Name: Preshit Desai

Year: 3rd Year AIML

Subject: Advance Computer Vision (CA1)

Roll No.: AI4145

Importing the CV2 library & with the help of it inserting the image in it

```
import cv2
img = cv2.imread(r"G:/ACV CA 1/heart png.png")
cv2.imshow('Input image', img)
cv2.waitKey(0)
```

The following image is inserted in the code for image preprocessing. Different preprocessing filters will be applied on this image to get different insights.

```
## Converting the image in the gray by applying gray scale
import cv2
gray_img = cv2.imread(r"G:/ACV CA 1/istockphoto-109721217-612x612.jpg", cv2.IMREAD_GRAYSCALE)
cv2.imshow('Grayscale', gray_img)
cv2.waitKey(0)
```

It reads an image file in grayscale mode from the specified file using the **cv2.imread()** function with the **cv2.IMREAD_GRAYSCALE** flag. This means that the image will be loaded in grayscale, removing color information and resulting in a black and white image.

Output Image:

```
import cv2
img = cv2.imread(r"G:/ACV CA 1/brain-4314636-640-64f94363d838c.jpg", cv2.IMREAD_COLOR)
gray_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2.imshow('Grayscale image', gray_img)
cv2.waitKey(0)
```

It converts the color image img to grayscale using the **cv2.cvtColor()** function with the **cv2.COLOR_RGB2GRAY** conversion code.

Output Image:

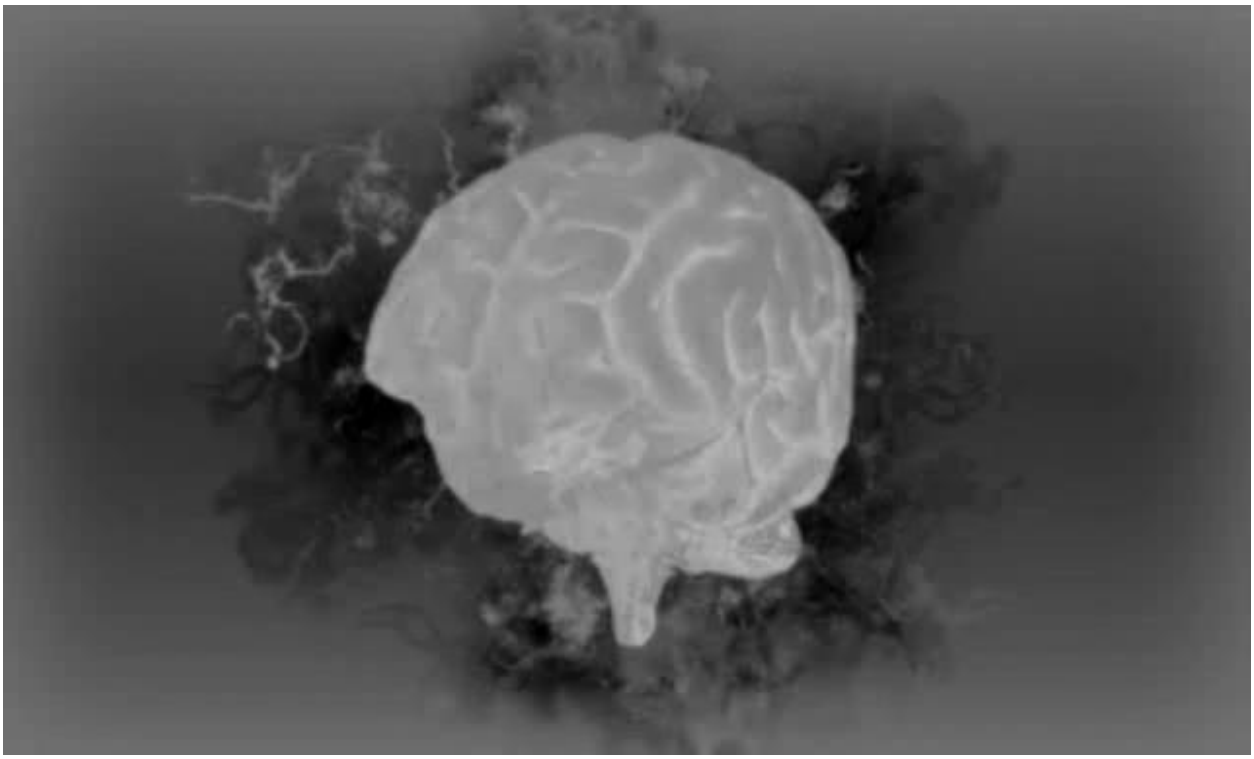
```
yuv_img = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
y,u,v = cv2.split(yuv_img)
cv2.imshow('Y channel', y)
cv2.imshow('U channel', u)
cv2.imshow('V channel', v)
cv2.waitKey(0)
```

- It splits the YUV image into its Y, U, and V channels using **cv2.split()**.
- It displays each of the Y, U, and V channels separately in their respective windows with titles 'Y channel', 'U channel', and 'V channel'.
- It waits for key presses after each set of channel displays, allowing the user to view each channel independently.

```
cv2.imshow('Y channel', yuv_img[:, :, 0])
cv2.imshow('U channel', yuv_img[:, :, 1])
cv2.imshow('V channel', yuv_img[:, :, 2])
cv2.waitKey(0)
```

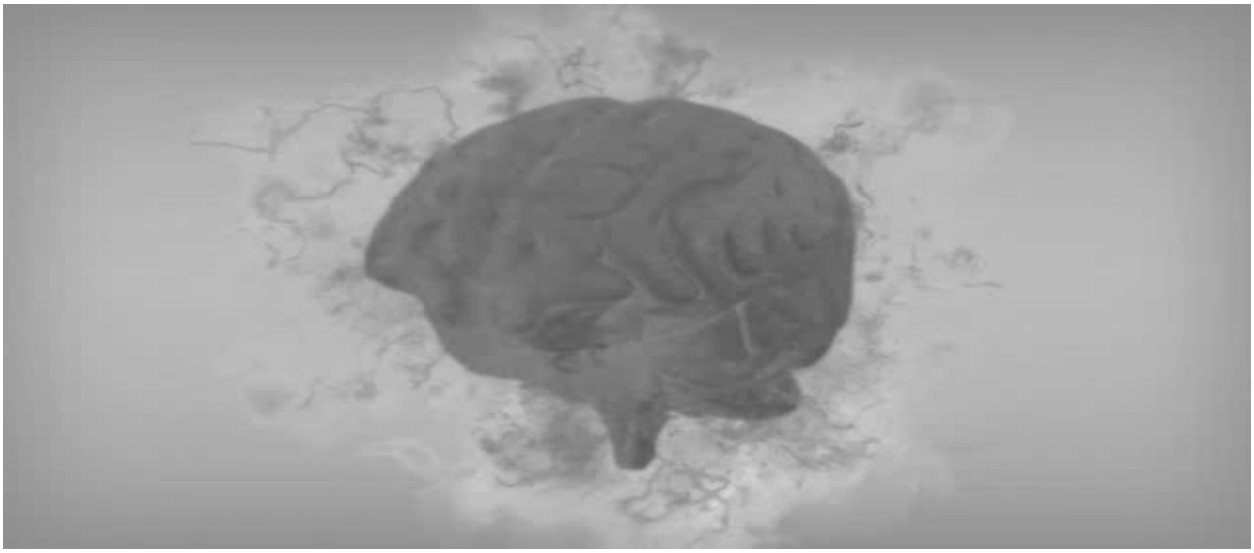
it displays the Y, U, and V channels directly from the YUV image using indexing (**yuv_img[:, :, 0]**, **yuv_img[:, :, 1]**, **yuv_img[:, :, 2]**) and waits for key presses after each display.

Output Image (V):



In the V image contrast of the image is decreases & the image get less sharpen than the original image. The image is less clear than the original image

Output Image (U):



In the V image is less clear than the original image but the pixels of the brighter part is get reduces means image is get more compressed & in the U image overall pixel intensity decreases. Contrast of the image is less.

Output Image (Y):

Y image is very clear image than V & U images.

```
img = cv2.imread(r"G:/ACV CA 1/brain-4314636-640-64f94363d838c.jpg", cv2.IMREAD_COLOR)
g,b,r = cv2.split(img)
gbr_img = cv2.merge((g,b,r))
rbr_img = cv2.merge((r,b,r))
bbr_img = cv2.merge((b,b,r))
cv2.imshow('Original', img)
cv2.imshow('GRB', gbr_img)
cv2.imshow('RBR', rbr_img)
cv2.imshow('BBR', bbr_img)
cv2.waitKey(0)
```

It creates three new images by merging the color channels in different orders:

- **gbr_img**: Merging the green channel as the first channel, blue as the second channel, and red as the third channel.

- **rbr_img**: Merging the red channel as the first channel, blue as the second channel, and red as the third channel (essentially swapping the green and red channels compared to the original image).
- **bbr_img**: Merging the blue channel as the first channel, blue as the second channel, and red as the third channel (which accentuates the blue component while using red for the third channel).

Output Images (GBR):

Output Image (RBR):

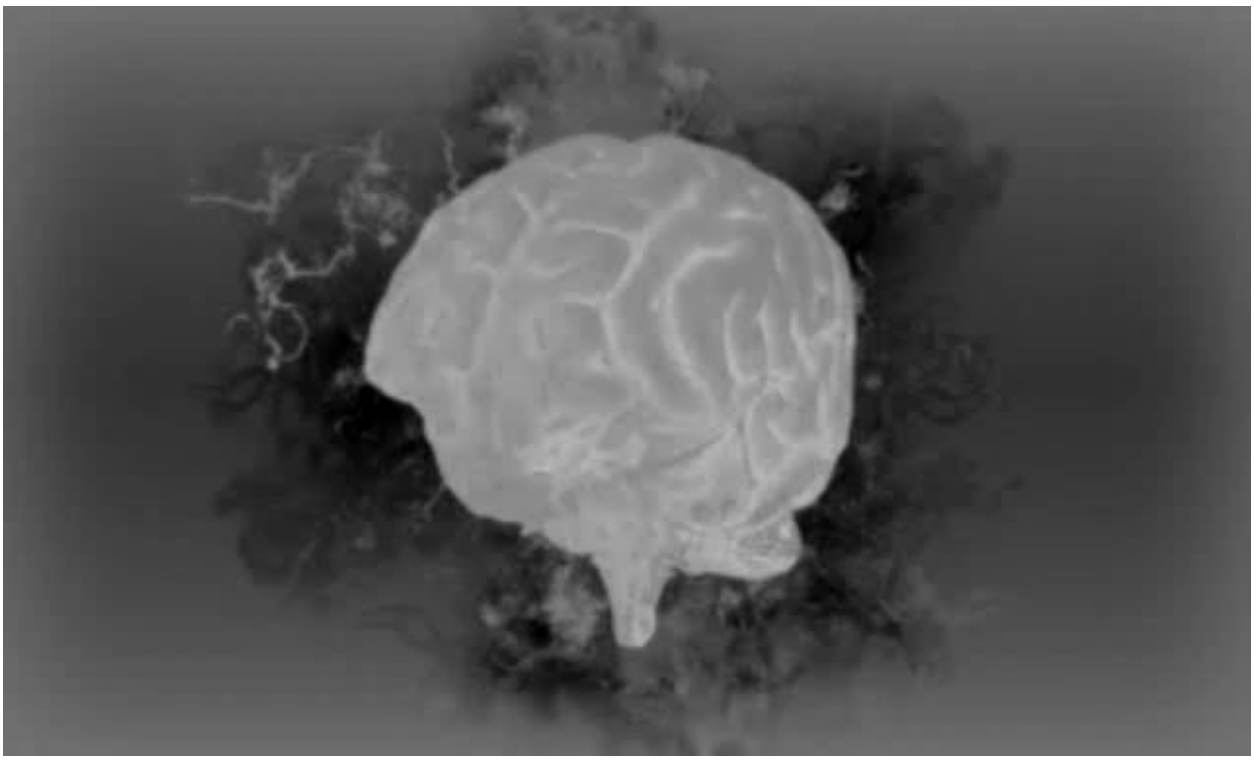
Output Image (BBR):

```
#Image translation
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/anatomy-7077549_1280.png")
num_rows, num_cols = img.shape[:2]
translation_matrix = np.float32([ [1,0,70], [0,1,110] ])
img_translation = cv2.warpAffine(img, translation_matrix, (num_cols, num_rows), cv2.INTER_LINEAR) #
cv2.imshow('Translation', img_translation)
cv2.waitKey(0)
```

