Digital Image Processing

Exercise 1 Solution

Atheer Hadi AL-Rammahi

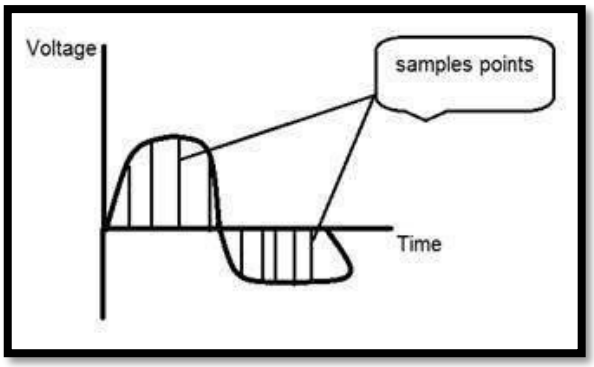
اثير هادى عيسى الرماحي

Exercise 1-Digital Image Processing (DIP)

Sampling and quantization: (Sampling and quantization.ipynb)

In Digital Image Processing, signals captured from the physical world need to be translated into digital form by "Digitization" Process. In order to become suitable for digital processing, an image function f(x,y) must be digitized both spatially and in amplitude. This digitization process involves two main processes called:

1. **Sampling**: Digitizing the co-ordinate value is called sampling. Since an analogue image is continuous not just in its co-ordinates (x axis), but also in its amplitude (y axis), so the part that deals with the digitizing of co-ordinates is known as sampling. In digitizing sampling is done on independent variable. In case of equation y = sin(x), it is done on x variable.

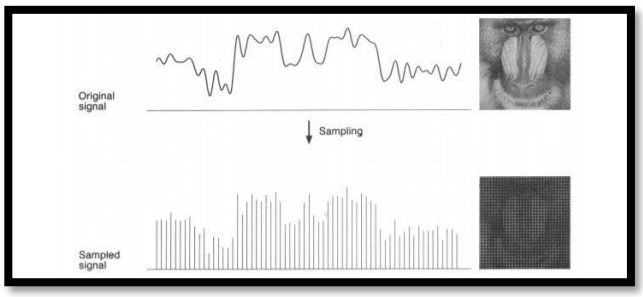


When looking at this image, we can see there are some random variations in the signal caused by noise. In sampling we reduce this noise by taking samples. It is obvious that more samples we take, the quality of the image would be more better, the noise would be more removed and same happens vice versa. However, if you take sampling on the x axis, the signal is not converted to digital format, unless you take sampling of the y-axis too which is known as quantization.

Sampling has a relationship with image pixels. The total number of pixels in an image can be calculated as Pixels = total no of rows * total no of columns. For example, let's say we have total of 36 pixels, that means we have a square image of 6X 6. As we know in sampling, that more samples eventually result in more pixels. So it means that of our continuous signal, we have taken 36 samples on x axis. That

refers to 36 pixels of this image. Also the number sample is directly equal to the number of sensors on CCD array.

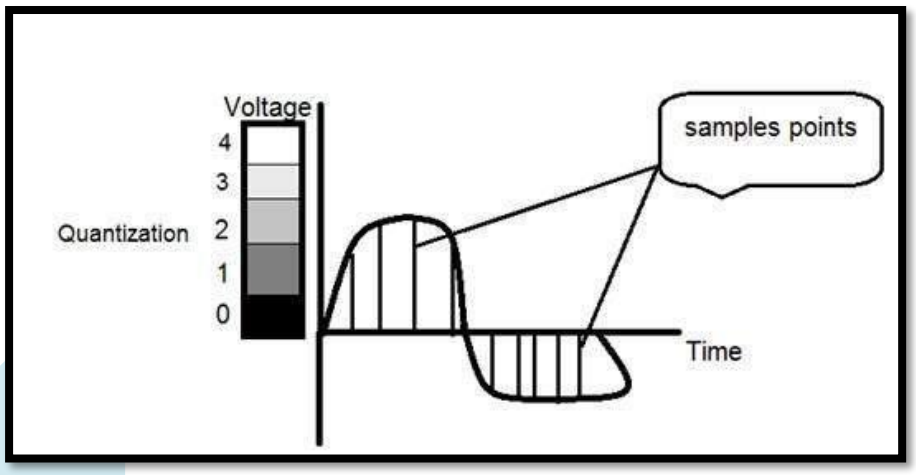
Here is an example for image sampling and how it can be represented using a graph[1].



2. **Quantization**: Digitizing the amplitude value is called quantization

Quantization is opposite to sampling because it is done on "y axis" while sampling is done on "x axis". Quantization is a process of transforming a real valued sampled image to one taking only a finite number of distinct values. Under quantization process the amplitude values of the image are digitized. In simple words, when you are quantizing an image, you are actually dividing a signal into quanta(partitions).

Now let's see how quantization is done. Here we assign levels to the values generated by sampling process. In the image showed in sampling explanation, although the samples has been taken, but they were still spanning vertically to a continuous range of gray level values. In the image shown below, these vertically ranging values have been quantized into 5 different levels or partitions. Ranging from 0 black to 4 white. This level could vary according to the type of image you want.

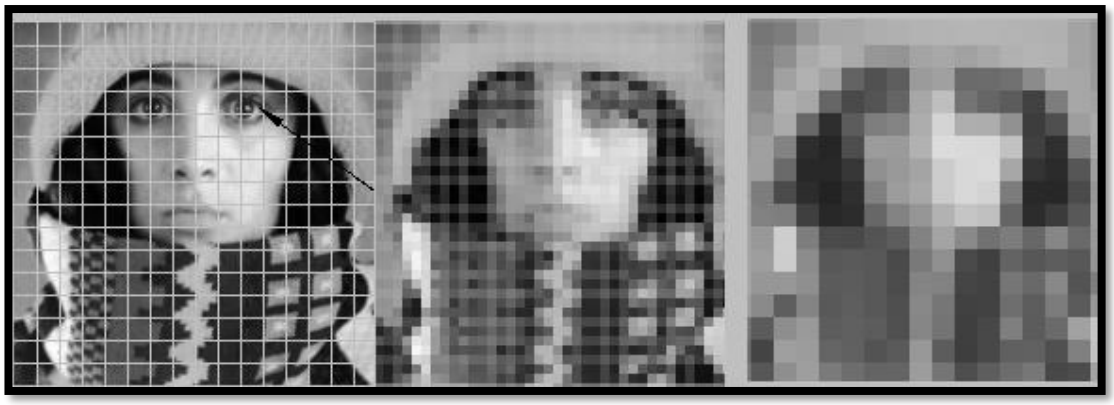


There is a relationship between Quantization with gray level resolution. The above quantized image represents 5 different levels of gray and that means the image formed from this signal, would only have 5 different colors. It would be a black and white image more or less with some colors of gray.

When we want to improve the quality of image, we can increase the levels assign to the sampled image. If we increase this level to 256, it means we have a gray scale image. Whatever the level which we assign is called as the gray level. Most digital IP devices uses quantization into k equal intervals. If b-bits per pixel are used,

$$\text{No. of quantization levels} = k = 2^b$$

The number of quantization levels should be high enough for human perception of fine shading details in the image. The occurrence of false contours is the main problem in image which has been quantized with insufficient brightness levels. Here is an example for image quantization process[2].



In Exercise:

1- Load and display image1 and answer the below question:

a) Computes the difference between values in neighboring pixels and Find the number of bytes and size of the image, and display each case.

b) Representation the image with unit-8 data type, default type (float 64) and by adding bias (128) and plot it.

c) How many bits can manage so that we still have a good image? Plot all of the cases.

Exercise 1-Digital Image Processing (DIP)

I believe you have already installed Python Shell/Jupyter Notebook/PyCharm or Visual Studio Code (to name a few) to program in python. Let's install the widely used package (OpenCV) to get started with and we are going to run the codes in each cell in a Jupyter Notebook[3].

Installing OpenCV Package for Image Preprocessing:

OpenCV is a pre-built, open-source CPU-only library (package) that is widely used for computer vision, machine learning, and image processing applications. It supports a good variety of programming languages including Python.

Install the OpenCV package using:

```
pip install opencv-python
```

Importing the Package:

```
import cv2
```

Reading an Image

Digital images could be classified into; colour images, grey-scale images, binary images, and multispectral images. A color image includes the color information for each pixel. Images having shades of grey as their only color are grayscale images while a binary image has exactly two colors, mostly black and white pixels. Multispectral images are images that capture image data ranging across the electromagnetic spectrum within some specific wavelength[4].

Let's get back to the coding part and read an image, for example, the image is shown below:

Exercise 1-Digital Image Processing (DIP)

This is an image of a coin. I am currently reading the image from my local directory.

```
# cv2.imread(path_to_image_with_file_extension, flag)
```

The usage code looks like this:

cv2.imread("D:/image processing/images/image1.jpg")

Here we are using the “imread” method of the cv2 package to read the image and the first parameter corresponds to the path of the image with its filename and extension. Here we are using the “imread” method of the cv2 package to read the image and the first parameter corresponds to the path of the image with its filename and extension, and the second one is the flag that you can set which tells the way, how to read in the image. If you like, you can replace the absolute path to the image here and try reading it from your local computer or even from the internet! If the image is present in your current working directory, you only need to specify the image name with its extension type.As far as the second parameter is concerned, if you like to read it as a grayscale image, you can specify the parameter to 0, -1 for reading the image as unchanged (reads the image as alpha or transparency channel if it is there) and by default, it is 1, as a color image.

Properties of an Image:

Shape:

Every image has a shape. The length of boundaries exhibited by the picture might be referred to as the shape i.e, the height and width.

```
print(img)
```

Result:

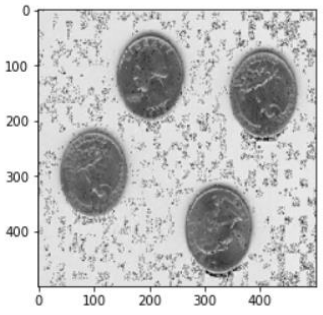
```
[[[ 17  17  17]
  [ 14  14  14]
  [205 205 205]
  ...
  [224 224 224]
  [218 218 218]
  [216 216 216]]

 [[ 12  12  12]
  [ 16  16  16]
  [226 226 226]
  ...
  [246 246 246]
```

```
  [248 248 248]
  [250 250 250]]

 [[ 11  11  11]
  [ 14  14  14]
  [230 230 230]
  ...
  [247 247 247]
  [249 249 249]
  [250 250 250]]

 ...

 [[ 12  12  12]
  [ 11  11  11]
  [237 237 237]
  ...
  [ 29  29  29]
  [235 235 235]
  [235 235 235]]

 [[ 12  12  12]
  [ 11  11  11]
  [237 237 237]
  ...
  [141 141 141]
  [239 239 239]
  [245 245 245]]

 [[ 11  11  11]
  [ 11  11  11]
  [236 236 236]
  ...
  [249 249 249]
  [244 244 244]
  [248 248 248]]]
```

Type:

We can know the type of the image using the "type" method. Using this method helps us to know how the image data is represented. Run the code as follows:

```
In [9]: print(img.dtype)
        plt.imshow(img)

        uint8

Out[9]: <matplotlib.image.AxesImage at 0x1fbe4140ca0>
```



Result:

```
uint8
```

Exercise 1-Digital Image Processing (DIP)

Image pixel values:

We can think of an image as a set of small samples. These samples are called pixels. For a better understanding, try zoom in on an image as much as possible. We can see the same divided into different squares. These are the pixels and when they are combined together they form an image[5].

Image Resolution: Image resolution could be defined as the number of pixels present in an image. The quality of the image increases when the number of pixels increases. We have seen earlier, the shape of the image which gives the number of rows and columns. This could be said as the resolution of that image. Some of the standard resolutions are that almost everyone knows are 320 x 240 pixels (mostly suitable on small screen devices), 1024 x 768 pixels (appropriate to view on standard computer monitors), 720 x 576 pixels(good to view on standard definition TV sets having 4:3 aspect ratio), 1280 x 720 pixels (for viewing on widescreen monitors),1280 x 1024 pixels (good for viewing on the full-screen size on LCD monitors with 5:4 aspect ratio), 1920 x 1080 pixels (for viewing on HD tv's) and now we even have 4K, 5K, and 8K resolutions which are 3840 x 2160 pixels, 5120 × 2880 pixels and 7,680 x 4,320 pixels respectively supported by ultra high definition monitors and televisions[5].

When we multiply the number of columns and number of rows, we can obtain the total number of pixels present in the image. For example, in a 320 x 240 image, the total number of pixels present in it is 76,800 pixels.

```
In [35]: # We can obtain a total number of elements by using img.size
         total_number_of_elements= img.size
         print(total_number_of_elements)

         750000
```

Result:

```
750000
```

Exercise 1-Digital Image Processing (DIP)

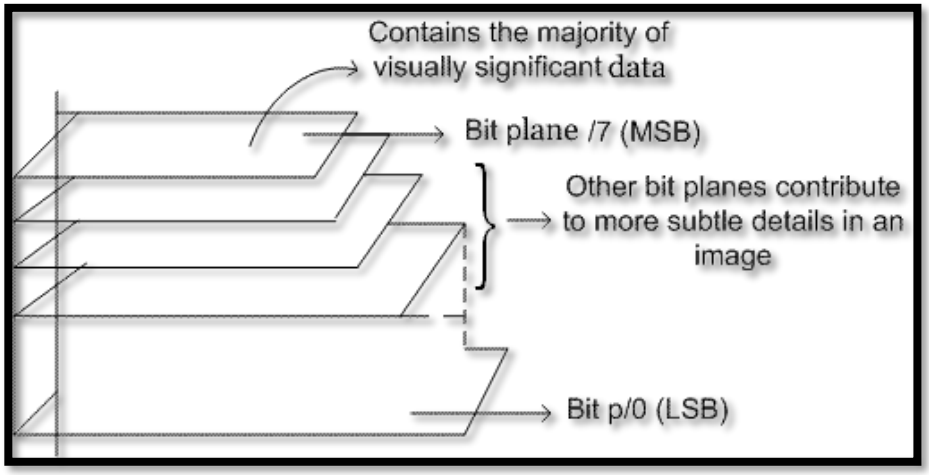
Viewing the Image:

Let us see how to display the image in a window. For that, we have to create a GUI window to display the image on the screen. The first parameter has to be the title of the GUI window screen, specified in string format. We can show the image in a pop-up window using the cv2.imshow() method. But, when you try to close it, you might feel stuck with its window. So to combat that, we can use a simple "waitKey" method. Try out this code part in new a cell:

```
# Displaying the image
cv2.imshow(window_name, image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Extracting the image bit planes and reconstructing them

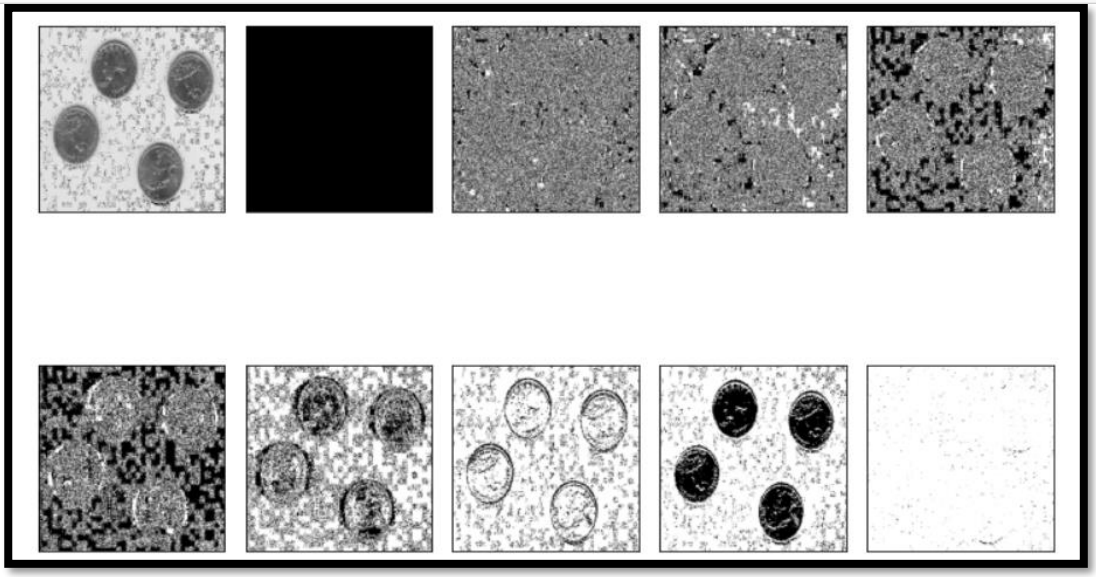
We can divide an image into different levels of bit planes. For example, divide an image into 8-bit (0-7) planes, where the last few planes contain the majority of information for an image[6].



Exercise 1-Digital Image Processing (DIP)

Now we are defining a function to extract each of the 8 level bit planes of the image.

In [33]:
```
def extract_bit_plane(cd):
    #  extracting all bit one by one
    # from 1st to 8th in variable
    # from c1 to c8 respectively
    c1 = np.mod(cd, 2)
    c2 = np.mod(np.floor(cd/2), 2)
    c3 = np.mod(np.floor(cd/4), 2)
    c4 = np.mod(np.floor(cd/8), 2)
    c5 = np.mod(np.floor(cd/16), 2)
    c6 = np.mod(np.floor(cd/32), 2)
    c7 = np.mod(np.floor(cd/64), 2)
    c8 = np.mod(np.floor(cd/128), 2)
    # combining image again to form equivalent to original grayscale image
    cc = 2 * (2 * (2 * c8 + c7) + c6) # reconstructing image  with 3 most significant bit planes
    to_plot = [cd, c1, c2, c3, c4, c5, c6, c7, c8, cc]
    fig, axes = plt.subplots(nrows=2, ncols=5,figsize=(10, 8), subplot_kw={'xticks': [], 'yticks': []})
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for ax, i in zip(axes.flat, to_plot):
        ax.imshow(i, cmap='gray')
    plt.tight_layout()
    plt.show()
    return cc
reconstructed_image = extract_bit_plane(img)
```



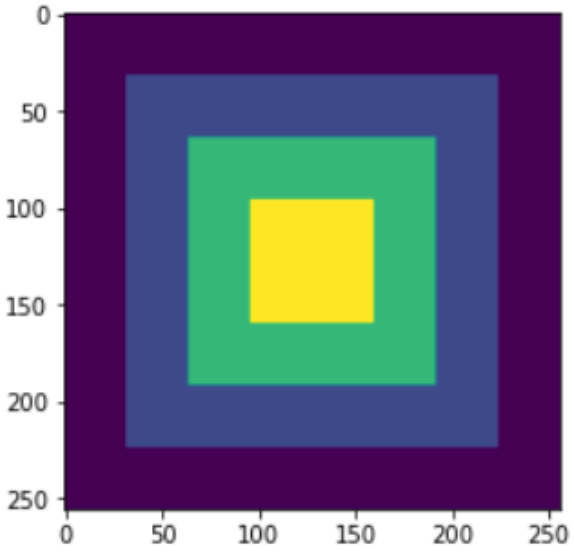
Constructing a small synthetic image

We can try to generate a synthetic image containing four concentric squares with four different pixel intensity values, 40, 80, 160, and 220.

```
In [34]: con_img = np.zeros([256, 256])
con_img[0:32, :] = 40 # upper row
con_img[:, :32] = 40 #left column
con_img[:, 224:256] = 40 # right column
con_img[224:, :] = 40 # lower row
con_img[32:64, 32:224] = 80 # upper row
con_img[64:224, 32:64] = 80 # left column
con_img[64:224, 192:224] = 80 # right column
con_img[192:224, 32:224] = 80 # lower row
con_img[64:96, 64:192] = 160 # upper row
con_img[96:192, 64:96] = 160 # left column
con_img[96:192, 160:192] = 160 # right column
con_img[160:192, 64:192] = 160 # lower row
con_img[96:160, 96:160] = 220
plt.imshow(con_img)
```

The resulting image would be looking like this:

```
Out[34]: <matplotlib.image.AxesImage at 0x2aa33ab7130>
```



Neighbors of a pixel

A pixel p at (x,y) has 4-horizontal/vertical neighbours at (x+1,y), (x-1,y), (x,y+1) and (x,y-1). These are called the 4-neighbours of p : N4(p).

A pixel p at (x,y) has 4 diagonal neighbours at (x+1,y+1), (x+1,y-1), (x-1,y+1) and (x-1,y-1). These are called the diagonal-neighbours of p : ND(p).

Exercise 1-Digital Image Processing (DIP)

The 4-neighbours and the diagonal neighbours of p are called 8-neighbours of p : N8(p)[7].

F(x-1, y-1)	F(x-1, y)	F(x-1, y+1)
F(x, y-1)	F(x,y)	F(x, y+1)
F(x+1, y-1)	F(x+1, y)	F(x+1, y+1)

N_8 (p)

 # finding the difference between neighboring elements

```
result = np.diff(img)
print(result)
```

Result:

```
[[[0 0]
  [0 0]
  [0 0]
  ...
  [0 0]
  [0 0]
  [0 0]]

 [[0 0]
  [0 0]
  [0 0]
  ...
  [0 0]
  [0 0]
  [0 0]]

 [[0 0]
  [0 0]
  [0 0]
  ...
  [0 0]
  [0 0]
  [0 0]]

 ...

 [[0 0]
  [0 0]
  [0 0]
  ...
  [0 0]
  [0 0]
  [0 0]]

 [[0 0]
```

```
result = np.diff(img, axis = 1)
print(result)
```

```
[[[253 253 253]
  [191 191 191]
  [101 101 101]
  ...
  [249 249 249]
  [250 250 250]
  [254 254 254]]

 [[  4   4   4]
  [210 210 210]
  [170 170 170]
  ...
  [254 254 254]
  [  2   2   2]
  [  2   2   2]]

 [[  3   3   3]
  [216 216 216]
  [168 168 168]
  ...
  [255 255 255]
  [  2   2   2]
  [  1   1   1]]

 ...

 [[255 255 255]
  [226 226 226]
  [248 248 248]
  ...
  [ 80  80  80]
  [206 206 206]
  [  0   0   0]]

 [[255 255 255]
  [226 226 226]
```

```
result = np.diff(img, axis = 0)
print(result)
```

```
[[[251 251 251]
  [  2   2   2]
  [ 21  21  21]
  ...
  [ 22  22  22]
  [ 30  30  30]
  [ 34  34  34]]

 [[255 255 255]
  [254 254 254]
  [  4   4   4]
  ...
  [  1   1   1]
  [  1   1   1]
  [  0   0   0]]

 [[  1   1   1]
  [253 253 253]
  [  0   0   0]
  ...
  [241 241 241]
  [245 245 245]
  [249 249 249]]

 ...

 [[255 255 255]
  [255 255 255]
  [  0   0   0]
  ...
  [ 10  10  10]
  [252 252 252]
  [254 254 254]]

 [[  0   0   0]
  [  0   0   0]
```

```
  [0 0]                 [248 248 248]          [  0   0   0]
  [0 0]                 ...                    ...
  ...                   [179 179 179]          [112 112 112]
  [0 0]                 [ 98  98  98]          [  4   4   4]
  [0 0]                 [  6   6   6]]         [ 10  10  10]]
  [0 0]]
                       [[  0   0   0]         [[255 255 255]
 [[0 0]                 [225 225 225]          [  0   0   0]
  [0 0]                 [249 249 249]          [255 255 255]
  [0 0]                 ...                    ...
  ...                   [ 18  18  18]          [108 108 108]
  [0 0]                 [251 251 251]          [  5   5   5]
  [0 0]                 [  4   4   4]]]        [  3   3   3]]]
  [0 0]]
```

File size:

As the resolution and bit depth of digital detectors continue to improve, the user often gets excited about the improved image quality. Your IT person however may not share your enthusiasm when they see the size of the files being stored. One quick way to calculate the file size of your new system is to calculate the total number of pixels in the detector, multiply that by the number of bits of bit depth and divide the result by 8 (because there are 8 bits in a byte)

1 Byte = 8 Bit 1 Kilobyte = 1,024 Bytes 1 Megabyte = 1,048,576 Bytes 1 Gigabyte = 1,073,741,824 Bytes

Step 1: Multiply the detectors number of horizontal pixels by the number of vertical pixels to get the total number of pixels of the detector.

Step 2: Multiply total number of pixels by the bit depth of the detector (16 bit, 14 bit etc.) to get the total number of bits of data.

Step 3: Dividing the total number of bits by 8 equals the file size in bytes.

Step 4: Divide the number of bytes by 1024 to get the file size in kilobytes. Divide by 1024 again and get the file size in megabytes.

Examples:

Perkin Elmer 1621: 2048 x 2048 = 4,194,304 (4.2 megapixel Detector)

4,194,304 pixels X 16 bit = 67,108,864 ÷ 8bits = 8,388,608 Bytes ÷1024 = 8,192 Kilobytes ÷ 1024 = 8 Megabytes

VARIAN 2520: 1,920 x 1536 = 2949120 (2.95 Megapixel Detector)

2,949,120 X 16 bit = 47185920 ÷ 8bits = 5,898,240 Bytes 5,760 Kilobytes ÷ 1024 = 5.625 Megabytes)

In Sumary:

Of Pixels X Bit Depth ÷ 8 ÷ 1024 ÷ 1024 = File Size in Megabytes (MB)

```
import os
file_size = os.path.getsize('D:/image processing/excrise/HW1-1400-2/images/image1.jpg')
print("File Size is :", file_size, "bytes")
```

```
File Size is : 138257 bytes
```

Result:

```
File Size is : 138257 bytes
```

Data type:

Various kinds of images can be displayed in image processing . This list describes them:

- Binary Images: Binary images contain only two values (off or on). The off value is usually 0 and the on value is usually 1. This type of image is commonly used as a multiplier to mask regions within another image.
- Grayscale Images: Grayscale images represent intensities. Pixels range from least intense (black) to most intense (white). Pixel values usually range from 0 to 255 or are scaled to this range when displayed.
- Indexed Images: Instead of intensities, a pixel value within an indexed image relates to a color value within a color lookup table. Since indexed images reference color tables composed of up to 256 colors, the data values of these images are usually scaled to range from 0 to 255.
- RGB Images: Within the three-dimensional array of an RGB image, two of the dimensions specify the location of a pixel within an image. The other dimension specifies the color of each pixel. The color dimension always has a size of 3 and is composed of the red, green, and blue color bands (channels) of the image[8].

Pixel Values in Image Data

Pixel values in an image file can be stored in many different data types. The original data type of an image is reflected in IDL when importing the image, but you can convert

the type once the image is stored in an IDL variable. The following types are commonly used for images[8]:

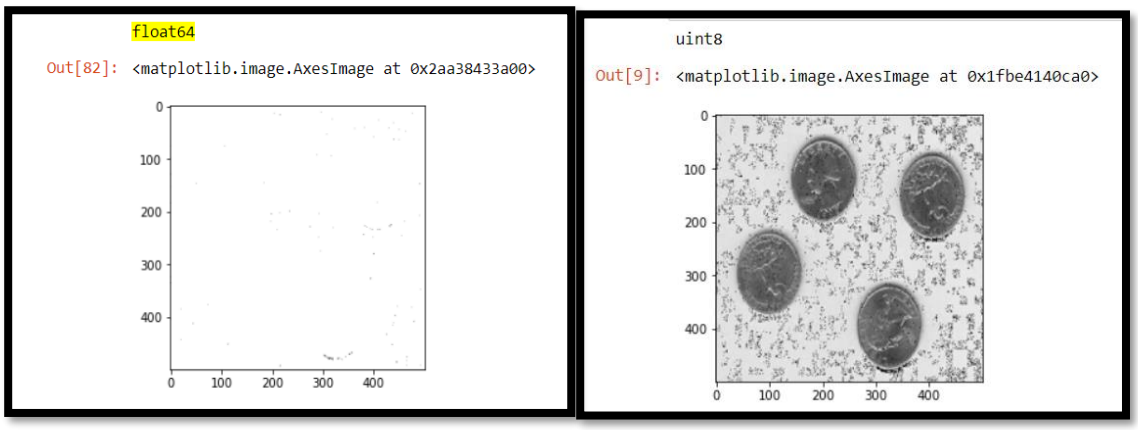
- Byte: An 8-bit unsigned integer ranging from 0 to 255. Pixels in images are commonly represented as byte data.
- Unsigned Integer: A 16-bit unsigned integer ranging from 0 to 65535.
- Signed Integer: A 16-bit signed integer ranging from -32,768 to +32,767.
- Unsigned Longword Integer: A 32-bit unsigned integer ranging from 0 to approximately four billion.
- Longword Integer: A 32-bit signed integer ranging in value from approximately minus two billion to plus two billion.
- Floating-point: A 32-bit, single-precision, floating-point number ranging from -1038 to 1038, with approximately 6 or 7 decimal places of significance.
- Double-precision: A 64-bit, double-precision, floating-point number ranging from -10308 to 10308 with approximately 14 decimal places of significance.

While pixel values are commonly stored in files as whole numbers, they are usually converted to floating-point or double-precision data types prior to performing numerical computations. See the examples section of REFORM for more information. Use these IDL functions to convert data from one type to another:

- ✓ BYTE: Convert to byte
- ✓ BYTSCL: Scale data to range from 0 to 255 and then convert to byte
- ✓ UINT: Convert to 16-bit unsigned integer
- ✓ FIX: Convert to 16-bit integer, or optionally other type
- ✓ ULONG: Convert to 32-bit unsigned integer
- ✓ LONG: Convert to 32-bit integer
- ✓ FLOAT: Convert to floating-point
- ✓ DOUBLE: Convert to double-precision floating-point

```
In [82]: img = img.astype('float64')
         print(img.dtype)
         plt.imshow(img)
```

```
print(img.dtype)
plt.imshow(img)
```

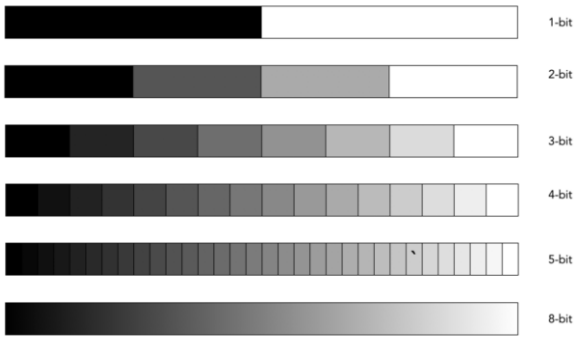


How many bits in an image?

When it comes to bits and images it can become quite confusing. For example, are JPEGs 8-bit, or 24-bit? Well they are both.

Basic bits

A bit is a binary digit, i.e. it can have a value of 0 or 1. When something is X-bit, it means that it has X binary digits, and 2X possible values. Figure 1 illustrates various values for X as grayscale tones. For example, a 2-bit image will have 22, or 4 values (0,1,2,3)[9].



An 8-bit image has 28 possible values for bits – i.e. 256 values ranging from 0..255. In terms of binary values, 255 in binary is 11111111, 254 is 11111110, …, 1 is 00000001, and 0 is 00000000. Similarly, a 16-bit means there are 216 possible values, from 0..65535. The number of bits is sometimes called the *bit-depth*.

Bits-per-pixel

Images typically describe bits in terms of bits-per-pixel (BPP). For example a grayscale image may have 8-BPP, meaning each pixel can have one of 256 values from 0 (black)

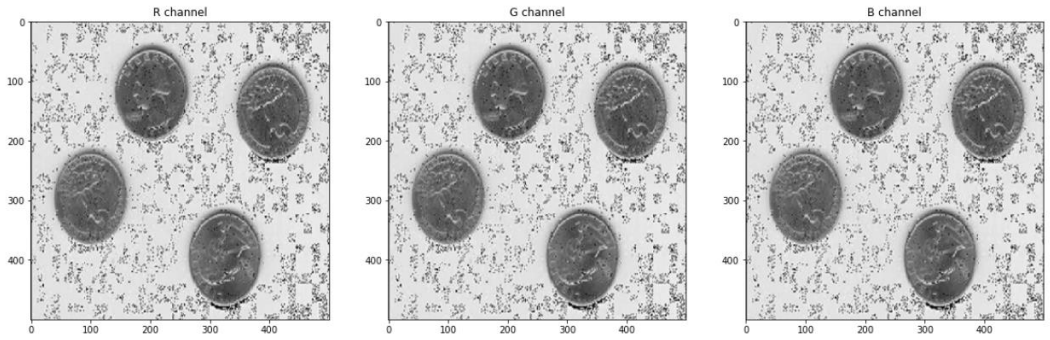
to 255 (white). Colour images are a little different because they are typically composed of three component images, red (R), green (G), and blue (B). Each component image has its own bit-depth. So a typical 24-bit RGB image is composed of three 8-BPP component images, i.e. 24-BPP RGB = 8-BPP (R) + 8-BPP (G) + 8-BPP (B).

```
#View and change bit depth
from PIL import Image
img = Image.open("D:/image processing/excrise/HW1-1400-2/images/image1.jpg")
print(img.getbands())
```

```
('R', 'G', 'B')
```

```
import matplotlib.image as mpimg
rgb_image = mpimg.imread("D:/image processing/excrise/HW1-1400-2/images/image1.jpg")
r = rgb_image[:, :, 0]
g = rgb_image[:, :, 1]
b = rgb_image[:, :, 2]
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize = (20,10))
ax1.set_title('R channel')
ax1.imshow(r, cmap = 'gray')
ax2.set_title('G channel')
ax2.imshow(g, cmap = 'gray')
ax3.set_title('B channel')
ax3.imshow(b, cmap = 'gray')
```

```
Out[93]: <matplotlib.image.AxesImage at 0x2aa38549790>
```



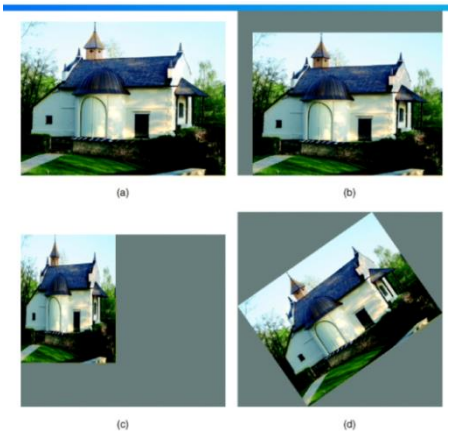
The *colour* depth of the image is then 2563 or 16,777,216 colours (or 2563, 28=256 for each of the component images). A 48-bit RGB image contains three component images, R, G, and B, each having 16-BPP, for 248 or 281,474,976,710,656 colours

Exercise 1-Digital Image Processing (DIP)

Geometrical spatial operations:

Geometric operations modify the geometry of an image by repositioning pixels in a constrained way. In other words, rather than changing the pixel values of an image, they modify the spatial relationships between groups of pixels representing features or objects of interest within the image[10]

Examples of typical geometric operations: (a) original image; (b) translation (shifting); (c) scaling (resizing); (d) rotation.



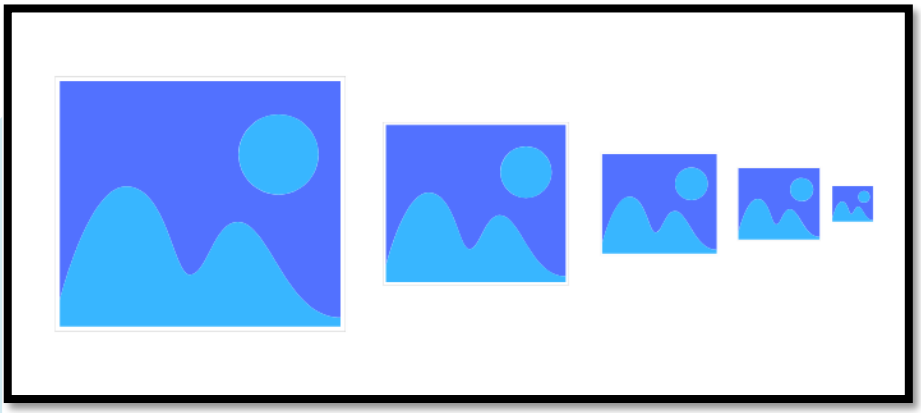
2- Perform the scaling (1.5*1.5), translation and rotation (90°) operations for image2 and plot the final image in each case

Scaling an Image: -

Scaling operation increases/reduces size of an image.[11]

To resize an image in Python, you can use cv2.resize() function of OpenCV library cv2. Resizing, by default, does only change the width and height of the image. The aspect ratio can be preserved or not, based on the requirement. Aspect Ratio can be preserved by calculating width or height for given target height or width respectively.

In this tutorial, we shall learn how to resize image in Python using OpenCV library.



Exercise 1-Digital Image Processing (DIP)

```
cv2.resize(src, dsize[, dst[, fx[, fy[, interpolation]]]])
```

where

- `src` is the source, original or input image in the form of numpy array
- `dsize` is the desired size of the output image, given as tuple
- `fx` is the scaling factor along X-axis or Horizontal axis
- `fy` is the scaling factor along Y-axis or Vertical axis
- `interpolation` could be one of the following values.
  - INTER_NEAREST
  - INTER_LINEAR
  - INTER_AREA
  - INTER_CUBIC
  - INTER_LANCZOS4

In [15]:
```
def image2():
    """
    Edit image2.jpg.

    Returns
    -------
    None.

    """
    img = cv2.imread("D:/image processing/images/image2.jpg", 0)
    filtered_img = cv2.resize(img, (0, 0), fx=1.5, fy=1.5)
    height, width = img.shape[:2]
    center = (width/2, height/2)
    #the above center is the center of rotation axis
    # use cv2.getRotationMatrix2D() to get the rotation matrix
    rotate_matrix = cv2.getRotationMatrix2D(center=center, angle=90, scale=1)
    # Rotate the image using cv2.warpAffine
    rotated_image = cv2.warpAffine(src=img, M=rotate_matrix, dsize=(width, height))
    # get tx and ty values for translation
    # you can specify any value of your choice
    tx, ty = width / 4, height / 4
    # create the translation matrix using tx and ty, it is a NumPy array
    translation_matrix = np.array([
        [1, 0, tx],
        [0, 1, ty]
        ], dtype=np.float32)
    # apply the translation to the image
    translated_image = cv2.warpAffine(src=img, M=translation_matrix, dsize=(width, height))
    image_median = cv2.medianBlur(img,5)
    plot_before_after(img, filtered_img)
    plot_before_after(img, rotated_image)
    plot_before_after(img, translated_image)
    plot_before_after(img, image_median)
```

Translation of Images using OpenCV:

In computer vision, translation of an image means shifting it by a specified number of pixels, along the x and y axes. Let the pixels by which the image needs to shifted be tx and ty. Then you can define a translation matrix M:[12]

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

Now, there are a few points you should keep in mind while shifting the image by tx and ty values.

- Providing positive values for tx will shift the image to right and negative values will shift the image to the left.
- Similarly, positive values of ty will shift the image down while negative values will shift the image up.

Follow these steps to translate an image, using OpenCV:

a) First, read the image and obtain its width and height.

b) Next, like you did for rotation, create a transformation matrix, which is a 2D array. This matrix contains the information needed to shift the image, along the x and y axes.

c) Again, as in rotation, use the warpAffine() function, in this final step, to apply the affine transformation.

Exercise 1-Digital Image Processing (DIP)

Image Rotation using OpenCV

You can rotate an image by a certain angle θ by defining a transformation matrix M. This matrix is usually of the form[12]:

$$M = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

OpenCV provides the ability to define the center of rotation for the image and a scale factor to resize the image as well. In that case, the transformation matrix gets modified.

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot c_x - \beta \cdot c_y \\ -\beta & \alpha & \beta \cdot c_x + (1-\alpha)\cdot c_y \end{bmatrix}$$

In the above matrix:

$$\alpha = scale \cdot \cos\theta$$
$$\beta = scale \cdot \sin\theta$$

where c_x & c_y are the coordinates along which the image is rotated.

OpenCV provides the getRotationMatrix2D() function to create the above transformation matrix.

The following is the syntax for creating the 2D rotation matrix:

```
getRotationMatrix2D(center, angle, scale)
```

The `getRotationMatrix2D()` function takes the following arguments:

- `center`: the center of rotation for the input image
- `angle`: the angle of rotation in degrees
- `scale`: an isotropic scale factor which scales the image up or down according to the value provided

If the *angle* is positive, the image gets rotated in the counter-clockwise direction. If you want to rotate the image clockwise by the same amount, then the angle needs to be negative.

Rotation is a three-step operation:

1. First, you need to get the center of rotation. This typically is the center of the image you are trying to rotate.

2. Next, create the 2D-rotation matrix. OpenCV provides the getRotationMatrix2D() function that we discussed above.
3. Finally, apply the affine transformation to the image, using the rotation matrix you created in the previous step. The warpAffine() function in OpenCV does the job.

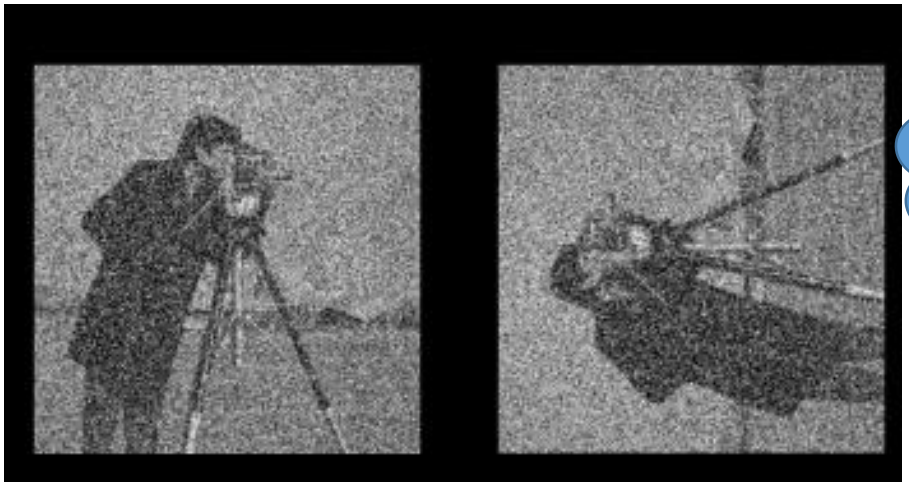
The warpAffine() function applies an affine transformation to the image. After applying affine transformation, all the parallel lines in the original image will remain parallel in the output image as well.

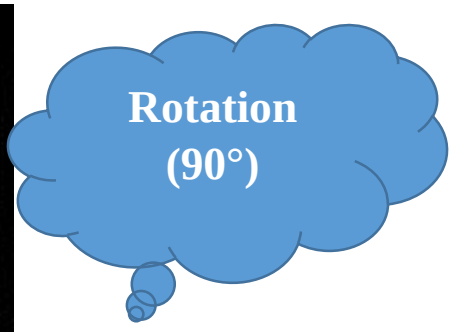
The complete syntax for warpAffine() is given below:

```
warpAffine(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]])
```

The following are the arguments of the function:

- `src`: the source mage
- `M`: the transformation matrix
- `dsize`: size of the output image
- `dst`: the output image
- `flags`: combination of interpolation methods such as INTER_LINEAR or INTER_NEAREST
- `borderMode`: the pixel extrapolation method
- `borderValue`: the value to be used in case of a constant border, has a default value of 0





Exercise 1-Digital Image Processing (DIP)

Filtering in Spatial Domain:

Spatial Filtering: technique is used directly on pixels of an image. Mask is usually considered to be added in size so that it has specific center pixel. This mask is moved on the image such that the center of the mask traverses all image pixels[13].

Classification on the basis of linearity:

There are two types:

1. Linear Spatial Filter

2. Non-linear Spatial Filter

Spatial Filter Types:

- **Smoothing Filters**: Aim to remove some small isolated detailed pixels by some form of averaging of the pixels in the masked neighborhood. These are also called low pass filters.

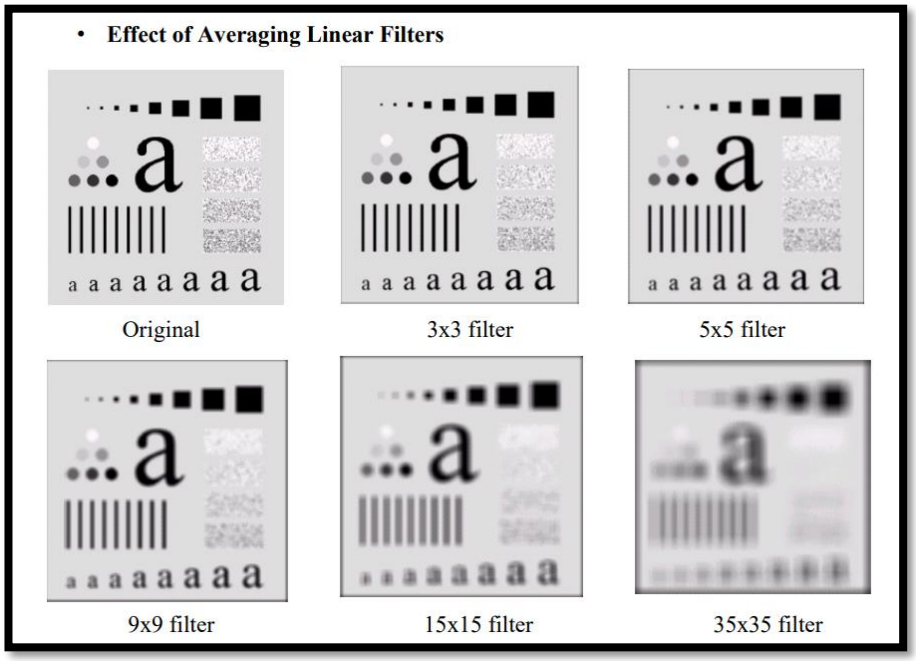
  Examples include **Weighted Average**, **Gaussian**, and order statistics filters.
- **Sharpening Filters**: aiming at highlighting some features such as edges or boundaries of image objects.

  Examples include the **Laplacian**, and **Gradient filters.**

Spatial filters are also classified in terms of mask size (e.g. 3x3, 5x5, or 7x7).

Smoothing Filters[14]:

- Particularly useful for blurring and noise reduction.
- It works by reducing sharp transition in grey levels.
- Noise reduction is accomplished by blurring with linear or nonlinear filters (e.g. order statistics filters).
- Beside reducing noise, smoothing filters may remove some significant features and reduce image quality.
- Increased filter size result in increased level of blurring and reduced image quality.
- Subtracting a blurred version of an image from the original may be used as a sharpening procedure.
-



Sharpening Spatial Filters[15]:

- Sharpening aims to highlight fine details (e.g. edges) in an image, or enhance detail that has been blurred through errors or imperfect capturing devices.
- Image blurring can be achieved using averaging filters, and hence sharpening can be achieved by operators that invert averaging operators.
- In mathematics averaging is equivalent to the concept of integration along the gray level range:

3- Apply each spatial filters to the image according to the below table and compare the original image and the image after using the filter

section	Filter name	Size of Filter	Image
a	Average(Mean) filter	5 * 5	image1
b	Median filter	5 * 5	Image2
c	Laplacian filter	5 * 5	Image3
d	Sobel(X and Y) filter	7 * 7	Image4

Exercise 1-Digital Image Processing (DIP)

Images may contain various types of noises that reduce the quality of the image. Blurring or smoothing is the technique for reducing the image noises and improve its quality. Usually, it is achieved by convolving an image with a low pass filter that removes high-frequency content like edges from the image. In this tutorial, we will see methods of Averaging, Gaussian Blur, and Median Filter used for image smoothing and how to implement them using python OpenCV, built-in functions of **cv2.blur()**, **cv2.GaussianBlur()**, **cv2.medianBlur().**

Let us first import the OpenCV library.

Importing OpenCV Library:

```
import cv2
```

a	Average(Mean) filter	5 * 5	image1

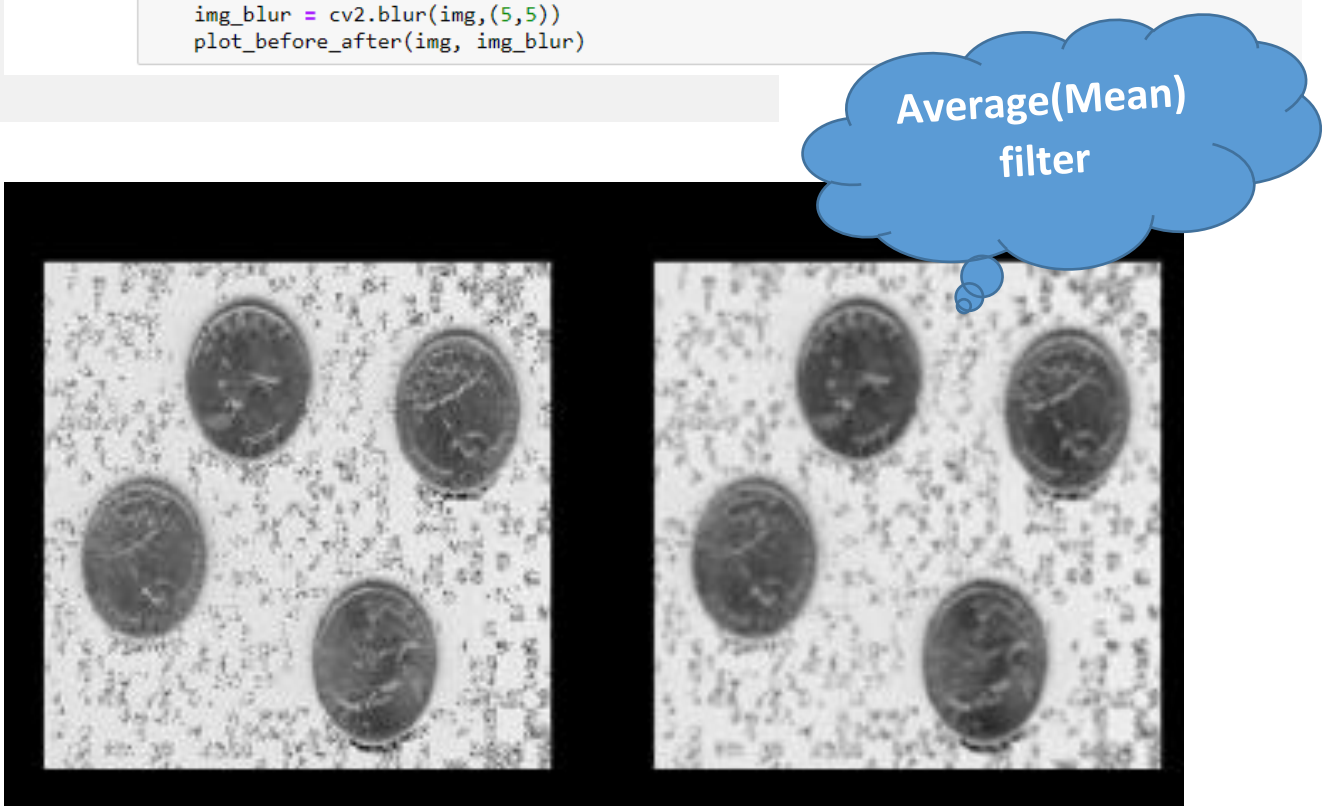
a) Averaging Filter: Averaging of the image is done by applying a convolution operation on the image with a normalized box filter. In convolution operation, the filter or kernel is slides across an image and the average of all the pixels is found under the kernel area and replace this average with the central element of the image[16].

Note: The smoothing of an image depends upon the kernel size. If Kernel size is large then it removes the small feature of the image. But if the kernel size is too small then it is not able to remove the noise.

cv2.blur(src, ksize, dst, anchor, borderType)

src: It is the image whose is to be blurred.
ksize: A tuple representing the blurring kernel size.
dst: It is the output image of the same size and type as src.
anchor: It is a variable of type integer representing anchor point and it's default value Point is (-1, -1) which means that the anchor is at the kernel center.
borderType: It depicts what kind of border to be added. It is defined by flags like cv2.BORDER_CONSTANT, cv2.BORDER_REFLECT, etc
Return Value: It returns an image.

```
In [14]: def image1():
             """
             Edit image1.jpg.

             Returns
             -------
             None.

             """
             img = cv2.imread("D:/image processing/images/image1.jpg", 0)
             img_blur = cv2.blur(img,(5,5))
             plot_before_after(img, img_blur)
```



a) **Median filter**: The median filter technique is very similar to the averaging filtering technique shown above. The only difference is cv2.medianBlur() computes the median of all the pixels under the kernel window and the central pixel is replaced with this median value instead of the average value[17].

Note: This is highly effective in removing salt-and-pepper noise.

cv2.medianBlur(src,dst,ksize)
src : It is the image that is to be blurred.
dst : destination array of the same size and type as src.
ksize : aperture linear size; it must be odd and greater than 1, for example 3, 5, 7 ...

Median Blurring always reduces the noise effectively because in this filtering technique the central element is always replaced by some pixel value in the image. But in the above filters, the central element is a newly calculated value which may be a pixel value in the image or a new value.

```
In [14]: def image1():
             """
             Edit image1.jpg.

             Returns
             -------
             None.

             """
             img = cv2.imread("D:/image processing/images/image1.jpg", 0)
             img_blur = cv2.blur(img,(5,5))
             plot_before_after(img, img_blur)
```

b) **Laplacian filter:** Unlike the Sobel edge detector, the Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass[18]. A kernel used in this Laplacian detection looks like this:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

If we want to consider the diagonals, we can use the kernel below:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
cv2.Laplacian(src, ddepth, other_options...)
```

where **ddepth** is the desired depth of the destination image.

c	Laplacian filter	5 * 5	Image3

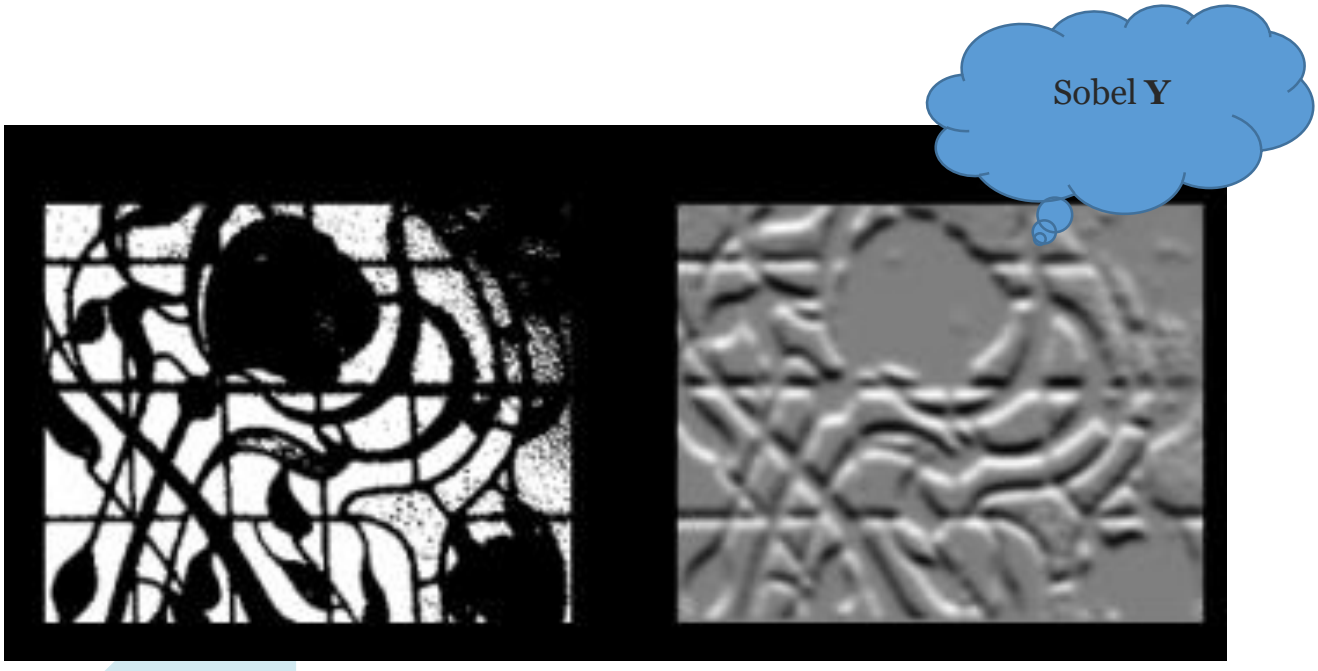
```
In [16]: def image3():
             """
             Edit image3.jpg.

             Returns
             -------
             None.

             """
             img = cv2.imread("D:/image processing/images/image3.jpg", 0)
             # remove noise
             filtered_img = cv2.GaussianBlur(img,(5,5),0)
             # convolute with proper kernels
             laplacian = cv2.Laplacian(filtered_img,cv2.CV_64F)
             plot_before_after(img, laplacian)
```

c) **Sobel Edge Detection**: Sobel edge detector is a gradient based method based on the first order derivatives. It calculates the first derivatives of the image separately for the X and Y axes[19]. The operator uses two 3X3 kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. The picture below shows Sobel Kernels in x-dir and y-dir:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

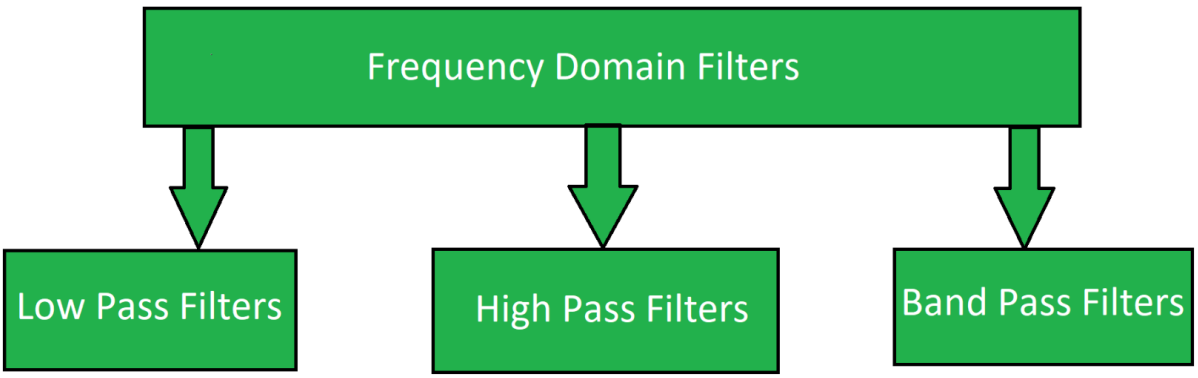
Exercise 1-Digital Image Processing (DIP)

In [17]:
```
def image4():
    """
    Edit image4.jpg.

    Returns
    -------
    None.

    """
    img = cv2.imread("D:/image processing/images/image4.jpg", 0)
    # remove noise
    filtered_img = cv2.GaussianBlur(img,(7,7),0)
    # convolute with proper kernels
    sobelx = cv2.Sobel(filtered_img,cv2.CV_64F,1,0,ksize=7)  # x
    sobely = cv2.Sobel(filtered_img,cv2.CV_64F,0,1,ksize=7)  # y
    plot_before_after(img, sobelx)
    plot_before_after(img, sobely)
```



Filtering in Frequency Domain:

are used for smoothing and sharpening of image by removal of high or low frequency components. Sometimes it is possible of removal of very high and very low frequency. Frequency domain filters are different from spatial domain filters as it basically focuses on the frequency of the images. It is basically done for two basic operation i.e., Smoothing and Sharpening[20].



Classification of Frequency Domain Filters

1. **Low pass filter**: Low pass filter removes the high frequency components that means it keeps low frequency components. It is used for smoothing the image. It is used to smoothen the image by attenuating high frequency components and preserving low frequency components. Mechanism of low pass filtering in frequency domain is given by:

```
G(u, v) = H(u, v) . F(u, v)
where F(u, v) is the Fourier Transform of original image
and H(u, v) is the Fourier Transform of filtering mask
```

2. **High pass filter:** High pass filter removes the low frequency components that means it keeps high frequency components. It is used for sharpening the image. It is used to sharpen the image by attenuating low frequency components and preserving high frequency components. Mechanism of high pass filtering in frequency domain is given by:

```
H(u, v) = 1 - H'(u, v)
where H(u, v) is the Fourier Transform of high pass filtering
and H'(u, v) is the Fourier Transform of low pass filtering
```

3. **Band pass filter:** Band pass filter removes the very low frequency and very high frequency components that means it keeps the moderate range band of frequencies. Band pass filtering is used to enhance edges while reducing the noise at the same time.

Frequency Domain Filtering on an Image:

Frequency Domain Filters are used for smoothing and sharpening of images by removal of high or low-frequency components.

Frequency domain filters are different from spatial domain filters as it mainly focuses on the frequency of the images. It is done for two basic operations i.e., Smoothing and Sharpening[21].

1- Apply frequency filters in each case and compare the original image and the image after using the filter:

(a) Apply frequency spectrum with Fast Fourier Transformation (FFT), centered spectrum, decentralized spectrum and inverse FFT on image5 and show them.

(b) Apply Ideal Low Pass Filter (ILPF) and Ideal High Pass Filter (IHPF) on image5 and show them.

(c) Apply Gaussian Smoothing with 3, 5, 7 and 9 kernel size on image6 and show them.

Fast Fourier transform: in image processing, the most common way to represent pixel location is in the spatial domain by column (x), row (y), and z (value). But sometimes image processing routines may be slow or inefficient in the spatial domain, requiring a transformation to a different domain that offers compression benefits.

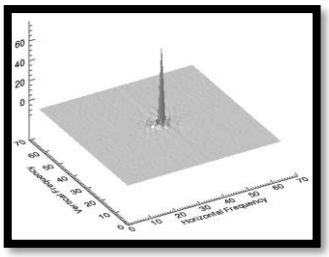
A common transformation is from the spatial to the frequency (or Fourier) domain. The frequency domain is the basis for many image filters used to remove noise, sharpen an image, analyze repeating patterns, or extract features. In the frequency domain, pixel location is represented by its x- and y-frequencies and its value is represented by amplitude.

Exercise 1-Digital Image Processing (DIP)

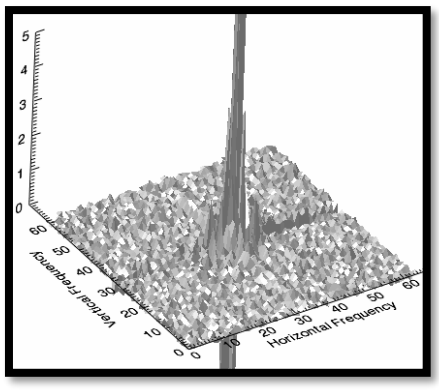
The Fast Fourier Transform (FFT) is commonly used to transform an image between the spatial and frequency domain. Unlike other domains such as Hough and Radon, the FFT method preserves all original data. Plus, FFT fully transforms images into the frequency domain, unlike time-frequency or wavelet transforms. The FFT decomposes an image into sines and cosines of varying amplitudes and phases, which reveals repeating patterns within the image[22].

Low frequencies represent gradual variations in the image; they contain the most information because they determine the overall shape or pattern in the image. High frequencies correspond to abrupt variations in the image; they provide more detail in the image, but they contain more noise. One way to filter out background noise is to apply a mask. See Use FFT to Reduce Background Noise for an example.

When using a forward FFT to transform an image from the spatial to frequency domain, the lowest frequencies are often shown by a large peak in the center of the data. The following image shows an example. Here, the result of an FFT function was plotted as a surface, with the origin (0,0) of the x- and y-frequencies shifted to the center. Frequency of magnitude then increases with distance from the origin[23]:



The range of values from the low-frequency peak to the high-frequency noise is extreme, which reveals that the image has some background noise. For a more detailed view, you can create another surface plot of the power spectrum while rescaling the z-axis to a smaller range. The following image shows an example. Here, the z-axis ranges from 0 to 5:



Exercise 1-Digital Image Processing (DIP)

A surface representation of the power spectrum helps to determine the threshold needed to remove the noise from the image. Then you could create a mask to filter out the noise and compute an inverse FFT to produce a clearer image. See Use FFT to Reduce Background Noise for an example of this entire process[24].

Fourier Transform is used to analyze the frequency characteristics of various filters. For images, 2D Discrete Fourier Transform (DFT) is used to find the frequency domain. A fast algorithm called Fast Fourier Transform (FFT) is used for calculation of DFT. Details about these can be found in any image processing or signal processing textbooks. Please see Additional Resources section.

For a sinusoidal signal, $x(t) = A\sin(2\pi ft)$, we can say f is the frequency of signal, and if its frequency domain is taken, we can see a spike at f. If signal is sampled to form a discrete signal, we get the same frequency domain, but is periodic in the range $[-\pi, \pi]$ or $[0, 2\pi]$ (or $[0, N]$ for N-point DFT). You can consider an image as a signal which is sampled in two directions. So taking fourier transform in both X and Y directions gives you the frequency representation of image.

More intuitively, for the sinusoidal signal, if the amplitude varies so fast in short time, you can say it is a high frequency signal. If it varies slowly, it is a low frequency signal. You can extend the same idea to images. Where does the amplitude varies drastically in images ? At the edge points, or noises. So we can say, edges and noises are high frequency contents in an image. If there is no much changes in amplitude, it is a low frequency component.

Fourier Transform in OpenCV:

Digital images are now part of our daily life. People can hardly live without it. Therefore, digital image processing becomes more and more important these days. Fourier Transformation can help us out. We can utilize Fourier Transformation to transform our image information - gray scaled pixels into frequencies and do further process.

Exercise 1-Digital Image Processing (DIP)

Today, I'll talk about how to utilize Fast Fourier Transformation in digital image processing, and how to implement it in Python.

1. Implement Fast Fourier Transformation to transform gray scaled image into frequency
2. Visualize and Centralize zero-frequency component
3. Apply low/high pass filter to filter frequencies
4. Decentralize
5. Implement inverse Fast Fourier Transformation to generate image data

- **Fast Fourier Transformation:**

Digital images, unlike light wave and sound wave in real life, are discrete because pixels are not continuous. That means we should implement Discrete Fourier Transformation (DFT) instead of Fourier Transformation. However, DFT process is often too slow to be practical. That is the reason why I chose Fast Fourier Transformation (FFT) to do the digital image processing.

Step 1: Compute the 2-dimensional Fast Fourier Transform.

The result from FFT process is a complex number array which is very difficult to visualize directly. Therefore, we have to transform it into 2-dimension space. Here are two ways that we can visualize this FFT result:

1. Spectrum

From *Figure above* there are some symmetric patterns on the four corners. These patterns can be translated to the center of the image in the next step.

The white area in the spectrum image show the high power of frequency. The corners in the spectrum image represent low frequencies. Therefore, combining two points above, the white area on the corner indicates that there is high energy in low/zero frequencies which is a very normal situation for most images.

On the other side, it is hard to identify any noticeable patterns from *phase angle*. This did not indicate that the phase angle of FFT is totally useless because the phase preserves the shape characteristics which is an indispensable information for an image.

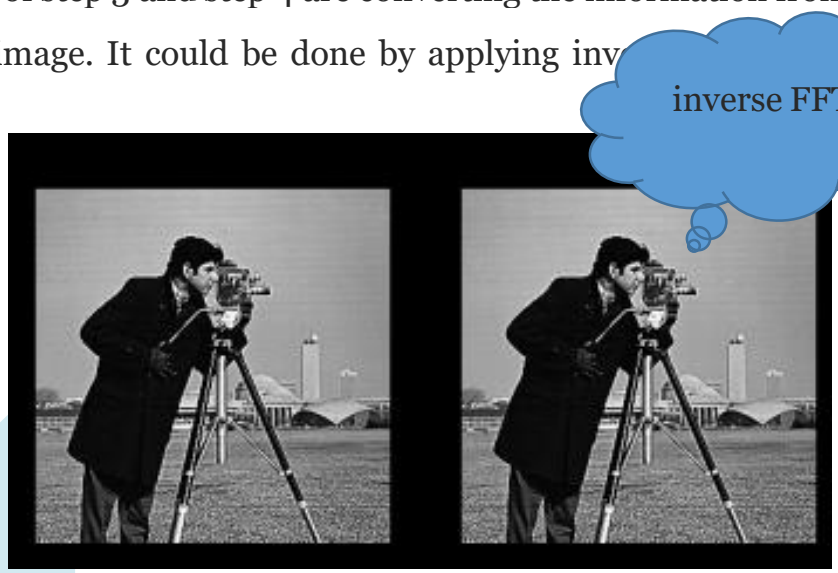
Step 2: Shift the zero-frequency component to the center of the spectrum.

2-D FFT has translation and rotation properties, so we can shift frequency without losing any piece of information. I shifted the zero-frequency component to the center of the spectrum which makes the spectrum image more visible for human. Moreover, this translation could help us implement high/low-pass filter easily.

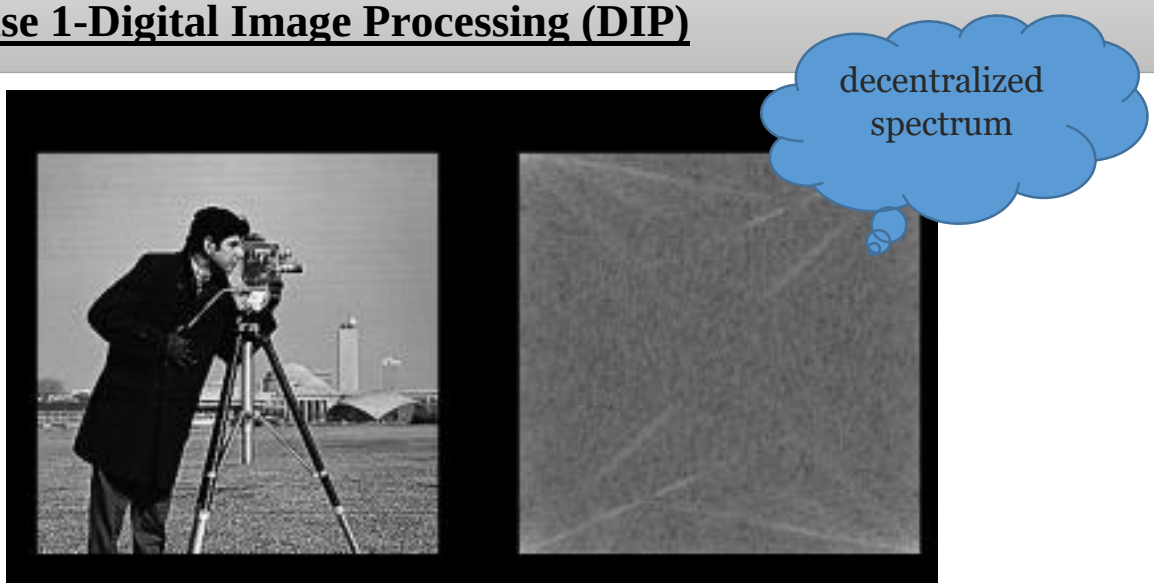
Step 3: Inverse of Step 2. Shift the zero-frequency component back to original location

Step 4: Inverse of Step 1. Compute the 2-dimensional inverse Fast Fourier Transform.

The processes of step 3 and step 4 are converting the information from spectrum back to gray scale image. It could be done by applying inv rse FFT operation.

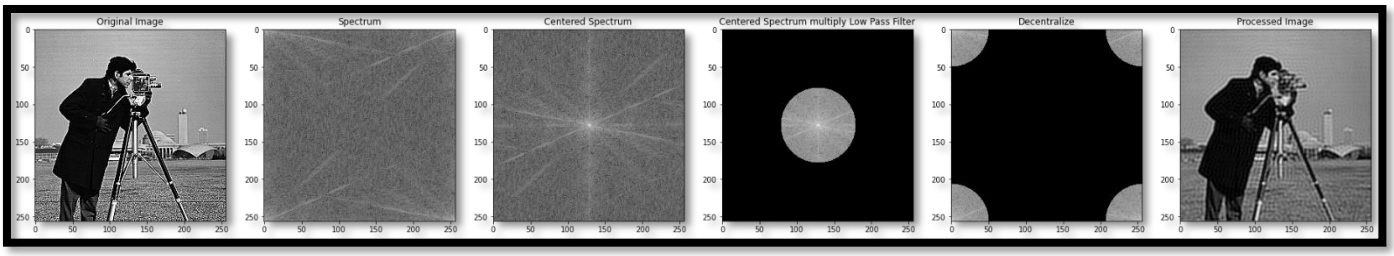


Exercise 1-Digital Image Processing (DIP)



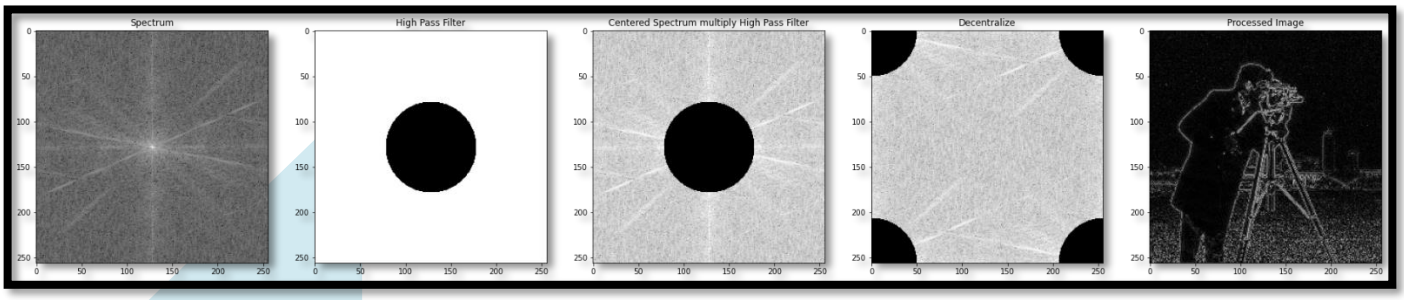
After understanding the basic theory behind Fourier Transformation, it is time to figure out how to manipulate spectrum output to process images. First, we need to understand the low/high pass filter.

- **Low Pass Filter**



Low pass filter is a filter that only allow low frequencies to pass through. Low frequencies in images mean pixel values that are changing slowly. For example, smooth area with slightly color changing in the image such as the center of new blank white paper is considered as a low frequency content. Since the output of low pass filter only allow low frequencies to pass through, the high frequencies contents such as noises are blocked which make processed image has less noisy pixels. Therefore, low pass filter is highly used to remove the noises in images.

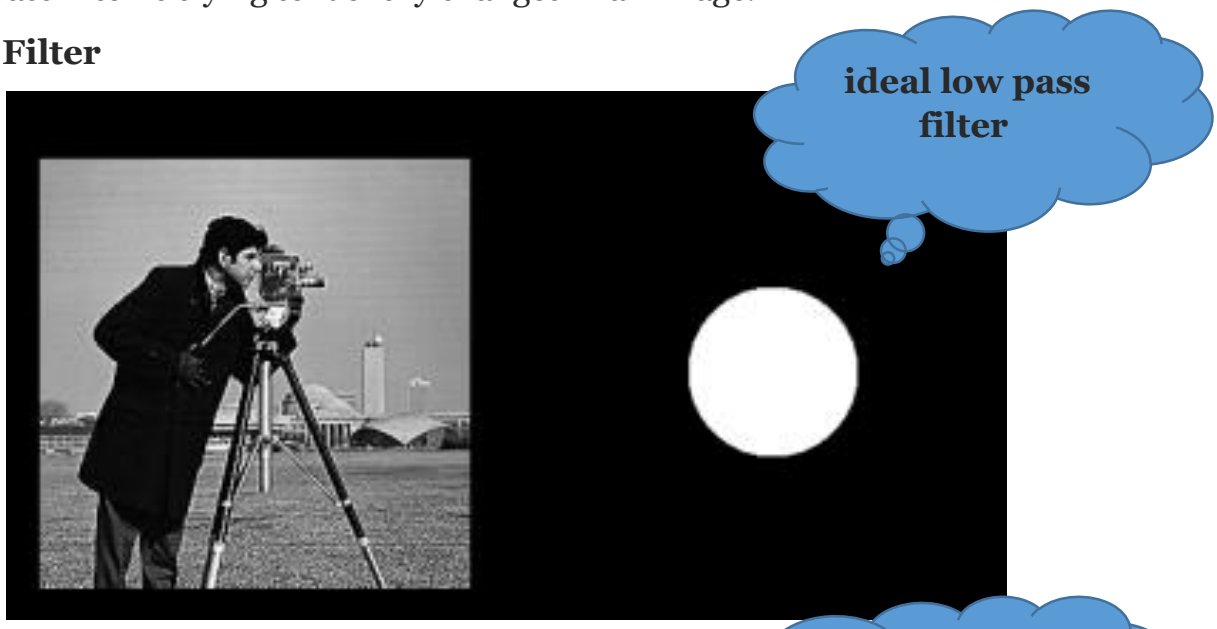
- **High Pass Filter**

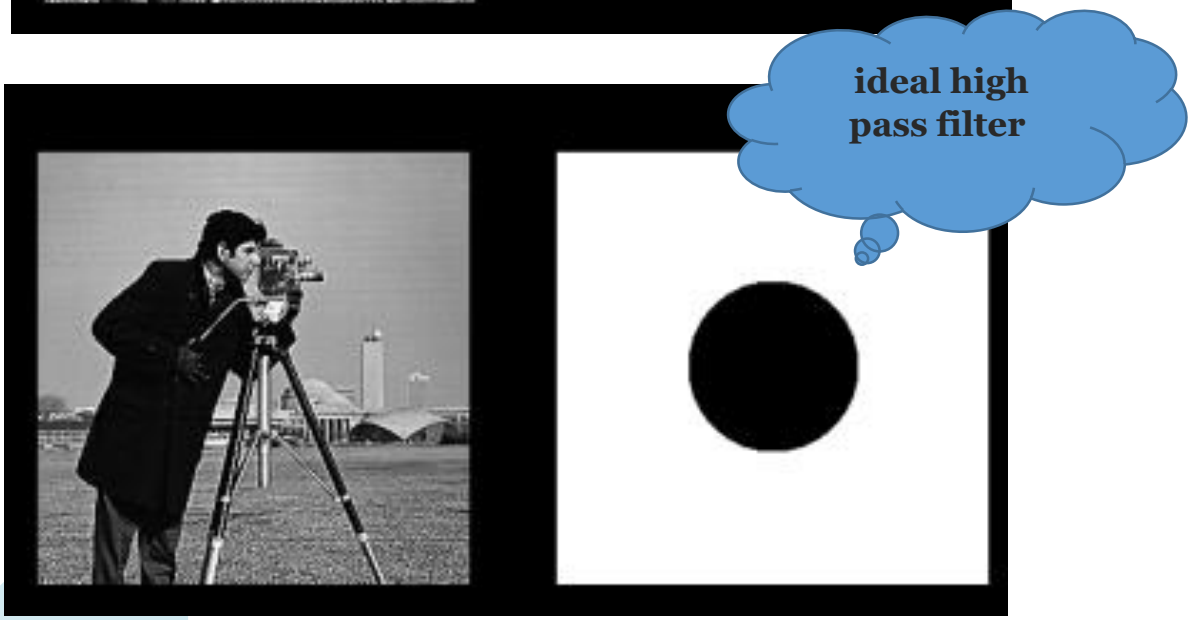


High Pass filter, on the contrary, is a filter that only allow high frequencies to pass through. High frequencies in images mean pixel values that are changing dramatically. For example, Edge areas in the image with huge color changing such as the edge between two overlap white and black paper is consider as the high frequency content. The output from high pass filter captures the edges in image which could be used to sharpen the original image with proper overlap calculation. This will enhance sharpness in original image making edges more clear.

- we could notice that these two filters present different characteristics. Low pass filter tends to preserve overall information in an image. On the other hand, high pass filter is trying to identify changes in an image.

- **Ideal Filter**





Formula for **ideal low pass filter** where D*o* is a positive constant and D(u, v) is the distance between a point (u, v) in the frequency domain and the center of the frequency rectangle

$$H(u,v) = \begin{cases} 1 & \text{if } D(u,v) \le D_0 \\ 0 & \text{if } D(u,v) > D_0 \end{cases}$$

Formula for **ideal high pass filter** where D*o* is a positive constant and D(u, v) is the distance between a point (u, v) in the frequency domain and the center of the frequency rectangle

$$H(u,v) = \begin{cases} 0 & \text{if } D(u,v) \le D_0 \\ 1 & \text{if } D(u,v) > D_0 \end{cases}$$

In [18]:
```
def image5():
    """
    Edit image5.jpg.

    Returns
    -------
    None.

    """
    img = cv2.imread("D:/image processing/images/image5.jpg", 0)
    shifted_transformed_img, magnitude_spectrum = dft(img)
    plot_before_after(img, magnitude_spectrum, True)
    filtered_img = idft(shifted_transformed_img)
    plot_before_after(img, filtered_img)
    inv_center = np.fft.ifftshift(magnitude_spectrum)
    plot_before_after(img, inv_center)
    LowPass = idealFilterLP(50,img.shape)
    plot_before_after(img, LowPass)
    HighPass = idealFilterHP(50,img.shape)
    plot_before_after(img, HighPass)
```

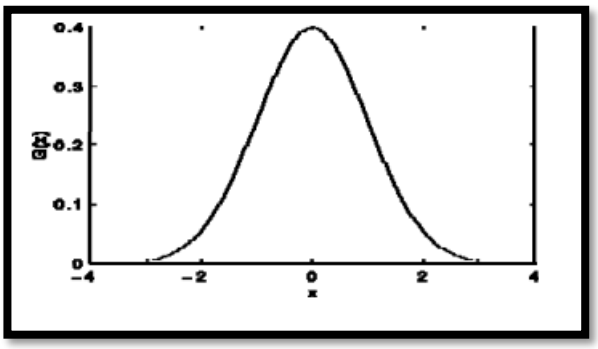
- **Gaussian Smoothing**

Gaussian filtering g is used to blur images and remove noise and detail. In one dimension, the Gaussian function is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Where σ is the standard deviation of the distribution. The distribution is assumed to have a mean of 0. Shown graphically, we see the familiar bell shaped Gaussian distribution.

Gaussian distribution with mean 0 and σ = 1



• Significant values

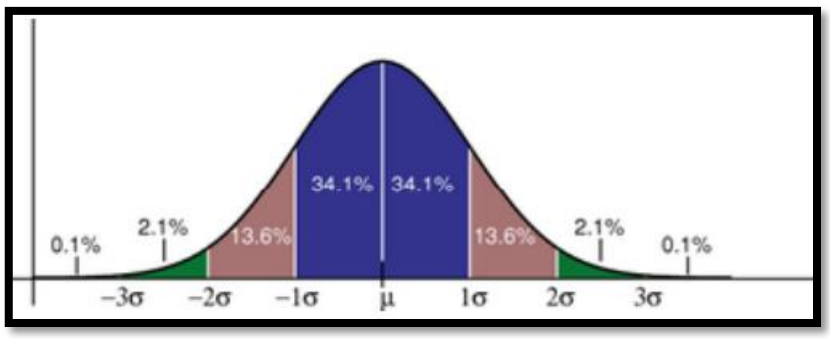
x	0	1	2	3	4
$\sigma * G(x) / 0.399$	1	$e^{-0.5/\sigma^2}$	e^{-2/σ^2}	$e^{-9/4\sigma^2}$	e^{-8/σ^2}
$G(x) / G(0)$	1	$e^{-0.5/\sigma^2}$	e^{-2/σ^2}	$e^{-9/4\sigma^2}$	e^{-8/σ^2}

For σ=1:

x	0	1	2
$G(x)$	0.399	0.242	0.05
$G(x) / G(0)$	1	0.6	0.125

Standard Deviation

The Standard deviation of the Gaussian function plays an important role in its behavior. The values located between +/- σ account for 68% of the set, while two standard deviations from the mean (blue and brown) account for 95%, and three standard deviations (blue, brown and green) account for 99.7%. account for 99.7%. This is very important when designing a Gaussian kernel of fixed length.



The Gaussian function is used in numerous research areas: – It defines a probability distribution for noise or data.

– It is a smoothing operator.

– It is used in mathematics.

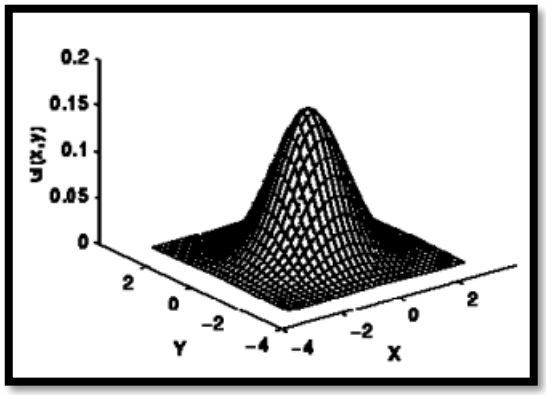
The Gaussian function has important properties which are verified with The Gaussian function has important properties which are verified with respect to its integral:

In probabilistic terms, it des $I = \int_{-\infty}^{\infty} \exp\left(-x^2\right) dx = \sqrt{\pi}$ lues of any given space when varying from negative to positive values, Gauss function is never equal to zero, It is a symmetric function.

When working with images we need to use the two dimensional Gaussian function, this is simply the product of two 1D Gaussian functions (one for each direction) and is given by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

A graphical representation of the 2D Gaussian distribution with mean(0,0) and σ = 1 is shown to the right.



The Gaussian filter works by using the 2D distribution as a point-spread function. This is achieved by convolving the 2D Gaussian distribution function with the image. We need to produce a discrete approximation to the Gaussian function. This theoretically requires an infinitely large convolution kernel, as the Gaussian distribution is non-zero everywhere. Fortunately, the distribution has approached very close to zero at about three standard deviations from the mean. 99% of the distribution falls within 3 standard deviations. This means we can normally limit the kernel size to contain only

values 23 This means we can normally limit the kernel size to contain only values within three standard deviations of the mean.

Gaussian kernel coefficients are sampled from the 2D Gaussian function; the distribution is assumed to have a mean of zero. We need to discretize the continuous Gaussian functions to store it as discrete pixels. An integer valued 5 by 5 convolution kernel approximating a Gaussian with a σ of 1 is shown to the right,

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

In [19]:
```
def image6():
    """
    Edit image6.jpg.

    Returns
    -------
    None.

    """

    img = cv2.imread("D:/image processing/images/image6.jpg")
    img_gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
    img_g = img_gray.copy()
    for k in (3, 5, 7, 9):
        size = 2*k+1
        kernel = gausskernel(size,k,1.5)
        #print(kernel)
        img_B,img_G,img_R = cv2.split(img)
        img_gauss_B = mygaussFilter(img_B,kernel)
        img_gauss_G = mygaussFilter(img_G,kernel)
        img_gauss_R = mygaussFilter(img_R,kernel)
        img_gauss = cv2.merge([img_gauss_B,img_gauss_G,img_gauss_R])
        #img_comp = np.hstack((img,img_gauss))
        plot_before_after(img, img_gauss)
```

Exercise 1-Digital Image Processing (DIP)

Gaussian Smoothing with kernel size=3

Gaussian Smoothing with kernel size=5

Gaussian Smoothing with kernel size=7

Gaussian Smoothing with 9kernel size=

Exercise 1-Digital Image Processing (DIP)

5- Remove the noise of images 1-2-3-4 with proper filters.

- **Image noise:**

Image noise is random variation of brightness or color information in images, and is usually an aspect of electronic noise. It can be produced by the image sensor and circuitry of a scanner or digital camera. Image noise can also originate in film grain and in the unavoidable shot noise of an ideal photon detector. Image noise is an undesirable by-product of image capture that obscures the desired information.

The original meaning of "noise" was "unwanted signal"; unwanted electrical fluctuations in signals received by AM radios caused audible acoustic noise ("static"), by analogy, unwanted electrical fluctuations are also called "noise"[25].

Image noise can range from almost imperceptible specks on a digital photograph taken in good light, to optical and radio astronomical images that are almost entirely noise, from which a small amount of information can be derived by sophisticated processing. Such a noise level would be unacceptable in a photograph since it would be impossible even to determine the subject.

- **Types of noises:**

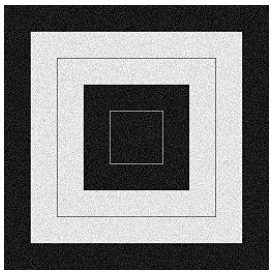
1. **Gaussian noise**

Principal sources of Gaussian noise in digital images arise during acquisition. The sensor has inherent noise due to the level of illumination and its own temperature, and the electronic circuits connected to the sensor inject their own share of electronic circuit noise[26].

A typical model of image noise is Gaussian, additive, independent at each pixel, and independent of the signal intensity, caused primarily by Johnson–Nyquist noise (thermal noise), including that which comes from the reset noise of capacitors ("kTC noise"). Amplifier noise is a major part of the "read noise" of an image sensor, that is, of the constant noise level in dark areas of the images, In color cameras where more amplification is used in the blue color channel than in the green or red channel, there can be more noise in the blue channel. At

higher exposures, however, image sensor noise is dominated by shot noise, which is not Gaussian and not independent of signal intensity. Also, there are many Gaussian denoising algorithms[27].

With Gaussian noise



2. **Salt-and-pepper noise**

Fat-tail distributed or "impulsive" noise is sometimes called salt-and-pepper noise or spike noise, An image containing salt-and-pepper noise will have dark pixels in bright regions and bright pixels in dark regions. This type of noise can be caused by analog-to-digital converter errors, bit errors in transmission, etc.[10][11] It can be mostly eliminated by using dark frame subtraction, median filtering, combined median and mean filtering and interpolating around dark/bright pixels, Dead pixels in an LCD monitor produce a similar, but non-random, display[28].

Image with salt and pepper noise

3. **Shot noise**

The dominant noise in the brighter parts of an image from an image sensor is typically that caused by statistical quantum fluctuations, that is, variation in the number of photons sensed at a given exposure level. This noise is known as photon shot noise. Shot noise has a root-mean-square value proportional to the square root of the image intensity, and the noises at different pixels are independent of one another. Shot noise follows a Poisson distribution, which except at very high intensity levels approximates a Gaussian distribution[29].

In addition to photon shot noise, there can be additional shot noise from the dark leakage current in the image sensor; this noise is sometimes known as "dark shot noise" or "dark-current shot noise". Dark current is greatest at "hot pixels" within the image sensor. The variable dark charge of normal and hot pixels can be subtracted off (using "dark frame subtraction"), leaving only the shot noise, or random component, of the leakage. If dark-frame subtraction is not done, or if the exposure time is long enough that the hot pixel charge exceeds the linear charge capacity, the noise will be more than just shot noise, and hot pixels appear as salt-and-pepper noise[30].

4. **Quantization noise (uniform noise)**

The noise caused by quantizing the pixels of a sensed image to a number of discrete levels is known as quantization noise. It has an approximately uniform distribution. Though it can be signal dependent, it will be signal independent if other noise sources are big enough to cause dithering, or if dithering is explicitly applied.[31]

5. **Periodic noise**

A common source of periodic noise in an image is from electrical or electromechanical interference during the image capturing process.[8] An image affected by periodic noise will look like a repeating pattern has been added on top of the original image. In the frequency domain this type of noise can be seen as discrete spikes. Significant reduction of this noise can be achieved by applying notch filters in the frequency domain. The following images illustrate an image affected by periodic noise, and the result of reducing the noise using

frequency domain filtering. Note that the filtered image still has some noise on the borders. Further filtering could reduce this border noise, however it may also reduce some of the fine details in the image. The trade-off between noise reduction and preserving fine details is application specific. For example, if the fine details on the castle are not considered important, low pass filtering could be an appropriate option. If the fine details of the castle are considered important, a viable solution may be to crop off the border of the image entirely[32].

- **Noise reduction:**

  An image is a picture, photograph or any other form of 2D representation of any scene. Most algorithms for converting image sensor data to an image, whether in-camera or on a computer, involve some form of noise reduction. There are many procedures for this, but all attempt to determine whether the actual differences in pixel values constitute noise or real photographic detail, and average out the former while attempting to preserve the latter. However, no algorithm can make this judgment perfectly (for all cases), so there is often a tradeoff made between noise removal and preservation of fine, low-contrast detail that may have characteristics similar to noise.

  A simplified example of the impossibility of unambiguous noise reduction: an area of uniform red in an image might have a very small black part. If this is a single pixel, it is likely (but not certain) to be spurious and noise; if it covers a few pixels in an absolutely regular shape, it may be a defect in a group of pixels in the image-taking sensor (spurious and unwanted, but not strictly noise); if it is irregular, it may be more likely to be a true feature of the image. But a definitive answer is not available.

  This decision can be assisted by knowing the characteristics of the source image and of human vision. Most noise reduction algorithms perform much more aggressive chroma noise reduction, since there is little important fine chroma detail that one risks losing. Furthermore, many people find luminance noise less

objectionable to the eye, since its textured appearance mimics the appearance of film grain.

The high sensitivity image quality of a given camera (or RAW development workflow) may depend greatly on the quality of the algorithm used for noise reduction. Since noise levels increase as ISO sensitivity is increased, most camera manufacturers increase the noise reduction aggressiveness automatically at higher sensitivities. This leads to a breakdown of image quality at higher sensitivities in two ways: noise levels increase and fine detail is smoothed out by the more aggressive noise reduction.

In cases of extreme noise, such as astronomical images of very distant objects, it is not so much a matter of noise reduction as of extracting a little information buried in a lot of noise; techniques are different, seeking small regularities in massively random data[33].

Noise of images 1:

A notch filter (also known as a bandstop filter or reject filter) is defined as a device that rejects or blocks the transmission of frequencies within a specific frequency range and allows frequencies outside that range. Notch filters eliminate transmission of a narrow band of frequencies and allow transmission of all the frequencies above and below this band. As it eliminates frequencies hence, it is also called a band elimination filter.

A notch filter is essentially a band stop filter with a narrow stopband and two passbands. As in the band-pass case, a band-reject filter can be either wideband or narrow-band.

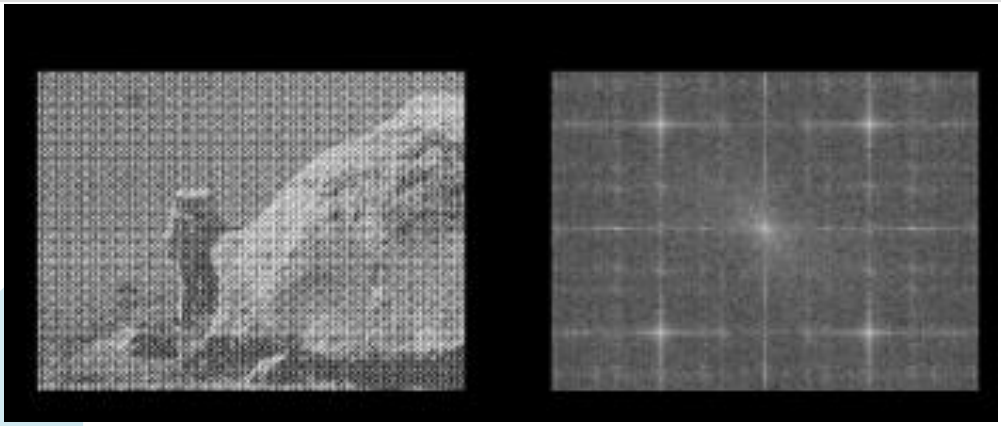
uses notch filtering to reduce the moire ripple in the digital printed image.

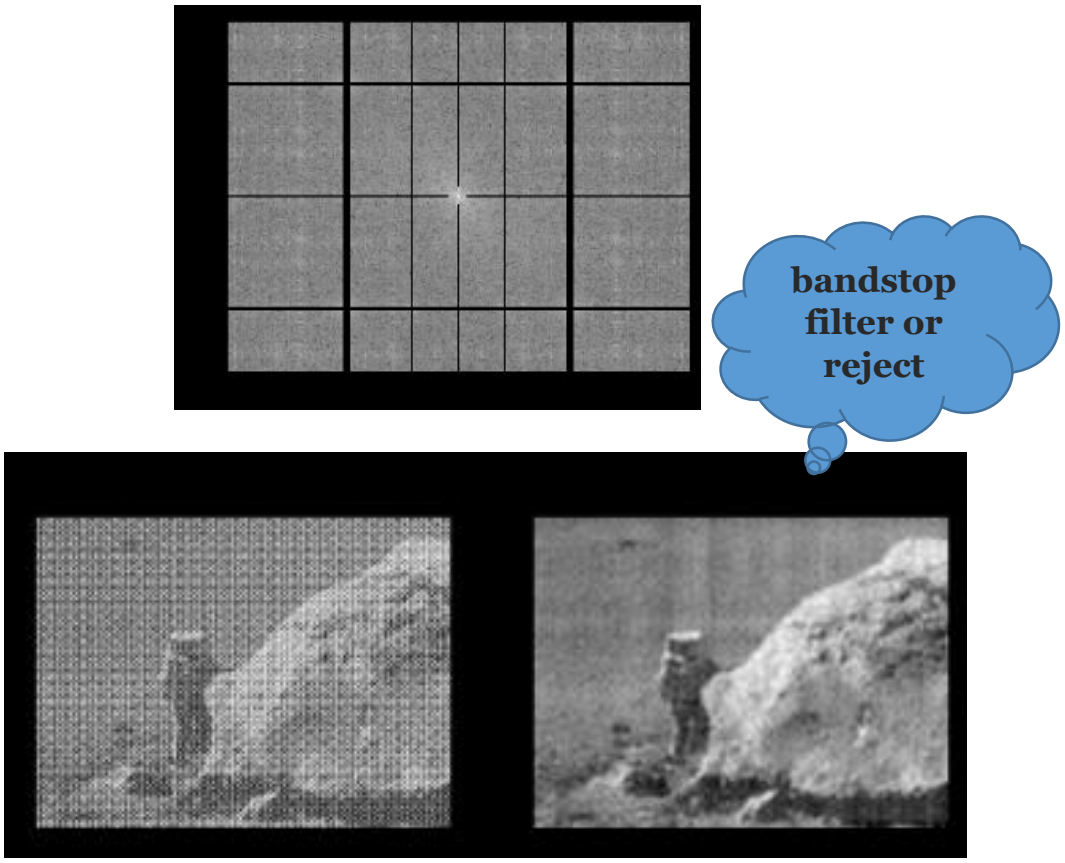
Moire pattern is the expression of beat difference principle. After the superposition of two equal amplitude sine waves with similar frequencies, the signal amplitude will change according to the difference between the two frequencies. If the spatial frequency of the pixel of the photosensitive element is close to the spatial frequency of the stripe in the image, the moire will be generated.

Exercise 1-Digital Image Processing (DIP)

Notch filtering is to selectively modify the local region of DFT. The typical processing method is interactive operation. Directly use the mouse to select the rectangular region in the Fourier spectrum and find the maximum point as (uk,vk). In order to simplify the program, this routine deletes the mouse interaction part and only retains the filtering process after (uk,vk)[34].

In [20]:
```
def im1():
    """
    Edit image 1.jpg.

    Returns
    -------
    None.

    """
    img = cv2.imread('D:/image processing/images/1.jpg', 0)
    shifted_transformed_img, magnitude_spectrum = dft(img)

    plot_before_after(img, magnitude_spectrum, True)

    w, h = img.shape
    mask = np.ones(img.shape, dtype=np.uint8)

    for i in range(w):
        if np.mean(magnitude_spectrum[i, :]) >= 9.:
            # magnitude_spectrum[i, :] = 0
            mask[i, :] = 0

    for j in range(h):
        if np.mean(magnitude_spectrum[:, j]) >= 9.:
            # magnitude_spectrum[:, j] = 0
            mask[:, j] = 0
    mask[w // 2 - 5:w // 2 + 5, h // 2 - 5:h // 2 + 5] = 1

    plt.imshow(mask * magnitude_spectrum, cmap='gray')
    plt.show()

    filtered_img = idft(mask * shifted_transformed_img)
    filtered_img = cv2.GaussianBlur(filtered_img, (5, 5), 0)

    plot_before_after(img, filtered_img)
```





Noise of images 2:

Removal of horizontal stripes using openCV2:

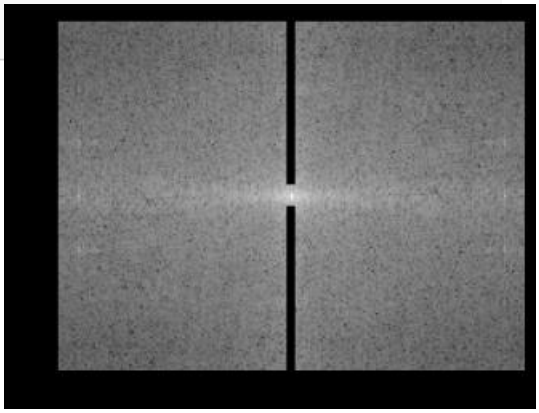
Here is how to mitigate (reduce, but not totally eliminate) the lines using Fourier Transform and notch filtering processing with Python/OpenCV/Numpy. Since the horizontal lines in the input are very close, there will be horizontal linear structures spaced far apart in the Fourier Transform spectrum. So what I did was:

- ✓ Read input as grayscale
- ✓ Get min and max values of image.
- ✓ Convert image to floats and do Discrete Fourier Transform (DFT) saving as complex output
- ✓ Apply shift of origin from upper left corner to center of image.
- ✓ Extract magnitude and phase images.
- ✓ Get spectrum
- ✓ Create mask from spectrum keeping only the brightest spots as the notches.
- ✓ Dilate mask

Exercise 1-Digital Image Processing (DIP)

- ✓ Cover center DC component by circle of black leaving only a few white spots on black background
- ✓ Apply mask to magnitude such that magnitude is made zero where mask is one, ie at spots.
- ✓ Convert new magnitude and old phase into cartesian real and imaginary components.
- ✓ Combine cartesian components into one complex image
- ✓ Shift origin from center to upper left corner.
- ✓ Do Inverse Discrete Fourier Transform (iDFT) saving as complex output.
- ✓ Combine complex components into original image again.
- ✓ Re-normalize to 8-bits in range of original.

In [21]:
```
def im2():
    """
    Edit image 2.jpg.

    Returns
    -------
    None.

    """
    img = cv2.imread("D:/image processing/images/2.jpg", 0)
    shifted_transformed_img, magnitude_spectrum = dft(img)

    w, h = img.shape
    mask = np.ones(img.shape)
    mask[0:w // 2 - 5, h // 2 - 2:h // 2 + 2] = 0
    mask[w // 2 + 5:w, h // 2 - 2:h // 2 + 2] = 0

    plt.imshow(magnitude_spectrum * mask, cmap='gray')
    plt.show()

    masked_fourier = shifted_transformed_img * mask
    filtered_img = idft(masked_fourier)
    filtered_img = cv2.blur(filtered_img, (2, 2))

    plot_before_after(img, filtered_img)
```

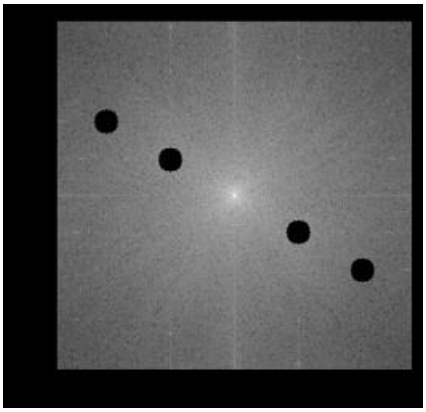


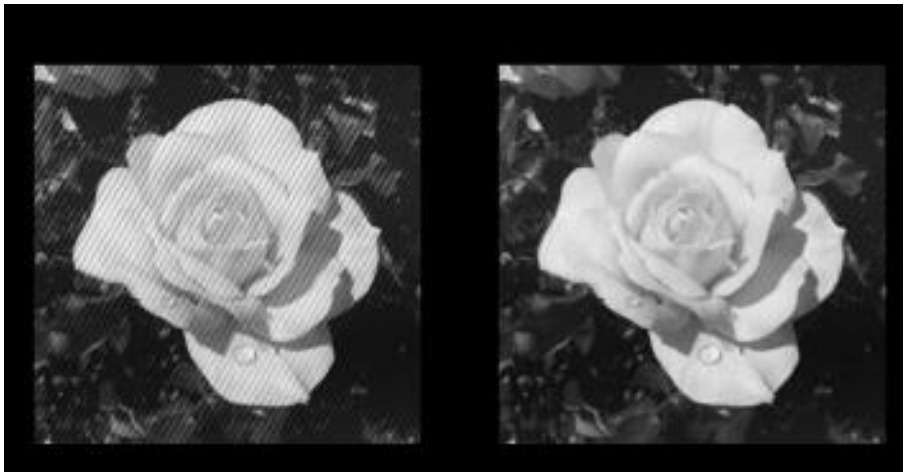
Noise of images 3:

Removal of stripes using openCV2:

- ✓ Read input.
- ✓ Do dft saving as complex output.
- ✓ Apply shift of origin to center of image.
- ✓ Generate spectrum from magnitude image (for viewing only)
- ✓ Create circle mask
- ✓ Blur the mask
- ✓ Apply mask to dft_shift
- ✓ Shift origin from center to upper left corner.
- ✓ Do idft saving as complex output.
- ✓ Combine complex real and imaginary components to form (the magnitude for) the original image again.

In [22]:
```
def im3():
    """
    Edit image 3.jpg.

    Returns
    -------
    None.

    """
    img = cv2.imread("D:/image processing/images/3.jpg", 0)
    shifted_transformed_img, magnitude_spectrum = dft(img)
    w, h = img.shape
    mask = np.ones((img.shape[0], img.shape[1], 3), dtype=np.uint8)

    cv2.circle(mask, (96, 117), 10, (0, 0, 0), -1)
    cv2.circle(mask, (42, 85), 10, (0, 0, 0), -1)
    cv2.circle(mask, (204, 178), 10, (0, 0, 0), -1)
    cv2.circle(mask, (258, 210), 10, (0, 0, 0), -1)

    mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)

    plt.imshow(magnitude_spectrum * mask, cmap='gray')
    plt.show()
    masked_fourier = shifted_transformed_img * mask
    filtered_img = idft(masked_fourier)
    plot_before_after(img, filtered_img)
```





Noise of images 4: **(Remove the noise of images 4.ipynb)**

Using Highpass Filtering and Threshold for Image Enhancement:

A high-pass filter (HPF) is an electronic filter that passes signals with a frequency higher than a certain cutoff frequency and attenuates signals with frequencies lower than the cutoff frequency. The amount of attenuation for each frequency depends on the filter design. A high-pass filter is usually modeled as a linear time-invariant system. It is sometimes called a low-cut filter or bass-cut filter in the context of audio engineering. High-pass filters have many uses, such as blocking DC from circuitry sensitive to non-zero average voltages or radio frequency devices. They can also be used in conjunction with a low-pass filter to produce a bandpass filter.

In the optical domain filters are often characterised by wavelength rather than frequency. High-pass and low-pass have the opposite meanings, with a "high-pass" filter (more commonly "long-pass") passing only longer wavelengths (lower frequencies), and vice versa for "low-pass" (more commonly "short-pass")[35].

Exercise 1-Digital Image Processing (DIP)

Threshold:

In digital image processing, thresholding is the simplest method of segmenting images. From a grayscale image, thresholding can be used to create binary images, To make thresholding completely automated, it is necessary for the computer to automatically select the threshold T. Sezgin and Sankur (2004) categorize thresholding methods into the following six groups based on the information the algorithm manipulates:

- Histogram shape-based methods, where, for example, the peaks, valleys and curvatures of the smoothed histogram are analyzed
- Clustering-based methods, where the gray-level samples are clustered in two parts as background and foreground (object), or alternately are modeled as a mixture of two Gaussians
- Entropy-based methods result in algorithms that use the entropy of the foreground and background regions, the cross-entropy between the original and binarized image, etc.[2]
- Object Attribute-based methods search a measure of similarity between the gray-level and the binarized images, such as fuzzy shape similarity, edge coincidence, etc.
- Spatial methods [that] use higher-order probability distribution and/or correlation between pixels
- Local methods adapt the threshold value on each pixel to the local image characteristics. In these methods, a different T is selected for each pixel in the image. The T can be of many types like mean, gaussian, median, mode(not used generally)[36].

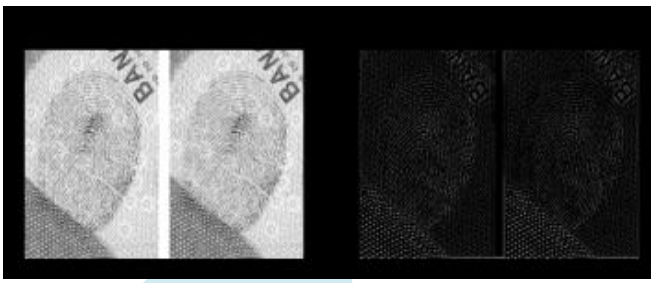
Removal of noises using openCV2:

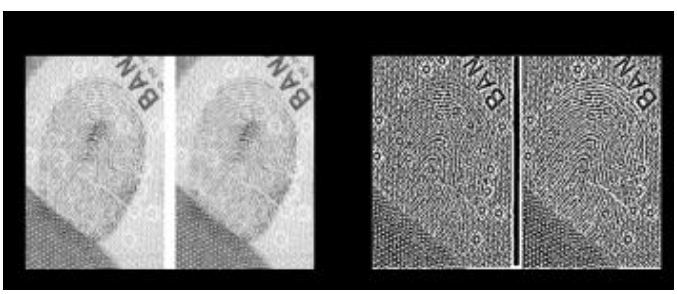
- ✓ Gaussian Highpass Filter
- ✓ Optimal extended fast Fourier transform
- ✓ Optimal DFT expansion size
- ✓ Edge extension (complement 0), fast Fourier transform

Exercise 1-Digital Image Processing (DIP)

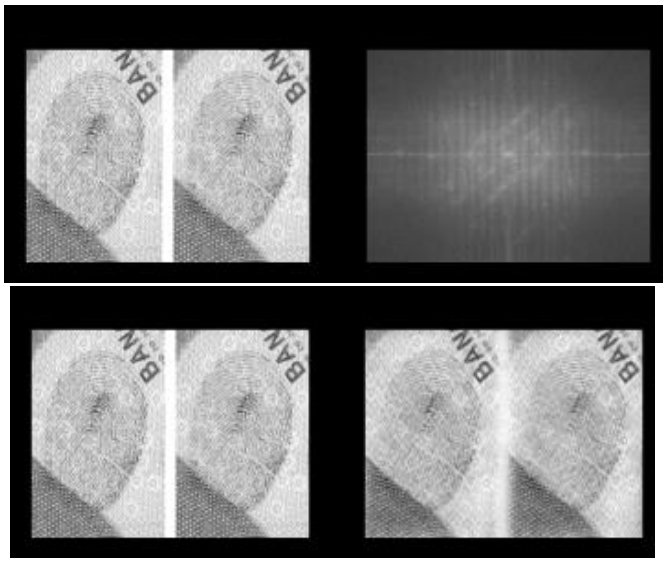
- ✓ fast Fourier transform
- ✓ Generate spectrum from magnitude image
- ✓ Construct Gaussian low pass filter
- ✓ Modify Fourier transform in frequency domain: Fourier transform point multiplication high pass filter
- ✓ The inverse Fourier transform is performed on the high pass Fourier transform and only the real part is taken
- ✓ Centralized 2D array g (x, y) * - 1 ^ (x + y)
- ✓ Intercept the upper left corner, the size is equal to the input image

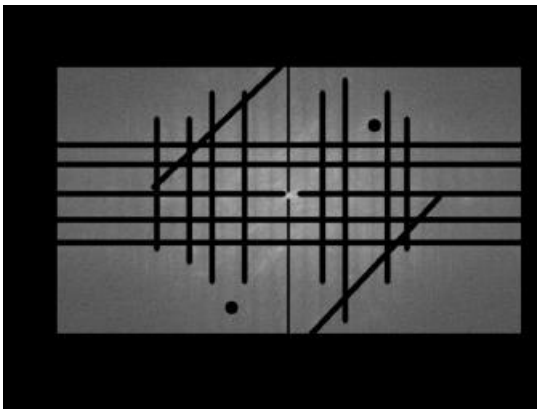
In [23]:
```
def im4():
    """  Edit image 4.jpg. Returns    -------    None.    """
    img = cv2.imread("D:/image processing/images/4.jpg", 0)
    rows, cols = img.shape[:2]  # The height and width of the picture
    imgHPF = imgHPFilter(img, D0=50)
    imgThres = np.clip(imgHPF, 0, 1)
    plot_before_after(img, imgHPF)
    plot_before_after(img, imgThres)
    shifted_transformed_img, magnitude_spectrum = dft(img)
    plot_before_after(img, magnitude_spectrum, True)
    w, h = img.shape
    mask = np.ones((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    cv2.line(mask, (155, 150), (155, 350), (0, 0, 0), 8)
    cv2.line(mask, (205, 150), (205, 370), (0, 0, 0), 8)
    cv2.line(mask, (240, 110), (240, 400), (0, 0, 0), 8)
    cv2.line(mask, (0, 265), (350, 265), (0, 0, 0), 8)
    cv2.line(mask, (375, 265), (h, 265), (0, 0, 0), 8)
    cv2.line(mask, (290, 110), (290, 400), (0, 0, 0), 8)
    cv2.line(mask, (410, 110), (410, 400), (0, 0, 0), 8)
    cv2.line(mask, (510, 110), (510, 400), (0, 0, 0), 8)
    cv2.line(mask, (345, 70), (150, 255), (0, 0, 0), 8)
    cv2.line(mask, (370, 505), (590, 270), (0, 0, 0), 8)
    cv2.line(mask, (445, 90), (445, 460), (0, 0, 0), 8)
    cv2.line(mask, (540, 150), (540, 350), (0, 0, 0), 8)
    cv2.line(mask, (0, 190), (h, 190), (0, 0, 0), 8)
    cv2.line(mask, (0, 340), (h, 340), (0, 0, 0), 8)
    cv2.line(mask, (0, 305), (h, 305), (0, 0, 0), 8)
    cv2.line(mask, (0, 220), (h, 220), (0, 0, 0), 8)
    cv2.circle(mask, (270, 440), 10, (0, 0, 0), -1)
    cv2.circle(mask, (490, 160), 10, (0, 0, 0), -1)
    cv2.rectangle(mask, (0, 0), (h, 70), (0, 0, 0), -1)
    cv2.rectangle(mask, (0, 480), (h, w), (0, 0, 0), -1)
    mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
    mask[0:w // 2 - 5, h // 2 - 2:h // 2 + 2] = 0
    mask[w // 2 + 5:w, h // 2 - 2:h // 2 + 2] = 0
    plt.imshow(20 * magnitude_spectrum * mask, cmap='gray')
    plt.show()
    masked_fourier = shifted_transformed_img * mask
    filtered_img = idft(masked_fourier)
    plot_before_after(img, filtered_img)
```



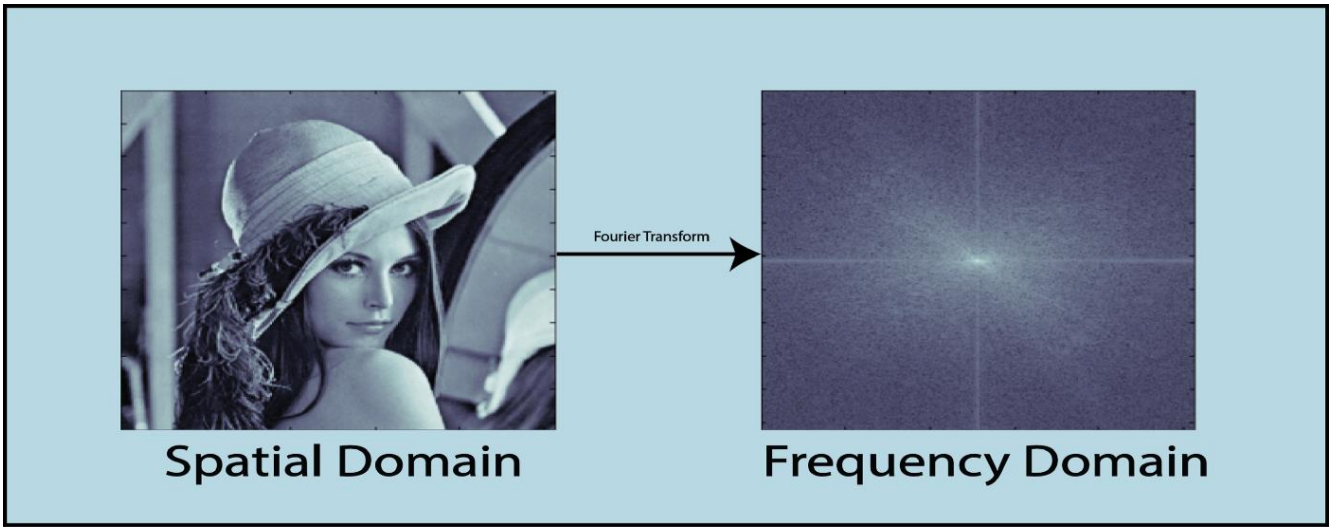


Exercise 1-Digital Image Processing (DIP)



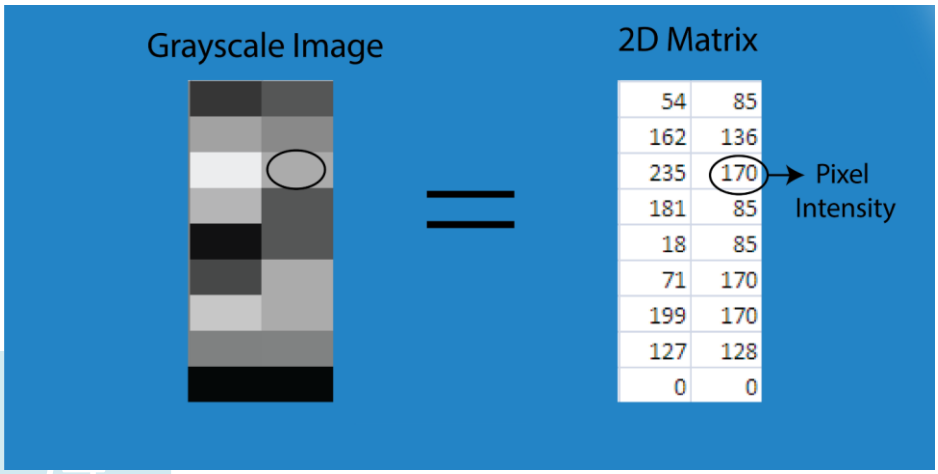


Spatial and Frequency Domain:

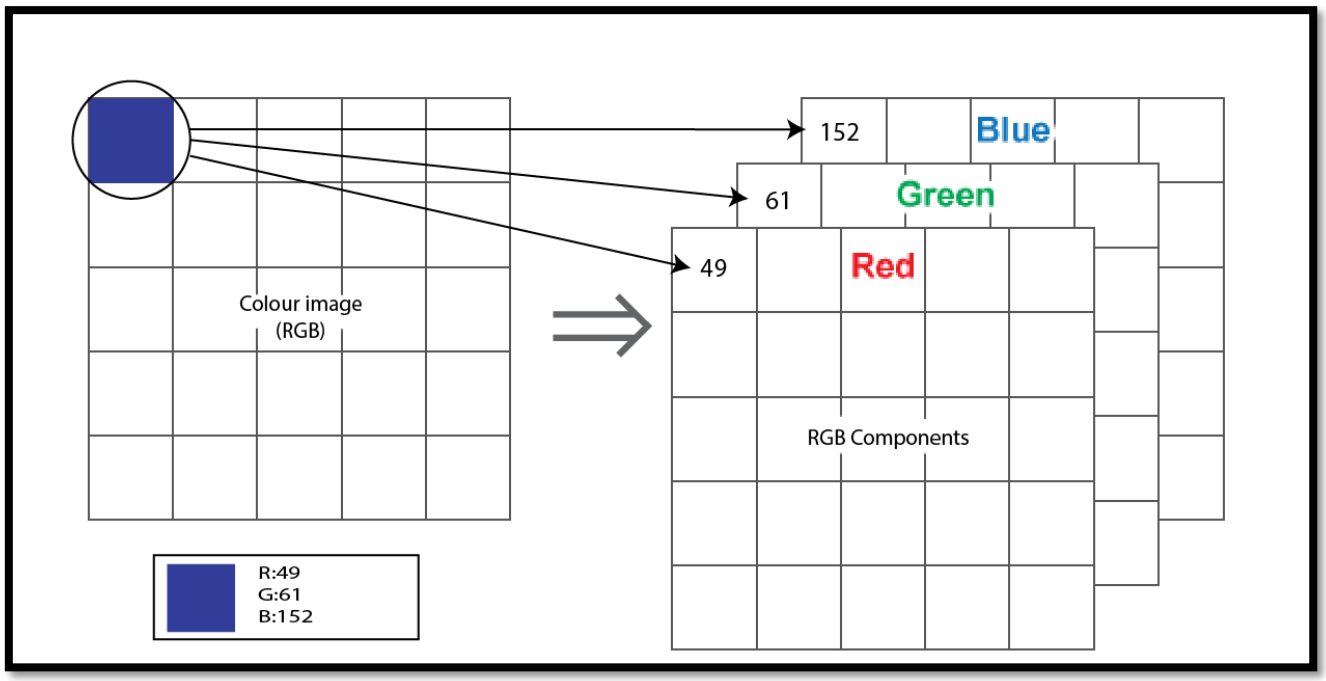


Spatial Domain

An image can be represented in the form of a 2D matrix where each element of the matrix represents pixel intensity. This state of 2D matrices that depict the intensity distribution of an image is called Spatial Domain. It can be represented as shown below:



For the RGB image, the spatial domain is represented as a 3D vector of 2D matrices. Each 2D matrix contains the intensities for a single color as shown below



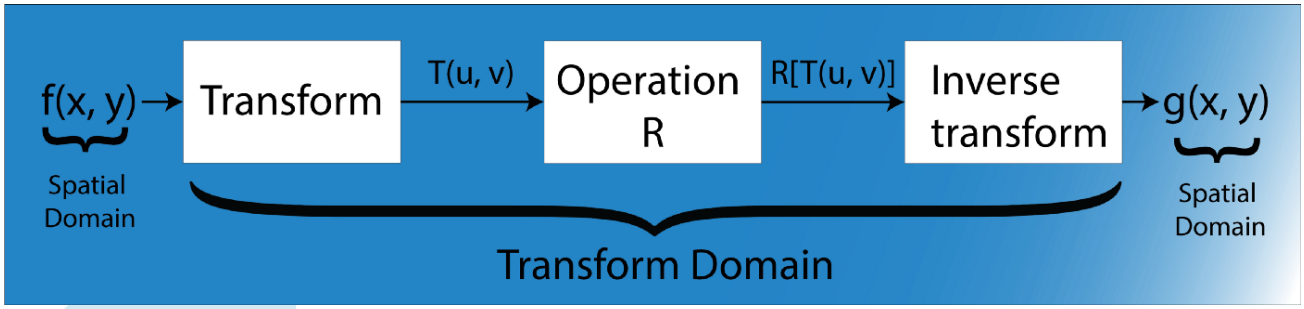
Spatial domain for color image(RGB)

Each pixel intensity is represented as I(x,y) where x,y is the co-ordinate of the pixel in the 2D matrix. Different operations are carried out in this value. For example- operation T(say, addition of 5 to all the pixel) is carried out in I(x,y) which means that each pixel value is increased by 5. This can be written as: I'(x,y) = T[I(x,y)], where, I'(x,y) is the new intensity after adding 5 to I(x,y).

Frequency Domain

In frequency-domain methods are based on Fourier Transform of an image. Roughly, the term frequency in an image tells about the rate of change of pixel values.

Image Transformation mainly follows three steps:



- ✓ **Step-1.** Transform the image.
- ✓ **Step-2.** Carry the task(s) in the transformed domain.
- ✓ **Step-3.** Apply inverse transform to return to the spatial domain.

6- Improve the details and visibility of images 5 to 10 with whatever you know from spatial to frequency domain operations.

Improve the details and visibility of images using openCV2:

- ✓ Gaussian Highpass Filter
- ✓ Optimal extended fast Fourier transform
- ✓ Optimal DFT expansion size
- ✓ Edge extension (complement 0), fast Fourier transform
- ✓ High frequency filtering + histogram equalization
- ✓ fast Fourier transform
- ✓ Construct Gaussian low pass filter
- ✓ Modify Fourier transform in frequency domain: Fourier transform point multiplication low-pass filter
- ✓ The inverse Fourier transform is performed on the high pass Fourier transform and only the real part is taken
- ✓ Centralized 2D array g (x, y) * - 1 ^ (x + y)
- ✓ Intercept the upper left corner, the size is equal to the input image
- ✓ Modify Fourier transform in frequency domain: Fourier transform point multiplication low-pass filter
- ✓ The inverse Fourier transform is performed on the high pass Fourier transform and only the real part is taken
- ✓ Intercept the upper left corner, the size is equal to the input image

Exercise 1-Digital Image Processing (DIP)

Exercise 1-Digital Image Processing (DIP)

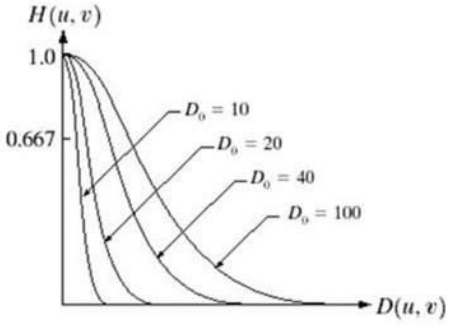
Exercise 1-Digital Image Processing (DIP)

In [24]:
```
def im5678910():
    """
    Edit image 5, 6, 7, 8, 9, 10.jpg. Returns -------None.    """
    # High frequency filtering + histogram equalization
    path= "D:/image processing/images/23/*.jpg"
    image_number=1
    for file in glob.glob(path):
        print (file)
        image =cv2.imread(file, 0)
        rows, cols = image.shape[:2]  # The height and width of the picture
        print(rows, cols)
        # fast Fourier transform
        dftImage = dft2Image(image)  # Fast Fourier transform (rPad, cPad, 2)
        rPadded, cPadded = dftImage.shape[:2]  # Fast Fourier transform size, original image size optimization
        # Construct Gaussian low pass filter
        hpFilter = gaussHighPassFilter((rPadded, cPadded), radius=40)  # Gaussian Highpass Filte
        # Modify Fourier transform in frequency domain: Fourier transform point multiplication low-pass filter
        dftHPfilter = np.zeros(dftImage.shape, dftImage.dtype)  # Size of fast Fourier transform (optimized size)
        for j in range(2):
            dftHPfilter[:rPadded, :cPadded, j] = dftImage[:rPadded, :cPadded, j] * hpFilter
            # The inverse Fourier transform is performed on the high pass Fourier transform and only the real part is taken
            idft = np.zeros(dftImage.shape[:2], np.float32)  # Size of fast Fourier transform (optimized size)
            cv2.dft(dftHPfilter, idft, cv2.DFT_REAL_OUTPUT + cv2.DFT_INVERSE + cv2.DFT_SCALE)
        # Centralized 2D array g (x, y) * - 1 ^ (x + y)
        mask2 = np.ones(dftImage.shape[:2])
        mask2[1::2, ::2] = -1
        mask2[::2, 1::2] = -1
        idftCen = idft * mask2  # g(x,y) * (-1)^(x+y)
        # Intercept the upper left corner, the size is equal to the input image
        result = np.clip(idftCen, 0, 255)  # Truncation function, limiting the value to [0255]
        imgHPF = result.astype(np.uint8)
        imgHPF = imgHPF[:rows, :cols]
        # # =======High frequency enhanced filtering==================
        k1 = 0.5
        k2 = 0.75
        # Modify Fourier transform in frequency domain: Fourier transform point multiplication low-pass filter
        hpEnhance = np.zeros(dftImage.shape, dftImage.dtype)  # Size of fast Fourier transform (optimized size)
        for j in range(2):
            hpEnhance[:rPadded, :cPadded, j] = dftImage[:rPadded, :cPadded, j] * (k1 + k2*hpFilter)
            # The inverse Fourier transform is performed on the high pass Fourier transform and only the real part is taken
            idft = np.zeros(dftImage.shape[:2], np.float32)  # Size of fast Fourier transform (optimized size)
            cv2.dft(hpEnhance, idft, cv2.DFT_REAL_OUTPUT + cv2.DFT_INVERSE + cv2.DFT_SCALE)
        # Centralized 2D array g (x, y) * - 1 ^ (x + y)
        mask2 = np.ones(dftImage.shape[:2])
        mask2[1::2, ::2] = -1
        mask2[::2, 1::2] = -1
        idftCen = idft * mask2  # g(x,y) * (-1)^(x+y)
        # Intercept the upper left corner, the size is equal to the input image
        result = np.clip(idftCen, 0, 255)  # Truncation function, limiting the value to [0255]
        imgHPE= result.astype(np.uint8)
        imgHPE = imgHPE[:rows, :cols]
        # =======Histogram equalization===================
        imgEqu = cv2.equalizeHist(imgHPE)  # Use CV2 Equalizehist completes histogram equalization transformation
        plot_before_after(image, imgHPE)
        plot_before_after(image, imgEqu)
    image_number +=1
```

7- Make image 11 and 12 younger with frequency domain operator:

Gaussian Low pass filter:

The concept of filtering and low pass remains the same, but only the transition becomes different and become more smooth. The Gaussian low pass filter can be represented as:



Note the smooth curve transition, due to which at each point, the value of Do, can be exactly defined.

Gaussian low-pass filtering is a common post-process operation which is exploited to blur and conceal these discontinuities at the border of tampered objects introduced by copy & paste operation, making the tampered image more realistic.

Removal of noises using openCV2:

- ✓ Gaussian low pass filter
- ✓ Optimal extended fast Fourier transform
- ✓ Optimal DFT expansion size
- ✓ Edge extension (complement 0), fast Fourier transform
- ✓ Construct Gaussian low pass filter
- ✓ Modify Fourier transform in frequency domain: Fourier transform point multiplication low-pass filter
- ✓ The inverse Fourier transform is performed on the low-pass Fourier transform, and only the real part is taken
- ✓ Centralized 2D array g (x, y) * - 1 ^ (x + y)
- ✓ Intercept the upper left corner, the size is equal to the input image

Exercise 1-Digital Image Processing (DIP)

In [25]:
```
def im1112():
    """
    Edit image 12.jpg. Returns   -------  None.    """
        # (1) Read original image
    path= "D:/image processing/images/24/*.jpg"
    image_number=1
    for file in glob.glob(path):
        print (file)
        imgGray =cv2.imread(file, 0)
        rows, cols = imgGray.shape[:2]  # The height and width of the picture
        # (2) Fast Fourier transform
        dftImage = dft2Image(imgGray)  # Fast Fourier transform (rPad, cPad, 2)
        rPadded, cPadded = dftImage.shape[:2]  # Fast Fourier transform size, original image size optimization
        print("dftImage.shape:{}".format(dftImage.shape))
        D0 = [10, 30, 60, 80, 100]  # radius
        for k in range(5):
            # (3) Construct Gaussian low pass filter
            lpFilter = gaussLowPassFilter((rPadded, cPadded), radius=D0[k])

            # (5) Modify Fourier transform in frequency domain: Fourier transform point multiplication low-pass filter
            dftLPfilter = np.zeros(dftImage.shape, dftImage.dtype)  # Size of fast Fourier transform (optimized size)
            for j in range(2):
                dftLPfilter[:rPadded, :cPadded, j] = dftImage[:rPadded, :cPadded, j] * lpFilter

            # (6) The inverse Fourier transform is performed on the low-pass Fourier transform, and only the real part is taken
            idft = np.zeros(dftImage.shape[:2], np.float32)  # Size of fast Fourier transform (optimized size)
            cv2.dft(dftLPfilter, idft, cv2.DFT_REAL_OUTPUT + cv2.DFT_INVERSE + cv2.DFT_SCALE)

            # (7) Centralized 2D array g (x, y) * - 1 ^ (x + y)
            mask2 = np.ones(dftImage.shape[:2])
            mask2[1::2, ::2] = -1
            mask2[::2, 1::2] = -1
            idftCen = idft * mask2  # g(x,y) * (-1)^(x+y)

            # (8) Intercept the upper left corner, the size is equal to the input image
            result = np.clip(idftCen, 0, 255)  # Truncation function, limiting the value to [0255]
            imgLPF = result.astype(np.uint8)
            imgLPF = imgLPF[:rows, :cols]
            plot_before_after(imgGray, imgLPF)

    image_number +=1
```

References:

[1] H. Johansson, "Sampling and quantization," in *Academic Press Library in Signal Processing*, vol. 1, Elsevier, 2014, pp. 169–244.

[2] R. L. Easton Jr, *Digital Image Processing I*. September, 2010.

[3] S. Gollapudi, *Learn computer vision using OpenCV*. Springer, 2019.

[4] S. Gollapudi, "OpenCV with Python," in *Learn Computer Vision Using OpenCV*, Springer, 2019, pp. 31–50.

[5] J. Howse, *OpenCV computer vision with python*. Packt Publishing Birmingham, 2013.

[6] M. Beyeler, *OpenCV with Python blueprints*. Packt Publishing Ltd, 2015.

[7] T. Y. Kong and A. Rosenfeld, *Topological algorithms for digital image processing*. Elsevier, 1996.

[8] J. F. Blinn, "What is a pixel?," *IEEE Comput. Graph. Appl.*, vol. 25, no. 5, pp. 82–87, 2005.

[9] P. Heckbert, "Color image quantization for frame buffer display," *ACM Siggraph Comput. Graph.*, vol. 16, no. 3, pp. 297–307, 1982.

[10] P. Q. Le, A. M. Iliyasu, F. Dong, and K. Hirota, "Fast Geometric Transformations on Quantum Images.," *IAENG Int. J. Appl. Math.*, vol. 40, no. 3, 2010.

[11] H. S. Prashanth, H. L. Shashidhara, and K. N. B. Murthy, "Image scaling comparison using universal image quality index," in *2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies*, 2009, pp. 859–863.

[12] G. B. García, O. D. Suarez, J. L. E. Aranda, J. S. Tercero, I. S. Gracia, and N. V. Enano, *Learning image processing with OpenCV*. Packt Publishing Birmingham, 2015.

[13] A. Adeli and X. Emery, "Geostatistical simulation of rock physical and geochemical properties with spatial filtering and its application to predictive geological mapping," *J. Geochemical Explor.*, vol. 220, p. 106661, 2021.

[14] W. Gander and U. von Matt, "Smoothing filters," in *Solving problems in scientific computing using maple and Matlab®*, Springer, 1995, pp. 121–139.

[15] P. J. Bex, G. K. Edgar, and A. T. Smith, "Sharpening of drifting, blurred images," *Vision Res.*, vol. 35, no. 18, pp. 2539–2546, 1995.

[16] K. Dawson-Howe, *A practical introduction to computer vision with opencv*. John Wiley & Sons, 2014.

[17] G. Xie and W. Lu, "Image edge detection based on opencv," *Int. J. Electron. Electr. Eng.*, vol. 1, no. 2, pp. 104–106, 2013.

[18] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to OpenCV," in *2012 proceedings of the 35th international convention MIPRO*, 2012, pp. 1725–1730.

[19] D. Matveev, "OpenCV Graph API," *Intel Corp.*, vol. 1, 2018.

[20] B. D. V. Reddy and D. T. J. Prasad, "Frequency domain filtering of colour images using

quaternion Fourier transforms," *IJCST*, vol. 1, no. 2, pp. 46–52, 2010.

[21] S. Brahmbhatt, *Practical OpenCV*. Apress, 2013.

[22] E. W. Weisstein, "Fast fourier transform," *https//mathworld. wolfram. com/*, 2015.

[23] E. O. Brigham, *The fast Fourier transform and its applications*. Prentice-Hall, Inc., 1988.

[24] C. Van Loan, *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[25] J. Suler and R. D. Zakia, *Perception and imaging: Photography as a way of seeing*. Routledge, 2017.

[26] J. Nader, Z. A. A. Alqadi, and B. Zahran, "Analysis of Color Image Filtering Methods," *Int. J. Comput. Appl.*, vol. 174, no. 8, pp. 12–17, 2017.

[27] K. Patel and H. Mewada, "A review on different image de-noising methods," *Int. J. Recent Innov. Trends Comput. Commun.*, vol. 2, no. 1, pp. 155–159, 2014.

[28] A. A. Gulhane and A. S. Alvi, "Noise reduction of an image by using function approximation techniques," *Int. J. Soft Comput. Eng.*, vol. 2, no. 1, pp. 60–62, 2012.

[29] H. Y. Alborz, "Image Denoising Analysis over Wireless Networks," *Image (IN).*, vol. 5, no. 01, 2018.

[30] R. E. Jacobson, G. G. Attridge, N. R. Axford, and S. F. Ray, "The Course of Photography of the United Kingdom," *Reed Educ. Prof. Publ. Ltd*, 2000.

[31] M. P. Sampat, M. K. Markey, and A. C. Bovik, "Computer-aided detection and diagnosis in mammography," *Handb. image video Process.*, vol. 2, no. 1, pp. 1195–1217, 2005.

[32] R. C. Gonzalez, *Digital image processing*. Pearson education india, 2009.

[33] N. Chervyakov, P. Lyakhov, and N. Nagornov, "Analysis of the Quantization Noise in Discrete Wavelet Transform Filters for 3D Medical Imaging," *Appl. Sci.*, vol. 10, no. 4, p. 1223, 2020.

[34] M. Mojiri and A. R. Bakhshai, "An adaptive notch filter for frequency estimation of a periodic signal," *IEEE Trans. Automat. Contr.*, vol. 49, no. 2, pp. 314–318, 2004.

[35] C.-G. Bénar, L. Chauvière, F. Bartolomei, and F. Wendling, "Pitfalls of high-pass filtering for detecting epileptic oscillations: a technical note on 'false' ripples," *Clin. Neurophysiol.*, vol. 121, no. 3, pp. 301–310, 2010.

[36] R. C. Gonzalez and R. E. Woods, "Thresholding," *Digit. image Process.*, pp. 595–611, 2002.