Visión Computacional para imágenes y video (Gpo 10)

Alumnos:

- Armando Bringas Corpus - A01200230
- Guillermo Alfonso Muñiz Hermosillo - A01793101
- Jorge Luis Arroyo Chavela - A01793023
- Samantha R Mancias Carrillo - A01196762
- Sofia E Mancias Carrillo - A01196563

Profesores:

- Dr. Gilberto Ochoa Ruiz
- Mtra. Yetnalezi Quintas Ruiz

4. Image Convolution

Table of Contents

Importing Libraries

In [26]:
```
import cv2
import math
import matplotlib.pyplot as plt
import numpy as np
```

```
from PIL import Image
from scipy.ndimage.filters import median_filter
```

Simple Convolution

Definition

- **I**: Image to convolve.
- **H**: filter matrix to convolve the image with.
- **J**: Result of the convolution.

The following graphics shows exemplary the mathematical operations of the convolution. The filter matrix **H** is shifted over the input image **I**. The values 'under' the filter matrix are multiplicated with the corresponding values in **H**, summed up and writen to the result **J**. The target position is usually the position under the center of **H**.

In order to implement the convolution with a block filter, we need two methods. The first one will create the block filter matrix **H** depending on the filter width/height **n**.

A block filter holds the value $\frac{1}{n \cdot n}$ at each position:

In [2]:
```
def block_filter(n):
    H = np.ones((n, n)) / (n * n)  # each element in H has the value 1/(n*n)
    return H
```

We will test the method by creating a filter with `n = 5`:

In [5]:
```
H = block_filter(5)
print(H)
```

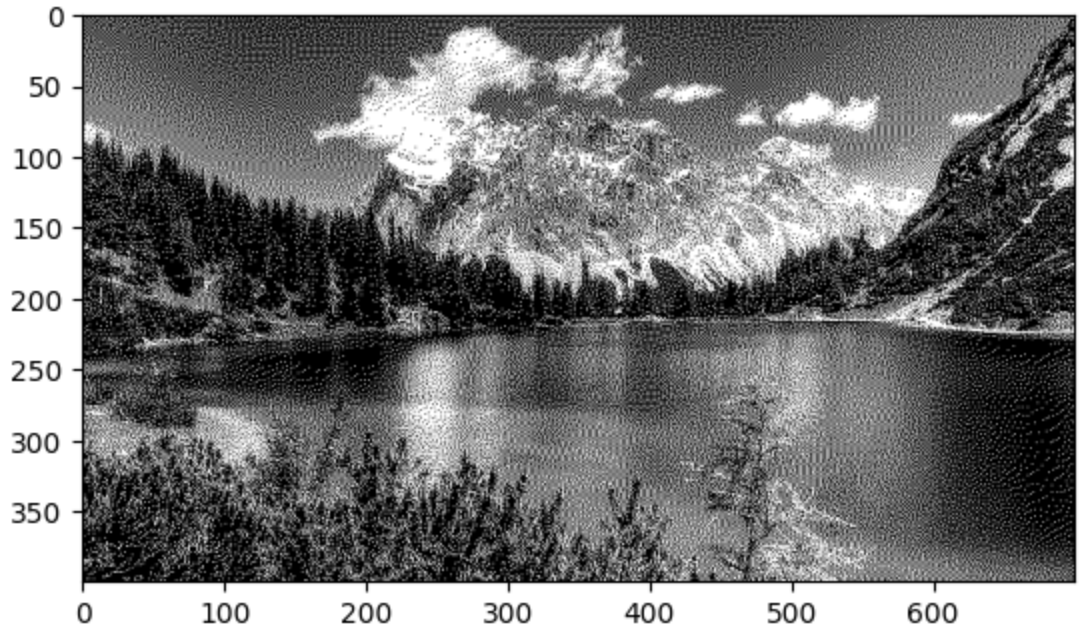
```
[[0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]
 [0.04 0.04 0.04 0.04 0.04]]
```

Next, we define the actual convolution operation. To prevent invalid indices at the border of the image, we introduce the padding **p**.

In [6]:
```
def apply_filter(I, H):
    h, w = I.shape                          # image dimensions (height, width)
    n = H.shape[0]                          # filter size
    p = n // 2                              # padding size
    J = np.zeros_like(I)                    # output image, initialized with zeros

    for x in range(p, h-p):
        for y in range(p, w-p):
            J[x, y] = np.sum(I[x-p:x+n-p, y-p:y+n-p] * H)
    return J
```

In [7]:
```
image = Image.open('data/image.jpg')
image = image.convert('1') # convert image to black and white

image = np.array(image)
```

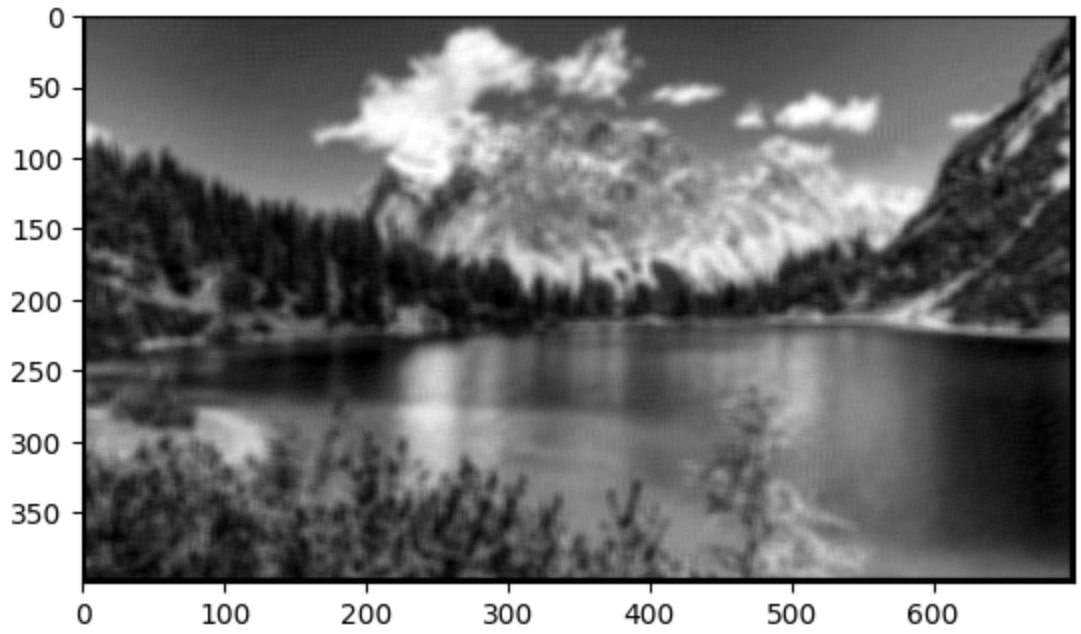
```
# image = np.zeros((200, 200), dtype=np.float)
# for x in range(200):
#     for y in range(200):
#         d = ((x-100)**2+(y-100)**2)**0.5
#         image[x, y] = d % 8 < 4

plt.imshow(image, cmap='gray',vmin=0.0, vmax=1.0)
plt.show()
```



In [8]:
```
image = image.astype(float)
```

Next we test our implementation and apply a block filter with size 7

In [9]:
```
n = 7
H = block_filter(n)
J = apply_filter(image, H)

plt.imshow(J, cmap='gray')
plt.show()
```



PyTorch Convolution

In [10]:
```
from PIL import Image

img = Image.open('data/image.jpg')
img.thumbnail((256,256), Image.ANTIALIAS) # Resize to half to reduce the size of this no
```

In [11]:
```
img
```

Out[11]:

In [13]:
```
import torch, torchvision
from torchvision import transforms
from torch import nn
```

In [14]:
```
to_tensor = transforms.Compose([
   transforms.Grayscale(),  # Convert image to grayscale.
   transforms.ToTensor()    # Converts a PIL Image in the range [0, 255] to a torch.Floa
])

to_pil = transforms.Compose([
    transforms.ToPILImage()
])
```

In [15]:
```
input = to_tensor(img)
input.shape
```

Out[15]:
```
torch.Size([1, 146, 256])
```

In [16]:
```
to_pil(input)
```

Out[16]:

2D convolution over an input image:

- `in_channels = 1` : an input is a grayscale image
- `out_channels = 1` : an output is a grayscale image
- `kernel_size = (3, 3)` : the kernel (filter) size is 3 x 3
- `stride = 1` : the stride for the cross-correlation is 1
- `padding = 1` : zero-paddings on both sides for 1 point for each dimension
- `bias = False` : no bias parameter (for simplicity)

