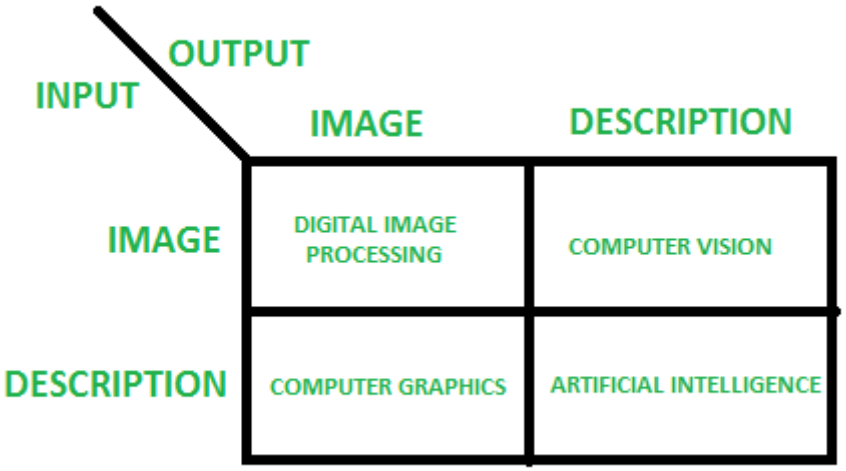
Computer Vision



Digital Image Processing

Table of Contents

DIP Basics

What is Image Processing?

Image processing involves manipulating and analyzing images to enhance their quality or extract useful information. This field encompasses a range of techniques and applications, including filtering, segmentation, and feature extraction.

What is an Image in DIP?

In Digital Image Processing, an image is defined as a ***two-dimensional array of pixels***, where each pixel represents a specific point in the image. Each pixel can have one or more values, depending on the image type (e.g., grayscale, color).

What is Digital Image Processing?

Digital Image Processing refers to the use of computer algorithms to perform image processing on digital images. It involves techniques to improve the visual appearance of images or to convert them into a form that is more suitable for analysis and understanding.

Fundamental Steps in DIP

1. **Image Acquisition:** Capturing an image using sensors (e.g., cameras, scanners).
2. **Preprocessing**: Enhancing the image quality by removing noise, adjusting contrast, etc.
3. **Segmentation:** Dividing the image into meaningful regions or objects.
4. **Feature Extraction:** Identifying and extracting relevant features from the segmented image.
5. **Image Analysis:** Interpreting the extracted features to gain insights or perform specific tasks.
6. **Image Display and Storage:** Presenting the processed image and storing it for future use.

Components of Digital Image Processing

1. **Image Formation:** Understanding how images are created and represented digitally.
2. **Image Enhancement:** Techniques to improve the visual quality of an image (e.g., contrast adjustment, filtering).
3. **Image Restoration:** Methods to recover an image that has been degraded by various factors.
4. **Image Compression:** Reducing the amount of data required to represent an image without significant loss of quality.
5. **Image Segmentation:** Techniques to partition an image into distinct regions for further analysis.
6. **Image Analysis:** The study of extracted features for recognition, classification, or other purposes.
7. **Machine Learning:** Applying algorithms to learn from data and improve image processing tasks, such as classification and detection.

Image as a Matrix

Images are represented in rows and columns we have the following syntax in which images are represented:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & f(0,2) & \dots & f(0,N-1) \\ f(1,0) & f(1,1) & f(1,2) & \dots & f(1,N-1) \\ \vdots & \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & f(M-1,2) & \dots & f(M-1,N-1) \end{bmatrix}$$

Dimensions of Images

The dimensions of an image refer to its size in terms of width and height, typically measured in pixels.

Types of Dimensions:

- **2D Images:** Most common type, represented as ***a matrix of pixels***.
- **3D Images:** These include additional information such as ***color channels***.
- **Multispectral and Hyperspectral Images:** Used in remote sensing, these images capture data at multiple wavelengths, resulting in more than three dimensions. Each pixel contains information from several spectral bands.

Pixels

A pixel (short for "picture element") is the smallest unit of a digital image. It represents a single point in the image and contains information about color or intensity.

- ***Pixel Value:*** Represents the intensity (in grayscale) or color (in color images). In an 8-bit grayscale image, pixel values range from 0 (black) to 255 (white).
- ***Bit Depth:*** The bit depth of an image defines how many bits are used to represent each pixel. Common bit depths include:
  - **1-bit:** Black and white (2 colors)
  - **8-bit:** Grayscale (256 levels of gray)
  - **24-bit:** True color (16.7 million colors using RGB)
  - **32-bit:** Often includes an alpha channel for transparency.
- ***Color Representation:*** In color images, pixels are often represented using color models such as RGB, where each pixel's value is a combination of red, green, and blue intensities. For example, a pixel might have a value of (255, 0, 0) for pure red.
- ***Spatial Arrangement:*** Pixels are arranged in a grid, with each pixel having a specific location defined by its coordinates (x, y). The position determines how the image is displayed.
- **Pixel Density:** Measured in pixels per inch (PPI) or dots per inch (DPI). Higher pixel density results in sharper images, especially when printed.

Resolution

The resolution of an image refers to the total number of pixels it contains, typically expressed as width x height (e.g. 1920x1080). Higher resolution images have more pixels, leading to finer detail.

- **Common resolutions include:**
  - **HD (720p):** 1280 x 720
  - **Full HD (1080p):** 1920 x 1080
  - **4K:** 3840 x 2160
  - **8K:** 7680 x 4320
- **Spatial Resolution:** Refers to the density of pixels in an image, affecting clarity and detail.

Bits Per Pixel (BPP)

Bits Per Pixel (BPP) is a measure of the amount of data used to represent the color of a single pixel in a digital image. It indicates ***how many bits are allocated for each pixel*** in the image.

Color Depth:

BPP determines the color depth of an image, which influences the range of colors that can be displayed. Higher BPP values allow for more colors.

Higher BPP generally results in better image quality and smoother gradients because there are more color options available for each pixel.

For example:

- **1 BPP:** Can represent 2 colors (black and white).
- **8 BPP:** Can represent 256 colors (suitable for grayscale or indexed color images).
- **24 BPP:** Commonly used for true color images, allowing for over 16 million colors (256 shades for each of the three color channels: Red, Green, Blue).

File Size:

The total file size of an image is directly influenced by BPP. More bits per pixel mean more data to store, which can lead to larger file sizes.

For example, a 1920x1080 image at 24 BPP would have a size of approximately:

$$\text{File Size} = \text{Width} \times \text{Height} \times \text{BPP} / 8$$

$= 1920 \times 1080 \times 24/8 = 6{,}220{,}800 \text{ bytes} \approx 6.22 \text{ MB}$

Image Formation / Image Acquisition

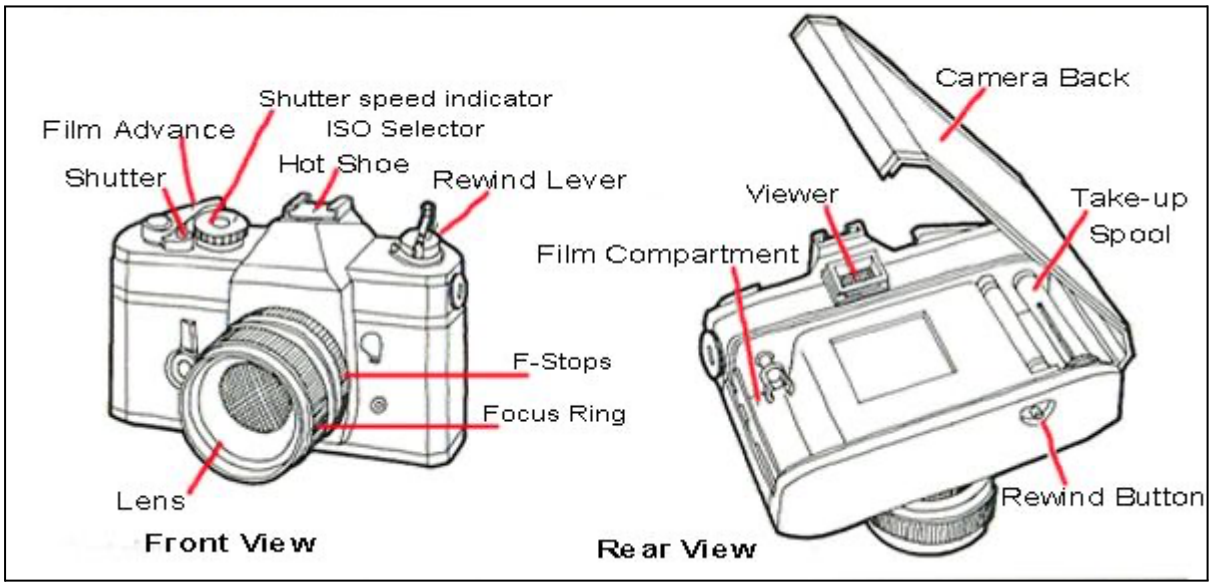
The process of converting the sensed light into a digital image format. This involves several steps and components:

- **Light Interaction:** Light reflects off surfaces, refracts through mediums, or emits from sources, forming the basis of image creation.
- **Optical System:** A lens or optical system focuses the incoming light onto a sensor (e.g., CCD, CMOS) or film. This can include various types of lenses that control focus, depth of field, and distortion.
- **Sensor Capture:** The sensor converts the light into electrical signals. Each pixel corresponds to a specific point in the scene, capturing intensity (and sometimes color) information.
- **Digitization:** The electrical signals are converted into digital form through sampling and quantization. This involves:
  - **Sampling:** Choosing discrete points in the scene to represent the image.
  - **Quantization:** Assigning a finite number of values to the sampled points (e.g., 256 levels for 8-bit grayscale).
- **Image Representation:** The resulting digital image is stored as a matrix of pixel values, where each value represents brightness (in grayscale) or color (in RGB or other color spaces).

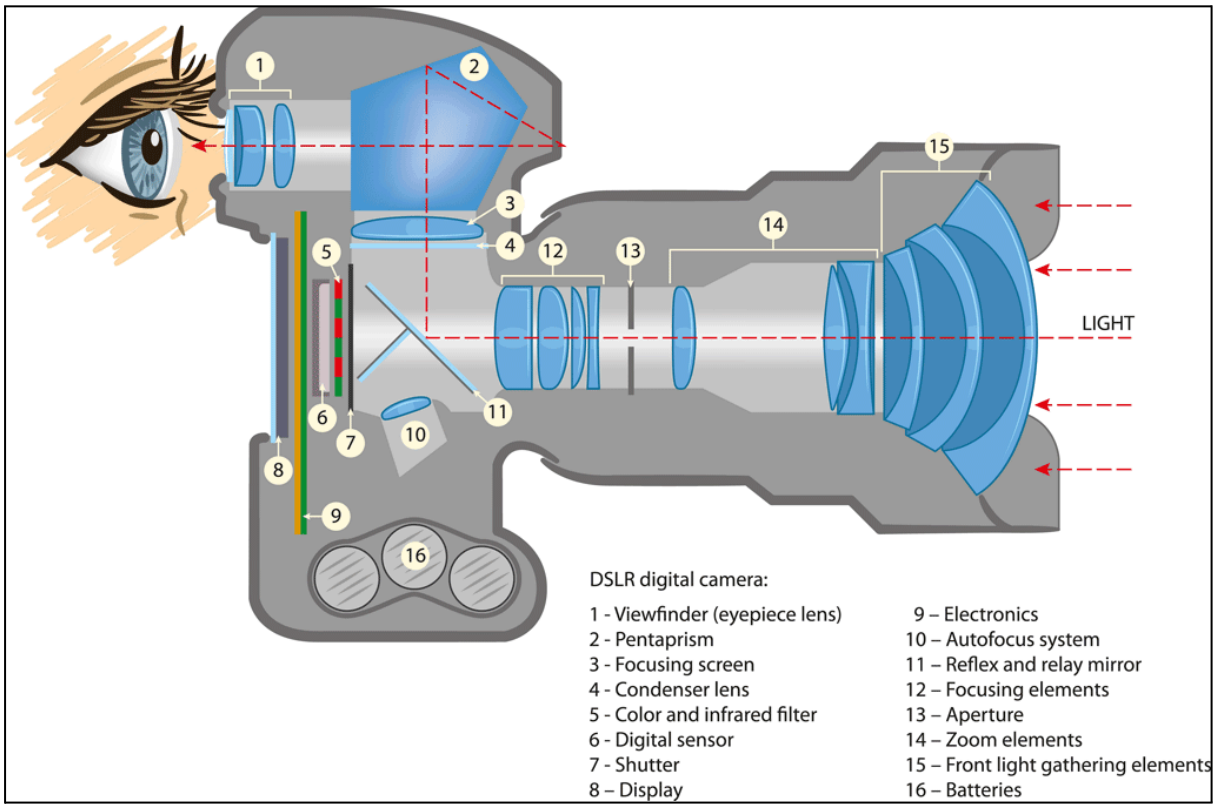
Digital vs. Analog Image Formation

- **Analog Imaging:**
  - Uses continuous signals to represent images.
  - Light is captured on a photosensitive medium (like film) through a chemical process.
  - Light strikes the film coated with silver halide crystals, causing a chemical reaction that forms a latent image.
  - This image is developed through a series of chemical baths.
  - Images are stored on physical film.
- **Digital Imaging:**
  - Uses discrete signals to represent images.
  - Light is captured by sensors (like **CCD** or **CMOS**), which convert the light into digital data through sampling and quantization.
  - Each pixel in the sensor measures light intensity and converts it into an electrical signal, creating a digital representation of the image.
  - Images are stored as digital files (e.g., JPEG, RAW, PNG) on memory cards, hard drives, or cloud storage. They can be easily edited, shared, and reproduced.

Analog SLR 35-millimeter camera parts



Digital DSLR Camera Parts



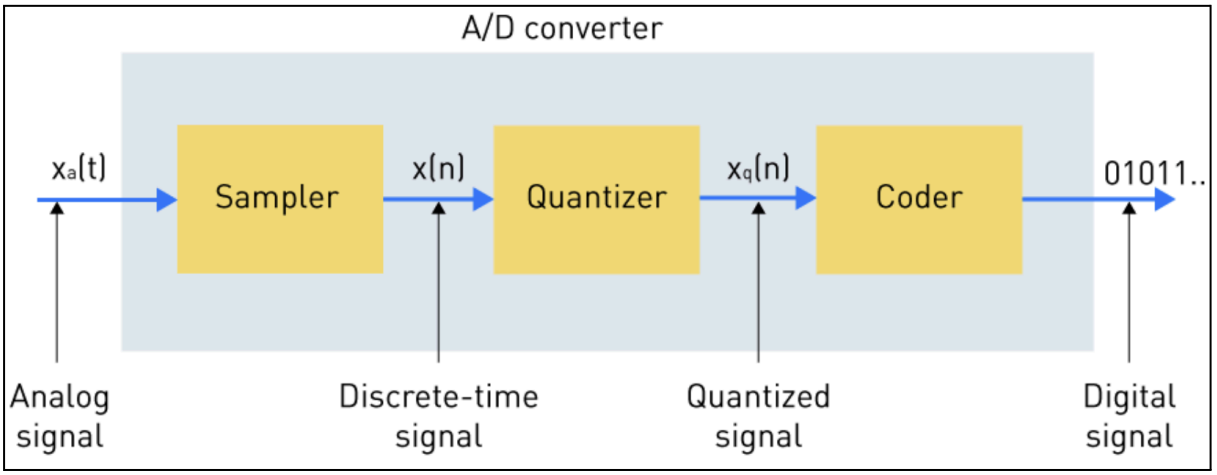
Digitization

Digitization is the process of ***converting analog information into a digital format***.

In order to become suitable for digital processing, an image function f(x,y) must be digitized both spatially and in amplitude. It is the combined operations of sampling and quantization, also called analog-to-digital (A/D) conversion.

This digitization process involves two main processes called

1. **Sampling:** Digitizing the ***coordinate value*** is called sampling.
2. **Quantization:** Digitizing the ***amplitude value*** is called quantization



Sampling

Sampling is the process of *converting a continuous signal (analog signal) into a discrete signal* by measuring its amplitude at regular intervals.

Key Points:

- A ***sample*** is a numerical value representing the *height of a waveform at a particular time*, or *the brightness of an image at a particular point*.
- A ***digital signal*** is a set of sampled values represented in binary form as bits (binary digits) that can be turned back into the original form.

We see and hear ***continuous variations*** in color, light, and sound. Digital devices can record and process only ***sequences of numbers***, not the original sounds, pictures, and movies.

How is it possible that information can be turned into a list of numbers without losing any quality?

Sampling is the process of recording values (samples) of a signal at distinct points in time or space.

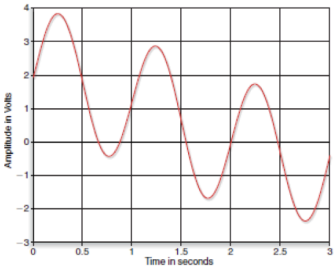
- Waveforms are sampled in time.
- Images are sampled in space.
- Time-varying scenes (video) are sampled in both time and space.

Let's understand the basic concept of sampling. For simplicity, we will focus on ***sampling of a waveform in time***.

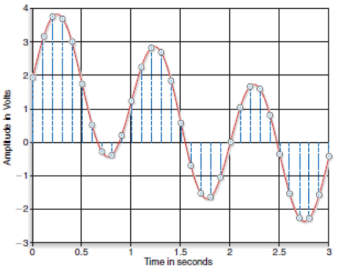
Sampling of a waveform

Sampling of a waveform refers to the process of recording values at distinct time points along a signal.

- Usually, samples of a signal are observed at ***uniformly-spaced time instants***, e.g., every 0.1 sec, as shown below.
- The ***heights*** of the dots above or below the ***time axis (horizontal axis)*** represent the samples.

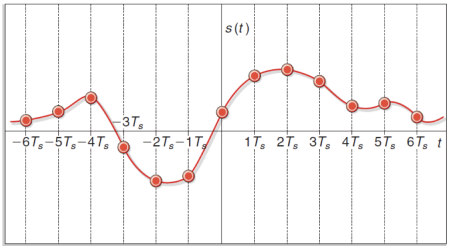


analog signal



samples of the analog signal

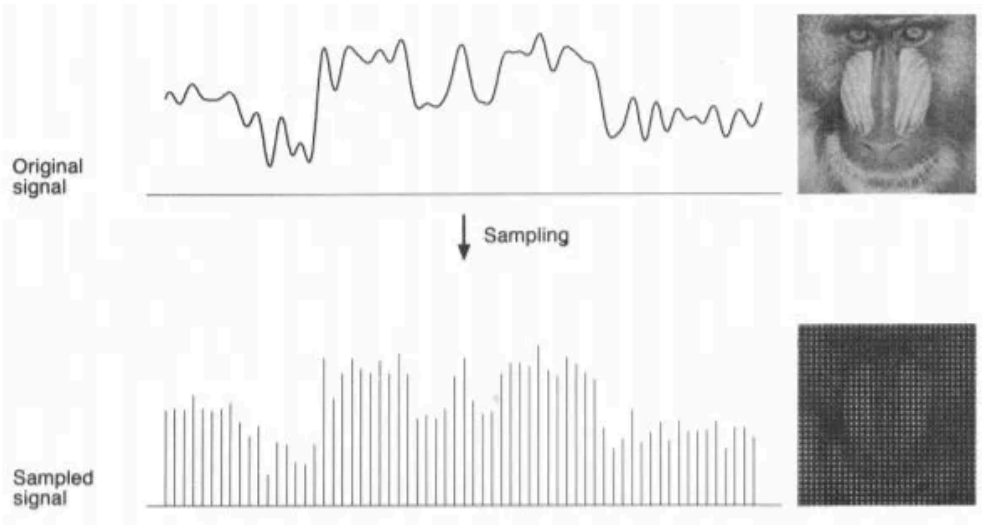
- ➢ The samples of a signal are a sequence of numbers s[n].
  - s[n] = s(nT) where n= 0, +-1, +-2,...
- ➢ A sequence of samples is the set of numerical sample values of a signal.



sampled values of an analog signal s(t)

- ➢ **The sampling period Ts** is the spacing between two adjacent samples, i.e., *seconds per sample*.
- ➢ **The sampling rate or frequency fs** is the number of *samples per second* (Hz).
- ➢ A higher sampling rate captures more detail from the original signal, resulting in better quality.

$$f_s = \frac{1}{T_s}$$



For an analog signal to be reconstructed from the digitized signal, the sampling rate should be highly considered. The ***rate of sampling*** should be such that the data in the message signal should neither be lost nor it should get over-lapped. Hence, a rate was fixed for this, called the ***Nyquist rate***.

Sampling Theorem

The Sampling Theorem, also known as the ***Nyquist-Shannon Sampling Theorem***, states that a continuous signal can be accurately represented and reconstructed from its samples if it is sampled at a rate *greater than twice its highest frequency component*.

Nyquist Rate

The minimum sampling rate required to *avoid **aliasing***, which occurs when higher frequency components are misrepresented as lower frequencies.

The Nyquist rate is defined as *twice the maximum frequency* of the signal.

For example, if an audio signal contains frequencies up to 20 kHz, it should be sampled at a minimum of 40 kHz.

Which means,

$$f_S=2W$$

f_S is the sampling rate

W is the highest frequency

Aliasing

If a signal is sampled below the Nyquist rate, higher frequency components can be misinterpreted as lower frequencies, *leading to distortion in the reconstructed signal*.

This phenomenon can produce unwanted artifacts and loss of fidelity in audio and images.

Quantization

Quantization is opposite to sampling because it is done on the “y axis” while sampling is done on “x axis”.

Each sampled value is quantized to the nearest value from a finite set of discrete levels, introducing quantization error. Quantization is the process of mapping a continuous range of values (such as the amplitude of an analog signal) into a finite range of discrete values.

- Quantization comes after sampling as a crucial step.
- The quantization of an analog signal is done by discretizing the signal with a number of ***quantization levels***.

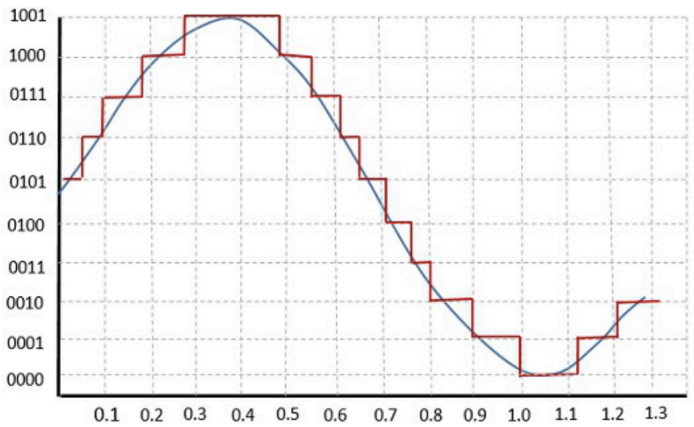


Image Representation

Types of Images

1. Bitmap (Raster) Images

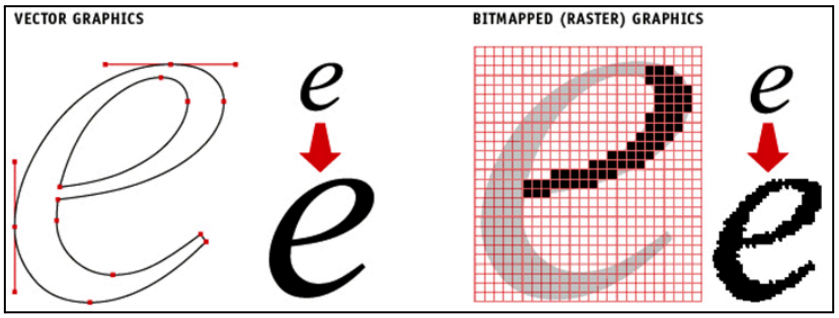
Composed of a grid of pixels, where each pixel has a defined color.

- Resolution-dependent: Image quality decreases when scaled up.
- File formats include JPEG, PNG, GIF, BMP, and TIFF.
- Applications: Photographs, detailed images, and any graphics requiring intricate color detail.

2. Vector Images

Made up of paths defined by mathematical equations, allowing for scalability without loss of quality.

- Resolution-independent: Can be resized without losing clarity.
- Common file formats include SVG, AI (Adobe Illustrator), and EPS.
- Applications: Logos, icons, illustrations, and graphics requiring clean lines and scalability.



3. Grayscale Images

Represent images using shades of gray, with no color information.

- Each pixel represents an intensity level (0 for black, 255 for white in 8-bit images).
- Smaller file sizes compared to color images.
- Applications: Medical imaging (like X-rays), documents, and artistic photography.

4. Color Images

Images that contain color information, represented typically in RGB (Red, Green, Blue).

- Can display millions of colors, depending on bit depth (e.g., 24-bit images have 16.7 million possible colors).
- Applications: Photography, digital art, and web graphics.

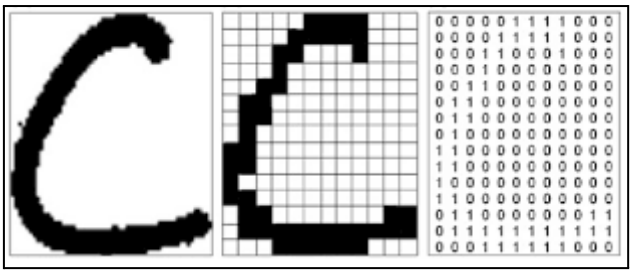
5. High Dynamic Range (HDR) Images

Capture a wider range of brightness levels than standard images, combining multiple exposures.

- Allows for more detail in highlights and shadows.
- Typically saved in formats like HDR or EXR.
- Applications: Landscape photography, video games, and visual effects.

Binary Images

A binary image consists of pixels that can have only two values, typically 0 and 1, representing two colors (commonly black and white).



- **Pixel Values:**
  - 0: Represents one color (e.g., black).
  - 1: Represents the other color (e.g., white).
- Color Depth: 1 bit per pixel, allowing for only two levels of intensity.
- They have a very small file size compared to grayscale or color images.
- Quality can decrease when the image is resized larger than its original dimensions.

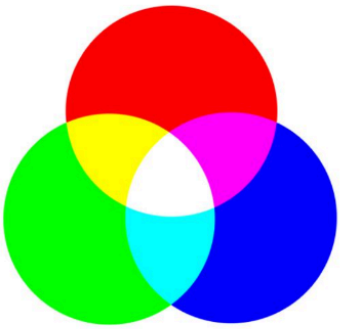
Grayscale images

Grayscale images are digital images that represent visual information in shades of gray, ranging from black to white. Each pixel in a grayscale image contains intensity information but no color information.

- **Pixel Values:**
  - Grayscale images typically use 8 bits per pixel, allowing for 256 different shades (from 0 for black to 255 for white).
  - In higher bit depths (e.g., 16 bits), they can represent 65,536 shades.
- **Color Depth:** Commonly represented in 8-bit (1 byte) format for standard grayscale images.
- **Resolution:** Like other bitmap images, the resolution is important; enlarging the image can lead to pixelation.
- **File Formats:** Common formats for grayscale images include PNG, JPEG, TIFF, and BMP.

RGB (Red, Green, Blue)

Definition: The RGB color model combines red, green, and blue light in varying intensities to create a broad spectrum of colors.

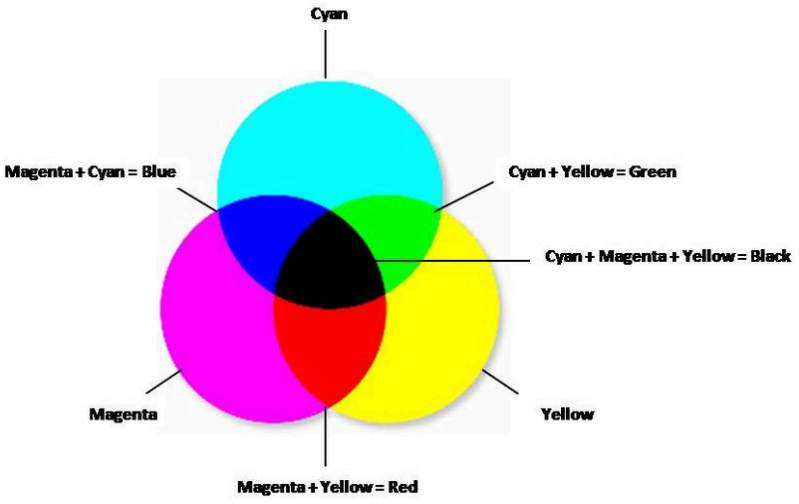


- Colors are created by adding light. When all colors are combined at full intensity, they produce white.
- Pixel Representation: Each color is represented by three values (R, G, B), typically ranging from 0 to 255 in 8-bit images.
- Color Space: Commonly used in digital screens, cameras, and scanners.

CMYK (Cyan, Magenta, Yellow and Black)

CMYK color model is widely used in printers. It stands for Cyan, Magenta, Yellow and Black (key).

- It is a subtractive color model.
  - 1 - RGB = CMY

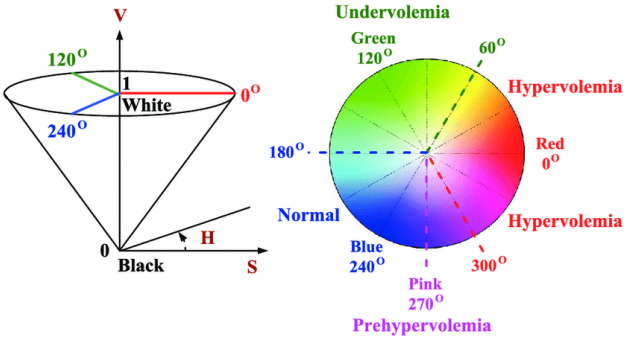


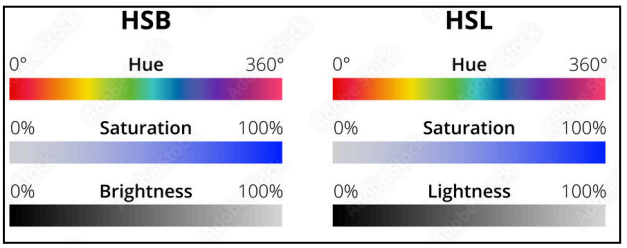
- Each color is represented by four values, indicating the percentage of each ink used (0% to 100%).
  - For example, 0% C, 100% M, 100% Y, and 0% K would produce a vivid red.

HSV (Hue, Saturation, Value)

The image consists of three channels. Hue, Saturation and Value are three channels.

- This color model does not use primary colors directly. It uses color in the way humans perceive them.
- HSV color is represented by a cone.





- **Hue:** Represents the color type (e.g., red, green, blue) and is measured as an angle on a color wheel (0° to 360°).
- **Saturation:** Indicates the intensity or purity of the color (0% is gray, 100% is the full color).
- **Value:** Represents the brightness of the color (0% is black, 100% is the brightest version of the color).

YIQ

The YIQ color model is used primarily in black-and-white television broadcasting in North America.

- It separates the luminance (Y) from the chrominance (I and Q) components.

From RGB to YIQ

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \approx \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.5959 & -0.2746 & -0.3213 \\ 0.2115 & -0.5227 & 0.3112 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

From YIQ to RGB

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.619 \\ 1 & -0.272 & -0.647 \\ 1 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

- **Y (Luminance):** Represents the brightness of the image.
- The y value is similar to the grayscale part. The color information is represented by the IQ part.
- **I (In-phase):** Represents the color information, specifically the orange-blue component.
- **Q (Quadrature):** Represents the color information, specifically the purple-green component.

Image Formats

An image file format is a file format for a digital image. There are many formats that can be used, such as JPEG, PNG, and GIF.

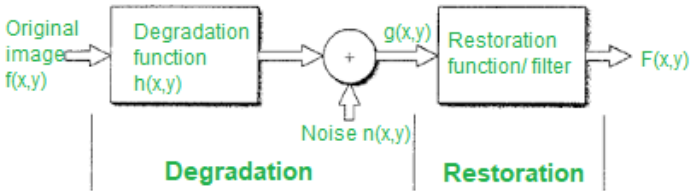
- ➔ The data stored in an image file format may be ***compressed*** or ***uncompressed***.
- ➔ If the data is compressed, it may be done so using ***lossy compression*** or ***lossless compression***.
- ➔ For graphic design applications, vector formats are often used. Some image file formats support transparency.

A comparison table highlighting the key differences among common image formats:

Format	Compression Type	Color Depth	Transparency Support	Animation Support	Typical Use Cases
BMP	Uncompressed	1-bit, 8-bit, 24-bit	No	No	High-quality images, graphics applications
JPEG	Lossy	Up to 24-bit	No	No	Photographs, web images, digital photography
PNG	Lossless	Up to 24-bit	Yes	No	Graphics, logos, images requiring transparency
GIF	Lossless	8-bit (256 colors)	Yes	Yes	Simple graphics, animations, web use
TIFF	Lossless/Lossy	Variable (up to 32-bit)	Yes	No	Professional photography, printing
WEBP	Lossy/Lossless	Up to 24-bit	Yes	Yes	Web images, improved loading times
SVG	Vector (not pixel-based)	N/A	Yes	Yes	Logos, icons, scalable graphics for web
HEIF	Lossy	Up to 16-bit	Yes	No	Mobile photography, efficient storage

Noise Models in DIP

The principal source of noise in digital images arises during image acquisition and transmission. Generally, a mathematical model of image degradation and its restoration is used for processing.



The figure above shows the presence of a degradation function h(x,y) and an external noise n(x,y) component coming into the original image signal f(x,y) thereby producing a final degraded image g(x,y).

$$g(x,y)=h(x,y)*f(x,y)+n(x,y)$$

Different types of noise have different probability density functions (PDFs). Here are some common noise models used in DIP:

1. **Gaussian Noise:** Normal distribution.
    - **Characteristics:** Random variations in pixel values, typically caused by sensor noise and electronic interference.
    - **Effect:** Smooths overall brightness but can degrade fine details.
2. **Salt-and-Pepper Noise**
    - **PDF:** Discrete distribution with spikes at minimum and maximum values.
    - **Characteristics:** Randomly replaces pixels with maximum (white) and minimum (black) values, simulating pixel corruption.
    - **Effect:** Introduces sharp disturbances, often requiring median filtering for removal.
3. **Poisson Noise:** Poisson distribution.
    - **Characteristics:** The noise level varies with the signal intensity, common in low-light conditions.
    - **Effect:** More prominent in brighter areas, leading to high variance.
4. **Speckle Noise**
    - **PDF:** Often modeled as multiplicative noise, related to the signal intensity.
    - **Characteristics:** Common in coherent imaging systems like ultrasound and radar.
    - **Effect:** Produces a grainy texture, complicating feature extraction.
5. **Uniform Noise:** Uniform distribution across a specified range.
    - **Characteristics:** Each pixel has an equal chance of being any value within a certain range.
    - **Effect:** Causes a constant shift, impacting overall image contrast.
6. **Burr Noise:** Heavy-tailed distribution.
    - **Characteristics:** Can introduce significant outliers in the image data.
    - **Effect:** May require robust filtering techniques for effective restoration.

Adding Noise to an Image Using OpenCV

```
import cv2
import numpy as np
import random

def add_gaussian_noise(image, mean=0, sigma=25):
    gauss = np.random.normal(mean, sigma, image.shape).astype('uint8')
    noisy = cv2.add(image, gauss)
    return noisy

def add_rayleigh_noise(image, scale=25):
    noise = np.random.rayleigh(scale, image.shape).astype('uint8')
    noisy = cv2.add(image, noise)
    return noisy

def add_uniform_noise(image, low=0, high=50):
    noise = np.random.randint(low, high, image.shape).astype('uint8')
    noisy = cv2.add(image, noise)
    return noisy

def add_impulse_noise(image, prob=0.05):
    noisy = image.copy()
    total_pixels = image.size
    num_noise = int(prob * total_pixels)

    for _ in range(num_noise):
        x = random.randint(0, image.shape[0] - 1)
        y = random.randint(0, image.shape[1] - 1)
        noisy[x, y] = 255 if random.random() > 0.5 else 0  # Salt and pepper
    return noisy

# Load an image
image = cv2.imread('path_to_your_image.jpg')

# Add different types of noise
gaussian_noise = add_gaussian_noise(image)
rayleigh_noise = add_rayleigh_noise(image)
uniform_noise = add_uniform_noise(image)
impulse_noise = add_impulse_noise(image)
```

Color Space Transformation

From	To	Transformation Formula
RGB	Gray scale	$Y=0.2989 \cdot R+0.5870 \cdot G+0.1140 \cdot B$
RGB	HSV	1. Compute Cmax=max(R,G,B)
		2. Compute Cmin=min(R,G,B)
		3. Δ=Cmax−Cmin
		$$H = \begin{cases} 60 \cdot \frac{G-B}{\Delta} + 360 & \text{if } C_{max} = R \\ 60 \cdot \frac{B-R}{\Delta} + 120 & \text{if } C_{max} = G \\ 60 \cdot \frac{R-G}{\Delta} + 240 & \text{if } C_{max} = B \end{cases}$$ $$S = \begin{cases} 0 & \text{if } C_{max} = 0 \\ \frac{\Delta}{C_{max}} & \text{otherwise} \end{cases}$$ $$V = C_{max}$$

HSV	RGB	def hsv_to_rgb(h, s, v):
		  if s == 0: # Achromatic case
		  r = g = b = v * 255
		  else:
		  c = v * s
		  x = c * (1 - abs((h / 60) % 2 - 1))
		  m = v - c
		
		  if 0 <= h < 60:
		  r, g, b = c, x, 0
		  elif 60 <= h < 120:
		  r, g, b = x, c, 0
		  elif 120 <= h < 180:
		  r, g, b = 0, c, x
		  elif 180 <= h < 240:
		  r, g, b = 0, x, c
		  elif 240 <= h < 300:
		  r, g, b = x, 0, c
		  elif 300 <= h < 360:
		  r, g, b = c, 0, x
		  r = (r + m) * 255
		  g = (g + m) * 255
		  b = (b + m) * 255
		  return int(r), int(g), int(b)

RGB	LAB	1. Convert RGB to XYZ $$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

		2. Compute LAB from XYZ using specific formulas $$L^* = 116 \cdot f(Y/Y_n) - 16$$ $$a^* = 500 \cdot (f(X/X_n) - f(Y/Y_n))$$ $$b^* = 200 \cdot (f(Y/Y_n) - f(Z/Z_n))$$ Where $f(t)$ is defined as: $$f(t) = \begin{cases} t^{1/3} & \text{if } t > (6/29)^3 \\ \frac{t}{3(6/29)^2} + \frac{4}{29} & \text{otherwise} \end{cases}$$
LAB	RGB	1. Convert LAB to XYZ
		2. Convert XYZ to RGB $$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$ The inverse transformation is also done using a matrix that depends on the RGB space.

```
# Load an image in RGB format
image = cv2.imread('image.jpg')

# Convert BGR to RGB (OpenCV loads images in BGR by default)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert RGB to Grayscale
image_gray = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2GRAY)

# Convert RGB to HSV
image_hsv = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2HSV)

# Convert RGB to LAB
image_lab = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2LAB)
```

Geometric primitives and Transformation

Geometric primitives are basic shapes or elements that are used to describe the geometry of a scene or object.

Some common geometric primitives include:

- **Points:** Points are the most basic geometric primitive and are used to represent a location or position in space.
- **Lines:** Lines are composed of a series of connected points and can be used to represent straight or curved paths in space.
- **Circles and ellipses:** These primitives are used to represent round or oval shapes and can be defined by their center point and radius or major and minor axes.
- **Polygons:** Polygons are composed of a series of connected lines and can be used to represent shapes with straight edges, such as triangles, rectangles, and polygons with many sides.
- **Splines:** Splines are used to represent smooth, curved shapes and can be defined by a set of control points that determine the shape of the curve.

Primitive	Definition	Representation	Applications
Points	Basic element representing a location in space.	2D: $p = (x, y)$ Homogeneous: $p' = (x, y, 1)$	Location marking, feature detection
Lines	Composed of connected points; can be straight or curved.	2D: $ax + by + c = 0$ Homogeneous: $l' = (a, b, c)$	Edge detection, shape analysis
Conics	Curves from the intersection of a plane with a cone.	$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ Matrix: $[x, y, 1]C[x, y, 1]^T = 0$	Shape recognition, camera calibration
3D Points	Representing locations in 3D space.	Inhomogeneous: $x = (x, y, z)$ Homogeneous: $x' = (x, y, z, 1)$	3D modeling, spatial analysis
3D Planes	Flat surfaces extending infinitely in 3D space.	Homogeneous: $m' = (a, b, c, d)$ Plane equation: $ax + by + cz + d = 0$	Scene reconstruction, object placement
3D Lines	Straight paths defined by points in 3D space.	$r = p + \lambda d'$ Matrix: $L = \tilde{p}\tilde{q}^T - \tilde{q}\tilde{p}^T$	Pathfinding, trajectory analysis
3D Quadrics	Surfaces representing various 3D shapes.	$Q(x, y, z) = x^T Ax + 2b^T x + c$ Homogeneous: $Q(\tilde{x}) = \tilde{x}^T Q\tilde{x} = 0$	Object modeling, surface fitting

2-D Geometric Transformations Cheat Sheet

Transformation	Description	Mathematical Representation	Transformation Matrix
Translation	Moves points or shapes by a fixed distance in the x and y directions.	$p' = p + \begin{pmatrix} tx \\ ty \end{pmatrix}$	$T = \begin{pmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{pmatrix}$
Scaling	Changes the size of an object by a scale factor in the x and y directions.	$p' = p \cdot s$ where $s = \begin{pmatrix} sx \\ sy \end{pmatrix}$	$S = \begin{pmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Rotation	Rotates points around the origin by a specified angle θ.	$p' = R\theta \cdot p$	$R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Shearing	Distorts the shape by shifting points in one direction.	$p' = p + \begin{pmatrix} shx \cdot y \\ shy \cdot x \end{pmatrix}$	$H = \begin{pmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Reflection	Flips points over a specified line (e.g., x-axis, y-axis, or any line).	$p' = R \cdot p$ (depends on the axis of reflection)	Reflection over x-axis: $Rx = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ Reflection over y-axis: $Ry = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Affine Transformation	Combines translation, scaling, rotation, and shearing in a single transformation.	General form: $p' = A \cdot p + b$	$A = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$ with $b = \begin{pmatrix} tx \\ ty \\ 0 \end{pmatrix}$

OpenCV Tutorial.ipynb

Image Filtering

Image filtering techniques are methods used to enhance or modify images by altering pixel values based on their neighborhood. These techniques are essential in digital image processing for tasks such as noise reduction, feature extraction, and image enhancement.

Convolution

Convolution is a mathematical operation that combines two functions to produce a third function, expressing how the shape of one is modified by the other.

In the context of image processing, it involves a kernel (or filter) that is applied to an image to extract features or modify the image in some way.

The basic idea behind convolution is to slide a small matrix called a kernel or filter over an image and perform a pixel-wise multiplication between the kernel and the corresponding image patch. The results of these multiplications are summed up to obtain a new value for the central pixel in the output image.

Mathematically, For a discrete function (such as a digital image), the convolution of an image I with a kernel K is defined as:

$$(I * K)(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} I(m, n) \cdot K(x - m, y - n)$$

```
# Perform convolution using filter2D function
filtered_image = cv2.filter2D(src=img, ddepth=-1, kernel, anchor=None, delta=0,
borderType=cv2.BORDER_DEFAULT)
```

No change - kernel

```
kernel = np.array([
    [0, 0, 0],
    [0, 1, 0],
    [0, 0, 0]
])

# Perform convolution using filter2D function
filtered_image = cv2.filter2D(image, -1, kernel)
```

Shifted Right - kernel

```
right_shift_kernel = np.array([
    [0, 0, 0],
    [0, 0, 1],
    [0, 0, 0]
])

# Perform convolution using filter2D function
filtered_image = cv2.filter2D(image, -1, right_shift_kernel)
```

Images also can be filtered with various low-pass filters (LPF), high-pass filters (HPF), etc. LPF helps in removing noise, blurring images, etc. HPF filters help in finding edges in images.

Image Blurring

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (eg: noise, edges) from the image.

Spatial Filtering

Spatial filtering ***operates directly on the pixel values*** in the image domain. It applies a convolution operation where a filter kernel (a small matrix) is used to perform calculations on neighboring pixels, influencing the value of the center pixel.

Types of Spatial Filtering

1. Linear Spatial Filtering

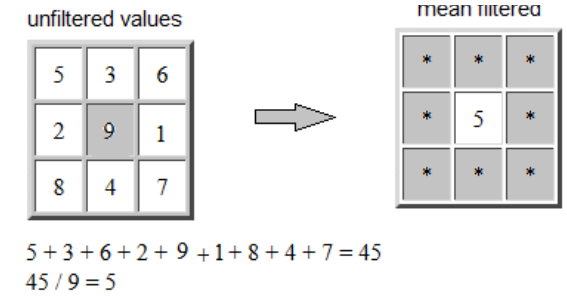
Linear filters apply a linear combination of pixel values based on the weights defined in the filter kernel.

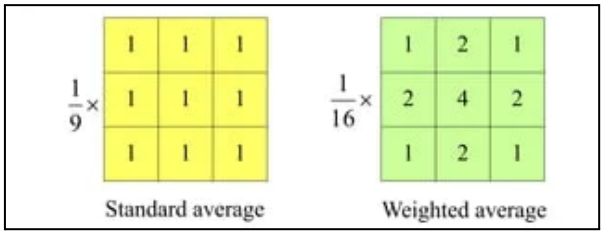
Mean Filter (Average Filter):

Purpose: Reduces noise by averaging the pixel values in a neighborhood.

Below are the types of mean filter:

- **Averaging filter:** It is used in reduction of the detail in image. All coefficients are equal.
- **Weighted averaging filter:** In this, pixels are multiplied by different coefficients. Center pixel is multiplied by a higher value than the average filter.





Effect: Smoothens the image but can blur edges.

```
# Apply the mean filter using filter2D function
kernel = np.ones((3, 3), np.float32) / 9
meanFilter_image = cv2.filter2D(image, -1, kernel)
```

cv2.boxFilter()

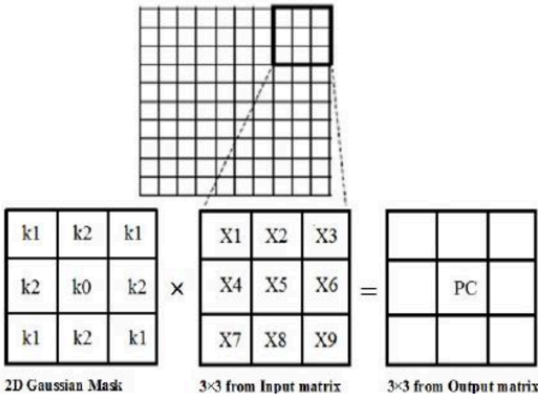
A box filter, also known as an average filter, is a simple linear filter used in image processing for smoothing and blurring images. It works by averaging the pixel values within a defined rectangular (or square) neighborhood around each pixel in the image.

```
boxFilter_image = cv2.boxFilter(image,-1,(3,3))
```

Gaussian Filter

Purpose: Applies a Gaussian function to weight neighboring pixels, allowing for a smoother blur.

Effect: Better preserves edges compared to the mean filter.



It is based on the Gaussian function, which is a bell-shaped curve that describes a probability distribution.

The Gaussian kernel is defined by two parameters: the standard deviation (σ) and the size of the kernel (often denoted by N).

G(x, y) = (1 / (2 * π * σ^2)) * exp(-(x^2 + y^2) / (2 * σ^2))

```
import numpy as np

def generate_gaussian_kernel(size, sigma):
    if size % 2 == 0:
        raise ValueError("Kernel size must be odd.")
    kernel = np.zeros((size, size))
    center = size // 2
    for i in range(size):
        for j in range(size):
            x, y = i - center, j - center
            kernel[i, j] = np.exp(-(x**2 + y**2) / (2 * sigma**2))
    kernel /= np.sum(kernel)  # Normalize the kernel to ensure its sum is 1
    return kernel

size = 3  # Size of the kernel
sigma = 1.0  # Standard deviation
kernel = generate_gaussian_kernel(size, sigma)
```

```
[[0.07511361 0.1238414  0.07511361]
 [0.1238414  0.20417996 0.1238414 ]
 [0.07511361 0.1238414  0.07511361]]
```

```
GaussianFilter_image = cv2.filter2D(image, -1, kernel)
```

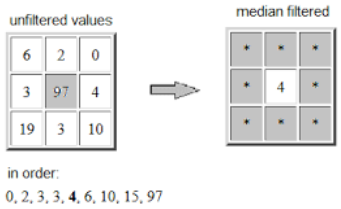
```
GaussianBlurFilter_image = cv2.GaussianBlur(image,(3,3),1)
```

2. Non-Linear Spatial Filtering

Non-linear filters use non-linear operations to process pixel values, making them more effective for certain tasks like edge preservation.

Median Filter

Purpose: Replaces a pixel with the median of the pixel values in its neighborhood.



Effect: Effectively ***removes salt-and-pepper noise*** while preserving edges.

```
MedianBlurFilter_image = cv2.medianBlur(image,3)
```

Bilateral Filter

Purpose: Smooths images while preserving edges by considering both spatial distance and intensity difference. Reduces noise without significant loss of detail.

$$I_{\text{filtered}}(x,y) = \frac{1}{W_p} \sum_{i,j} I(i,j) \cdot G_d(d_{ij}) \cdot G_r(r_{ij})$$

Where:
$G_d(d_{ij})$: Spatial Gaussian based on the distance between pixels.

$$G_d(d_{ij}) = e^{-\frac{d_{ij}^2}{2\sigma_d^2}} \qquad d_{ij} = \sqrt{(x-i)^2 + (y-j)^2}$$

$G_r(r_{ij})$: Range Gaussian based on the intensity difference.

$$G_r(r_{ij}) = e^{-\frac{r_{ij}^2}{2\sigma_r^2}} \qquad r_{ij} = |I(x,y) - I(i,j)|$$

W_p : Normalization factor to ensure the weights sum to one.

```
Bilateral_filtered_image = cv2.bilateralFilter(image, d = 10, sigmaColor = 75, sigmaSpace = 75)
```

Max Filter - dilate()

It is used to enhance bright regions or extract the maximum pixel values within a neighborhood

```
dilated_image = cv2.dilate(image, kernel, iterations=1)
```

Min Filter - erode()

The min filter, also known as erosion in morphological image processing, is a technique used to enhance features in an image by replacing each pixel with the minimum value from its neighborhood defined by a structuring element (kernel).

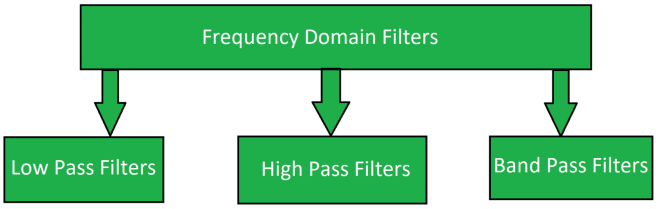
```
MinBlurFilter_image = cv2.erode(image,kernel, iterations=1)
```

Note:

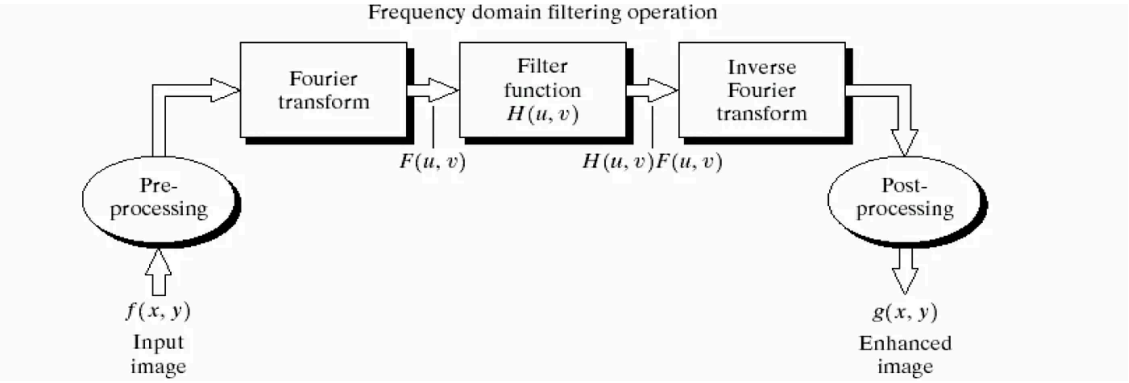
- Dilation: Expands the boundaries of objects.
- Erosion: Shrinks the boundaries, useful for removing small noise.

Frequency Domain Filters

Frequency Domain Filters are used for smoothing and sharpening of image by removal of high or low frequency components.
It is basically done for two basic operations i.e., Smoothing and Sharpening.



- ***Fourier Transform:*** The Fourier Transform (FT) converts an image from the spatial domain (pixel values) to the frequency domain (sine and cosine coefficients). The Inverse Fourier Transform (IFT) converts it back.
- For ***smoothing*** an image, a low filter is implemented and for ***sharpening*** an image, a high pass filter is implemented.



Low Pass filters (smoothing)

Low pass filter removes the high frequency components that means it keeps low frequency components. It is used to smoothen the image by attenuating high frequency components and preserving low frequency components.

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

Highpass filters (sharpening)

A high pass filter is used for passing high frequencies but the strength of the frequency is lower as compared to cut off frequency. Sharpening is a high pass operation in the frequency domain. As a low pass filter, it also has standard forms such as Ideal highpass filter, Butterworth highpass filter, Gaussian high pass filter.

1. Ideal Lowpass Filters

Ideal low pass filters simply cut off all high frequency components that are a specified distance D0 from the origin of the transform.

Below is the transfer function of an ideal lowpass filter.

$$H(u,v) = \begin{cases} 1 & if\ D(u,v) \le D0 \\ 0 & if\ D(u,v) > D0 \end{cases}$$

$$D(u,v) = \left[\left(u - \frac{M}{2}\right)^2 + \left(v - \frac{N}{2}\right)^2\right]^{\frac{1}{2}}$$

2. Butterworth Low Pass Filters

Butterworth Low Pass Filter is used to remove high-frequency noise with very minimal loss of signal components.

$$H(u,v) = \frac{1}{1+\left[\frac{D(u,v)}{D0}\right]^{2n}}$$

3. Gaussian Low Pass Filters

The transfer function of Gaussian Low Pass filters is shown below:

$$H(u,v) = e^{-D^2(u,v)/2D_0^2}$$

```
def apply_filter(image, filter_type='low', cutoff=30):
    # Convert image to float32
    image_float = np.float32(image) / 255.0

    # Perform Fourier Transform
    dft = np.fft.fft2(image_float)
    dft_shifted = np.fft.fftshift(dft)  # Shift zero frequency components to the center

    # Create filter mask
    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2
    x = np.linspace(-ccol, ccol - 1, cols)
    y = np.linspace(-crow, crow - 1, rows)
    X, Y = np.meshgrid(x, y)
    D = np.sqrt(X**2 + Y**2)  # Distance from the center

    if filter_type == 'low':
        # Low-Pass Filter
        mask = np.exp(-(D**2) / (2 * (cutoff**2)))
    elif filter_type == 'high':
        # High-Pass Filter
        mask = 1 - np.exp(-(D**2) / (2 * (cutoff**2)))

    # Apply filter
    filtered_dft = dft_shifted * mask
    filtered_image = np.fft.ifft2(np.fft.ifftshift(filtered_dft))
    filtered_image = np.abs(filtered_image)

    return filtered_image

# Load image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply low-pass filter
low_pass_image = apply_filter(image, filter_type='low', cutoff=30)

# Apply high-pass filter
high_pass_image = apply_filter(image, filter_type='high', cutoff=30)
```

Window Size for a Moving Average Filter?

The window size of a moving average filter is determined by the number of points to be used in computing the average. This number is usually referred to as the “order” of the filter.

To decide the window size for a moving average filter, consider these key factors:

- ➔ Larger windows reduce noise more effectively but can blur details, while smaller windows preserve details but may not smooth noise as well.
- ➔ For high-frequency noise, a larger window is better; for smooth signals, a smaller window works.
- ➔ ***Trial and Error:*** Start with standard sizes (like 3x3 or 5x5 for images) and test different sizes, evaluating the results visually or quantitatively.
- ➔ Larger windows increase computational load, so balance size with processing speed, especially in real-time applications.

Unsharp Masking

Unsharp masking is a popular ***image enhancement technique*** used primarily to improve the sharpness and clarity of an image.

Mathematical Representation

The unsharp masking process can be mathematically described as follows:

- Let I be the original image.
- Let G be the blurred image created by convolving I with a Gaussian kernel.
- The unsharp mask MM can be computed as:
  $$M = I - G$$
- The sharpened image SS is then calculated by:
  $$S = I + k \cdot M$$

Where:

- k is a scaling factor that determines the amount of sharpening applied. Typically, k is a positive constant.

Parameters

Unsharp masking involves a few key parameters that influence the final result:

- **Radius**:
  - Determines the extent of the Gaussian blur applied to the original image. A larger radius will result in a broader effect, emphasizing larger features and edges, while a smaller radius focuses on finer details.
- **Amount**:
  - Controls the strength of the effect (the value of k). A higher amount increases the contrast at the edges, resulting in a sharper image.
- **Threshold**:
  - Specifies which areas of the image are affected by the sharpening. Pixels with intensity differences below a certain threshold will not be sharpened, which helps to prevent noise amplification in flat areas or regions of low detail.

Edge Detection Filters

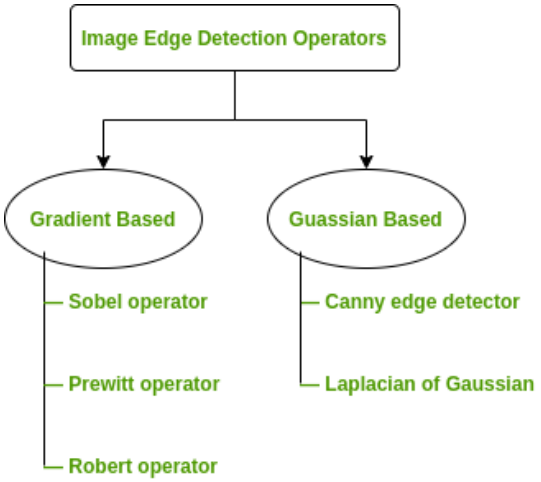
Edges are significant local changes of intensity in a digital image. An edge can be defined as a set of connected pixels that forms a boundary between two disjoint regions.

There are three types of edges:

1. Horizontal edges
2. Vertical edges
3. Diagonal edges

Edge Detection Operators are of two types:

1. **Gradient –** based operator which computes **first-order derivations** in a digital image like, Sobel operator, Prewitt operator, Robert operator
2. **Gaussian –** based operator which computes **second-order derivations** in a digital image like, Canny edge detector, Laplacian of Gaussian



All the masks that are used for edge detection are also known as derivative masks or sometimes referred as ***image gradients***

Note

All the derivative masks should have the following properties:

- Opposite signs should be present in the mask.
- Sum of the mask should be equal to zero.
- More weight means more edge detection.

First-Order Derivatives

The first-order derivative measures the rate of change of pixel intensity values in an image. In a 2D image, the intensity function can be represented as I(x,y), where x and y are the spatial coordinates.

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

Mathematical Representation

1. **Gradient**: The gradient vector ∇I of the image is given by:

$$\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)$$

   - $\frac{\partial I}{\partial x}$ is the partial derivative with respect to the x-direction (horizontal).
   - $\frac{\partial I}{\partial y}$ is the partial derivative with respect to the y-direction (vertical).

2. **Magnitude of Gradient**: The magnitude of the gradient can be computed as:

$$G = \sqrt{\left(\frac{\partial I}{\partial x} \right)^2 + \left(\frac{\partial I}{\partial y} \right)^2}$$

3. **Direction of Gradient**: The direction (angle) of the gradient can be determined using:

$$\theta = \arctan\left(\frac{\partial I / \partial y}{\partial I / \partial x} \right)$$

Second-Order Derivatives

The second-order derivative measures the rate of change of the first-order derivative. It captures curvature and is used to identify regions of rapid intensity change, such as edges or corners.

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

Mathematical Representation

1. **Laplacian**: The Laplacian operator, which is a common second-order derivative, is defined as:

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

   - $\frac{\partial^2 I}{\partial x^2}$ is the second derivative with respect to the x-direction.
   - $\frac{\partial^2 I}{\partial y^2}$ is the second derivative with respect to the y-direction.

2. **Laplacian of Gaussian (LoG)**: This combines Gaussian smoothing with the Laplacian operator to reduce noise. It can be mathematically expressed as:

$$LoG(x, y) = \nabla^2 (G(x, y))$$

   where $G(x, y)$ is a Gaussian function defined as:

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

3. **Magnitude of the Laplacian**: The magnitude is often used for edge detection, with zero-crossings indicating edges.

Sobel Filter

The Sobel filter is a popular edge detection technique in image processing that emphasizes edges in an image *by calculating the gradient of the image intensity*. It uses two 3x3 convolution kernels, one for detecting horizontal edges and one for vertical edges.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Magnitude and Direction: The gradient magnitude can be computed using:

$$G = \sqrt{G_x^2 + G_y^2}$$

The direction of the gradient can be calculated using:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Edge Thresholding: To highlight the edges, a threshold can be applied to the gradient magnitude. Pixels with values above the threshold are considered edges.

```
    # using cv2.sobel(src, ddepth, dx, dy, ksize, scale, delta, borderType)
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

# Calculate gradient magnitude
magnitude = np.sqrt(sobel_x**2 + sobel_y**2)

# Normalize for display
magnitude = np.uint8(magnitude / np.max(magnitude) * 255)
```

- ❖ By modifying the weights, you can increase or decrease the sensitivity of the edge detection.

Prewitt Filter

Like the Sobel filter, it employs convolution with ***specific kernels*** to determine the edges in both horizontal and vertical directions.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

```
# Define Prewitt kernels
prewitt_x = np.array([[-1, 0, 1],
                      [-1, 0, 1],
                      [-1, 0, 1]])

prewitt_y = np.array([[1, 1, 1],
                      [0, 0, 0],
                      [-1, -1, -1]])

# Apply convolution using the Prewitt kernels
gradient_x = cv2.filter2D(image, -1, prewitt_x)
gradient_y = cv2.filter2D(image, -1, prewitt_y)
```

Robert operator

This gradient-based operator computes the sum of squares of the differences between diagonally adjacent pixels in an image through discrete differentiation. Then the gradient approximation is made.
It uses the following 2 x 2 kernels or masks –

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Scharr Operator

The Scharr operator is an advanced edge detection technique used in image processing, designed to provide better edge detection results compared to simpler operators like the Sobel and Roberts operators.

$$\begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \qquad \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

Scharr-x Scharr-y

Robinson Compass masks & Krisch Compass Masks

In these operators we take one mask and rotate it in all the 8 compass major directions that are following: North, North West, West, South West, South, South East, East, North East

The only difference between Robinson and Kirsch compass masks is that in Kirsch we have a standard mask but in Kirsch we change the mask according to our own requirements.

```
  # define north kernel
north_kernel = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
])
  # define north_west kernel
north_west_kernel = np.array([
    [0, 1, 2],
    [-1, 0, 1],
    [-2, -1, 0]
])
    # define west kernel
west_kernel = np.array([
    [1, 2, 1],
    [0, 0, 0],
    [-1, -2, -1]
])
    # define south_west kernel
south_west_kernel = np.array([
    [2, 1, 0],
    [1, 0, -1],
    [0, -1, -2]
])
```

```
    # define south kernel
south_kernel = np.array([
    [1, 0, -1],
    [2, 0, -2],
    [1, 0, -1]
])
    # define south_east kernel
south_east_kernel = np.array([
    [0, -1, -2],
    [1, 0, -1],
    [2, 1, 0]
])
    # define east kernel
east_kernel = np.array([
    [-1, -2, -1],
    [0, 0, 0],
    [1, 2, 1]
])
    # define north_east kernel
north_east_kernel = np.array([
    [-2, -1, 0],
    [-1, 0, 1],
    [0, 1, 2]
])
```

Laplacian Operator

The major difference between Laplacian and other operators like Prewitt, Sobel, Robinson and Kirsch is that these all are first order derivative masks but Laplacian is a second order derivative mask.
The Laplacian L(x,y) of an image with pixel intensity values I(x,y) is given by:

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

How is the Laplacian filter calculated?

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

f''(x) = f'(x+1) - f'(x)
       = f(x+2) - f(x+1) - f(x+1) + f(x)
       = f(x+2) - 2*f(x+1) + f(x)
similarly,
f''(y) = f(y+2) - 2*f(y+1) + f(y)
Applying this filter, and also its transposed, and adding the results, is equivalent to convolving with the kernel
I*L   =    I * f''(x,y)
      =    I*  (f''(x) + f''(y))
                [ 0, 0, 0        [ 0, 1, 0          [ 0, 1, 0
f''(x,y)   =      1,-2, 1   +     0,-2, 0     =      1,-4, 1
                  0, 0, 0 ]       0, 1, 0 ]          0, 1, 0 ]
```

```
     # Define Positive Laplacian kernel
laplacian_kernel = np.array([
    [0, 1, 0],
    [1, -4, 1],
    [0, 1, 0]], dtype=np.float32)

     # Define negative Laplacian kernel
laplacian_kernel = np.array([
    [0, -1, 0],
    [-1, 4, -1],
    [0, -1, 0]], dtype=np.float32)
```

Alternative Kernel: Another commonly used version is the following:

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

```
# Apply Laplacian filter
laplacian = cv2.Laplacian(image, cv2.CV_64F)
```

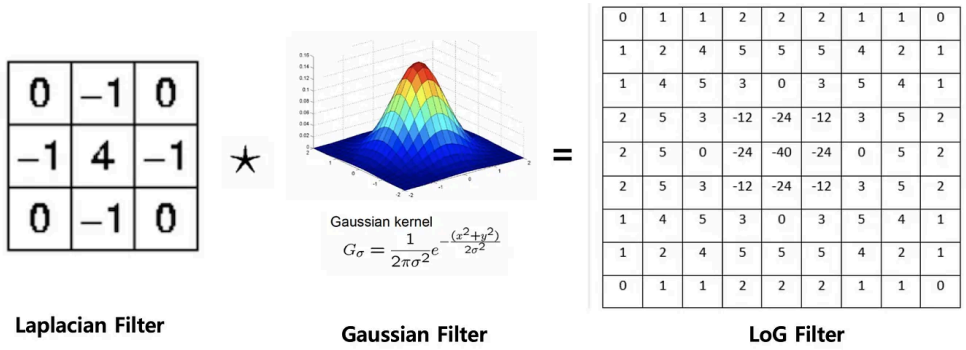
Laplacian of Gaussian (LoG)

The Laplacian filter is used to detect the edges in the images. But it has a disadvantage over noisy images. It amplifies the noise in the image.
Hence, first, we use a Gaussian filter on the noisy image to smoothen it and then subsequently use the Laplacian filter for edge detection.

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

1. Convert the image to grayscale
2. Apply Gaussian smoothing
3. Apply the Laplacian operator
4. Find zero-crossings or thresholding.

$$LoG = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$



Laplacian Filter **Gaussian Filter** **LoG Filter**

```
import cv2
# Load the image to grayscale
image = cv2.imread('doraemon.webp')

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Gaussian smoothing
blurred = cv2.GaussianBlur(gray, (5, 5), 0)

# Apply Laplacian operator
laplacian = cv2.Laplacian(blurred, cv2.CV_64F)

# Find zero crossings
zero_crossings = np.zeros_like(laplacian, dtype=np.uint8)
zero_crossings[laplacian > 0] = 255

# Thresholding
threshold = 20
edges = np.zeros_like(zero_crossings)
edges[zero_crossings > threshold] = 255
```

Edge detection using Local Variance

Once the variance is computed for each pixel, the next step is to set a threshold on the variance values. If the variance exceeds a certain threshold, this suggests that there is a significant change in pixel values, which corresponds to an edge. Otherwise, the region is considered smooth.

```
def edge_detection_using_variance(image, window_size=5, threshold=500):
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    rows, cols = gray_image.shape

    # Initialize an empty image for edges
    edges = np.zeros_like(gray_image)

    # Define the padding size
    pad_size = window_size // 2

    # Iterate over each pixel of the image
    for i in range(pad_size, rows - pad_size):
        for j in range(pad_size, cols - pad_size):
            # Extract the local window around the pixel (i, j)
            window = gray_image[i - pad_size:i + pad_size + 1, j - pad_size:j + pad_size + 1]

            # Calculate the mean and variance of the window
            mean = np.mean(window)
            variance = np.var(window)

            # If the variance is above the threshold, mark the pixel as an edge
            if variance > threshold:
                edges[i, j] = 255  # White pixel indicating an edge

    return edges
```

Canny Edge Detection Algorithm

Canny edge detection is a popular and effective edge detection algorithm that aims to identify the edges in an image while minimizing noise. Developed by John F. Canny in 1986, the algorithm is well-regarded for its accuracy and robustness.

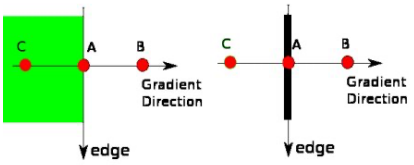
1. Convert the image to grayscale
2. Apply Gaussian smoothing
3. Compute Gradients: Calculate the gradient magnitude and direction of the smoothed image using derivative operators such as the Sobel operator.
4. Non-maximum suppression: Suppress non-maximum edges by thinning the edges to a single-pixel width.
5. Apply thresholding: to determine which edges to keep and which to discard

```
    # Apply Canny edge detection by using cv2.canny()
edges = cv2.Canny(image, threshold_low, threshold_high)
cv2_imshow(edges)
```

Non-Maximum Suppression

The image magnitude produced results in thick edges. Ideally, the final image should have thin edges. Thus, we must perform non maximum suppression to thin out the edges.

The goal is to refine the gradient magnitude obtained in the previous step, eliminating non-edge pixels and retaining only the strongest edge pixels.



Edge Thinning: For each pixel, NMS looks at the pixels along the direction of the gradient:

- It identifies the two neighboring pixels in the gradient direction.
- If the gradient magnitude of the current pixel is not greater than those of its neighbors, it is suppressed (set to zero).

```
# 2. Apply Gaussian smoothing
Guassian_blurred_image = cv2.GaussianBlur(image, (5, 5), 0)
cv2_imshow(Guassian_blurred_image)

# 3. Calculate gradients using Sobel operators
gradient_x = cv2.Sobel(Guassian_blurred_image, cv2.CV_64F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(Guassian_blurred_image, cv2.CV_64F, 0, 1, ksize=3)

# Compute the magnitude and direction of gradients
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
gradient_direction = np.arctan2(gradient_y, gradient_x)

# 4. Non-maximum suppression
suppressed_image = np.zeros_like(gradient_magnitude)
height, width = gradient_magnitude.shape

# at every pixel, the pixel is checked if it is a local maximum in its neighborhood in the direction of gradient.
for y in range(1, height - 1):
    for x in range(1, width - 1):
        angle = gradient_direction[y, x]
        if (0 <= angle < np.pi / 8) or (15 * np.pi / 8 <= angle <= 2 * np.pi):
            neighbor1 = gradient_magnitude[y, x + 1]
            neighbor2 = gradient_magnitude[y, x - 1]
        elif (np.pi / 8 <= angle < 3 * np.pi / 8) or (9 * np.pi / 8 <= angle < 11 * np.pi / 8):
            neighbor1 = gradient_magnitude[y + 1, x - 1]
            neighbor2 = gradient_magnitude[y - 1, x + 1]
        elif (3 * np.pi / 8 <= angle < 5 * np.pi / 8) or (11 * np.pi / 8 <= angle < 13 * np.pi / 8):
            neighbor1 = gradient_magnitude[y + 1, x]
            neighbor2 = gradient_magnitude[y - 1, x]
        else:
            neighbor1 = gradient_magnitude[y - 1, x - 1]
            neighbor2 = gradient_magnitude[y + 1, x + 1]
        if gradient_magnitude[y, x] >= neighbor1 and gradient_magnitude[y, x] >= neighbor2:
            suppressed_image [y, x] = gradient_magnitude[y, x]

# Thresholding
threshold_low = 50
threshold_high = 100
edges = np.zeros_like(suppressed_image )

# find strong_edges & weak_edges of each pixel according to threshold
strong_edges_y, strong_edges_x = np.where(suppressed_image  >= threshold_high)
weak_edges_y, weak_edges_x = np.where((suppressed_image  >= threshold_low) & (suppressed_image  <
threshold_high))
edges[strong_edges_y, strong_edges_x] = 255
edges[weak_edges_y, weak_edges_x] = 50
```

Image Restoration

Image restoration is the operation of taking a corrupt/noisy image and estimating the clean, original image. Corruption may come in many forms such as motion blur, noise and camera mis-focus.

Image degradation

Image degradation refers to the decline in quality of an image due to various factors, which can occur during acquisition, processing, or transmission. Here are some common types of image degradation:

1. **Noise**:
    - **Gaussian Noise**: Random variations in brightness or color that follow a Gaussian distribution.
    - **Salt-and-Pepper Noise**: Random occurrences of black and white pixels, resembling grains of salt and pepper.
    - **Poisson Noise**: Variability in image intensity due to random photon arrivals, common in low-light conditions.
2. **Blur**:
    - **Motion Blur**: Caused by the movement of the camera or subject during exposure.
    - **Gaussian Blur**: Uniform blurring applied for smoothing, which can lead to loss of detail.
    - **Out-of-Focus Blur**: Results from the camera being out of focus during capture.
3. **Distortion**:
    - **Geometric Distortion**: Alteration of the image shape due to lens imperfections (e.g., barrel or pincushion distortion).
    - **Chromatic Aberration**: Color fringing at the edges of objects, often caused by lens refraction.
4. **Compression Artifacts**:
    - **JPEG Artifacts**: Loss of quality and detail due to lossy compression, leading to blockiness and blurring.
    - **Color Banding**: Gradations of color are reduced to distinct bands rather than smooth transitions.
5. **Fading**:
    - Gradual loss of color and detail, often due to prolonged exposure to light or chemical reactions.

Image restoration techniques aim to *reverse the effects of degradation* and restore the image as closely as possible to its original or desired state.

The process involves analyzing the image and applying algorithms and filters to remove or reduce the degradations.

Image restoration can be broadly categorized into two main types:

1. spatial domain methods and
2. frequency domain methods.

Spatial domain techniques operate directly on the image pixels, while frequency domain methods transform the image into the frequency domain using techniques such as the Fourier transform, where restoration operations are performed.

1. Spatial Domain Methods

- **Filtering Techniques**:
    - **Linear Filters**: Such as averaging filters (mean filters) that reduce noise and blur an image.
    - **Median Filters**: Effective for removing salt-and-pepper noise by replacing each pixel value with the median of its neighbors.
    - **Gaussian Filters**: Used to smooth images and reduce noise while preserving edges to some extent.
- **Unsharp Masking**:
    - A sharpening technique that enhances the edges of an image by subtracting a blurred version of the image from the original.
- **Inverse Filtering**:
    - Attempts to reverse the effects of degradation (like blurring) by applying a filter that corresponds to the degradation function.
- **Adaptive Filtering**:
    - Filters that adjust their parameters based on the local image characteristics, effectively handling varying noise levels.
- **Image Restoration using Wiener Filter**:
    - A statistical approach that minimizes the mean square error between the estimated and the true image. It combines the degraded image and noise information.

2. Frequency Domain Methods

- **Frequency Domain Filtering**:
    - **Low-pass Filtering**: Removes high-frequency noise (such as Gaussian noise) while preserving low-frequency information (image structure).
    - **High-pass Filtering**: Enhances high-frequency components, which are often associated with edges and fine details.
- **Homomorphic Filtering**:
    - A technique that separates the illumination and reflectance components of an image, allowing for improved contrast and dynamic range.
- **Wavelet Transform**:
    - An advanced technique that provides a multi-resolution analysis of the image. It is effective in handling noise and provides good results in compressing and reconstructing images.

Inverse Filtering

➔ **Degradation Model:**

★ The process begins with a degradation model, often represented as:

$$g(x,y) = f(x,y) * h(x,y) + n(x,y)$$

where:

- ◆ g(x,y) and f(x,y) are the degraded image and the original image.
- ◆ h(x,y) is the point spread function (PSF) that describes the blurring process.
- ◆ n(x,y) is the noise in the image.

➔ **Fourier Transform:**

★ Inverse filtering typically involves transforming the images to the frequency domain using the Fourier transform.

$$G(u,v) = F(u,v) \cdot H(u,v) + N(u,v)$$

where G,F,H,and N are the Fourier transforms of the degraded image, original image, PSF, and noise, respectively.

★ **Inverse Filtering Equation:**

- ◆ The aim is to solve for F(u,v) = G(u,v) / H(u,v)

  or F(u,v) = G(u,v) / H(u,v) - N(u,v) / H(u,v)
- ◆ This equation provides a direct way to estimate the original image from the degraded image by dividing the Fourier transform of the degraded image by the PSF in the frequency domain.

```
                    # Function to create a Gaussian PSF
def gaussian_psf(size, sigma):
    x = np.linspace(-size//2, size//2, size)
    y = np.linspace(-size//2, size//2, size)
    x, y = np.meshgrid(x, y)
    psf = np.exp(-(x**2 + y**2) / (2 * sigma**2))
    psf /= np.sum(psf)  # Normalize the PSF
    return psf

                    # Inverse Filtering Function
def inverse_filter(degraded_img, psf):
    # Compute the Fourier Transforms
    G = np.fft.fft2(degraded_img)
    H = np.fft.fft2(psf, s=degraded_img.shape)

    # Avoid division by zero
    H_inv = np.where(H != 0, 1 / H, 0)

    # Apply the inverse filter
    F_hat = G * H_inv

    # Compute the inverse Fourier Transform
    restored_img = np.fft.ifft2(F_hat).real

    return restored_img

image = cv2.imread('your_image.jpg', cv2.IMREAD_GRAYSCALE)

# Simulate degradation using a Gaussian blur
psf = gaussian_psf(size=15, sigma=5)
degraded_img = cv2.filter2D(image, -1, psf)

# Apply inverse filtering
restored_img = inverse_filter(degraded_img, psf)
```

Wiener filtering

Wiener filtering is a popular and effective technique used in image processing and signal processing for noise reduction and image restoration.

The Wiener filter aims to produce an estimate of the original image that minimizes the mean square error between the estimated and the true image.

Wiener Filter: The Wiener filter is defined as:

$$F(u,v) = \frac{G(u,v) \cdot H^*(u,v)}{|H(u,v)|^2 + K}$$

Where:

- $H^*(u,v)$: The complex conjugate of $H(u,v)$.
- $|H(u,v)|^2$: The power spectrum of the PSF.
- K: A regularization parameter representing the noise power. This term helps stabilize the filter by preventing division by zero and controlling the influence of noise.

Wavelet Transform

Unlike traditional Fourier Transform methods that analyze signals in terms of sine and cosine functions, wavelet transforms use localized waveforms called wavelets. This allows for a more flexible representation of signals that can capture both frequency and location (time or space).

Key Concepts of Wavelet Transform

1. **Wavelets**:
    - Wavelets are small oscillatory functions that are localized in both time (or space) and frequency. They can be stretched, compressed, or translated to analyze various parts of a signal or image.
    - Common types of wavelets include:
        - **Haar wavelet**: The simplest wavelet, which resembles a step function.
        - **Daubechies wavelet**: A family of wavelets with compact support that allow for smoother approximations.
        - **Symlets**: Modified versions of Daubechies wavelets that are more symmetric.
        - **Coiflets**: Wavelets that have both compact support and vanishing moments.
2. **Continuous Wavelet Transform (CWT)**:
    - The CWT decomposes a signal into wavelets at various scales and positions. It is defined as:

$$C(a,b) = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} f(t) \psi^* \left(\frac{t-b}{a} \right) dt$$

Where:

- $f(t)$ is the original signal.
- $\psi(t)$ is the wavelet function.
- a is the scale factor (dilation or compression).
- b is the translation factor (position).
- ψ^* is the complex conjugate of the wavelet function.

3. **Discrete Wavelet Transform (DWT)**:
    - The DWT provides a multi-resolution analysis of a signal by decomposing it into approximation and detail coefficients. The approximation captures the low-frequency components, while the details capture the high-frequency components.
    - The DWT can be performed using a process called **filter bank analysis**, which involves convolving the signal with low-pass and high-pass filters, followed by downsampling.
    - The DWT can be computed efficiently using an algorithm known as the **Mallat algorithm**.
4. **Inverse Wavelet Transform**:
    - The inverse transform reconstructs the original signal from the wavelet coefficients. The reconstruction is done using the approximation and detail coefficients obtained from the DWT.

Image Enhancement

Image enhancement is the process of improving the visual quality of an image or making it more suitable for analysis.

This can include techniques such as:

1. **Contrast Adjustment**: Enhancing the difference between the darkest and lightest parts of an image.
2. **Brightness Adjustment**: Increasing or decreasing the overall lightness or darkness of the image.
3. **Sharpening**: Making edges more distinct to enhance details.
4. **Noise Reduction**: Removing unwanted random variations in brightness or color.
5. **Color Correction**: Adjusting colors to make them more accurate or vibrant.
6. **Saturation Adjustment**: Changing the intensity of colors in the image.
7. **Resizing**: Changing the dimensions of the image for better fit or quality.
8. **Filters and Effects**: Applying artistic effects to change the mood or style of the image.

Image enhancement techniques:

1. Histogram Equalization
2. Gamma Correction or Intensity Transform
3. Unsharp Masking
4. Low-Pass Filtering
5. High-Pass Filtering
6. Color Space Transformation
7. Image Smoothing
8. Edge Detection
9. Image Denoising
10. Dynamic Range Compression
11. Cropping and Resizing
12. Color Adjustment
13. Filters and Effects
14. Content-Aware Fill

Intensity Transformation

Intensity transformation involves ***modifying the pixel values*** of an image to achieve desired effects, such as contrast enhancement, brightness adjustment, or gamma correction.

A digital image is represented as a two-dimensional matrix I of size N×N (for square images), where each element I(r,c) corresponds to the intensity level of the pixel at row r and column c.

- The number of bits m used to represent each pixel determines the number of intensity levels $L = 2^m$

Common Intensity Transformation Functions:

1. Linear Transformation

For a linear transformation, the output intensity I' is computed from the input intensity I using the equation:

$$I' = aI + b$$

- a: scaling factor (contrast adjustment)
- b: offset (brightness adjustment)

Example:

- Increasing brightness: Set $a = 1$ and $b > 0$.
- Increasing contrast: Set $a > 1$ and $b = 0$.

2. Gamma Correction

- **Formula**:

$$I' = c \cdot I^{\gamma}$$

- **Description**: Nonlinear adjustment of brightness. c is a scaling constant, and γ controls the shape of the curve:
  - $\gamma < 1$: Brightens the image.
  - $\gamma > 1$: Darkens the image.

3. Logarithmic Transformation

- **Formula**:

$$I' = c \cdot \log(1 + I)$$

- **Description**: Enhances dark areas in an image, making low-intensity values more pronounced.

4. Exponential Transformation

- **Formula**:

$$I' = c \cdot e^{kI} - 1$$

- **Description**: Used to enhance bright areas. This function can produce a dramatic increase in intensity for higher values.

Histogram Equalization

Histogram equalization is a technique used to improve the contrast of an image by redistributing the intensity levels of the pixels.

It aims to make the histogram of the pixel intensity values approximately uniform, which enhances the visibility of details in the image, both bright and dark areas.

Steps Involved

1. **Compute the Histogram**:
   - Calculate the histogram $h(i)$, which counts the number of pixels for each intensity level i.
2. **Calculate the Cumulative Distribution Function (CDF)**:
   - The CDF gives the cumulative number of pixels up to intensity level i:

$$CDF(i) = \sum_{j=0}^{i} h(j)$$

3. **Normalize the CDF**:
   - Normalize the CDF to map the intensity levels to the range [0, L-1], where L is the total number of intensity levels:

$$CDF'(i) = \frac{CDF(i) - CDF_{min}}{M \cdot N - CDF_{min}} \cdot (L - 1)$$

   Here, M and N are the image dimensions, and CDF_{min} is the minimum value of the CDF.
4. **Map the Intensity Levels**:
   - Replace each intensity level I in the original image with the corresponding value from the normalized CDF:

$$I' = CDF'(I)$$

```
# Apply histogram equalization
equalized_image = cv2.equalizeHist(image)
```

For example,
https://www.tutorialspoint.com/dip/histogram_equalization.htm

Practical Questions:

1. **Can histogram equalization be applied to color images?**
   - Yes, but it's generally applied to the luminance channel (brightness) of a color space (like HSV or YCbCr) to avoid unnatural color changes.
2. **What are the potential drawbacks of histogram equalization?**
   - It may cause over-enhancement, introduce noise, or result in loss of detail in low-contrast regions.
3. **What are some alternatives to histogram equalization?**
   - Alternatives include adaptive histogram equalization (e.g., CLAHE), contrast stretching, and more advanced methods like gamma correction.

Adaptive Histogram Equalization (CLAHE)

Adaptive Histogram Equalization (AHE) is an advanced version of histogram equalization that enhances the contrast of images by applying the equalization process *to small regions, or "tiles,"* of the image rather than the entire image at once. This approach helps preserve local details and avoids over-enhancing noise in uniform areas.

1. Local Histogram Calculation:

For a given image, I, with dimensions $M \times N$, AHE divides the image into non-overlapping tiles (or windows) of size $W \times W$. For each tile, the histogram Hw is computed:

$$Hw(i) = \text{number of pixels with intensity } i \text{ in the window } w$$

where i ranges from 0 to the maximum intensity level (e.g., 255 for an 8-bit image).

2. Cumulative Distribution Function (CDF):

The cumulative distribution function for the histogram of each tile is calculated as follows:

$$CDFw(i) = \sum_{j=0}^{i} Hw(j)$$

The CDF represents the cumulative count of pixels for intensities up to i in the tile.

3. Normalization of CDF:

To normalize the CDF so that it maps to the range of intensity values, the following formula is used:

$$CDFnorm(i) = \frac{CDFw(i) - CDFw(min)}{Mw - CDFw(min)}$$

where Mw is the total number of pixels in the window w and $CDFw(min)$ is the minimum value of the CDF for that tile.

4. Mapping Pixel Intensities:

The normalized CDF is then scaled to the range of possible intensity values:

$$new\_intensity = CDFnorm(i) \cdot (L - 1)$$

where L is the number of intensity levels (e.g., 256 for 8-bit images).

5. Constructing the Output Image:

For each pixel in the original image, the corresponding pixel intensity in the output image O is computed based on the mapping derived from the local histogram equalization:

$$O(x, y) = new\_intensity(I(x, y))$$

6. Contrast Limited AHE (CLAHE):

In CLAHE, before computing the histogram, the histogram is clipped at a certain limit T:

$$Hw'(i) = \begin{cases} Hw(i) & \text{if } Hw(i) \leq T \\ T & \text{if } Hw(i) > T \end{cases}$$

After clipping, the remaining pixel counts are redistributed uniformly across the histogram to maintain the total number of pixels.

Contrast Stretching

Contrast stretching is a simple and effective image enhancement technique used to improve the visibility of features in an image by expanding the range of intensity values. This method is particularly useful for images that are too dark or too bright, making it easier to interpret details.

$$I_{output} = \frac{I_{input} - I_{min}}{I_{max} - I_{min}} \times (L - 1)$$

Example of Contrast Stretching:

Given an image with intensity values ranging from 50 to 200, you can stretch this range to the full 0-255 range. The transformation would be:

1. Set $I_{min} = 50$ and $I_{max} = 200$.
2. Apply the formula for each pixel I_{input}:

$$I_{output} = \frac{I_{input} - 50}{200 - 50} \times 255$$

Image Segmentation

At its core, image segmentation involves *dividing an image into distinct parts* that correspond to different objects or areas of interest. Each segment is treated as a separate entity, allowing for easier analysis and interpretation of the visual data.

Types of Segmentation

1. **Semantic Segmentation:**
    - Involves classifying each pixel in an image into predefined categories (e.g., road, car, tree).
    - All pixels belonging to the same class are treated as *a single entity*, lacking differentiation between individual objects of the same class.
2. **Instance Segmentation:**
    - Builds on semantic segmentation by not only classifying pixels but also differentiating between *distinct instances of objects* within the same class.
    - For example, in an image containing three cars, instance segmentation would label each car as a separate entity, allowing for individual tracking and analysis.

Traditional Segmentation Techniques

1. Thresholding

Thresholding is a simple yet effective method that converts *a grayscale image into a binary image*. It involves setting a specific threshold value that separates pixels into foreground and background based on their intensity.

Global Thresholding: A *single threshold value* is applied to the entire image.

$$B(x,y) = \begin{cases} 1 & \text{if } I(x,y) > T \\ 0 & \text{if } I(x,y) \leq T \end{cases}$$

In global thresholding, a threshold T is chosen to segment the image. The binary output B(x,y) is defined as:

```
threshold_T = 64       # Choose a threshold value
max_value = 255      # greater than this threshold_T will be set to max_value
return_val, binary_img = cv2.threshold(image, threshold_T, max_value, cv2.THRESH_BINARY)
```

- **return_val:** The computed threshold value (useful when using methods like Otsu's, where the function calculates the optimal threshold).

The type of thresholding to be applied. Common options include:

- ❖ **cv2.THRESH_BINARY:** Sets pixels above the threshold to maxval and those below to 0.
- ❖ **cv2.THRESH_BINARY_INV:** Inverts the binary output (sets pixels below the threshold to maxval and above to 0).
- ❖ **cv2.THRESH_TRUNC:** Sets pixels above the threshold to the threshold value and leaves others unchanged.
- ❖ **cv2.THRESH_TOZERO:** Leaves pixels below the threshold unchanged and sets those above to 0.
- ❖ **cv2.THRESH_TOZERO_INV:** Inverts the behavior of THRESH_TOZERO.

Otsu's Method: Otsu's method finds the *optimal threshold T* by minimizing the intra-class variance. The threshold isn't chosen but is determined automatically (**cv2.THRESH_OTSU**)

```
threshold_value, binary_image = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Adaptive Thresholding:

The threshold value is computed locally based on the pixel neighborhood, making it effective for images with varying lighting conditions.

For a pixel located at coordinates (x,y), the adaptive threshold T(x,y) is calculated using the pixel values in its surrounding neighborhood N(x,y). This neighborhood can be defined as a square window of size k×k, centered on the pixel.

Two common approaches:

- **Mean Adaptive Thresholding:** The threshold is the mean of the local neighborhood.

$$T(x,y) = \frac{1}{N} \sum_{(i,j)\in N(x,y)} I(i,j)$$

- **Gaussian Adaptive Thresholding:** The threshold is a weighted sum of the local neighborhood, considering Gaussian weights.

$$T(x,y) = \frac{\sum_{(i,j)\in N(x,y)} I(i,j) \cdot G(i,j)}{W}$$

```
dst = cv2.adaptiveThreshold(src, maxValue, adaptiveMethod, thresholdType, blockSize, C)
```

adaptiveMethod: The method used to calculate the threshold for a pixel:

- cv2.ADAPTIVE_THRESH_MEAN_C
- cv2.ADAPTIVE_THRESH_GAUSSIAN_C

thresholdType: The type of thresholding to apply:

- cv2.THRESH_BINARY: Sets pixels above the threshold to maxValue and those below to 0.
- cv2.THRESH_BINARY_INV: Inverts the binary output.

```
# Apply Mean Adaptive Thresholding
mean_threshold = cv2.adaptiveThreshold(image, 255,
                                       cv2.ADAPTIVE_THRESH_MEAN_C,
                                       cv2.THRESH_BINARY,
                                       11, 2)

# Apply Gaussian Adaptive Thresholding
gaussian_threshold = cv2.adaptiveThreshold(image, 255,
                                           cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                           cv2.THRESH_BINARY,
                                           11, 2)
```

Applications: Object detection, document image analysis.

---

2. Region-Based Segmentation

It focuses on partitioning an image into distinct regions based on predefined criteria, such as color, intensity, or texture.

- **Regions:** In this context, a region is defined as *a set of connected pixels* that share similar properties.
- **Homogeneity Criteria:**
    - **Intensity Values:** Pixels with similar brightness or color levels.
    - **Texture:** Pixels that share similar texture patterns.
- **Connectivity:** To determine if pixels belong to the same region, a connectivity rule is applied, which can be:
    - **4-connectivity:** Pixels are connected if they share an edge.
    - **8-connectivity:** Pixels are connected if they share an edge or a corner.

It typically involves two main approaches:

- **Region Growing**: Starts with seed points and grows regions by adding neighboring pixels that meet certain criteria (e.g., similar intensity).

```
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Parameters for region growing
seed = (100, 100)  # Example seed point (x, y)
threshold = 15     # Intensity similarity threshold

# Create a mask to keep track of segmented areas
height, width = image.shape
segmented = np.zeros_like(image, dtype=np.uint8)

# Stack to hold pixels to visit
to_visit = [seed]

# Starting intensity of the seed point
seed_intensity = image[seed[1], seed[0]]

while to_visit:
    x, y = to_visit.pop()
    if segmented[y, x] == 0:  # If the pixel is not yet segmented
        segmented[y, x] = 255  # Mark as part of the region
        # Check the 8-connected neighborhood
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                if dx == 0 and dy == 0:
                    continue
                nx, ny = x + dx, y + dy
                if 0 <= nx < width and 0 <= ny < height:
                    if segmented[ny, nx] == 0:  # Not yet segmented
                        if abs(int(image[ny, nx]) - int(seed_intensity)) < threshold:
                            to_visit.append((nx, ny))
```

- **Region Splitting and Merging**: The image is divided into non-overlapping regions, which are then merged based on similarity criteria.

Splitting: The process begins by treating the entire image as a single region. If the region is found to be non-homogeneous (i.e., it contains pixels with varying properties), it is split into smaller, non-overlapping regions.

Merging: After splitting, adjacent regions that meet a certain similarity criterion can be merged to form larger, more coherent regions. This step helps to eliminate noise and small artifacts from the segmentation.

Homogeneity Criteria: The criteria for determining whether regions are homogeneous or similar can include:

i. Intensity values.
ii. Color or texture features.
iii. Statistical properties like variance.

```
def is_homogeneous(region):
        """Check if the region is homogeneous based on variance."""
    return np.var(region) < 500  # Arbitrary threshold for variance

def split_region(image, region):
        """Split a region into four quadrants."""
    h, w = region.shape
    mid_x, mid_y = w // 2, h // 2
    return [region[:mid_y, :mid_x], region[:mid_y, mid_x:],
            region[mid_y:, :mid_x], region[mid_y:, mid_x:]]

def region_splitting(image):
        """Perform region splitting."""
    regions = [image]  # Start with the entire image as a single region
    result = []
    while regions:
        current_region = regions.pop()
        if is_homogeneous(current_region):
            result.append(current_region)
        else:
            # Split the region and add subregions to the list
            subregions = split_region(image, current_region)
            regions.extend(subregions)
    return result

image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Perform region splitting
regions = region_splitting(image)
```

Applications: Medical imaging, satellite image analysis.

---

3. Clustering-Based Segmentation

Clustering methods group pixels based on their features, such as color or intensity. The most common algorithm used is:

- **K-Means Clustering**: Divides the image into K clusters, with each pixel assigned to the nearest cluster centroid based on feature similarity.

Applications: Image compression, scene segmentation.

K-Means Clustering

---

4. Watershed Segmentation

This technique treats the grayscale image as a topographic surface. The method is based on the concept of "flooding" the landscape, where low-intensity areas correspond to valleys and high-intensity areas correspond to ridges.

```
image = cv2.imread('image.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Gaussian blur to smooth the image
blurred = cv2.GaussianBlur(gray, (5, 5), 0)

# Perform edge detection
edges = cv2.Canny(blurred, 30, 150)

# Find contours and create a mask
contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
mask = np.zeros_like(gray)
for cnt in contours:
    cv2.drawContours(mask, [cnt], -1, (255), thickness=cv2.FILLED)

# Perform morphological operations to remove small noise
kernel = np.ones((3, 3), np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

# Distance transform
dist_transform = cv2.distanceTransform(mask, cv2.DIST_L2, 5)
_, markers = cv2.threshold(dist_transform, 0.7 * dist_transform.max(), 255, 0)

# Convert markers to uint8
markers = np.uint8(markers)

# Add one to all markers to distinguish background
markers = markers + 1
markers[mask == 255] = 0  # Mark the foreground as 0

# Apply the watershed algorithm
cv2.watershed(image, markers)

# Mark the boundaries on the original image
image[markers == -1] = [255, 0, 0]  # Mark boundaries in red
```

Applications: Biomedical imaging, where precise delineation of structures is crucial.

---

5. Morphological Segmentation

Description: Morphological operations manipulate the shapes within an image. Common techniques include dilation, erosion, opening, and closing. These operations are often used to refine the segmentation results obtained from other methods.

Applications: Shape analysis, noise reduction in binary images.
Morphological Transformations

Operation	Formula	OpenCV Function
Dilation	$(A \oplus B)(x,y) = \max_{(i,j)\in B} A(x-i, y-j)$	`cv2.dilate(image, kernel, iterations=1)`
Erosion	$(A \ominus B)(x,y) = \min_{(i,j)\in B} A(x+i, y+j)$	`cv2.erode(image, kernel, iterations=1)`
Opening	$A \circ B = (A \ominus B) \oplus B$	`cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)`
Closing	$A \bullet B = (A \oplus B) \ominus B$	`cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)`
Morphological Gradient	$\nabla A = (A \oplus B) - (A \ominus B)$	`cv2.morphologyEx(image, cv2.MORPH_GRADIENT, kernel)`
Top Hat	$A \circ B = A - (A \circ B)$	`cv2.morphologyEx(image, cv2.MORPH_TOPHAT, kernel)`
Black Hat	$A \bullet B = (A \bullet B) - A$	`cv2.morphologyEx(image, cv2.MORPH_BLACKHAT, kernel)`

Common Structuring Elements

Element Type	OpenCV Function
Square	`kernel = np.ones((size, size), np.uint8)`
Ellipse	`kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (size, size))`
Cross	`kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (size, size))`

Feature Extraction and Description

Method	Description	OpenCV Function
Harris Corner Detection	Detects corners using the Harris matrix.	`cv2.cornerHarris()`
Shi-Tomasi Corner Detection	Improved version of Harris for detecting corners.	`cv2.goodFeaturesToTrack()`
SIFT (Scale-Invariant Feature Transform)	Detects and describes local features in images.	`cv2.SIFT_create()`
SURF (Speeded Up Robust Features)	Similar to SIFT but faster; detects and describes features.	`cv2.xfeatures2d.SURF_create()`
ORB (Oriented FAST and Rotated BRIEF)	Efficient and fast alternative to SIFT and SURF.	`cv2.ORB_create()`
FAST (Features from Accelerated Segment Test)	Detects corners quickly and efficiently.	`cv2.FastFeatureDetector_create()`
BRIEF (Binary Robust Invariant Elementary Features)	Describes features using binary strings.	Custom implementation needed.
AKAZE (Accelerated KAZE)	Detects features and computes descriptors quickly.	`cv2.AKAZE_create()`
HOG (Histogram of Oriented Gradients)	Describes image regions based on gradient orientations.	`cv2.HOGDescriptor()`
LBP (Local Binary Patterns)	Textural feature descriptor used for image classification.	`cv2.calcHist()` + custom LBP code
Color Histograms	Represents color distribution in an image.	`cv2.calcHist()`

OpenCV Tutorial.ipynb

Interpolation methods

What is Interpolation?

Interpolation is a technique used to estimate values between known data points. Imagine you have some points on a graph, and you want to find the value at a point that lies between these known points. Interpolation helps you do that!

Why Use Interpolation?

When you collect data, it often comes in discrete points. For example, you might have temperature readings taken every hour. If you want to know the temperature at half an hour, interpolation provides a way to estimate it.

Example: If you have three points, you can find a quadratic (parabolic) equation that passes through them, giving a smoother curve.

For Example

Let's say we have the following three data points: (1,2), (2,3), (3,5). We want to find a polynomial P(x) that passes through these points.

Set Up the General Form

For three points, we will use a polynomial of degree 2

$$P(x)=ax^2+bx+c$$

Set Up the System of Equations

We can plug in the known points into the polynomial to create a system of equations:

For (1,2): $a(1)^2+b(1)+c = 2 \Rightarrow a+b+c=2$ …..(1)

For (2,3):$a(2)^2+b(2)+c = 3 \Rightarrow 4a+2b+c=3$ …..(2)

For (3,5):$a(3)^2+b(3)+c = 5 \Rightarrow 9a+3b+c=5$ …..(3)

By solving, we can write the polynomial:

Now we have: $a=1/2,\ b=-1/2,\ c=2$

$$P(x)=1/2x^2-1/2x+2$$

OpenCV supports *<u>several interpolation methods</u>* that *can be applied when* ***<u>resizing images</u>*** *or performing* ***<u>geometric transformations</u>*** (e.g., rotation, affine, perspective). These methods are available through the **cv2.resize()** function or other transformation functions.

```
resized_image = cv2.resize(image, (width, height), interpolation=cv2.INTER_NEAREST)
```

Here are the most common interpolation techniques available in OpenCV:

1. Nearest Neighbor Interpolation (cv2.INTER_NEAREST)
2. Bilinear Interpolation (cv2.INTER_LINEAR)
3. Cubic Interpolation (cv2.INTER_CUBIC)
4. Lanczos Interpolation (cv2.INTER_LANCZOS4)
5. Area-based Interpolation (cv2.INTER_AREA)

1. Linear Interpolation

Formula:
For two points (x_0, y_0) and (x_1, y_1), the linear interpolation formula to estimate y at a point x is:

$$y = y_0 + \frac{(y_1 - y_0)}{(x_1 - x_0)} \cdot (x - x_0)$$

2. Polynomial Interpolation

For a set of $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, the interpolating polynomial $P(x)$ can be expressed using Lagrange interpolation:

Lagrange Polynomial:

$$P(x) = \sum_{i=0}^{n} y_i \cdot L_i(x)$$

where $L_i(x)$ is given by:

$$L_i(x) = \prod_{\substack{0 \le j \le n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

3. Newton's Divided Difference Interpolation

The interpolating polynomial can also be constructed using divided differences. The formula is:

$$P(x) = f[x_0] + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] + \ldots$$

where $f[x_i, x_j, \ldots]$ denotes the divided differences.

4. Spline Interpolation

Cubic Spline: This method involves fitting a cubic polynomial between each pair of data points. The spline $S_i(x)$ between points (x_i, y_i) and (x_{i+1}, y_{i+1}) is defined as:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

The coefficients a_i, b_i, c_i, d_i are determined by ensuring continuity and smoothness at each data point.

5. Barycentric Interpolation

The barycentric form of the Lagrange polynomial is given by:

$$P(x) = \frac{\sum_{i=0}^{n} \frac{w_i}{x - x_i} y_i}{\sum_{i=0}^{n} \frac{w_i}{x - x_i}}$$

where w_i are weights typically set as $w_i = \frac{1}{x_i - x_j}$ for $j \neq i$.

6. Nearest Neighbor Interpolation

This method does not involve a mathematical formula like the others. It simply assigns the value of the nearest known point:

$$y = \text{Value of nearest } (x_i, y_i)$$

7. Bilinear Interpolation

For a point (x, y) within a rectangle defined by points (x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2), the formula is:

$$f(x, y) = \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} f(x_1, y_1) + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} f(x_1, y_2) + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} f(x_2, y_1) + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} f(x_2, y_2)$$

Nearest Neighbour interpolation

It is one of the simplest interpolation method where the unknown values are assigned the value of its nearest neighbour .Let's get back to our example and see how this works.How would the resized image look like when we use nearest interpolation method.

Resized image(before interpolation)

0	-1	-1	255	-1	-1	0
-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1
255	-1	-1	0	-1	-1	255
-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1
0	-1	-1	255	-1	-1	0

Resized image (after interpolation)

0	0	255	255	255	0	0
0	0	255	255	255	0	0
255	255	0	0	0	255	255
255	255	0	0	0	255	255
255	255	0	0	0	255	255
0	0	255	255	255	0	0
0	0	255	255	255	0	0

Bilinear Interpolation

In bilinear interpolation, we use the four nearest neighbors to estimate the intensity at a given location rather than just the nearest one.

Before getting into bilinear interpolation let us have a look into linear interpolation and then extend it into 2D.

Consider a 1D array ,arr=[100 _ _ 200].Our objective here is to find out the unknown values at 2nd and 3rd position of the array.

In linear interpolation we consider the two nearest neighbours and find their weighted average value,therefore:

arr(2)=100*(⅔)+200*(⅓) Similarly ,

arr(3)=100*(⅓)+200*(⅔)

The contribution the pixel values depends on the distance ,the more closer it is the contribution will be more.

This can now be extended into 2D using four nearest neighbours.

Let us now have a look at how would the resized image look like using bilinear interpolation:

Original image:

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix}$$

Resized image:

$$\begin{bmatrix} 0 & 85 & 170 & 255 & 170 & 85 & 0 \\ 85 & 113 & 141 & 170 & 141 & 113 & 85 \\ 170 & 141 & 113 & 85 & 113 & 141 & 170 \\ 255 & 170 & 85 & 0 & 85 & 170 & 255 \\ 170 & 141 & 113 & 85 & 113 & 141 & 170 \\ 850 & 113 & 141 & 170 & 141 & 113 & 85 \\ 0 & 85 & 170 & 255 & 170 & 85 & 0 \end{bmatrix}$$