IMAGE PROCESSING

INTRODUCTION

Digital images are made of pixels and each pixel contains numerical values that can be interpreted as the color in that part of the image.

Matrix representation of color images depends on the color system used by the program that is processing the image. RGB (the most popular one), where each pixel specifies the amount of Red (R), Green (G) and Blue (B), and each color can vary from 0 to 255. Thus, in the RGB, a pixel can be represented as a tri-dimensional vector (r, g, b) where r, g and b are integer numbers from 0 to 255.Each color matrix is called a 'channel'. Different combination of RGB produce different colors.

There are two main kind of image processing:

1. When the color of every pixel is changed, using a function that gets as input the original pixel, or in more complex cases, a submatrix of pixels (usually submatrices around the pixel in the matrix, depending on an extra factor).
2. When the pixels change their position inside the image, or most precisely, when every pixel in the matrix is build based on another pixel of the matrix, but without altering its color.

Image processing procedures of the first kind are usually called filters. Among the most used there are: adjustment of brightness, contrast and colors, grayscale conversion, color inversion (negative), gamma correction, blur and noise reduction.

SHADES TINTS AND HUES

From the point of view of linear algebra, filters are applied to each pixel of the matrix using the filter function. As explained before, the input of this function can be just a pixel like the adjustment of brightness, or a submatrix of pixels like the blur, where the order of the submatrix will depend on the blur ratio.

Let's consider the matrix M, as the matrix associated to a full color image:

$$M = \begin{bmatrix} p11 & p12 & \cdots & p1n \\ p21 & p22 & \cdots & p2n \\ \vdots & \vdots & \ddots & \vdots \\ pm1 & pm2 & \cdots & pmn \end{bmatrix}$$

Here, p_{ij} is the pixel in the position (i, j), which is represented as the vector:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

In the simplest case (the filter needs only a pixel as input), the function can be a linear transformation, that transforms a tridimensional vector (pixel) into another tridimensional vector, or not.

When it's a linear transformation, the transformation can be represented as a 3x3 matrix T, where:

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = T \bullet \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

CODE:

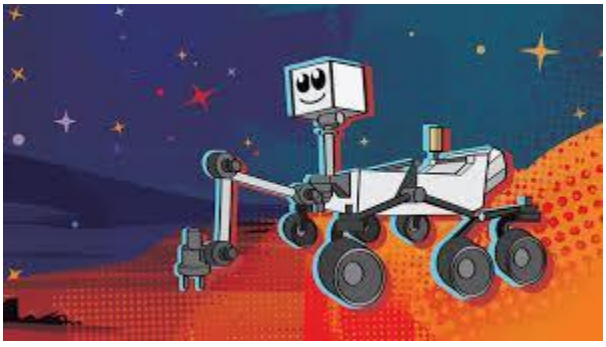
```
from PIL import Image

img=Image.open(r"C:\Users\User\Pictures\SavedPictures\rover.jpg").convert("RGB")
```

```
width,height = img.size
pixels = img.load()
def red(r,g,b):
    newr = r
    newg = 0
    newb = 0
    return(newr,newg,newb)
def darkpink(r,g,b):
    newr = g
    newg = b
    newb = r
    return(newr,newg,newb)
def skyblue(r,g,b):
    newr = b
    newg = g
    newb = r
    return(newr,newg,newb)
def lemongreen(r,g,b):
    newr = g
    newg = r
    newb = b
    return(newr,newg,newb)
def grey(r,g,b):
```

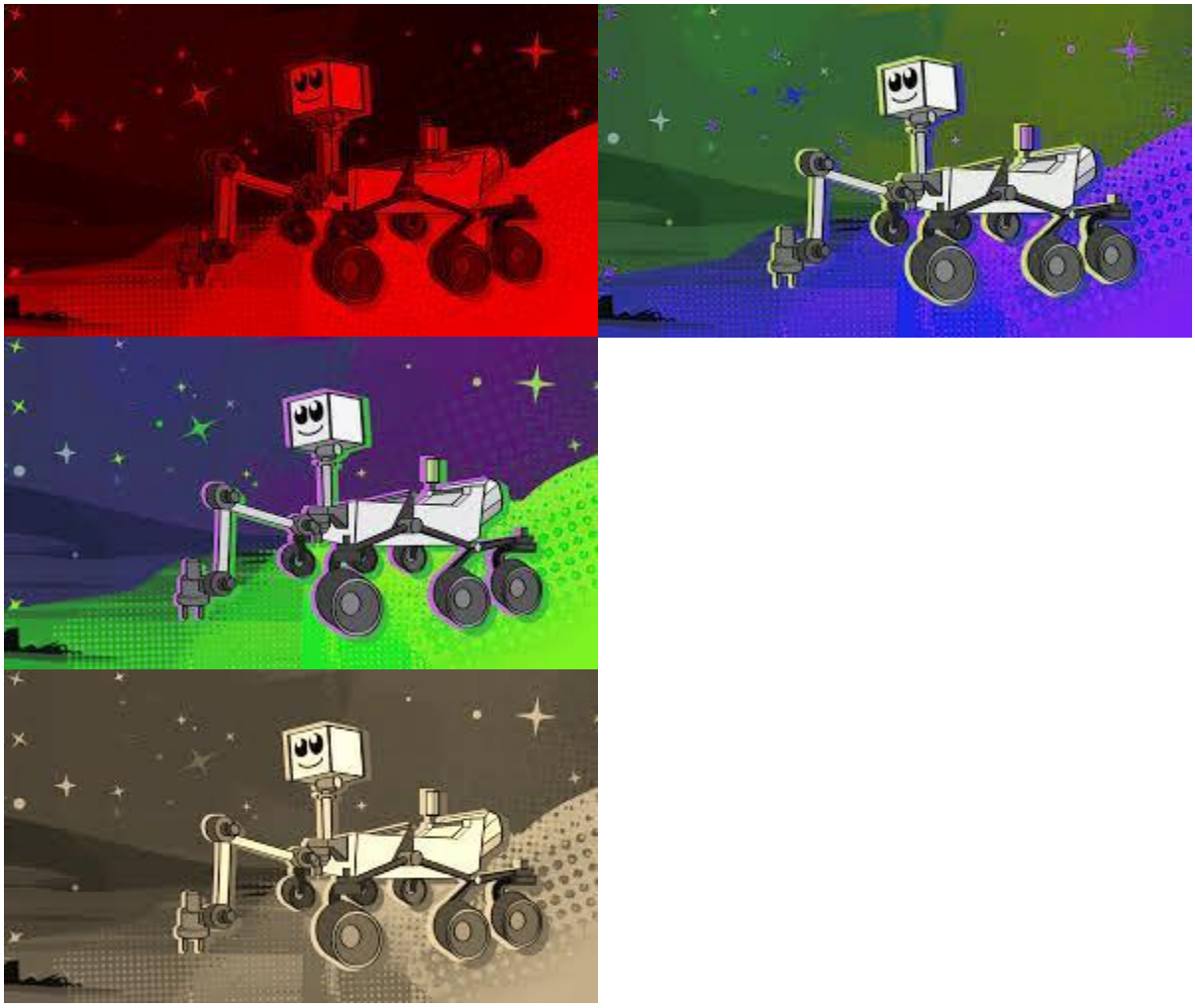
```
    newr = (r+g+b)//3
    newg = (r+g+b)//3
    newb = (r+g+b)//3
    return(newr,newg,newb)
def sepia(r,g,b):
    newr = int((r*.393) + (g*.769) + (b*.189))
    newg = int((r*.349) + (g*.686) + (b*.168))
    newb = int((r*.272) + (g*.534) + (b*.131))
    return(newr,newg,newb)
choice = '''
enter your choice
1 red
2 darkpink
3 skyblue
4 lemonyellow
5 grey
6 sepia
'''
print(choice)
no = int(input())
for py in range(height):
    for px in range(width):
        r,g,b = img.getpixel((px,py))
```

```
    if no==1:
        pixels[px,py] = red(r,g,b)
    if no==2:
        pixels[px,py] = darkpink(r,g,b)
    if no==3:
        pixels[px,py] = skyblue(r,g,b)
    if no==4:
        pixels[px,py] = lemongreen(r,g,b)
    if no==5:
        pixels[px,py] = grey(r,g,b)
    if no==6:
        pixels[px,py] = sepia(r,g,b)
img.show()
img.save(r"C:\Users\User\Pictures\Saved Pictures\newfilterimg.jpg")
```

BEFORE IMAGE:



AFTER IMAGES :



GRAYSCALING

A grayscale (or graylevel) image is simply one in which the only colors are shades of gray. The reason for differentiating such images from any other sort of color image is that less information needs to be provided for each pixel. In fact a `gray' color is one in which the red, green and blue components all have equal intensity in RGB space, and so it is only necessary to specify a single intensity value for each pixel, as opposed to the three intensities needed to specify each pixel in a full color image.

Grayscale images are very common, in part because much of today's display and image capture hardware can only support 8-bit images. In addition, grayscale images are entirely sufficient for many tasks and so there is no need to use more complicated and harder-to-process color images.

Grayscale conversion

$$T = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

The components of each new pixel is obtained by calculating the average of the three components.

CODE:

```
from matplotlib.image import imread

import matplotlib.pyplot as plt

import numpy as np

input_image = imread(r"C:\Users\User\Pictures\Saved Pictures\rover.jpg")

r,g,b = input_image[:,:,0],input_image[:,:,1],input_image[:,:,2]

gamma = 1.04

r_const,g_const,b_const = 0.2126, 0.7152, 0.0722

grayscale_image = r_const * r ** gamma + g_const * g ** gamma + b_const + b **gamma

fig = plt.figure(1)

img1,img2 = fig.add_subplot(121), fig.add_subplot(122)

img1.imshow(input_image)

img2.imshow(grayscale_image, cmap=plt.cm.get_cmap('gray'))

fig.show()

plt.show()
```