In [17]:
```
conv = nn.Conv2d(1, 1, (3, 3), stride=1, padding=1, bias=False)
```

In [18]:
```
# The code below does not work because the convolution layer requires the dimension for
conv(input)
```

Out[18]:
```
tensor([[[-0.0008,  0.0100,  0.0099,  ...,  0.0122,  0.0371,  0.1503],
         [ 0.0017,  0.0805,  0.0813,  ...,  0.1232,  0.1750,  0.1756],
         [ 0.0017,  0.0849,  0.0836,  ...,  0.1757,  0.1461,  0.1435],
         ...,
         [ 0.0097, -0.0294,  0.0751,  ...,  0.0902,  0.0812,  0.2132],
         [-0.0401,  0.0466,  0.0898,  ...,  0.0967,  0.0887,  0.2387],
         [-0.0362, -0.0026, -0.0170,  ..., -0.0224, -0.0267,  0.0562]]],
       grad_fn=<SqueezeBackward1>)
```

We need to insert a dimension for a batch at dim=0.

In [19]:
```
input = input.unsqueeze(0)
input.shape
```

Out[19]:
```
torch.Size([1, 1, 146, 256])
```

In [20]:
```
output = conv(input)
output.shape
```

Out[20]:
```
torch.Size([1, 1, 146, 256])
```

Setting `padding=1` in the convolution layer, we obtain an image of the same size.

In [21]:
```
output.shape
```

Out[21]:
```
torch.Size([1, 1, 146, 256])
```

We need to remove the first dimension before converting to a PIL object.

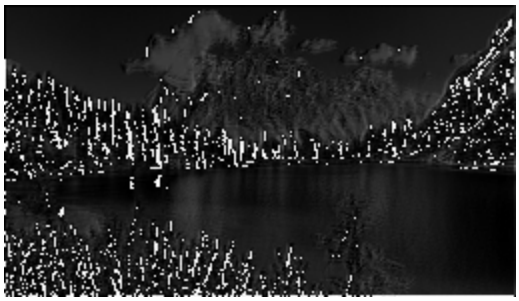
In [22]:
```
output.data.squeeze(dim=0).shape
```

Out[22]:
```
torch.Size([1, 146, 256])
```

Display the output from the convolution layer by converting `output` to a PIL object.

In [23]:
```
to_pil(output.data.squeeze(dim=0))
```

Out[23]:



Clip every value in the output tensor within the range of [0, 1].

In [24]:
```
to_pil(torch.clamp(output, 0, 1).data.squeeze(dim=0))
```

Out[24]:

In [25]:
```
def display(img1, img2):
    im1 = to_pil(torch.clamp(img1, 0, 1).data.squeeze(dim=0))
    im2 = to_pil(torch.clamp(img2, 0, 1).data.squeeze(dim=0))
    dst = Image.new('RGB', (im1.width + im2.width, im1.height))
    dst.paste(im1, (0, 0))
    dst.paste(im2, (im1.width, 0))
    return dst
```

In [26]:
```
display(input, output)
```

Out[26]:

Identity

In [27]:
```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 1, 0.],
    [0., 0., 0.],
]]])

output = conv(input)
display(input, output)
```

Out[27]:

Brighten

In [30]:
```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 1.5, 0.],
    [0., 0., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.0000, 0.0000, 0.0000],
```

```
           [0.0000, 1.5000, 0.0000],
           [0.0000, 0.0000, 0.0000]]]])
```

Out[30]:

Darken

In [31]:
```
conv.weight.data = torch.tensor([[[
    [0., 0., 0.],
    [0., 0.5, 0.],
    [0., 0., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.0000, 0.0000, 0.0000],
          [0.0000, 0.5000, 0.0000],
          [0.0000, 0.0000, 0.0000]]]])
```

Out[31]:

Box blur

In [32]:
```
conv.weight.data = torch.ones((1, 1, 3,3), dtype=torch.float) / 9.
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.1111, 0.1111, 0.1111],
          [0.1111, 0.1111, 0.1111],
          [0.1111, 0.1111, 0.1111]]]])
```

Out[32]:

Gaussian blur

In [33]:
```
conv.weight.data = torch.tensor([[[
    [1., 2., 1.],
    [2., 4., 2.],
    [1., 2., 1.],
```

```
]]])/16.
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[0.0625, 0.1250, 0.0625],
          [0.1250, 0.2500, 0.1250],
          [0.0625, 0.1250, 0.0625]]]])
```

Out[33]:

Sharpen

In [34]:

```
conv.weight.data = torch.tensor([[[
    [0., -1., 0.],
    [-1., 5., -1.],
    [0., -1., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[ 0., -1.,  0.],
          [-1.,  5., -1.],
          [ 0., -1.,  0.]]]])
```

Out[34]:

In [35]:

```
conv.weight.data = torch.tensor([[[
    [0., -2., 0.],
    [-2., 10., -2.],
    [0., -2., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[ 0., -2.,  0.],
          [-2., 10., -2.],
          [ 0., -2.,  0.]]]])
```

Out[35]:

Edge detection

In [36]:
```
conv.weight.data = torch.tensor([[[
    [0., 1., 0.],
    [1., -4., 1.],
    [0., 1., 0.],
]]])
print(conv.weight.data)
output = conv(input)
display(input, output)
```

```
tensor([[[[ 0.,  1.,  0.],
          [ 1., -4.,  1.],
          [ 0.,  1.,  0.]]]])
```

Out[36]:

In [37]:
```
conv.weight.data = torch.tensor([[[
    [-1., -1., -1.],
    [-1., 8., -1.],
    [-1., -1., -1.],
]]])
output = conv(input)
display(input, output)
```

Out[37]:

Excercises

1. Edge detection algorithm implementation

In [3]:
```
# Function to plot images
def plot_images(orig_img, trans_img):
    plt.figure(figsize=([20, 20]))
    plt.subplot(121),plt.imshow(orig_img)
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(trans_img)
    plt.title('Transformed Image'), plt.xticks([]), plt.yticks([])

# Function to plot original and grayscaled Image
def plot_images_gray(orig_img, gray_img):
    plt.figure(figsize=([20, 20]))
    plt.subplot(121),plt.imshow(orig_img)
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(gray_img, cmap=plt.get_cmap('gray'))
    plt.title('Grayscale Image'), plt.xticks([]), plt.yticks([])
```

```
# Function to plot original, grayscaled and Edge Detected Images
def plot_images_edgeDetection(orig_img, gray_img, edge_img):
    plt.figure(figsize=([40, 40]))
    plt.subplot(131),plt.imshow(orig_img)
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(132),plt.imshow(gray_img, cmap=plt.get_cmap('gray'))
    plt.title('Grayscale Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(133),plt.imshow(edge_img, cmap=plt.get_cmap('gray'))
    plt.title('Edge Detected Image'), plt.xticks([]), plt.yticks([])
```

Prewitt

In [66]:

```
# 0. Load the original image:
img = Image.open('./data/tiger.jpeg')

# 1.0 Transform image to a numpy array:
img = np.array(img)

# 2.1. Extract the red, green, and blue channels:
r, g, b = img[:,:,0], img[:,:,1], img[:,:,2]

# 2.2. Scalars to convert the image to grayscale:
grey_scalars = [0.2989, 0.5870, 0.1140]

# 3.0. We apply the scalar to the channels to create the grey-scale image (this is our s
greyimg = grey_scalars[0] * r + grey_scalars[0] * g +grey_scalars[0] * b

# 4.0. Let's define the kernels of Prewitt:
Gx = np.array([
    [-1, 0, 1],
    [-1, 0, 1],
    [-1, 0, 1]
])

Gy = np.array([
    [-1, -1, -1],
    [ 0,  0,  0],
    [ 1,  1,  1]
])

# 5.0. Let's go row by row and then col by col (nested loop),
# applying the mask of the prewitt kernel on X and Y over the original image to get the

# 5.1. First let's get the image dimensions:
rows, cols = greyimg.shape

# 5.1. Then let's create an empty image:
prewittimg = np.zeros((rows, cols))

# 5.2 Let's loop through the image rows:
for i in range(1, rows-1):
  # 5.3. And let's loop through the columns:
        for j in range(1, cols-1):
            # Apply the x kernel to the current pixel
            x_edge = np.sum(greyimg[i-1:i+2, j-1:j+2] * Gx)
            # Apply the y kernel to the current pixel
            y_edge = np.sum(greyimg[i-1:i+2, j-1:j+2] * Gy)
            # Compute the gradient magnitude
            prewittimg[i, j] = np.sqrt(x_edge**2 + y_edge**2)

# 6.0 voila!

## Output:
plot_images_edgeDetection(img, greyimg, prewittimg)
```

Sobel

In [16]:
```
# First we read our image file and converted to gray scale.
img = Image.open('data/tiger.jpeg')

# Then we convert our image to a numpy array where the height, width and colors are sepa
img = np.array(img)

# Then we separate the RGB colors on individual variables.
redCh, greenCh, blueCh = img[:, :, 0], img[:, :, 1], img[:, :, 2]

# As we are not allowed to use libraries, we apply a Gamma filter to convert our image t
# If we recall the gamma correctionwill let us play with the luminance and contrast of t
# By elevating our array to an specific value and multiplying it by the specified gamma
# A gamma > 1 is called decodin g and with help us to change the color of our image to g

gamma_val = 1.400

# We also need to add specific weights for each RGB channel, this values are the ones be
# We found this values through research as the ideal for this convertion to an optimized

redConst, greenConst, blueConst = 0.2126, 0.7152, 0.0722
grayimg = redConst * redCh ** gamma_val + greenConst * greenCh ** gamma_val + blueConst
```

In [23]:
```
"""
Now that we got our image on a grayscale mode, we need to apply the Sobel Filter to dete
As we can remember for the lectures, a Sobel Operator is a specific type of 2D derivativ
detecting the edges of an image. When using this 2D spatial gradient mesuarement of an i
which correspond to edges.


Sobel operator works by applying the operator that consists of a pair of 3x3 convolution
One kernel is simply the other rotated by 90°. One of the kernels is applied to the X (h
is applied to the Y (Vertical) direction.

As an example we could define the following:

     _               _                 _                 _
    |                 |               |                   |
    | -1.0   0.0  1.0 |               | -1.0  -2.0  -1.0  |
Gx = | -2.0   0.0  2.0 |    and    Gy = |  0.0   0.0   0.0  |
    | -1.0   0.0  1.0 |               |  1.0   2.0   1.0  |
    |_               _|               |_                 _|
"""

# Naturally the first step will be to define our kernels onto 2D arrays
kernX = np.array([[-1.0, 0.0, 1.0], [-2.0, 0.0, 2.0], [-1.0, 0.0, 1.0]])
kernY = np.array([[-1.0, -2.0, -1.0], [0.0, 0.0, 0.0], [1.0, 2.0, 1.0]])

# Then our next step is to obtain the shape of our grayscale image, wich will help us on
# In this numpy array we will be saving the sobel filter image we build.
[r, col] = np.shape(grayimg)
sobelImg = np.zeros(shape=(r, col))

# Next we go though the x and y values of our image
```

```
# then we store the computed value of the sobel transformation on the corresponding posi
for i in range(r - 2):
    for j in range(col - 2):
        # To find the Gradient on each direction, we multiply each of our kernels to the
        # We apply this operation separetly to produce a measure of the gradient compone
        gx = np.sum(np.multiply(kernX, grayimg[i:i + 3, j:j + 3]))
        gy = np.sum(np.multiply(kernY, grayimg[i:i + 3, j:j + 3]))
        # Finally we find the absolute magnitude of the gradient at each point, which is
        sobelImg[i + 1, j + 1] = np.sqrt(gx ** 2 + gy ** 2)

plot_images_edgeDetection(img, grayimg, sobelImg)
```

Laplacian

In [68]:
```
def laplacian_filter(original_image):
    #We first ned to get the originl image dimensions
    w, h = image.size
    #After we define the Laplacian kernel
    kernel = [[0, 1, 0], [1, -4, 1], [0, 1, 0]]
    #Then we proceed to create a new image with the same size and mode as the original i
    new_filtered_image = Image.new("L", (w, h))
    pixels = new_filtered_image.load()

    for y in range( 1, h - 1):
        for x in range( 1, w - 1):
            # We now procced to apply the Laplacian filter using convolution
            convolution = 0
            for i in range( -1, 2):
                for j in range( -1, 2):
                    convolution += kernel[i + 1][j + 1] * image.getpixel((x + j, y + i))
            pixels[x, y] = convolution

    return new_filtered_image

img_path = 'data/tiger.jpeg'
original_image = Image.open(img_path)
image_for_filter = Image.open(img_path).convert('L')
laplacian_image = laplacian_filter(image_for_filter)

plot_images_edgeDetection(original_image, image_for_filter, laplacian_image.convert('P'))
```

Which is more efficient?

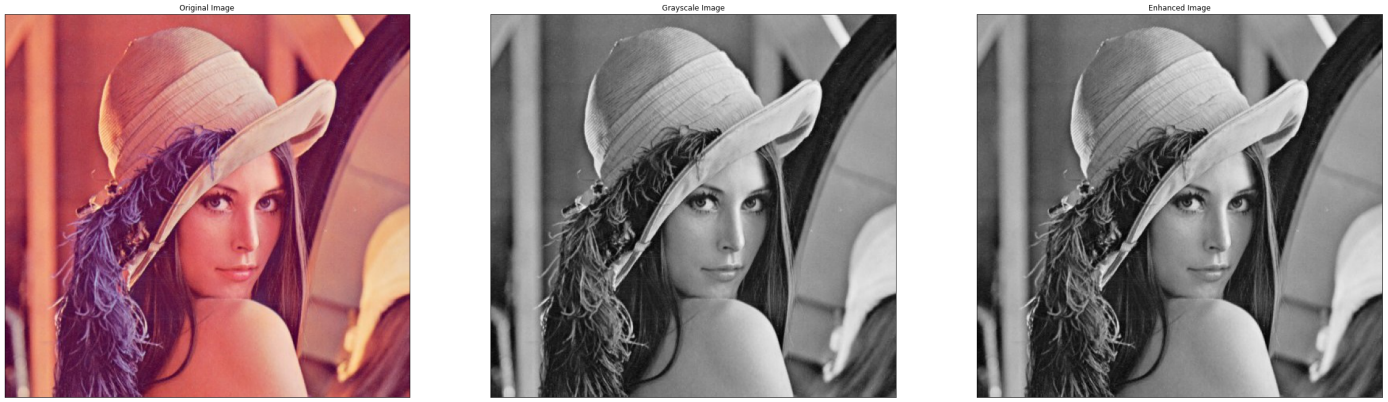
Sobel and Prewitt same complexity, they are the same algorithm with different mask [1], they have the same

time complexity of $O(n)$, n is defined as the number of the pixels in the image. The operation of the convolution has a constant time of $O(1)$ therefore the time complexity is determined by the size of the image, that means, correlated with the number of the pixels in the image. Meanwhile, Laplacian has a similar complexity that Sobel and Prewitt but due that its operator can be applied in a different manners, for example by using different kernels, its complexity will be determined by the used operators and the size of the kernel. However, Laplacian sometimes demonstrates an advantage for edge detections [2]

2. Image enhancement and image masking

In [21]:
```
# Function to plot original and grayscaled Image
def plot_images_enhanced(orig_img, gray_img, enhanced_img):
    orig_img = cv2.cvtColor(orig_img, cv2.COLOR_BGR2RGB  )
    plt.figure(figsize=([40, 40]))
    plt.subplot(131),plt.imshow(orig_img, cmap=plt.get_cmap('gray'))
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(132),plt.imshow(gray_img, cmap=plt.get_cmap('gray'))
    plt.title('Grayscale Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(133),plt.imshow(enhanced_img, cmap=plt.get_cmap('gray'))
    plt.title('Enhanced Image'), plt.xticks([]), plt.yticks([])
```

In [24]:
```
# Convert image from color to grayscale, then applies a median filter and soften image,
# canny to apply the mask with the soften and finally image is sharpened with the alpha
def line_enhancement(img, alpha=3):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kernel = np.ones((7,7),np.float32)/5
    soften = cv2.filter2D(gray,-1,kernel)
    edges = cv2.Canny(soften, 80, 200)
    mask = cv2.subtract(soften, edges)
    enhanced = cv2.addWeighted(gray, alpha, mask, 1-alpha, 0)
    plot_images_enhanced(img, gray, enhanced)

img = cv2.imread("data/lena.jpg")
line_enhancement(img, 1)
```

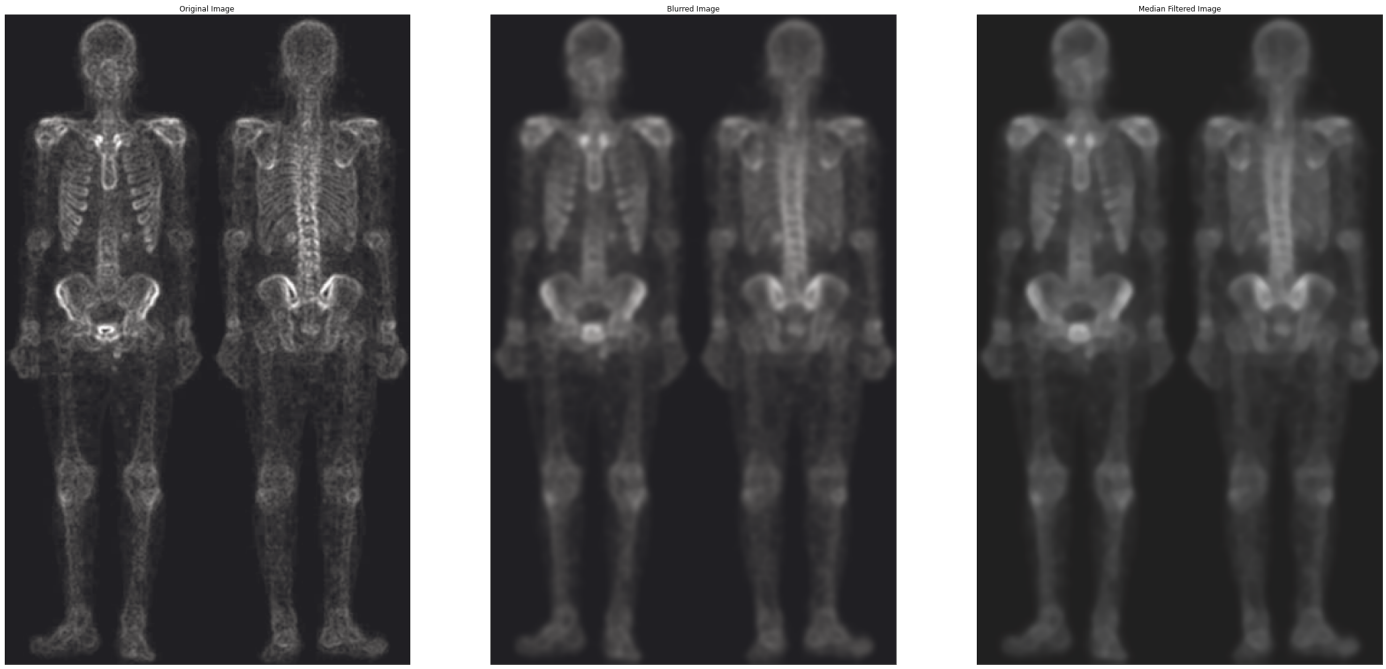


3. Median filtering

In [29]:
```
# Function to plot original, grayscaled and Edge Detected Images
def plot_images_medianFilter(orig_img, blurred_image, median_image):
    plt.figure(figsize=([40, 40]))
    plt.subplot(131),plt.imshow(orig_img)
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(132),plt.imshow(blurred_image, cmap=plt.get_cmap('gray'))
    plt.title('Blurred Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(133),plt.imshow(median_image, cmap=plt.get_cmap('gray'))
    plt.title('Median Filtered Image'), plt.xticks([]), plt.yticks([])
```

In [8]:
```
# Create the gaussian kernel function with size and sigma values
def gaussian_kernel(size, sigma):
    size = int(size) // 2
    x, y = np.mgrid[-size:size+1, -size:size+1]
    normal = 1 / (2.0 * np.pi * sigma**2)
    g = np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
    return g
```

In [33]:
```
# load original image
image = cv2.imread("data/PET.png")

# define kernel and apply gaussian filter to image
kernel = gaussian_kernel(17, 3)
blurred_image = cv2.filter2D(image, -1, kernel)

# apply median filter
median_image = median_filter(blurred_image, 7)

# Plot the images
plot_images_medianFilter(image, blurred_image, median_image)
```



References

[1] Ahmed, A. S. (2018). Comparative study among Sobel, Prewitt and Canny edge detection operators used in image processing. J. Theor. Appl. Inf. Technol, 96(19), 6517-6525.

[2] Asst. lecturer Aadel I. Khalil, I. S. A. A.-K. “COMPARATIVE STUDY FOR EDGE DETECTION OF NOISY IMAGE USING SOBEL AND LAPLACE OPERATORS”. Journal of the College of Education for Women, vol. 23, no. 2, Feb. 2019, https://jcoeduw.uobaghdad.edu.iq/index.php/journal/article/view/853.