Digital Image Processing in Python Applications Book

2024

Introduction

This is an introductory book for digital image processing in Python. The main purpose of this book is to help students understanding the main concepts of image processing field and how to write the appropriate code for accomplishing the required task using the different image processing techniques for contrast manipulation and noise removing for obtaining an image with good quality suitable for humans or machines perception.

At first, you have to install the required software package and libraries for applying the digital image processing techniques which you study in this course.

Install Python on your computer

Go to the link below to download python installation file
https://www.python.org/downloads/

after downloading, install Python on your computer

open windows *cmd* and write
pip --version to check that python has been installed.

Installing some image processing libraries in Python

In Python, there are many libraries that we can use for image processing. The ones we are going to use are: NumPy, SciPy, scikit-image, PIL (Pillow), OpenCV, scikit-learn, SimpleITK, and Matplotlib.

The matplotliblibrary will primarily be used for display purposes, whereas numpy will be used for storing an image. The scikit-learn library will be used for building machine-learning models for image processing, and scipy will be used mainly for image enhancements.
The scikitimage, mahotas, and opencv libraries will be used for different image processing algorithms.

The following code block shows how the libraries that we are going to use can be downloaded and installed with pip from a Python prompt (interactive mode):

Open windows command prompt *cmd* then write

```
>>> pip install numpy
>>> pip install scipy
>>> pip install scikit-image
>>> pip install scikit-learn
>>> pip install pillow
>>> pip install SimpleITK
>>> pip install opencv-python
>>> pip install matplotlib
```

For upgrading Python

```
python -m pip install --upgrade pip
```

Installing Jupyter Notebook

We are going to use **Jupyter** notebooks to write our Python code. So, we need to install the `jupyter` package first from a Python prompt with `>>> pip install jupyter`, and then launch the Jupyter Notebook app in the browser using `>>> jupyter notebook`. From there, we can create new Python notebooks and choose a kernel. If we use Anaconda, we do not need to install Jupyter explicitly; the latest Anaconda distribution comes with Jupyter.

In *cmd* write

> jupyter notebook

Create your own folder in jupyter to be used in your exercises

You are now ready to write your first program in Python.

Introduction to Image Processing with Python

Image processing refers to various techniques that allow computers to understand and modify digital images. It involves analyzing pixel information to perform operations like identifying objects, detecting edges, adjusting brightness/contrast, applying filters, recognizing text, etc.

Python is a popular language for image processing due to its extensive libraries, simple syntax, and active developer community. Key libraries like OpenCV, PIL/Pillow, scikit-image, and more enable you to work with images in Python.

Understanding the Basics of Image Processing

Image processing relies on various types of operations for. These operations include:

- **Image acquisition:** Capturing or importing images via cameras, scanners etc.
- **Reading/showing an image:** reading an image file to be stored into a variable and showing this variable as an image on a window.
- **Intensity transformation:** transferring pixels' intensity values according to the used transformation function (negative, threshold, piecewise, etc.)
- **Affine transformations:** transferring pixels intensity values according to the function such as (resizing, rotation, translation, shearing,…etc.)
- **Histogram processing:** Identifying an image histogram and handling this histogram for contrast manipulation.
- **Image filtering in spatial domain:** applying smoothing and sharpening filters on an image in a spatial domain.

- **Image filtering in frequency domain:** applying different types of filters (lowpass, highpass, bandpass, and bandstop) on an image in a frequency domain.

The Advantages of Python in Image Processing

Python is a preferred language for image processing due to:

- **Extensive libraries** like OpenCV, PIL/Pillow, scikit-image etc. offering specialized functionality.
- **Simple and readable code** thanks to its clean syntax. Easy for beginners to adopt.
- Vibrant **developer community** providing abundant code examples and troubleshooting support.
- **Interoperability** with languages like C++ for performance-critical operations.
- **Rapid prototyping** enabled by Python's interpreted nature.

Overview of Python Image Processing Libraries

Some key image processing libraries in Python include:

- **OpenCV:** Comprehensive library with over 2500 algorithms ranging from facial recognition to shape analysis.
- **PIL/Pillow:** Offers basic image handling and processing functionality.
- **scikit-image:** Implements algorithms for segmentation, filtering, feature detection etc.
- **Mahotas:** Specialized library for computer vision operations.
- **SimpleCV:** Provides an easy interface to OpenCV for rapid prototyping.

With these mature libraries, Python makes an excellent choice for developing image processing and computer vision applications.

Types of digital images:

1- Binary: each pixel is just black or white so we need only one bit per pixel

and may be suitable for text, fingerprints, and architectural plans.

2- Gray scale: each pixel is a shade of gray, normally from 0 (black) to 255 (white). This range means that each pixel can be represented by 8 bits.

3- True color or RGB: each pixel has an amount of red, green, and blue colors. Each color is represented by 8 bits so the total number of bits is 24 for each pixel. Each image may has 2553=16,777,216 different possible color.

4- Indexed: most color images have only a small subset of the more than 16 million possible colors. For convenience of storage and file handling the image has an associated color map, or color palette, which is a list of all colors used in that image. Each pixel has a value that does not give its color but an index to the color in the map.

Consider a 512 x 512 binary image so its size will be

512 x 512 x 1= 262,144 bits

= ……. kb

a gray scale image of the same size will be

512 x 512 x 1 = 262,144 bytes

= …….kb

a color image of the same size will be

512 x 512 x 3 = 262,144 bytes

= ……. kb

Let's practice writing some python code for image processing

Task (1): Load and show an image

The process of images loading consists of multiple parts

1. Locating and reading the file bytes
2. decoding those bytes into the matrix format used for image manipulation

Images are encoded for efficiency in storage and network transfer some algorithm do better compression than others for example JPEG

```
# Python program to read image using OpenCV

# importing OpenCV(cv2) module
import cv2
import numpy as np
# Save image in set directory
# Read RGB image
img = cv2.imread('g4g.png',1) # Replace with your image file
# 0 --> for gray scale images
# 1 --> for colored images
#Note that OpenCV loads color image files in BGR order, not RGB.
#image = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#brighter=np.clip(img*1.5,0,255).astytpe(np.uint8)
# Output img with window name as 'image'
cv2.imshow('image', image)

# Maintain output window until
# user presses a key
cv2.waitKey(0)

# Destroying present windows on screen
cv2.destroyAllWindows()
```

---

1. **Read an image:** This line uses the cv2. imread function to read an image file named `'g4g.png'` and load it into memory. The result is stored in the variable image.

2. **Display the original image:** The ‘cv2.imshow’ function is employed to display the image in a graphical window. The window’s title is set to ‘Original Image’, and the image data comes from the “image” variable.

3. **Wait for a key press:** The cv2 waitKey (0) • line pauses the program, waiting for a key press. The argument “O’ means it will wait indefinitely until any key is pressed. This is commonly used to keep the image window open until the user decides to close it by pressing a key.

Put your running code and the output window below:

Task (2): You can do the same job of reading and showing an image using different libraries in python as follows:

```
# Python program to read
# image using matplotlib

# importing matplotlib modules
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Read Images
img = mpimg.imread('g4g.png') # Replace with your image file

# Output Images
plt.imshow(img)
```

```
# Python program to read
# image using PIL module

# importing PIL
from PIL import Image

# Read image
img = Image.open('g4g.png')

# Output Images
img.show()

# prints format of image
print(img.format)

# prints mode of image
print(img.mode)
```

Put your running code and the output window below:

Task(3): Get image size (width, height) with Python, OpenCV, Pillow (PIL).

You can obtain the image size as a tuple using the `shape` attribute of `ndarray` in OpenCV and the `size` attribute of `PIL.Image` in Pillow (PIL). It's important to note that the order of width and height is different in OpenCV and Pillow (PIL).

- OpenCV: Get image size (width, height) with `ndarray.shape`
  - For color images
  - For grayscale (monochrome) images
- Pillow (PIL): Get image size (width, height) with `size`, `width`, `height`

OpenCV treats an image as a NumPy array `ndarray`. The image size (width, height) can be obtained using the `shape` attribute, which returns a tuple of dimensions.

For color images

For color images, the `ndarray` is a 3D array with dimensions `(height, width, 3)`.

```
import cv2

im = cv2.imread('data/src/lena.jpg') # Replace with your image file

print(type(im))
# <class 'numpy.ndarray'>

print(im.shape)
print(type(im.shape))
# (225, 400, 3)
# <class 'tuple'>
```

To assign each value to a variable, unpack the tuple as follows:

- Unpack a tuple and list in Python

```
h, w, c = im.shape
print('width:  ', w)
print('height: ', h)
print('channel:', c)
```

```
# width:   400
# height:  225
# channel: 3
```

When unpacking a tuple, it is a common convention to assign unused values to _. In the following example, the number of colors (or channels) is not used:

```
h, w, _ = im.shape
print('width: ', w)
print('height:', h)
# width:  400
# height: 225
```

Of course, you can also directly access them by index.

```
print('width: ', im.shape[1])
print('height:', im.shape[0])
# width:  400
# height: 225
```

For grayscale (monochrome) images

For grayscale images, the `ndarray` is a 2D array with dimensions `(height, width)`.

```
import cv2

im_gray = cv2.imread('data/src/lena.jpg', cv2.IMREAD_GRAYSCALE)
# Replace with your image file
print(im_gray.shape)
print(type(im_gray.shape))
# (225, 400)
# <class 'tuple'>
```

The procedure for grayscale images is essentially the same as for color images:

```
h, w = im_gray.shape
print('width: ', w)
print('height:', h)
# width:  400
# height: 225

print('width: ', im_gray.shape[1])
print('height:', im_gray.shape[0])
# width:  400
# height: 225
```

To assign width and height to variables, the following approach works for both color and grayscale images.

```
h, w = im.shape[0], im.shape[1]
print('width: ', w)
print('height:', h)
# width:  400
# height: 225
```

A `PIL.Image` object, obtained by reading an image using Pillow (PIL), has the `size`, `width`, and `height` attributes.

The `size` attribute returns a `(width, height)` tuple.

```
from PIL import Image

im = Image.open('data/src/lena.jpg') # Replace with your image file

print(im.size)
print(type(im.size))
# (400, 225)
# <class 'tuple'>
```



```
w, h = im.size
print('width: ', w)
print('height:', h)
# width:  400
# height: 225
```

You can also get the width and height using the `width` and `height` attributes:

```
print('width: ', im.width)
print('height:', im.height)
# width:  400
# height: 225
```

The process is the same for grayscale (monochrome) images.

The data type of the image is uint8, i.e., 8 bit unsigned int (numbers from 0 to 255) as indicated by img.dtype. Minimum and maximum number in the matrix can be checked using the min and max functions. Total number of pixels can be obtained by img.size

print(type(image.shape))

```
print('image.shape', image.shape)
print('image.dtype', image.dtype) # 8 bit unsigned int (numbers from 0 to 255) as indicated by
img.dtype.
print('image.min', image.min())
print('image.max', image.max())
print('image.size', image.size)
```

Put your running code and the output window below:

Task (4): Image cropping

Image cropping is a way of photo editing that involves removing a portion of an image to emphasize a subject, change the ***aspect ratio***, or improve the framing. You can then reframe a subject or direct the viewer's attention to a certain part of the photo. Here are the key terms that relate to image cropping:

- **Crop rectangle** — The region, defined by four coordinates, to which to crop the image. The cropping operation removes all the details outside the crop rectangle, delivering a modified image of the crop rectangle's size.
- **Aspect ratio** — The ratio of the image's width to its height. This ratio is denoted by two numbers separated by a colon, for example, 4:3 or four-to-three.
- **Pixels** — The grid of vertical and horizontal pixels of which digital photos are composed. Cropping cuts out certain sections of the image, reducing the number of pixels and shrinking the image size.

How to Crop Images in Python with Pillow

The Pillow library offers an easy way through the `crop()` function for cropping images in Python. Here's syntax:

```
Image.crop(box=None)
```

The `box` parameter accepts a tuple with four values of the four coordinates for the crop rectangle: left, upper, right, and lower. The crop rectangle is drawn within the image, which depicts the part of the image you want to crop. `Image.crop` returns an `image` object.

To crop an image with Pillow:

1. Import Pillow's Image class:

```
from PIL import Image
```

2. Load an image from the file system and, with the `Image.open()` class, convert the image into an instance of the `Image` class. Adding `Image.show()` displays the image in an external viewer.

```
3. img = Image.open('./myimage.png')
```

```
img.show()
```

4. Crop the image. Assuming that the image `myimage.png` is 1,000×1,000 pixels, crop it to a square of 500×500 pixels at the center of the image:

```
5. box = (250, 250, 750, 750)
```

```
img2 = img.crop(box)
```

6. Save the cropped image, a Python object called `img2`, to the file system:

```
7. img2.save('myimage_cropped.jpg')
```

```
img2.show()
```

Note the following issues when cropping images with Pillow:

- ***Content-aware cropping***. Most cropping operations depend on the context. For example, you don't want to remove important parts of the image, but that's difficult to ensure programmatically because the crop() function is not sensitive to the image content.
- **Cropping and resizing.** Oftentimes, you must resize and crop an image at the same time. Even though you can resize images in Python with a similar technique, combining cropping and resizing can get tricky, let alone that it's challenging to generate the exact image you need for your design.

Crop an Image in Python With OpenCV

Python OpenCV is a library with advanced computer-vision capabilities, in particular a set of functions for handling processing and transforming images. To import the OpenCV library into your program, type:

```
import cv2
```

Here's the syntax for image cropping:

```
image[start_x:end_x, start_y:end_y]
```

Interestingly, this syntax slices the `image[]` in the form of an array by passing the start and end index x and y coordinates for each segment. The portion of the image between the start and end coordinates is returned as a truncated array object.

For example:

```
import cv2
image = cv2.imread(r"C:\file\path\imagename.png")# Replace with your image file
y=0
x=0
h=300
w=510
crop_image = image[x:w, y:h]
cv2.imshow("Cropped", crop_image)
cv2.waitKey(0)
```

The statement `cv2.imread(r"image path")` opens the image in read-only mode. The `cv2.imshow()` function near the bottom displays the cropped image.

Put your running code and the output window below:

Task (5): Arithmetic operations on images

Arithmetic operations on images are where you take two images, do calculations on their pixel values, and get a new resulting image that is a combination of the two. Arithmetic Operations like Addition, Subtraction, and Bitwise Operations (AND, OR, NOT, XOR) can be applied to the input images. These operations can be helpful in enhancing the properties of the input images.

Addition operation

The addition of two images is performed straightforwardly in a single pass. The output pixel values are given by:

Q(i,j) = P1(i,j) + P2(i,j)

Or if it is simply desired to add a constant value C to a single image then:

Q(i,j) = P1(i,j) + C

Addition is when you add the corresponding pixel values of the images to each other. Each pixel value is a number between 0 and 255, so if the sum of the two colors becomes higher than 255, it has to be truncated to 255 again, by taking the minimum of the result and 255.

In Opencv, we can add two images by using function **cv2.add()**. This directly adds up image pixels in the two images.

```
Syntax: cv2.add(img1, img2)
```

In the situations where a weight is required to be assigned for each image. So, we use cv2.addweighted(). Remember, both images should be of equal size and depth.

Syntax: cv2.addWeighted(img1, wt1, img2, wt2, gammaValue)
Parameters:
img1: First Input Image array(Single-channel, 8-bit or floating-point)
wt1: Weight of the first input image elements to be applied to the final image
img2: Second Input Image array(Single-channel, 8-bit or floating-point)
wt2: Weight of the second input image elements to be applied to the final image
gammaValue: Measurement of light

Consider the two images of Jerry and Tom, the result of addition is shown below.

```
tom=cv2.imread('jj.png')
tom=cv2.cvtColor(tom, cv2.COLOR_BGR2RGB)
jerry=cv2.imread('tt.png')
jerry=cv2.cvtColor(jerry, cv2.COLOR_BGR2RGB)
tANDJ=cv2.add(tom,jerry)
# cv2_imshow(cv2.cvtColor(tANDJ, cv2.COLOR_RGB2BGR))
plt.figure(figsize=(20, 20))
plt.subplot(1,3,1), plt.imshow(tom)
plt.subplot(1,3,2), plt.imshow(jerry)
plt.subplot(1,3,3), plt.imshow(tANDJ)
plt.show()
```

Put your running code and the result of addition:

Image Subtraction

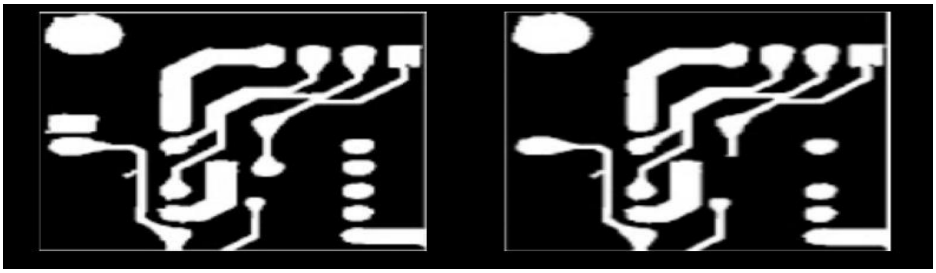
Subtraction works in a similar way, but now you have to truncate negative results to 0.

The subtraction of two images is used for example to detect changes.

```
img1_sub=cv2.imread('/content/img1_sub.png',0)
img2_sub=cv2.imread('/content/img2_sub.png',0)
changes=cv2.subtract (img1_sub,img2_sub)

# Difference is almost the same as subtract, only now instead of truncating negative values
# , you take their absolute value.
#Image subtraction is most beneficial in the area of medical image processing called mask mode radiography.

plt.figure(figsize=(20, 20))
plt.subplot(1,3,1), plt.imshow(img1_sub,cmap='gray')
plt.subplot(1,3,2), plt.imshow(img2_sub,cmap='gray')
plt.subplot(1,3,3), plt.imshow(changes,cmap='gray')
plt.show()
```



Put your running code and the result of subtraction:

```
lena=cv2.imread('lena_gray_256.tif',0)
# lena=cv2.cvtColor(lena, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(20, 20))
plt.subplot(1,3,1), plt.imshow(lena,cmap='gray')
plt.subplot(1,3,2), plt.imshow(cv2.subtract(lena,128),cmap='gray')
plt.subplot(1,3,3), plt.imshow(lena-128,cmap='gray')
plt.show()
```

Put your running code and the result:

Image Multiplication

Image multiplication is used to increase the average gray level of the image by multiplying with a constant (contrast).

It is used for masking operations. Masking is used in Image Processing to output the Region of Interest, or simply the part of the image that we are interested in.

So, let's get started with masking!

The process of masking images

We have three steps in masking.

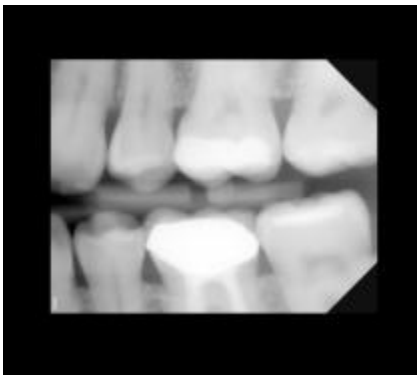
1. Creating a black canvas with the same dimensions as the image, and naming it as mask.
2. Changing the values of the mask by drawing any figure in the image and providing it with a white color.
3. Performing the multiplication operation on the image with the mask.

NOTE : To multiply RGB images, don't multiply the color component values from 0-255 with each other, then the maximum value would be 255 * 255 = 65025, and with such a big color value, you can't do much. Instead, convert the values to floating point numbers between 0 and 1, and multiply those. The result will then also always be between 0 and 1. After multiplication, multiply it with 255 again.

```
# Test on the X-ray dental image
xray = cv2.imread('/content/dental_xray.jpg',0)

mask_xray = np.zeros(xray.shape, dtype="uint8")
mask_xray=cv2.rectangle(mask_xray, (400,200), (650,700), 1, -1)

 plt.figure(figsize=(10, 10))
plt.subplot(131), plt.imshow(xray, cmap='gray')
plt.subplot(132), plt.imshow(mask_xray, cmap='gray')
plt.subplot(133),plt.imshow(cv2.multiply(xray, mask_xray), cmap='gray')
plt.show()
```



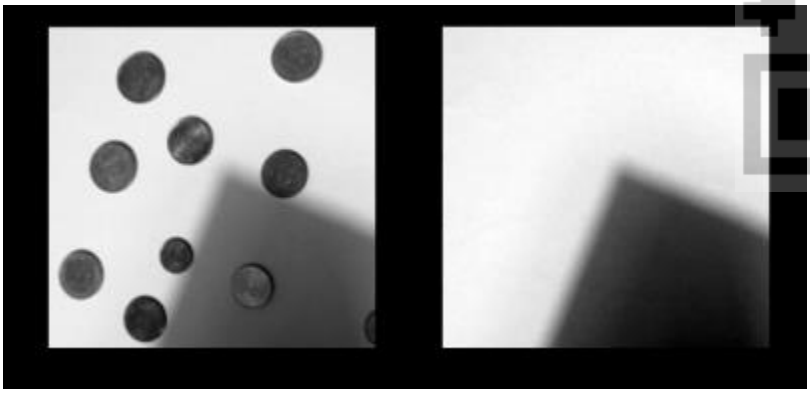
Put your running code and the result:

Image division

The division of two images is used to correct non-homogeneous illumination. For example, consider the following two images which will be used to illustrate the removal of shadow.

```
img1 = cv2.imread('/content/img1_dev.png',0)
img2 = cv2.imread('/content/img2_dev.png',0)

plt.figure(figsize=(10, 10))
plt.subplot(131), plt.imshow(img1, cmap='gray')
plt.subplot(132), plt.imshow(img2, cmap='gray')
plt.subplot(133),plt.imshow(img1/img2, cmap='gray')
plt.show()
```



Put your running code and the result:

Logical Operations

Logical operations are done on pixel-by-pixel basis.

The AND and OR operations are used for selecting segments in an image.

This masking operation is referred as Region of Interest processing.

```
# creating a square of zeros using a variable
rectangle = np.zeros((300, 300), dtype="uint8")
cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)

# creating a circle of zeros using a variable
circle = np.zeros((300, 300), dtype="uint8")
cv2.circle(circle, (150, 150), 150, 255, -1)
plt.figure(figsize=(20, 20))
```

```
plt.subplot(2,3,1).set_title("Rectangle "),plt.imshow(rectangle,cmap='gray')
plt.subplot(2,3,2).set_title("Circle "), plt.imshow(circle,cmap='gray')
plt.subplot(2,3,3).set_title("And "), plt.imshow(cv2.bitwise_and(rectangle,
circle),cmap='gray')
plt.subplot(2,3,4).set_title("OR "), plt.imshow(cv2.bitwise_or(rectangle,
circle),cmap='gray')
plt.subplot(2,3,5).set_title("XOR "), plt.imshow(cv2.bitwise_xor(rectangle,
circle),cmap='gray')
plt.subplot(2,3,6).set_title("NOT "), plt.imshow(cv2.bitwise_not(circle),cmap='gray')

plt.show()
```

Put your running code and the result:

Task (6): Intensity Transformation

Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image. The following are commonly used intensity transformations:

1. **Image Negatives (Linear)**
2. **Log Transformations**
3. **Power-Law (Gamma) Transformations**
4. **Piecewise-Linear Transformation Functions**

Negative Transformation

The negative transformation is achieved by taking the complement of the original pixel intensity values. If the original pixel intensity is denoted by "r," the corresponding negative intensity, denoted by "s," can be calculated as:

s = L - 1 - r

Here, "L" represents the maximum intensity level in the image, often 255 in the case of 8-bit images. This equation subtracts the original intensity from the maximum possible intensity, effectively inverting the pixel values. For example, if the original intensity is 100 in an 8-bit image (L = 255), the negative intensity will be (255–1–100) = 154.

```
import cv2
import matplotlib.pyplot as plt

# Load the original image
original_image = cv2.imread("owl-8285565_1280.png")  # Replace with your
image file

# Check if the image was loaded successfully

# Perform the negative transformation
negative_image = 256 - 1 - original_image

# Display the original and negative images side by side
plt.figure(figsize=(10, 5))  # Create a figure with a specified size
plt.subplot(1, 2, 1)  # Subplot for the original image
plt.imshow(cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB))
plt.title("Original Image")
plt.axis('off')  # Turn off axis labels

plt.subplot(1, 2, 2)  # Subplot for the negative image
plt.imshow(cv2.cvtColor(negative_image, cv2.COLOR_BGR2RGB))
```

```
plt.title("Negative Image")
plt.axis('off')

plt.show()  # Show the figure with both images
```

Put your running code and the result:

Log Transformation (Logarithm Function)

The transformation equation is given by:

$$S=c.\log(1+R)$$

- *S represents the transformed pixel value.*
- *R is the original pixel value.*
- *c is a constant that adjusts the degree of enhancement.*
- *The logarithm function compresses high-intensity values and stretches low-intensity values.*

Inverse-Log Transformation (Exponential Function)

The inverse-log transformation is the reverse operation, aimed at expanding the range of brighter pixels. This is especially useful for images with overexposed regions. The transformation equation is:

$$S = e^{R/C} - 1$$

- *S represents the transformed pixel value.*
- *R is the original pixel value.*
- *c is a constant that controls the degree of enhancement.*
- *The exponential function stretches high-intensity values.*

```
import cv2
import numpy as np

# Load an image
image = cv2.imread('owl-8285565_1280.png', cv2.IMREAD_GRAYSCALE)
# Log Transformation
c = 45
log_transformed = c * (np.log(image + 1))

# Convert to 8-bit unsigned integer format
log_transformed = np.uint8(log_transformed)

# Inverse-Log Transformation
inverse_log_transformed = c * (np.exp(log_transformed / c) - 1)

# Convert to 8-bit unsigned integer format
inverse_log_transformed = np.uint8(inverse_log_transformed)
# Display the original and negative images side by side
plt.figure(figsize=(10, 5))  # Create a figure with a specified size
plt.subplot(1, 3, 1)  # Subplot for the original image
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Original Image")
plt.axis('off')  # Turn off axis labels

plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(log_transformed, cv2.COLOR_BGR2RGB))
```

```
plt.title("log_transformed")
plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(inverse_log_transformed, cv2.COLOR_BGR2RGB))
plt.title("inverse_log_transformed")
plt.axis('off')
plt.show()
```

Put your running code and the result:

Power-law transformations

The basic idea behind power-law transformations is to raise the pixel values of an image to a certain power (exponent) in order to adjust the image's overall brightness and contrast. The general form of a power-law transformation is:

$$S=c.\ R^{\gamma}$$

- *S is the output pixel value (transformed value).*
- *R is the input pixel value (original value).*

- *γ is the exponent, which controls the degree of transformation.*
- *c is a constant that scales the result to fit within the desired intensity range.*

Gamma Correction: When γ>1, it is known as gamma correction, and it brightens the image. When γ<1, it darkens the image.

```
import cv2
import numpy as np

# Read an image
image = cv2.imread('test2.png', cv2.IMREAD_GRAYSCALE)

# Apply gamma correction (e.g., gamma = 1.5)
gamma = 2
gamma2=4
adjusted_image = np.power(image / 255.0, gamma) * 255.0
adjusted_image = adjusted_image.astype(np.uint8)
adjusted_image2 = np.power(image / 255.0, gamma2) * 255.0
adjusted_image2 = adjusted_image2.astype(np.uint8)
plt.figure(figsize=(10, 5))  # Create a figure with a specified size
plt.subplot(1, 3, 2)  # Subplot for the original image
plt.imshow(cv2.cvtColor(adjusted_image, cv2.COLOR_BGR2RGB))
plt.title("adjusted_image gamma =2")

plt.subplot(1, 3, 1)  # Subplot for the original image
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Original Image")

plt.subplot(1, 3, 3)  # Subplot for the original image
plt.imshow(cv2.cvtColor(adjusted_image2, cv2.COLOR_BGR2RGB))
plt.title("Original Image gamma = 4")
plt.axis('off')
```

Put your running code and the result:

Affine Transformation

Image Resizing

Scaling operations increase or reduce the size of an image.

- **The cv2.resize() function** is used to resize an image in OpenCV. It takes the following arguments:

  cv2.resize(src, dsize,interpolation)

  Here,

  src :The image to be resized.

  dsize :The desired width and height of the resized image.

  interpolation:The interpolation method to be used.
- When the image is resized, the **interpolation** method defines how the new pixels are computed. There are several interpolation techniques, each of which has its own quality vs. speed trade-offs.
- **It is important to note that resizing an image can reduce its quality**. This is because the new pixels are calculated by interpolating between the existing pixels, and this can introduce some blurring.

```
# Import the necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('Ganeshji.webp')

# Convert BGR image to RGB
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Define the scale factor
# Increase the size by 3 times
scale_factor_1 = 3.0
# Decrease the size by 3 times
scale_factor_2 = 1/3.0

# Get the original image dimensions
height, width = image_rgb.shape[:2]

# Calculate the new image dimensions
new_height = int(height * scale_factor_1)
new_width = int(width * scale_factor_1)
```

```
# Resize the image
zoomed_image = cv2.resize(src =image_rgb,
                          dsize=(new_width, new_height),
                          interpolation=cv2.INTER_CUBIC)

# Calculate the new image dimensions
new_height1 = int(height * scale_factor_2)
new_width1 = int(width * scale_factor_2)

# Scaled image
scaled_image = cv2.resize(src= image_rgb,
                          dsize =(new_width1, new_height1),
                          interpolation=cv2.INTER_AREA)

# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(10, 4))

# Plot the original image
axs[0].imshow(image_rgb)
axs[0].set_title('Original Image Shape:'+str(image_rgb.shape))

# Plot the Zoomed Image
axs[1].imshow(zoomed_image)
axs[1].set_title('Zoomed Image Shape:'+str(zoomed_image.shape))

# Plot the Scaled Image
axs[2].imshow(scaled_image)
axs[2].set_title('Scaled Image Shape:'+str(scaled_image.shape))

# Remove ticks from the subplots
for ax in axs:
    ax.set_xticks([])
    ax.set_yticks([])

# Display the subplots
plt.tight_layout()
plt.show()
```

Put your running code and the result:

Image Rotation

Images can be rotated to any degree clockwise or otherwise. We just need to define rotation matrix listing rotation point, degree of rotation and the scaling factor.

- **The cv2.getRotationMatrix2D() function** is used to create a rotation matrix for an image. It takes the following arguments:
  - The center of rotation for the image.
  - The angle of rotation in degrees.
  - The scale factor.
- **The cv2.warpAffine() function** is used to apply a transformation matrix to an image. It takes the following arguments:
  - The image to be transformed.
  - The transformation matrix.
  - The output image size.
- **The rotation angle can be positive or negative**. A positive angle rotates the image clockwise, while a negative angle rotates the image counterclockwise.

- **The scale factor can be used to scale the image up or down**. A scale factor of 1 will keep the image the same size, while a scale factor of 2 will double the size of the image.

```
# Import the necessary Libraries
import cv2
import matplotlib.pyplot as plt

# Read image from disk.
img = cv2.imread('Ganesh.jpg')

# Convert BGR image to RGB
image_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Image rotation parameter
center = (image_rgb.shape[1] // 2, image_rgb.shape[0] // 2)
angle = 30
scale = 1

# getRotationMatrix2D creates a matrix needed for transformation.
rotation_matrix = cv2.getRotationMatrix2D(center, angle, scale)

# We want matrix for rotation w.r.t center to 30 degree without scaling.
rotated_image = cv2.warpAffine(image_rgb, rotation_matrix, (image.shape[1], imag ape[

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(7, 4))

# Plot the original image
axs[0].imshow(image_rgb)
axs[0].set_title('Original Image')

# Plot the Rotated image
axs[1].imshow(rotated_image)
axs[1].set_title('Image Rotation')

# Remove ticks from the subplots
for ax in axs:
    ax.set_xticks([])
    ax.set_yticks([])

# Display the subplots
plt.tight_layout()
plt.show()
```

Put your running code and the result:

Image Translation

Translating an image means shifting it within a given frame of reference that can be along the x-axis and y-axis.

- **To translate an image using OpenCV**, we need to create a transformation matrix. This matrix is a 2×3 matrix that specifies the amount of translation in each direction.
- **The cv2.warpAffine() function** is used to apply a transformation matrix to an image. It takes the following arguments:
  - The image to be transformed.
  - The transformation matrix.
  - The output image size.
- **The translation parameters** are specified in the transformation matrix as the tx and ty elements. The tx element specifies the amount of translation in the x-axis, while the ty element specifies the amount of translation in the y-axis.

```python
# Import the necessary Libraries
import cv2
import matplotlib.pyplot as plt

# Read image from disk.
img = cv2.imread('Ganesh.jpg')
# Convert BGR image to RGB
image_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

width = image_rgb.shape[1]
height = image_rgb.shape[0]

tx = 100
ty = 70

# Translation matrix
translation_matrix = np.array([[1, 0, tx], [0, 1, ty]], dtype=np.float32)
# warpAffine does appropriate shifting given the Translation matrix.
translated_image = cv2.warpAffine(image_rgb, translation_matrix, (width, height)

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(7, 4))

# Plot the original image
axs[0].imshow(image_rgb)
axs[0].set_title('Original Image')

# Plot the transalted image
axs[1].imshow(translated_image)
axs[1].set_title('Image Translation')

# Remove ticks from the subplots
for ax in axs:
    ax.set_xticks([])
    ax.set_yticks([])

# Display the subplots
plt.tight_layout()
plt.show()
```

Put your running code and the result:

Image shearing

Image shearing is a geometric transformation that skews an image along one or both axes i.e x or y axis.

- **To shear an image using OpenCV**, we need to create a transformation matrix. This matrix is a 2×3 matrix that specifies the amount of shearing in each direction.
- **The cv2.warpAffine() function** is used to apply a transformation matrix to an image. It takes the following arguments:
    - The image to be transformed.
    - The transformation matrix.
    - The output image size.
- **The shearing parameters** are specified in the transformation matrix as the shearX shearY elements. The shearX element specifies the amount of shearing in the x-axis, while the shearY element specifies the amount of shearing in the y-axis.

```
# Import the necessary Libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```python
# Load the image
image = cv2.imread('Ganesh.jpg')

# Convert BGR image to RGB
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Image shape along X and Y
width = image_rgb.shape[1]
height = image_rgb.shape[0]

# Define the Shearing factor
shearX = -0.15
shearY = 0

# Define the Transformation matrix for shearing
transformation_matrix = np.array([[1, shearX, 0],
                                  [0, 1, shearY]], dtype=np.float32)
# Apply shearing
sheared_image = cv2.warpAffine(image_rgb, transformation_matrix, (width, height))

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(7, 4))

# Plot the original image
axs[0].imshow(image_rgb)
axs[0].set_title('Original Image')

# Plot the Sheared image
axs[1].imshow(sheared_image)
axs[1].set_title('Sheared image')

# Remove ticks from the subplots
for ax in axs:
    ax.set_xticks([])
    ax.set_yticks([])

# Display the subplots
plt.tight_layout()
plt.show()
```

Put your running code and the result:

Image Normalization

Image normalization is a process of scaling the pixel values in an image to a specific range. This is often done to improve the performance of image processing algorithms, as many algorithms work better when the pixel values are within a certain range.

- **In OpenCV**, the cv2.normalize() function is used to normalize an image. This function takes the following arguments:
    - The input image.
    - The output image.
    - The minimum and maximum values of the normalized image.
    - The normalization type.
    - The dtype of the output image.
- **The normalization type** specifies how the pixel values are scaled. There are several different normalization types available, each with its own trade-offs between accuracy and speed.
- **Image normalization** is a common preprocessing step in many image processing tasks. It can help to improve the performance of algorithms such as image classification, object detection, and image segmentation.

```python
# Import the necessary Libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('Ganesh.jpg')

# Convert BGR image to RGB
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Split the image into channels
b, g, r = cv2.split(image_rgb)

# Normalization parameter
min_value = 0
max_value = 1
norm_type = cv2.NORM_MINMAX

# Normalize each channel
b_normalized = cv2.normalize(b.astype('float'), None, min_value, max_value, norm_type)
g_normalized = cv2.normalize(g.astype('float'), None, min_value, max_value, norm_type)
r_normalized = cv2.normalize(r.astype('float'), None, min_value, max_value, norm_type)

# Merge the normalized channels back into an image
normalized_image = cv2.merge((b_normalized, g_normalized, r_normalized))
# Normalized image
print(normalized_image[:,:,0])

plt.imshow(normalized_image)
plt.xticks([])
plt.yticks([])
plt.title('Normalized Image')
plt.show()
```

Put your running code and the result:

Intensity level slicing

This technique is used to highlight a specific range of gray levels in a given image. Useful for highlighting features in an image

- It can be implemented in several ways, but the two basic themes are: transformation brightens the desired range of gray
  - One approach is to display a high value for all gray levels in the range of interest and a low value for all other gray levels.
  - The second approach, based on the levels but preserves Hanan Hardan gray levels unchanged.

```
# Load the image
img = cv2.imread('/content/Slicing_img.png',0)
# Find width and height of image
row, column = img.shape
# Create an zeros array to store the sliced image
img1 = np.zeros((row,column),dtype = 'uint8')
img2 = np.zeros((row,column),dtype = 'uint8')


# Specify the min and max range
min_range = 51
```

```
max_range = 140

# Loop over the input image and if pixel value lies in desired range set it to 255
otherwise set it to 0.
for i in range(row):
   for j in range(column):
      if img[i,j]>min_range and img[i,j]<max_range:
         img1[i,j] = 255
      else:
         img1[i,j] = 0


# Load the image
img = cv2.imread('/content/Slicing_img.png',0)
# Find width and height of image
row, column = img.shape
# Create an zeros array to store the sliced image
img2 = np.zeros((row,column),dtype = 'uint8')


# Specify the min and max range
min_range = 51
max_range = 140

for i in range(row):
   for j in range(column):
      if img[i,j]>min_range and img[i,j]<max_range:
         img2[i,j] = 255
      else:
         img2[i,j] = img[i,j]

fig=plt.figure(figsize=(20, 20))
plt.subplot(1,3,1), plt.imshow(img,cmap='gray')
plt.subplot(1,3,2), plt.imshow(img1,cmap='gray')
plt.subplot(1,3,3), plt.imshow(img2,cmap='gray')
```

Put your running code and the result:

Bit Plane Slicing

Pixels are digital numbers, each one composed of bits. Instead of highlighting gray-level range, we could highlight the contribution made by each bit. This method is useful and used in image compression.
In Bit-plane slicing, we divide the image into bit planes. This is done by first converting the pixel values in the binary form and then dividing it into bit planes. For simplicity let's take a 3x3, 3-bit as shown below. we know that the pixel values of 3 bit can take values between 0 to 7.

```
# Read the image in greyscale
img = cv2.imread('/content/dollar.jfif',0)

#Iterate over each pixel and change pixel value to binary using np.binary_repr() and store it in a list.
lst = []
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        lst.append(np.binary_repr(img[i][j] ,width=8)) # width = no. of bits
```

```
# We have a list of strings where each string represents binary pixel value. To extract
bit planes we need to iterate over the strings and store the characters corresponding
to bit planes into lists.
# Multiply with 2^(n-1) and reshape to reconstruct the bit image.
plane=[]
for i in range (7,-1,-1):
  plane.append( (np.array([int(pxl[i])* (2**(7-i)) for pxl in lst],dtype = np.uint8)
).reshape(img.shape[0],img.shape[1]))
# set up side-by-side image display
fig = plt.figure()
fig.set_figheight(20)
fig.set_figwidth(20)

# Higher order bits usually
# contain most of the significant visual information Lower order bits contain
# subtle details

for i in range(1,9):
   plt.subplot(4,2,i)
   plt.imshow(plane[i-1], cmap='gray')
```

Put your running code and the result:

Histogram Processing

Histogram is considered as a graph or plot which is related to frequency of pixels in an Gray Scale Image with pixel values (ranging from 0 to 255).

cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])

images : it is the source image of type uint8 or float32. it should be given in square brackets, ie, "[img]".

channels : it is also given in square brackets. It is the index of channel for which we calculate histogram. For example, if input is grayscale image, its value is [0]. For color image, you can pass [0], [1] or [2] to calculate histogram of blue, green or red channel respectively.

mask : mask image. To find histogram of full image, it is given as "None". But if you want to find histogram of particular region of image, you have to create a mask image for that and give it as mask. (I will show an example later.)

histSize : this represents our BIN count. Need to be given in square brackets. For full scale, we pass [256].

ranges : this is our RANGE. Normally, it is [0,256].


```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
import matplotlib.pyplot as plt

hist = cv2.calcHist([black],[0],None,[50],[0,256])

# different methods for displaying a histogram
plt.bar(range(50), hist.ravel())
plt.title('Histogram of the black image')
plt.xlabel('Gray values')
plt.ylabel('Frequency')

# Another method
hist,bins = np.histogram(black.ravel(),256,[0,256])
plt.plot(hist)

# Let's read two other images
high = cv2.imread('/content/hist_highkey.jpg')
low = cv2.imread('/content/hist_lowkey.jpg')
# show images
```

```
plt.subplot(121), plt.imshow(high)
plt.grid(False), plt.xticks([]), plt.yticks([])

plt.subplot(122), plt.imshow(low)
plt.grid(False), plt.xticks([]), plt.yticks([])
plt.show()
# Calculate histogram of both images for the last channel.
# Channels can differ from 0 to 2.
hist_high = cv2.calcHist([high],[2],None,[256],[0,256])
hist_low = cv2.calcHist([low],[2],None,[256],[0,256])

# Plot histograms
plt.subplot(121)
plt.plot(hist_high)

plt.subplot(122)
plt.plot(hist_low)

plt.show()
```

Put your running code and the result:

Histogram equalization

One usual method to stretch the intensity values of an image in order to make its histogram similar to the perfect histogram shape (uniformly distributed), is the *histogram equalization*. In this method, image histogram will be stretched with respect to its cumulative distribution function.

cv2.equalizeHist(src[, dst])
src : the only required argument is the original image to be equalized.

```
img=cv2.imread('/content/equl.png')
gray_img=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_eq = cv2.equalizeHist(gray_img)

grid = plt.GridSpec(3, 4, wspace=0.4, hspace=0.3)

plt.subplot(grid[:2, :2])
plt.imshow(img, cmap='gray')
plt.grid(False), plt.xticks([]), plt.yticks([])

plt.subplot(grid[:2, 2:])
plt.imshow(img_eq, cmap='gray')
plt.grid(False), plt.xticks([]), plt.yticks([])

plt.subplot(grid[2, :2])
plt.bar(range(256),
        cv2.calcHist([img],[0],None,[256],[0,256]).ravel())

plt.subplot(grid[2, 2:])
plt.bar(range(256),
        cv2.calcHist([img_eq],[0],None,[256],[0,256]).ravel())
```

Put your running code and the result:

However, while histogram equalization can be beneficial for many applications, it's not always the best method for every image. It can sometimes over-enhance noise or cause loss of detail in areas where the contrast was already adequate.

```
img2=cv2.imread('/content/noise.png')
gray_img2=cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

img_eq = cv2.equalizeHist(gray_img2)

grid = plt.GridSpec(3, 4, wspace=0.4, hspace=0.3)

plt.subplot(grid[:2, :2])
plt.imshow(img2, cmap='gray')
plt.grid(False), plt.xticks([]), plt.yticks([])

plt.subplot(grid[:2, 2:])
plt.imshow(img_eq, cmap='gray')
plt.grid(False), plt.xticks([]), plt.yticks([])

plt.subplot(grid[2, :2])
plt.bar(range(256),
        cv2.calcHist([img],[0],None,[256],[0,256]).ravel())

plt.subplot(grid[2, 2:])
plt.bar(range(256),
        cv2.calcHist([img_eq],[0],None,[256],[0,256]).ravel())
```

Put your running code and the result:

Histogram Matching

What is Histogram Matching

Histogram matching is the transformation of an image so that its histogram matches a specified histogram. In order to match the histogram of images A and B, we need to first equalize the histogram of both images. Then, we need to map each pixel of A to B using the equalized histograms. Then we modify each pixel of A based on that of B. Histogram matching may be used to balance detector responses. It can be used to equalise two pictures that were taken in the same place with the same local lighting (such as shadows), but with different sensors, atmospheric conditions, or global illumination.

match_histograms() method:

This method is used to modify the cumulative histogram of one picture to match the histogram of another. For each channel, the modification is made independently.

Syntax: *skimage.exposure.match_histograms(image, reference, *, channel_axis=None, multichannel=False)*

Parameters:

- ***image****: ndarray. This parameter is our input image it can be grayscale or a colour image.*
- ***reference****: reference image. reference image to match histograms.*
- ***channel_axis****: optional parameter. int or None. If None is specified, the picture will be presumed to be grayscale (single channel). if not, this argument specifies the array axis that corresponds to channels.*
- ***multichannel:*** *optional parameter. boolean value. Matching should be done independently for each channel. This option has been deprecated; instead, use the channel axis.*


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

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import colors


%matplotlib inline
from google.colab.patches import cv2_imshow

def load_image(number):
  img = cv2.imread(f"/content/original_images/img{number}.bmp")
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
  return gray

from skimage import exposure

src = load_image(3)
dst = load_image(4)
matched_src = exposure.match_histograms(src,dst)

# compare_matched_hist(src,dst,matched_src)
```

Put your running code and the result:

Image Smoothing | Blurring

Image blurring is the technique of reducing the detail of an image by averaging the pixel values in the neighborhood. This can be done to reduce noise, soften edges, or make it harder to identify a picture. In many image processing tasks, image blurring is a common preprocessing step. It is useful in the optimization of algorithms such as image classification, object identification, and image segmentation. In OpenCV, a variety of different blurring methods are available, each with a particular trade-off between blurring strength and speed.

Some of the most common blurring techniques include:

- Mean filter: This blurring technique uses the mean of the pixel values in a neighborhood to smooth out the image.

- **Gaussian blurring:** This is a popular blurring technique that uses a Gaussian kernel to smooth out the image.
- **Median blurring:** This blurring technique uses the median of the pixel values in a neighborhood to smooth out the image.

Mean Filter

The mean filter is used to blur an image in order to remove noise. It involves determining the mean of the pixel values within a n x n kernel. The pixel intensity of the center element is then replaced by the mean. This eliminates some of the noise in the image and smooths the edges of the image.
This is done by the function cv.blur() or cv.boxFilter().

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

img = cv2.imread('/content/early_1800.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

kernel = np.ones((3, 3), np.float32)
kernel=kernel/9
dst = cv2.filter2D(img, -1, kernel)
blur = cv2.blur(img, (3, 3))

fig=plt.figure(figsize=(20, 20))
plt.subplot(1,3,1), plt.imshow(img)
plt.subplot(1,3,2), plt.imshow(dst)
plt.subplot(1,3,3), plt.imshow(blur)
```

Put your running code and the result:

```
kernel = np.ones((5, 5), np.float32)
kernel=kernel/25
dst = cv2.filter2D(img,-1, kernel)
blur = cv2.blur(img, (5, 5))
fig=plt.figure(figsize=(20, 20))
plt.subplot(1,3,1), plt.imshow(img)
plt.subplot(1,3,2), plt.imshow(dst)
plt.subplot(1,3,3), plt.imshow(blur)
```

Put your running code and the result:

Gaussian Blurring

The Gaussian Smoothing Operator performs a *weighted average* of surrounding pixels based on the Gaussian distribution. It gives more weight at the central pixel, and less weight to the neighbors. It is done with the function, cv.GaussianBlur()

In [ ]:

```
gblur = cv2.GaussianBlur(img, (5, 5), 0)
plt.imshow(gblur)
```

Put your running code and the result:

Median Filter

The median filter calculates the median of the pixel intensities that surround the center pixel in a n x n kernel. The median then replaces the pixel intensity of the center pixel. The median filter does a better job of removing **salt and pepper noise** than the mean and Gaussian filters. It smooth pixels whose value differ significantly from its surrounding without affecting the other pixels.
the function **cv.medianBlur()** takes the median of all the pixels under the kernel area and the central element is replaced with this median value.

```
img=cv2.imread('/content/s_p_noise.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
blur = cv2.blur(img, (5, 5))
median = cv2.medianBlur(img, 5)

fig=plt.figure(figsize=(10, 10))
plt.subplot(1,3,1), plt.imshow(img)
plt.subplot(1,3,2), plt.imshow(blur)
plt.subplot(1,3,3), plt.imshow(median)
```

Put your running code and the result:

Sharpening

to **highlight fine detail** in an image or to enhance detail that has been blurred. As we know that blurring can be done in spatial domain by pixel averaging in a neighbors since *averaging is similar to integration* thus, we can guess that the *sharpening must be accomplished by spatial spatial differentiation*.

Sharpening Spatial Filter: It is also known as derivative filter. The purpose of the sharpening spatial filter is just the opposite of the smoothing spatial filter. Its main focus in on the removal of blurring and highlight the edges. It is based on the first and second order derivative.

First order derivative:

- Must be zero in flat segments.
- Must be non zero at the onset of a grey level step.
- Must be non zero along ramps.

First order derivative in 1-D is given by:

f' = f(x+1) - f(x)

Second order derivative:

- Must be zero in flat areas.
- Must be zero at the onset and end of a ramp.
- Must be zero along ramps. Second order derivative in 1-D is given by:

f'' = f(x+1) + f(x-1) - 2f(x)

Sobel Filter

```
import cv2
import matplotlib.pyplot as plt

# Read the original image
img = cv2.imread('/content/lena_gray_256.tif')
# converting because opencv uses BGR as default
RGB_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(RGB_img)

# converting to gray scale
gray = cv2.cvtColor(RGB_img, cv2.COLOR_BGR2GRAY)

# remove noise
img = cv2.GaussianBlur(gray,(3,3),0)

# convolute with sobel kernels
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)  # x
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)  # y

fig=plt.figure(figsize=(10, 10))
plt.subplot(1,3,1), plt.imshow(RGB_img), plt.xticks([]),plt.yticks([])
plt.subplot(1,3,2), plt.imshow(sobelx,cmap = 'gray'), plt.xticks([]),plt.yticks([])
plt.subplot(1,3,3), plt.imshow(sobely,cmap = 'gray'), plt.xticks([]),plt.yticks([])
```

Put your running code and the result:

Laplacian Filter

The Laplacian of an image highlights the areas of rapid changes in intensity and can thus be used for edge detection.

- Sharpening enhances the edges as well as noise associated so we need to make noise reduction before it or otherwise it will be highlighted as well.
- The Laplacian operator is implemented in OpenCV by the function **Laplacian**()

we have to apply just one but the thing to remember is that if we apply positive Laplacian operator on the image then we subtract the resultant image from the original image to get the sharpened image. Similarly, if we apply negative Laplacian operator then we have to add the resultant image onto original image to get the sharpened image.

```
img=cv2.imread('/content/moon.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

#using built in cv2 function
laplacian = cv2.Laplacian(img,cv2.CV_64F)
laplacian = np.uint8(np.absolute(laplacian))


img_new = img + laplacian

fig=plt.figure(figsize=(10, 10))
plt.subplot(1,3,1), plt.imshow(img), plt.xticks([]),plt.yticks([])
```

```
plt.subplot(1,3,2), plt.imshow(laplacian,cmap = 'gray'), plt.xticks([]),plt.yticks([])
plt.subplot(1,3,3), plt.imshow(img_new,cmap = 'gray'), plt.xticks([]),plt.yticks([])
```

Put your running code and the result:

```
blur1 = cv2.blur(img, (3, 3))
laplacian = cv2.Laplacian(blur1,cv2.CV_64F)
laplacian = np.uint8(np.absolute(laplacian))
```



```
img_new = img + laplacian

fig=plt.figure(figsize=(10, 10))
plt.subplot(1,3,1), plt.imshow(img), plt.xticks([]),plt.yticks([])
plt.subplot(1,3,2), plt.imshow(laplacian,cmap = 'gray'), plt.xticks([]),plt.yticks([])
plt.subplot(1,3,3), plt.imshow(img_new,cmap = 'gray'), plt.xticks([]),plt.yticks([])
```

Put your running code and the result:

Unsharp Masking and Highboost Filtering

A process that has been used for many years by the printing and publishing industry to sharpen images consists of subtracting an unsharp (smoothed) version of an image from the original image.This process, called unsharp masking, consists of the following steps:

1. Blur the original image.
2. Subtract the blurred image from the original (the resulting difference is called the mask.)
3. Add the mask to the original.

Original image - Smoothed image = Detailed image (Edges)

Original image + k* Detailed image (Edges) = Sharpened image

- **When k > 1, the process is referred to as highboost filtering**

```
img=cv2.imread('/content/dip_xe.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
blur= cv2.GaussianBlur(img, (25,25), 0)
mask=cv2.subtract(img,blur)

# fig=plt.figure(figsize=(10, 10))
plt.subplot(1,5,1), plt.imshow(img), plt.xticks([]),plt.yticks([])
plt.subplot(1,5,2), plt.imshow(blur,cmap = 'gray'), plt.xticks([]),plt.yticks([])
plt.subplot(1,5,3), plt.imshow(mask,cmap = 'gray'), plt.xticks([]),plt.yticks([])
plt.subplot(1,5,4), plt.imshow(cv2.add(img,mask),cmap = 'gray'),
plt.xticks([]),plt.yticks([])
plt.subplot(1,5,5), plt.imshow(cv2.add(img,5*mask),cmap = 'gray'),
plt.xticks([]),plt.yticks([])
```

Put your running code and the result:

Edge detection of Image

The process of image edge detection involves detecting sharp edges in the image. This edge detection is essential in the context of image recognition or object localization/detection. There are several algorithms for detecting edges due to its wide applicability.

In image processing and computer vision applications, Canny Edge Detection is a well-liked edge detection approach. In order to detect edges, the Canny edge detector first smoothes the image to reduce noise, then computes its gradient, and then applies a threshold to the gradient. The multi-stage Canny edge detection method includes the following steps:

- Gaussian smoothing: The image is smoothed using a Gaussian filter to remove noise.
- Gradient calculation: The gradient of the image is calculated using the Sobel operator.
- Non-maximum suppression: Non-maximum suppression is applied to the gradient image to remove spurious edges.
- Hysteresis thresholding: Hysteresis thresholding is applied to the gradient image to identify strong and weak edges.

The Canny edge detector is a powerful edge detection algorithm that can produce high-quality edge images. However, it can also be computationally expensive.

```
# Import the necessary Libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read image from disk.
img = cv2.imread('Ganesh.jpg')
# Convert BGR image to RGB
image_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Apply Canny edge detection
edges = cv2.Canny(image= image_rgb, threshold1=100, threshold2=700)

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(7, 4))

# Plot the original image
axs[0].imshow(image_rgb)
axs[0].set_title('Original Image')

# Plot the blurred image
axs[1].imshow(edges)
```

```
axs[1].set_title('Image edges')

# Remove ticks from the subplots
for ax in axs:
    ax.set_xticks([])
    ax.set_yticks([])

# Display the subplots
plt.tight_layout()
plt.show()
```

Put your running code and the result:

Morphological Image Processing

Morphological image processing is a set of image processing techniques based on the geometry of objects in an image. These procedures are commonly used to eliminate noise, separate objects, and detect edges in images.

Two of the most common morphological operations are:

- **Dilation**: This operation expands the boundaries of objects in an image.
- **Erosion**: This operation shrinks the boundaries of objects in an image.

Morphological procedures are often used in conjunction with other image processing methods like segmentation and edge detection.

```
# Import the necessary Libraries
import cv2
```

```
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('Ganesh.jpg')

# Convert BGR image to gray
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Create a structuring element
kernel = np.ones((3, 3), np.uint8)

# Perform dilation
dilated = cv2.dilate(image_gray, kernel, iterations=2)

# Perform erosion
eroded = cv2.erode(image_gray, kernel, iterations=2)

# Perform opening (erosion followed by dilation)
opening = cv2.morphologyEx(image_gray, cv2.MORPH_OPEN, kernel)

# Perform closing (dilation followed by erosion)
closing = cv2.morphologyEx(image_gray, cv2.MORPH_CLOSE, kernel)

# Create subplots
fig, axs = plt.subplots(2, 2, figsize=(7, 7))

# Plot the Dilated Image
axs[0,0].imshow(dilated, cmap='Greys')
axs[0,0].set_title('Dilated Image')
axs[0,0].set_xticks([])
axs[0,0].set_yticks([])

# Plot the Eroded Image
axs[0,1].imshow(eroded, cmap='Greys')
axs[0,1].set_title('Eroded Image')
axs[0,1].set_xticks([])
axs[0,1].set_yticks([])

# Plot the opening (erosion followed by dilation)
axs[1,0].imshow(opening, cmap='Greys')
axs[1,0].set_title('Opening')
axs[1,0].set_xticks([])
```

```
axs[1,0].set_yticks([])

# Plot the closing (dilation followed by erosion)
axs[1,1].imshow(closing, cmap='Greys')
axs[1,1].set_title('Closing')
axs[1,1].set_xticks([])
axs[1,1].set_yticks([])

# Display the subplots
plt.tight_layout()
plt.show()
```

Put your running code and the result:

Sheet(1)

1. Electromagnetic and acoustic are two different imaging modalities. The source energy is passed (or reflected) through (or by) the objects to be imaged. This energy is then senses and converted to a digital image.
    a) Mention some applications for each modality
    b) Choose one imaging application for each of the two modalities and give details about it.

2. Explain how the image shown below is typically produced

3. What are the two main application areas of digital image processing?

4. There are no clear-cut boundaries in the continuum from image processing at one end to computer vision at the other. However, one useful paradigm is to consider three types of computerized processes in this continuum: low-, mid-, and high-level processes. Explain.

5. The invention in the early 1970s of computerized axial tomography (CAT), also called computerized tomography (CT) for short, is one of the most important events in the application of image processing in medical diagnosis. Explain how the CAT image is constructed.

6. Describe how, in general, digital images are formed.

7. Mention some example for gamma-ray imaging

8. Show how the X-rays are generated for the purpose of medical imaging and mention one example of the medical x-ray imaging.

9. Explain the fluorescence microscopy as one example in Ultraviolet light imaging.

10. Mention some application for the imaging in the visible and infrared bands

11. One common application of microwave imaging is the radar systems. Explain how are radar image generated and mentions some application areas.

12. Explain how images are produced using electron microscopy.

13. Using block diagram show the main steps and components used in a general purpose digital image processing system.

14. Explain the categories of digital storage for image processing

Sheet (2)

1) Thinking purely in geometric terms, estimate the diameter of the smallest printed dot that the eye can discern if the page on which the dot is printed is 0.2 m away from the eyes. Assume for simplicity that the visual system ceases to detect the dot when the image of the dot on the fovea becomes smaller than the diameter of one receptor (cone) in that area of the retina. Assume further that the fovea can be modeled as a square array of dimensions 1.5 mm X 1.5 mm having 580x580 cones. Assume also that cones and spaces between the cones are of the same size and are distributed uniformly throughout this array.

2) Explain why:

a) Objects that appear brightly colored in day light when seen by moon light appear as color less forms

b) Details of the image seen by human are conveyed through cones rather than rods on the eye retina.

3) Differentiate between:

a) Radiance and Luminance of a chromatic light source

b) Illumination and reflectance of an object when lighting it .

4) What is dynamic range, subjective brightness, weber ratio?

5) Explain briefly the human perception phenomenon simultaneous contrast and optical illusion?

6) A common measure of transmission for digital data is the baud rate, defined as

the number of bits transmitted per second. Generally, transmission is accomplished in packets consisting of a start bit, a byte (8 bits) of information, and a stop bit.

Using these facts, answer the following:

(a) How many minutes would it take to transmit a 1024x1024 image with 256 intensity levels using a 56K baud modem?

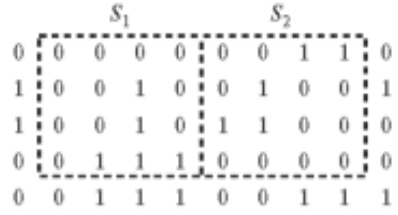
(b) What would the time be at 750K baud, a representative medium speed of a phone DSL (Digital Subscriber Line) connection?

Sheet (3)

1. Determine if each of the following statements is true or not. If it is not, modify it to become true.
   a) When the acquisition of an image is done using a single sensor both x and y dimension resolutions are controlled by the precession of mechanical movement
   b) When the acquisition of an image is done using a sensor strip both x and y dimension spatial resolutions are controlled by the precession of mechanical movement
   c) If two pixels are 8-adjacent then they are 4-adjacent
   d) The pixels having a D_4 distance from (x , y) less than or equal to some value r form a circle centered at (x, y)
   e) Gama ray imaging are typically used in exploring minerals and oil.
   f) In digital image processing, decreasing the contrast decreases the dynamic range of the image.
   g) Nearest neighborhood interpolation gives better results than bilinear interpolation when dealing with digital images.

2. High-definition television (HDTV) generates images with 1125 horizontal TV lines interlaced (where every other line is painted on the tube face in each of two fields, each field being 1/60th of a second in duration). The width-to-height aspect ratio of the images is 16:9. The fact that the number of horizontal lines is fixed determines the vertical resolution of the images. A company has designed an image capture system that generates digital images from HDTV images. The resolution of each TV (horizontal) line in their system is in proportion to vertical resolution, with the proportion being the width-to-height ratio of the images. Each pixel in the color image has 24 bits of intensity resolution, 8 bits each for a red, a green, and a blue image. These three "primary" images form a color image. How many bits would it take to store a 2-hour HDTV movie?
(Problem 2.10, Digital image processing, Gonzalez and Wood, 3ED)

3. Consider the two image subsets: S_1 and S_2, shown in the following figure. For V = {1}, determine whether these two subsets are
   (a) 4-adjacent
   (b) 8-adjacent
   (c) m-adjacent



(Problem 2.11, Digital image processing, Gonzalez and Wood, 3ED)

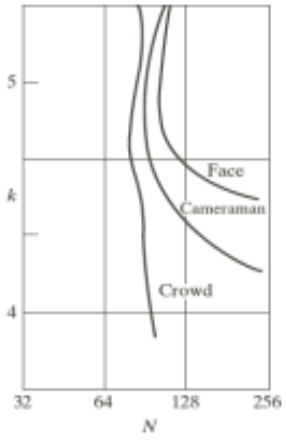
4. Consider the image segment shown below.
   a) Let V = {0, 1} and compute the lengths of the shortest 4-, 8-, and m-path between p and q. If a particular path does not exist between these two points, explain why.
   b) Repeat for V= {1, 2}.

3	1	2	1 (q)
2	2	0	2
1	2	1	1
(p) 1	0	1	2

(Problem 2.15, Digital image processing, Gonzalez and Wood, 3ED)

5. Discuss three different representations of digital images. Show the context in which each of the three representations is used

6. What do you understand from the following *isopreference* curves (shown below) for three types of images: Low details images represented by a face image, Medium details images represented by a camera-man image and high details images represented by a crowd image.



Given a region of an image with 5x5 pixels, according to Euclidian, Manhattan, and Chessboard distances, compute the distance between the pixels with ? sign and the central pixel p(x,y).

Sheet (4)

1. Prove that finding the maximum intensity value in a digital image is a nonlinear operation

2. Image filtering using the median of the pixel neighborhood is useful in removing certain kind of noise. The median, ζ, of a set of numbers is such that half the values in the set are below ζ and the other half are above it. For example, the median of the set of values {2, 3, 8, 20, 21, 25, 31} is 20.
   a) Show that an operator that computes the median of a sub-image area, S, is nonlinear.
   b) Is it possible to apply the median on images using a mask and the correlation? Why?

3. Imaging under very low light levels causes the imaging system sensor noise largely affects the resulting images. Let $g(x,y)$ denote a corrupted image formed by the addition of sensor noise: $\eta(x,y)$ to the scene noiseless image $f(x,y)$; that is $g(x,y) = f(x,y) + \eta(x,y)$. In a process of noise reduction, a set of noisy images $\{g_i(x,y)\}$ for the same scene are averaged to get a less affected by noise image. Assume that at every pair of coordinates (x,y) the noise in the added images is uncorrelated and has zero average value.
   a) Prove that the expected value of the image formed by averaging K different noisy images is the noiseless image $f(x,y)$
   b) How the variance and the standard deviation of the average image is related to the variance and the standard deviation, respectively, of the noise.
   c) Mention the condition that must be satisfied when practically applying the process for reducing the noise

4. Image subtraction is used often in industrial applications for detecting missing components in product assembly. The approach is to store a "golden" image that corresponds to a correct assembly; this image is then subtracted from incoming images of the same product. Ideally, the difference would be zero if the new products are assembled correctly. Difference images for products with missing components would be non-zero in the area where they are differs from the golden image. What conditions do you think have to be met in practice for this method to work?

5. Consider two 8 bit images whose intensity levels span the full range from 0 to 255. Discuss the limiting effect of repeatedly subtracting image 2 from image 1. Assume that the result also is represented in 8 bits

6. For the figure shown, sketch the set $A \cap B) \cup (A \cup B)^c$

Sheet (5)

1. Determine if each of the following statements is true or not. If it is not, modify it to become true.
   a) When dealing with intensity values as random variable. Its variance is proportional to the image contrast.
   b) An intensity mapping on a 8 bit image has the form $s = T(z) = 2^8 - z$, where z is the pixel intensity in the input image and s is the pixel intensity in the output image, washes the image.
   c) In image processing terminology, image registration is the process of aligning input images to a master image so that comparison and measurements could be done across all the images.
   d) In digital image processing, when dealing with image transform, it is always possible to formulate the transform using matrix multiplications
   e) In digital image processing spatial operations are usually more computationally intensive compared to transform operation

2. Prove that the Fourier transform kernels are separable and symmetric

$$r(x, y, u, v) = e^{-j2\pi(\frac{ux}{M}+\frac{vy}{N})}$$
$$s(x, y, u, v) = \frac{1}{MN} e^{j2\pi(\frac{ux}{M}+\frac{vy}{N})}$$

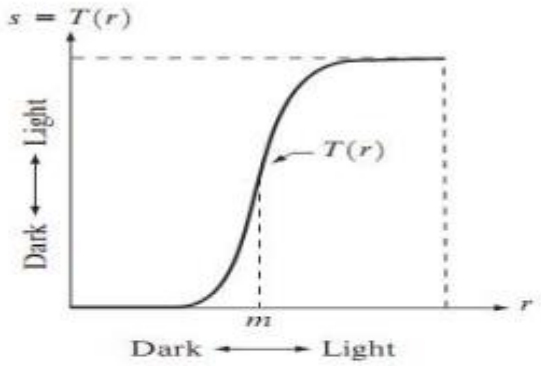
3. Show that the two dimension transform with separable, symmetric kernels can be computed by (1) computing 1-D transform along the individual rows (columns) of the input, followed by (2) computing 1-D transform along the columns(rows) of the result of step (1)

4. a) An intensity mapping on a 8 bit image has the form $s = T(z) = 2^8 - z$, where z is the pixel intensity in the input image and s is the pixel intensity in the output image, has the effect of:
      I. Washing the image
      II. Increasing the contrast of the image
      III. Decreasing the contrast of the image
      IV. Getting the negative of the image
   b) When dealing with intensity values as random variable, positive third moment means
      I. Pixel values have bias to values higher than the mean
      II. Pixel values have bias to values smaller than the mean
      III. Pixel values have bias to be uniformly distributed around the mean
   c) When dealing with intensity values as random variable, negative third moment means
      I. Pixel values have bias to values higher than the mean
      II. Pixel values have bias to values smaller than the mean
      III. Pixel values have bias to be uniformly distributed around the mean

Sheet (6)

1) Give a single intensity transformation function for spreading the intensities of an image so the lowest intensity is 0 and the highest is L-1.

2) Give a continuous function for implementing the contrast stretching transformation shown in Fig. 3.2(a). In addition to m, your function must include a parameter, E, for controlling the slope of the function as it transitions from low to high gray-level values. Your function should be normalized so that its minimum and maximum values are 0 and 1, respectively.



3) Propose a set of gray-level-slicing transformations capable of producing all the individual bit planes of an 8-bit monochrome image. (For example, a transformation function with the property $T(r) = 0$ for r in the range $[0, 127]$, and $T(r) = 255$ for r in the range $[128, 255]$ produces an image of the 7th bit plane in an 8-bit image.)

4) State whether each of the following statements true or false, in case the statement is false, give its correction:

1. Low contrast image may be generated from lack of dynamic range in the sensors.
2. For washed-out images, dynamic range stretching is required with gamma less than 1.

Sheet (7)

1. Consider a continuous image with intensity values in the range [0, L-1]. If we model the intensity level a pixel can take on as a random variable R and the value that variable takes at any pixel as r, we have the following probability density function (PDF) for the variable: $P_R(r)$. By definition, the PDF is the same as the normalized histogram for the image. It is well known in probability theorem that a continuous in r and differentiable mapping (transformation function) $s = T(r)$ that maps $P_R(r)$ to $P_S(s)$ satisfies the equation: $P_S(s) = P_R(r)\left|\frac{dr}{ds}\right|$. A mapping of particular importance in image processing has the form: $= T(r) = (L-1)\int_0^r P_W(w)dw$, where w is a dumpy integration variable.
   a) Explain the importance of the mapping in image processing
   b) In which application this mapping is used in image processing, explain.

2. A 3 bit image (L=8) of size 64x64 pixels (MxN=4096) has the intensity distribution shown in the table below, where the intensity levels are integers in the range [0-(L-1)]=[0-7].
   a) Calculate and sketch the histogram components for the image
   b) Calculate and sketch the histogram components of the image after the application of histogram equalization.

   Hint: histogram equalization intensity transformation is specified by:

$$s_k = T(r_k) = \frac{(L-1)}{MN}\sum_{j=0}^{k} n_j$$

r_k	n_k
r_0	790
r_1	1023
r_2	850
r_3	656
r_4	329
r_5	245
r_6	122
r_7	81

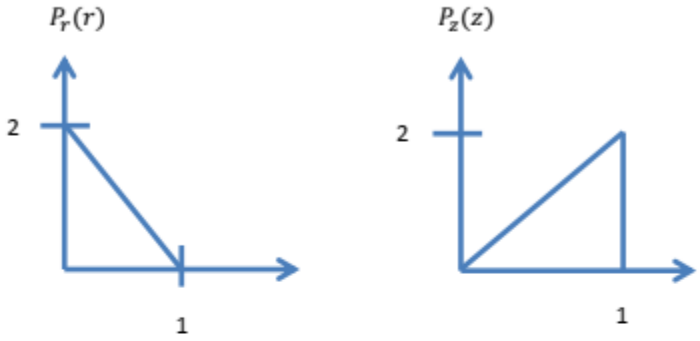
3. a) What effect would setting to zero the lower-order bit planes have on the histogram of an image in general?
   b) What would be the effect on the histogram if we set to zero the higher-order bit planes instead? (problem 3.5 Gonzalez 3ED)

4. Explain why the discrete histogram equalization technique does not, in general, yield a flat histogram(problem 3.6 Gonzalez 3ED)

5. Suppose that a digital image is subjected to histogram equalization. Show that a second pass of

Sheet (8)

1. An image with intensities in the range [0, 1] has the PDF $P_r(r)$ shown in the figure below. It's desired to transform the intensity levels of this image so that they will have the specified $P_z(z)$ shown also in the figure below. Assume continuous quantities and find the transformation (in term of r and z) that will accomplish this.



2. A 3 bit image (L=8) of size 64x64 pixels (MxN=4096) has the intensity distribution shown in the table1 below, where the intensity levels are integers in the range [0-(L-1)]=[0-7]. It's required to transform the intensity values of the original image so that we get an image with the specified histogram shown in table 2.

2 Hint: histogram equalization intensity transformation is specified by:

$$s_k = T(r_k) = \frac{(L-1)}{MN}\sum_{j=0}^{k} n_j$$

r_k	n_k
r_0	790
r_1	1023
r_2	850
r_3	656
r_4	329
r_5	245
r_6	122
r_7	81

Table 1

z_q	Specified $p_z(z_q)$
$z_0 = 0$	0.00
$z_1 = 1$	0.00
$z_2 = 2$	0.00
$z_3 = 3$	0.15
$z_4 = 4$	0.20
$z_5 = 5$	0.30
$z_6 = 6$	0.20
$z_7 = 7$	0.15

Table 2

Workout the transformation steps.

3. Propose a method for updating the local histogram for use in the local histogram equalization

Sheet (9)

1. The three images shown below were blurred using square averaging masks of sizes n = 23, 25, and 45, respectively. The vertical bars on the left lower part of (a) and (c) are blurred, but a clear separation exists between them. However, the bars have merged in image (b), in spite of the fact that the mask that produced this image is significantly smaller than the mask that produced image (c). If you know that the vertical bars are 5 pixels wide, 100 pixels high, and their separation is 20 pixels. Explain bar merging in image b.

(a) (b) (c)

2. Explain one way for digital image smoothing.

3. The two 256 gray levels images shown in the figure below have the same size and border. Although they are quite different inside they have the same histogram. Suppose that each image is blurred with a 3x3 averaging mask(Problem 3.14 Gonzalez 3ED).
   a) Explain why the two images give have the same histogram
   b) Would the histogram of the blurred images still be equal? Explain.

(a) (b)

4. The implementation of linear spatial filters requires moving the center of a mask throughout an image and, at each location, computing the sum of products of the mask coefficients with the corresponding pixels at that location. In the case of low-pass filtering, all coefficients are 1, allowing use of a so-called box-filter or moving-average algorithm, which consists of updating only the part of the computation that changes from one location to the next.
   Formulate such an algorithm for an n X n filter, showing the nature of the computations involved and the scanning sequence used for moving the mask around the image.

Sheet (10)

1. Explain one way for image sharpening

2. In a given application an averaging mask is applied to input images to reduce noise, and then Laplacian mask is applied to enhance small details. Would the result be the same if the order of these operations were reversed?

3. Show that the Laplacian operator defined by

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Is isotropic (invariant to rotation). Use the following relationships between not rotated and a rotated by θ images pixel.

$$x = \acute{x}\cos\theta - \acute{y}\sin\theta$$
$$y = \acute{x}\sin\theta + \acute{y}\cos\theta$$

Where (x,y) are the not rotated and $(\acute{x}, \acute{y})$ are the rotated coordinates

4. Explain why the Laplacian operator applied using the mask shown in the right image gives a sharper image than that shown in the left image

0	1	0
1	−4	1
0	1	0

1	1	1
1	−8	1
1	1	1

In a given application an averaging mask is applied to input images to reduce noise, and then a Laplacian mask is applied to enhance small details. Would the result be the same if the order of these operations were reversed?

You saw in Fig. 3.38 that the Laplacian with a -8 in the center yields sharper results than the one with a -4 in the center. Explain the reason in detail.

With reference to Problem 3.25,

(a) Would using a larger "Laplacian-like" mask, say, of size 5×5 with a -24 in the center, yield an even sharper result? Explain in detail.

(b) What happens when the size of the mask becomes equal to the image size.

Show that subtracting the Laplacian from an image is proportional to unsharp masking. Use the definition for the Laplacian given in Eq. (3.6-6).

Sheet (11)

Q.1) what are performance criteria of pattern recognition algorithms?

Q.2) Mention six areas related to pattern recognition?

Q.3) State four applications of pattern recognition?

Q.4) State some examples of classifiers and explain in detail one of them?

Q.5) Mention the steps involved in implementing the PCA algorithm?

Q.6) Get the principal components of the following data samples and project these data:

Data =

x	y
2.5	2.4
0.5	0.7
2.2	2.9
1.9	2.2
3.1	3.0
2.3	2.7
2	1.6
1	1.1
1.5	1.6
1.1	0.9