BEFORE AND AFTER IMAGE

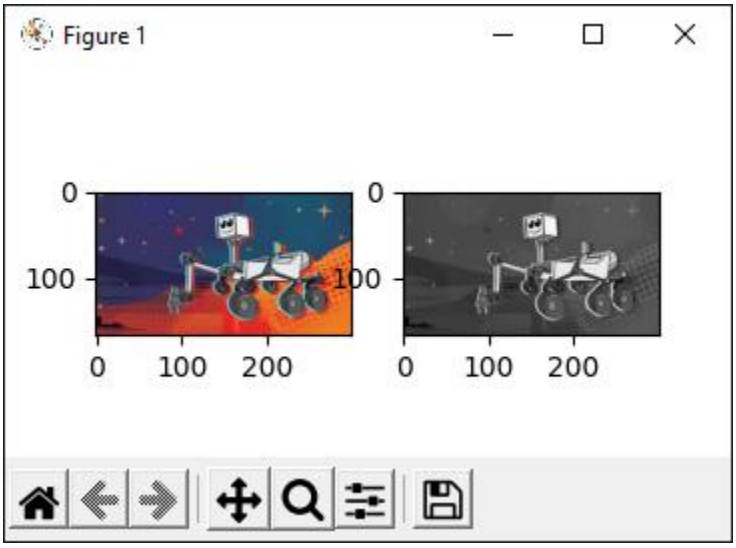


IMAGE RECONSTRUCTION USING SVD

INTRODUCTION

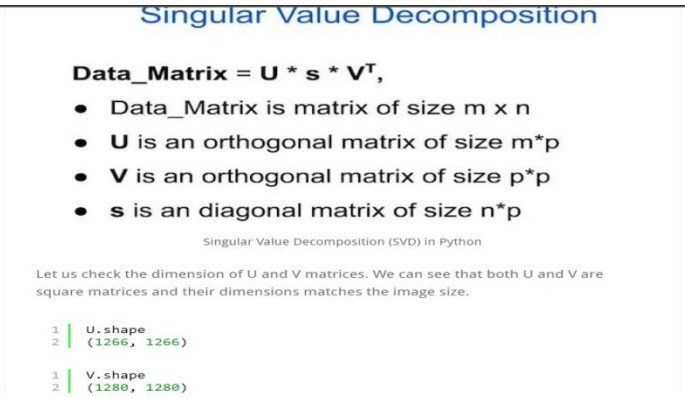
->Image reconstruction is a technique deployed to reconstruct the image approximately towards the original image

->It uses the concept of Singular value decomposition i.e, we detect the top 2 singular vectors and use them for reconstruction

->Greater the singular value ,greater is the resolution of the picture .

->Singular Value Decomposition is one of many matrix decomposition Technique that decomposes a matrix into 3 sub-matrices namely U, S, V.

->We can reconstruct SVD of an image by using **linalg.svd()** method of NumPy module.



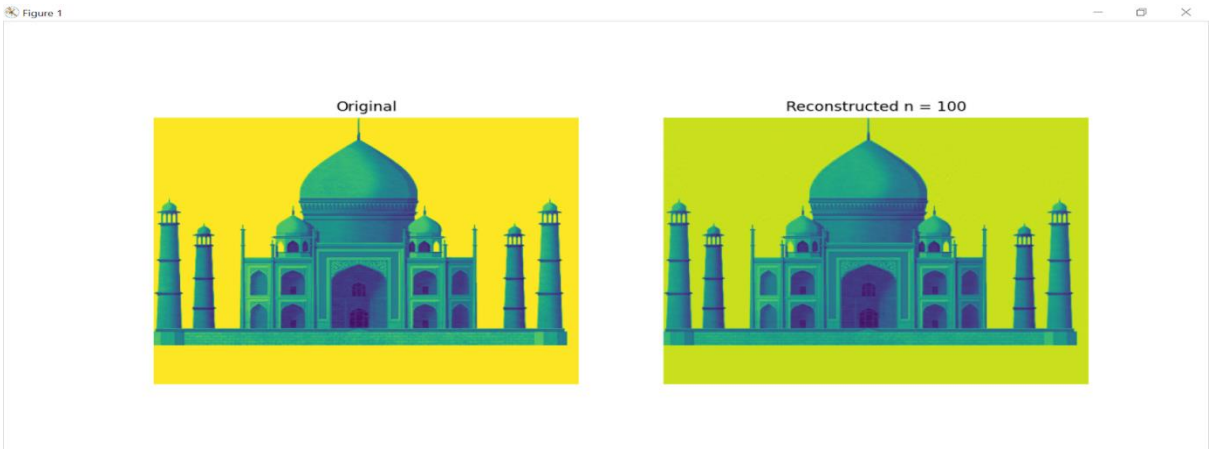
PYTHON CODE:

```
def reconstruction():
    img = Image.open('test.jpg')
    img = np.mean(img,2)

    U,s,V = np.linalg.svd(img)
    n = 10
    S = np.zeros(np.shape(img))
    for i in range(0, n):
        S[i,i] = s[i]
    recon_img = U @ S @ V
    fig, ax = plt.subplots(1, 2)

    ax[0].imshow(img)
    ax[0].axis('off')
    ax[0].set_title('Original')
    ax[1].imshow(recon_img)
    ax[1].axis('off')
    ax[1].set_title(f'Reconstructed n = {n}')

    plt.show()
```

```
m=100
for i in range(0, m):
    S[i,i] = s[i]

recon_img = U @ S @ V
fig, ax = plt.subplots(1, 2)
ax[0].imshow(img)
ax[0].axis('off')
ax[0].set_title('Original')
ax[1].imshow(recon_img)
ax[1].axis('off')
ax[1].set_title(f'Reconstructed n = {m}')
plt.show()
```

OUTPUT:

When the n value is 10 the image is not that clear but when it is increased to 100, its quite clear.Her n is the singular value.



SHEARING

INTRODUCTION

->Shear mapping is a linear map that displaces each point in fixed direction, it substitutes every point horizontally or vertically by a specific value in proportional to its x or y coordinates.

->There are two types of shearing effects

->Shearing in x-direction

->When shearing is done in the x-axis direction, the boundaries of the image that are parallel to the x-axis keep their location, and the edges parallel to y-axis changes their place depending on the shearing factor.

->Shearing in y-direction

->When shearing is done in the y-axis direction, the boundaries of the image that are parallel to the y-axis keep their location, and the edges parallel to x-axis changes their place depending on the shearing factor.

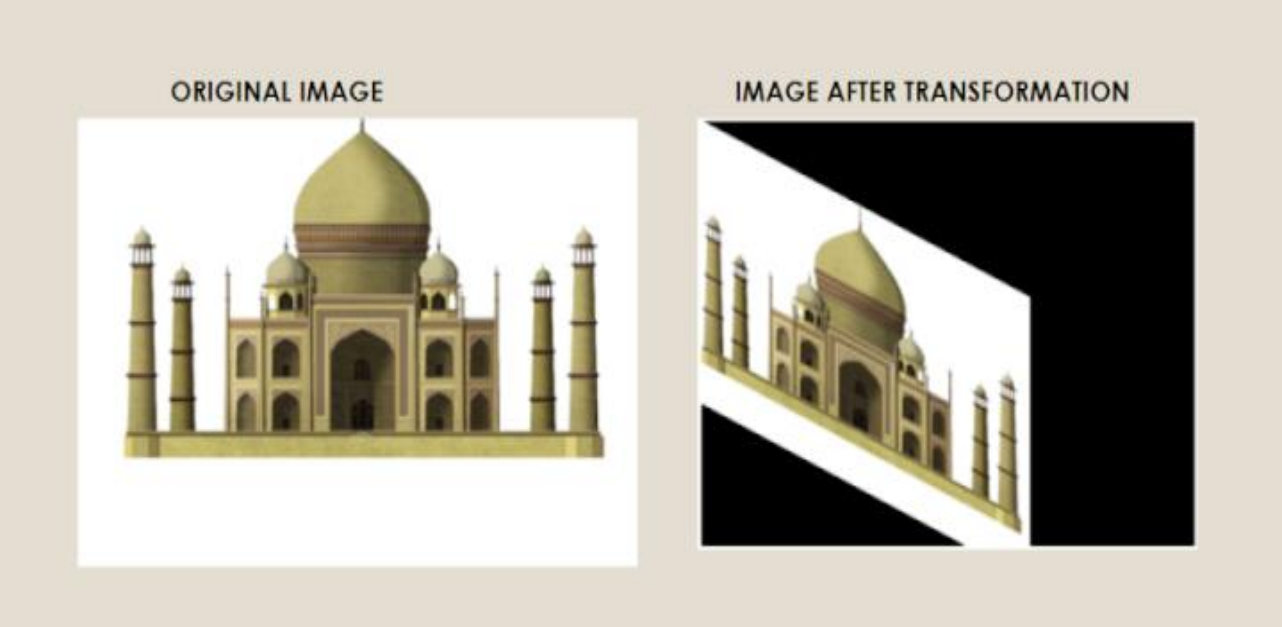
PYTHON CODE:

```
import numpy as np

import cv2
```

```
import matplotlib.pyplot as plt

# read the input image
img = cv2.imread("test.jpg")
# convert from BGR to RGB so we can plot using matplotlib
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# disable x & y axis
plt.axis('off')
# show the image
plt.imshow(img)
plt.show()
# get the image shape
rows, cols, dim = img.shape
# transformation matrix for Shearing
# shearing applied to x-axis
#M = np.float32([[1, 0.5, 0],
  #              [0, 1  , 0],
  #              [0, 0  , 1]])
# shearing applied to y-axis
M = np.float32([[1,   0.5, 0],
                [0, 1, 0],
          [0,   0, 1]])
# apply a perspective transformation to the image
sheared_img = cv2.warpPerspective(img,M,(int(cols*1.5),int(rows*1.5)))
# disable x & y axis
plt.axis('off')
# show the resulting image
```

```
plt.imshow(sheared_img)
plt.show()
# save the resulting image to disk
plt.imsave("city_sheared.jpg", sheared_img)
```

OUTPUT



ROTATION

- INTRODUCTION:

  - Implementation of a rotation filter on an image that rotates the image around its center by the specified degrees .

- Program which takes two arguments, first argument as image which is to be rotated by amount of degrees provided as second argument.

- CONCEPT:

  - Uses the concept of linear transformation particularly rotation to fill in intensity values for the coordinates required

Linear transformation used

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Where theta is the angle to be rotated

- IMPLEMENTATION

  - Converting degrees into radians
  - The center point of output image is calculated.

- The intensity values of the output image is filled in using the linear transformation equation given.

- We first find the height and width of the rotated image followed by finding the center point coordinates of both original as well as rotated image. Then we create a rotated image with appropriate height and width .

- PYTHON CODE:

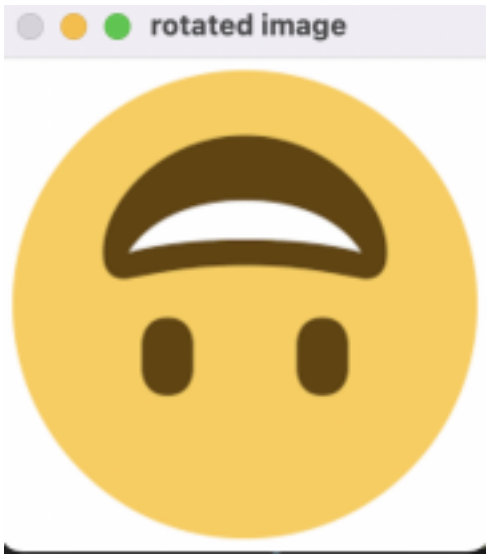
```
import numpy as np
import cv2
import math
def naive_image_rotate(image, degrees, option='same'):
 # First we will convert the degrees into radians  rads = math.radians(degrees)
 # Finding the center point of the original image  cx, cy = (image.shape[1]//2, image.shape[0]//2)
 if(option!='same'):
 # Let us find the height and width of the rotated image  height_rot_img = round(abs(image.shape[0]*math.sin(rads))) + \
 round(abs(image.shape[1]*math.cos(rads)))  width_rot_img = round(abs(image.shape[1]*math.cos(rads))) + \
 round(abs(image.shape[0]*math.sin(rads)))
 rot_img = np.uint8(np.zeros((height_rot_img,width_rot_img,image.shape[2] )))
 # Finding the center point of rotated image.
 midx,midy = (width_rot_img//2, height_rot_img//2)  else:
 rot_img = np.uint8(np.zeros(image.shape))
```

```
 for i in range(rot_img.shape[0]):
 for j in range(rot_img.shape[1]):
 if(option!='same'):
 x= (i-midx)*math.cos(rads)+(j-midy)*math.sin(rads)  y= -(i-midx)*math.sin(rads)+(j-midy)*math.cos(rads)  x=round(x)+cy
 y=round(y)+cx
 else:
 x= (i-cx)*math.cos(rads)+(j-cy)*math.sin(rads)  y= -(i-cx)*math.sin(rads)+(j-cy)*math.cos(rads)  x=round(x)+cx
 y=round(y)+cy
 if (x>=0 and y>=0 and x<image.shape[0] and  y<image.shape[1]):
 rot_img[i,j,:] = image[x,y,:]
 return rot_img
if __name__=='__main__':
 path = r'/Users/aratimenon/Desktop/laa/project/image.png'  image = cv2.imread(path)
 rotated_image = naive_image_rotate(image,180,'full')  cv2.imshow("original image", image)
 cv2.imshow("rotated image",rotated_image)
 cv2.waitKey(0)
 cv2.destroyAllWindows()
```

INPUT AND OUTPUT

ORIGINAL IMAGE

ROTATED IMAGE(180 DEGREE)



ROTATED IMAGE(45 DEGREE)

GAUSSIAN FILTER

- INTRODUCTION:

  - Implementation of a Gaussian filter/kernel to smooth/blur an image
  - Typically to reduce image noise and reduce detail using the Gaussian function.

- CONCEPT:

  - Uses the concept of gaussian function and matrix transfomations.

- IMPLEMENTATION:

  - We implement it in 2 parts:
    - Creating a Gaussian Kernel that is a mask or a small matrix for performing operations such as blurring in this case.
    - Performing Convolution i.e. transforming an image by applying a kernel over each pixel and its local neighbours across the entire image.
  - Steps involved in the image transformation:
    - Creation of kernel of required dimension.
    - Conversion of bgr to grayscale as the function takes only one channel.

- Padding the image using zero padding i.e. adding rectangular strip of zeroes outside the rectangular edge.
- Applying the kernel over each pixel of the image to get the output image

- CODE:

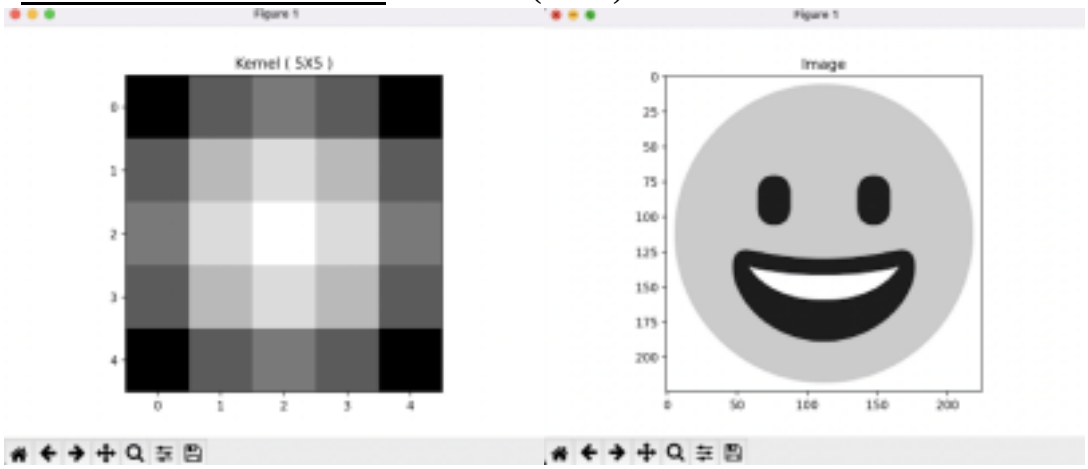
```
import numpy as np
import cv2
import argparse
import matplotlib.pyplot as plt
import math



def convolution(image, kernel, average=False,
verbose=False):  if len(image.shape) == 3:
 print("Found 3 Channels : {}".format(image.shape))  image
= cv2.cvtColor(image,
cv2.COLOR_BGR2GRAY)  print("Converted to Gray
Channel. Size :
 {}".format(image.shape))
 else:
 print("Image Shape :

{}".format(image.shape))  print("Kernel Shape :

{}".format(kernel.shape))

 if verbose:
 plt.imshow(image, cmap='gray')
```

```
 plt.title("Image")
 plt.show()

 image_row, image_col = image.shape
 kernel_row, kernel_col = kernel.shape

 output = np.zeros(image.shape)

 pad_height = int((kernel_row - 1) / 2)
 pad_width = int((kernel_col - 1) / 2)

 padded_image = np.zeros((image_row + (2 *
pad_height),  image_col + (2 * pad_width)))

 padded_image[pad_height:padded_image.shape[0] -
pad_height, pad_width:padded_image.shape[1] - pad_width]
=  image

 if verbose:
 plt.imshow(padded_image, cmap='gray')
 plt.title("Padded Image")
 plt.show()

 for row in range(image_row):
 for col in range(image_col):
 output[row, col] = np.sum(kernel *
padded_image[row:row + kernel_row, col:col +
kernel_col])  if average:
 output[row, col] /= kernel.shape[0] *

kernel.shape[1]  print("Output Image size :

{}".format(output.shape))
```

```
 if verbose:
 plt.imshow(output, cmap='gray')
 plt.title("Output Image using {}X{}
Kernel".format(kernel_row, kernel_col))
 plt.show()

 return output

def dnorm(x, mu, sd):
 return 1 / (np.sqrt(2 * np.pi) * sd) * np.e ** (-np.power((x
- mu) / sd, 2) / 2)
def gaussian_kernel(size, sigma=1,
verbose=False):  kernel_1D = np.linspace(-(size //
2), size // 2, size)  for i in range(size):
 kernel_1D[i] = dnorm(kernel_1D[i], 0,
sigma)  kernel_2D = np.outer(kernel_1D.T,
kernel_1D.T)

 kernel_2D *= 1.0 / kernel_2D.max()

 if verbose:
 plt.imshow(kernel_2D, interpolation='none',
cmap='gray')  plt.title("Kernel ( {}X{} )".format(size,
size))  plt.show()

 return kernel_2D


def gaussian_blur(image, kernel_size,
verbose=False):  kernel =
gaussian_kernel(kernel_size,
sigma=math.sqrt(kernel_size), verbose=verbose)
```

```
 return convolution(image, kernel,
average=True,  verbose=verbose)


if __name__ == '__main__':
 ap = argparse.ArgumentParser()
 ap.add_argument("-i", "--image", required=True, help="Path
to  the image")
 args = vars(ap.parse_args())

 image = cv2.imread(args["image"])

 gaussian_blur(image, 5, verbose=True)
```

- INPUT AND OUTPUT: KERNEL(5 X 5) GRAYSCALE IMAGE



PADDED IMAGE OUTPUT IMAGE

LINEAR FILTER

- <u>INTRODUCTION:</u>

  •These process time-varying input signals to produce output signals, subject to the constraint of linearity.

  •It is a filtering in which the value of an output pixel is a linear combination of the values of the input pixel's neighborhood.

- <u>CONCEPT:</u>

  - The concept used here is convolution in 2 dimension.
  - [ie,It uses a 2D matrix]

- <u>APPLICATIONS:</u>

- Generic tasks such as image/video contrast improvement
- Denoising
- sharpening
- feature enhancement

- PYTHON CODE:

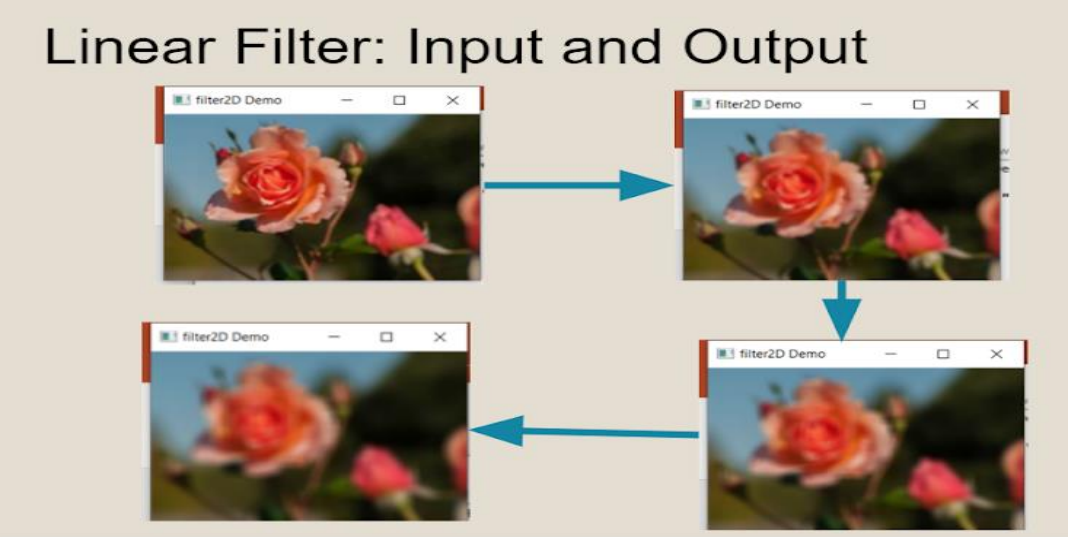
```
import sys
import cv2 as cv
import numpy as np
def main():
    window_name = 'filter2D Demo'
    img=input("Enter image file name:")

    imageName = img if len(img) > 0 else 'img1.jpg'
    # Loads an image
    src = cv.imread(cv.samples.findFile(imageName),
cv.IMREAD_COLOR)
    # Check if image is loaded fine
    if src is None:
        print ('Error opening image!')
        print ('Usage: filter2D.py [image_name -- default lena.jpg] \n')
        return -1

    ddepth = -1

    ind = 0
    while ind<11:

        kernel_size = 3 + 2 * (ind % 5)
```

```
    kernel = np.ones((kernel_size, kernel_size), dtype=np.float32)
    kernel /= (kernel_size * kernel_size)

    dst = cv.filter2D(src, ddepth, kernel)

    cv.imshow(window_name, dst)
    c = cv.waitKey(1000)
    if ind == 10:
       break
    ind += 1
  return 0

main()
```

- WORKING OF CODE:

  The above code works in the following way:
  1.It first loads the original image and initializes the variable (ddepth).
  2.Then we define the kernel .
  3.This can be done by defining Kernel size.
  4. Filling the kernel with 1's and normalizing it.
  5.Then we use the source image and Kernel to get output.
  6.The output is that every 1 second the kernel size of this filter will be updated.