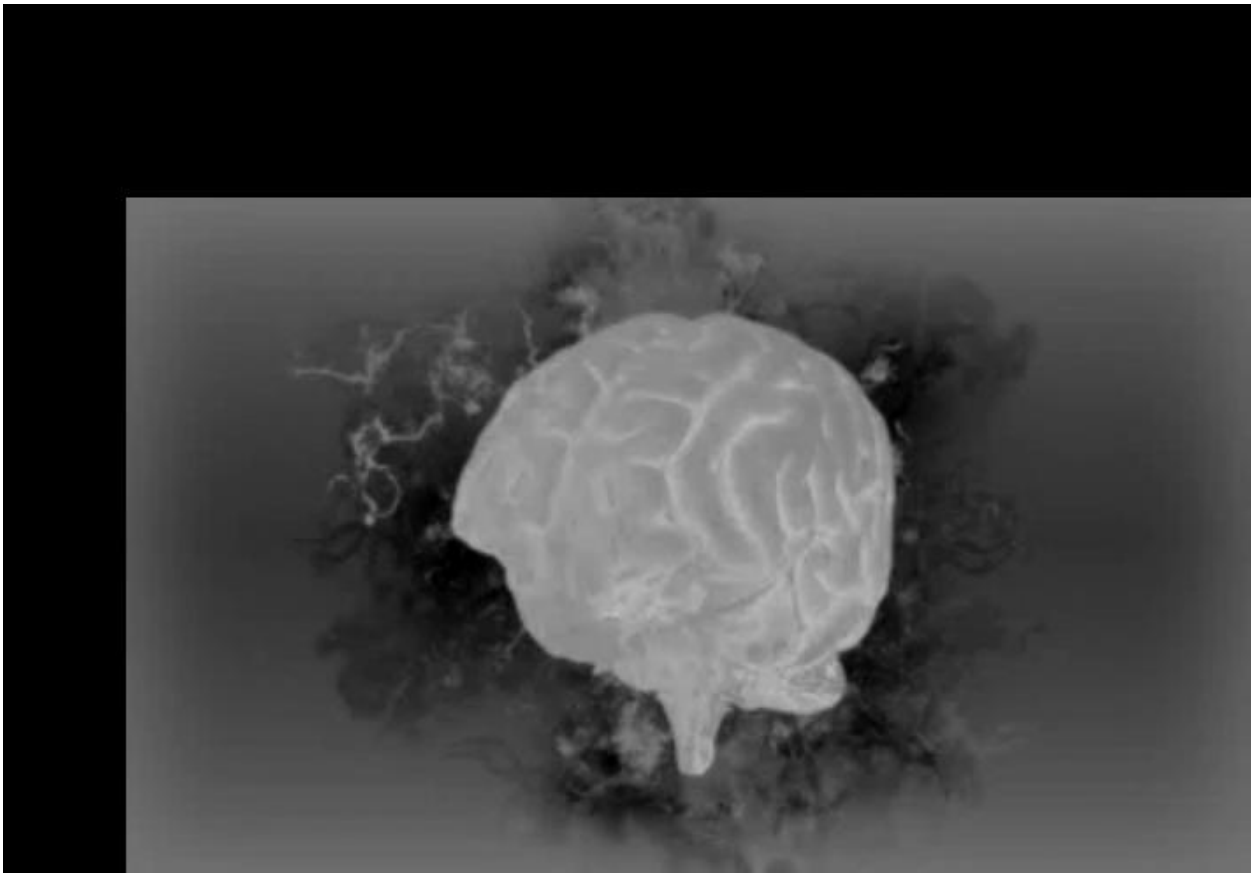
It defines a translation matrix (**translation_matrix**) using NumPy. This matrix specifies the translation operation, shifting the image 70 pixels to the right (in the X-axis) and 110 pixels down (in the Y-axis).

It performs the image translation using **cv2.warpAffine()**.

Original Image:



Output Translated Image:



```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/V channel brain.png")
num_rows, num_cols = img.shape[:2]
translation_matrix = np.float32([ [1,0,70], [0,1,110] ])
img_translation = cv2.warpAffine(img, translation_matrix, (num_cols, num_rows), cv2.INTER_LINEAR, cv2.
cv2.imshow('Translation', img_translation)
cv2.waitKey(0)
```

Changing the size & dimensions of the Translational matrix. So, the output image will having few changes in the dimensions

Output Image:

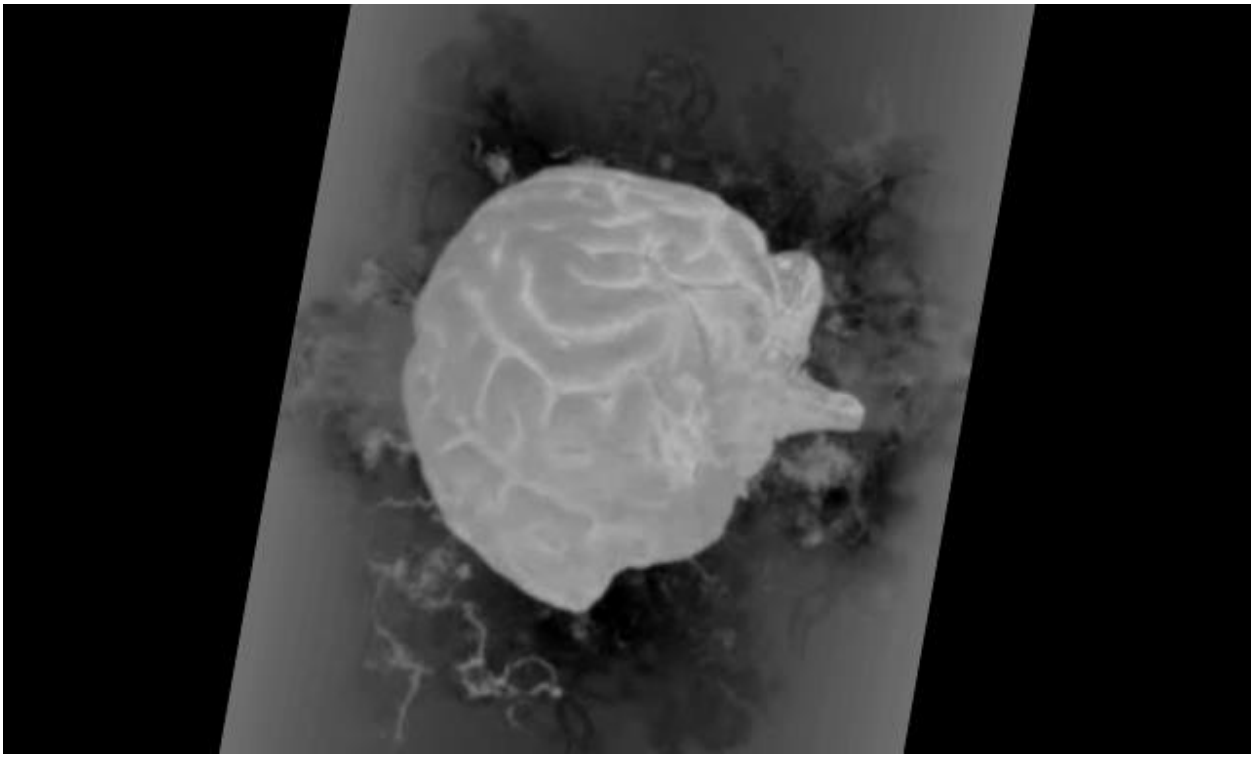


Image get rotated format.

```
import cv2
img = cv2.imread(r"G:/ACV CA 1/Rotation img.png")
img_scaled = cv2.resize(img,None,fx=1.2, fy=1.2, interpolation =cv2.INTER_LINEAR)
cv2.imshow('Scaling - Linear Interpolation', img_scaled)
img_scaled = cv2.resize(img,None,fx=1.2, fy=1.2, interpolation = cv2.INTER_CUBIC)
cv2.imshow('Scaling - Cubic Interpolation', img_scaled)
img_scaled = cv2.resize(img,(450, 400), interpolation = cv2.INTER_AREA)
cv2.imshow('Scaling - Skewed Size', img_scaled)
cv2.waitKey(0)
```

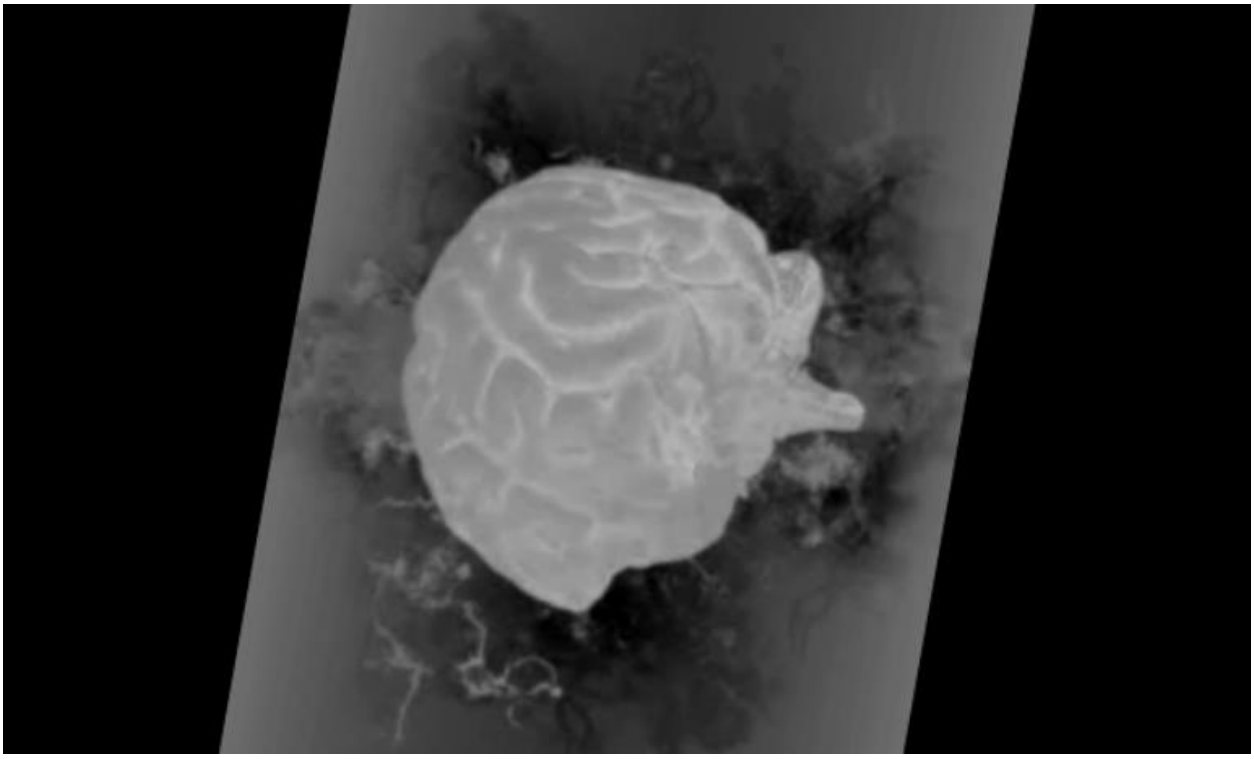
Above code describe the following method:

1. **Linear Interpolation**:
    - Linear interpolation is an interpolation method used in image processing to estimate pixel values between two known pixel values.
    - It calculates intermediate pixel values as a weighted average of the nearest neighboring pixels.

