
OpenCV for Image Processing

Image:

An image is a visual representation. An image can be two-dimensional, such as a drawing, painting, or photograph, or three-dimensional, such as a carving or sculpture. An image is represented as a matrix of pixel values. For a grayscale image, each pixel value ranges from 0 to 255, representing the intensity of that pixel. For example, a 20x20 image would be represented by a matrix of 400-pixel values

Unit of an image: DPI – Dots Per Inches. DPI is essential for determining the quality of printed assets, whether it's a poster, brochure, or glossy magazine. Higher DPI values result in crisper and more high-quality images. PPI (Pixels per Inch) describes the number of pixels within an inch on a digital screen.

Types of Images:

1. **Binary Images:**

- These are the simplest type of images.
- Binary images take only two values: Black (0) and White (1).
- Each pixel is represented by a single binary digit.
- They are commonly used for general shapes or outlines, such as in Optical Character Recognition (OCR) applications.

2. **Grayscale Images:**

- Grayscale images are monochrome, meaning they have only one color (various shades of Gray).
- Each pixel in a grayscale image represents a different gray level.
- A typical grayscale image contains 8 bits per pixel, allowing for 256 different gray levels.
- In grayscale, each pixel is represented by a **single channel** of intensity. This means that the color information is reduced to a single value, typically ranging from 0 (black) to 255 (white). The resulting image appears in shades of gray, with darker areas having lower intensity values and lighter areas having higher intensity values. Essentially, grayscale removes the color aspect and focuses solely on the brightness or luminance of the image. It's like viewing the world in black and white, where the absence of color reveals the underlying structure and contrast.

3. **Color Images:**

- Color images contain information about color and are represented in three bands: red, green, and blue (RGB).
- Each color band contributes to the overall color of the image.

- Standard color images have 24 bits per pixel (8 bits for each RGB channel).
- The 24-bit format is known as “true color.”

Color Formats:

- **8-bit color:** Used for storing image information in a computer’s memory or files. It has 256 different colors (0-255), with 0 for black, 255 for white, and 127 for gray.
- **16-bit color (high color):** Offers 65,536 different color shades. It’s divided into RGB format (5 bits for R, 6 bits for G, and 5 bits for B).
- **24-bit color (true color):** Equally distributed between R, G, and B (8 bits each).

Black & white images will not have colour channels. i.e `img.shape` will return height & width (250 x 260) but not (250 x 260 x 3). `img.shape[0] == height == 250`

Pixel:

Pixel = Picture + Element

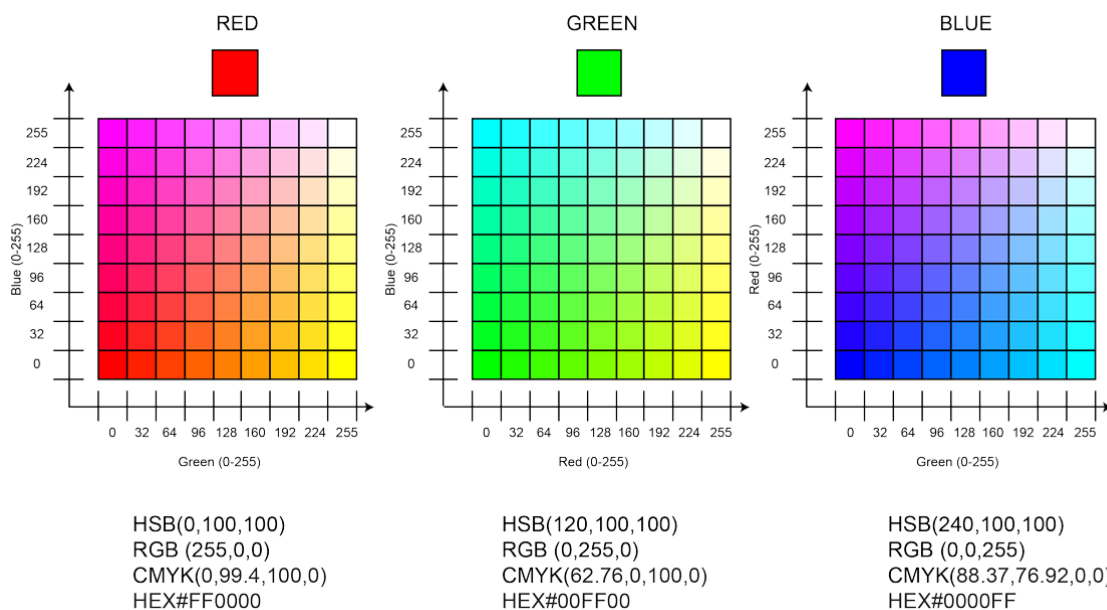
Pixel is a small element in a picture. Each pixel represents a colour which is combination of RGB. With this combination we can make millions of colours. Screen consist of RGB transistors which will turn on/off for each pixel based on Video/Image colour.

For a Full HD Video:

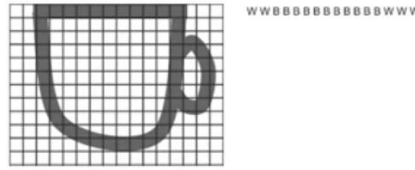
Each frame will have $1920 \times 1080 = 2073600$ Pixels

Each pixel will have 3 RGB transistors, so $2073600 \times 3 = 6220800$ Transistors working in background to see a full HD picture.

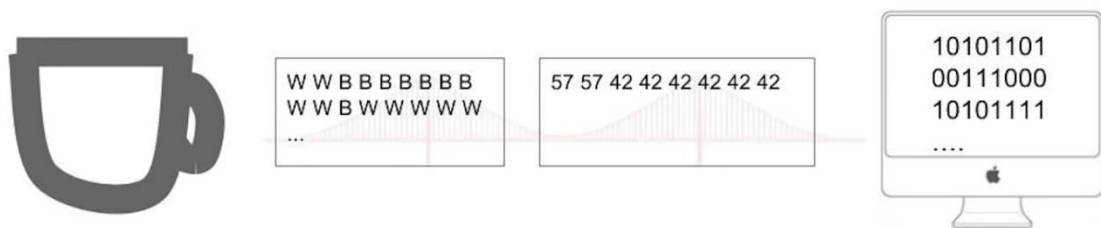
Our computers have 4 billion of transistors inside the processor (2 x 2 inches Size)



Each R, G, B colours are represented by an 8-bit binary code. Each pixel stores three 8-bit binary code that get multiplied depending on resolution of display.



Unicode for each pixel-> Mapping to numeric value -> conversion to 8-bit code



Aspect Ratio vs Pixel Interdependence

1. Aspect Ratio:

- The aspect ratio refers to the proportional relationship between the **height** and **width** of a rectangular shape, such as an image, video frame, or display screen.
- It is expressed as a **ratio** (e.g., 16:9, 4:3, 21:9), where the first number represents the width, and the second number represents the height.
- For example:
 - A **16:9** aspect ratio means that the width is 16 units, and the height is 9 units.
 - A **4:3** aspect ratio implies a width-to-height ratio of 4:3.

2. Pixel Resolution:

- Pixel resolution defines the **number of pixels** (individual picture elements) in an image or display.
- It is typically represented as two numbers: **horizontal pixels** (width) by **vertical pixels** (height).
- For instance:
 - A **1920x1080** pixel resolution display has a **16:9** aspect ratio.
 - The resolution of a display determines its corresponding aspect ratio.

Aspect Ratio is the most simplified fraction of Pixel

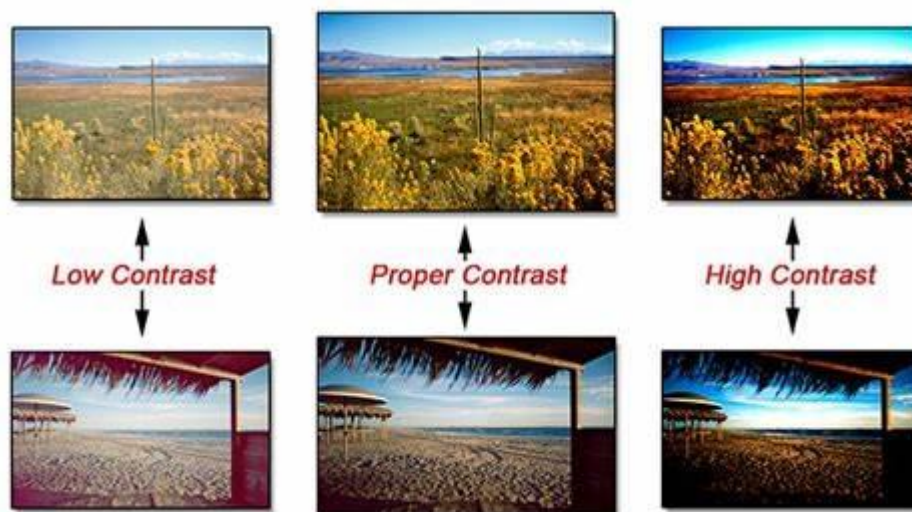
3. Interdependence:

- Aspect ratio and resolution are **co-dependent**. Changing one affects the other:
 - Altering the aspect ratio will lead to a change in resolution, and vice versa.
 - For example, adjusting the aspect ratio from 16:9 to 4:3 will modify the pixel resolution accordingly.

Other Important Image Terminologies:

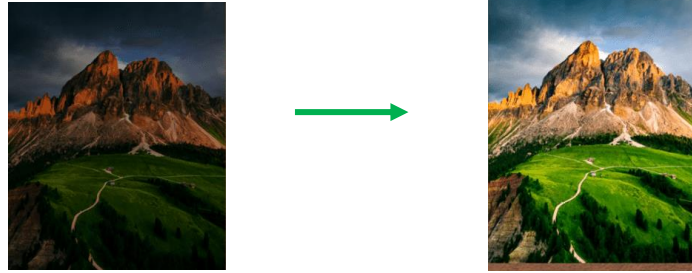
1. Contrast:

- Contrast refers to the difference in brightness between the light and dark areas of an image. High contrast means there is a significant difference between the brightest and darkest parts of the image, while low contrast indicates a smaller difference.
- Adjusting contrast can enhance the visual quality of an image by making it appear sharper and more detailed. Increasing contrast can make objects stand out more, while decreasing contrast can soften the image and reduce harshness.



2. Brightness:

- Brightness refers to the overall lightness or darkness of an image. Increasing brightness makes the image appear lighter, while decreasing brightness makes it darker.
- Adjusting brightness is useful for correcting underexposed or overexposed images. It can also be used to create a specific mood or atmosphere in the image, such as making it brighter for a cheerful scene or darker for a dramatic effect.



3. Sharpening:

- Sharpening enhances the clarity and detail of an image by increasing the contrast along edges and boundaries.
- Sharpening is commonly used to improve the overall sharpness of an image, especially in photographs where fine details need to be emphasized. It can help make images appear crisper and more defined, but excessive sharpening can introduce artifacts and noise.

4. Intensity:

- Intensity in an image refers to the brightness or darkness of a pixel, typically represented by the intensity of light reflected or emitted from a surface. In digital images, intensity values are often represented as grayscale values ranging from 0 (black) to 255 (white) in an 8-bit image, where 0 indicates the absence of light and 255 represents maximum brightness.

5. Blurring (Smoothing):

- Blurring, also known as smoothing, reduces the sharpness of an image by averaging neighbouring pixel values. This process reduces high-frequency components, resulting in a softer appearance.
- Blurring is used for various purposes such as noise reduction, removing imperfections, and creating artistic effects like soft focus or motion blur. It can also be employed as a pre-processing step before other image processing operations.

6. Thresholding:

- Thresholding is a method used to separate objects or features from the background in an image by converting grayscale or colour values to binary values (black and white) based on a specified threshold.
- Thresholding is commonly used for image segmentation, where it helps identify objects of interest by setting a threshold value that distinguishes between foreground and background pixels. It's useful for tasks like object detection, character recognition, and image analysis.

7. Morphological Operations:

- Morphological operations are a set of operations used to analyze and manipulate the structure of objects in an image based on their shapes and spatial relationships.
- Morphological operations, such as erosion, dilation, opening, and closing, are used for tasks like noise removal, edge detection, and image enhancement. They are particularly effective in processing binary images and are often employed in conjunction with other image processing techniques.

8. Edge Detection:

- Edge detection is a process used to identify the boundaries or edges of objects within an image. These edges represent significant changes in intensity or color between neighbouring pixels.
- Edge detection is crucial for tasks like object detection, image segmentation, and feature extraction. It helps identify regions of interest and provides valuable information about the structure and shape of objects in the image. Common edge detection algorithms include Sobel, Prewitt, and Canny.

Computer Vision:

- **Computer vision**, also known as **machine vision**, is a fascinating field of science that empowers computers and devices to recognize objects and patterns, much like human beings.
- Computer vision has a massive impact across industries, including retail, security, healthcare, automotive, and agriculture.

OpenCV Installation:

```
pip install opencv-python
```

```
pip install opencv-contrib-python
```

Importing Library:

```
import cv2 as cv
```

NOTE: Don't use Jupyter Notebook/ Colab for implementation. It may crash sessions.

Suggested to use VS Code or other Python IDLEs.

Use Cases of OpenCV Functionalities

Reading & Displaying Image	
Function	Purpose
<code>cv.imread()</code>	To read an image
<code>cv.imshow()</code>	To display an image. It displays the image up to which the image fits the screen.
<code>cv.imwrite()</code>	To save an output image. <code>cv.imwrite("Flower.jpg",img)</code>
<code>cv.waitKey()</code>	Determines how long the image should last in display window. It waits for specific delay for a key to be pressed. Accepts integer as an argument eg. 5000 (milli seconds). i.e 5 Seconds
<code>cv.destroyAllWindows()</code>	To close all the windows that have been created using the cv.imshow() function.

Reading Video	
Function	Purpose
<code>Video = cv.VideoCapture()</code>	To read a video
<code>Video.isOpened()</code>	To check whether image/ video is opened/not. Returns Boolean value
<code>istru, frame = cap.read()</code>	Reading video frame by frame. read() in OpenCV returns 2 things, boolean and data. <ul style="list-style-type: none"> • A boolean value indicating whether the frame was read successfully • The actual image data of the frame.
<code>frame.shape()</code>	To get the shape of a frame. <ul style="list-style-type: none"> • Shape[0] – Height • Shape[1] – Width

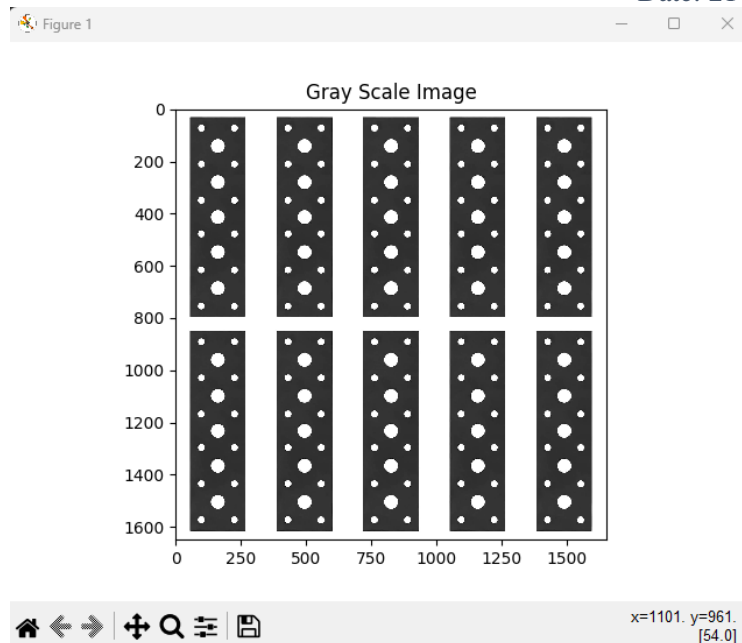
To plot the image with scales/axis:

```
import matplotlib.pyplot as plt

plt.imshow(gray_image, interpolation='none', cmap='gray')

plt.title("Gray Scale Image")

plt.show()
```



The parameter **interpolation='none'** specifies that no interpolation should be applied when displaying the image. Interpolation is a method used to estimate pixel values in between existing pixels when resizing or transforming an image. Sometimes it smooth out the image or introduce artificial details. Setting it to **'none'** means that no interpolation will be used, preserving the original pixel values. Use **cmap='gray'** to preserve the color of gray image.

Example for Rescaling Video:

```
def rescaleFrame(frame, scale=0.75):
    width = int(frame.shape[1] * scale)
    height = int(frame.shape[0] * scale)
    dimensions = (width,height)
    return cv.resize(frame, dimensions,
interpolation=cv.INTER_AREA)

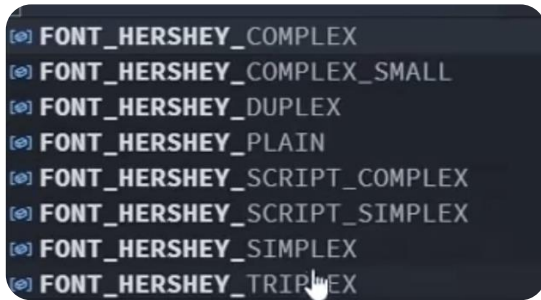
rescaled_frame = rescaleFrame(frame)

# Press Q on keyboard to exit
if cv.waitKey(25) & 0xFF == ord('q'):
    break

# When everything done, release the video capture object
cap.release()

# Closes all the frames
cv.destroyAllWindows()
```

INTER_AREA: This method is generally preferred for shrinking images

Drawing Shapes	
Function	Purpose
<code>blank = np.zeros((500,500,3), dtype='uint8')</code>	For creating blank images with 500 height and 500 width. 3 indicates no of color channels. i.e BGR (Blue, Green, Red)
<code>blank[200:300, 300:400] = 0,0,255</code>	Painting the image a certain colour
<code>cv.rectangle(blank, (0,0), (blank.shape[1]//2, blank.shape[0]//2), (0,255,0), thickness=-1)</code>	For drawing shapes in an image Blank – ImageVariable (0,0) – (x1,y1) point from the top (blank.shape[1]//2, blank.shape[0]//2) – (x2,y2) point (0,255,0) – BGR Color code thickness=-1 or cv.FILLED – Fills Green color for the entire rectangle
<code>cv.circle(blank, (blank.shape[1]//2, blank.shape[0]//2), 40, (0,0,255), thickness=-1)</code>	(blank.shape[1]//2, blank.shape[0]//2) – mid point of the circle 40 – Radius of circle
<code>cv.line(blank, (100,250), (300,400), (255,255,255), thickness=3)</code>	For drawing line in an image
<code>cv.putText(blank, 'Hello, This is Jk!!!', (0,225), cv.FONT_HERSHEY_TRIPLEX, 1.0, (0,255,0), 2)</code>	(0,225) – Origin of the text from where it needs to be displayed cv.FONT_HERSHEY_TRIPLEX – font face <i>Some other font faces (styles):</i>  1.0 – Scale of font 2 – Thickness
<code>cv.imshow('Text', blank)</code>	Remember to use this function to display the image in a display window.

Kernel:

In **image processing**, a **kernel**, also known as a **convolution matrix** or **mask**, is a small matrix used for various operations like blurring, sharpening, embossing, and edge detection.

What is a Kernel?

- A kernel is like a filter that modifies the value of a pixel based on the values of its surrounding pixels. These surrounding pixels are called the **neighborhood pixels** of the central pixel.
- When we apply a kernel to an image, we perform a **convolution** operation between the kernel and the image.

2. How Does Convolution Work?

- Imagine sliding the kernel over the entire image, placing it on each pixel.
- For each position, we multiply the kernel's values with the corresponding pixel values in the image.
- The result of this multiplication is summed up, and the central pixel in the output image is set to this sum.

3. Common Kernels and Their Effects:

- Here are some examples of kernels and their effects:
 - **Identity Kernel:** Leaves the image unchanged.
 - Kernel:
 - $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
 - **Edge Detection Kernel (Sobel or Prewitt):**
 - Enhances edges in the image.
 - Kernel:
 - $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

```
import cv2

sobel_kernel_x = cv2.getDerivKernels(1, 0, 3, normalize=True)
sobel_kernel_y = cv2.getDerivKernels(0, 1, 3, normalize=True)
```

- **Box Blur Kernel/ Average/ Blur (Normalized):**

- Blurs the image.

- Kernel:

- [1 1 1]

- [1 1 1]

- [1 1 1]

- **Gaussian Blur Kernel (3x3 approximation):**

- Smooths the image.

- Kernel:

- [1 2 1]

- [2 4 2]

- [1 2 1]

- **Unsharp Masking Kernel (5x5):**

- Enhances edges while reducing noise.

- Kernel:

- [1 4 6 4 1]

- [4 16 24 16 4]

- [6 24 -476 24 6]

- [4 16 24 16 4]

- [1 4 6 4 1]

4. Origin of the Kernel:

- The **origin** of the kernel is the position above (conceptually) the current output pixel.
- It corresponds to one of the kernel elements (usually the center element for symmetric kernels).

0.111	0.111	0.111
0.111	0.111	0.111
0.111	0.111	0.111

Kernel

x

10	20	13
19	25	16
22	26	21

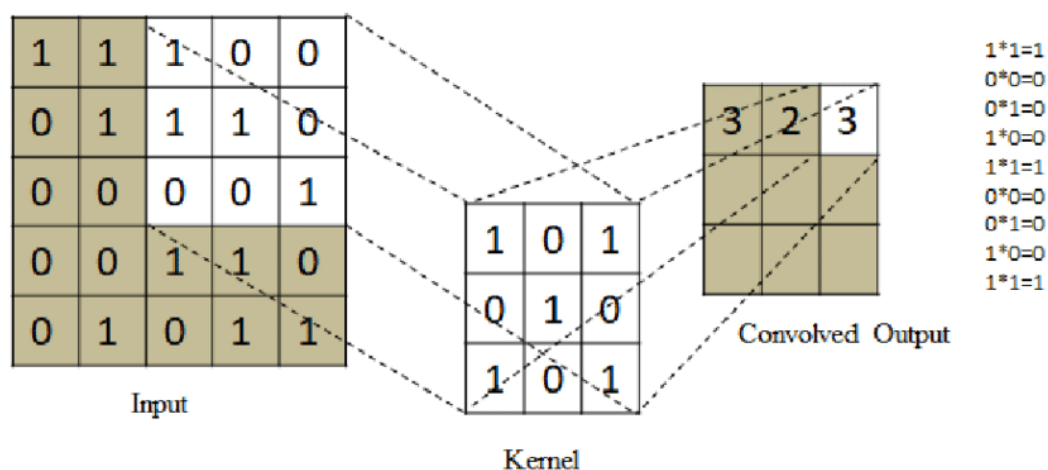
=

0.11 * 10 = 1	0.11 * 20 = 2	0.11 * 13 = 1
0.11 * 19 = 2	0.11 * 25 = 3	0.11 * 16 = 2
0.11 * 22 = 2	0.11 * 26 = 3	0.11 * 21 = 2

= 1 + 2 + 1 + 2 + 3 + 2 + 2 + 3 + 2 = **18**



In short, each pixel is the average of its neighbors this results in a blurred image.



Example for Sobel Edge Detection Kernel:

- OpenCV provides built-in functions to create edge detection kernels like Sobel and Prewitt. We can use `cv2.getDerivKernels()` to generate the kernels:

```
sobel_kernel_x =
cv2.getDerivKernels(
1,
0,
3,
normalize=True
)
```

1. First parameter (dx): Specifies the order of the derivative in the x-direction. In simpler terms, it indicates how many times we want to differentiate the image intensity values with respect to the x-coordinate.

- When **dx=1**, it means we are calculating the first-order derivative along the x-axis. This calculates how intensity values change from one pixel to its neighboring pixel along the horizontal direction. It's commonly used for edge detection because edges in images correspond to rapid changes in intensity, and

<pre>sobel_kernel_y = cv2.getDerivKernels(0, 1, 3, normalize=True)</pre>	<p>the first derivative can help identify these changes.</p> <ul style="list-style-type: none"> • If dx=2, it would calculate the second-order derivative along the x-axis, which measures the rate of change of the first derivative (the gradient). This is less commonly used in edge detection but can be used for more specific purposes like corner detection or fine edge detection. • Similarly, dx=0 would imply no derivative in the x-direction, which means we're not considering any changes along the x-axis, and so forth. <p>2. Second parameter (dy): This specifies the order of the derivative in the y-direction. In this case, dy=0 indicates that we do not want to compute any derivative along the y-axis.</p> <p>3. Third parameter (ksize): This specifies the size of the kernel. In this case, ksize=3 indicates that we want to create a kernel with a size of 3x3.</p> <p>4. Fourth parameter (normalize): This is a boolean parameter that indicates whether the kernel should be normalized. When normalize=True, the kernel will be divided by the number of elements in the kernel to ensure that the sum of all elements in the kernel equals 1. This normalization is commonly used to preserve the magnitude of the image gradient.</p>
--	--

Blur (Smoothing)	
<ul style="list-style-type: none"> • Blurring helps reduce noise and small-scale details in the image. • This can be useful for preparing the image for edge detection or removing unwanted artifacts 	
Function	Purpose
<pre>blurred_image = cv2.GaussianBlur(image, (kernel_size, kernel_size), sigmaX)</pre> <ul style="list-style-type: none"> • Low sigmaX value: If sigmaX is small, the Gaussian kernel is narrow, leading to less blurring. This preserves more high-frequency details in the image. • High sigmaX value: If sigmaX is large, the Gaussian 	<p>1. Gaussian Blur:</p> <ul style="list-style-type: none"> • Gaussian blur is a widely used blurring technique that applies a Gaussian kernel to the image. • It is effective in reducing noise while preserving edges in the image. • The amount of blurring is controlled by the standard deviation (σ) of the Gaussian kernel.

<p>kernel becomes wider, resulting in more blurring. This smooths out fine details and reduces noise in the image.</p>	<p>2. Average/ Box Blur:</p> <ul style="list-style-type: none"> • It applies a simple averaging filter to the image. • It replaces each pixel value with the average value of its neighboring pixels. • Average blur provides uniform blurring across the image. <p>3. Median Blur:</p> <ul style="list-style-type: none"> • Median blur replaces each pixel value with the median value of its neighboring pixels. • It is effective in removing salt-and-pepper noise while preserving edges in the image. • Median blur is particularly useful for images corrupted by impulse noise. <p>4. Bilateral Filter:</p> <ul style="list-style-type: none"> • Bilateral filter preserves edges while reducing noise by applying a weighted average based on both spatial proximity and pixel intensity similarity. • It is effective in smoothing images while preserving edges and fine details. • Bilateral filter is <i>slower</i> compared to other blurring techniques but provides superior results in many cases.
<pre>blurred_image = cv2.blur(image, (kernel_size, kernel_size))</pre>	
<pre>blurred_image = cv2.medianBlur(image, ksize)</pre>	
<pre>blurred_image = cv2.bilateralFilter(image, d, sigmaColor, sigmaSpace)</pre>	

Morphological Operations

Morphological operations play a crucial role in **image processing**, allowing us to manipulate images based on their shape and structure.

1. Erosion:

- Erosion removes pixels from the boundaries of objects in an image.
- It is similar to “shrinking” the object by eroding its edges.
- The size and shape of the **structuring element** used influence the extent of erosion.
- Erosion is often employed to separate connected objects or eliminate small details.

2. Dilation:

- Dilation adds pixels to the boundaries of objects.
- It “expands” the object by adding pixels around its edges.
- Like erosion, dilation also depends on the structuring element.
- Dilation can be useful for filling gaps, joining broken lines, or thickening objects.

3. Opening:

- Opening is a combination of erosion followed by dilation.
- It tends to remove bright foreground pixels from the edges of regions of foreground pixels.
- The primary purpose of opening is to eliminate internal noise in an image.
- Syntax in Python (using OpenCV): `cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)`
- Example: Consider an input frame with a blue book. The opening operation simplifies the internal noise in the region of interest, preserving the book’s shape.

Working of erosion

- ◆ A kernel(a matrix of odd size(3,5,7)) is convolved with the image.
- ◆ A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero).
- ◆ Thus all the pixels near boundary will be discarded depending upon the size of kernel.
- ◆ So the thickness or size of the foreground object decreases or simply white region decreases in the image.

Working of dilation

- ◆ A kernel(a matrix of odd size(3,5,7)) is convolved with the image
- ◆ A pixel element in the original image is '1' if atleast one pixel under the kernel is '1'.
- ◆ It increases the white region in the image or size of foreground object increases

Coding Tamilan

4. Closing:

- Closing is a dual operation used alongside **opening** in digital image processing.
- It is particularly effective for **restoring eroded images**.

Process:

- First, **dilation** is performed on the image.
- Then, **erosion** follows.
- This sequence helps to **smooth the contour** of the distorted image and **fuse back narrow breaks** and **long, thin gulfs**.

Purpose:


Closing is useful for:

- **Smoothing contours** of objects.
- **Filling small holes** in the foreground.
- **Preserving the shape and size** of larger objects in the image.

Working of Opening

◆ Opening is just another name of **erosion** followed by **dilation**. It is useful in removing noise, as we explained above. Here we use the function,


```
opening = cv.morphologyEx(img, cv.MORPH_OPEN, kernel)
```



Working of Closing

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

```
closing = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel)
```



Basic & Most Common Operations in OpenCV	
Function	Purpose
<pre>gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)</pre>	# Converting to grayscale: Many computer vision algorithms work better with grayscale images because they focus on intensity variations rather than color information. "Intensity variations in an image refer to changes in the brightness or darkness of individual pixels."

<pre>blur = cv.GaussianBlur(img, (7,7), cv.BORDER_DEFAULT)</pre>	<p># Blur: Blurring helps reduce noise and small-scale details in the image. This can be useful for preparing the image for edge detection or removing unwanted artifacts</p> <p>(7,7) – Ksize (Kernel Size) Decides the level of blurriness. Larger the no more blur the image will be.</p>
<p style="text-align: center;">Syntax</p> <pre>Cv.Canny (image, T_lower, T_upper, aperture_size, L2Gradient)</pre> <pre>canny = cv.Canny(blur, 125, 175)</pre>	<p># Edge Cascade: Edge detection helps identify object boundaries and sharp transitions in intensity within the image. This is crucial for many computer vision tasks like object recognition, shape analysis, and motion tracking.</p> <p>We can either pass original / blurred image for edge detection.</p> <ol style="list-style-type: none"> 1. Blurred images are often used for getting reduced edges. 2. 125, 175 are 2 threshold values passed. <ul style="list-style-type: none"> • Pixels with gradient values above 175 are strong edges. • Pixels with gradient values between 125 and 175 are weak edges. • Pixels with gradient values below 125 are not considered edges at all. (non edges) 3. aperture_size: The aperture size of the Sobel filter (optional). Increasing this value can help detect more detailed features. 4. L2Gradient: A boolean parameter that specifies whether to calculate the usual gradient equation or use the L2Gradient algorithm (optional).
<pre>dilated = cv.dilate(canny, (7,7), iterations=3)</pre>	<p># Dilating the image: Strengthening weak edges or closing small gaps between edge segments. It can also be used to compensate for slight inaccuracies in edge detection.</p> <p>The resultant image of canny is being passed as src image to this.</p> <p>(7,7) – Kernel Size</p>
<pre>eroded = cv.erode(dilated, (7,7), iterations=3)</pre>	<p># Eroding: Erosion is often used in conjunction with dilation to remove unwanted noise or thin out excessively thick edges. It can also help in segmenting objects by separating them from their background.</p>

	<p>Takes dilated image as an argument.</p> <p>By this, we can convert the dilated image back to Canny edge</p>
<pre>resized = cv.resize(img, (500,500), interpolation=cv.INTER_CUBIC)</pre>	<p># Resize: To resize an image.</p> <p>(500,500) – Target size of the image to be resized.</p> <p>Interpolation – for maintaining the image quality.</p> <p>interpolation=cv.INTER_AREA – can be used when shrinking images to dimensions that are smaller than that of the original dimensions</p> <p>interpolation=cv.INTER_LINEAR [Zooming] (or) interpolation=cv.INTER_CUBIC (Produces more clarity results so preferable) – can be used for enlarging images</p>
<pre>cropped = img[50:200, 200:400]</pre>	<p># Cropping Images</p> <p>Scaling the images</p>

Image Transformations	
Function	Purpose
<pre>def translate(img, x, y): transMat = np.float32([[1,0,x],[0,1,y]]) dimensions = (img.shape[1], img.shape[0]) return cv.warpAffine(img, transMat, dimensions) translated = translate(img, - 100, 100)</pre>	<p># Translation</p> <p>Shifting image by position</p> <p># -x → Left # -y → Up # x → Right # y → Down</p>
<p><u>cv.warpAffine:</u></p> <p>1. Affine Transformation:</p> <ul style="list-style-type: none"> An affine transformation is a geometric operation that can be expressed as a combination of linear transformations (matrix multiplication) and translation (vector addition). It is commonly used for tasks like rotation, translation, and scaling. 	

<ul style="list-style-type: none"> ○ The transformation is represented by a 2x3 matrix. i.e. $\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \end{bmatrix}$ ○ The first two columns represent the linear transformation, and the third column represents the translation. <p>✚ The first row of the matrix corresponds to the X-axis:</p> <ul style="list-style-type: none"> ▪ First element: Scaling along the X-axis. ▪ Second element: Shearing along the X-axis. ▪ Third element: X-coordinate translation. <p>✚ The second row corresponds to the Y-axis:</p> <ul style="list-style-type: none"> ▪ First element: Shearing along the Y-axis. ▪ Second element: Scaling along the Y-axis. ▪ Third element: Y-coordinate translation. <p>2. Purpose of cv.warpAffine:</p> <ul style="list-style-type: none"> ○ The cv.warpAffine function applies an affine transformation to an input image. ○ It takes the following parameters: <ul style="list-style-type: none"> ▪ src: The input image. ▪ M: The 2x3 transformation matrix. ▪ dsize: The size of the output image. ▪ dst: The output image (optional). ▪ flags: Additional transformation options (optional). ▪ borderMode: Border handling mode (optional). ▪ borderValue: Border value (optional). 	
<pre>def rotate(img, angle, rotPoint=None): (height,width) = img.shape[:2] if rotPoint is None: rotPoint = (width//2,height//2) rotMat = cv.getRotationMatrix2D(rotPoint, angle, 1.0) dimensions = (width,height) return cv.warpAffine(img, rotMat, dimensions) rotated = rotate(img, -45)</pre>	<p># Rotating Image</p>
<pre>flip = cv.flip(img, -1) cv.imshow('Flip', flip)</pre>	<p># Flipping 1 = Horizontal across y axis 0 = Vertical flipping over x axis -1 = Both Horizontal & Vertical</p>

Thresholding

1. Simple/Global

- `cv.THRESH_BINARY`
- `cv.THRESH_BINARY_INV`
- `cv.THRESH_TRUNC`
- `cv.THRESH_TOZERO`
- `cv.THRESH_TOZERO_INV`

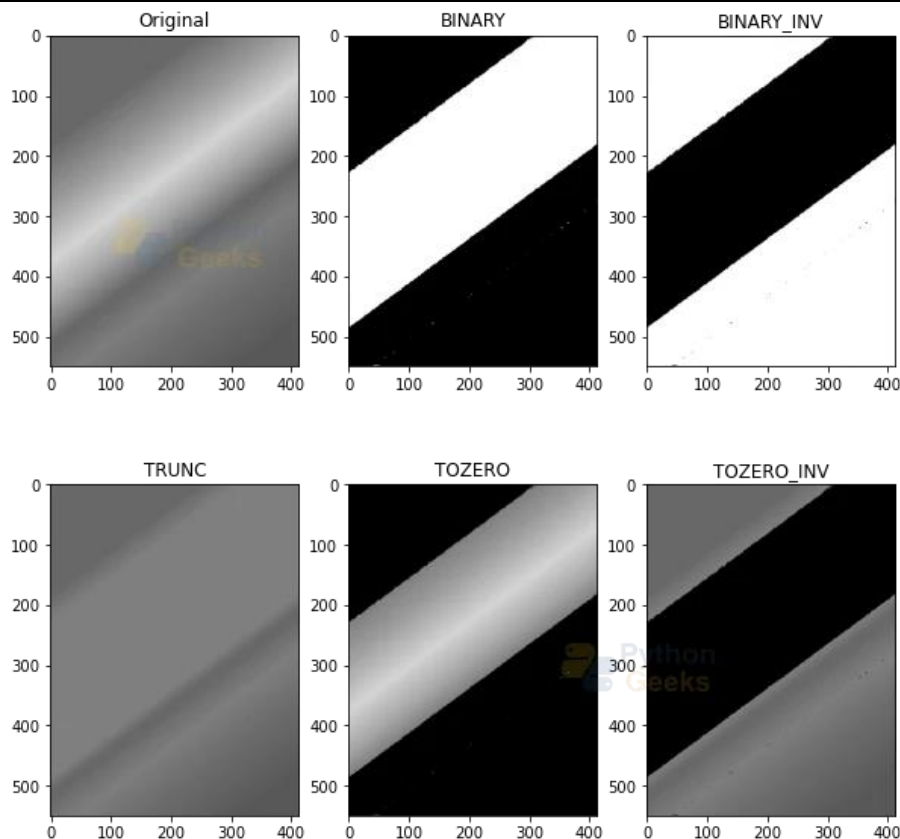
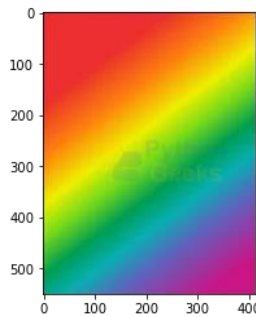
2. Adaptive

- `cv.ADAPTIVE_THRESH_MEAN_C`
- `cv.ADAPTIVE_THRESH_GAUSSIAN_C`

Function	Purpose
<pre># Apply thresholding _, binary_image = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY)</pre>	<ul style="list-style-type: none"> • cv2.threshold() is a function in OpenCV used for thresholding images. • gray_image is the grayscale image that we want to threshold. • 127 is the threshold value. Pixels with intensity values below this threshold will be set to 0 (black), and pixels with intensity values equal to or above this threshold will be set to 255 (white). • 255 is the maximum value that a pixel can have after thresholding. • cv2.THRESH_BINARY is a flag indicating the type of thresholding to be applied. In this case, it means that pixels with intensity values above the threshold will be set to the maximum value (255) and those below will be set to 0. • The underscore <code>_</code> is a convention in Python to indicate that a variable is unused or not needed. In this case, cv2.threshold() returns two values: a return code (indicating <i>success or failure</i> of the operation) and the thresholded image. Since we are not interested in the return code, we use <code>_</code> as a placeholder for it. • binary_image is the variable that will hold the resulting binary image after thresholding.
<pre>ret, image = cv2.threshold(img, 50, 200, cv2.THRESH_BINARY_INV)</pre>	In this type of thresholding all pixels below the threshold value are set to the maximum value

	and all pixels above the threshold value are set to zero.
<pre>ret, image = cv2.threshold(img, 127, 255, cv2.THRESH_TRUNC)</pre>	<ul style="list-style-type: none"> • If a pixel's intensity value exceeds the threshold, it is truncated (clipped) to the threshold value. • Conversely, if the pixel value is below the threshold, it remains unchanged. • Truncated thresholding can be useful for reducing noise in an image.
<pre>ret, image = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO)</pre>	To Zero Thresholding: Pixels below the threshold become black, and others remain unchanged.
<pre>ret, image = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO_INV)</pre>	Opposite of to zero thresholding.

Original Image



Adaptive/ Local/ Dynamic Thresholding

- It is applied for the cases where the lighting conditions are different in different regions of the image and the threshold value is calculated for smaller regions.
- The threshold value for a pixel is determined on the basis of the region around it. Different threshold values are obtained for the different regions of the same image.
- Unlike global thresholding, where a single threshold value is applied to the entire image, adaptive thresholding adjusts the threshold value for each pixel based on the local neighborhood around it.

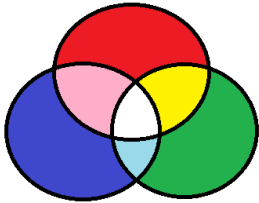
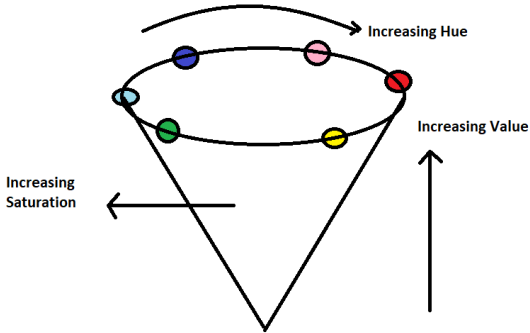
```
cv2.adaptiveThreshold(src, maxVal, adaptiveMethod,
                     thresholdType, blockSize, C)
```

- ✓ **src**: The source image, which first has to be converted to grayscale color space.
- ✓ **maxVal**: The maximum value that pixels exceeding the threshold value can take.
- ✓ **adaptiveMethod**: Type of adaptive thresholding method to be used
- ✓ **thresholdType**: Type of adaptive thresholding to be applied.
- ✓ **blockSize**: Size of a neighborhood of the pixel used to calculate the threshold value. A larger block size may be appropriate if the objects are relatively large and have intricate details or variations in color. However, too large a block size can blur out smaller details and boundaries.
- ✓ **constant**: A constant value that is subtracted from the mean or weighted mean.

```
th2 = cv2.adaptiveThreshold(
    img, 255,
    cv2.ADAPTIVE_THRESH_MEAN_C,
    cv2.THRESH_BINARY, 11, 2)
```

- This method calculates the threshold value for a pixel based on the mean (average) intensity value of its local neighborhood.
 - It computes the mean of the pixel values in a local region defined by a specified window size (block size) centered around each pixel.
 - The threshold value for each pixel is then calculated as the mean value minus a constant value (C), which can be adjusted based on the application requirements.
1. **255**: This is the maximum intensity value that will be assigned to pixels that pass the thresholding criteria. In binary thresholding, pixels with values higher than the threshold are set to this maximum value.
 2. **cv2.ADAPTIVE_THRESH_MEAN_C**: This specifies the adaptive thresholding method to use. In this case, it indicates that the mean (average) of the neighborhood area will be used to calculate the threshold value for each pixel.

	<p>3. cv2.THRESH_BINARY: This specifies the type of thresholding to apply after computing the adaptive threshold value. In this case, it indicates that a binary thresholding operation will be applied, where pixels with intensity values higher than the threshold will be set to the maximum value (255 in this case) and pixels with intensity values lower than the threshold will be set to 0.</p> <p>4. 11: This is the size of the neighborhood area used to compute the adaptive threshold value for each pixel. It defines the size of the square window around each pixel, and the mean of the pixel intensities within this window is used to compute the threshold value.</p> <p>5. 2: This is a constant value (C) subtracted from the mean of the neighborhood area to compute the adaptive threshold value for each pixel. It helps to adjust the threshold value. In this case, it's set to 2.</p>
<pre>th3 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_ C, cv2.THRESH_BINARY, 11, 2)</pre>	<ul style="list-style-type: none"> • This method calculates the threshold value for a pixel based on a weighted sum of the intensity values in its local neighborhood, where the weights are defined by a Gaussian window. • It computes the weighted sum of pixel values in a local region defined by the specified window size (block size) centered around each pixel. • The threshold value for each pixel is then calculated as the weighted sum minus a constant value (C), which can be adjusted based on the application requirements. <p>cv2.ADAPTIVE_THRESH_GAUSSIAN_C: This specifies the adaptive thresholding method to use. In this case, it indicates that a weighted sum of the intensity values in the local neighborhood, with weights defined by a Gaussian window, will be used to calculate the threshold value for each pixel.</p>

Colour Spaces	
<p>Color spaces are a way to represent the color channels present in the image that gives the image that particular hue.</p> <ol style="list-style-type: none"> 1. <i>RGB</i> (Red, Green, Blue), 2. <i>HSV</i> (Hue, Saturation, Value), 3. BGR color space 4. LAB 5. Gray Scale 	
Function	Purpose
 <pre># BGR to RGB image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)</pre>	<p>BGR & RGB Color Space:</p> <ul style="list-style-type: none"> ✚ OpenCV's default color space is RGB. However, it actually stores color in the BGR format. ✚ The different intensities of Blue, Green and Red give different shades of color.
 <pre># Convert to HSV bright_hsv = cv2.cvtColor(bright, cv2.COLOR_BGR2HSV)</pre> <p>Hue values are distributed such that red spans from 0 to 60, yellow from 61 to 120, green from 121 to 180, cyan from 181 to 240, blue from 241 to 300, and magenta from 301 to 360 (looping back to red).</p>	<p>HSV color space (Hue, Saturation, and Value)</p> <ol style="list-style-type: none"> 1. Hue (H): Represents the dominant wavelength of the color. It ranges from 0 to 179 (or 0 to 360 degrees). Hue describes the actual color, such as red, green, blue, etc. 2. Saturation (S): Indicates the purity or shades of the color. It ranges from 0 to 255. Higher saturation values mean more vivid colors, while lower values approach grayscale. Ranging from 0% (gray) to 100% (fully saturated). A saturation value of 0% results in a grayscale image, where all colors appear as shades of gray. As saturation increases, colors become more vivid and vibrant. 3. Value (V): Represents the brightness or intensity of the color. It also ranges from 0 to 255. Higher values indicate brighter colors. Ranging from 0% (black) to 100% (white). <p>The HSV color space is particularly useful for color-based segmentation tasks, such as detecting specific objects or regions based on their color properties. HSV provides a way to</p>

	specify color using just one channel (H), making it robust to lighting variations.
<pre># BGR to Grayscale image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)</pre>	<p>Grayscale Color Space:</p> <ul style="list-style-type: none"> • In the grayscale color space, images are represented as single-channel arrays where pixel values indicate the intensity or brightness of each pixel. • Grayscale images are often represented using 8-bit unsigned integers, where pixel values range from 0 to 255. • Grayscale images require less storage space compared to color images since they have only one channel instead of three (RGB). • Processing grayscale images is computationally less intensive compared to color images, as there's only one channel to process.
<pre>lab_image = cv2.cvtColor(rgb_image, cv2.COLOR_BGR2LAB)</pre>	<p>LAB/ CIELAB Color Space:</p> <ul style="list-style-type: none"> • Designed to approximate human vision and perception more accurately than other color spaces. • It consists of three components: L* (lightness), a* (green-red), and b* (blue-yellow). LAB color space is often used in color-related applications such as color correction, color matching, and image analysis tasks where accurate color representation is important. <ol style="list-style-type: none"> 1. <i>L</i> (Lightness): <ul style="list-style-type: none"> • L* represents the lightness or brightness of the color. It ranges from 0 to 100, where 0 represents black and 100 represents white. • The L* component is independent of color information and only represents the brightness of the color. 2. <i>a</i> (Green-Red): <ul style="list-style-type: none"> • The a* component represents the position of the color along the green-red axis. Positive values indicate shades of red,

	<p>while negative values indicate shades of green.</p> <ul style="list-style-type: none"> The range of values for the a* component typically extends from -128 to +127, with 0 representing neutral gray. <p>3. <i>b</i> (Blue-Yellow):</p> <ul style="list-style-type: none"> The b* component represents the position of the color along the blue-yellow axis. Positive values indicate shades of yellow, while negative values indicate shades of blue. Similar to the a* component, the range of values for the b* component typically extends from -128 to +127, with 0 representing neutral gray.
--	--

Edge Detection	
<ul style="list-style-type: none"> Edge detection is a fundamental technique in image processing used to identify boundaries within images 	
Function	Purpose
<pre>Cv.Canny (image, T_lower, T_upper, aperture_size, L2Gradient) canny = cv.Canny(blur, 125, 175)</pre>	<p>Canny Edge Detection:</p> <ul style="list-style-type: none"> It detects edges by identifying areas of significant intensity gradients in the image. Canny edge detection involves several steps, including noise reduction, gradient calculation, non-maximum suppression, and edge linking by hysteresis thresholding. It produces high-quality edges while minimizing false positives. <p>Refer previous section for entire syntax</p>
<pre>sobelx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=kernel_size) sobely = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=kernel_size) edges = cv2.magnitude(sobelx, sobely)</pre>	<p>Sobel Edge Detection:</p> <ul style="list-style-type: none"> Sobel edge detection applies convolution with Sobel kernels to compute the gradient magnitude of the image. It calculates gradients separately in the horizontal and vertical directions and then combines them to obtain the overall gradient magnitude.

	<ul style="list-style-type: none"> Sobel edge detection is simpler than Canny and can be faster, but it may produce weaker edges and more noise. <p>Refer previous section for entire syntax</p>
<pre>scharrx = cv2.Scharr(image, cv2.CV_64F, 1, 0) scharry = cv2.Scharr(image, cv2.CV_64F, 0, 1) edges = cv2.magnitude(scharrx, scharry)</pre>	<p>Scharr Edge Detection:</p> <ul style="list-style-type: none"> Scharr edge detection is similar to Sobel but uses a different kernel for gradient calculation. It provides better edge detection results compared to Sobel due to the use of a more accurate kernel. <p>cv2.CV_64F is a constant representing the data type of the image or kernel. CV_64F stands for a 64-bit floating-point data type.</p> <ul style="list-style-type: none"> A 64-bit floating point number (double precision) uses 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. This format allows for a wider range of representable values and greater precision compared to 32-bit floating point numbers. Using 64-bit floating point can help avoid loss of information due to rounding errors
<pre>edges = cv2.Laplacian(image, cv2.CV_64F)</pre>	<p>Laplacian Edge Detection:</p> <ul style="list-style-type: none"> Laplacian edge detection calculates the second derivative of the image to detect areas of rapid intensity changes. It is sensitive to noise and may produce thicker edges compared to other techniques.
<pre>laplacian = cv2.Laplacian(image, cv2.CV_64F) edges = cv2.Canny(laplacian, threshold1, threshold2)</pre>	<p>Zero Crossing Edge Detection:</p> <ul style="list-style-type: none"> Zero crossing edge detection identifies edges by detecting sign changes in the second derivative of the image. It is often used in combination with Laplacian edge detection to produce thinner and more precise edges.

Contour Detection

- Boundaries of object, the line/ curves that joins the continuous points along boundary.
- It is not as same as edges. Contours are curves formed by joining continuous points along the boundary of an image. These points have the same color or intensity.
- Used for shape analysis, object detection & recognition.

Contour Detection Overview:**1. Edge Detection:**

- Before detecting contours, it's common to perform edge detection to highlight the boundaries of objects in the image.
- Various edge detection algorithms such as Canny, Sobel, or Laplacian can be used to generate a binary image containing edge pixels.

2. Contour Detection:

- Once edges are detected, contours can be identified using the **findContours()** function in OpenCV.
- This function takes a binary image as input and identifies contours based on connected components of edge pixels.
- Contours are represented as lists of points (contour coordinates) or as hierarchical structures.

3. Hierarchy of Contours:

- Contours in OpenCV can be hierarchical, meaning that they may have parent-child relationships.
- The hierarchy of contours can be useful for tasks such as finding contours within other contours or identifying holes within objects.

4. Contour Features:

- After detecting contours, various properties or features of the contours can be computed.
- Common contour features include area, perimeter, centroid, bounding box, and convex hull.
- These features can be used for object classification, shape analysis, and other tasks.

5. Contour Visualization:

- OpenCV provides functions for visualizing contours on images, such as **drawContours()**.
- This function draws the contours on a specified image, allowing for visualization and analysis of the detected contours.

Function	Purpose
<p>Syntax</p> <pre>contours, hierarchy = cv.findContours(image, mode, method[, contours[, hierarchy[, offset]]) contours, hierarchy = cv2.findContours(canny_edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)</pre>	<p>The findContours function in OpenCV is used to detect contours in an image. It identifies the boundaries of objects or shapes within the image.</p> <ul style="list-style-type: none"> • image: The input image (usually a binary or grayscale image). • mode: The contour Retrieval mode. Common values include: <ul style="list-style-type: none"> ○ cv2.RETR_EXTERNAL: Retrieves only the external contours. ○ cv2.RETR_LIST: Retrieves all contours without any hierarchy. ○ cv2.RETR_TREE: Retrieves all contours and arranges them

	<p>in a hierarchical tree structure. This mode captures both external and internal contours (nested contours)</p> <ul style="list-style-type: none"> method: The contour approximation method. Common values include: <p>cv2.CHAIN_APPROX_SIMPLE: Compresses horizontal, vertical, and diagonal segments into their endpoints. (Reduce memory consumption and speeds up processing.) This parameter specifies the contour approximation method used during contour detection.</p> <p>When contours are detected, they consist of a series of points that form the boundary of an object.</p> <p>However, not all points are necessary to represent the contour accurately. Many points can be redundant, especially when the contour is relatively simple (e.g., a straight line or a simple curve). It simplifies the contour by removing redundant points and compressing it. It retains only the essential points needed to describe the shape.</p> <p>cv2.CHAIN_APPROX_NONE: Stores all the contour points.</p> <ol style="list-style-type: none"> contours: A list containing the detected contours. hierarchy: A representation of the hierarchical relationships between contours (parent-child relationships). offset: An optional parameter to adjust the contour coordinates. <p>If you're interested in just the contours and not the hierarchy, you can simplify the assignment as follows:</p> <pre>contours = cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)[0]</pre> <p>This will give you a list of contours without the hierarchy information.</p>
<pre>cv2.drawContours(image, [contour], -1, (0, 255, 0), 2)</pre>	<ol style="list-style-type: none"> image: <ul style="list-style-type: none"> This parameter specifies the image on which the contours will be drawn.

	<ul style="list-style-type: none"> • It is the original image where contours were detected and will be visualized. <p>2. [contour]:</p> <ul style="list-style-type: none"> • This parameter specifies the contours to be drawn. • Contours are typically represented as lists of points, but cv2.drawContours() expects contours to be passed as a list of contours, hence the square brackets around contour. • Here, we're drawing a single contour specified by the contour variable. <p>3. -1:</p> <ul style="list-style-type: none"> • This parameter specifies the index of the contour to draw. • When set to -1, all contours in the list are drawn. • If set to a non-negative integer, only the contour at that index in the list will be drawn. <p>4. (0, 255, 0):</p> <ul style="list-style-type: none"> • This parameter specifies the color of the drawn contours. • It is specified as a tuple representing the (B, G, R) values of the color. • In this case, the color is green, as (0, 255, 0) represents full green with no blue or red components. <p>5. 2:</p> <ul style="list-style-type: none"> • This parameter specifies the thickness of the contour lines. • It determines how thick the contour lines will be drawn. • Here, the contour lines will be drawn with a thickness of 2 pixels.
<p>1. Area and Perimeter Calculation:</p> <ul style="list-style-type: none"> • Contours can be used to compute the area and perimeter of objects in an image using the cv2.contourArea() and cv2.arcLength() functions, respectively. <p>2. Bounding Box:</p>	

<ul style="list-style-type: none"> Contours can be enclosed within a bounding box using functions like cv2.boundingRect() or cv2.minAreaRect(). These functions provide information about the bounding rectangle or minimum area rotated rectangle that encloses the contour. 	
<pre>area = cv2.contourArea(cnt)</pre>	<p>cv2.contourArea():</p> <ul style="list-style-type: none"> This function calculates the area enclosed by a contour. It can be used directly on a contour or derived from the moments of the contour. For a given contour cnt, you can compute the area as follows: The result is the total area within the contour.
<pre>perimeter = cv2.arcLength(cnt, True)</pre>	<p>cv2.arcLength():</p> <ul style="list-style-type: none"> Also known as contour perimeter, it calculates the length of the contour. The second argument specifies whether the shape is a closed contour (if passed True) or just a curve. For a given contour cnt, you can compute the perimeter as follows: The result represents the total length of the contour.
<pre>x, y, w, h = cv2.boundingRect(contour) rect = (x, y, w, h)</pre>	<p>cv2.boundingRect():</p> <ul style="list-style-type: none"> The boundingRect() function computes an upright rectangle that encloses the given contour. It returns a rectangle whose bottom edge is parallel to the x-axis. The rectangle is the smallest possible bounding box aligned with the image axes. It does not consider rotations or orientation of the contour. Useful for quickly enclosing a shape with a simple rectangle.
<pre>rect = cv2.minAreaRect(contour)</pre>	<p>cv2.minAreaRect():</p> <ul style="list-style-type: none"> The minAreaRect() function computes the minimum area rectangle that encloses the given contour.

	<ul style="list-style-type: none"> ○ It considers rotations about the contour, allowing it to find the smallest possible rectangle relative to the total area. ○ The resulting rectangle may be rotated (not necessarily upright). ○ Useful when you need to find the tightest bounding box around a rotated object.
--	---

Masking

Image masking allows us to focus on specific regions of an image by creating a mask that highlights the areas of interest.

1. Bitwise Operations and Masks:

- We can use bitwise operations and masks to construct ROIs that are non-rectangular.
- For example, if we want to recognize faces in an image, we can create a mask to show only the face regions.
- To perform image masking using OpenCV, you can use the **cv2.bitwise_and()** function to compute the bitwise AND between the mask and the image.
- **Steps:**
 1. Import the required libraries (OpenCV and NumPy).
 2. Read the input image using **cv2.imread()**.
 3. Convert the image from BGR to HSV (if needed).
 4. Define a mask in HSV color space using **cv2.inRange()** with lower and upper limits of color values.
 5. Apply the mask using **cv2.bitwise_and()** to extract the regions of interest.

Example Code:

```
import cv2
import numpy as np

# Read the input image
image = cv2.imread('input_image.jpg')

# Convert BGR to HSV
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Define a mask for a specific color (e.g., blue)
lower_blue = np.array([100, 50, 50])
upper_blue = np.array([130, 255, 255])
mask = cv2.inRange(hsv_image, lower_blue, upper_blue)
```



```
# Apply the mask to the original image
masked_image = cv2.bitwise_and(image, image, mask=mask)

# Display the results
cv2.imshow('Original Image', image)
cv2.imshow('Masked Image', masked_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Steps involved in Processing an Image:

Image processing involves a **fixed sequence of operations** that are performed at each pixel of an image.

1. **Image Acquisition:** The process begins with capturing an image using a camera. If the camera output is not already digitized, it's converted into digital form using an analog-to-digital converter. This digitized image becomes the input for further processing in a computer.
2. **Image Enhancement:** Enhancing the quality of the image by adjusting brightness, contrast, and sharpness. Techniques like histogram equalization, contrast stretching, and noise reduction fall under this category.
3. **Image Restoration:** Restoring images that have been degraded due to noise, blur, or other factors. Restoration techniques aim to recover the original image from its degraded version.
4. **Color Image Processing:** Handling color images by manipulating their color channels (such as RGB or HSV). Tasks include color correction, colorization, and color-based segmentation.
5. **Wavelets and Multi-Resolution Processing:** Using wavelet transforms to analyze and process images at different scales. This helps in compression, denoising, and feature extraction.
6. **Image Compression:** Reducing the size of an image while preserving essential information. Common compression methods include JPEG and PNG.
7. **Morphological Processing:** Morphological operations involve shape-based transformations on an image. Techniques like erosion, dilation, and opening/closing are used for feature extraction and noise reduction.
8. **Image Segmentation:** Dividing an image into meaningful regions or objects. Segmentation is crucial for tasks like object recognition, tracking, and medical image analysis.

OpenCV for Image Processing and Computer Vision

1. Accessing and Modifying Pixel Values:

- You can access and modify pixel values in an image using coordinates. For example, to change the color of a pixel at position (100, 100) to white, you can do:
- `img[100, 100] = [255, 255, 255]`

2. Image Properties:

- You can retrieve information about an image, such as its shape (rows, columns, and channels), size, and data type:
- `shape = img.shape`
- `size = img.size`
- `dtype = img.dtype`

3. Geometric Transformations:

- Rotate, translate, scale, or warp an image using geometric transformations. For example, to rotate an image by 45 degrees:
- `M = cv.getRotationMatrix2D((cols/2, rows/2), 45, 1)`
- `rotated_img = cv.warpAffine(img, M, (cols, rows))`

4. Thresholding:

- Convert a grayscale image into a binary image by setting a threshold. Useful for segmentation and object detection.

5. Smoothing (Blurring):

- Apply filters to reduce noise or blur an image. Common filters include Gaussian blur and median blur.

6. Morphological Transformations:

- Perform operations like erosion, dilation, opening, and closing to manipulate image shapes.

7. Edge Detection:

- Use techniques like the Canny edge detector to find edges in an image.

8. Template Matching:

- Locate a template (small image) within a larger image.

9. Color Recognition:

- Identify specific colors in an image using color space conversions and thresholding.

10. Object Detection:

- Detect faces, eyes, or other objects in an image using pre-trained models or custom classifiers.

Reference:

1. <https://cheatography.com/thatguyandy27/cheat-sheets/open-cv/pdf/?last=1505666439>
2. <https://github.com/jasmcaus/opencv-course/>
3. <https://www.naturefocused.com/articles/photography-image-processing-kernel.html>
4. <https://codefather.tech/blog/image-edge-detection-python/>