- SAMPLE INPUT AND OUTPUT:



RESIZING

- <u>INTRODUCTION:</u>
    - It is the process of changing the scale of a discrete or continuous image by way of resolution.
    - The whole image is present after applying resizing.
- <u>CONCEPT:</u>
    - The concept used here is, if we want to resize an image, we shall have to change its ‘pixel information’.
    - [This is done with the help of matrices]

- APPLICATIONS:

  - transliteration of the image
  - correcting for lens distortion
  - changing perspective
  - rotating a picture

- PYTHON CODE:

```
# open-cv library is installed as cv2 in python
# import cv2 library into this program
import cv2

# import numpy library as np into this program
import numpy as np

# read an image using imread() function of cv2
# we have to  pass only the path of the image

img = cv2.imread(r'3.jpg')

# displaying the image using imshow() function of cv2
# In this : 1st argument is name of the frame
# 2nd argument is the image matrix
cv2.imshow('original image',img)

# print shape of the image matrix
# using shape attribute
print("original image shape:",img.shape)

# assigning number of rows, coulmns and
# planes to the respective variables
row,col,plane = img.shape
```

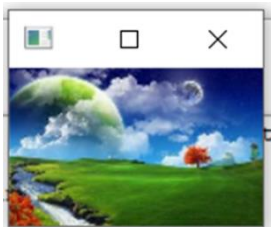
```
# give value by which you want to resize an image
# here we want to resize an image as one half of the original
image
x, y = 2, 2

# assign Blue plane of the BGR image
# to the blue_plane variable
blue_plane = img[:,:,0]

# assign Green plane of the BGR image
# to the green_plane variable
green_plane = img[:,:,1]

# assign Red plane of the BGR image
# to the red_plane variable
red_plane = img[:,:,2]

# we take one-half pixel of rows and columns from
# each plane respectively so that, it is one-half of image matrix.

# here we take alternate row,column pixel of blue plane.
resize_blue_plane = blue_plane[1::x,1::x]

# here we take alternate row,column pixel of green plane.
resize_green_plane = green_plane[1::x,1::x]

# here we take alternate row,column pixel of red plane.
resize_red_plane = red_plane[1::x,1::x]

# here image is of class 'uint8', the range of values
# that each colour component can have is [0 - 255]

# create a zero matrix of specified order of 3-dimension
resize_img = np.zeros((row//x, col//y, plane),np.uint8)
```

```
# assigning resized blue, green and red plane of image matrix to
the
# corresponding blue, green, red plane of resize_img matrix
variable.
resize_img[:,:,0] = resize_blue_plane
resize_img[:,:,1] = resize_green_plane
resize_img[:,:,2] = resize_red_plane

cv2.imshow('resize image',resize_img)

print("resize image shape:",resize_img.shape)
```

- WORKING OF CODE:

  The code works in the following way:
  1.It first loads the original image and initializes the variables (x and y).
  2.Then we assign 3 variables[blue plane, green plane, red plane.
  3.Each variable has the specific plane value, which is got from that specific plane value of the blue-green-red image.
  4.It take 1/x pixels of the row & 1/y pixels of the column for each value.
  5.Then we create a zero matrix for the resized image variable.
  6.Assigning the new ‘blue plane’, ‘red plane’ and ‘green plane’ values to get the resized image and displaying the same.

- SAMPLE INPUT AND OUTPUT:



Median Filter:

The **Median Filter in Image Processing** is normally used to reduce noise in an image, somewhat like the mean filter. However, it often does a better job than the mean filter of preserving useful detail in the image.

Median Filter is a simple and powerful non-linear filter.

- It is used for reducing the amount of intensity variation between one pixel and the other pixel.
- In this filter, we replace pixel value with the median value.
- The median is calculated by first sorting all the pixel values into ascending order and then replace the pixel being calculated with the middle pixel value
- Reduces salt and pepper noise.

Working of Median Filter:

The median filter considers each pixel in the image in turn and looks at its nearby neighbour to decide whether or not it is representative of its surroundings. Replacing the pixel value with the ***median*** of those values.

The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighborhood under consideration contains an even number of pixels, the average of the two middle pixel values is used)

Code for Median Filter in Python:

```
#importing all the required modules

import cv2

import numpy as np

#reading the image whose noise is to be removed using imread() function

img = cv2.imread('brain.jpg')

#using medinaBlur() function to remove the noise from the given image

median = cv2.medianBlur(img, 5)

compare = np.concatenate((img, median), axis=1)

#displaying the noiseless image as the output on the screen

#side by side comparison

cv2.imshow('img', compare)
```

cv2.waitKey(0)

