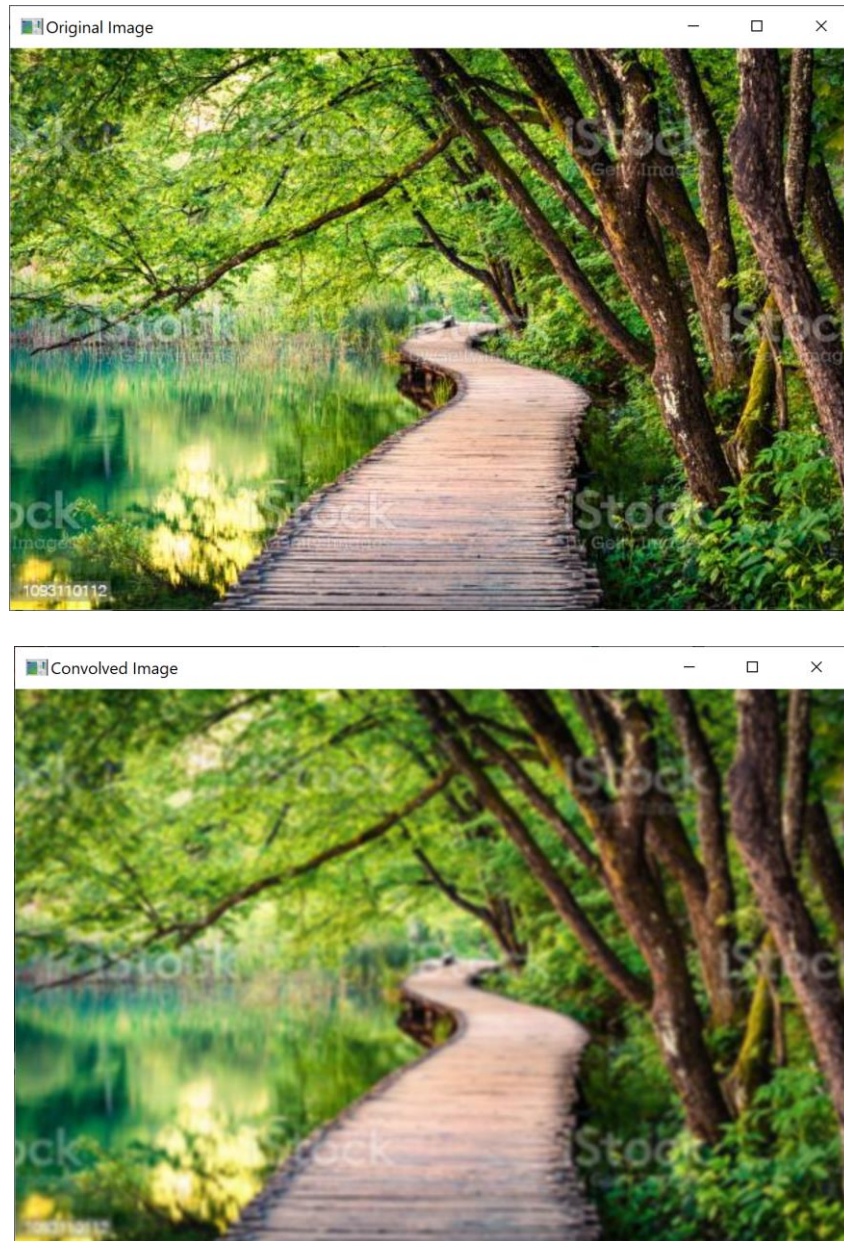
If you run the program, you can see that the Gaussian kernel blurs the image (low pass filter).



```

C:\WINDOWS\system32\cmd.exe
[[1.          1.94773404  2.43242545  1.94773404  1.          ]
 [1.94773404  3.79366789  4.73771786  3.79366789  1.94773404]
 [2.43242545  4.73771786  5.91669359  4.73771786  2.43242545]
 [1.94773404  3.79366789  4.73771786  3.79366789  1.94773404]
 [1.          1.94773404  2.43242545  1.94773404  1.          ]] 69.35381075355559

```

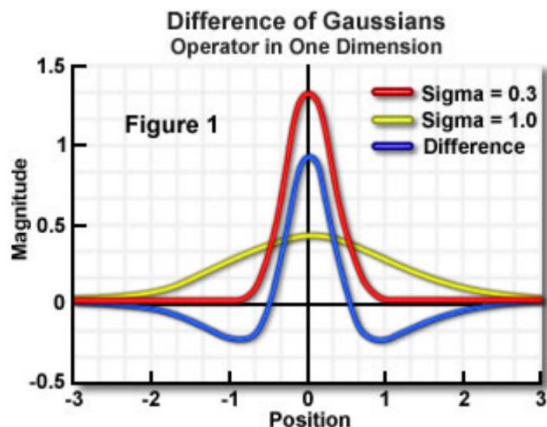


Experiment with different values of standard deviation, e.g., 0.5, 1, 1.5, 3. Which one of these values causes the most blurriness (more strict low pass filtering or noise removal)?

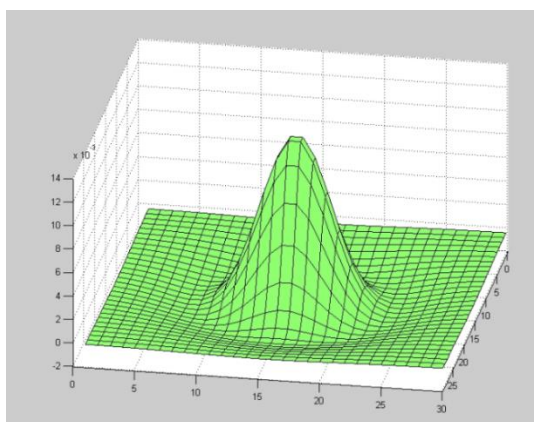
Problem #3: Program the difference of Gaussian (DoG) kernels and test it for different values of σ_1 and σ_2 .

Since a Gaussian is a low pass filter whose cut off frequency depends upon the value of σ , the difference of Gaussian ends up being a band-pass filter, where the band depends upon the value of σ_1 and σ_2 .

In 1-d, the difference of Gaussians can be observed as:



By observing the difference (blue line) in the above figure, you can guess that the 2-d kernel will produce positive and negative values (unlike a 2-d Gaussian by itself which produces positive values). It has been conjectured that the human vision system uses a differencing approach perhaps similar to DoG (Ref: <https://muse.union.edu/visualmotion/center-surround/>). In 2-D, the DoG kernel appears as:



Modify the Utils.py file to appear as (division by the minimum value has been removed in compute_gaussian_kernel function).

```
import numpy as np
import math

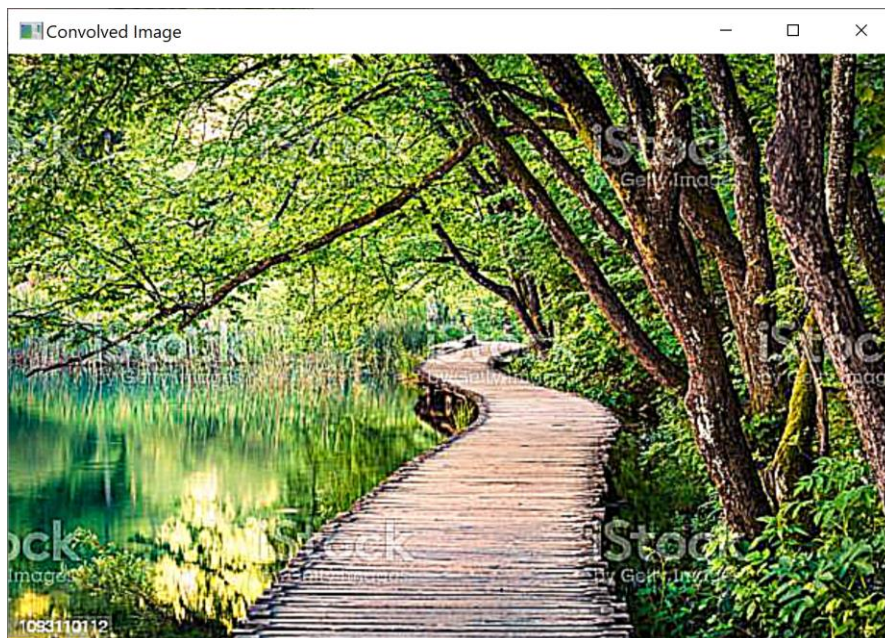
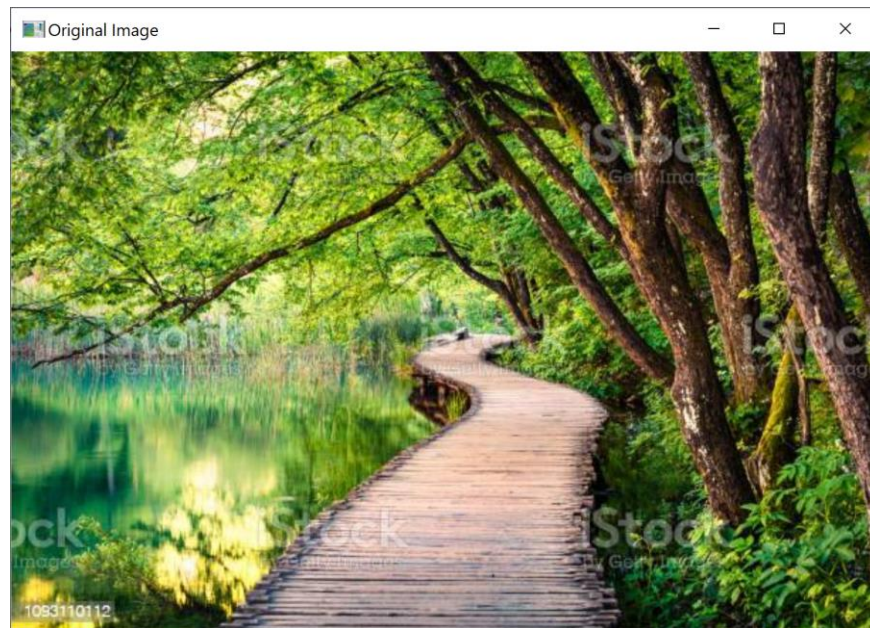
def compute_gaussian_kernel(kernel_size, sigma):
    kernel = np.zeros((kernel_size, kernel_size), dtype=float)
    for x in range(-kernel_size//2+1, kernel_size//2+1):
        for y in range(-kernel_size//2+1, kernel_size//2+1):
            kernel[x+kernel_size//2, y+kernel_size//2] =
(1/(2*math.pi*sigma**2))*math.exp(-((x**2+y**2)/(2*sigma**2)))
    kernel = kernel
    return kernel, np.sum(kernel)

def compute_DoG_kernel(kernel_size, sigma1, sigma2):
    kernel1, sum1 = compute_gaussian_kernel(kernel_size, sigma1)
    kernel2, sum2 = compute_gaussian_kernel(kernel_size, sigma2)
    kernel = kernel1 - kernel2
    return kernel, np.sum(kernel)
```


Modify the first part of main in AdvancedConvolutionFilters.py to appear as:

```
def main():  
    #g,denom = Utils.compute_gaussian_kernel(5,1.5)  
    #print(g, denom)  
    #kernel = g/denom  
    dog, denom = Utils.compute_DoG_kernel(5, 0.5, 1.5)  
    print(dog, denom)  
    kernel = dog/denom
```

The result on the test image appears as:



Problem #4: Program the Laplacian of Gaussian (LoG) and test it in some images for different parameters.

As explained in the lecture, the Laplacian is defined as:

$$\frac{d^2 P}{dx^2} + \frac{d^2 P}{dy^2}$$

LoG starts out by applying the Gaussian filtering first to remove the noise and then applies the Laplacian i.e., sum of the second derivatives in the x and y direction. We can obtain the closed form for the LoG as:

$$LoG(x, y) = -\frac{1}{\pi\sigma^4} \left[1 - \frac{x^2 + y^2}{2\sigma^2} \right] e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

This equation can be easily programmed.

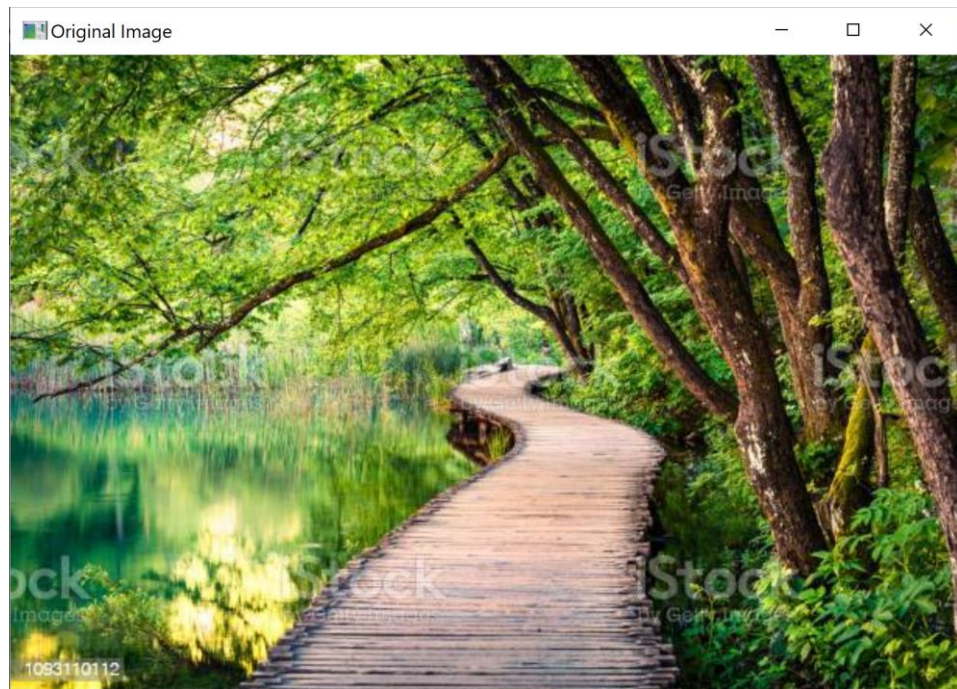
Add another function to the Utils.py to compute the LoG as:

```
def compute_Log_kernel(kernel_size, sigma):
    kernel = np.zeros((kernel_size, kernel_size), dtype=float)
    for x in range(-kernel_size//2+1, kernel_size//2+1):
        for y in range(-kernel_size//2+1, kernel_size//2+1):
            kernel[x+kernel_size//2, y+kernel_size//2] = ((-
1/(math.pi*sigma**4))
                *(1-((x**2+y**2)/(2*sigma**2)))*math.exp(-
((x**2+y**2)/(2*sigma**2))))
    return kernel, np.sum(kernel)
```

Write the test code in the main function to test the LoG as:

```
#g,denom = Utils.compute_gaussian_kernel(5,1.5)
#print(g, denom)
#kernel = g/denom
#dog, denom = Utils.compute_DoG_kernel(5, 0.5, 1.5)
#print(dog, denom)
#kernel = dog/denom
LoG,denom = Utils.compute_Log_kernel(5,1.0)
print(LoG, denom)
kernel = LoG/denom
```

Depending on the value of sigma, the output will produce different enhancement of edges because of the second derivatives involved.



After LoG with sigma of 1.0, the image appears as:

