

Practical 1

Aim: Write a program for generation and manipulation of various signals.

Theory:

Image Processing is processing of images using mathematical operations by using any form of signal processing for which the input is an image, a series of images, or a video, such as a photograph or video frame; the output of image processing may be either an image or a set of characteristics or parameters related to the image. Most image-processing techniques involve treating the image as a two-dimensional signal and applying standard signal-processing techniques to it. Images are also processed as three-dimensional signals where the third-dimension being time or the z-axis.

Image processing usually refers to digital image processing, but optical and analog image processing also are possible. This article is about general techniques that apply to all of them. Signal can be analog (continuous in time) or discrete (discrete in time). Here we generate analog as well as discrete signals. Digital images are discrete signals which are generally sampled from an analog signal. Signal manipulation is important as it results in signal processing and hence image processing. Manipulations include transformations, mathematical operations etc.

Code:

```
//utility function
function [x]=multiply(a, b)
    for i = 1:201
        x(i) = a(i) * b(i);
    end
endfunction

t = 0:0.1:20;
unit_step = ones(1, 201)
ramp = 0.1 * t;                //k = 0.1
expo1 = exp(0.2 * t)           //a = 0.2
expo2 = exp(-0.2 * t)          //a = -0.2
sint = sin(t);
cost = cos(t);

subplot(3, 1, 1);
plot(t, unit_step);
xlabel("t", "fontsize", 2);
ylabel("u(t)", "fontsize", 2);
legend("unit step function");
subplot(3, 1, 2);
plot(t, sint, t, cost);
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
```

```

legend("sin(t)", "cos(t)");
subplot(3, 1, 3);
plot(t, expo2, t, expo1);
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("e^(at)", "e^(-at)");
plot(t, multiply(ramp, sint)', t, multiply(ramp, cost)', t, multiply(sint,
cost)');
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("r(t) * sin(t)", "r(t) * cos(t)", "sin(t) * cos(t)");

plot(t, multiply(expo1, sint)', t, multiply(expo1, cost)', t, multiply(sint,
cost)');
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("e^(at) * sin(t)", "e^(at) * cos(t)", "sin(t) * cos(t)");

plot(t, ramp+sint, t, ramp+cost, t, sint+cost);
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("r(t) + sin(t)", "r(t) + cos(t)", "sin(t) + cos(t)");

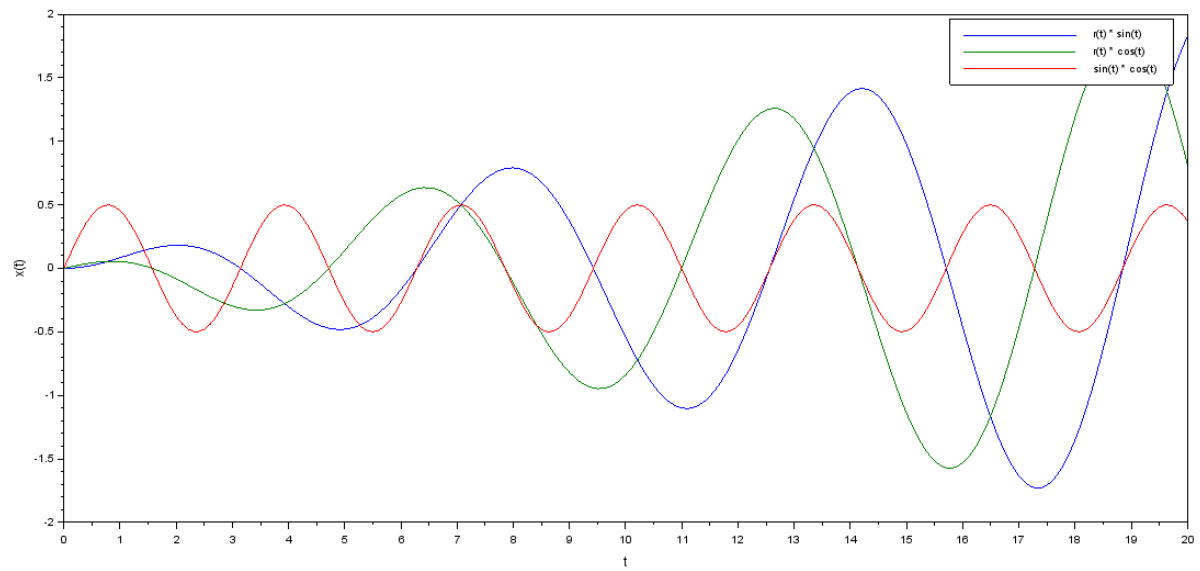
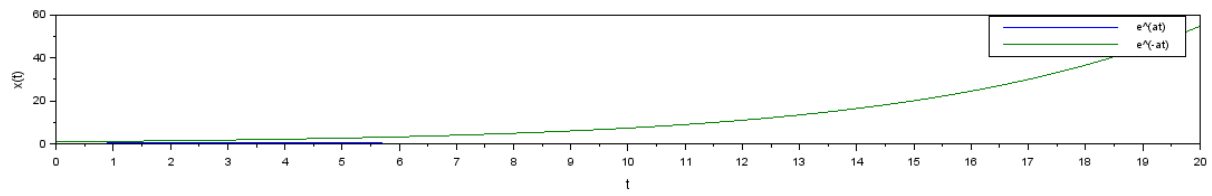
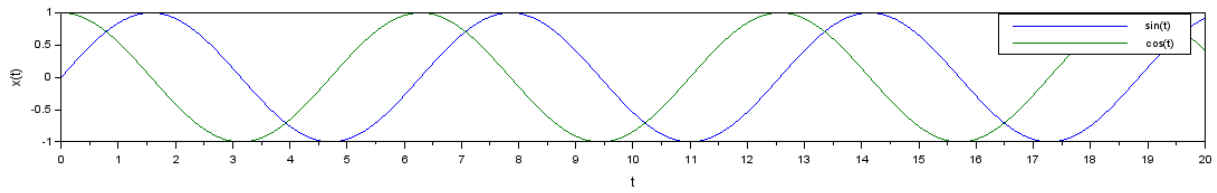
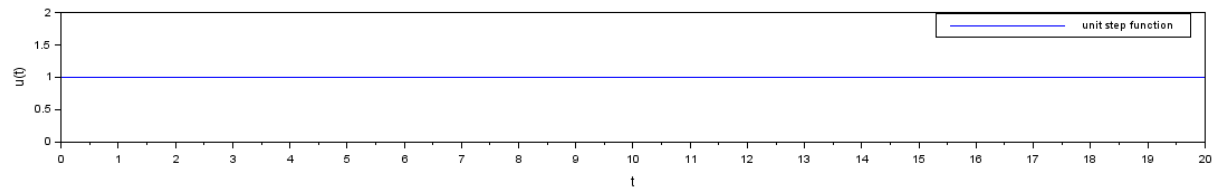
plot(t, expo2+sint, t, expo2+cost, t, sint+cost);
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("e^(-at) + sin(t)", "e^(-at) + cos(t)", "sin(t) + cos(t)");

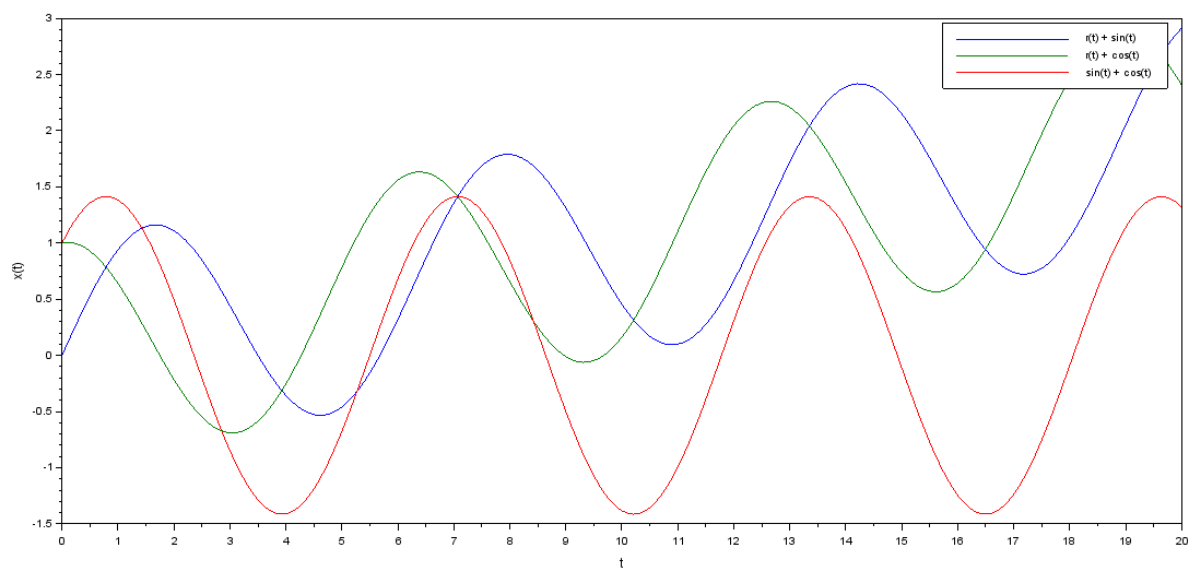
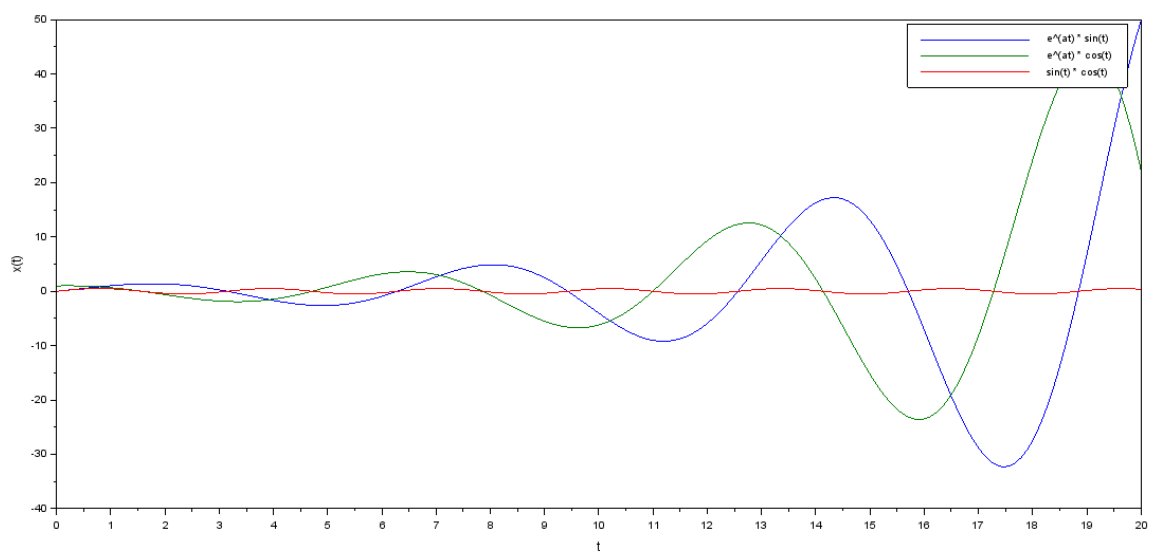
plot(t, multiply(expo1, ramp), t, expo1-ramp, t, expo1+ramp);
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("e^(at) * r(t)", "e^(at) - r(t)", "e^(at) + r(t)");

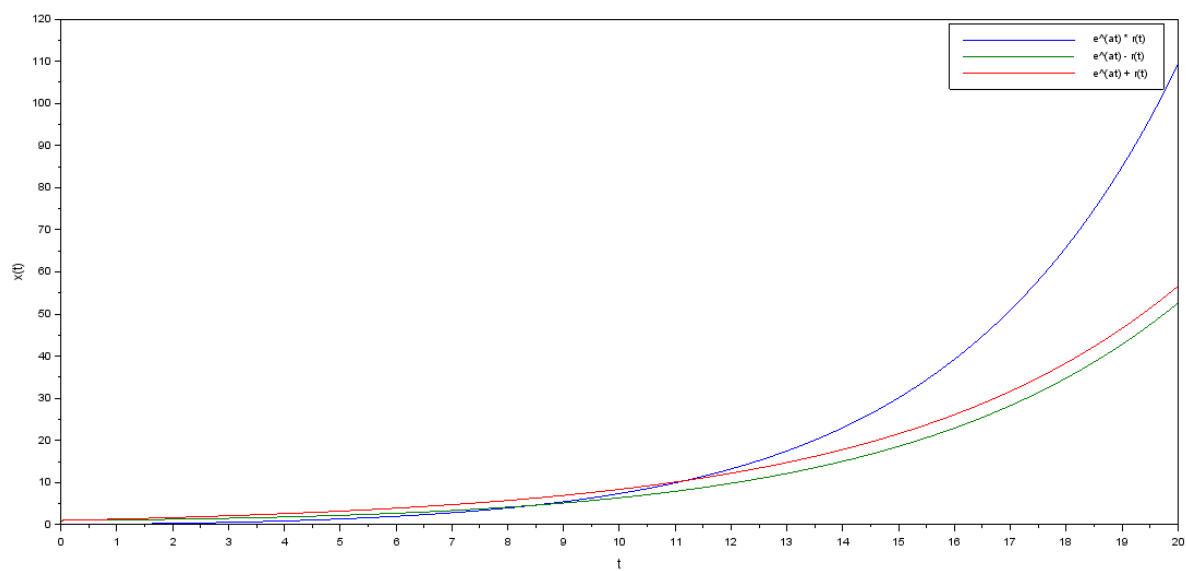
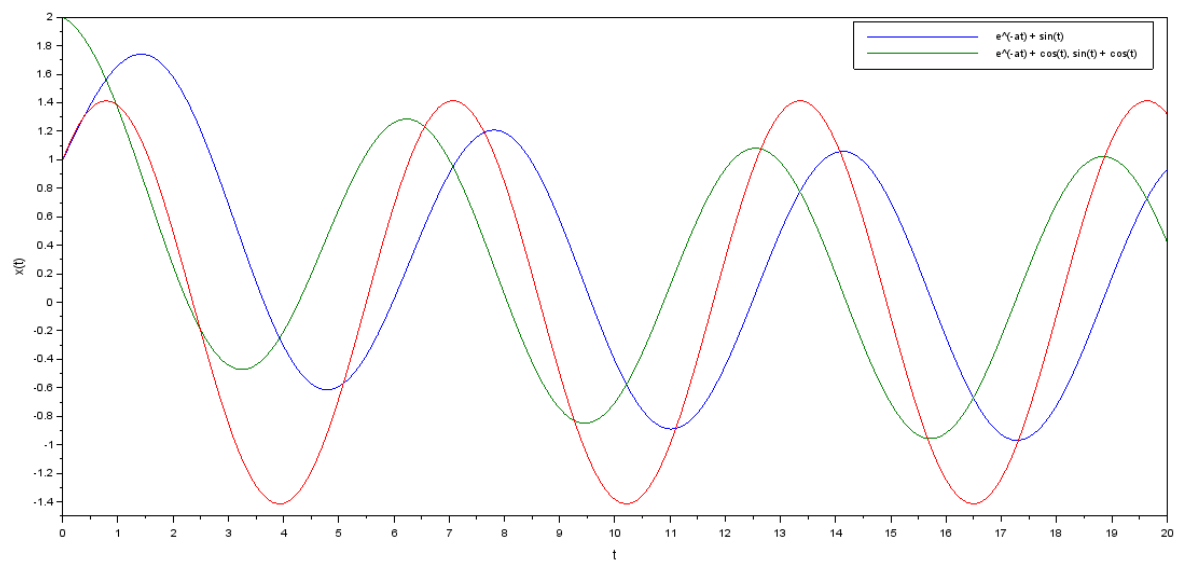
plot(t, sint+cost+ramp, t, sint-cost+ramp, t, multiply(sint, cost));
xlabel("t", "fontsize", 2);
ylabel("x(t)", "fontsize", 2);
legend("r(t) + sin(t) + cos(t)", "sin(t) - cos(t) + r(t)", "sin(t) *
cos(t)");

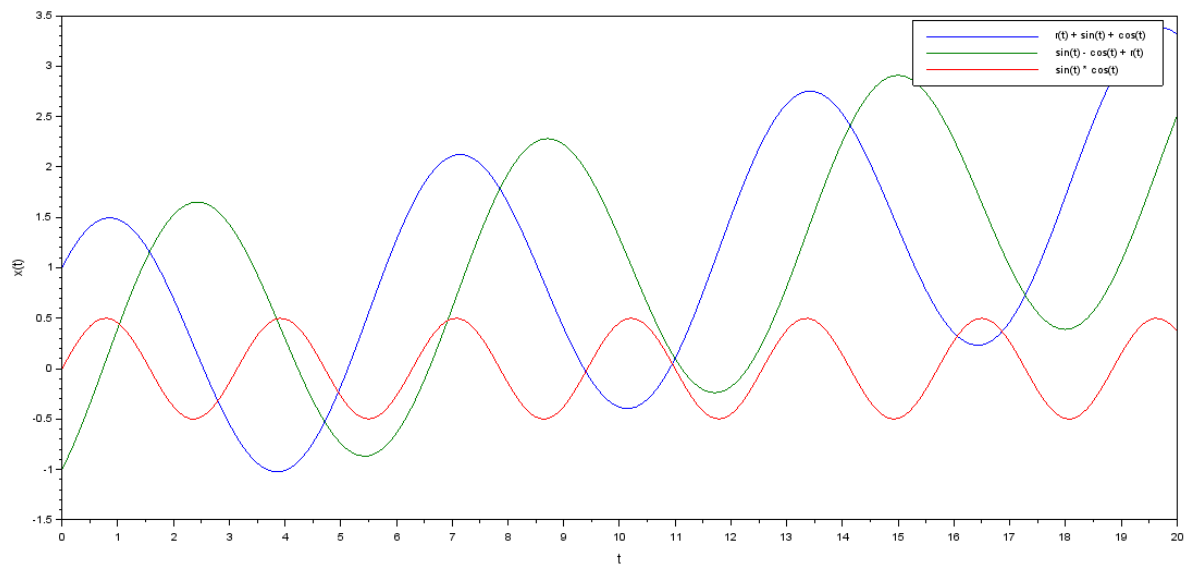
```

Output:









Conclusion: Hence we have studied how to generate signals and manipulate them and the importance of signal processing in image processing.

Practical 2

Aim: Write a program to demonstrate the principle of convolution and correlation

Theory:

Convolution is a mathematical way of combining two signals to form a third signal. It is the single most important technique in Digital Signal Processing. Using the strategy of impulse decomposition, systems are described by a signal called the impulse response. Convolution is used in image filtering. Convolution of two signals $x[n]$ and $h[n]$ is given by:

$$y[n] = x[n] ** h[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k]$$

Correlation of signal means to compare one reference signal with another one to determine the similarity between them. If signal is compared with itself, it is known as autocorrelation if it is compared with another it is cross correlation. Correlation is used to match images. The higher the correlation, more similar are the images. Correlation is given by:

$$r = \frac{1}{n-1} \sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s_x} \right) \left(\frac{y_i - \bar{y}}{s_y} \right)$$

Code:

```
funcprot(0)

function [] = convolution(img)
    g = 1/25. * ones(3, 3);
    conv_img = imfilter(img, g);
    imshow(conv_img);
endfunction

function [] = correlation(img, g1)
    i1 = img;
    i1 = rgb2gray(i1);
    g1 = rgb2gray(g1);
    i2 = imcomplement(i1);
    //imshow(i2);
    g2 = imcomplement(g1);
    //imshow(g2);
    i3 = im2double(i2);
    g3 = im2double(g2);
    i4 = imfilter(i3, g3);
    i5 = imnorm(i4);
    imshow(i5);
    loc = i5 > 0.9;
    [rows, cols] = find(loc);
```

```
    imshow(img);
    sz = size(img);
    plot(cols, sz(1) - rows, 'r.');
endfunction

function [] = main()
    cd "G:\Academics\Semester 7\Image Processing\Practicals\Practical 2\";
    /*img = imread('lconv.jpg');
    imshow(img);
    convolution(img);*/
    i1 = imread('l.png');
    g1 = imread('a.png');
    acorrelation(i1, g1);
endfunction

main
```

Output:

Convolution:



Before Convolution

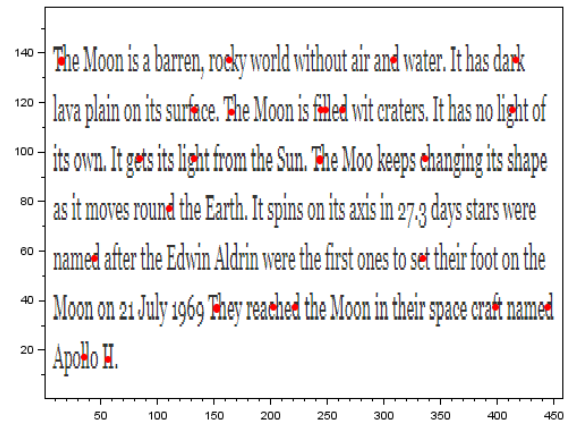


After Convolution

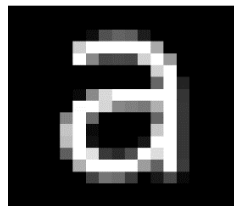
Correlation:

The Moon is a barren, rocky world without air and water. It has dark lava plain on its surface. The Moon is filled wit craters. It has no light of its own. It gets its light from the Sun. The Moo keeps changing its shape as it moves round the Earth. It spins on its axis in 27.3 days stars were named after the Edwin Aldrin were the first ones to set their foot on the Moon on 21 July 1969 They reached the Moon in their space craft named Apollo II.

Original Image



After template matching



Template

Conclusion: Thus, we have successfully implemented certain principles that demonstrate correlation and convolution.

Practical 3

Aim: Write a Program to find Discrete Fourier Transform of a given Image

Theory:

The Fourier Transform is an important image processing tool which is used to decompose an image into its sine and cosine components. The output of the transformation represents the image in the Fourier or frequency domain, while the input image is the spatial domain equivalent. In the Fourier domain image, each point represents a particular frequency contained in the spatial domain image. The Fourier Transform is used in a wide range of applications, such as image analysis, image filtering, image reconstruction and image compression. The DFT is the sampled Fourier Transform and therefore does not contain all frequencies forming an image, but only a set of samples which is large enough to fully describe the spatial domain image. The number of frequencies corresponds to the number of pixels in the spatial domain image, i.e. the image in the spatial and Fourier domain are of the same size.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N-1} x_n \cdot [\cos(2\pi kn/N) - i \cdot \sin(2\pi kn/N)], \end{aligned}$$

Code:

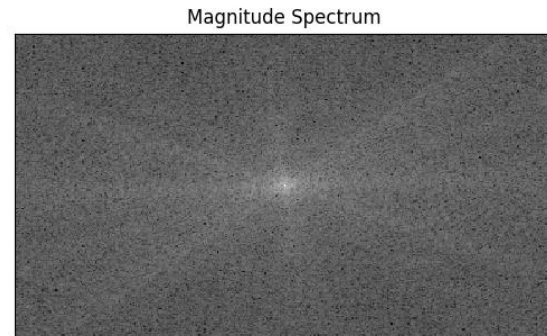
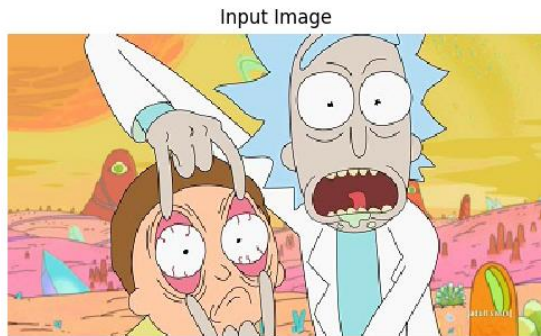
```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

def main():
    img = cv.imread('1.jpg')
    img_grayscale = cv.imread('1.jpg', 0)
    dft = cv.dft(np.float32(img_grayscale), flags =
cv.DFT_COMPLEX_OUTPUT)
    dft_shift = np.fft.fftshift(dft)
    spectrum = 20 *
np.log(cv.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]))
    # show results
    plt.subplot(121), plt.axis('off'), plt.imshow(cv.cvtColor(img,
cv.COLOR_BGR2RGB))
    plt.title('Input Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(spectrum, cmap = 'gray')
    plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
    plt.show()

    print "Done"
```

```
if __name__ == "__main__":  
    main()
```

Output:



Conclusion: Hence we have studied how to convert spatial domain to frequency domain using Fourier Transform. Since the range in Fourier domain is very high we take the natural log of the gray level value to display the spectrum. We learnt the use of Fourier Transform in manipulation of image in frequency domain.

Practical 4

Aim: Write a program to study different point operations on an Image

Theory:

The simplest image filters are point operations, where the new value of a pixel are only determined by the original value of that single pixel alone. Point operation has following

Contents: Thresholding - select pixels with given values to produce binary image

Code:

```
import cv2 as cv2
import numpy as np

def main():
    img = cv2.imread('1.jpg', 0)
    ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
    ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
    ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)
    cv2.imshow("Image", img)
    cv2.imshow("Binary Threshold", thresh1)
    cv2.imshow("Truncated Threshold", thresh3)
    cv2.imshow("To Zero Threshold", thresh5)
    cv2.waitKey(0)

if __name__ == '__main__':
    main()
```

Output:





Conclusion: Hence we have studied about various point operations that operate only on one pixel at a time independent of other pixels. These are the simplest transformations. There are certain other operations like piece-wise linear transformation and contrast stretching that operate on a single pixel but depend partially on other pixels.

Practical 5

Aim: Write a program to enhance image using histogram equalization

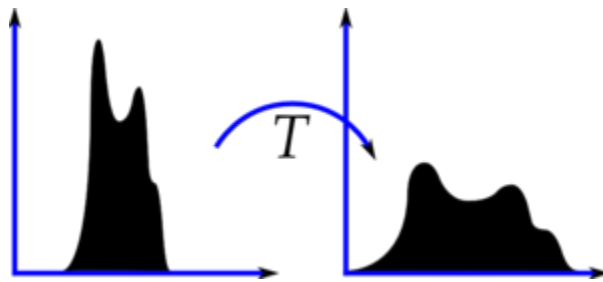
Theory:

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.

A key advantage of the method is that it is a fairly straightforward technique and an invertible operator. So, in theory, if the histogram equalization function is known, then the original histogram can be recovered. The calculation is not computationally intensive. A disadvantage of the method is that it is indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal.

$$h(v) = \text{round} \left(\frac{cdf(v) - cdf_{min}}{(M \times N) - 1} \times (L - 1) \right)$$

Consider an image whose pixel values are confined to some specific range of values only. For example, brighter image will have all pixels confined to high values. But a good image will have pixels from all regions of the image. So, you need to stretch this histogram to either ends (as given in below image, from Wikipedia) and that is what Histogram Equalization does (in simple words). This normally improves the contrast of the image.



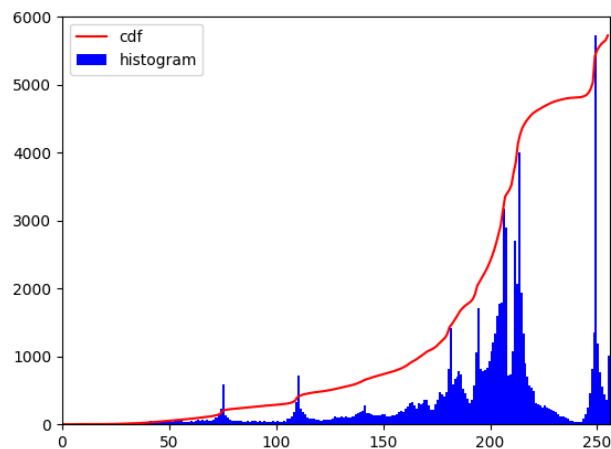
Code:

```
import cv2
from matplotlib import pyplot as plt
import numpy as np

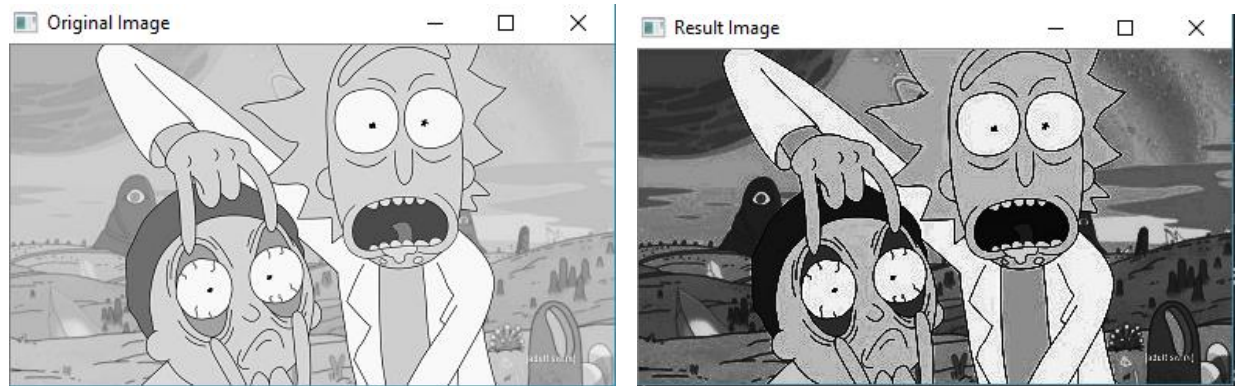
def main():
    img = cv2.imread('1.jpg', 0)
    # histogram of image
    hist,bins = np.histogram(img.flatten(),256,[0,256])
    cdf = hist.cumsum()
    cdf_normalized = cdf * hist.max() / cdf.max()
    plt.plot(cdf_normalized, color = 'r')
    plt.hist(img.flatten(),256,[0,256], color = 'b')
    plt.xlim([0,256])
    plt.legend(('cdf','histogram'), loc = 'upper left')
    plt.show()
    # equalization
    cv2.imshow("Original Image", img)
    res = cv2.equalizeHist(img)
    cv2.imwrite('res.png', res)
    cv2.imshow("Result Image", res)
    cv2.waitKey(0)

if __name__ == '__main__':
    main()
```

Output:



Histogram of Image



Conclusion: Hence, we studied how histogram equalization is used to enhance images. It adjusts the contrast by increasing it. This increase in contrast results in better images.

Practical 6

Aim: Write a program for image smoothing in spatial domain

Theory:

Image smoothing is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (e.g: noise, edges) from the image resulting in edges being blurred when this filter is applied.

Here, we use the averaging technique. This is done by convolving the image with a normalized box filter. It simply takes the average of all the pixels under kernel area and replaces the central element with this average. A 3x3 normalized box filter would look like this:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Code:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def main():
    img = cv2.imread('1.jpg')
    kernel = np.ones((3,3),np.float32)/9
    dst = cv2.filter2D(img,-1,kernel)
    plt.subplot(121),plt.imshow(img),plt.title('Original')
    plt.xticks([], plt.yticks([]))
    plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
    plt.xticks([], plt.yticks([]))
    plt.show()