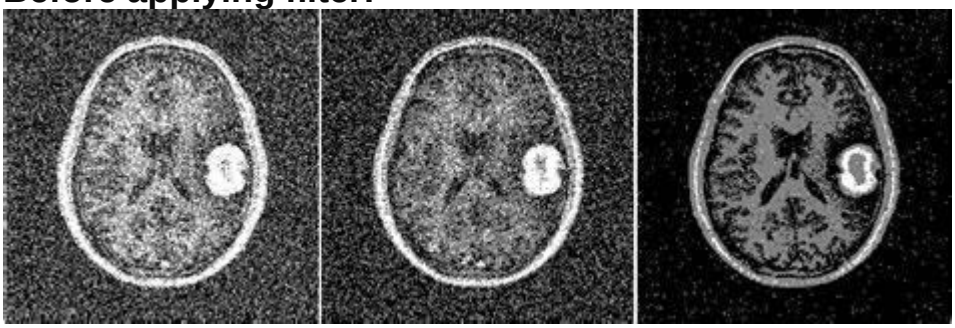
cv2.destroyAllWindows

Output:

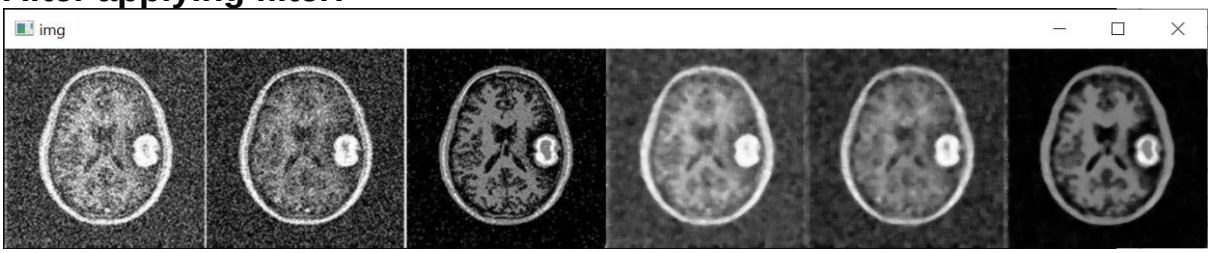
```
D:\>python Median.py
```

Example 1:

Before applying filter:



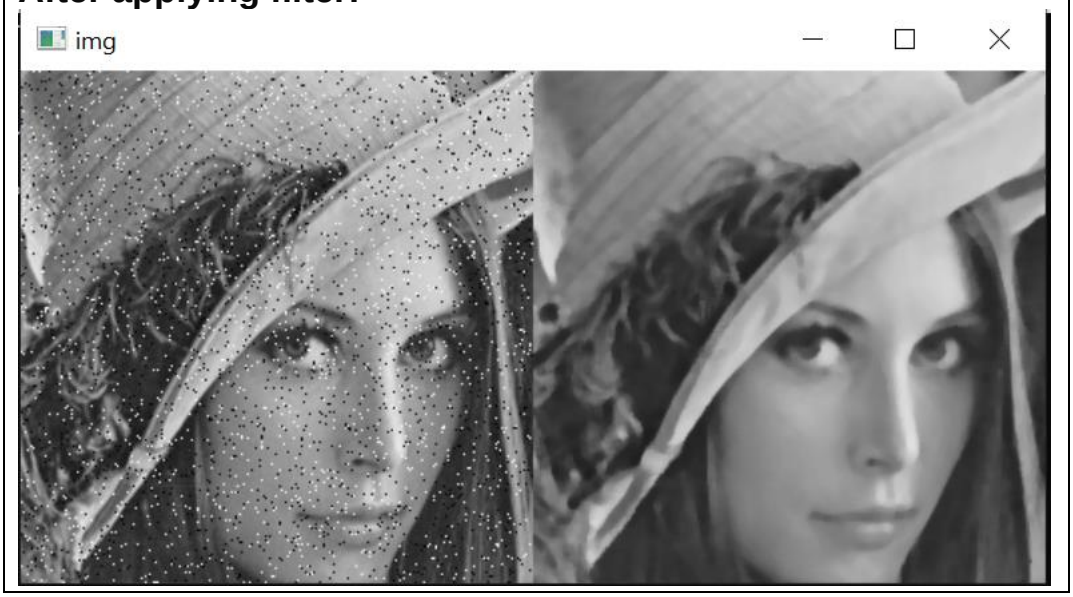
After applying filter:



Example 2:

Before applying filter:

After applying filter:



Scaling transformations:

The scale operator performs a geometric transformation which can be used to shrink or zoom the size of an image (or part of an image). Image reduction, commonly known as *subsampling*, is performed by replacement (of a group of by pixel values one arbitrarily chosen pixel value from within this group) or by *interpolating* between pixel values in a local neighborhoods.

Image zooming is achieved by *pixel replication* or by interpolation. Scaling is used to change the visual appearance of an image, to alter the

quantity of information stored in a scene representation, or as a low-level preprocessor in multi-stage image processing chain which operates on features of a particular scale. Scaling is a special case of **affine transformation**.

Code for Scaling transformation in python:

```
import cv2


import numpy as np




try:

   # Read image from disk.


   img = cv2.imread('nature.jpg')


   # Get number of pixel horizontally and vertically.


   (height, width) = img.shape[:2]


   # Specify the size of image along with interploation methods.


   # cv2.INTER_AREA is used for shrinking, whereas cv2.INTER_CUBIC
```

```
    # is used for zooming.


    res = cv2.resize(img, (int(width / 2), int(height / 2)), interpolation =
cv2.INTER_CUBIC)


    # Write image back to disk.


    cv2.imwrite('result.jpg', res)


except IOError:


    print ('Error while reading files !!!')
```

Output:

```
D:\>python Scaling.py
```

Example 1:

Before applying transformation:

After applying transformation :

Example 2:

Before applying transformation:

After applying filter: