**UNIT - 1**

**Geometric Primitives and Transformations**

Computer vision deals with the automatic extraction, analysis, and understanding of useful information from images and videos. Geometric primitives and transformations are essential concepts in computer vision, as they form the building blocks for various image processing and object recognition tasks.

1. **Geometric Primitives:** Geometric primitives are basic shapes or entities that serve as the foundation for describing objects in an image. Some common geometric primitives used in computer vision include:

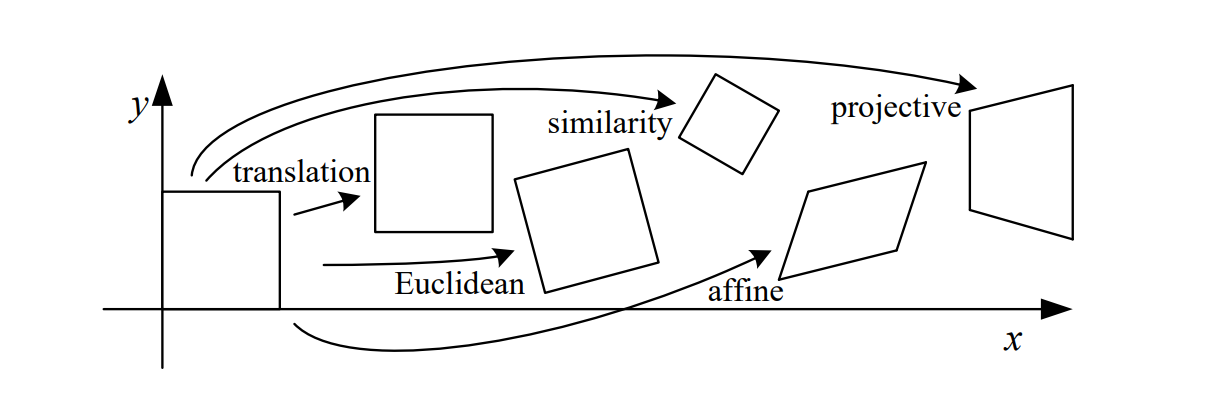
* Points: A point represents a single coordinate in an image, typically denoted as (x, y) in a 2D image.
* Lines: Lines are represented by two endpoints or as an equation (e.g., y = mx + b). They are used to represent edges or boundaries of objects in an image.
* Rectangles: Rectangles are used to represent bounding boxes around objects of interest in an image. They are often used for object localization and detection tasks.

1. **Geometric Transformations:** Geometric transformations involve modifying the geometric properties of an image while preserving its essential characteristics. These transformations are used to align, scale, rotate, or warp images, which can be helpful for various computer vision tasks. Some common geometric transformations include:

* Translation: A translation moves the entire image in a specified direction by shifting all the points by a constant displacement (dx, dy).
* Rotation: Rotation transforms an image by rotating it around a specified point (often the center) by a given angle.
* Scaling: Scaling changes the size of an image by multiplying the coordinates of each point by scaling factors in the x and y directions.
* Shearing: Shearing skews an image along one axis while leaving the other axis unchanged.
* Affine Transformation: Affine transformations include a combination of translation, rotation, scaling, and shearing. They can be used to rectify perspective distortions and align images.
* Perspective Transformation: Perspective transformations are used to correct foreshortening effects that occur when viewing a 3D scene from an arbitrary viewpoint.

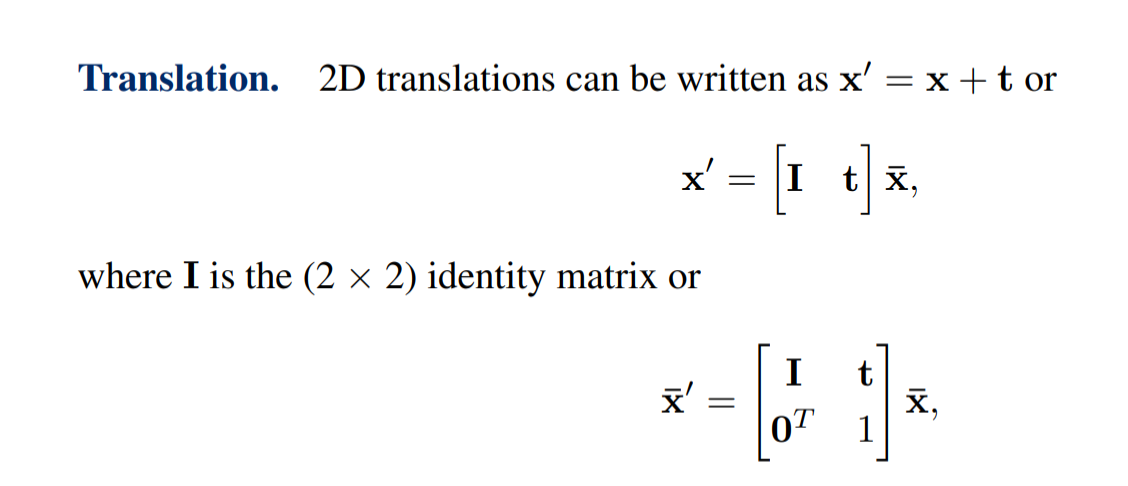
These transformations are crucial for image preprocessing, image registration (aligning images), and other computer vision tasks like object recognition, image stitching, and augmented reality.

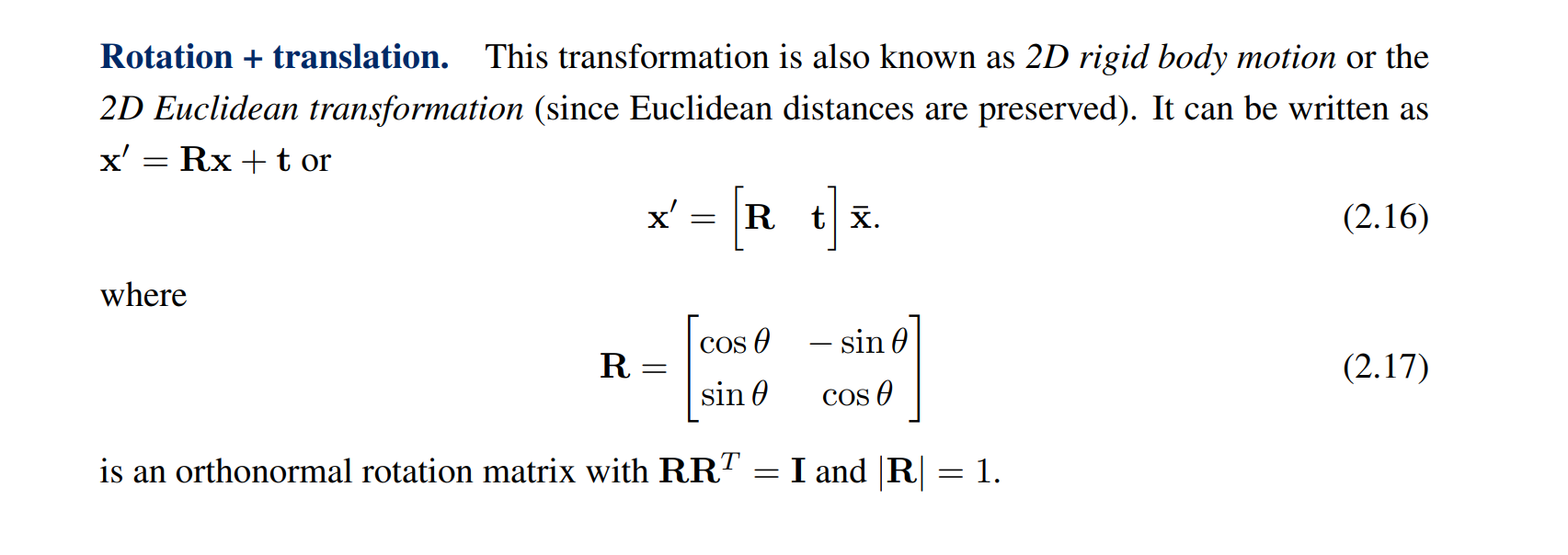
Following picture describes the geometric transformations in detail

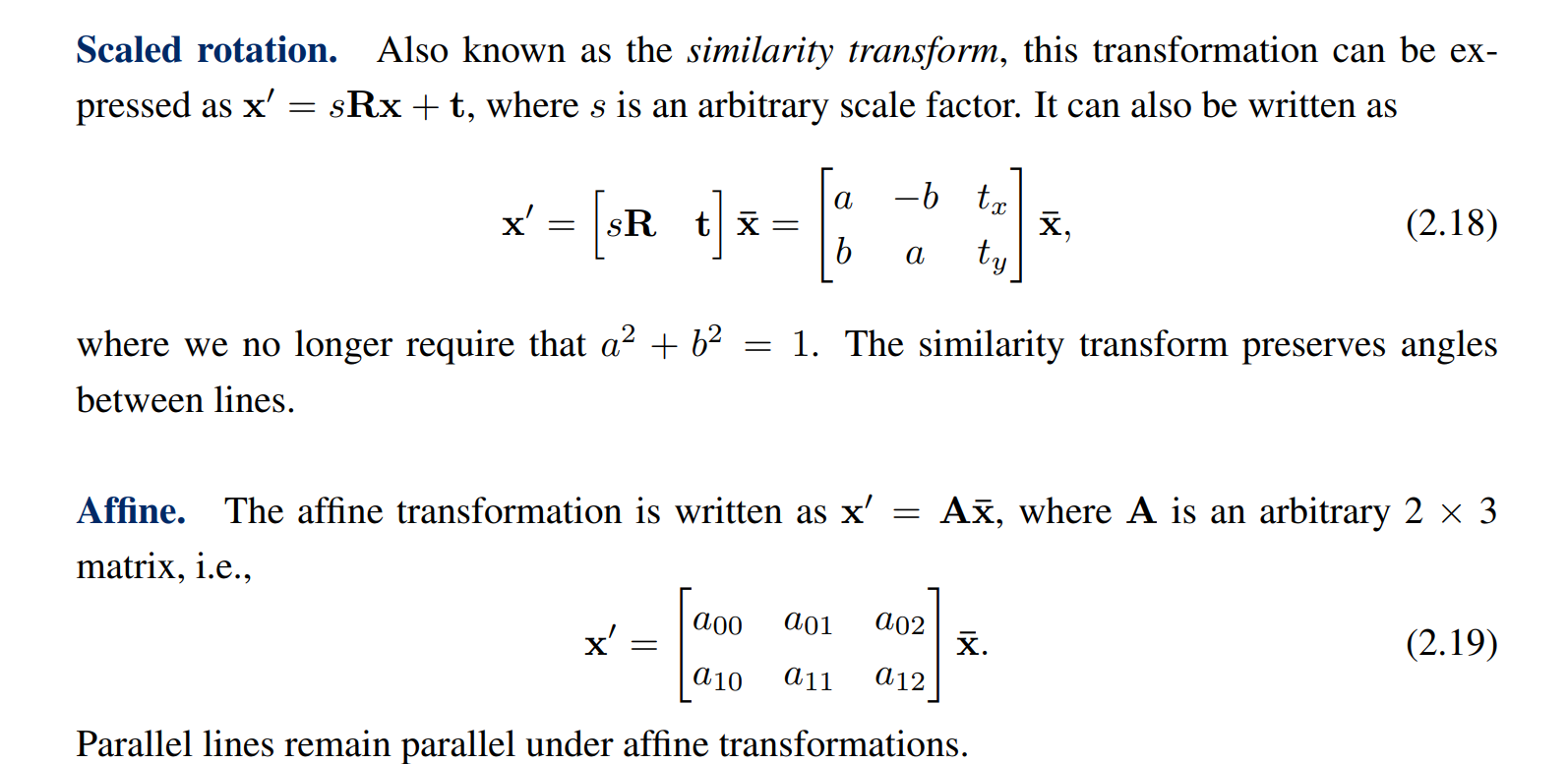


TTop of Form

Mathematic Description of Transformations





C

Implementation of Transformations in OpenCV

import cv2

import numpy as np

# Load the image

image = cv2.imread('c:/users/kalla/desktop/computer vision/computer\_vision/computer\_vision/image\_data/elephant.png')

# Translation

tx, ty = 50, 30

translation\_matrix = np.float32([[1, 0, tx], [0, 1, ty]])

translated\_image = cv2.warpAffine(image, translation\_matrix, (image.shape[1], image.shape[0]))

# Rotation

angle = 45

rotation\_matrix = cv2.getRotationMatrix2D((image.shape[1] // 2, image.shape[0] // 2), angle, 1)

rotated\_image = cv2.warpAffine(image, rotation\_matrix, (image.shape[1], image.shape[0]))

# Scaling

scale\_factor = 1.5

scaled\_image = cv2.resize(image, None, fx=scale\_factor, fy=scale\_factor, interpolation=cv2.INTER\_LINEAR)

# Shearing

shear\_factor = 0.3

shear\_matrix = np.float32([[1, shear\_factor, 0], [0, 1, 0]])

sheared\_image = cv2.warpAffine(image, shear\_matrix, (image.shape[1], image.shape[0]))

# Affine Transformation

# Define three points in the original image and their corresponding locations in the output image

original\_points = np.float32([[50, 50], [200, 50], [50, 200]])

output\_points = np.float32([[70, 100], [220, 50], [150, 250]])

affine\_matrix = cv2.getAffineTransform(original\_points, output\_points)

affine\_image = cv2.warpAffine(image, affine\_matrix, (image.shape[1], image.shape[0]))

# Display the results

cv2.imshow('Original Image', image)

cv2.imshow('Translated Image', translated\_image)

cv2.imshow('Rotated Image', rotated\_image)

cv2.imshow('Scaled Image', scaled\_image)

cv2.imshow('Sheared Image', sheared\_image)

cv2.imshow('Affine Image', affine\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Photometric Image Formation**

Photometric image formation is a fundamental concept in computer vision that deals with understanding how light interacts with surfaces and objects in a scene to generate the image captured by a camera. It involves the study of the factors that influence the appearance of objects in an image, such as lighting conditions, surface properties, and the camera's characteristics.

Here are some key components of photometric image formation in computer vision:

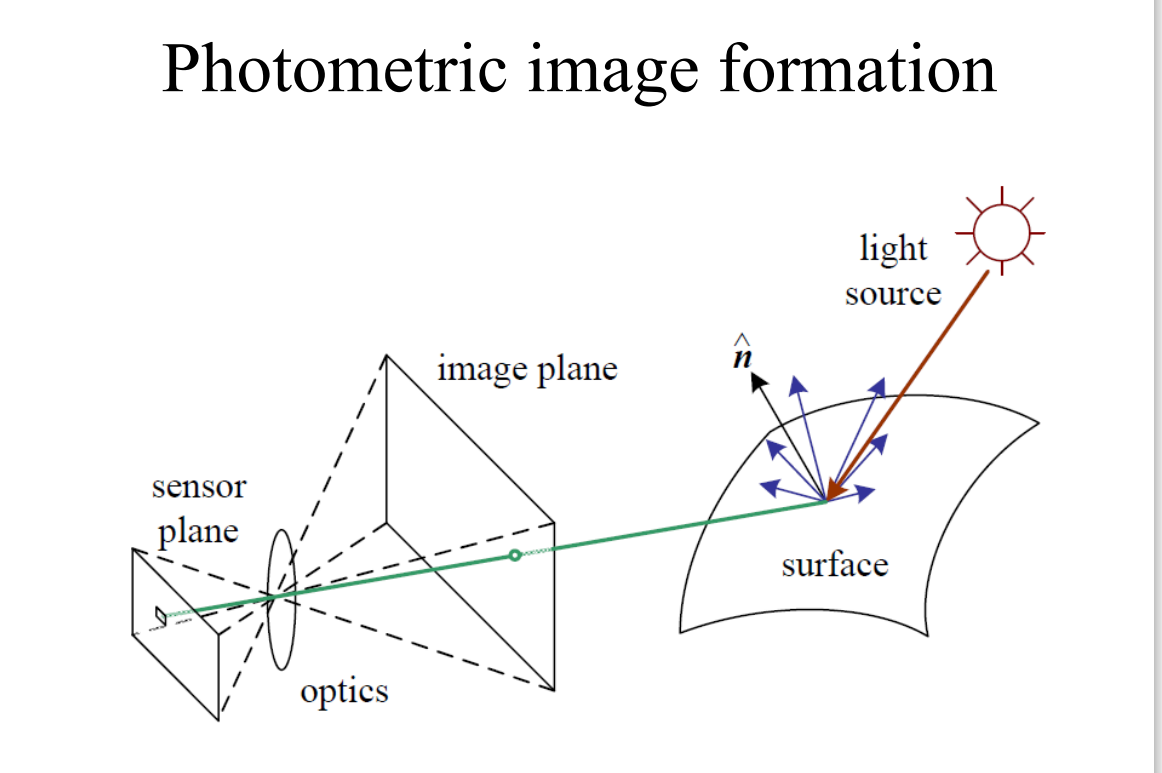
**Illumination**: Illumination refers to the light sources present in the scene and their characteristics. The amount, direction, and color of light affect how objects appear in the image. Understanding and modeling illumination is crucial for various computer vision tasks, such as object recognition, tracking, and scene understanding.

**Reflectance**: Reflectance describes the property of a surface to reflect light. Different materials and surfaces have varying levels of reflectance, which leads to variations in brightness and color in the image. Understanding the reflectance properties of surfaces is essential for tasks like material recognition and texture analysis.

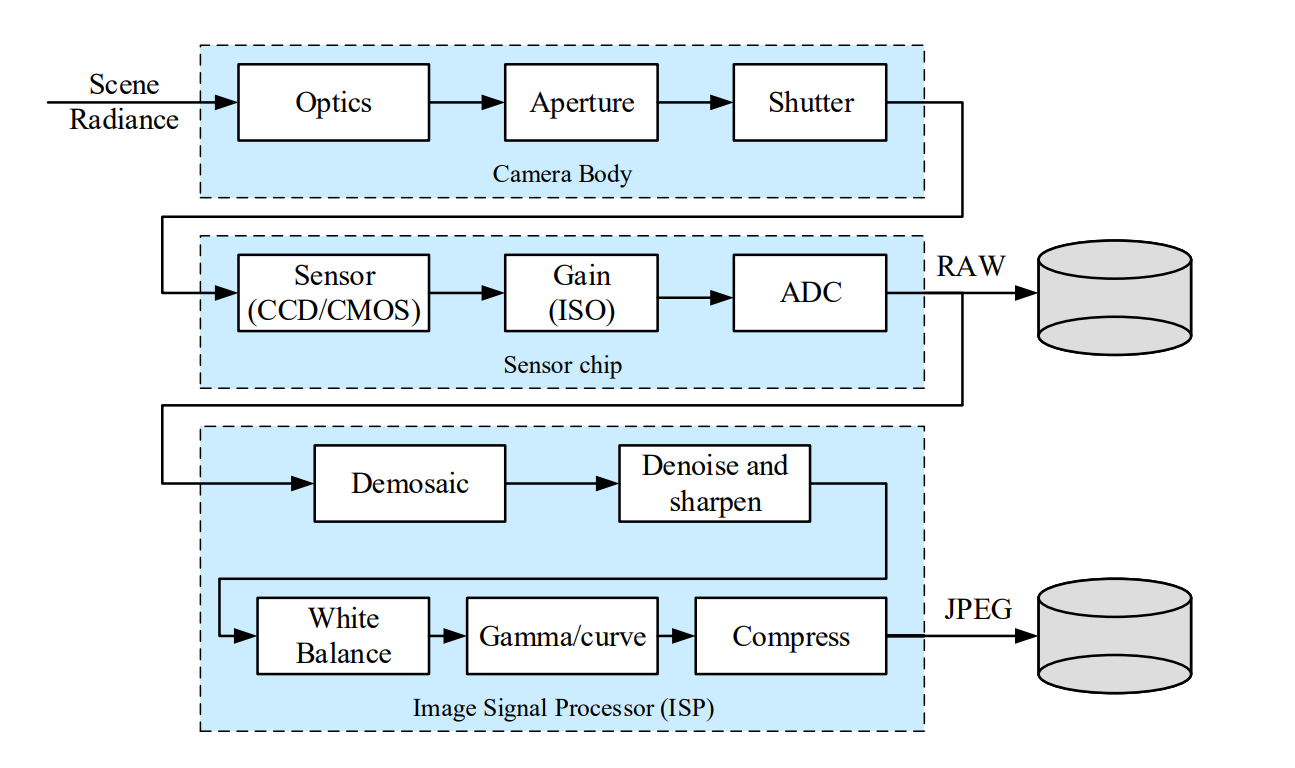
**Shadows and Highlights**: When light interacts with objects, it can create shadows and highlights. Shadows occur in areas where light is obstructed, while highlights are bright areas caused by direct illumination. These shadow and highlight regions are essential for understanding the 3D structure of the scene.

**BRDF (Bidirectional Reflectance Distribution Function)**: The BRDF is a mathematical function that models the reflection of light from a surface. It characterizes how light scatters in different directions, depending on the incident light direction and the surface normal. The BRDF is often used in computer graphics and computer vision to simulate the appearance of objects under different lighting conditions.

**Camera Model**: The camera's characteristics, such as the sensor, lens, and other intrinsic parameters, also play a role in image formation. Understanding the camera model allows for accurate image calibration, rectification, and pose estimation.



**The Digital Camera**



The above picture describes the various phases of image formation and sequence of workflow done in the digital camera. It consists of 3 parts.

1. Camera Body
2. Sensor Chip
3. Image Signal Processor (ISP)

Shutter Speed directly controls the amount of light reaching the sensor and hence determines if the images are over-exposed or under-exposed.

The light falling on the imaging sensor is usually picked up by an active sensing area. There are two types of sensor. They are CCD (Charge Coupled Device) and CMOD (Complementory Metal Oxide on Silicon)

ADC ( Analog to Digital Conversion) is done. The outcome consists of noise, mosaic etc… It is basically stored in RAW form

The Image Signal Processor performs demosaic and removes noise and sharpening the brightness.

**Point Operators**

Point operators are simplest kind of image processing transforms where each pixel’s value depends only on the corresponding input pixel’s value.

Some of the point operators:

1. Pixel Transforms
2. Histogram Equalization
3. Thresholding
4. Composting and Matting

**Pixel transforms** are operations that modify the pixel values of an image. They can be applied to individual pixels or groups of pixels to achieve various effects. Some common pixel transforms include:

* **Brightness and Contrast Adjustment:** Changing the overall brightness and contrast of an image to make it lighter or darker and enhance the visual quality.

*Brightness Adjustment:*

To adjust the brightness of an image, you can use the cv2.convertScaleAbs() function, which scales and shifts the pixel values

Example for Brightness adjustment

import cv2

# Read the image

img = cv2.imread('c:/images/nature.jpg')

img = cv2.resize(img,(500,700))

# Define the brightness adjustment factor (positive value increases brightness, negative value decreases brightness)

brightness\_factor = 50

# Apply the brightness adjustment

adjusted\_image = cv2.convertScaleAbs(img, beta=brightness\_factor)

# Display the original and adjusted images

cv2.imshow('Original Image', img)

cv2.imshow('Adjusted Image', adjusted\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

Example for Contrast adjustment

*Contrast Adjustment:*

To adjust the contrast of an image, you can use the cv2.addWeighted() function, which blends the original image with a copy that has been adjusted to change the contrast.

import cv2

# Read the image

img = cv2.imread('c:/images/nature.jpg')

img = cv2.resize(img,(500,700))

# Define the contrast adjustment factor (alpha value)

contrast\_factor = 1.5

# Adjust the contrast by blending the original image with a copy that has been scaled by the contrast factor

adjusted\_image = cv2.addWeighted(img, contrast\_factor, img, 0, 0)

# Display the original and adjusted images

cv2.imshow('Original Image', img)

cv2.imshow('Adjusted Image', adjusted\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

* **Gamma Correction:** Adjusting the gamma value to correct the brightness levels and improve the display on different devices.

Gamma correction is a technique used in image processing to adjust the perceived brightness of an image. It involves applying a nonlinear operation to the pixel values, effectively changing the relationship between the pixel intensities and the displayed brightness. The purpose of gamma correction is to compensate for the nonlinear response of display devices, such as monitors or screens, and to ensure that images appear more natural and visually pleasing to the human eye.

Gamma is a parameter that represents the nonlinear relationship between pixel values and brightness. The gamma value determines the shape of the gamma correction curve. A gamma value of 1 means no correction is applied, and the image appears as is. Gamma values greater than 1 (typically between 1.8 and 2.5) make the image brighter, while gamma values less than 1 make the image darker.

*Implemenation for Gamma Correction*

import cv2

import numpy as np

# Read the image

img = cv2.imread('c:/images/nature.jpg')

img = cv2.resize(img,(500,700))

# Define the gamma value (typically between 0.1 and 5)

gamma = 2.5

# Perform gamma correction

gamma\_corrected = np.power(img / 255.0, 1 / gamma)

gamma\_corrected = np.uint8(gamma\_corrected \* 255)

# Display the original and gamma-corrected images

cv2.imshow('Original Image', img)

cv2.imshow('Gamma Corrected Image', gamma\_corrected)

cv2.waitKey(0)

cv2.destroyAllWindows()

* **Thresholding:** Converting a grayscale image to binary by setting a threshold value to separate foreground and background pixels.

Thresholding is a widely used technique in image processing and computer vision to convert a grayscale image into a binary image, where each pixel is either classified as foreground (white) or background (black) based on a specified threshold value. The process is straightforward and helps to separate objects of interest from the background, making it easier to perform further analysis or object recognition tasks

*Implementation for Thresholding*

import cv2

# Read the image in grayscale mode

img = cv2.imread('c:/images/nature.jpg',cv2.IMREAD\_GRAYSCALE)

img = cv2.resize(img,(500,700))

# Choose a threshold value (you can experiment with different values)

threshold\_value = 128

# Apply global thresholding

\_, binary\_image = cv2.threshold(img, threshold\_value, 255, cv2.THRESH\_BINARY)

# Display the original and binary images

cv2.imshow('Original Image', img)

cv2.imshow('Binary Image', binary\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

Color Transforms

* Color transforms involve modifying the color representation of an image.
* Color spaces are used to represent colors, and different color spaces have specific applications.
* Some popular color spaces include RGB (Red, Green, Blue), CMYK (Cyan, Magenta, Yellow, Black), HSV (Hue, Saturation, Value).
* Color transforms enable tasks such as color correction, colorization, and channel separation.

*Implementation for Color Transforms*

*BGR to RGB and Vice Versa:*

In OpenCV, images are typically read as BGR (Blue, Green, Red) channels instead of RGB (Red, Green, Blue). If you need to convert BGR to RGB or vice versa, you can use the cv2.cvtColor() function.

import cv2

# Read the BGR image

bgr\_image = cv2.imread('c:/images/nature.jpg')

bgr\_image=cv2.resize(bgr\_image,(500,700))

cv2.imshow('BGR COLOR',bgr\_image)

# Convert BGR to RGB

rgb\_image = cv2.cvtColor(bgr\_image, cv2.COLOR\_BGR2RGB)

cv2.imshow('RGB COLOR',rgb\_image)

# Convert RGB back to BGR

bgr\_image\_again = cv2.cvtColor(rgb\_image, cv2.COLOR\_RGB2BGR)

cv2.imshow('BGR COLOR AGAIN',bgr\_image\_again)

cv2.waitKey(0)

**Histogram Equalization**

* Enhancing the contrast of an image by redistributing pixel intensities to cover the entire dynamic range

Histogram equalization is especially effective when an image's histogram is not well-distributed across the intensity range. It can be particularly useful for images that appear too dark or too bright, as it can bring out more details and make the image visually appealing.

**Implementation of Histogram Equalization in OpenCV(Before application of histogram Equalization)**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read the color image

img = cv2.imread('c:/images/nature.jpg')

img = cv2.resize(img,(500,700))

# Split the image into its color channels

b, g, r = cv2.split(img)

# Compute the histograms for each channel

hist\_b = cv2.calcHist([b], [0], None, [256], [0, 256])

hist\_g = cv2.calcHist([g], [0], None, [256], [0, 256])

hist\_r = cv2.calcHist([r], [0], None, [256], [0, 256])

# Plot the histograms

plt.figure(figsize=(8, 6))

plt.plot(hist\_b, color='b', label='Blue', alpha=0.8)

plt.plot(hist\_g, color='g', label='Green', alpha=0.8)

plt.plot(hist\_r, color='r', label='Red', alpha=0.8)

plt.title('Color Image Histogram')

plt.xlabel('Pixel Intensity')

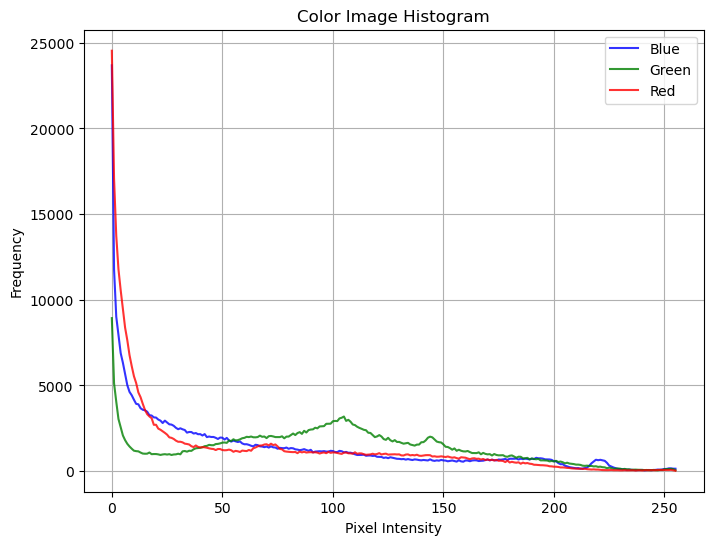
plt.ylabel('Frequency')

plt.legend()

plt.grid(True)

plt.show()

***Output:***



**Implementation of Histogram Equalization in OpenCV(After application of histogram Equalization)**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Split the image into its color channels

b, g, r = cv2.split(equalized\_img)

# Compute the histograms for each channel

hist\_b = cv2.calcHist([b], [0], None, [256], [0, 256])

hist\_g = cv2.calcHist([g], [0], None, [256], [0, 256])

hist\_r = cv2.calcHist([r], [0], None, [256], [0, 256])

# Plot the histograms

plt.figure(figsize=(8, 6))

plt.plot(hist\_b, color='b', label='Blue', alpha=0.8)

plt.plot(hist\_g, color='g', label='Green', alpha=0.8)

plt.plot(hist\_r, color='r', label='Red', alpha=0.8)

plt.title('Color Image Histogram')

plt.xlabel('Pixel Intensity')

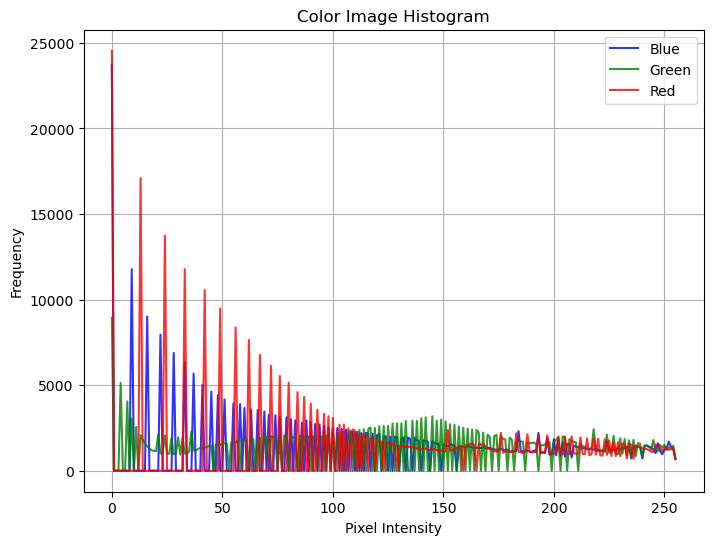
plt.ylabel('Frequency')

plt.legend()

plt.grid(True)

plt.show()

***Output:***

******

**Composting and Matting**

Compositing is the process of combining multiple images or visual elements to create a final composite image.

This technique is commonly used in film, photography, and computer graphics to create realistic scenes or special effects.

Compositing allows you to overlay foreground elements on a background, adjust transparency, and apply blending modes to achieve seamless integration.

Matting is a fundamental task in computer vision and image editing that involves separating the foreground object from its background.

The goal is to generate an alpha matte, which represents the opacity or transparency of each pixel.

Once the alpha matte is obtained, the foreground object can be placed on a different background or used for further image manipulation.

Techniques for matting include color-based methods, depth-based methods, and machine learning-based approaches

**Implementation of Composting**

import cv2

import numpy as np

my\_image\_color\_1 = cv2.imread('c:/users/kalla/desktop/computer vision/computer\_vision/computer\_vision/image\_data/forest.jpg')

my\_image\_color\_2 = cv2.imread('c:/users/kalla/desktop/computer vision/computer\_vision/computer\_vision/image\_data/greenscreen.jpg')

my\_image\_color\_3 = cv2.imread('c:/images/nature.jpg')

**# get the minimum parameters on which resize can happen**

my\_image\_color\_1.shape, my\_image\_color\_2.shape # h = 375, w = 500

new\_h = min(my\_image\_color\_1.shape[0], my\_image\_color\_2.shape[0],my\_image\_color\_3.shape[0])

new\_w = min(my\_image\_color\_1.shape[1], my\_image\_color\_2.shape[1],my\_image\_color\_3.shape[1])

**#resize on same scale**

my\_image\_color\_1 = cv2.resize(my\_image\_color\_1, (new\_w, new\_h), cv2.INTER\_AREA)

my\_image\_color\_2 = cv2.resize(my\_image\_color\_2, (new\_w, new\_h), cv2.INTER\_AREA)

my\_image\_color\_3 = cv2.resize(my\_image\_color\_3, (new\_w, new\_h), cv2.INTER\_AREA)

**# Simple addition**

image\_added = cv2.add(my\_image\_color\_1,my\_image\_color\_2,my\_image\_color\_3)

cv2.imshow('image\_added',image\_added); cv2.waitKey(0); cv2.destroyAllWindows()

#**Weighted addition**

image\_weighted = cv2.addWeighted(my\_image\_color\_1,0.3,my\_image\_color\_3,0.8,gamma=0)

cv2.imshow('image\_weighted',image\_weighted); cv2.waitKey(0); cv2.destroyAllWindows()

**Implementation of Matting**

import cv2

import numpy as np #%% Put one image on another background

**# Load two images**

my\_image\_color\_1 = cv2.imread('c:/users/kalla/desktop/computer vision/computer\_vision/computer\_vision/image\_data/forest.jpg')

my\_image\_color\_2 = cv2.imread('c:/users/kalla/desktop/computer vision/computer\_vision/computer\_vision/image\_data/123.PNG') cv2.imshow('Image1',my\_image\_color\_1)

cv2.imshow('Image2',my\_image\_color\_2)

cv2.waitKey(0)

cv2.destroyAllWindows()

**# get the minimum parameters on which resize can happen**

my\_image\_color\_1.shape, my\_image\_color\_2.shape # h = 375, w = 500

new\_h = min(my\_image\_color\_1.shape[0], my\_image\_color\_2.shape[0])

new\_w = min(my\_image\_color\_1.shape[1], my\_image\_color\_2.shape[1])

**#resize on same scale**

my\_image\_color\_1 = cv2.resize(my\_image\_color\_1, (new\_w, new\_h), cv2.INTER\_AREA)

my\_image\_color\_2 = cv2.resize(my\_image\_color\_2, (new\_w, new\_h), cv2.INTER\_AREA)

**# Now create a mask of second image and create its inverse mask**

my\_image\_color\_2gray = cv2.cvtColor(my\_image\_color\_2,cv2.COLOR\_BGR2GRAY)

cv2.imshow('second image',my\_image\_color\_2)

cv2.imshow('image\_weighted',my\_image\_color\_2gray); cv2.waitKey(0); cv2.destroyAllWindows()

cv2.imwrite('./image\_output/1.my\_image\_color\_2gray.png',my\_image\_color\_2gray)

**# add a threshold**

ret, mask = cv2.threshold(my\_image\_color\_2gray, 20, 255, cv2.THRESH\_BINARY\_INV)

cv2.imshow('image\_weighted',mask); cv2.waitKey(0); cv2.destroyAllWindows()

cv2.imwrite('./image\_output/2.mask.png',mask) mask\_inv = cv2.bitwise\_not(mask)

cv2.imshow('image\_weighted',mask\_inv); cv2.waitKey(0); cv2.destroyAllWindows()

cv2.imwrite('./image\_output/3.mask\_inv.png',mask\_inv)

**# Now black-out the area of second image in my\_image\_color\_1**

my\_image\_color\_1\_bg = cv2.bitwise\_and(my\_image\_color\_1,my\_image\_color\_1,mask = mask\_inv)

cv2.imshow('image\_weighted',my\_image\_color\_1\_bg); cv2.waitKey(0); cv2.destroyAllWindows()

cv2.imwrite('./image\_output/4.my\_image\_color\_1\_bg.png',my\_image\_color\_1\_bg)

**LINEAR FILTERS**

In computer vision, linear filters are essential tools used for various image processing tasks like noise reduction, edge detection, and feature extraction.

A linear filter is a mathematical operation applied to each pixel in an image, where the output value at a particular pixel is a linear combination of the pixel values in a local neighborhood around that pixel. These filters are usually represented by small matrices, also known as kernels or masks, which are convolved with the image to produce the filtered output.

The convolution operation involves sliding the kernel over the entire image, and at each location, the elements of the kernel are multiplied with the corresponding pixel values in the local neighborhood, and the sum of these products gives the value of the output pixel. This process is repeated for all pixels in the image.

**Identity Filter (or Pass-through filter)**:

The identity filter does not alter the image and is defined as a square matrix with all elements as 0 except for the central element, which is set to 1.

[ 0 0 0 ]

[ 0 1 0 ]

[ 0 0 0 ]

**Blur (or Box) Filter**: A blur filter is used to reduce noise and smooth the image. It is represented by a square matrix with all elements equal, and the sum of the elements is usually set to 1.

[ 1/9 1/9 1/9 ]

[ 1/9 1/9 1/9 ]

[ 1/9 1/9 1/9 ]

**Edge Detection Filters**:

Edge detection filters highlight regions of significant intensity transitions in an image, which correspond to edges or boundaries. The two popular edge detection filters are the Sobel and Scharr filters.

**Sobel X Filter:**

[ -1 0 1 ]

[ -2 0 2 ]

[ -1 0 1 ]

**Sobel Y Filter:**

[ -1 -2 -1 ]

[ 0 0 0 ]

[ 1 2 1 ]

Example

import cv2

import numpy as np

image = cv2.imread('c:/images/lion.jpg')

kernel\_blur = np.ones((3, 3), dtype=np.float32) / 9.0

kernel\_sobel\_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)

kernel\_sobel\_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float32)

filtered\_image\_blur = cv2.filter2D(image, -1, kernel\_blur)

filtered\_image\_sobel\_x = cv2.filter2D(image, -1, kernel\_sobel\_x)

filtered\_image\_sobel\_y = cv2.filter2D(image, -1, kernel\_sobel\_y)

cv2.imshow('Original Image', image)

cv2.imshow('Blurred Image', filtered\_image\_blur)

cv2.imshow('Sobel X', filtered\_image\_sobel\_x)

cv2.imshow('Sobel Y', filtered\_image\_sobel\_y)

**Gaussian Filter**:

The Gaussian filter is a specific type of blur filter that applies a weighted average to the neighboring pixels to reduce noise while preserving important image features. The kernel size and weights are determined based on the desired level of blurring (or standard deviation) for the Gaussian distribution.

import cv2

import numpy as np

**# Load the image (replace 'image\_path' with the actual path to your image)**

image = cv2.imread('image\_path')

**# Define the kernel size (must be an odd number)**

kernel\_size = (5, 5)

**# Apply the Gaussian filter**

filtered\_image = cv2.GaussianBlur(image, kernel\_size, sigmaX=0)

**# Display the original and filtered images**

cv2.imshow('Original Image', image)

cv2.imshow('Filtered Image', filtered\_image)

**# Wait for a key press and then close the windows**

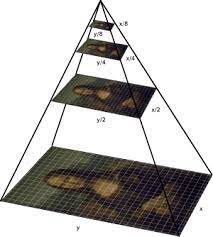
cv2.waitKey(0)

cv2.destroyAllWindows()

**Image Pyramids and Wavelets**

Computer vision pyramids and wavelets are techniques used in image processing to analyze and process images at multiple scales. They are particularly useful for tasks like image enhancement, feature extraction, object detection, and image compression.

Image Pyramids: An image pyramid is a collection of scaled-down versions of an original image, forming a multi-resolution representation.



There are two common types of image pyramids:

a. Gaussian Pyramid: The Gaussian pyramid is created by applying a series of Gaussian blurs and downsampling the image. At each level, the image is smoothed to reduce noise and then resized to half its original resolution. This process can be repeated to create multiple levels, resulting in a stack of images, each representing the original image at different scales.

b. Laplacian Pyramid: The Laplacian pyramid is derived from the Gaussian pyramid. Each level of the Laplacian pyramid is obtained by subtracting the up-sampled version of the Gaussian pyramid level from the level just above it. This represents the details or high-frequency information at different scales.

Image pyramids are useful for applications like image blending, image stitching, and object detection, where analyzing the image at multiple scales helps to capture both fine and coarse details.

Wavelets: Wavelet transform is a mathematical transform that decomposes a signal (in this case, an image) into a set of wavelet basis functions. These basis functions are localized in both time and frequency, allowing for a multi-resolution analysis of the image. There are two main types of wavelet transforms used in image processing:

a. Continuous Wavelet Transform (CWT): The CWT applies wavelet basis functions of varying scales and positions to an image, resulting in a continuous representation of the image in the wavelet domain. However, it can be computationally expensive.

b. Discrete Wavelet Transform (DWT): The DWT, on the other hand, uses a discrete set of wavelet basis functions, making it computationally more efficient than CWT. It decomposes the image into different frequency bands and provides a multi-resolution representation.

Implementation for Gaussian Pyramids

import cv2

# Read an image

image = cv2.imread('c:/images/nature.jpg')

#image = cv2.resize(image,(500,700))

#cv2.imshow('Original Image',image)

# Create a Gaussian pyramid

gaussian\_pyramid = [image]

for i in range(4): # Create 3 levels

image = cv2.pyrDown(image)

gaussian\_pyramid.append(image)

for i in range(4):

cv2.imshow('Level '+str(i+1),gaussian\_pyramid[i] )

cv2.waitKey(0)

cv2.destroyAllWindows()

Implementation for Laplacian Pyramids

import cv2

# Read an image

image = cv2.imread('c:/images/nature.jpg')

# Create a Gaussian pyramid

gaussian\_pyramid = [image]

for i in range(3): # Create 3 levels

image = cv2.pyrDown(image)

gaussian\_pyramid.append(image)

# Create a Laplacian pyramid

laplacian\_pyramid = []

for i in range(3):

expanded = cv2.pyrUp(gaussian\_pyramid[i + 1])

laplacian = cv2.subtract(gaussian\_pyramid[i], expanded)

laplacian\_pyramid.append(laplacian)

# Display the images or do further processing

cv2.imshow('Original Image', gaussian\_pyramid[0])

for i, laplacian in enumerate(laplacian\_pyramid):

cv2.imshow(f'Laplacian Level {i}', laplacian)

cv2.waitKey(0)

cv2.destroyAllWindows()

Implementation for Wavelets

import cv2

import numpy as np

import pywt

# Read an image

image = cv2.imread('c:/images/nature.jpg', cv2.IMREAD\_GRAYSCALE)

# Perform DWT using PyWavelets

wavelet = 'haar' # Choose a wavelet type (e.g., 'haar', 'db1', etc.)

coeffs = pywt.dwt2(image, wavelet)

# Extract approximation and details coefficients

cA, (cH, cV, cD) = coeffs

# Display the images or do further processing

cv2.imshow('Original Image', image)

cv2.imshow('Approximation Coefficients', cA)

cv2.imshow('Horizontal Detail Coefficients', cH)

cv2.imshow('Vertical Detail Coefficients', cV)

cv2.imshow('Diagonal Detail Coefficients', cD)

cv2.waitKey(0)

cv2.destroyAllWindows()