if __name__ == '__main__':
    main()
```

Output:

Original



Averaging



Conclusion: Hence, we learnt how image smoothing is performed. We studied the filters applied to blur the image and implemented smoothing using averaging filter.

Practical 7

Aim: Write a program for image sharpening in spatial domain.

Theory:

Human perception is highly sensitive to edges and fine details of an image, and since they are composed primarily by high frequency components, the visual quality of an image can be enormously degraded if the high frequencies are attenuated or completely removed. In contrast, enhancing the high-frequency components of an image leads to an improvement in the visual quality. Image sharpening refers to any enhancement technique that highlights edges and fine details in an image. In principle, image sharpening consists of adding to the original image a signal that is proportional to a high-pass filtered version of the original image.

This formula has been used in the following code to sharpen image:

Sharpened = Original + (Original - Blurred) * Amount

Where: Amount is how much contrast is added at the edges. Such kernels can be used to sharpen images:

$$M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

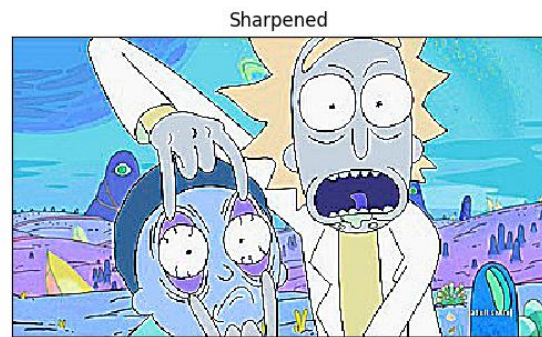
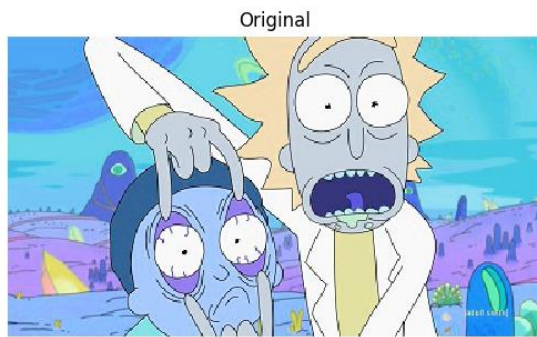
Code:

```
import cv2
from matplotlib import pyplot as plt
import numpy as np

def main():
    img = cv2.imread('1.jpg')
    shar = cv2.GaussianBlur(img, (5,5),0)
    shar = cv2.addWeighted(img, 5, shar, -4, 0);
    plt.subplot(121),plt.imshow(img),plt.title('Original')
    plt.xticks([], plt.yticks([], plt.axis('off')
    plt.subplot(122),plt.imshow(shar),plt.title('Sharpened')
    plt.xticks([], plt.yticks([],
    plt.show()

if __name__ == '__main__':
    main()
```

Output:



Conclusion: Hence, we studied how image sharpening is achieved. Here, we used Gaussian blur to obtain a blurred image which was then processed to obtain a sharpened image. The result was an image with more defined edges than the original.

Practical 8

Aim: Write a program for image compression.

Theory:

Image compression is minimizing the size in bytes of a graphics file without degrading the quality of the image to an unacceptable level. The reduction in file size allows more images to be stored in a given amount of disk or memory space. It also reduces the time required for images to be sent over the Internet or downloaded from Web pages.

The functions used here are:

- `cv2.imencode(ext, img[, params]) → retval, buf`

The function compresses the image and stores it in the memory buffer that is resized to fit the result.

- `cv2.imdecode(buf, flags) → retval`

The function reads an image from the specified buffer in the memory. If the buffer is too short or contains invalid data, the empty matrix/image is returned.

The param used is `encode_param=[int(cv2.IMWRITE_JPEG_QUALITY), 50]`. Lesser the quality, lesser the data size.

Code:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def main():
    img = cv2.imread("1.jpg", 1)
    encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), 50]
    result, encimg = cv2.imencode('1.jpg', img, encode_param)
    decimg = cv2.imdecode(encimg, 1)
    plt.subplot(121), plt.imshow(img), plt.title('Original Image')
    plt.xticks([], plt.yticks([]), plt.axis('off'))
    plt.subplot(122), plt.imshow(decimg), plt.title('Compressed
Image')
    plt.xticks([], plt.yticks([]))
    plt.show()
    cv2.imwrite('compress1.jpg', decimg)

if __name__ == '__main__':
    main()
```

Output:

Original Image




Compressed Image



 1.jpg

11/2/2017 5:11 AM JPG File

42 KB

 compress1.jpg

11/2/2017 1:58 PM JPG File

34 KB

Image Size

Conclusion: In this practical, we studied image compression. We first encoded the image using appropriate flags and then decoded it to form the compressed image.

Practical 9

Aim: Write a program for image segmentation.

Theory:

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as super-pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.)

Any grayscale image can be viewed as a topographic surface where high intensity denotes peaks and hills while low intensity denotes valleys. You start filling every isolated valley (local minima) with different colored water (labels). As the water rises, depending on the peaks (gradients) nearby, water from different valleys, obviously with different colors will start to merge. To avoid that, you build barriers in the locations where water merges. You continue the work of filling water and building barriers until all the peaks are under water. Then the barriers you created give you the segmentation result. This is the "philosophy" behind the watershed. You can visit the CMM webpage on watershed to understand it with the help of some animations.

Code:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def main():
    img = cv2.imread('water_coins.jpg')
    cv2.imshow("Original Image", img)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(img_gray, 0, 255, cv2.THRESH_BINARY_INV
+ cv2.THRESH_OTSU)
    # remove small noises using opening
    kernel = np.ones((3,3), np.uint8)
    img_open = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel,
iterations = 2)
    # increase object boundary as we are not sure of background
    sure_bg = cv2.dilate(img_open, kernel, iterations=3)
    # foreground detection
    dist_transform = cv2.distanceTransform(img_open, cv2.DIST_L2, 5)
    _, sure_fg = cv2.threshold(dist_transform, 0.7 *
dist_transform.max(), 255, 0)
    sure_fg = np.uint8(sure_fg)
    # boundary where bg and fg meet
    unknown = cv2.subtract(sure_bg, sure_fg)
    _, markers = cv2.connectedComponents(sure_fg)
```



```
markers += 1
markers[unknown == 255] = 0
markers = cv2.watershed(img, markers)
img[markers == -1] = [255,0,0]
cv2.imshow("Segmented Image", img)
cv2.waitKey(0)

if __name__ == '__main__':
    main()
```

Output:



Conclusion: In this practical, we learnt how to use the watershed algorithm to perform the image segmentation which leads to defined boundaries or edges around individual objects in the image.

Practical 10

Aim: Write a program for edge detection

Theory:

Edge detection includes a variety of mathematical methods that aim at identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities. The points at which image brightness changes sharply are typically organized into a set of curved line segments termed edges.

Canny Edge Detection is a popular edge detection algorithm. It is a multi-stage algorithm.

1. Noise Reduction

Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter.

2. Finding Intensity Gradient of the Image

Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (G_x) and vertical direction (G_y). From these two images, we can find edge gradient and direction for each pixel as follows:

$$\text{Edge_Gradient } (G) = \sqrt{G_x^2 + G_y^2}$$
$$\text{Angle } (\theta) = \tan^{-1} \left(\frac{G_y}{G_x} \right)$$

3. Non-maximum Suppression

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a local maximum in its neighborhood in the direction of gradient.

4. Hysteresis Thresholding

This stage decides which are all edges are really edges and which are not. For this, we need two threshold values, minVal and maxVal. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to “sure-edge” pixels, they are considered to be part of edges. Otherwise, they are also discarded.

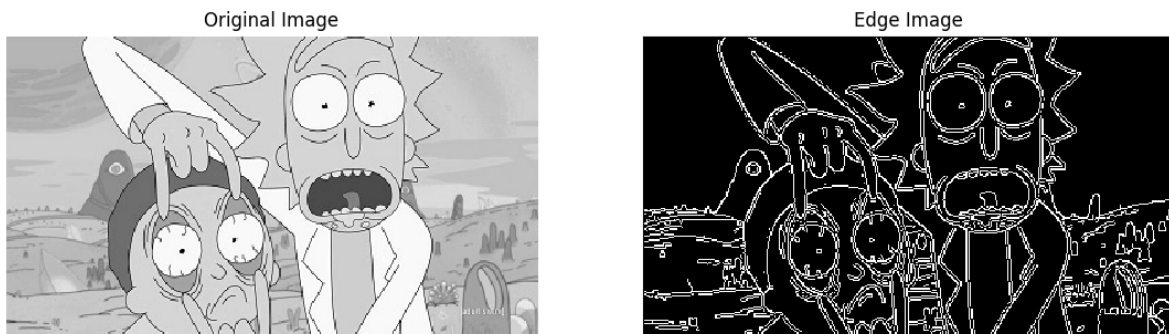
Code:

```
import cv2
from matplotlib import pyplot as plt

def main():
    img = cv2.imread('1.jpg', 0)
    edges = cv2.Canny(img, 100, 200)
    plt.subplot(121), plt.imshow(img, cmap = 'gray'), plt.axis('off')
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(edges, cmap = 'gray')
    plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
    plt.show()

if __name__ == '__main__':
    main()
```

Output:



Conclusion: We learned how to perform edge detection in an image using Canny Edge Detection Algorithm in this practical and also implemented it successfully.

Practical 11

Aim: Write a program for image morphology

Theory:

Morphology is a broad set of image processing operations that process images based on shapes. Morphological operations apply a structuring element to an input image, creating an output image of the same size. In a morphological operation, the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. By choosing the size and shape of the neighborhood, you can construct a morphological operation that is sensitive to specific shapes in the input image. The most basic morphological operations are dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries.

1. Erosion:

The basic idea of erosion is just like soil erosion only, it erodes away the boundaries of foreground object. The kernel slides through the image. A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero).

2. Dilation:

It is just opposite of erosion. Here, a pixel element is '1' if at least one pixel under the kernel is '1'. So, it increases the white region in the image or size of foreground object increases. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So, we dilate it. Since noise is gone, they won't come back, but our object area increases. It is also useful in joining broken parts of an object.

3. Opening:

Opening is just another name of erosion followed by dilation. It is useful in removing noise, as we explained above.

4. Closing:

Closing is reverse of Opening, **Dilation followed by Erosion**. It is useful in closing small holes inside the foreground objects, or small black points on the object.

Code:

```
import cv2
from matplotlib import pyplot as plt
import numpy as np

def main():
    img = cv2.imread('2.jpg', 0)
    _, thresh = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)
    # erosion
    kernel = np.ones((11, 11), np.uint8)
    erosion = cv2.erode(thresh, kernel, iterations=1)
    # dilation
    dilation = cv2.dilate(thresh, kernel, iterations=1)
    # Opening
    opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)
    # Closing
    closing = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
    # Display Images
    plt.subplot(321), plt.imshow(img, cmap = 'gray'), plt.axis('off')
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(322), plt.imshow(thresh, cmap = 'gray')
    plt.title('Binary Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(323), plt.imshow(erosion, cmap = 'gray')
    plt.title('Eroded Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(324), plt.imshow(dilation, cmap = 'gray')
    plt.title('Dilated Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(325), plt.imshow(opening, cmap = 'gray')
    plt.title('Opening Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(326), plt.imshow(closing, cmap = 'gray')
    plt.title('Closing Image'), plt.xticks([]), plt.yticks([])
    plt.show()

if __name__ == '__main__':
    main()
```

Output:

Binary Image



Original Image



Dilated Image



Eroded Image



Closing Image



Opening Image



Conclusion: Here, we learnt about different image morphology techniques used in image processing.