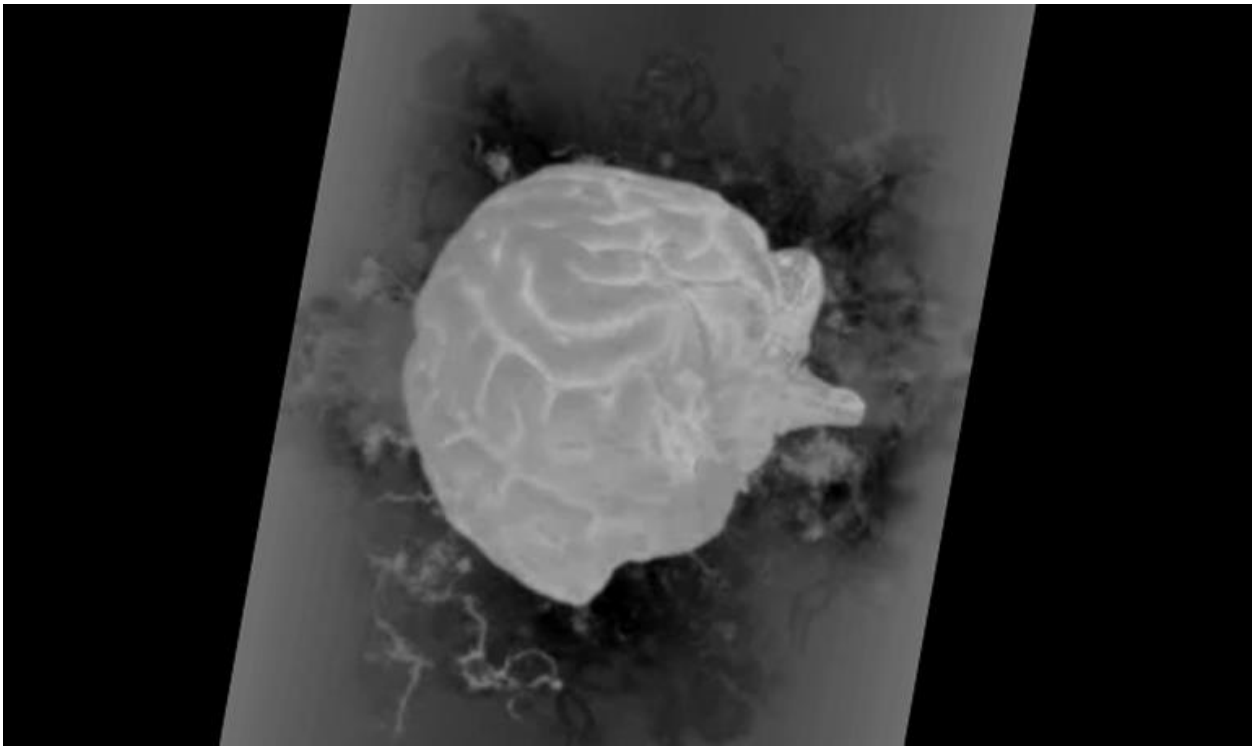
- It results in smoother transitions between pixels but may not handle complex patterns or details as effectively.

2. **Cubic Interpolation**:
    - Cubic interpolation is a more advanced interpolation technique that uses cubic polynomials to estimate pixel values.
    - It provides a smoother and more accurate representation of pixel values between known points.
    - It's especially useful for resizing images while preserving fine details and minimizing artifacts.
3. **Skewed Size (Area-Based) Interpolation**:
    - Skewed size, or area-based interpolation, is a resizing method where an image is scaled to a specific size, which might not maintain the original aspect ratio.
    - It works by averaging pixel values in the source image to compute the values in the resized image.
    - While it can be used for resizing images to non-proportional dimensions, it may result in a loss of image quality, especially when scaling down significantly.

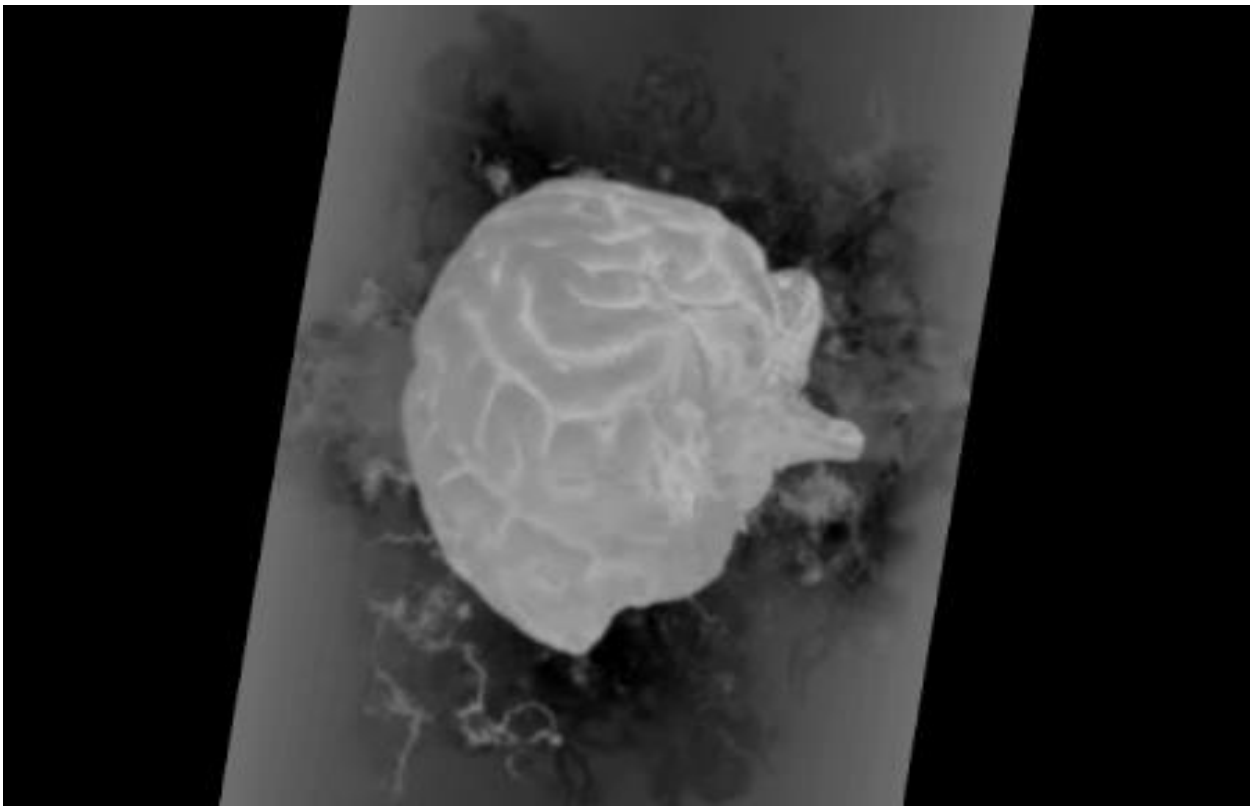
Output Image (Linear interpolation):



Output Image (Cubic Interpolation):



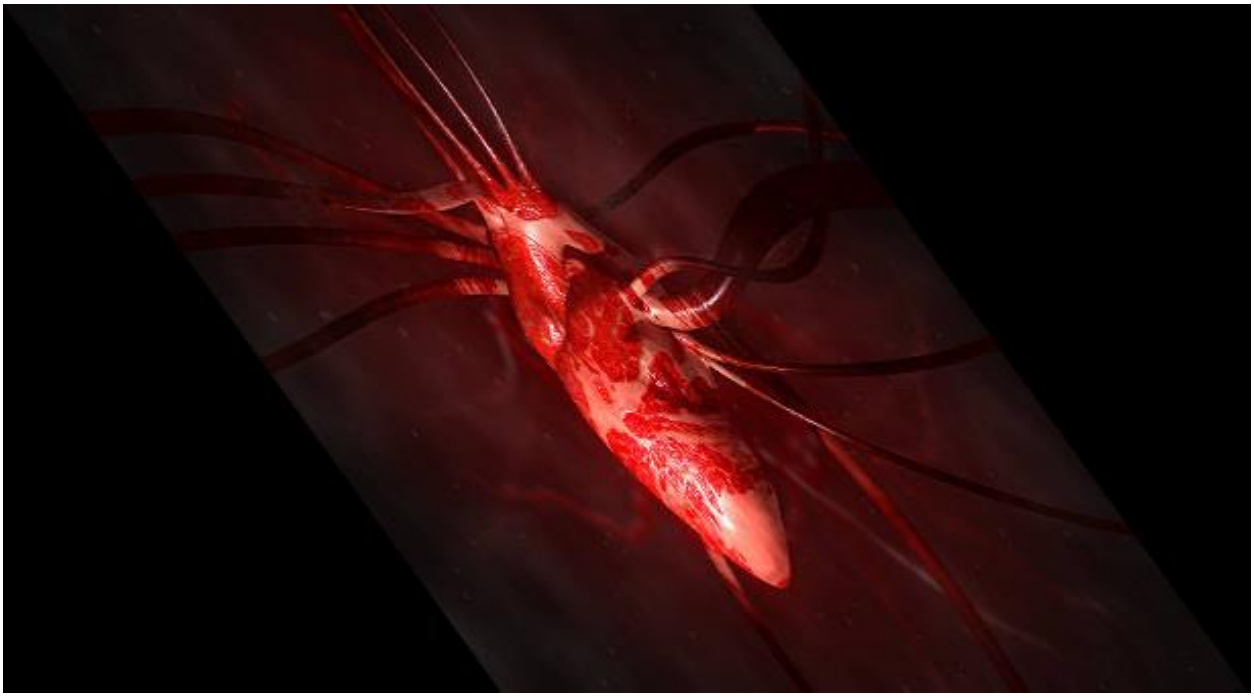
Output Image (Skewed):



```
#Affine Transformations Euclidean Transformation
import cv2
import numpy as np
img = cv2.imread('G:/ACV CA 1/heart png.png')
rows, cols = img.shape[:2]
src_points = np.float32([[0,0], [cols-1,0], [0,rows-1]])
dst_points = np.float32([[0,0], [int(0.6*(cols-1)),0],[int(0.4*(cols-1)),rows-1]])
affine_matrix = cv2.getAffineTransform(src_points, dst_points)
img_output = cv2.warpAffine(img, affine_matrix, (cols,rows))
cv2.imshow('Input', img)
cv2.imshow('Output', img_output)
cv2.waitKey(0)
```

Applying Euclidean Transformation

Output Image (Euclidean Transformation)



```
# Projective transformation
import cv2
import numpy as np
img = cv2.imread('G:/ACV CA 1/heart png.png')
rows, cols = img.shape[:2]
src_points = np.float32([[0,0], [cols-1,0], [0,rows-1], [cols-1,rows-1]])
dst_points = np.float32([[0,0], [cols-1,0], [int(0.33*cols),rows-1],[int(0.66*cols),rows-1]])
projective_matrix = cv2.getPerspectiveTransform(src_points, dst_points)
img_output = cv2.warpPerspective(img, projective_matrix, (cols,rows))
cv2.imshow('Input', img)
cv2.imshow('Output', img_output)
cv2.waitKey(0)
```

Applying the Projective Transformation.

Output Image (Projective Transformation):

```
import cv2
import numpy as np
img = cv2.imread('G:/ACV CA 1/heart png.png')
rows, cols = img.shape[:2]
src_points = np.float32([[0,0], [0,rows-1], [cols/2,0],[cols/2,rows-1]])
dst_points = np.float32([[0,100], [0,rows-101],[cols/2,0],[cols/2,rows-1]])
projective_matrix = cv2.getPerspectiveTransform(src_points, dst_points)
img_output = cv2.warpPerspective(img, projective_matrix, (cols,rows))
cv2.imshow('Input', img)
cv2.imshow('Output', img_output)
cv2.waitKey(0)
wait = True
```

Applying the Perspective Transformation.

Output Image (Perspective Transform):

```
import cv2
import numpy as np
import math
img = cv2.imread('G:/ACV CA 1/BBR brain.png', cv2.IMREAD_GRAYSCALE)
rows, cols = img.shape
img_output = np.zeros(img.shape, dtype=img.dtype)
for i in range(rows):
    for j in range(cols):
        offset_x = int(25.0 * math.sin(2 * 3.14 * i / 180))
        offset_y = 0
        if j+offset_x < rows:
            img_output[i,j] = img[i,(j+offset_x)%cols]
        else:
            img_output[i,j] = 0
cv2.imshow('Input', img)
cv2.imshow('Vertical wave', img_output)
cv2.waitKey(0)
```

Gives the Vertical Wave Type of Image.

Output Image (Vertical Wave):

```
#Blurring is called as low pass filter. Why?
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/BBR brain.png")
kernel_identity = np.array([[0,0,0], [0,1,0], [0,0,0]])
kernel_3x3 = np.ones((3,3), np.float32) / 9.0 # Divide by 9 to normalize the kernel
kernel_5x5 = np.ones((5,5), np.float32) / 25.0 # Divide by 25 to normalize the kernel
cv2.imshow('Original', img)
output = cv2.filter2D(img, -1, kernel_identity)
cv2.imshow('Identity filter', output)
output = cv2.filter2D(img, -1, kernel_3x3)
cv2.imshow('3x3 filter', output)
output = cv2.filter2D(img, -1, kernel_5x5)
cv2.imshow('5x5 filter', output)
cv2.waitKey(0)
output = cv2.blur(img, (3,3))
cv2.imshow('blur', output)
cv2.waitKey(0)
```

In this code different filters are applied to the image. The filters are:

- Identify Filter
- 3x3 Filter
- 5x5 Filter

These are different blurring filters.

Output image (Identify Filter):

Output Image (3x3 Filter):

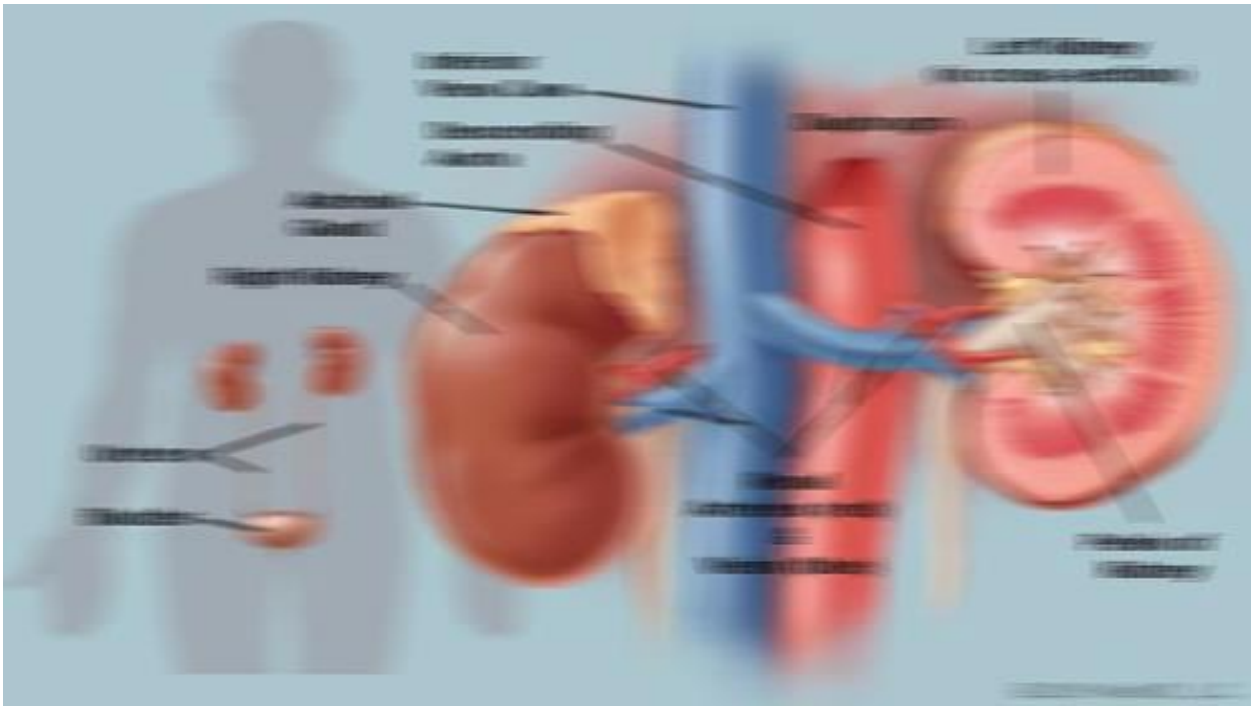
Output Image (5x5 Filter):

In the above filter Identify Filter have more clear image & the 5x5 filter have the blur image means as the filter size increases the image will more get blurred.

```
import cv2
import numpy as np
img = cv2.imread('G:/ACV CA 1/kidneys-anatomy.jpeg')
cv2.imshow('Original', img)
size = 15
# generating the kernel
kernel_motion_blur = np.zeros((size, size))
kernel_motion_blur[int((size-1)/2), :] = np.ones(size)
kernel_motion_blur = kernel_motion_blur / size
# applying the kernel to the input image
output = cv2.filter2D(img, -1, kernel_motion_blur)
cv2.imshow('Motion Blur', output)
cv2.waitKey(0)
```

Giving the motion blur image

Output Image (Motion Blur):



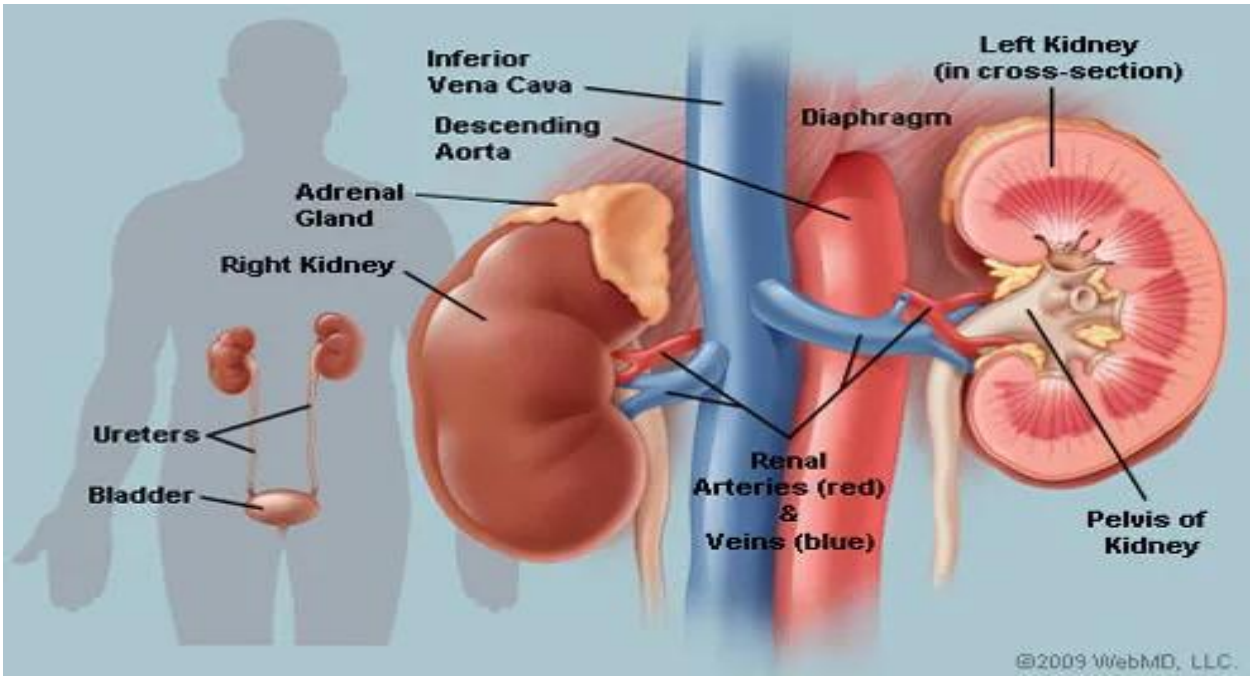
```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/kidneys-anatomy.jpeg")
cv2.imshow('Original', img)
# generating the kernels
kernel_sharpen_1 = np.array([[-1,-1,-1], [-1,9,-1], [-1,-1,-1]])
kernel_sharpen_2 = np.array([[1,1,1], [1,-7,1], [1,1,1]])
kernel_sharpen_3 = np.array([[-1,-1,-1,-1,-1],[-1,2,2,2,-1],[-1,2,8,2,-1],[-1,2,2,2,-1],[-1,-1,-1,-1,-
# applying different kernels to the input image
output_1 = cv2.filter2D(img, -1, kernel_sharpen_1)
output_2 = cv2.filter2D(img, -1, kernel_sharpen_2)
output_3 = cv2.filter2D(img, -1, kernel_sharpen_3)
cv2.imshow('Sharpening', output_1)
cv2.imshow('Excessive Sharpening', output_2)
cv2.imshow('Edge Enhancement', output_3)
cv2.waitKey(0)
```

Above code is for the purpose of sharpening the images.

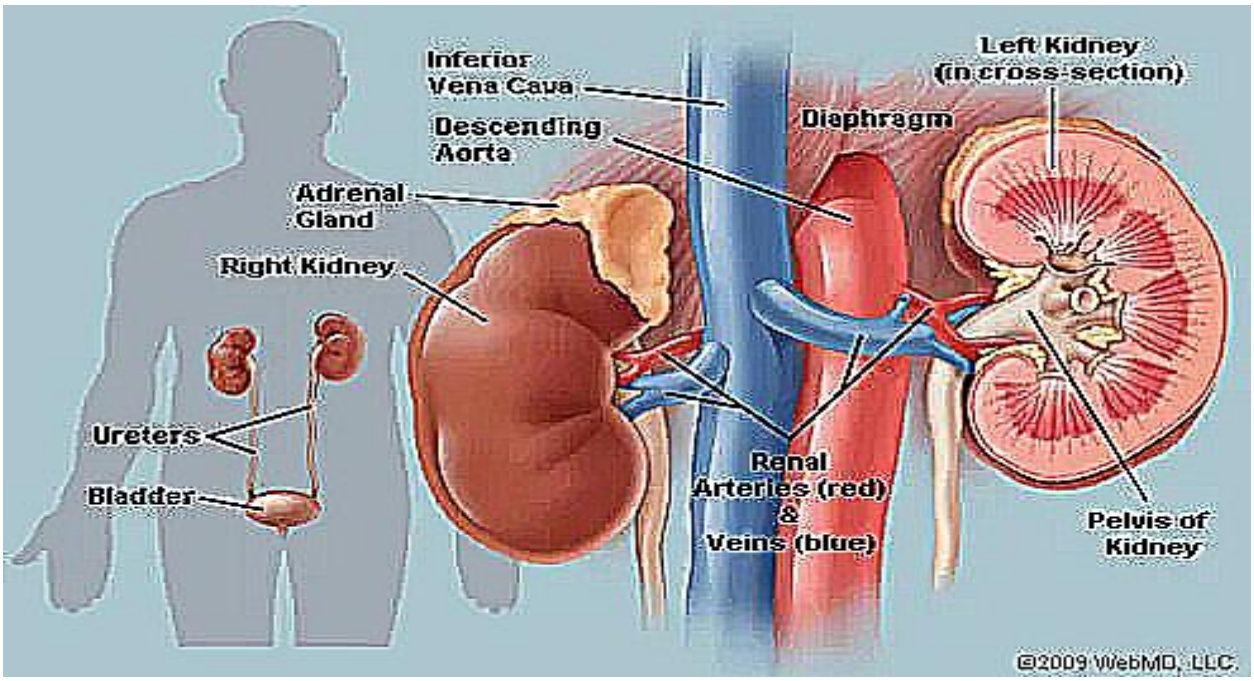
Here 3 types of Image sharpening's are performed:

- Sharpening
- Excessive Sharpening
- Edge Enhancement

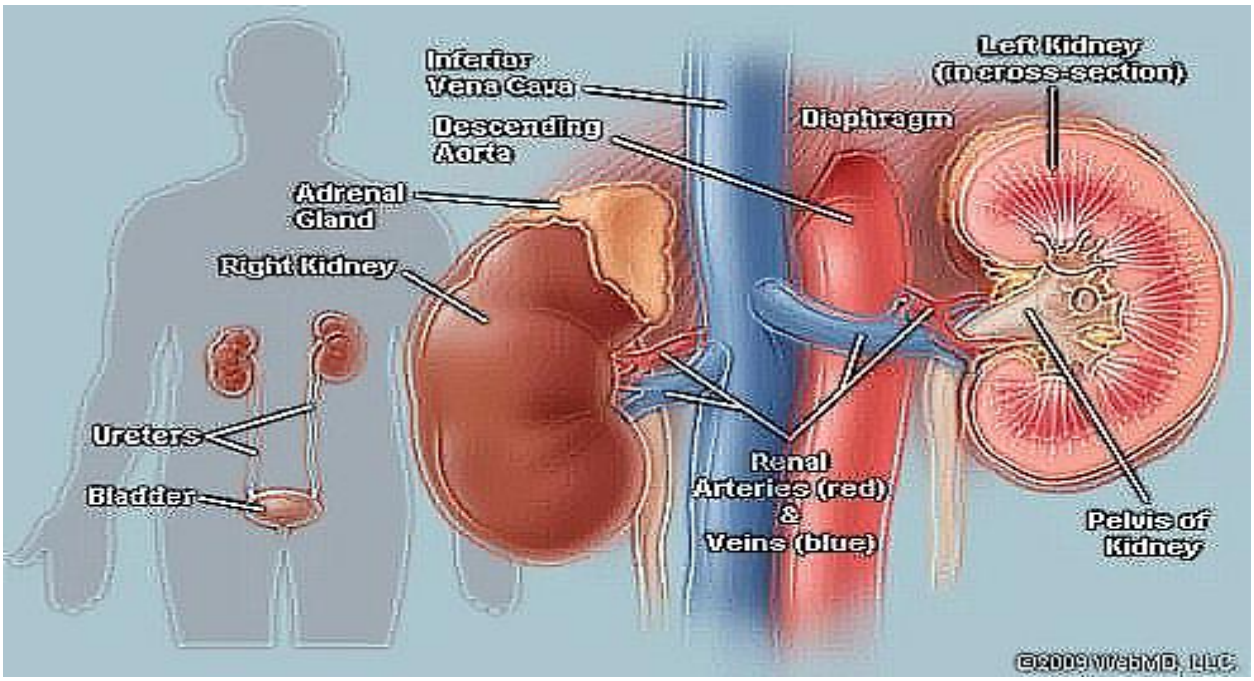
Original Image:



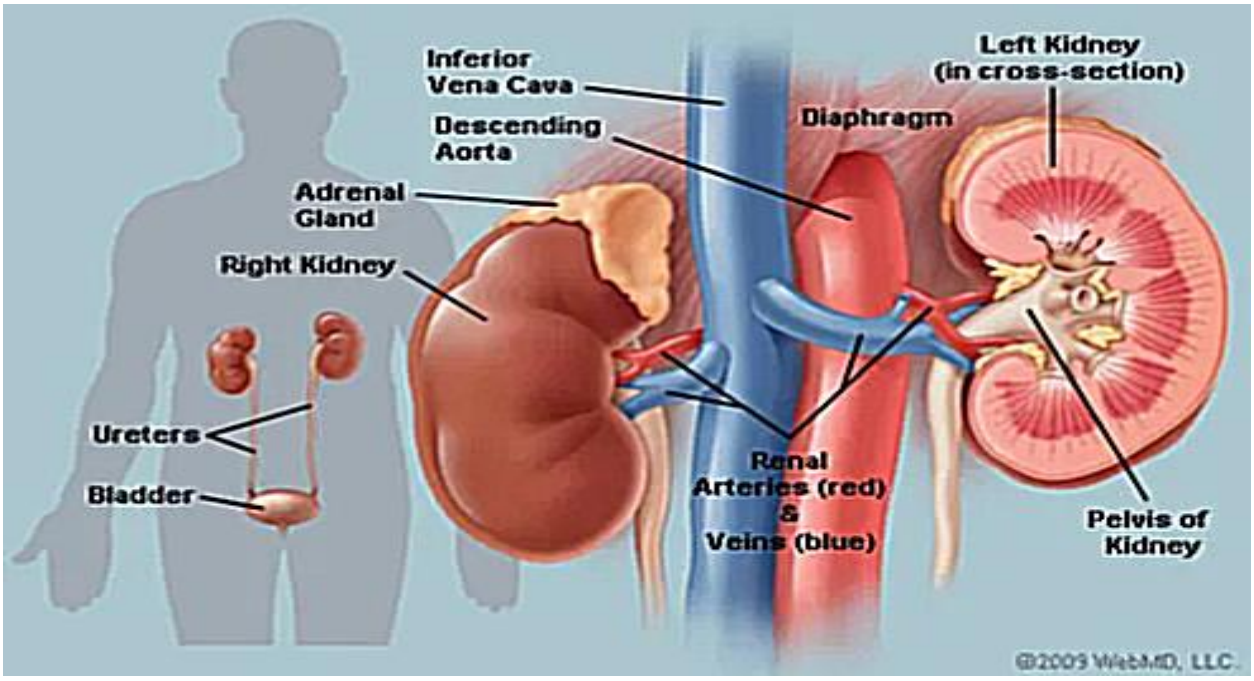
Output Image(Sharpening):



Output Image(Excessive Sharpening):



Output Image (Edge Enhancement):

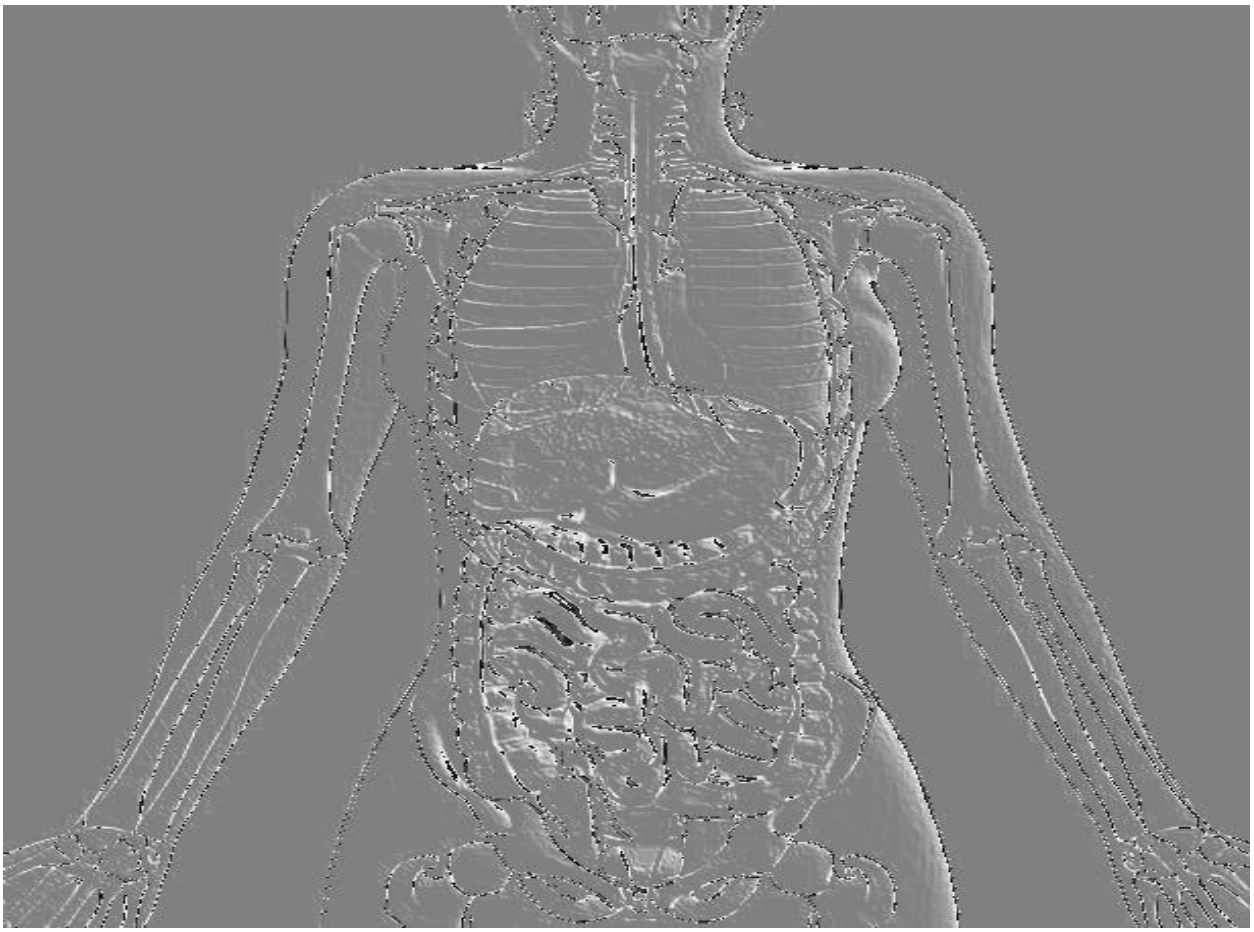



```python
import cv2
import numpy as np
img_emboss_input = cv2.imread(r"G:/ACV CA 1/human-digestive-system-artwork.jpeg")
kernel_emboss_1 = np.array([[0,-1,-1],[1,0,-1],[1,1,0]])
kernel_emboss_2 = np.array([[-1,-1,0],[-1,0,1],[0,1,1]])
kernel_emboss_3 = np.array([[1,0,0],[0,0,0],[0,0,-1]])
# converting the image to grayscale
gray_img = cv2.cvtColor(img_emboss_input,cv2.COLOR_BGR2GRAY)
# applying the kernels to the grayscale image and adding the offset toproduce the shadow
output_1 = cv2.filter2D(gray_img, -1, kernel_emboss_1) + 128
output_2 = cv2.filter2D(gray_img, -1, kernel_emboss_2) + 128
output_3 = cv2.filter2D(gray_img, -1, kernel_emboss_3) + 128
cv2.imshow('Input', img_emboss_input)
cv2.imshow('Embossing - South West', output_1)
cv2.imshow('Embossing - South East', output_2)
cv2.imshow('Embossing - North West', output_3)
cv2.waitKey(0)
```

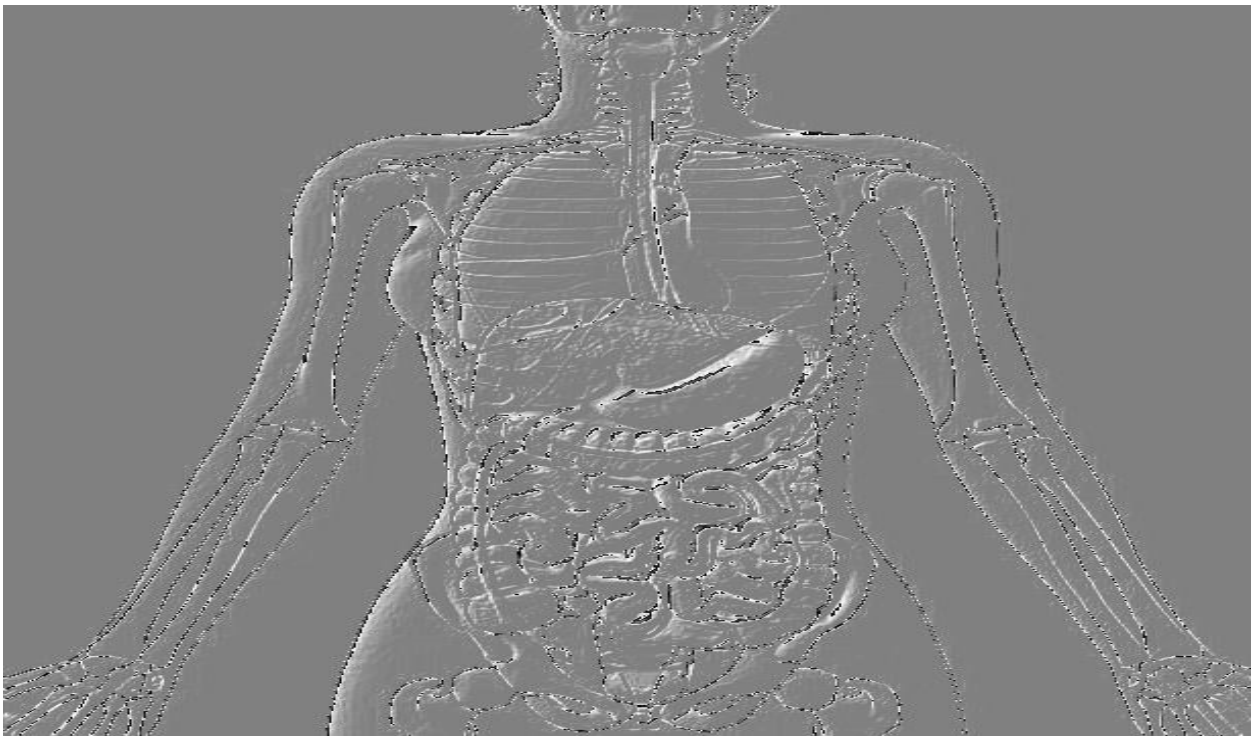
It defines three different embossing kernels (**kernel_emboss_1**, **kernel_emboss_2**, and **kernel_emboss_3**) as NumPy arrays. These kernels are used to create different embossing effects in the image.

1. **Embossing - South West**:
    - This refers to an embossing effect created by applying a convolution kernel where the most significant weights are in the bottom-left (southwest) portion of the kernel.
    - When this kernel is convolved with an image, it creates an effect where the image appears to be raised or embossed in the southwest direction. It highlights edges and details that are sloping downward to the southwest.
2. **Embossing - South East**:
    - This refers to an embossing effect created by applying a convolution kernel where the most significant weights are in the bottom-right (southeast) portion of the kernel.
    - When this kernel is convolved with an image, it creates an effect where the image appears to be raised or embossed in the southeast direction. It highlights edges and details that are sloping downward to the southeast.
3. **Embossing - North West**:
    - This refers to an embossing effect created by applying a convolution kernel where the most significant weights are in the top-left (northwest) portion of the kernel.
    - When this kernel is convolved with an image, it creates an effect where the image appears to be raised or embossed in the northwest direction. It highlights edges and details that are sloping upward to the northwest.

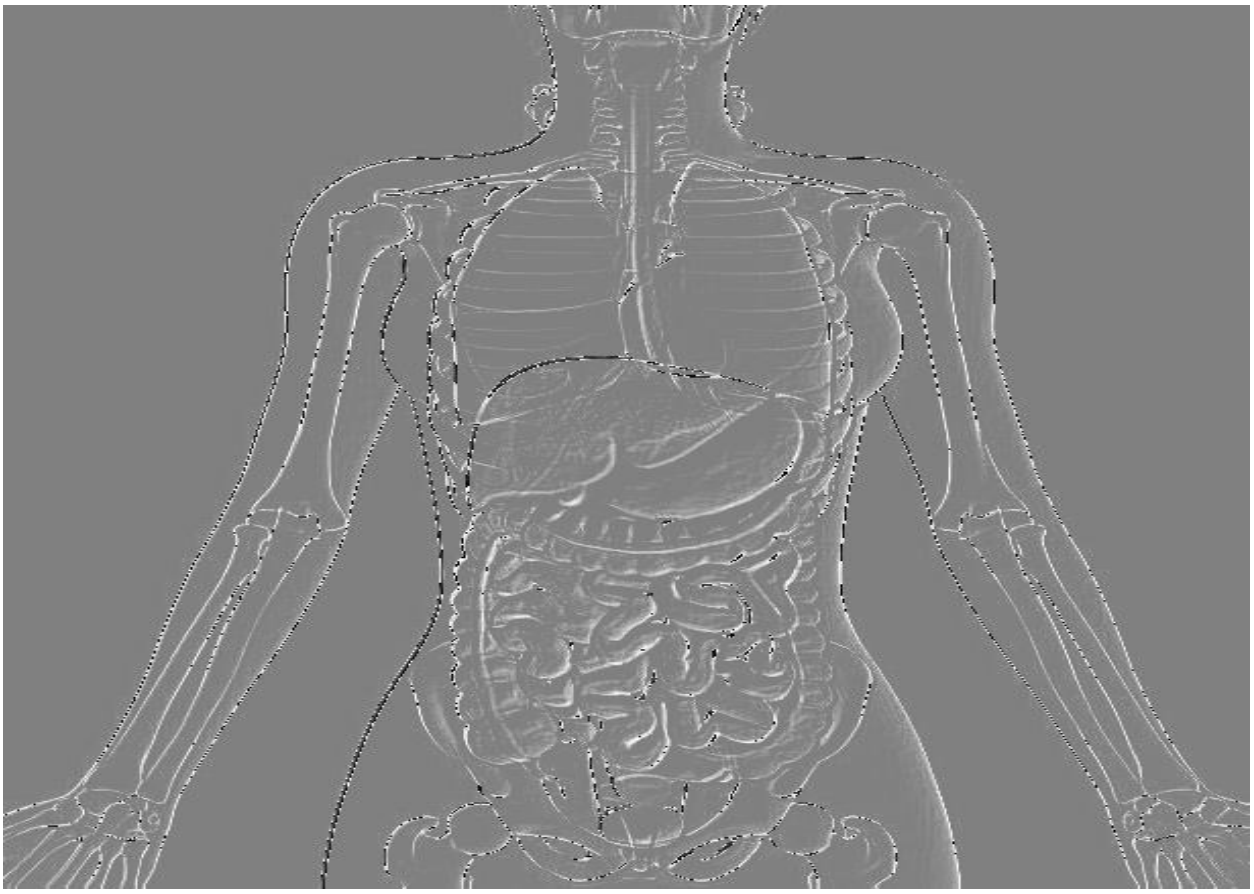
Output Image (Embossing - South West):



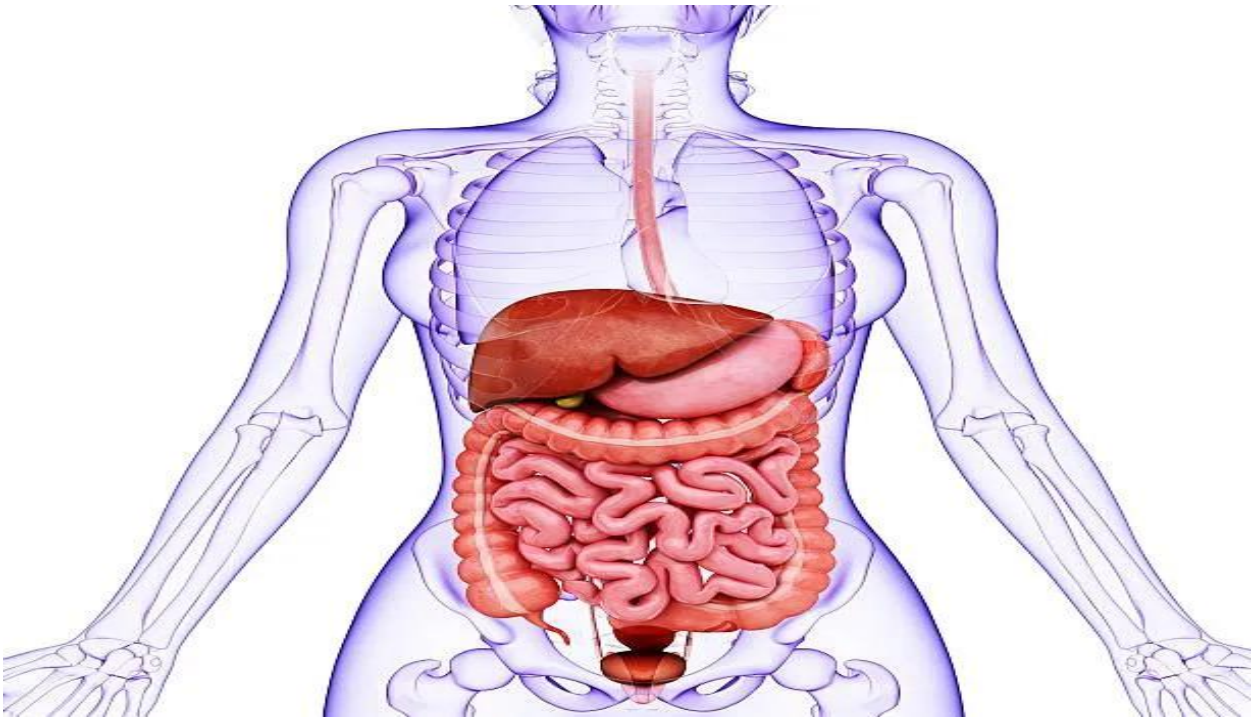
Output Image (Embossing - South East)



Output Image (Embossing – North West)



Original Image:



```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/heart png.png", cv2.IMREAD_GRAYSCALE)
rows, cols = img.shape
# It is used to indicate depth of cv2.CV_64F.
sobel_horizontal = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
# Kernel size can be: 1,3,5 or 7.
sobel_vertical = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)
cv2.imshow('Original', img)
cv2.imshow('Sobel horizontal', sobel_horizontal)
cv2.imshow('Sobel vertical', sobel_vertical)
cv2.waitKey(0)
```

In the above code the sobel filter is used.

Sobel horizontal & sobel vertical are the inversely proportional. Horizontal filter as a Vertical & Vertical filter as Horizontal filter.

Original Filter:

Output Image (Sobel Vertical):

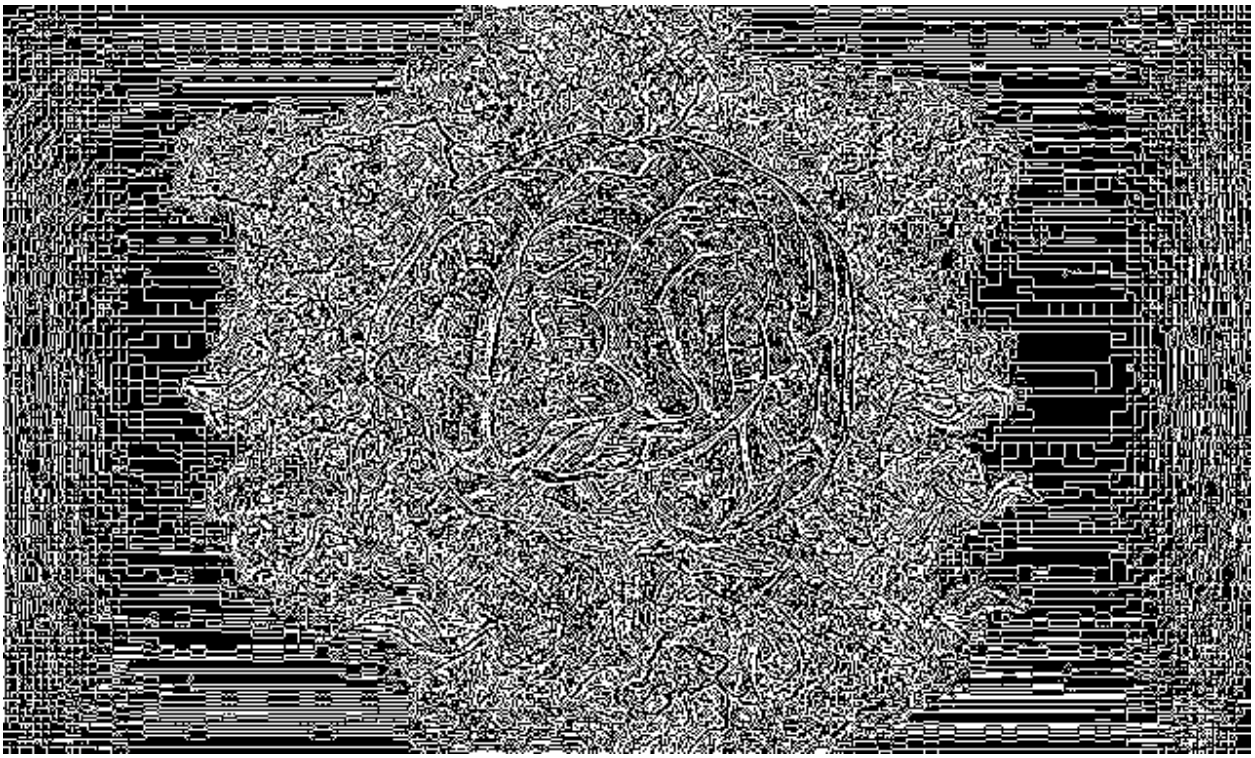
Output Image (Sobel Horizontal):

```python
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/GRB original img.png", cv2.IMREAD_GRAYSCALE)
rows, cols = img.shape
# It is used depth of cv2.CV_64F.
laplacian = cv2.Laplacian(img, cv2.CV_64F)
cv2.imshow('Original', img)
cv2.imshow('Laplacian', laplacian)
cv2.waitKey(0)
```

It applies the Laplacian edge detection filter to the grayscale image using **cv2.Laplacian().** The result is stored in the laplacian variable. The **cv2.CV_64F** depth argument specifies the desired output data type

Used for the Noise Reduction.

Output Image (Laplacian Filter):

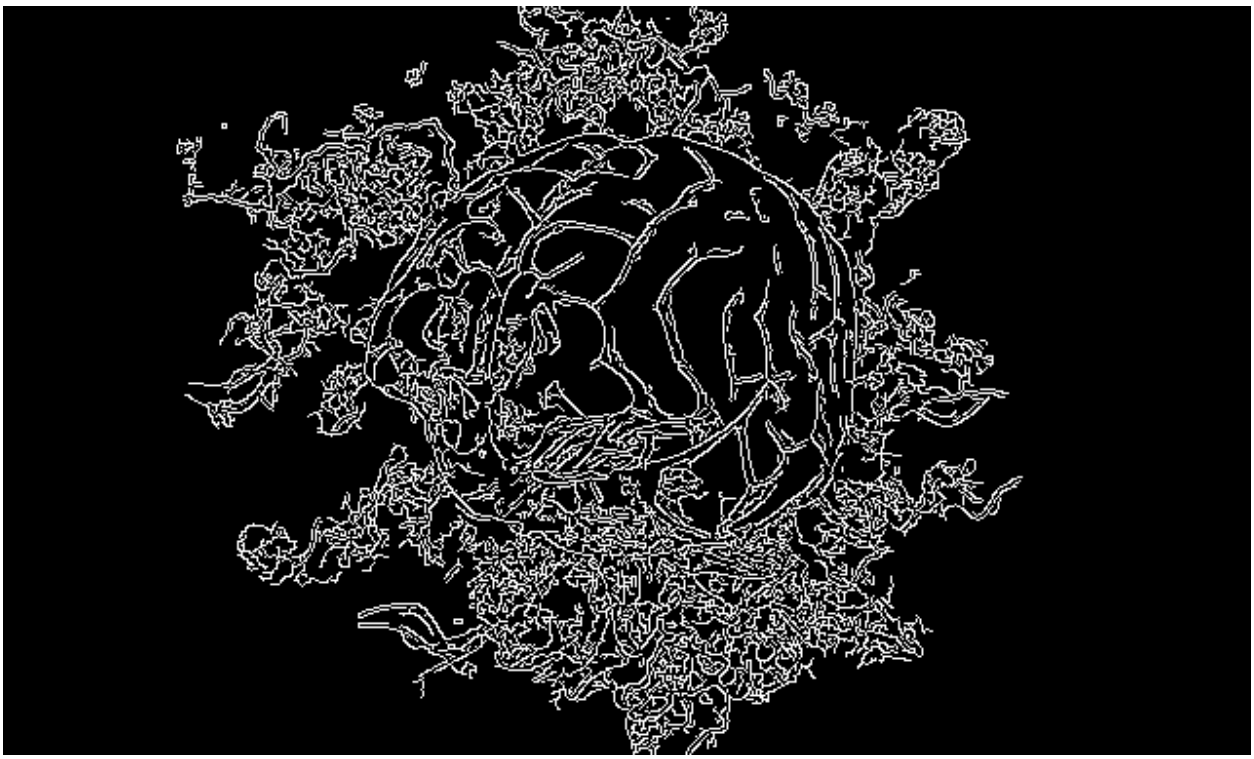


```python
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/GRB original img.png", cv2.IMREAD_GRAYSCALE)
rows, cols = img.shape
canny = cv2.Canny(img, 50, 240)
cv2.imshow('Canny', canny)
cv2.waitKey(0)
```

It applies the Canny edge detection algorithm to the grayscale image using **cv2.Canny().** The algorithm takes three arguments: the input image (img), a lower threshold (50 in this case), and an upper threshold (240 in this case).

It helps identify and highlight edges or boundaries within an image by detecting areas of rapid intensity change. This filter is useful for various computer vision and image analysis tasks, including object recognition, image segmentation, and feature extraction.

Output Image (Canny Filter):



```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/GRB original img.png", 0)
kernel = np.ones((5,5), np.uint8)
img_erosion = cv2.erode(img, kernel, iterations=1)
img_dilation = cv2.dilate(img, kernel, iterations=1)
cv2.imshow('Input', img)
cv2.imshow('Erosion', img_erosion)
cv2.imshow('Dilation', img_dilation)
cv2.waitKey(0)
```

It performs image erosion on the grayscale image using **cv2.erode()**. Erosion is a morphological operation that shrinks or erodes the boundaries of foreground objects in the image. The result is stored in the **img_erosion** variable.

It performs image dilation on the grayscale image using **cv2.dilate()**. Dilation is a morphological operation that expands or dilates the boundaries of foreground objects in the image. The result is stored in the **img_dilation** variable.

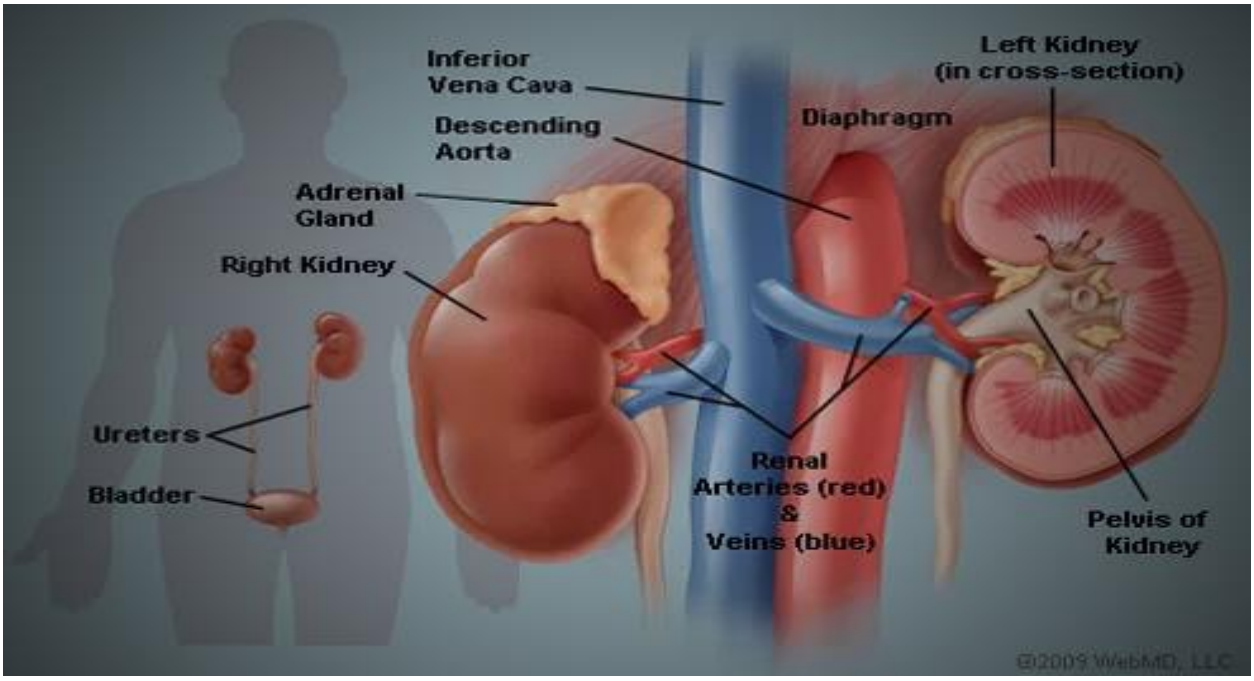
Output Image (Erosion):

Output Image (Dilution):

```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/kidneys-anatomy.jpeg")
rows, cols = img.shape[:2]
# generating vignette mask using Gaussian kernels
kernel_x = cv2.getGaussianKernel(cols,200)
kernel_y = cv2.getGaussianKernel(rows,200)
kernel = kernel_y * kernel_x.T
mask = 255 * kernel / np.linalg.norm(kernel)
output = np.copy(img)
# applying the mask to each channel in the input image
for i in range(3):
    output[:,:,i] = output[:,:,i] * mask
cv2.imshow('Original', img)
cv2.imshow('Vignette', output)
cv2.waitKey(0)
```

- The Gaussian kernel is a two-dimensional matrix with a bell-shaped distribution of values.
- It is used for smoothing or blurring images and reducing noise while preserving edges and important details.

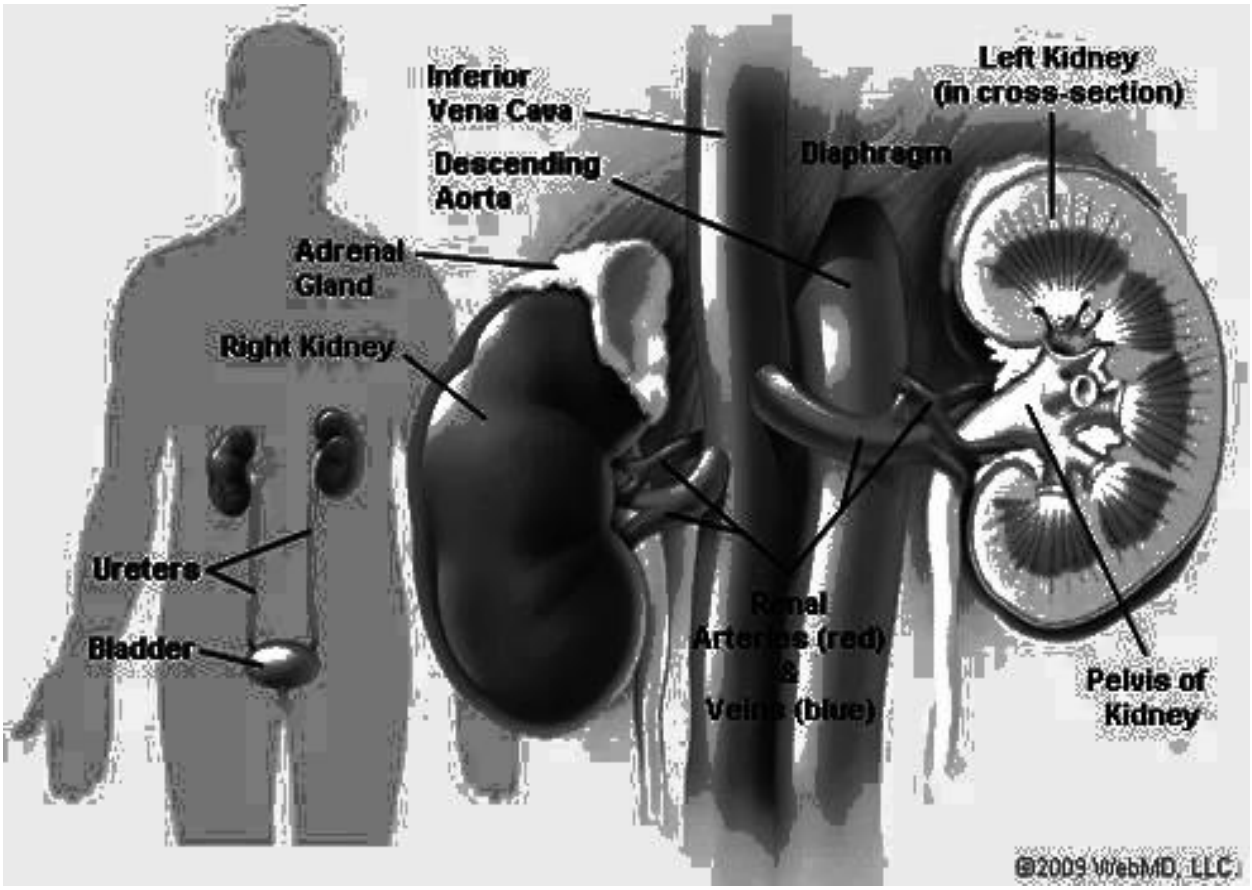
Output Image (Gaussian Kernel):



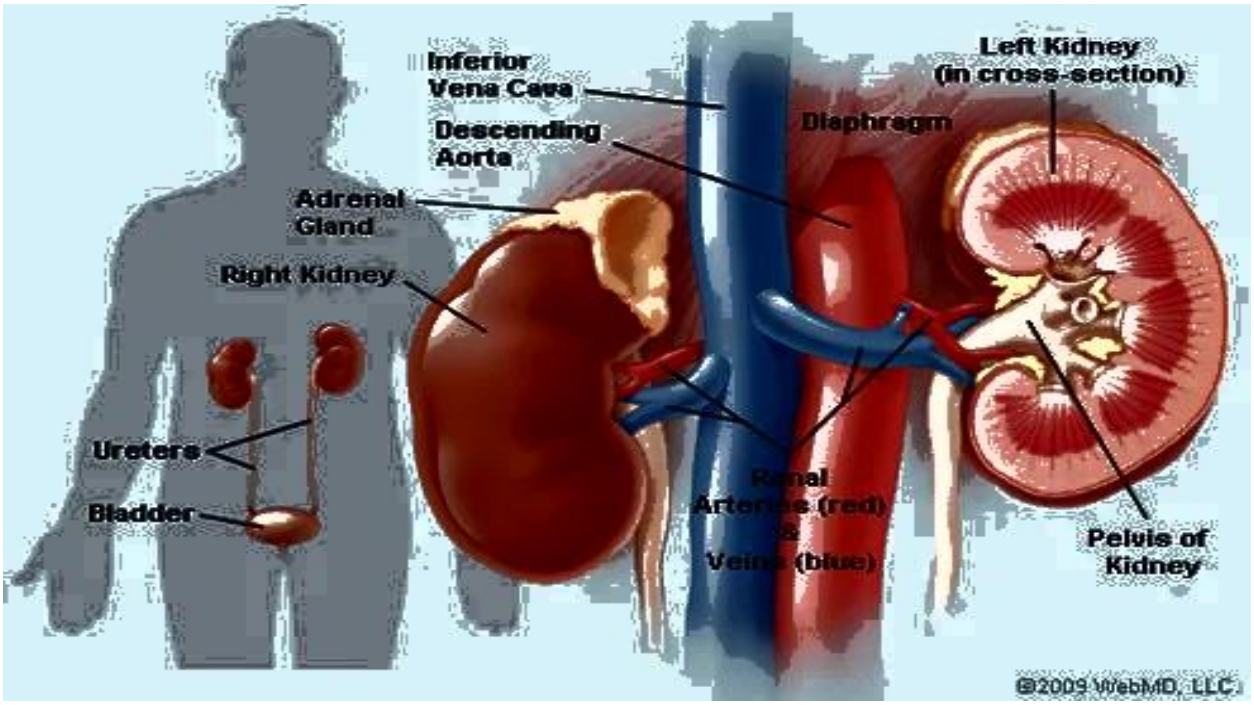

```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/kidneys-anatomy.jpeg", 0)
# equalize the histogram of the input image
histeq = cv2.equalizeHist(img)
cv2.imshow('Input', img)
cv2.imshow('Histogram equalized', histeq)
cv2.waitKey(0)
```

- Histogram equalization is a method that redistributes the pixel intensity values in an image such that the resulting histogram is approximately uniform. This means that it spreads out the pixel values across the entire range of possible values.
- The primary goal of histogram equalization is to improve the contrast of an image by making the dark and light regions more distinguishable.

Output Image (Histogram Equalizer):



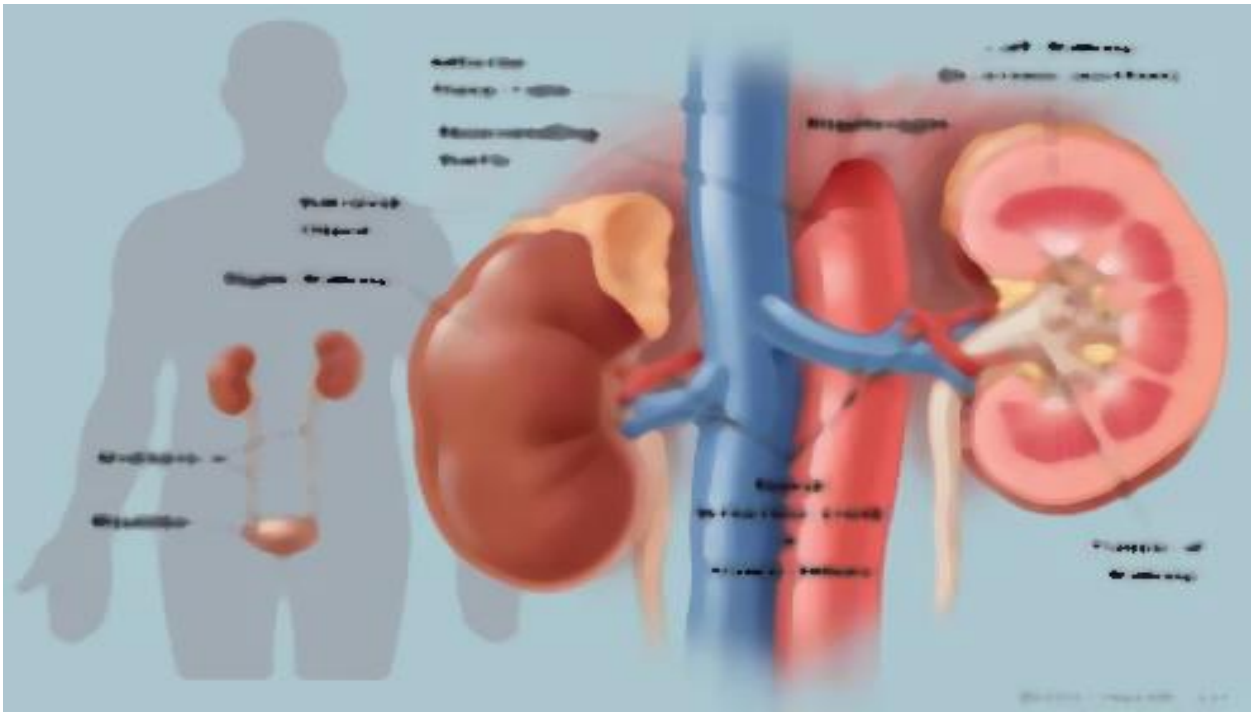
Output Image (Histogram Equalizer with RGB):




```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/kidneys-anatomy.jpeg")
output = cv2.medianBlur(img, ksize=7)
cv2.imshow('Input', img)
cv2.imshow('Median filter', output)
cv2.waitKey()
```

Median Filter neutralizes all the pixels & takes the average of the pixels.

Output Image (Median Filter):

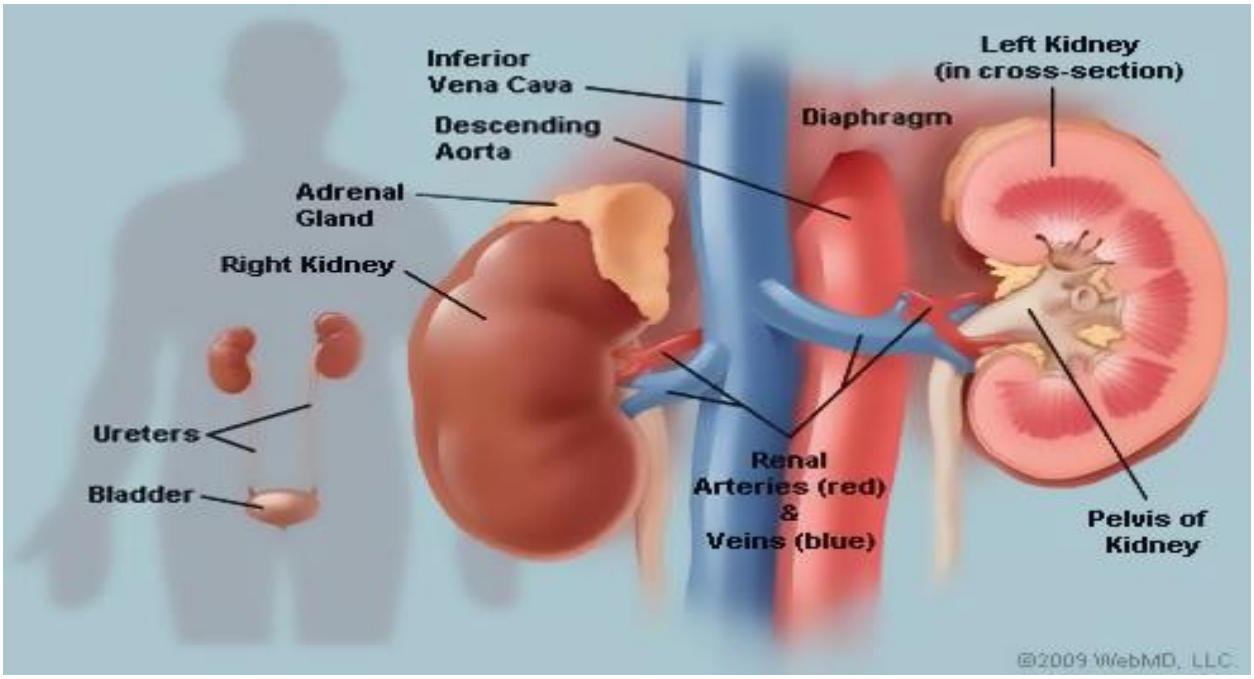


```
import cv2
import numpy as np
img = cv2.imread(r"G:/ACV CA 1/kidneys-anatomy.jpeg")
img_gaussian = cv2.GaussianBlur(img, (13,13), 0) # Gaussian Kernel Size 13x13
img_bilateral = cv2.bilateralFilter(img, 13, 70, 50)
cv2.imshow('Input', img)
cv2.imshow('Gaussian filter', img_gaussian)
cv2.imshow('Bilateral filter', img_bilateral)
cv2.waitKey()
```

Bilateral blur is used in image processing to smooth or blur an image while preserving important edges and details.

Gaussian blur is used to perform a simple and effective smoothing or blurring of an image.

Output Image (Bilateral Filter):



Output Image (Gaussian Blur Filter):