Table of Contents

[Chapter One:- Image Reading and Basic 5](#_Toc210425151)

[Access Google Drive files: 5](#_Toc210425152)

[Confirm working directory & files: 5](#_Toc210425153)

[Set Path: 5](#_Toc210425154)

[Read Image 5](#_Toc210425155)

[Inspect the Height, Width and Channel 5](#_Toc210425156)

[Display Image 5](#_Toc210425157)

[Color Space Conversion 5](#_Toc210425158)

[WebCame View Using OpenCv 6](#_Toc210425159)

[Displaying Each Color Channel ( Red, Green and Blue) 6](#_Toc210425160)

[HSV image into H, S, V channels and display them individually. 6](#_Toc210425161)

[Practical Uses of HSV 7](#_Toc210425162)

[HSV Color Range 7](#_Toc210425163)

[Extracting Green Color of Image using Masking of HSV image: 7](#_Toc210425164)

[Splitting Image into three Channel 8](#_Toc210425165)

[Point Operation: 10](#_Toc210425166)

[Brightness & Contrast Adjustment: cv2.convertScaleAbs() 10](#_Toc210425167)

[Manual Point Operation Using NumPy: 10](#_Toc210425168)

[Safe Image Saving Function 11](#_Toc210425169)

[Webcam Video Capture & Recording 11](#_Toc210425170)

[Image Manipulations: Crop & Draw: 12](#_Toc210425171)

[Drawing a Line, Rectangle, Circle, Eclipse, Text on the Image: 12](#_Toc210425172)

[Brainstorm/Experiments 13](#_Toc210425173)

[Chapter Two- Image Sculpting 14](#_Toc210425174)

[Learning Goals 14](#_Toc210425175)

[Histogram Calculation & Visualization 14](#_Toc210425176)

[Histogram: cv2.calcHist() 14](#_Toc210425177)

[Color Image Histogram 14](#_Toc210425178)

[Matplotlib Histogram (All Pixels) 15](#_Toc210425179)

[Grayscale Histogram 15](#_Toc210425180)

[Side-by-Side Histogram Visualization 15](#_Toc210425181)

[Histogram Interpretation 16](#_Toc210425182)

[Global Histogram Equalization 16](#_Toc210425183)

[Global Histogram Equalization – Visualization 16](#_Toc210425184)

[Compute Histogram and CDF 16](#_Toc210425185)

[Apply Histogram Equalization 17](#_Toc210425186)

[Key Notes 17](#_Toc210425187)

[Contrast Limited Adaptive Histogram Equalization (CLAHE) 17](#_Toc210425188)

[Convolution 18](#_Toc210425189)

[Image Filtering with 2‑D Convolution: Averaging, Sharpening, Sobel 20](#_Toc210425190)

[Averaging (Blurring) 20](#_Toc210425191)

[Manually: Convolution Image with Box3 filter: 20](#_Toc210425192)

[Using OpenCV (cv2.filter2D) 20](#_Toc210425193)

[Manually: Convolution Image Sharpen Filter : 21](#_Toc210425194)

[Manual Convolution : Sobel Kernel 22](#_Toc210425195)

[Sobel Filter: Using OpenCV 23](#_Toc210425196)

[Apply 2‑D Convolution: Defining Custom Function: conv2d 24](#_Toc210425197)

[Key Notes 25](#_Toc210425198)

[Kernel Types and Effects 25](#_Toc210425199)

[Filtering with OpenCV: cv2.filter2D 25](#_Toc210425200)

[filter2D vs Custom Convolution 25](#_Toc210425201)

[Custom 2D Convolution 26](#_Toc210425202)

[Image Blurring Techniques: Mean Blurring 26](#_Toc210425203)

[Manual Implementation of Mean Blurring 27](#_Toc210425204)

[OpenCV Implementation of Mean Blurring 27](#_Toc210425205)

[Median Blurring: cv2.medianBlur(img\_gray, kernel\_size) 27](#_Toc210425206)

[Gaussian Blurring: cv2.GaussianBlur(img, (Kernel\_size, Kernel\_size), sigmaX=sigmaX, sigmaY=sigmaY) 28](#_Toc210425207)

[Manually Implementation: 28](#_Toc210425208)

[OpenCv Implementation of Gaussian Blur 29](#_Toc210425209)

[Different Values of Kernel and Sigma in Gaussian Filter 30](#_Toc210425210)

[Reports of Different Size of Kernel and Sigma Value of Gaussian Filter 30](#_Toc210425211)

[Applying: 'Averaging', 'Gaussian', 'Median': Kernel Size 5, SigmaX = 0 31](#_Toc210425212)

[**Findings** 31](#_Toc210425213)

[Different Kernel Size Effects on Averaging / Median / Gaussian 32](#_Toc210425214)

[Key Takeaways 32](#_Toc210425215)

[Bilateral Filter: cv2.bilateralFilter(img, d=d, sigmaColor=sigma, sigmaSpace=sigma) 33](#_Toc210425216)

[**Key Points about OpenCV Bilateral Filter:** 33](#_Toc210425217)

[Manual Implementation of Bilateral Filter 33](#_Toc210425218)

[Opencv Implementation Bilateral Filter 34](#_Toc210425219)

[Different Values of d\_values and Sigma values Bilateral Filter 34](#_Toc210425220)

[**SigmaColor:** 35](#_Toc210425221)

[**SigmaSpace:** 35](#_Toc210425222)

[**d value:** 35](#_Toc210425223)

[**Combined Effect** 35](#_Toc210425224)

[Salt & Pepper Noise Filtering: Using Median, Guassian, Bilateral 35](#_Toc210425225)

[Gaussian Noise Addition and Filtering 37](#_Toc210425226)

[Code to Add gaussian Noise: 37](#_Toc210425227)

[Gaussian Noise Filtering: Using Mean, Median, Guassian, Bilateral 37](#_Toc210425228)

[**1️. Mean Filter (Averaging)** 38](#_Toc210425229)

[**2️. Gaussian Blur** 39](#_Toc210425230)

[**3️. Median Blur** 39](#_Toc210425231)

[**4️. Bilateral Filter** 39](#_Toc210425232)

[2. Filtering Noisy Images 39](#_Toc210425233)

[4. Notes for Color Images 39](#_Toc210425234)

[Chapter Three:- Finding Primitives (Edge and Corner Detection) 40](#_Toc210425235)

[Gradient Edge Detection (Sobel) 40](#_Toc210425236)

[Laplacian Edge Detection 42](#_Toc210425237)

[**Visual analogy** 42](#_Toc210425238)

[Canny Edge Detection 43](#_Toc210425239)

[Steps Involved 43](#_Toc210425240)

[OpenCV Implementation 44](#_Toc210425241)

[Key Parameters in OpenCV 47](#_Toc210425242)

[**1️. Increasing thresholds** 47](#_Toc210425243)

[**2️. Decreasing thresholds** 47](#_Toc210425244)

[**3️. Practical tip** 47](#_Toc210425245)

[Harris Corner Detection 47](#_Toc210425246)

[Manual Implementation Using Gradients 48](#_Toc210425247)

[Key Points / Observations 50](#_Toc210425248)

[Marking Detected Harris Corners 50](#_Toc210425249)

[Harris Response vs Dilated Response 52](#_Toc210425250)

[Harris Response (harris\_response) 52](#_Toc210425251)

[2. Harris Dilated (harris\_dilated) 52](#_Toc210425252)

[Hough Line Transform (HLT) 52](#_Toc210425253)

[Applications of Hough Line Transform 52](#_Toc210425254)

[Steps in Hough Line Transform 53](#_Toc210425255)

[Step 3: Hough Line Transform (Standard) 53](#_Toc210425256)

[Step 4: Draw detected lines 53](#_Toc210425257)

[**Hough Circle:** 53](#_Toc210425258)

[6. Key Points 54](#_Toc210425259)

[Summary 55](#_Toc210425260)

[Morphological Transformations 55](#_Toc210425261)

[Applications of Dilution / Dilation 56](#_Toc210425262)

[Opening 56](#_Toc210425263)

[Top Hat (White Top Hat) 58](#_Toc210425264)

[Black Hat 58](#_Toc210425265)

[Chapter Four- Feature Description: 60](#_Toc210425266)

[Replacing the certain portion of Perspective image with normal image 60](#_Toc210425267)

[Feature Matching using SIFT and Image Alignment: 61](#_Toc210425268)

[Cell 1: Load and Display Images 61](#_Toc210425269)

[Cell 2 – Detect SIFT Keypoints and Descriptors 62](#_Toc210425270)

# Chapter One:- Image Reading and Basic

## Access Google Drive files:

from google.colab import drive

drive.mount('/content/drive')

## Confirm working directory & files:

!pwd

!ls "drive/MyDrive/Colab Notebooks/CV\_2025/LAB01/assets/"

## Set Path:

Use correct file path (Colab/local):

img\_path = '/content/drive/MyDrive/.../photo.png'

img\_path\_local = 'assets/photo.png'

## Read Image

Load image with OpenCV:

import cv2

img\_bgr = cv2.imread(img\_path\_local)

assert img\_bgr is not None, "Image not found." # verifying uploaded

|  |  |  |
| --- | --- | --- |
| **Flag** | **Value** | **Meaning** |
| cv2.IMREAD\_COLOR | 1 (default) | Loads a **color image** (ignores alpha channel). |
| cv2.IMREAD\_GRAYSCALE | 0 | Loads the image in **grayscale** mode. |
| cv2.IMREAD\_UNCHANGED | -1 | Loads the image **as-is**, including alpha channel if present. |

## Inspect the Height, Width and Channel

Check size & data type:

print(img\_bgr.shape) # (H, W, C)

print(img\_bgr.dtype) # usually uint8

## Display Image

**Colab:**

from google.colab.patches import cv2\_imshow

cv2\_imshow(img\_bgr)

**Jupyter (matplotlib):**

import matplotlib.pyplot as plt

plt.imshow(cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB))

plt.axis('off')

plt.show()

**Local script:**

cv2.imshow('Image', img\_bgr)

cv2.waitKey(0)

cv2.destroyAllWindows()

## Color Space Conversion

# --- BGR ↔ RGB ---

img\_path = ''  
img\_bgr = cv2.imread(img\_path)

img\_rgb = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# --- BGR ↔ Grayscale ---

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# --- BGR ↔ HSV ---

img\_hsv = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2HSV)

# --- BGR ↔ LAB ---

img\_lab = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2LAB)

## WebCame View Using OpenCv

import cv2

cap = cv2.VideoCapture(0)

if not cap.isOpened():

print("Cannot open camera")

exit()

print(" Press 'q' to quit the video window.")

while True:

ret, frame = cap.read()

if not ret:

print(" Can't receive frame (stream end?). Exiting ...")

break

cv2.imshow('Webcam Feed (BGR)', frame)

if cv2.waitKey(1) & 0xFF == ord('q'):

break

cap.release()

cv2.destroyAllWindows()

## Displaying Each Color Channel ( Red, Green and Blue)

import cv2

import matplotlib.pyplot as plt

B, G, R = cv2.split(img\_bgr)

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.imshow(R, cmap='Reds')

plt.title('Red')

plt.axis('off')

plt.subplot(1, 3, 2)

plt.imshow(G, cmap='Greens')

plt.title('Green')

plt.axis('off')

plt.subplot(1, 3, 3)

plt.imshow(B, cmap='Blues')

plt.title('Blue')

plt.axis('off')

plt.show()

## HSV image into H, S, V channels and display them individually.

import cv2

import matplotlib.pyplot as plt

# Convert BGR to HSV

img\_hsv = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2HSV)

# Separate channels explicitly

H = img\_hsv[:, :, 0]

S = img\_hsv[:, :, 1]

V = img\_hsv[:, :, 2]

# Display each channel

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)

plt.imshow(H, cmap='hsv')

plt.title('Hue (H)')

plt.axis('off')

plt.subplot(1, 3, 2)

plt.imshow(S, cmap='gray')

plt.title('Saturation (S)')

plt.axis('off')

plt.subplot(1, 3, 3)

plt.imshow(V, cmap='gray')

plt.title('Value (V)')

plt.axis('off')

plt.show()

## Practical Uses of HSV

|  |  |
| --- | --- |
| Use Case | Explanation |
| Color Detection | Detects specific colors by checking Hue ranges (e.g., red ball detection). |
| Robust Segmentation | Ignoring Value (brightness) makes segmentation stable under different lighting. |
| Skin Detection / Human Tracking | Narrow Hue + Saturation ranges cover most skin tones, useful for tracking. |
| Image Enhancement | Adjust Saturation for vivid colors and Value for brightness control. |
| Feature Extraction | Hue is used for color-based recognition, while Value helps capture texture/brightness. |
| Color-Based Tracking | Tracking methods like CAMShift/MeanShift work effectively with hue masks. |

## HSV Color Range

|  |  |  |
| --- | --- | --- |
| **Color** | **Lower HSV** | **Upper HSV** |
| Red | [0,50,50] & [170,50,50] | [10,255,255] & [180,255,255] |
| Green | [35,50,50] | [85,255,255] |
| Blue | [100,50,50] | [130,255,255] |
| Yellow | [20,50,50] | [30,255,255] |
| Cyan | [85,50,50] | [100,255,255] |
| Magenta | [140,50,50] | [170,255,255] |

## Extracting Green Color of Image using Masking of HSV image:

**import cv2**

import numpy as np

import matplotlib.pyplot as plt

# Define green range in HSV

lower\_green = np.array([35, 50, 50])

upper\_green = np.array([85, 255, 255])

# Convert image to HSV if not already

img\_hsv = cv2.cvtColor(img\_rgb, cv2.COLOR\_RGB2HSV)

# Create mask for green

mask = cv2.inRange(img\_hsv, lower\_green, upper**\_green)**

# Apply mask to original image

result = cv2.bitwise\_and(img\_rgb, img\_rgb, mask=mask)

# Display results

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.imshow(img\_rgb)

plt.title("Original RGB")

plt.axis("off")

plt.subplot(1, 3, 2)

plt.imshow(mask, cmap="gray")

plt.title("Green Mask")

plt.axis("off")

plt.subplot(1, 3, 3)

plt.imshow(result)

plt.title("Filtered Green")

plt.axis("off")

plt.tight\_layout()

plt.show()

## Splitting Image into three Channel

import cv2

import matplotlib.pyplot as plt

B, G, R = cv2.split(img\_bgr)

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.imshow(R, cmap='Reds')

plt.title('Red')

plt.axis('off')

plt.subplot(1, 3, 2)

plt.imshow(G, cmap='Greens')

plt.title('Green')

plt.axis('off')

plt.subplot(1, 3, 3)

plt.imshow(B, cmap='Blues')

plt.title('Blue')

plt.axis('off')

plt.show()

Task1: # TASK:

1. Use the image redball.jpg

2. Load the image using opencv

3. Show the image using matplotlib

4. Show ONLY the red cricket ball in the image using the image.

### BONUS: Lighting conditions may require tuning: you can use track bars in OpenCV to find the perfect ranges interactively.

Answer:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load image

img\_bgr = cv2.imread("redball.jpg")

img\_rgb = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# Show original image

plt.imshow(img\_rgb)

plt.title("Original Image")

plt.axis("off")

plt.show()

# Convert to HSV

hsv = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2HSV)

# Define range for red color in HSV

# Red is tricky since it spans around 0° and 180° in Hue

lower\_red1 = np.array([0, 100, 100])

upper\_red1 = np.array([10, 255, 255])

lower\_red2 = np.array([160, 100, 100])

upper\_red2 = np.array([179, 255, 255])

# Create masks for red ranges

mask1 = cv2.inRange(hsv, lower\_red1, upper\_red1)

mask2 = cv2.inRange(hsv, lower\_red2, upper\_red2)

mask = mask1 | mask2

# Apply mask to original image

red\_ball = cv2.bitwise\_and(img\_rgb, img\_rgb, mask=mask)

# Show detected red ball

plt.imshow(red\_ball)

plt.title("Red Ball Detected")

plt.axis("off")

plt.show()

# ---------------- BONUS: Interactive HSV Trackbars ---------------- #

def nothing(x):

pass

cv2.namedWindow("Trackbars")

# Create trackbars for Hue, Saturation, and Value

cv2.createTrackbar("LH", "Trackbars", 0, 179, nothing)

cv2.createTrackbar("UH", "Trackbars", 179, 179, nothing)

cv2.createTrackbar("LS", "Trackbars", 100, 255, nothing)

cv2.createTrackbar("US", "Trackbars", 255, 255, nothing)

cv2.createTrackbar("LV", "Trackbars", 100, 255, nothing)

cv2.createTrackbar("UV", "Trackbars", 255, 255, nothing)

while True:

# Get values from trackbars

lh = cv2.getTrackbarPos("LH", "Trackbars")

uh = cv2.getTrackbarPos("UH", "Trackbars")

ls = cv2.getTrackbarPos("LS", "Trackbars")

us = cv2.getTrackbarPos("US", "Trackbars")

lv = cv2.getTrackbarPos("LV", "Trackbars")

uv = cv2.getTrackbarPos("UV", "Trackbars")

lower\_bound = np.array([lh, ls, lv])

upper\_bound = np.array([uh, us, uv])

mask = cv2.inRange(hsv, lower\_bound, upper\_bound)

result = cv2.bitwise\_and(img\_rgb, img\_rgb, mask=mask)

cv2.imshow("Mask", mask)

cv2.imshow("Result", result)

if cv2.waitKey(1) & 0xFF == 27: # Press ESC to exit

break

cv2.destroyAllWindows()

# Point Operation:

## Brightness & Contrast Adjustment: cv2.convertScaleAbs()

Function: cv2.convertScaleAbs(src, alpha=1, beta=0)

|  |  |  |
| --- | --- | --- |
| Parameter | Meaning | Extra Explanation |
| src | Input image (grayscale or color). | It is the original image whose pixel values will be transformed. |
| alpha | Scale factor (contrast). | Values >1 increase contrast, values between 0–1 decrease contrast. |
| beta | Offset added to pixels (brightness). | Positive values brighten the image, negative values darken it. |

Code:

img\_bgr = cv2.imread(img\_path)

alpha = 3 # contrast factor (>1 increase, <1 decrease)

beta = -50 # brightness offset (+ brighten, - darken)

bright\_contrast = cv2.convertScaleAbs(img\_bgr, alpha=alpha, beta=beta)

## Manual Point Operation Using NumPy:

alpha = 0.8

beta = -30

# Convert to float32 for computation

float\_img\_bgr = img\_bgr.astype(np.float32)

# Multiply by alpha (contrast)

float\_img\_bgr = float\_img\_bgr \* alpha

# Add beta (brightness)

float\_img\_bgr = float\_img\_bgr + beta

# Clip values to [0,255]

result = np.clip(float\_img\_bgr, 0, 255)

# Convert back to uint8

converted\_img = result.astype(np.uint8)

# Show

plt.imshow(cv2.cvtColor(converted\_img, cv2.COLOR\_BGR2RGB))

## Safe Image Saving Function

import os

import cv2

def safe\_imwrite(path, img, color\_space="BGR"):

os.makedirs(os.path.dirname(path), exist\_ok=True)

if color\_space.upper() == "RGB":

img\_to\_save = cv2.cvtColor(img, cv2.COLOR\_RGB2BGR)

elif color\_space.upper() == "HSV":

img\_to\_save = cv2.cvtColor(img, cv2.COLOR\_HSV2BGR)

elif color\_space.upper() == "GRAY":

img\_to\_save = img

else:

img\_to\_save = img

ok = cv2.imwrite(path, img\_to\_save)

print(f"Saved? {ok} → {path}")

## Webcam Video Capture & Recording

import cv2

import datetime

cap = cv2.VideoCapture(0) # default camera

if not cap.isOpened():

print("Cannot open camera")

exit()

recording = False

out = None

while True:

ret, frame = cap.read()

if not ret:

print("Can't receive frame. Exiting...")

break

cv2.imshow('BGR Frame', frame)

key = cv2.waitKey(1) & 0xFF

if key == ord('q'):

break

elif key == ord('s'): # Save snapshot

filename = f"snapshot\_{datetime.datetime.now().strftime('%Y%m%d\_%H%M%S')}.jpg"

cv2.imwrite(filename, frame)

print(f"Snapshot saved as {filename}")

elif key == ord('c'): # Toggle video recording

if not recording:

fourcc = cv2.VideoWriter\_fourcc(\*'XVID')

video\_filename = f"video\_{datetime.datetime.now().strftime('%Y%m%d\_%H%M%S')}.avi"

out = cv2.VideoWriter(video\_filename, fourcc, 20.0, (frame.shape[1], frame.shape[0]))

recording = True

print(f"Started recording: {video\_filename}")

else:

recording = False

out.release()

print("Stopped recording")

if recording and out is not None:

out.write(frame)

if out is not None:

out.release()

cap.release()

cv2.destroyAllWindows()

Notes / Shortcuts

* **q** → Quit webcam window.
* **s** → Save a snapshot with timestamp.
* **c** → Start/stop video recording.
* Uses **cv2.VideoWriter** to save AVI video with XVID codec.
* cv2.waitKey(1) ensures smooth frame updates.

## Image Manipulations: Crop & Draw:

**Cropping (ROI)**

# Another crop (Mouse)

y1, y2 = 100, 200 # example values

x1, x2 = 150, 300

crop\_mouse = img\_rgb[y1:y2, x1:x2]

plt.imshow(crop\_mouse)

plt.title("Mouse Crop")

plt.axis("off")

plt.show()

# Crop by slicing: img[y1:y2, x1:x2]

y1, y2 = 50, h//5

x1, x2 = 60, w//2

crop\_glass = img\_rgb[y1:y2, x1:x2]

plt.imshow(crop\_glass)

plt.title("Glass Crop")

plt.axis("off")

plt.show()

## Drawing a Line, Rectangle, Circle, Eclipse, Text on the Image:

import cv2

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

canvas = img\_bgr.copy()

# ------------------------

# 1️ Draw a Line

# ------------------------

# cv2.line(image, start\_point, end\_point, color, thickness)

start\_point = (50, 50) # Starting coordinate (x, y)

end\_point = (200, 50) # Ending coordinate (x, y)

line\_color = (255, 0, 0) # Color in BGR (Blue)

line\_thickness = 2 # Thickness of the line

cv2.line(canvas, start\_point, end\_point, line\_color, line\_thickness)

# ------------------------

# 2️ Draw a Rectangle

# ------------------------

# cv2.rectangle(image, top\_left, bottom\_right, color, thickness)

top\_left = (50, 100) # Top-left corner (x, y)

bottom\_right = (200, 200) # Bottom-right corner (x, y)

rect\_color = (0, 0, 255) # Red in BGR

rect\_thickness = 3 # Thickness of rectangle border (-1 for filled)

cv2.rectangle(canvas, top\_left, bottom\_right, rect\_color, rect\_thickness)

# ------------------------

# 3️ Draw a Circle

# ------------------------

# cv2.circle(image, center, radius, color, thickness)

center = (150, 300) # Center of circle (x, y)

radius = 50 # Radius of the circle

circle\_color = (0, 255, 0) # Green in BGR

circle\_thickness = 4 # Thickness (-1 for filled circle)

cv2.circle(canvas, center, radius, circle\_color, circle\_thickness)

# ------------------------

# 4️ Draw an Ellipse

# ------------------------

# cv2.ellipse(image, center, axes, angle, startAngle, endAngle, color, thickness)

ellipse\_center = (300, 100) # Center of ellipse (x, y)

axes = (80, 40) # Length of major and minor axes

angle = 30 # Rotation angle of ellipse in degrees

startAngle = 0 # Start of the elliptic arc in degrees

endAngle = 360 # End of the elliptic arc in degrees

ellipse\_color = (255, 255, 0) # Cyan in BGR

ellipse\_thickness = 3 # Thickness (-1 for filled)

cv2.ellipse(canvas, ellipse\_center, axes, angle, startAngle, endAngle, ellipse\_color, ellipse\_thickness)

# ------------------------

# 5️ Add Text

# ------------------------

# cv2.putText(image, text, org, font, fontScale, color, thickness, lineType)

text = "OpenCV Shapes"

org = (50, 400) # Bottom-left corner of text (x, y)

font = cv2.FONT\_HERSHEY\_SIMPLEX # Font type

font\_scale = 1 # Font scale (size)

text\_color = (255, 0, 255) # Magenta in BGR

text\_thickness = 2 # Thickness of text

cv2.putText(canvas, text, org, font, font\_scale, text\_color, text\_thickness)

# ------------------------

# Display Image

# ------------------------

plt.figure(figsize=(10,6))

plt.imshow(cv2.cvtColor(canvas, cv2.COLOR\_BGR2RGB))

plt.title("All Shapes and Text")

plt.axis("off")

plt.show()

# ------------------------

# Save Image

# ------------------------

cv2.imwrite("assets/image\_with\_shapes.jpg", canvas)

## Brainstorm/Experiments

1. Grayscale shape:
   * Converting to grayscale reduces the number of channels from 3 (RGB) → 1, but height & width stay the same.
2. **Brightness / Contrast:**
   * Play with alpha (contrast) and beta (brightness).
   * Example: alpha=0.5, beta=50 → washed out, alpha=1.5, beta=-50 → too dark.
   * Record your preferred values.
3. **HSV manipulation:**
   * Zero out **Saturation** channel:

img\_hsv = cv2.cvtColor(img\_rgb, cv2.COLOR\_RGB2HSV)

img\_hsv[:, :, 1] = 0

result = cv2.cvtColor(img\_hsv, cv2.COLOR\_HSV2RGB)

plt.imshow(result)

plt.title("Zero Saturation")

plt.axis("off")

plt.show()

# Chapter Two- Image Sculpting

**Image Sculpting** is essentially a creative and technical term for **enhancing, modifying, or manipulating** images to highlight features, improve quality, or extract useful information

## Learning Goals

1. Read and visualize images & histograms.
2. Apply **histogram equalization** (global & CLAHE).
3. Implement **2-D convolution from scratch** (with padding, stride=1).
4. Apply convolution for **blur, sharpen, edge detection**.
5. Compare with OpenCV optimized filters:
   * Gaussian, Median, Bilateral, filter2D.
6. Compare results and execution speed.

## Histogram Calculation & Visualization

## Histogram: cv2.calcHist()

hist = cv2.calcHist([image], [channel], mask, [num\_bins], [range\_start, range\_end])

**Parameters:**

* images → List of images as numpy arrays (same dtype & size).
* channels → List of channels to compute histogram (e.g., [0] for Blue in BGR).
* mask → Optional 8-bit array to focus on a region.
* histSize → Number of bins (e.g., [256]).
* ranges → Pixel value range (e.g., [0,256]).

## Color Image Histogram

colors = ['r', 'g', 'b']

plt.figure(figsize=(12, 4))

for i in range(3):

hist = cv2.calcHist([imgRGB], [i], None, [256], [0, 256])

plt.plot(hist, color=colors[i], label=f'{colors[i]} channel')

plt.title('Color Histogram')

plt.xlabel('Pixel Value')

plt.ylabel('Frequency')

plt.legend()

plt.grid()

plt.show()

## Matplotlib Histogram (All Pixels)

plt.figure(figsize=(12, 4))

plt.hist(imgRGB.ravel(), bins=256, range=(0, 255))

plt.title('Histogram')

plt.xlabel('Pixel Value')

plt.ylabel('Frequency')

plt.grid()

plt.show()

## Grayscale Histogram

img\_gray = cv2.cvtColor(img\_color, cv2.COLOR\_BGR2GRAY)

plt.figure(); plt.imshow(img\_gray, cmap='gray'); plt.axis('off')

hist\_counts, bin\_edges = np.histogram(img\_gray.ravel(), bins=256, range=(0, 255))

plt.figure()

plt.plot(bin\_edges[:-1], hist\_counts)

plt.title('Grayscale Histogram')

plt.xlabel('Pixel Value')

plt.ylabel('Frequency')

plt.grid()

plt.show()

## Side-by-Side Histogram Visualization

fig, axs = plt.subplots(2, 2, figsize=(16, 8))

# Original Color Image

axs[0, 0].imshow(cv2.cvtColor(img\_color, cv2.COLOR\_BGR2RGB))

axs[0, 0].set\_title('Original Color')

axs[0, 0].axis('off')

# Histogram of Color Image

for i in range(3):

hist = cv2.calcHist([imgRGB], [i], None, [256], [0, 256])

axs[0, 1].plot(hist, color=colors[i], label=f'{colors[i]} channel')

axs[0, 1].set\_title('Histogram (Color)')

# Grayscale Image

axs[1, 0].imshow(img\_gray, cmap='gray')

axs[1, 0].set\_title('Grayscale')

axs[1, 0].axis('off')

# Histogram of Grayscale

hist = cv2.calcHist([img\_gray], [0], None, [256], [0, 256])

axs[1, 1].plot(hist, color='gray')

axs[1, 1].set\_title('Histogram (Grayscale)')

plt.tight\_layout()

plt.show()

## Histogram Interpretation

The histogram shows the distribution of pixel intensities in an image.

* If most values are on the left, the image is **underexposed** (dark); if on the right, it is **overexposed** (bright).
* A **narrow histogram** indicates **low contrast**, while a **wide spread** across 0–255 indicates **high contrast**.
* Gaps or irregular spikes can suggest prior **quantization or heavy processing**.

## Global Histogram Equalization

**Definition:**  
Histogram equalization redistributes pixel intensities so that all values are more evenly spread across the available range.

**Benefits:**

* **Image enhancement** → Makes details in dark or bright regions more visible.
* **Contrast enhancement** → Improves visibility of features in low-contrast images.

**Example (Grayscale Image):**

import cv2

import matplotlib.pyplot as plt

bad\_img = cv2.imread('./Unequalized\_image.jpg', cv2.IMREAD\_GRAYSCALE)

assert bad\_img is not None, "Image not found."

# Apply global histogram equalization

equalized\_img = cv2.equalizeHist(bad\_img)

# Display

plt.figure(figsize=(12,5))

plt.subplot(1,2,1)

plt.imshow(bad\_img, cmap='gray')

plt.title('Original Image')

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(equalized\_img, cmap='gray')

plt.title('Equalized Image')

plt.axis('off')

plt.show()

## Global Histogram Equalization – Visualization

## Compute Histogram and CDF

import cv2

import matplotlib.pyplot as plt

import numpy as np

# Original grayscale image

bad\_img = cv2.imread('./Unequalized\_image.jpg', cv2.IMREAD\_GRAYSCALE)

hist = cv2.calcHist([bad\_img], [0], None, [256], [0, 256])

cdf = hist.cumsum() # Cumulative distribution function

cdfNorm = cdf \* hist.max() / cdf.max() # Normalize for plotting

## Apply Histogram Equalization

eq = cv2.equalizeHist(bad\_img)

# Histogram & CDF after equalization

hist\_eq = cv2.calcHist([eq], [0], None, [256], [0, 256])

cdf\_eq = hist\_eq.cumsum()

cdfNorm\_eq = cdf\_eq \* hist\_eq.max() / cdf\_eq.max()

#Plot Original vs Equalized

fig, axs = plt.subplots(2, 2, figsize=(10, 8))

# Original grayscale image

axs[0, 0].imshow(bad\_img, cmap='gray')

axs[0, 0].set\_title('Original Grayscale'); axs[0, 0].axis('off')

# Histogram + CDF (Original)

axs[0, 1].plot(cdfNorm, color='red')

axs[0, 1].plot(hist, color='gray')

axs[0, 1].set\_title('Histogram (Original)')

# Equalized grayscale image

axs[1, 0].imshow(eq, cmap='gray')

axs[1, 0].set\_title('Equalized Grayscale'); axs[1, 0].axis('off')

# Histogram + CDF (Equalized)

axs[1, 1].plot(cdfNorm\_eq, color='red')

axs[1, 1].plot(hist\_eq, color='gray')

axs[1, 1].set\_title('Histogram (Equalized)')

plt.tight\_layout()

plt.show()

## ****Key Notes****

* **CDF (Cumulative Distribution Function)** shows how pixel intensities accumulate.
* **Histogram Equalization** redistributes intensities so that CDF becomes more linear → enhances contrast.
* Original image: histogram may be skewed (under/overexposed).
* Equalized image: histogram spreads across full intensity range, improving visibility of details.

## Contrast Limited Adaptive Histogram Equalization (CLAHE)

**Definition:**  
CLAHE improves **local contrast** by applying histogram equalization on small tiles of the image while limiting over-amplification.

**Steps:**

1. Divide image into small tiles (e.g., 8×8).
2. Calculate histogram for each tile.
3. Clip bins above clipLimit → limits contrast locally.
4. Redistribute clipped excess uniformly.
5. Apply histogram equalization to each tile.
6. Interpolate tiles to form the final image.

**OpenCV Implementation:**

import cv2

clahe = cv2.createCLAHE(clipLimit=4.0, tileGridSize=(8,8))

clahe\_img = clahe.apply(bad\_img)

**Notes / Experiments:**

* Try clipLimit=2.0 → lower contrast enhancement, fewer artifacts.
* Try clipLimit=8.0 → higher local contrast, may introduce artifacts.
* Useful for **low-contrast or unevenly lit images**.

**Visualization (Image + Histogram + CDF):**

* Original grayscale vs. CLAHE image side by side.
* Histogram plotted with gray, CDF in red.

# Convolution

**Definition:**  
Convolution is a mathematical operation where a filter (kernel) slides over an image or signal to extract features or modify it.

**Uses:**

* Edge detection, blurring, and sharpening in images
* Feature extraction in computer vision and deep learning
* Signal processing and audio filtering

**Benefits:**

* Highlights important patterns or features
* Reduces noise while preserving details
* Forms the basis of CNNs for image recognitio

**Purpose:** Apply linear filters (blur, sharpen, edge detection) manually.

**Steps / Algorithm:**

import numpy as np

import cv2

import matplotlib.pyplot as plt

def convolution(image, kernel):

"""

Apply convolution on a grayscale image with a given kernel.

Parameters:

image : 2D numpy array (grayscale image)

kernel : 2D numpy array (square filter)

Returns:

output : Convolved image, same size as input

"""

# Step 1: Kernel & Padding

k = kernel.shape[0] # assume square kernel

pad = k // 2 # padding size

# Step 2: Pad the Image (reflect padding reduces border artifacts)

padded\_image = np.pad(image, ((pad, pad), (pad, pad)), mode='reflect')

# Step 3: Prepare Output Image

output = np.zeros\_like(image, dtype=np.float32)

# Step 4: Flip Kernel (horizontal + vertical) for true convolution

flipped\_kernel = np.flipud(np.fliplr(kernel)).astype(np.float32)

# Step 5: Apply Convolution

for i in range(image.shape[0]):

for j in range(image.shape[1]):

region = padded\_image[i:i+k, j:j+k]

output[i, j] = np.sum(region \* flipped\_kernel)

# Step 6: Clip and Convert to uint8

output = np.clip(output, 0, 255).astype(np.uint8)

# Step 7: Return Output

return output

# ------------------ Example Usage ------------------

# Load a grayscale image

img\_path = "image.jpg" # replace with your image path

img = cv2.imread(img\_path, cv2.IMREAD\_GRAYSCALE)

# Define a sample kernel (e.g., sharpening kernel)

kernel = np.array([[0, -1, 0],

[-1, 5, -1],

[0, -1, 0]])

# Apply convolution

conv\_img = convolution(img, kernel)

# Show original and convolved image

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img, cmap='gray')

plt.title("Original Image")

plt.axis("off")

plt.subplot(1,2,2)

plt.imshow(conv\_img, cmap='gray')

plt.title("After Convolution")

plt.axis("off")

plt.show()

|  |  |
| --- | --- |
| **Parameter** | **Effect / Explanation** |
| **Image** | The input data to process; affects the output since convolution depends on pixel values. |
| **Kernel / Filter** | Determines the type of operation (e.g., edge detection, blur, sharpen); its values define what features are highlighted. |
| **Padding** | Adds extra pixels around the border; ensures output size equals input size and reduces border artifacts. |
| **Stride** | Number of pixels the kernel moves each step; larger strides reduce output size and skip details. |
| **Kernel Flip** | Flipping horizontally and vertically gives true convolution; skipping it performs cross-correlation instead. |
| **Data Type / Clipping** | Output type (e.g., float32, uint8) and clipping control how values are stored and prevent overflow/underflow. |

|  |  |
| --- | --- |
| **Kernel Size** | **Effect / Explanation** |
| **Small kernel (e.g., 3x3)** | Captures fine details and edges; output is sensitive to small features. |
| **Medium kernel (e.g., 5x5, 7x7)** | Captures larger patterns; smooths small noise but still preserves some detail. |
| **Large kernel (e.g., 11x11, 15x15)** | Captures broad patterns; strong smoothing/blurring; fine details are lost. |

**Additional notes:**

* Larger kernels require **more computation**.
* Small kernels are often **stacked** in deep learning to emulate a larger receptive field while keeping fewer parameters.
* Kernel size also affects **padding**: for output same size as input, padding = kernel\_size // 2.

## Image Filtering with 2‑D Convolution: Averaging, Sharpening, Sobel

### ****Averaging (Blurring)****

**Kernel:** 3×3 box filter

box3 = np.ones((3,3), np.float32)/9.0

box3 = np.array([[0.111, 0.111, 0.111],

[0.111, 0.111, 0.111],

[0.111, 0.111, 0.111]], dtype=np.float32)

|  |  |
| --- | --- |
| **Aspect** | **Explanation** |
| **What is it?** | A simple smoothing technique where each pixel is replaced by the **average of its neighbors** using a box filter (e.g., 3×3 kernel). |
| **Parameters** | - **Kernel size** (e.g., 3×3) defines the neighborhood to average. - **Kernel values** (all equal, summing to 1) determine contribution of each neighbor. - **Padding / stride** can affect output size and edges. |
| **Purpose** | To **reduce noise** or minor variations in an image while making it smoother. |
| **Impact / Effect** | - Reduces small random noise. - Blurs sharp edges and fine details. - Larger kernels → stronger smoothing, more blurring. |

## Manually: Convolution Image with Box3 filter:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load grayscale image

img = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Define 3x3 averaging kernel

box3 = np.ones((3,3), np.float32)/9.0

# Padding

pad = box3.shape[0] // 2

padded\_img = np.pad(img, ((pad, pad), (pad, pad)), mode='reflect')

# Output image

output = np.zeros\_like(img, dtype=np.float32)

# Apply convolution

for i in range(img.shape[0]):

for j in range(img.shape[1]):

region = padded\_img[i:i+3, j:j+3]

output[i,j] = np.sum(region \* box3)

# Clip and convert

output = np.clip(output, 0, 255).astype(np.uint8)

# Show result

plt.imshow(output, cmap='gray')

plt.title("Manual Averaging")

plt.axis('off')

plt.show()

## Using OpenCV (cv2.filter2D)

# Apply averaging using OpenCV

import cv2

import numpy as np

# Load grayscale image

img = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Define 3x3 averaging kernel

box3 = np.ones((3,3), np.float32) / 9.0) # <-- define kernel first

# Apply convolution using OpenCV

blur\_img = cv2.filter2D(img, ddepth=-1, kernel=box3)

# Show result

cv2.imshow("Averaging", blur\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Key points:**

* kernel must be a **NumPy array** of type float32 or float64.
* Size of the kernel determines the neighborhood over which averaging occurs.
* ddepth=-1 ensures the output has the **same data type as input**.

## Manually: Convolution Image Sharpen Filter :

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load grayscale image

img = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Define 3x3 averaging kernel

sharpen = np.array([[0,-1,0],

[-1,5,-1],

[0,-1,0]], np.float32)

# Padding

pad = box3.shape[0] // 2

padded\_img = np.pad(img, ((pad, pad), (pad, pad)), mode='reflect')

# Output image

# Apply sharpening using OpenCV

sharpen\_img = cv2.filter2D(img, ddepth=-1, kernel=sharpen)

# Show result

plt.imshow(sharpen\_img, cmap='gray')

plt.title("OpenCV Sharpening")

plt.axis('off')

plt.show()

output = np.zeros\_like(img, dtype=np.float32)

# Apply convolution

for i in range(img.shape[0]):

for j in range(img.shape[1]):

region = padded\_img[i:i+3, j:j+3]

output[i,j] = np.sum(region \* box3)

# Clip and convert

output = np.clip(output, 0, 255).astype(np.uint8)

# Show result

plt.imshow(output, cmap='gray')

plt.title("Manual Averaging")

plt.axis('off')

plt.show()

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Box Filter (Averaging)** | **Sharpening Filter** |
| **Purpose** | Smooths the image, reduces noise | Enhances edges and fine details |
| **Kernel Values** | All elements equal, sum = 1 (e.g., 3×3 with 1/9) | Center positive, neighbors negative (e.g., [[0,-1,0],[-1,5,-1],[0,-1,0]]) |
| **Effect on Image** | Blurs edges, smooths textures | Makes edges and textures more prominent, may amplify noise |
| **Operation** | Replaces each pixel with the **average of neighbors** | Replaces each pixel with **center pixel minus neighbors** (emphasizes differences) |
| **Use Cases** | Noise reduction, preprocessing | Enhancing details, sharpening blurry images |

**Summary:**

* **Box filter = smoothing** → reduces variation, edges get blurred.
* **Sharpening filter = high-pass** → emphasizes variation, edges stand out.

## Manual Convolution : Sobel Kernel

The **Sobel filter** is an edge detection technique that highlights areas in an image where intensity changes sharply, indicating edges. It uses specific kernels to compute gradients in the **x (vertical edges)** and **y (horizontal edges)** directions. By combining these gradients, it produces a map of strong edges, which is useful for feature extraction, object detection, and image analysis.

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load grayscale image

img = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Define vertical and horizontal Sobel kernels

sobel\_vertical = np.array([[-1,0,1],

[-2,0,2],

[-1,0,1]], np.float32)

sobel\_horizontal = np.array([[-1,-2,-1],

[0, 0, 0],

[1, 2, 1]], np.float32)

# Padding

k = sobel\_vertical.shape[0]

pad = k // 2

padded\_img = np.pad(img, ((pad,pad),(pad,pad)), mode='reflect')

# Output images

output\_vertical = np.zeros\_like(img, dtype=np.float32)

output\_horizontal = np.zeros\_like(img, dtype=np.float32)

# Apply vertical Sobel

for i in range(img.shape[0]):

for j in range(img.shape[1]):

region = padded\_img[i:i+k, j:j+k]

output\_vertical[i,j] = np.sum(region \* sobel\_vertical)

# Apply horizontal Sobel

for i in range(img.shape[0]):

for j in range(img.shape[1]):

region = padded\_img[i:i+k, j:j+k]

output\_horizontal[i,j] = np.sum(region \* sobel\_horizontal)

# Combine edges (gradient magnitude)

combined = np.sqrt(output\_vertical\*\*2 + output\_horizontal\*\*2)

# Clip and convert to uint8

output\_vertical = np.clip(output\_vertical, 0, 255).astype(np.uint8)

output\_horizontal = np.clip(output\_horizontal, 0, 255).astype(np.uint8)

combined = np.clip(combined, 0, 255).astype(np.uint8)

# Show original + Sobel results

plt.figure(figsize=(16,4))

plt.subplot(1,4,1)

plt.imshow(img, cmap='gray')

plt.title("Original Image")

plt.axis('off')

plt.subplot(1,4,2)

plt.imshow(output\_vertical, cmap='gray')

plt.title("Vertical Sobel")

plt.axis('off')

plt.subplot(1,4,3)

plt.imshow(output\_horizontal, cmap='gray')

plt.title("Horizontal Sobel")

plt.axis('off')

plt.subplot(1,4,4)

plt.imshow(combined, cmap='gray')

plt.title("Combined Sobel")

plt.axis('off')

plt.show()

## Sobel Filter: Using OpenCV

import cv2

import matplotlib.pyplot as plt

# Load grayscale image

img = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Vertical Sobel edges (dx=1, dy=0)

sobel\_vertical = cv2.Sobel(img, ddepth=cv2.CV\_64F, dx=1, dy=0, ksize=3)

sobel\_vertical = cv2.convertScaleAbs(sobel\_vertical)

# Horizontal Sobel edges (dx=0, dy=1)

sobel\_horizontal = cv2.Sobel(img, ddepth=cv2.CV\_64F, dx=0, dy=1, ksize=3)

sobel\_horizontal = cv2.convertScaleAbs(sobel\_horizontal)

# Combine both edges

sobel\_combined = cv2.addWeighted(sobel\_vertical, 0.5, sobel\_horizontal, 0.5, 0)

**Weights assigned to each image** when combining them.

* 0.5 means **both vertical and horizontal edges contribute equally**.
* You can change the weights if you want to **emphasize one direction** more than the other.

The weights in cv2.addWeighted() determine how much each image contributes to the combined result. Adjusting them allows you to **emphasize one image or direction more than the other** in the final output.

If you change the weights to 0.7 for vertical and 0.3 for horizontal, the vertical edges will appear stronger, while horizontal edges will be less pronounced.

# Show results

plt.figure(figsize=(16,4))

plt.subplot(1,4,1)

plt.imshow(img, cmap='gray')

plt.title("Original Image")

plt.axis('off')

plt.subplot(1,4,2)

plt.imshow(sobel\_vertical, cmap='gray')

plt.title("Vertical Sobel")

plt.axis('off')

plt.subplot(1,4,3)

plt.imshow(sobel\_horizontal, cmap='gray')

plt.title("Horizontal Sobel")

plt.axis('off')

plt.subplot(1,4,4)

plt.imshow(sobel\_combined, cmap='gray')

plt.title("Combined Sobel")

plt.axis('off')

plt.show()

**✅ Notes:**

* cv2.Sobel() automatically handles the convolution.
* dx and dy control the derivative direction (vertical vs horizontal edges).
* cv2.addWeighted() combines both directions to get the full edge map.
* This is **much faster** and simpler than manual convolution.

## Apply 2‑D Convolution: Defining Custom Function: conv2d

blurred = conv2d(img\_gray, box3) # Apply averaging

sharpened = conv2d(img\_gray, sharpen) # Apply sharpening

edges\_x = conv2d(img\_gray, sobel\_vertical) # Apply Sobel X

import cv2

import numpy as np

import matplotlib.pyplot as plt

# ----------------- Define conv2d function -----------------

def conv2d(image, kernel):

k = kernel.shape[0]

pad = k // 2

# Pad the image to handle borders

padded\_img = np.pad(image, ((pad,pad),(pad,pad)), mode='reflect')

output = np.zeros\_like(image, dtype=np.float32)

# Apply convolution

for i in range(image.shape[0]):

for j in range(image.shape[1]):

region = padded\_img[i:i+k, j:j+k]

output[i,j] = np.sum(region \* kernel)

# Clip to 0-255 and convert to uint8

output = np.clip(output, 0, 255).astype(np.uint8)

return output

# ----------------- Load grayscale image -----------------

img\_gray = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# ----------------- Define kernels -----------------

box3 = np.ones((3,3), np.float32)/9.0 # Averaging

sharpen = np.array([[0,-1,0],

[-1,5,-1],

[0,-1,0]], np.float32) # Sharpening

sobel\_vertical = np.array([[-1,0,1],

[-2,0,2],

[-1,0,1]], np.float32) # Sobel X (vertical)

# ----------------- Apply 2D convolution -----------------

blurred = conv2d(img\_gray, box3)

sharpened = conv2d(img\_gray, sharpen)

edges\_x = conv2d(img\_gray, sobel\_vertical)

# ----------------- Display results -----------------

plt.figure(figsize=(12,4))

plt.subplot(1,4,1)

plt.imshow(img\_gray, cmap='gray')

plt.title("Original")

plt.axis('off')

plt.subplot(1,4,2)

plt.imshow(blurred, cmap='gray')

plt.title("Blurred (Box Filter)")

plt.axis('off')

plt.subplot(1,4,3)

plt.imshow(sharpened, cmap='gray')

plt.title("Sharpened")

plt.axis('off')

plt.subplot(1,4,4)

plt.imshow(edges\_x, cmap='gray')

plt.title("Sobel X (Vertical)")

plt.axis('off')

plt.show()

### ****Key Notes****

1. **Blurring** → reduces noise, smooths image, edges become softer.
2. **Sharpening** → emphasizes edges and details.
3. **Sobel Filter** → highlights edges, useful for edge detection and feature extraction.
4. Always use **padding** so output size = input size.
5. These kernels can be **combined** for creative effects or preprocessing for further tasks.

## Kernel Types and Effects

1. **Low-pass (Averaging / Box Filter)**

* **Purpose:** Smooths image by averaging neighboring pixels.
* **Effect:** Removes high-frequency content → **blurs edges and noise**.
* **Visual Response:** Edges appear soft; minimal edge response.

1. **High-boost (Sharpening Kernel)**

* **Purpose:** Enhances the center pixel while subtracting neighbors.
* **Effect:** Boosts high-frequency content → **sharpens edges and fine details**.
* **Visual Response:** Edges and textured areas become prominent.

1. **Derivative (Sobel / Edge Detection)**

* **Purpose:** Measures **rate of change** in intensity along a direction (X or Y).
* **Effect:** Highlights boundaries and edges.
* **Visual Response:** Edges show maximum response; uniform regions → near zero.

**Tip:**

* **Low-pass → removes edges**
* **High-boost → enhances edges**
* **Derivative → detects edges**

## Filtering with OpenCV: cv2.filter2D

filtering refers to the process of modifying or analyzing an image by applying a mathematical operation (kernel) to each pixel and its neighbors. Filters can be used to enhance features, detect edges, reduce noise, or extract patterns, depending on the type of kernel applied.

### ****filter2D vs Custom Convolution****

Syntax: dst = cv2.filter2D(src, ddepth, kernel, anchor=None, delta=0, borderType=cv2.BORDER\_DEFAULT)

|  |  |
| --- | --- |
| **Parameter** | **Meaning** |
| src | Input image (grayscale or color) |
| ddepth | Desired depth of output image (e.g., -1 = same as input) |
| kernel | Convolution kernel (e.g., 3×3 or 5×5 NumPy array) |
| anchor | Position of kernel’s anchor point (default: center) |
| delta | Value added to each pixel after convolution (default 0) |
| borderType | Pixel extrapolation method at borders (default: cv2.BORDER\_DEFAULT) |

import cv2

import numpy as np

# Load grayscale image

img\_gray = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Define 3x3 averaging filter

kernel = np.ones((3,3), np.float32)/9

# Apply filter2D with reflection padding

cv\_blurred = cv2.filter2D(img\_gray, -1, kernel, borderType=cv2.BORDER\_REFLECT)

# If you have a manual blurred image called 'blurred'

# Compute absolute difference

diff = cv2.absdiff(blurred, cv\_blurred)

# Print mean absolute difference

print('Mean abs diff:', diff.mean())

✅ **Notes:** Fast, optimized, handles borders automatically.

## Custom 2D Convolution

Syntax: You define a function like this:

def conv2d(image, kernel):

k = kernel.shape[0]

pad = k // 2

padded\_img = np.pad(image, ((pad,pad),(pad,pad)), mode='reflect')

output = np.zeros\_like(image, dtype=np.float32)

for i in range(image.shape[0]):

for j in range(image.shape[1]):

region = padded\_img[i:i+k, j:j+k]

output[i,j] = np.sum(region \* kernel)

return np.clip(output, 0, 255).astype(np.uint8)

blurred\_manual = conv2d(img, kernel) # calling the function

This is an **explanation of why results from cv2.filter2D may slightly differ from a manual convolution (conv2d)**, even when using the same kernel.

* **Border handling:** filter2D automatically manages image borders (e.g., BORDER\_REFLECT), while manual convolution may pad differently.
* **Kernel flip:** In true convolution, the kernel is flipped horizontally and vertically; filter2D performs correlation by default (no flip).
* **Rounding/precision:** filter2D may use different data types or rounding methods (float vs uint8), causing small numerical differences.

✅ **In short:** Minor differences are normal because **implementation details like padding, kernel orientation, and numeric precision differ**.

# Image Blurring Techniques: Mean Blurring

|  |  |  |  |
| --- | --- | --- | --- |
| Filter / Kernel | Purpose | Method | Effect |
| Box / Averaging | Smooth / reduce noise | Mean of neighborhood | Blurs edges |
| Median | Remove salt-pepper noise | Median of neighborhood | Smooths, preserves edges |
| Gaussian | Smooth naturally | Weighted mean (Gaussian) | Soft blur, edges slightly preserved |
| Sobel | Edge detection | Gradient along x or y | Highlights edges |
| Sharpen | Enhance edges/details | Center weighted - neighbors | Makes details more visible |

**💡 Key Idea:**

* **Blurring filters (average, median, Gaussian)** → reduce noise, smooth image
* **Edge/Detail filters (Sobel, Sharpen)** → detect or enhance edges
* **The difference is in what each kernel emphasizes**

What is Mean Blurring?

* **Definition:** Mean blurring smooths an image by replacing each pixel with the **average value of its surrounding neighbors** (defined by a kernel of size k×k).
* **Purpose / Uses:**
  + Reduce random noise in images
  + Smooth textures
  + Preprocess images before edge detection or other analysis
* **Effect:** Edges and fine details become **less sharp**, image looks softer.

## Manual Implementation of Mean Blurring

import cv2

import numpy as np

def mean\_blur\_manual(img, k):

pad = k // 2

padded\_img = np.pad(img, ((pad,pad),(pad,pad)), mode='reflect')

output = np.zeros\_like(img, dtype=np.float32)

kernel = np.ones((k,k), np.float32) / (k\*k)) # normalized box filter

for i in range(img.shape[0]):

for j in range(img.shape[1]):

region = padded\_img[i:i+k, j:j+k]

output[i,j] = np.sum(region \* kernel)

return np.clip(output, 0, 255).astype(np.uint8)

# Load grayscale image

img\_gray = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

blur\_manual = mean\_blur\_manual(img\_gray, k=3)

## OpenCV Implementation of Mean Blurring

import cv2

# Load grayscale image

img\_gray = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Apply mean blurring (box filter)

kernel\_size = 3

blur = cv2.blur(img\_gray, (kernel\_size, kernel\_size))

**Uses / Applications**

* **Noise reduction:** Smooths random noise in images (e.g., camera images).
* **Preprocessing before edge detection:** Reduces small variations so filters like Sobel or Canny detect meaningful edges.
* **Reducing fine texture details:** Removes minor textures or patterns to focus on larger structures.

**Impact of kernel size (k):** Small k → mild smoothing, edges and fine details mostly preserved; large k → strong smoothing, noise and fine details reduced, edges blurred.

## Median Blurring: cv2.medianBlur(img\_gray, kernel\_size)

Central pixel replaced by **median** of neighborhood pixels. Effective for **salt-and-pepper noise**. Kernel size must be **odd**.

Manual Implementation of Median Blur

import cv2

import numpy as np

def median\_blur\_manual(img, k):

"""

Apply median blurring manually.

Parameters:

img: Grayscale image (numpy array)

k: Kernel size (must be odd)

"""

pad = k // 2

# Reflect padding to handle borders

padded\_img = np.pad(img, ((pad, pad), (pad, pad)), mode='reflect')

output = np.zeros\_like(img, dtype=np.uint8)

# Iterate over each pixel

for i in range(img.shape[0]):

for j in range(img.shape[1]):

# Extract kxk neighborhood

region = padded\_img[i:i+k, j:j+k]

# Replace central pixel with median of neighborhood

output[i, j] = np.median(region)

return output

# Load grayscale image

img\_gray = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Apply manual median blur

kernel\_size = 3

blur\_median\_manual = median\_blur\_manual(img\_gray, kernel\_size)

Opencv Implementation of Median Blur

import cv2

# Load grayscale image

img\_gray = cv2.imread("image.jpg", cv2.IMREAD\_GRAYSCALE)

# Apply median blur

kernel\_size = 3 # Must be odd

blur\_median = cv2.medianBlur(img\_gray, kernel\_size)

**Notes:**

* kernel\_size must be **odd and greater than 1**.
* Replaces each pixel with the **median of its neighborhood**, effectively removing salt-and-pepper noise while **preserving edges** better than mean/average blurring.

## Gaussian Blurring: cv2.GaussianBlur(img, (Kernel\_size, Kernel\_size), sigmaX=sigmaX, sigmaY=sigmaY)

## Manually Implementation:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def gaussian\_kernel(size, sigma=1):

"""Generate a Gaussian kernel."""

ax = np.linspace(-(size // 2), size // 2, size)

xx, yy = np.meshgrid(ax, ax)

kernel = np.exp(-(xx\*\*2 + yy\*\*2) / (2 \* sigma\*\*2))

return kernel / np.sum(kernel) # normalize

def apply\_convolution(image, kernel):

"""Apply convolution manually (without cv2.filter2D)."""

# Get dimensions

img\_h, img\_w, channels = image.shape

k\_h, k\_w = kernel.shape

pad\_h, pad\_w = k\_h // 2, k\_w // 2

# Pad the image

padded = np.pad(image, ((pad\_h, pad\_h), (pad\_w, pad\_w), (0, 0)), mode='constant')

output = np.zeros\_like(image, dtype=np.float32)

# Convolution loop

for y in range(img\_h):

for x in range(img\_w):

for c in range(channels):

region = padded[y:y+k\_h, x:x+k\_w, c]

output[y, x, c] = np.sum(region \* kernel)

return np.clip(output, 0, 255).astype(np.uint8)

# Read image

img = cv2.imread("your\_image.jpg")

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

# Create Gaussian kernel (size=9, sigma=50)

kernel = gaussian\_kernel(size=9, sigma=50)

# Apply convolution

blurred = apply\_convolution(img, kernel)

# Show results

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img)

plt.title("Original")

plt.axis("off")

plt.subplot(1,2,2)

plt.imshow(blurred)

plt.title("Manual Gaussian Blur (k=9, σ=50)")

plt.axis("off")

plt.show()

## OpenCv Implementation of Gaussian Blur

import cv2

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# Gaussian Blur parameters

Kernel\_size = 9

sigmaX = 50

sigmaY = 50

# Apply Gaussian Blur

gaussian = cv2.GaussianBlur(img, (Kernel\_size, Kernel\_size), sigmaX=sigmaX, sigmaY=sigmaY)

# Show original and blurred images

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img)

plt.title("Original")

plt.axis("off")

plt.subplot(1,2,2)

plt.imshow(gaussian)

plt.title(f"Gaussian Blur k={Kernel\_size}, sigmaX={sigmaX}, sigmaY={sigmaY}")

plt.axis("off")

plt.tight\_layout()

plt.show()

**Kernel Size (**K**)**

* The kernel is the **window of pixels** used to compute the blur.
* Larger K → more pixels considered → **heavier blur**.
* Smaller K → fewer pixels → **lighter blur**.
* Must be **odd and positive** (3, 5, 7, …).

**SigmaX (**σx**)**

* Controls the **spread of the Gaussian function** along the x-axis (width of the bell curve).
* Larger sigmaX → **wider Gaussian** → pixels farther from the center contribute more → **heavier blur**.
* Smaller sigmaX → narrow Gaussian → mostly the central pixel dominates → **lighter blur**.

## Different Values of Kernel and Sigma in Gaussian Filter

import cv2

import matplotlib.pyplot as plt

img = cv2.imread("your\_image.jpg")

# Convert BGR to RGB

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

# Lists of parameters to test

kernel\_sizes = [5, 11, 15]

sigma\_values = [50, 100, 1500]

plt.figure(figsize=(15, 10))

plot\_num = 1

for k in kernel\_sizes:

for sigma in sigma\_values:

# Apply Gaussian blur

blurred = cv2.GaussianBlur(img, (k, k), sigmaX=sigma, sigmaY=sigma)

# Plot

ax = plt.subplot(len(kernel\_sizes), len(sigma\_values), plot\_num)

ax.imshow(blurred)

ax.set\_title(f'K={k}, σ={sigma}')

ax.axis('off')

plot\_num += 1

plt.tight\_layout()

plt.show()

## Reports of Different Size of Kernel and Sigma Value of Gaussian Filter

**How to Write the Answer**

1. **Introduction**

Explain what you are testing:

In this experiment, Gaussian blur was applied to the image of Geoffrey Hinton with different kernel sizes (K = 5, 11, 15) and sigma values (σ = 50, 100, 1500). The purpose was to analyze how kernel size and sigma influence the amount of blurring.

2. Observations

Go row by row or parameter by parameter:

* **Effect of Kernel Size (K):**
  + With **K = 5**, the blur is mild; details and edges are still visible.
  + With **K = 11**, the blur becomes stronger, smoothing textures and making facial details softer.
  + With **K = 15**, the image looks heavily blurred; fine details almost vanish.
* **Effect of Sigma (σ):**
  + With **σ = 50**, the blur is noticeable but edges remain partially sharp.
  + With **σ = 100**, the blur is stronger, edges and textures soften more.
  + With **σ = 1500**, the image is almost flat, losing nearly all detail — approaching a uniform blur.

3. Findings / Conclusion

Summarize the main takeaway:

The kernel size determines the **neighborhood region** over which blurring occurs, while sigma controls the **spread of the Gaussian function**. A larger kernel size or higher sigma value produces stronger blur. However, extremely large sigma values (e.g., 1500) cause the image to lose almost all detail, resulting in an unrealistic flattened effect.

**Gaussian Blur Experiments**

**1. Effect of Kernel Size**

* Larger kernel → more neighboring pixels averaged → stronger blur.
* Example: kernel sizes [3, 5, 7, 9] on same σ=3:

for k in [3,5,7,9]:

gaussian = cv2.GaussianBlur(img\_gray, (k,k), 3)

**Observation:**

* k=3: subtle blur, edges mostly preserved.
* k=9: strong blur, edges softened.

## Applying: 'Averaging', 'Gaussian', 'Median': Kernel Size 5, SigmaX = 0

import cv2 as cv

import matplotlib.pyplot as plt

# Read image

img = cv.imread("example.jpg")

# Convert to grayscale

img\_rgb = cv.cvtColor(img, cv.COLOR\_BGR2RGB)

# Apply blurring techniques

kernel\_size = 5

# Averaging (simple blur or mean blur)

blur = cv.blur(img\_rgb, (kernel\_size, kernel\_size))

# Gaussian blur

gaussian = cv.GaussianBlur(img\_rgb, (kernel\_size, kernel\_size), sigmaX=0)

# Median blur

median = cv.medianBlur(img\_rgb, kernel\_size)

# Visualization

titles = ['Original', 'Averaging', 'Gaussian', 'Median']

images = [img\_gray, blur, gaussian, median]

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

for i, (title, im) in enumerate(zip(titles, images), start=1):

ax = plt.subplot(2,2,i)

ax.imshow(im, cmap='gray', vmin=0, vmax=255)

ax.set\_title(title)

ax.axis('off')

plt.tight\_layout()

plt.show()

**Findings**

* **Original Image:** The original RGB image preserves full detail, colors, and edges without any smoothing.
* **Averaging Blur:** The image appears smoother, but edges and fine details are significantly blurred. It produces a uniform blur but may look unnatural.
* **Gaussian Blur:** The blurring effect is smoother and more natural compared to averaging. Edges are better preserved, making it suitable for preprocessing tasks like noise reduction.
* **Median Blur:** This filter reduces noise effectively while maintaining sharper edges. It is particularly effective for removing salt-and-pepper type noise without overly distorting important structures.

## Different Kernel Size Effects on Averaging / Median / Gaussian

* **Averaging / Median / Gaussian**: larger kernel → stronger smoothing.
* Visual check: edges become softer, noise reduces, but fine details may be lost.

kernel\_sizes = [3, 5, 7, 9]

# Loop to visualize effect of different kernel sizes

import cv2 as cv

import matplotlib.pyplot as plt

# Read and convert image to RGB

img = cv.imread("example.jpg")

img\_rgb = cv.cvtColor(img, cv.COLOR\_BGR2RGB)

kernel\_sizes = [3, 5, 7, 9]

methods = ['Averaging', 'Gaussian', 'Median']

plt.figure(figsize=(12, 9))

for i, k in enumerate(kernel\_sizes, start=1):

# Averaging blur

blur\_avg = cv.blur(img\_rgb, (k, k))

ax = plt.subplot(len(kernel\_sizes), len(methods), (i-1)\*len(methods) + 1)

ax.imshow(blur\_avg, cmap='gray', vmin=0, vmax=255)

ax.set\_title(f'{methods[0]} k={k}')

ax.axis('off')

# Gaussian blur

blur\_gauss = cv.GaussianBlur(img\_rgb, (k, k), sigmaX=0)

ax = plt.subplot(len(kernel\_sizes), len(methods), (i-1)\*len(methods) + 2)

ax.imshow(blur\_gauss, cmap='gray', vmin=0, vmax=255)

ax.set\_title(f'{methods[1]} k={k}')

ax.axis('off')

# Median blur

blur\_median = cv.medianBlur(img\_rgb, k)

ax = plt.subplot(len(kernel\_sizes), len(methods), (i-1)\*len(methods) + 3)

ax.imshow(blur\_median, cmap='gray', vmin=0, vmax=255)

ax.set\_title(f'{methods[2]} k={k}')

ax.axis('off')

plt.tight\_layout()

plt.show()

## ****Key Takeaways****

1. **Averaging** → smooths image, reduces noise, loses detail.
2. **Median** → robust to salt-and-pepper noise.
3. **Gaussian** → weighted smoothing, parameterizable by σ.
4. Use **larger kernels** for stronger blur; **small kernels** for subtle smoothing.
5. filter2D allows custom kernels efficiently with proper border handling.

## Bilateral Filter: cv2.bilateralFilter(img, d=d, sigmaColor=sigma, sigmaSpace=sigma)

The **Bilateral Filter** is a non-linear, edge-preserving, and noise-reducing filter. Unlike Gaussian or median blur, it smooths flat regions while keeping edges sharp by combining **spatial closeness** and **intensity similarity**.

**Key Points about OpenCV Bilateral Filter:**

* d: Diameter of the neighborhood (odd integer). Larger → more blur.
* sigmaColor: Strength of smoothing in color space. Larger → more color smoothing.
* sigmaSpace: Strength of smoothing in coordinate space. Larger → more spatial smoothing.
* **Edge-preserving:** Unlike Gaussian or averaging blur, it smooths flat regions but **keeps edges sharp**.
* Works for both **grayscale** and **color images**.

## Manual Implementation of Bilateral Filter

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

def bilateral\_filter\_manual(img, d=5, sigma\_color=12, sigma\_space=16):

rows, cols, ch = img.shape

padded\_img = cv2.copyMakeBorder(img, d, d, d, d, cv2.BORDER\_REFLECT)

filtered = np.zeros\_like(img, dtype=np.float32)

# Precompute spatial Gaussian (same for all pixels)

x, y = np.meshgrid(np.arange(-d, d+1), np.arange(-d, d+1))

spatial\_gauss = np.exp(-(x\*\*2 + y\*\*2) / (2 \* sigma\_space\*\*2))

for i in range(rows):

for j in range(cols):

for c in range(ch): # Process each channel

# Extract local region

region = padded\_img[i:i+2\*d+1, j:j+2\*d+1, c]

# Range Gaussian

intensity\_diff = region - img[i, j, c]

range\_gauss = np.exp(-(intensity\_diff\*\*2) / (2 \* sigma\_color\*\*2))

# Combined bilateral kernel

kernel = spatial\_gauss \* range\_gauss

kernel /= np.sum(kernel)

# Apply filter

filtered[i, j, c] = np.sum(kernel \* region)

return np.uint8(filtered)

# Manual bilateral filter

manual\_filtered = bilateral\_filter\_manual(img, d=3, sigma\_color=20, sigma\_space=30)

# Show results

plt.figure(figsize=(15,5))

plt.subplot(1,2,1)

plt.imshow(img)

plt.title("Original")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(manual\_filtered)

plt.title("Manual Bilateral")

plt.axis('off')

plt.tight\_layout()

plt.show()

## Opencv Implementation Bilateral Filter

import cv2

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# Apply OpenCV Bilateral Filter

# d = diameter of pixel neighborhood

# sigmaColor = filter sigma in color space

# sigmaSpace = filter sigma in coordinate space

bilateral\_filtered = cv2.bilateralFilter(img, d=9, sigmaColor=75, sigmaSpace=75)

# Display original and filtered images

plt.figure(figsize=(15,5))

plt.subplot(1,2,1)

plt.imshow(img)

plt.title("Original Image")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(bilateral\_filtered)

plt.title("OpenCV Bilateral Filtered")

plt.axis('off')

plt.tight\_layout()

plt.show()

## Different Values of d\_values and Sigma values Bilateral Filter

import cv2

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# Parameters to test

d\_values = [5, 15, 25]

sigma\_values = [25, 75, 150]

plt.figure(figsize=(15, 12))

plot\_num = 1

for d in d\_values:

for sigma in sigma\_values:

# Apply Bilateral Filter

filtered = cv2.bilateralFilter(img, d=d, sigmaColor=sigma, sigmaSpace=sigma)

# Plot

ax = plt.subplot(len(d\_values), len(sigma\_values), plot\_num)

ax.imshow(filtered)

ax.set\_title(f"d={d}, σColor=σSpace={sigma}")

ax.axis('off')

plot\_num += 1

plt.tight\_layout()

plt.show()

Parameter:

**SigmaColor:**

* Controls how much the filter **considers differences in pixel color**.
* **Small sigmaColor (e.g., 10–20):** Only very similar colors are averaged → edges are preserved tightly, little blur.
* **Medium sigmaColor (e.g., 50–75):** More colors are considered similar → stronger smoothing within regions.
* **Large sigmaColor (e.g., 150+):** Even very different colors are averaged → edges start to get blurred.

**Effect:** Higher sigmaColor → stronger color smoothing, but can reduce edge sharpness if too high.

**SigmaSpace:**

* Controls the **spatial extent** of the filter (how far neighbors are considered).
* **Small sigmaSpace (e.g., 5–10):** Only nearby pixels are considered → very local smoothing.
* **Medium sigmaSpace (e.g., 50–75):** Pixels farther away also contribute → stronger blur over larger areas.
* **Large sigmaSpace (e.g., 150+):** Very broad smoothing → flat regions are very smooth, edges may get softened.

**Effect:** Higher sigmaSpace → more global smoothing across spatially distant pixels.

**d value:**

* The kernel size (neighborhood around each pixel).
* **Small d (e.g., 3–5):** Less computation, smoother effect only locally.
* **Medium d (e.g., 9–15):** Larger neighborhood → more smoothing.
* **Large d (e.g., 25+):** Very large neighborhood → almost uniform blur in flat regions.

**Effect:** Increasing d increases the area over which the filter operates, but **sigmaColor and sigmaSpace still control edge preservation**.

**Combined Effect**

* **Small d, small sigmaColor, small sigmaSpace:** Very mild, local blur → edges almost untouched.
* **Large d, medium sigmaColor, medium sigmaSpace:** Strong smoothing within regions, edges mostly preserved.
* **Large sigmaColor or sigmaSpace too high:** Even edges may blur → image may look washed out.

--------------------------------------------------------------------------------------------------------------------------------

## Salt & Pepper Noise Filtering: Using Median, Guassian, Bilateral

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# -------------------------

# Add Salt & Pepper Noise

# -------------------------

prob = 0.2 # 20% pixels noisy and change this according to question

noisy\_img = np.copy(img)

# Generate random numbers for each pixel

salt\_pepper\_noise = np.random.rand(\*img.shape[:2])

# Add Salt (white)

noisy\_img[salt\_pepper\_noise < prob/2] = 255

# Add Pepper (black)

noisy\_img[salt\_pepper\_noise > 1 - prob/2] = 0

# -------------------------

# Apply Filters

# -------------------------

# Median Filter

median\_img = cv2.medianBlur(noisy\_img, 5)

# Gaussian Blur

gaussian\_img = cv2.GaussianBlur(noisy\_img, (5,5), sigmaX=50, sigmaY=50)

# Bilateral Filter

bilateral\_img = cv2.bilateralFilter(noisy\_img, d=9, sigmaColor=75, sigmaSpace=75)

# -------------------------

# Visualization

# -------------------------

titles = ['Original', 'Noisy', 'Median Filter', 'Gaussian Blur', 'Bilateral Filter']

images = [img, noisy\_img, median\_img, gaussian\_img, bilateral\_img]

plt.figure(figsize=(15,8))

for i, (title, im) in enumerate(zip(titles, images), start=1):

ax = plt.subplot(2,3,i)

ax.imshow(im)

ax.set\_title(title)

ax.axis('off')

plt.tight\_layout()

plt.show()

|  |  |  |
| --- | --- | --- |
| **Filter** | **Function / Notes** | **Effect on Salt & Pepper** |
| Averaging (Mean) | cv2.blur(noisy\_image, (3,3)) | Smooths but blurs edges, not ideal for sharp noise. |
| Gaussian Blur | cv2.GaussianBlur(noisy\_image, (3,3), 0.5) | Weighted smoothing, edges slightly preserved. |
| Median Blur | cv2.medianBlur(noisy\_image, 3) | Replaces pixel with median → very effective for salt & pepper noise. |
| Bilateral Filter | cv2.bilateralFilter(noisy\_image, 9, 75, 75) | Smooths while preserving edges → good for denoising without blurring edges. |

**Observation:**

* **Median** → best for salt & pepper noise.
* **Bilateral** → preserves edges, less smoothing in flat regions.

## Gaussian Noise Addition and Filtering

**Adding Gaussian Noise**

**Purpose:** Simulate sensor noise or image degradation.

**Steps:**

* 1. Define noise parameters:

mean = 50

std\_dev = 30

* 1. Generate noise with NumPy:

gaussian\_noise = np.random.normal(mean, std\_dev, img.shape)

* 1. Convert image to float to avoid overflow:

image\_float = img.astype(np.float32)

* 1. Add noise:

noisy\_image = image\_float + gaussian\_noise

* 1. Clip to valid range [0, 255] and convert to uint8:

noisy\_image = np.clip(noisy\_image, 0, 255).astype(np.uint8)

**Observation:** Bright/dark pixels are randomly perturbed → image looks “grainy.”

## Code to Add gaussian Noise:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# -------------------------

# Gaussian Noise Parameters

# -------------------------

mean = 50

std\_dev = 30

# Convert image to float32 to avoid overflow

image\_float = img.astype(np.float32)

# Generate Gaussian noise

gaussian\_noise = np.random.normal(mean, std\_dev, img.shape).astype(np.float32)

# Add noise to the image

noisy\_image = image\_float + gaussian\_noise

# Clip values to [0, 255] and convert to uint8

noisy\_image = np.clip(noisy\_image, 0, 255).astype(np.uint8)

# -------------------------

# Visualization

# -------------------------

plt.figure(figsize=(12,6))

plt.subplot(1,2,1)

plt.imshow(img)

plt.title("Original Image")

plt.axis("off")

plt.subplot(1,2,2)

plt.imshow(noisy\_image)

plt.title(f"Gaussian Noise Added (mean={mean}, std={std\_dev})")

plt.axis("off")

plt.tight\_layout()

plt.show()

## Gaussian Noise Filtering: Using Mean, Median, Guassian, Bilateral

import cv2

import numpy as np

import matplotlib.pyplot as plt

# -------------------------

# Read image

# -------------------------

img\_bgr = cv2.imread("Image/image.jpg")

img = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# -------------------------

# Add Gaussian Noise

# -------------------------

mean = 50

std\_dev = 30

# Convert to float32 to avoid overflow

image\_float = img.astype(np.float32)

# Generate Gaussian noise

gaussian\_noise = np.random.normal(mean, std\_dev, img.shape).astype(np.float32)

# Add noise

noisy\_img = image\_float + gaussian\_noise

# Clip to [0,255] and convert to uint8

noisy\_img = np.clip(noisy\_img, 0, 255).astype(np.uint8)

# -------------------------

# Apply Filters

# -------------------------

kernel\_size = 5 # For mean, Gaussian, median

# Mean Filter (Averaging)

mean\_img = cv2.blur(noisy\_img, (kernel\_size, kernel\_size))

# Gaussian Blur

gaussian\_img = cv2.GaussianBlur(noisy\_img, (kernel\_size, kernel\_size), sigmaX=50, sigmaY=50)

# Median Blur

median\_img = cv2.medianBlur(noisy\_img, kernel\_size)

# Bilateral Filter (edge-preserving)

bilateral\_img = cv2.bilateralFilter(noisy\_img, d=9, sigmaColor=75, sigmaSpace=75)

# -------------------------

# Visualization

# -------------------------

titles = ['Original', 'Noisy', 'Mean Filter', 'Gaussian Blur', 'Median Blur', 'Bilateral Filter']

images = [img, noisy\_img, mean\_img, gaussian\_img, median\_img, bilateral\_img]

plt.figure(figsize=(18,10))

for i, (title, im) in enumerate(zip(titles, images), start=1):

ax = plt.subplot(2,3,i)

ax.imshow(im)

ax.set\_title(title)

ax.axis('off')

plt.tight\_layout()

plt.show()

Findings:

**1️. Mean Filter (Averaging)**

1. The mean filter smooths the image by averaging the pixel values in the neighborhood.
2. It reduces Gaussian noise moderately but causes noticeable blurring of edges and fine details.
3. This filter is suitable for images with small, uniform noise where edge preservation is not critical.

**2️. Gaussian Blur**

1. Gaussian blur uses a weighted kernel to smooth the image, giving more importance to central pixels.
2. It effectively reduces Gaussian noise while slightly preserving edges compared to mean filtering.
3. This filter works well for images with Gaussian noise when moderate edge retention is required.

**3️. Median Blur**

1. Median blur replaces each pixel with the median value of its neighbors, preserving edge information.
2. It is less effective in removing Gaussian noise but works well for impulse (Salt & Pepper) noise.
3. Median filtering slightly preserves details compared to averaging filters but may leave some Gaussian noise.

**4️. Bilateral Filter**

1. Bilateral filter reduces noise while preserving edges by considering both spatial distance and intensity differences.
2. It effectively removes Gaussian noise without blurring important edges or fine details.
3. This filter is ideal for images where edge preservation and noise reduction are both important, though it is computationally more intensive.

### ****2.**** Filtering Noisy Images

* **Objective:** Reduce noise while preserving edges (depending on filter)

|  |  |  |
| --- | --- | --- |
| **Filter** | **Function / Notes** | **Effect on Gaussian Noise** |
| Averaging (Mean) | cv2.blur(noisy\_image, (k,k)) | Smooths image, blurs edges. |
| Gaussian Blur | cv2.GaussianBlur(noisy\_image, (k,k), sigma) | Weighted smoothing, better edge preservation than mean. |
| Median Blur | cv2.medianBlur(noisy\_image, k) | Replaces pixel with median, effective for impulsive noise. |
| Bilateral Filter | cv2.bilateralFilter(noisy\_image, d, sigmaColor, sigmaSpace) | Smooths noise while keeping edges sharp. |

**Tips:**

* Larger kernel → stronger smoothing.
* Gaussian σ → controls weight spread; higher σ → more blur.
* Median → robust to salt & pepper, less effective for Gaussian.
* Bilateral → best for edge-preserving smoothing.

### ****4. Notes for Color Images****

* Gaussian noise can be added per channel (R/G/B).
* Filters work the same; convert to grayscale only if you want single-channel analysis.

Which filtering operation do you think is more suitable for removing the Gaussian Noise?? > Hint: Since Gaussian noise has high-frequency variations, low-pass filters help reduce it.

|  |  |  |
| --- | --- | --- |
| **Filter** | **Suitability for Gaussian Noise** | **Notes** |
| **Averaging (Mean) Filter** | ✅ Good | Simple, smooths the image, but blurs edges. |
| **Gaussian Filter** | ✅ Very Good | Weighted averaging; preserves edges better than mean. Widely used for Gaussian noise removal. |
| **Median Filter** | ⚠ Moderate | Excellent for impulsive noise (salt & pepper) but less effective for Gaussian noise. |
| **Bilateral Filter** | ✅ Good | Smooths noise while preserving edges; slower than Gaussian. |

**Conclusion:**

* **Best choice:** **Gaussian Blur** — because it is designed to reduce high-frequency noise while retaining image structure.
* **Alternative:** **Bilateral Filter** if edge preservation is critical.

# Chapter Three:- ****Finding Primitives**** (Edge and Corner Detection)

**Edge Detection:** A technique in computer vision that identifies points in an image where pixel intensity changes sharply, highlighting object boundaries or texture changes.

**Types:** Gradient-based (Sobel, Prewitt), Laplacian-based, Canny edge detection, corner detection (Harris), and line detection (Hough Transform)

1. **Gradient Edge Detection (Sobel)** → Compute horizontal and vertical gradients.
2. **Laplacian Edge Detection** → Compute second-order derivatives to detect sharper intensity changes.
3. **Canny Edge Detection** – Detect edges using gradient-based filtering.
4. **Harris Corner Detection** – Detect corner points in an image.
5. **Hough Line Transform** – Detect straight lines from an edge map.

**Keywords to know:**

* **Keypoints:** Distinctive points in an image that are easily identifiable and repeatable.
* **Features:** Descriptive information extracted from keypoints or regions to characterize an image.
* **Keypoint features:** Specific descriptors computed at keypoints to enable matching across images.
* **Interest points:** Points in an image that are salient or informative, often selected for analysis.
* **Correspondences:** Matches between keypoints or features in two or more images.

## ****Gradient Edge Detection (Sobel)****

**Definition:** Detects edges by computing first-order derivatives in horizontal and vertical directions.  
**Application:** Object boundary detection, feature extraction, image preprocessing for segmentation.

**OpenCV Implementation:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Gradient Edge Detection (Sobel)

sobel\_x = cv2.Sobel(img\_gray, cv2.CV\_64F, 1, 0, ksize=3)

sobel\_y = cv2.Sobel(img\_gray, cv2.CV\_64F, 0, 1, ksize=3)

sobel\_edges = cv2.magnitude(sobel\_x, sobel\_y)

sobel\_edges = np.uint8(np.clip(sobel\_edges, 0, 255))

# Display using Matplotlib

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

plt.imshow(sobel\_edges, cmap='gray', vmin=0, vmax=255)

plt.title("Sobel Edge Detection")

plt.axis('off')

plt.show()

Manual Implementation (using NumPy convolution):

import numpy as np

import matplotlib.pyplot as plt

from scipy.ndimage import convolve

import cv2

# Read image

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Manual Sobel Edge Detection

Kx = np.array([[-1, 0, 1],

[-2, 0, 2],

[-1, 0, 1]])

Ky = np.array([[-1, -2, -1],

[0, 0, 0],

[1, 2, 1]])

grad\_x = convolve(img\_gray.astype(float), Kx)

grad\_y = convolve(img\_gray.astype(float), Ky)

sobel\_edges\_manual = np.sqrt(grad\_x\*\*2 + grad\_y\*\*2)

sobel\_edges\_manual = np.uint8(np.clip(sobel\_edges\_manual, 0, 255))

# ------------------------

# Display the result

# ------------------------

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

plt.imshow(sobel\_edges\_manual, cmap='gray', vmin=0, vmax=255)

plt.title("Manual Sobel Edge Detection")

plt.axis('off')

plt.show()

**1️. Smaller kernel size (e.g., 3×3)**

* Looks at a **small neighborhood** around each pixel.
* Detects **fine edges** and small details.
* But also **sensitive to noise**, because small variations in intensity (noise) are amplified.
* Example: Hair strands, thin lines are visible, but random bright spots may appear as false edges.

**2️. Larger kernel size (e.g., 7×7 or 9×9)**

* Looks at a **larger neighborhood**.
* Averages out small intensity variations → **reduces noise**.
* Detected edges become **thicker and smoother**.
* Example: Main object boundaries are clear, fine details may be lost.

## ****Laplacian Edge Detection****

**Laplacian Edge Detection** computes the **second-order derivative** of an image to detect regions where intensity changes sharply, highlighting edges more prominently than first-order methods (like Sobel). It is **isotropic**, meaning it detects edges in all directions.

**Applications**

* Detecting object boundaries in images.
* Medical imaging for detecting edges in MRI or CT scans.
* Feature extraction for computer vision tasks.
* Preprocessing for image segmentation and shape analysis.

Manual Implementation (Using Convolution)

import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy.ndimage import convolve

# Read image and convert to grayscale

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Laplacian kernel

laplacian\_kernel = np.array([[0, 1, 0],

[1,-4, 1],

[0, 1, 0]])

# Apply convolution

laplacian\_edges = convolve(img\_gray.astype(float), laplacian\_kernel)

laplacian\_edges = np.uint8(np.clip(np.abs(laplacian\_edges), 0, 255))

# Display

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

plt.imshow(laplacian\_edges, cmap='gray')

plt.title("Manual Laplacian Edge Detection")

plt.axis('off')

plt.show()

OpenCV Implementation

import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy.ndimage import convolve

# Read image and convert to grayscale

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Using OpenCV Laplacian function

laplacian\_edges\_cv = cv2.Laplacian(img\_gray, cv2.CV\_64F, ksize=3)

laplacian\_edges\_cv = np.uint8(np.clip(np.abs(laplacian\_edges\_cv), 0, 255))

# Display

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

plt.imshow(laplacian\_edges\_cv, cmap='gray')

plt.title("OpenCV Laplacian Edge Detection")

plt.axis('off')

plt.show()

**Visual analogy**

* **Small kernel** → fine brush: captures all tiny details but also captures noise.
* **Large kernel** → thick brush: smooths edges, removes noise, but loses finer details.

## ****Canny Edge Detection****

**Canny Edge Detection** is an algorithm that **detects edges in an image robustly** by computing intensity gradients, suppressing non-maximum points, and using double thresholding to identify strong and weak edges while reducing the effect of noise.

### ****Steps Involved****

1. **Noise reduction** – Apply a Gaussian filter to smooth the image.
2. **Gradient calculation** – Compute intensity gradients in x and y directions.
3. **Non-maximum suppression** – Thin out edges by keeping only local maxima.
4. **Double thresholding & edge tracking** – Classify strong/weak edges and link them.

Manual Implementation of Canny Edge Detection (Simplified)

import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy import ndimage

# Read image and convert to grayscale

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Step 1: Noise reduction (Gaussian smoothing)

blur = cv2.GaussianBlur(img\_gray, (5,5), sigmaX=1.4)

# Step 2: Compute gradients using Sobel

Gx = cv2.Sobel(blur, cv2.CV\_64F, 1, 0, ksize=3)

Gy = cv2.Sobel(blur, cv2.CV\_64F, 0, 1, ksize=3)

grad\_magnitude = np.sqrt(Gx\*\*2 + Gy\*\*2)

grad\_direction = np.arctan2(Gy, Gx) # in radians

# Step 3: Non-maximum suppression (simplified)

M, N = grad\_magnitude.shape

nms = np.zeros((M,N))

angle = grad\_direction \* 180. / np.pi

angle[angle < 0] += 180

for i in range(1,M-1):

for j in range(1,N-1):

q = 255

r = 255

# Horizontal

if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):

q = grad\_magnitude[i, j+1]

r = grad\_magnitude[i, j-1]

# 45 degrees

elif (22.5 <= angle[i,j] < 67.5):

q = grad\_magnitude[i+1, j-1]

r = grad\_magnitude[i-1, j+1]

# Vertical

elif (67.5 <= angle[i,j] < 112.5):

q = grad\_magnitude[i+1, j]

r = grad\_magnitude[i-1, j]

# 135 degrees

elif (112.5 <= angle[i,j] < 157.5):

q = grad\_magnitude[i-1, j-1]

r = grad\_magnitude[i+1, j+1]

if (grad\_magnitude[i,j] >= q) and (grad\_magnitude[i,j] >= r):

nms[i,j] = grad\_magnitude[i,j]

else:

nms[i,j] = 0

# Step 4: Double thresholding

high\_thresh = np.max(nms) \* 0.2

low\_thresh = high\_thresh \* 0.5

strong\_edges = (nms > high\_thresh)

weak\_edges = ((nms >= low\_thresh) & (nms <= high\_thresh))

canny\_manual = np.zeros\_like(img\_gray)

canny\_manual[strong\_edges] = 255

canny\_manual[weak\_edges] = 50 # mark weak edges lightly

# Display

plt.figure(figsize=(12,6))

plt.subplot(1,2,1)

plt.imshow(img\_gray, cmap='gray')

plt.title("Original Grayscale")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(canny\_manual, cmap='gray')

plt.title("Manual Canny Edge Detection (Simplified)")

plt.axis('off')

plt.show()

### ****OpenCV Implementation****

**Function:** cv2.Canny(image, threshold1, threshold2, L2gradient)

threshold1 → Lower threshold for edge linking.

threshold2 → Upper threshold for strong edges.

L2gradient → Use L2 norm (more accurate gradient magnitude).

**Example:**

import cv2

import matplotlib.pyplot as plt

# Read and grayscale

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Apply Canny

edges = cv2.Canny(img\_gray, threshold1=100, threshold2=200) # thresholds can be tuned

# Display

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img\_gray, cmap='gray')

plt.title("Original Grayscale")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(edges, cmap='gray')

plt.title("Canny Edge Detection - OpenCV")

plt.axis('off')

plt.show()

Canny Edge Detection: Class Based

import numpy as np

import cv2

from scipy import ndimage

import matplotlib.pyplot as plt

class CannyEdgeDetector:

def \_\_init\_\_(self, image\_path, sigma=1, low\_threshold=0.05, high\_threshold=0.15):

image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

if image is None:

raise FileNotFoundError(f"Error: Could not read image from path '{image\_path}'")

self.image = image

self.sigma = sigma

self.low\_threshold = low\_threshold

self.high\_threshold = high\_threshold

# -----------------------------

# Step 1: Gaussian Blur

# -----------------------------

def gaussian\_blur(self):

return ndimage.gaussian\_filter(self.image, self.sigma)

# -----------------------------

# Step 2: Gradient Magnitude & Direction

# -----------------------------

def sobel\_filters(self, img):

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

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

Gx = ndimage.convolve(img, sobel\_x)

Gy = ndimage.convolve(img, sobel\_y)

magnitude = np.hypot(Gx, Gy)

magnitude = (magnitude / magnitude.max()) \* 255

direction = np.arctan2(Gy, Gx) \* 180 / np.pi

direction[direction < 0] += 180

return magnitude, direction

# -----------------------------

# Step 3: Non-Maximum Suppression (Vectorized)

# -----------------------------

def non\_maximum\_suppression(self, magnitude, direction):

M, N = magnitude.shape

Z = np.zeros((M, N), dtype=np.float32)

angle = direction.copy()

angle[angle < 0] += 180

# Round angle to 4 main directions: 0, 45, 90, 135

angle = (np.round(angle / 45) \* 45) % 180

# Pad the image to avoid boundary issues

mag\_padded = np.pad(magnitude, ((1,1),(1,1)), mode='constant')

# Compare neighbors based on direction

for i in range(1, M+1):

for j in range(1, N+1):

q = 255

r = 255

if angle[i-1, j-1] == 0:

q = mag\_padded[i, j+1]

r = mag\_padded[i, j-1]

elif angle[i-1, j-1] == 45:

q = mag\_padded[i-1, j+1]

r = mag\_padded[i+1, j-1]

elif angle[i-1, j-1] == 90:

q = mag\_padded[i-1, j]

r = mag\_padded[i+1, j]

elif angle[i-1, j-1] == 135:

q = mag\_padded[i-1, j-1]

r = mag\_padded[i+1, j+1]

if mag\_padded[i, j] >= q and mag\_padded[i, j] >= r:

Z[i-1, j-1] = mag\_padded[i, j]

return Z

# -----------------------------

# Step 4: Double Threshold

# -----------------------------

def double\_threshold(self, img):

high\_val = img.max() \* self.high\_threshold

low\_val = img.max() \* self.low\_threshold

strong\_edges = np.zeros\_like(img, dtype=np.uint8)

weak\_edges = np.zeros\_like(img, dtype=np.uint8)

strong\_edges[img >= high\_val] = 255

weak\_edges[(img >= low\_val) & (img < high\_val)] = 75

return strong\_edges, weak\_edges

# -----------------------------

# Step 5: Edge Tracking by Hysteresis (Vectorized)

# -----------------------------

def hysteresis(self, strong\_edges, weak\_edges):

M, N = strong\_edges.shape

final\_edges = strong\_edges.copy()

# Identify weak pixels that have strong neighbors

shift\_coords = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

weak\_y, weak\_x = np.where(weak\_edges == 75)

for y, x in zip(weak\_y, weak\_x):

for dy, dx in shift\_coords:

ny, nx = y+dy, x+dx

if 0 <= ny < M and 0 <= nx < N:

if strong\_edges[ny, nx] == 255:

final\_edges[y, x] = 255

break

return final\_edges

# -----------------------------

# Full Pipeline

# -----------------------------

def process(self):

smoothed = self.gaussian\_blur()

magnitude, direction = self.sobel\_filters(smoothed)

suppressed = self.non\_maximum\_suppression(magnitude, direction)

strong, weak = self.double\_threshold(suppressed)

final\_edges = self.hysteresis(strong, weak)

return smoothed, magnitude, suppressed, strong, weak, final\_edges

# -----------------------------

# Run Example

# -----------------------------

if \_\_name\_\_ == '\_\_main\_\_':

image\_filename = 'assets/3d\_drawings.jpg'. # Change your Image Here

try:

detector = CannyEdgeDetector(image\_filename)

smoothed, magnitude, suppressed, strong, weak, final\_edges = detector.process()

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

titles = ['Original Image', 'Gaussian Blur', 'Gradient Magnitude',

'Non-Maximum Suppression', 'Double Threshold', 'Final Edges']

images = [detector.image, smoothed, magnitude,

suppressed, strong+weak, final\_edges]

for ax, img, title in zip(axes.ravel(), images, titles):

ax.imshow(img, cmap='gray')

ax.set\_title(title)

ax.axis('off')

plt.tight\_layout()

plt.show()

except FileNotFoundError as e:

print(e)

## ****Key Parameters in OpenCV****

|  |  |
| --- | --- |
| Parameter | Description |
| threshold1 | Lower bound for edge detection (weak edge). |
| threshold2 | Upper bound (strong edge). |
| apertureSize | Size of Sobel kernel used internally (default 3). |
| L2gradient | If True, computes gradient magnitude using sqrt(Gx² + Gy²). |

**1️. Increasing thresholds**

* **Higher threshold1 and threshold2** → fewer edges detected.
* Only the **strongest gradients** are considered edges.
* Weak edges (like textures or noise) are **ignored**.
* Result: **cleaner but sparser edge map**.

**2️. Decreasing thresholds**

* **Lower threshold1 and threshold2** → more edges detected.
* Even **weak gradients** are included.
* Can detect **fine details**, but also more **noise**.
* Result: **denser edge map**, sometimes cluttered.

**3️. Practical tip**

* Usually, threshold2 > threshold1.
* Use a **ratio of ~2:1 or 3:1** (e.g., 100 and 200, or 50 and 150).
* If the image is noisy, **increase thresholds** to ignore noise.

## ****Harris Corner Detection****

**Corner (in Computer Vision):**

A **corner** is a point in an image where **two or more edges meet**, representing a **distinctive feature** that is invariant to translation, rotation, and scaling, often used for **feature matching, object recognition, and tracking**.

**Harris Corner Detection** is an algorithm used to **identify corner points** or interest points in an image by analyzing local intensity variations in multiple directions. Corners are points where the intensity changes significantly in both horizontal and vertical directions. Detect corners in an image by analyzing intensity changes in all directions. Corners are points where intensity varies significantly along both X and Y directions.

**Parameters explanation:**

* blockSize → neighborhood size considered for corner detection.
* ksize → aperture parameter for Sobel derivative used internally.
* k → Harris detector free parameter (typical 0.04–0.06).

OpenCV Implementation (Recommended):

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read and grayscale

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

img\_rgb = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# Harris corner detection

dst = cv2.cornerHarris(img\_gray, blockSize=2, ksize=3, k=0.04)

# Dilate for marking the corners

dst\_dilated = cv2.dilate(dst, None)

# Threshold to mark corners in red

img\_harris = img\_rgb.copy()

img\_harris[dst\_dilated > 0.01 \* dst\_dilated.max()] = [255, 0, 0] # Red color

# Display

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img\_rgb)

plt.title("Original")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(img\_harris)

plt.title("Harris Corners")

plt.axis('off')

plt.show()

Manual Implementation Using Gradients

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read and grayscale

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Compute image gradients

Ix = cv2.Sobel(img\_gray, cv2.CV\_64F, 1, 0, ksize=3)

Iy = cv2.Sobel(img\_gray, cv2.CV\_64F, 0, 1, ksize=3)

# Products of derivatives

Ix2 = Ix\*\*2

Iy2 = Iy\*\*2

Ixy = Ix\*Iy

# Apply Gaussian filter to smooth the squared gradients

Sx2 = cv2.GaussianBlur(Ix2, (3,3), 0)

Sy2 = cv2.GaussianBlur(Iy2, (3,3), 0)

Sxy = cv2.GaussianBlur(Ixy, (3,3), 0)

# Harris response calculation

k = 0.04

R = (Sx2 \* Sy2 - Sxy\*\*2) - k\*(Sx2 + Sy2)\*\*2

# Threshold and mark corners

img\_manual = img\_rgb.copy()

img\_manual[R > 0.01\*R.max()] = [0,0,255] # Red corners

# Display

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img\_rgb)

plt.title("Original")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(img\_manual)

plt.title("Manual Harris Corners")

plt.axis('off')

plt.show()

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Typical Values / Effect** |
| src | Input grayscale image | Image where corners are to be detected |
| blockSize | Neighborhood size for corner detection | 2–5; Larger → smoother response, fewer corners; Smaller → more sensitive, more corners |
| ksize | Aperture size for Sobel derivative | 3, 5; Larger → smoother gradient, thicker edges; Smaller → more detailed, possibly noisy corners |
| k | Harris detector free parameter | 0.04–0.06; Higher → stricter corner detection; Lower → more corners detected |
| **Output** | Corner response map | Float32 image indicating corner strength at each pixel |

**Complete Python code** that shows the **Harris response map:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image and convert to grayscale and RGB for visualization

img\_bgr = cv2.imread("Image/image.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

img\_rgb = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB)

# Harris Corner Detection

blockSize = 2

ksize = 3

k = 0.04

harris\_response = cv2.cornerHarris(img\_gray, blockSize, ksize, k)

# 1️ Visualize the Harris Response Map

plt.figure(figsize=(8,5))

plt.imshow(harris\_response, cmap='rainbow')

plt.title("Harris Response Map")

plt.axis('off')

plt.show()

# 2️ Threshold and mark corners on the original image

img\_harris = img\_rgb.copy()

threshold = 0.01 \* harris\_response.max()

img\_harris[harris\_response > threshold] = [255, 0, 0] # Red corners

# Display original + corners

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(img\_rgb)

plt.title("Original Image")

plt.axis('off')

plt.subplot(1,2,2)

plt.imshow(img\_harris)

plt.title("Harris Corners Overlay")

plt.axis('off')

plt.show()

## ****Key Points / Observations****

|  |  |
| --- | --- |
| **Parameter** | **Effect / Observation** |
| **Corners** | Occur where intensity changes sharply in **both X and Y directions**. |
| **blockSize** | Size of the neighborhood considered for corner detection. • Larger → detects corners in larger areas. • Smaller → more sensitive to fine details. |
| **ksize** | Aperture size for Sobel derivative (gradient calculation). • Larger → smoother gradients, less noise-sensitive. • Smaller → more detailed gradients, may detect smaller corners. |
| **k** | Harris detector sensitivity parameter. • Too low → may **miss corners**. • Too high → may detect **false corners** (edges mistaken as corners). |

## ****Marking Detected Harris Corners****

**Step 1: Dilate the response**

kernel = np.ones((5, 5), np.uint8) # bigger kernel → bigger visual markers

harris\_response\_dilated = cv.dilate(harris\_response, kernel)

* **Purpose:** Make the corner points more visible.
* Dilation enlarges high-response regions for easier marking.

**Step 2: Threshold the response**

th = 0.1 \* harris\_response\_dilated.max()

* Only pixels with values above th are considered **strong corners**.
* **Tip:** Adjust 0.1 to control corner density:
  + Higher → fewer corners
  + Lower → more corners

**Step 3: Prepare image for marking**

marked\_bgr = img\_bgr.copy()

* Keep original image intact.
* Draw markers on the copy.

**Step 4: Find coordinates of corners**

coords = np.argwhere(harris\_response\_dilated > th) # returns (y, x)

* Returns **all pixel coordinates** above threshold.

**Step 5: Draw circles at corner locations**

for y, x in coords:

cv.circle(marked\_bgr, (x, y), radius=3, color=(0, 0, 255), thickness=-1)

* Draw **filled red circles** (BGR: (0,0,255)) at each corner.
* radius controls marker size.

**Step 6: Display the result**

marked\_rgb = cv.cvtColor(marked\_bgr, cv.COLOR\_BGR2RGB)

plt.imshow(marked\_rgb)

plt.title('Marked Corners')

plt.axis('off')

plt.show()

* Convert BGR → RGB for proper visualization in Matplotlib.
* All strong corners are now clearly visible.

✅ **Key Points**

* Dilating the response ensures visibility.
* Thresholding filters weak responses.
* Red circles indicate **detected corner points**.
* Parameter tuning (kernel size, threshold, circle radius) affects detection quality and visualization.

**Harris response map (heatmap)** and the **final detected corners** side by side:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/tshirt.jpg")

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Harris corner detection

harris\_response = cv2.cornerHarris(img\_gray, blockSize=2, ksize=3, k=0.04)

# Step 1: Dilate response to enhance corner regions

harris\_response\_dilated = cv2.dilate(harris\_response, None)

# Step 2: Threshold for strong corners

th = 0.02 \* harris\_response\_dilated.max() # threshold ratio

coords = np.argwhere(harris\_response\_dilated > th) # (y, x)

# Step 3: Copy original image and mark corners

marked\_bgr = img\_bgr.copy()

for y, x in coords:

cv2.circle(marked\_bgr, (x, y), radius=3, color=(255, 0, 0), thickness=1)

# Convert to RGB for matplotlib

marked\_rgb = cv2.cvtColor(marked\_bgr, cv2.COLOR\_BGR2RGB)

# ---- Visualization ----

plt.figure(figsize=(14,6))

# Harris response map (heatmap)

plt.subplot(1,2,1)

plt.imshow(harris\_response, cmap='jet')

plt.title("Harris Response Map")

plt.axis('off')

plt.colorbar(fraction=0.046, pad=0.04)

# Final corners marked

plt.subplot(1,2,2)

plt.imshow(marked\_rgb)

plt.title("Detected Harris Corners")

plt.axis('off')

plt.tight\_layout()

plt.show()

## ****Harris Response vs Dilated Response****

### ****Harris Response (****harris\_response****)****

* **What it is:**  
  The raw output from OpenCV’s cv.cornerHarris() function.  
  Each pixel contains a **corner strength score** (float).
* **Interpretation:**
  + **High values:** Likely a corner.
  + **Low or negative values:** Flat regions or edges.
* **Purpose:**  
  To decide which pixels are corners after applying a threshold.
* **Appearance:**  
  When visualized directly, it looks **noisy or fine-grained**.  
  Only a few pixels may have high values corresponding to corners.

### ****2. Harris Dilated (****harris\_dilated****)****

* **What it is:**  
  The Harris response after applying **morphological dilation** (cv.dilate).
* **Purpose of dilation:**
  + Does **not detect new corners**.
  + Makes **existing corner responses larger and more visible**.
  + Helps in visualization: small high-response pixels become blobs for easier display.
* **Effect:**
  + Larger dots at corner points.
  + Easier to overlay on the original image.

✅ **Key Points:**

* Thresholding happens **after dilation** to select strong corners.
* Dilation is **purely for visualization**, not a part of the detection algorithm.

## ****Hough Line Transform (HLT)****

The **Hough Line Transform (HLT)** is a **feature extraction technique** in computer vision used to **detect straight lines** in an image. HLT detects **straight lines** by converting points to parameter space (ρ,θ) and finding **accumulated peaks**, useful in roads, documents, and structural analysis

Any straight line in the image can be represented by the equation:

*y=mx+c or* ρ=xcosθ+ysinθ

*where:*

* *ρ = perpendicular distance from the origin to the line*
* *θ = angle of the perpendicular to the line from the origin*
* *The Hough Transform* ***maps points in the image space*** *to* ***curves in the Hough (parameter) space****.*
* ***Intersection points in Hough space*** *correspond to* ***lines passing through multiple points*** *in the image.*

### ****Applications of Hough Line Transform****

1. **Lane detection in autonomous vehicles** → detect road lane lines.
2. **Document analysis** → detecting text lines or table borders.
3. **Object recognition** → detecting rectangular shapes or boundaries.
4. **Robot navigation** → identifying walls or paths.
5. **Image processing** → detecting edges or straight structural lines in buildings, bridges, etc.

### ****Steps in Hough Line Transform****

1. Detect edges in the image (commonly using **Canny edge detector**).
2. Transform edge points into Hough space (ρ,θ\rho, \thetaρ,θ).
3. Count intersections (accumulator array) → stronger peaks = likely lines.
4. Convert peaks back to image space → draw detected lines.

Step 1: Import libraries and read image

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image

img\_bgr = cv2.imread("Image/road.jpg") # replace with your image path

img\_gray = cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2GRAY)

# Display original image

plt.imshow(cv2.cvtColor(img\_bgr, cv2.COLOR\_BGR2RGB))

plt.title("Original Image")

plt.axis('off')

plt.show()

**Step 2: Edge detection using Canny**

**Explanation of Canny parameters:**

* threshold1 → lower threshold for edge detection.
* threshold2 → upper threshold.
* Lower threshold: minimum gradient to consider an edge.
* Upper threshold: maximum gradient for strong edges.

edges = cv2.Canny(img\_gray, threshold1=90,

threshold2=250)

plt.imshow(edges, cmap='gray')

plt.title("Canny Edges")

plt.axis('off')

plt.show()

### **Step 3: Hough Line Transform (Standard)**

lines = cv2.HoughLines(edges,rho=1,theta=np.pi/180,threshold=100)

### **Step 4: Draw detected lines**

* edges → input binary edge image from Canny
* rho → distance resolution (1 pixel is typical
* theta → angle resolution (in radians, π/180 = 1 degree
* threshold → minimum number of votes (intersections in Hough space) required to consider a line.

img\_lines = img\_bgr.copy()

for line in lines:

rho, theta = line[0]

a = np.cos(theta)

b = np.sin(theta)

x0 = a \* rho

y0 = b \* rho

# Line endpoints (extend it for visualization)

x1 = int(x0 + 1000 \* (-b))

y1 = int(y0 + 1000 \* (a))

x2 = int(x0 - 1000 \* (-b))

y2 = int(y0 - 1000 \* (a))

cv2.line(img\_lines, (x1, y1), (x2, y2), (0, 0, 255), 2)

# Display result

plt.imshow(cv2.cvtColor(img\_lines, cv2.COLOR\_BGR2RGB))

plt.title("Detected Lines (Hough Transform)")

plt.axis('off')

plt.show()

**Hough Circle:**

import cv2 as cv

import numpy as np

import matplotlib.pyplot as plt

img\_path = 'Image/circle\_1.jpg'

img = cv.imread(img\_path)

assert img is not None, "file could not be read, check with os.path.exists"

# Convert to grayscale

gray = cv.cvtColor(img, cv.COLOR\_BGR2GRAY)

# Detect circles

circles = cv.HoughCircles(

gray,

cv.HOUGH\_GRADIENT,

dp=1, # inverse ratio of resolution

minDist=500, # minimum distance between circle centers

param1=100, # high threshold for internal Canny

param2=30, # accumulator threshold for circle detection

minRadius=0,

maxRadius=0

)

# Draw detected circles

if circles is not None:

circles = np.uint16(np.around(circles))

for i in circles[0, :]:

# Outer circle

cv.circle(img, (i[0], i[1]), i[2], (0, 255, 0), 2)

# Circle center

cv.circle(img, (i[0], i[1]), 2, (0, 0, 255), 3)

# Show result

plt.figure(figsize=(12, 8))

plt.imshow(cv.cvtColor(img, cv.COLOR\_BGR2RGB))

plt.title("Detected Circles (Hough Transform)")

plt.axis("off")

plt.show()

|  |  |  |
| --- | --- | --- |
| Parameter | Meaning | Typical Effect |
| **image** | Input image (must be grayscale). | Convert cv2.imread BGR → GRAY before passing. |
| **method** | Detection method. Currently only cv2.HOUGH\_GRADIENT. | Uses edges + gradients to find circles. |
| **dp** | Inverse ratio of accumulator resolution to image resolution. | 1 = same resolution (precise, slower). 2 = half resolution (faster, less precise). |
| **minDist** | Minimum distance between detected circle centers. | Small value = may detect multiple nearby circles. Large value = misses close circles. |
| **param1** | Upper threshold for the internal Canny edge detector (lower = half of this). | Higher → stricter edges (fewer circles). Lower → more edges (possible false detections). |
| **param2** | Accumulator threshold for circle center detection. | Higher → fewer but stronger circles. Lower → more circles, risk of false positives. |
| **minRadius** | Minimum radius of circles to detect. | Helps ignore tiny circles. |
| **maxRadius** | Maximum radius of circles to detect. | Helps ignore very large circles. |

### ****6. Key Points****

* Hough Transform is robust to gaps in edges because it votes in parameter space.
* **ρ and θ resolutions** affect accuracy: smaller values → finer detection.
* **Threshold** affects number of lines detected: higher → fewer, stronger lines.
* Can detect **infinite lines**; endpoints are computed for display

### ****Summary****

* **Canny** → finds edges.
* **Hough Transform** → finds lines via voting in ρ–θ space.
* **Polar-to-Cartesian** → converts detected lines to drawable points.
* **Visualization** → shows detected lines on the image.

# ****Morphological Transformations****

**Definition:**  
Morphological operations are image processing techniques based on the shape of objects in an image. Usually applied on **binary images**, but can also work on grayscale images.

**1. Erosion; Erosion** is a **morphological operation** used in image processing.

* It **shrinks white regions (foreground)** in a binary image.
* The operation uses a **structuring element (kernel)** and slides it over the image:
  + A pixel in the output is **white (1)** only if **all pixels under the kernel are white** in the input.
  + Otherwise, it becomes black (0).

👉 Effect:

* Removes small white noises.
* Breaks apart thin connections.
* Makes objects smaller.

**Code Example:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image in grayscale

img = cv2.imread("Image/chessboard.jpg",0)# if your image is already binary then No)

# Convert to binary (thresholding)

\_, binary = cv2.threshold(img, 127, 255, cv2.THRESH\_BINARY)

# Define kernel (structuring element)

kernel = np.ones((5,5), np.uint8)

# Apply erosion

erosion = cv2.erode(binary, kernel, iterations=1) # start with 1 iteration

# Show results

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.imshow(binary, cmap="gray")

plt.title("Original Binary Image")

plt.axis("off")

plt.subplot(1,2,2)

plt.imshow(erosion, cmap="gray")

plt.title("After Erosion")

plt.axis("off")

plt.show()

**Dilation:**

It’s a morphological operation that **increases the white region (foreground) in a binary image**, often used to fill small holes or connect regions.

### ****Applications of Dilution / Dilation****

* **Image Processing / Computer Vision**
  + - Fill small gaps or holes in binary images.
    - Connect disconnected components.
    - Enhance features for edge detection.
* **Medical / Biological Research**
  + - Preparing diluted solutions for experiments.
* **Chemistry**
  + - Reduce the concentration of a solution to achieve desired reaction rates

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read image in grayscale

img = cv2.imread("Image/chessboard.jpg", 0) # grayscale

# Threshold to get a binary image

\_, binary\_img = cv2.threshold(img, 127, 255, cv2.THRESH\_BINARY)

# Define kernel (structuring element)

kernel = np.ones((5, 5), np.uint8)

# Apply Dilation

dilated\_img = cv2.dilate(binary\_img, kernel, iterations=3)

# Show original and dilated images

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.title("Original Binary Image")

plt.imshow(binary\_img, cmap='gray')

plt.subplot(1,2,2)

plt.title("Dilated Image")

plt.imshow(dilated\_img, cmap='gray')

plt.show()

## ****Opening****

**Definition:**

* **Opening** is a **morphological operation** in image processing.
* It is a combination of **erosion followed by dilation**.
* Mainly used to **remove small noise** from an image without affecting the shape of the main object.

**Applications:**

* Removing **salt noise** (small white dots) in binary images.
* Preprocessing images before **object detection** or **contour extraction**.

**Python Code Example:**

import cv2 as cv

import numpy as np

import matplotlib.pyplot as plt

# Read image in grayscale

img = cv.imread("Image/noisy\_image.jpg", 0)

# Define kernel

kernel = np.ones((3,3), np.uint8)

# Apply Opening

opening = cv.morphologyEx(img, cv.MORPH\_OPEN, kernel)

# Show images

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.title("Original Image")

plt.imshow(img, cmap='gray')

plt.subplot(1,2,2)

plt.title("After Opening")

plt.imshow(opening, cmap='gray')

plt.show()

**2. Closing**

**Definition:**

* **Closing** is another **morphological operation**.
* It is a combination of **dilation followed by erosion**.
* Mainly used to **remove small holes or black spots** inside the foreground objects.

**Applications:**

* Removing **pepper noise** (small black dots).
* **Closing gaps** in detected shapes or text.
* Useful in **OCR preprocessing** or **shape smoothing**.

**Python Code Example:**

# Apply Closing

closing = cv.morphologyEx(img, cv.MORPH\_CLOSE, kernel)

# Show images

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

plt.title("Original Image")

plt.imshow(img, cmap='gray')

plt.subplot(1,2,2)

plt.title("After Closing")

plt.imshow(closing, cmap='gray')

plt.show()

**Morphological Gradient**

* **Definition:** The morphological gradient is the difference between the dilation and erosion of an image.
* **Effect / Use Case:**
  + Extracts object boundaries or outlines.
  + Useful for edge detection in binary or grayscale images.
* **Python Code Example:**

import cv2 as cv

import numpy as np

# Read image

img = cv.imread("image.jpg", 0) # grayscale

# Define kernel

kernel = np.ones((3,3), np.uint8)

# Morphological Gradient

gradient = cv.morphologyEx(img, cv.MORPH\_GRADIENT, kernel)

# Show result

cv.imshow("Morphological Gradient", gradient)

cv.waitKey(0)

cv.destroyAllWindows()

## ****Top Hat (White Top Hat)****

* **Definition:** The Top Hat operation is the difference between the input image and its opening.
* **Effect / Use Case:**
  + Highlights small bright regions on a dark background.
  + Useful for detecting small bright objects or features in images.
* **Python Code Example with Multiple Kernels:**

import cv2 as cv

import numpy as np

# Read image

img = cv.imread("image.jpg", 0)

# Define multiple kernel sizes

kernels = [5, 7, 9, 11]

# Apply Top Hat

tophat = [cv.morphologyEx(img, cv.MORPH\_TOPHAT, np.ones((k, k), np.uint8)) for k in kernels]

# Display results

for i, th in enumerate(tophat):

cv.imshow(f"Top Hat Kernel {kernels[i]}", th)

cv.waitKey(0)

cv.destroyAllWindows()

## ****Black Hat****

* **Definition:** The Black Hat operation is the difference between the closing of the image and the input image.
* **Effect / Use Case:**
  + Highlights small dark regions on a bright background.
  + Useful for detecting small dark spots or holes in images.
* **Python Code Example with Multiple Kernels:**

import cv2 as cv

import numpy as np

# Read image

img = cv.imread("image.jpg", 0)

# Define multiple kernel sizes

kernels = [5, 7, 9, 11]

# Apply Black Hat

blackhat = [cv.morphologyEx(img, cv.MORPH\_BLACKHAT, np.ones((k, k), np.uint8)) for k in kernels]

# Display results

for i, bh in enumerate(blackhat):

cv.imshow(f"Black Hat Kernel {kernels[i]}", bh)

cv.waitKey(0)

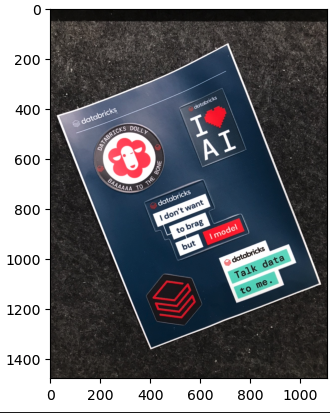
cv.destroyAllWindows()

**Summary Table**

|  |  |  |
| --- | --- | --- |
| Operation | Formula | Effect / Purpose |
| Erosion | erode(img) | Shrinks foreground, removes small white noise |
| Dilation | dilate(img) | Expands foreground, fills small holes |
| Opening | erode → dilate | Removes noise, preserves object shape |
| Closing | dilate → erode | Fills holes in foreground objects |
| Gradient | dilate - erode | Extracts object edges |
| Top Hat | img - opening | Highlights small bright regions |
| Black Hat | closing - img | Highlights small dark regions |

# Chapter Four- Feature Description:

Perspective Transformation:

**It** is a type of geometric transformation that maps the points in one plane to another plane in such a way that straight lines remain straight, but parallelism may not be preserved. It simulates how objects appear smaller as they move farther from the viewer, i.e., it accounts for **perspective distortion**.

**This code is selecting the four points of the original image and cropping only that to make**

**New image.**

import cv2

import numpy as np

import matplotlib.pyplot as plt

image\_path = "./Homography/databricks.jpg" # Replace with your image path

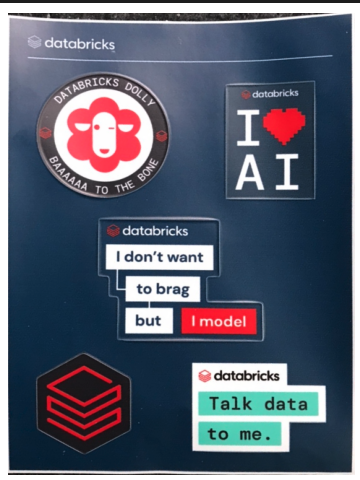
image = cv2.imread(image\_path)

# Source points: top-left, top-right, bottom-left, bottom-right

src\_corners = np.float32([[30, 423], [712, 144], [406, 1367], [1081, 1103]])

# Destination points (output rectangle)

dst\_corners = np.float32([[0, 0], [480, 0], [0, 640], [480, 640]]) # 480x640 output

****

M = cv2.getPerspectiveTransform(src\_corners, dst\_corners)

warped\_img = cv2.warpPerspective(image, M, (480, 640))

warped\_img\_rgb = cv2.cvtColor(warped\_img, cv2.COLOR\_BGR2RGB)

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

plt.imshow(warped\_img\_rgb)

plt.axis("off")

plt.title("Perspective Cropped Image")

plt.show()

### ****Replacing the certain portion of Perspective image with normal image****

import cv2

import numpy as np

import matplotlib.pyplot as plt

# -----------------------------

# 1️ Load Images

# -----------------------------

persp\_img = cv2.imread("Homography/desk-perspective.png") # Angled desk image

norm\_img = cv2.imread("Homography/desk-normal.png") # Flat image to overlay

if persp\_img is None or norm\_img is None:

raise FileNotFoundError("Check your image paths. One or both images not found.")

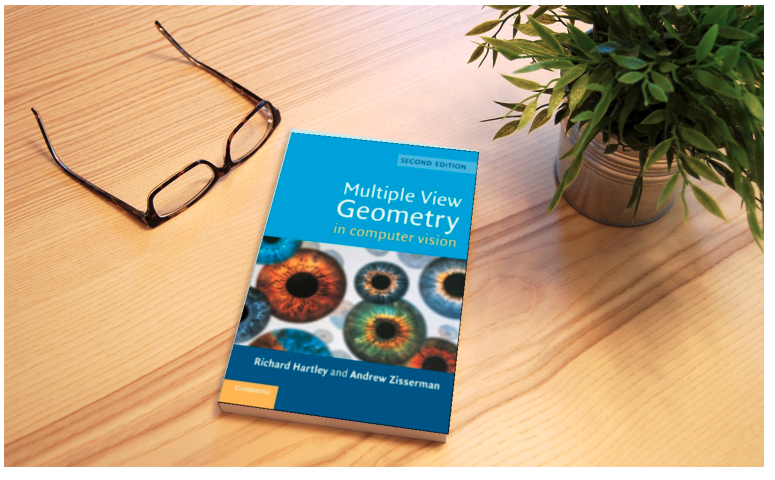
# -----------------------------

# 2️ Define Points

# -----------------------------

# Corners on perspective image (The area where normal image will replace)

corners\_perspective = np.array([

 [530.14, 235.25], # top-left

[874.5, 271.95], # top-right

[818.04, 791.30], # bottom-right

[394.66, 732.03] # bottom-left

], dtype=np.float32)

# Corners of normal image

h, w = norm\_img.shape[:2]

corners\_normal = np.array([

[0, 0], [w, 0], [w, h], [0, h]

], dtype=np.float32)

# -----------------------------

# 3️ Compute Homography & Warp

# -----------------------------

homography\_matrix, \_ = cv2.findHomography(corners\_normal, corners\_perspective)

warped\_normal\_image = cv2.warpPerspective(norm\_img, homography\_matrix, (persp\_img.shape[1], persp\_img.shape[0]))

# -----------------------------

# 4️ Create Mask for the Area

# -----------------------------

mask = np.zeros((persp\_img.shape[0], persp\_img.shape[1]), dtype=np.uint8) # single channel mask

cv2.fillConvexPoly(mask, corners\_perspective.astype(np.int32), 255) # white area = where normal image will appear

mask\_inv = cv2.bitwise\_not(mask) # invert mask for background

# -----------------------------

# 5️ Combine Images

# -----------------------------

# Keep perspective image where normal image will not go

persp\_background = cv2.bitwise\_and(persp\_img, persp\_img, mask=mask\_inv)

# Keep warped normal image where the mask is

normal\_foreground = cv2.bitwise\_and(warped\_normal\_image, warped\_normal\_image, mask=mask)

# Combine both

final\_image = cv2.add(persp\_background, normal\_foreground)

# -----------------------------

# 6️ Display Result

# -----------------------------

final\_image\_rgb = cv2.cvtColor(final\_image, cv2.COLOR\_BGR2RGB)

plt.figure(figsize=(10,6))

plt.imshow(final\_image\_rgb)

plt.axis('off')

plt.title("Normal Image in Perspective Position")

plt.show()

### ****Feature Matching using SIFT and Image Alignment:****

Using SIFT detector, detect features points and calculate descriptors. Then match correspondences between two given images.

## ****Cell 1: Load and Display Images****

import cv2 as cv

import numpy as np

import matplotlib.pyplot as plt

# -----------------------------

# 1️ Load Images

# -----------------------------

ref\_image\_path = 'Homography/refDatabricks.jpg'

target\_image\_path = 'Homography/databricks.jpg'

ref\_img = cv.imread(ref\_image\_path)

target\_img = cv.imread(target\_image\_path)

# Check images

assert ref\_img is not None, f"Image at {ref\_image\_path} not found"

assert target\_img is not None, f"Image at {target\_image\_path} not found"

# Convert BGR to RGB for Matplotlib display

ref\_img\_rgb = cv.cvtColor(ref\_img, cv.COLOR\_BGR2RGB)

target\_img\_rgb = cv.cvtColor(target\_img, cv.COLOR\_BGR2RGB)

# Display original images

plt.figure(figsize=[10,6])

plt.subplot(121); plt.imshow(ref\_img\_rgb); plt.axis('off'); plt.title("Reference")

plt.subplot(122); plt.imshow(target\_img\_rgb); plt.axis('off'); plt.title("Target")

plt.show()

### Cell 2 – Detect SIFT Keypoints and Descriptors

# -----------------------------

# 2️ Convert to Grayscale

# -----------------------------

ref\_gray = cv.cvtColor(ref\_img, cv.COLOR\_BGR2GRAY)

target\_gray = cv.cvtColor(target\_img, cv.COLOR\_BGR2GRAY)

# -----------------------------

# 3️Initialize SIFT and Detect Keypoints

# -----------------------------

sift = cv.SIFT\_create()

kp\_ref, des\_ref = sift.detectAndCompute(ref\_gray, None)

kp\_target, des\_target = sift.detectAndCompute(target\_gray, None)

# -----------------------------

# 4️ Visualize Keypoints

# -----------------------------

ref\_kp\_img = cv.drawKeypoints(ref\_img\_rgb, kp\_ref, outImage=np.array([]), color=(0,255,0), flags=cv.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

target\_kp\_img = cv.drawKeypoints(target\_img\_rgb, kp\_target, outImage=np.array([]), color=(255,0,0), flags=cv.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

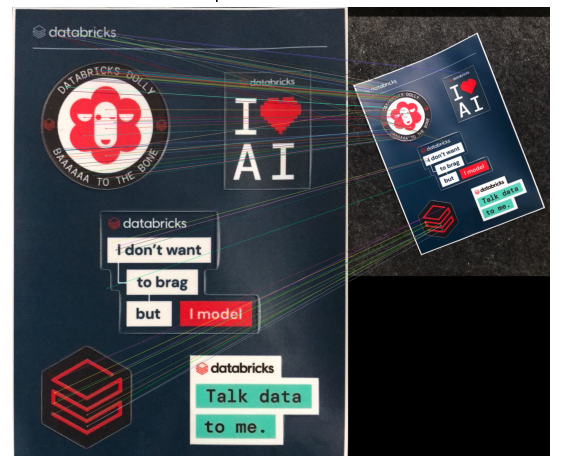
plt.figure(figsize=[10,6])

plt.subplot(121); plt.imshow(ref\_kp\_img); plt.axis('off'); plt.title("Reference Keypoints")

plt.subplot(122); plt.imshow(target\_kp\_img); plt.axis('off'); plt.title("Target Keypoints")

plt.show()

Cell 3 – Match Features, Draw Top 5 Matches, Inspect Descriptors

# -----------------------------

# 5️ Feature Matching (BFMatcher + Lowe's Ratio Test)

# -----------------------------

bf = cv.BFMatcher()

matches = bf.knnMatch(des\_ref, des\_target, k=2)

# Apply Lowe's ratio test

good\_matches = []

for m, n in matches:

if m.distance < 0.5 \* n.distance:

good\_matches.append([m])

# -----------------------------

# 6️ Draw Top 5 Matches

# -----------------------------

top\_matches = good\_matches[:5] # Take only the first 5 matches

matches\_img = cv.drawMatchesKnn(

ref\_img\_rgb, kp\_ref,

target\_img\_rgb, kp\_target,

top\_matches, None,

flags=cv.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS

)

plt.figure(figsize=[12,6])

plt.imshow(matches\_img)

plt.axis('off')

plt.title(f"Top 5 Good Matches")

plt.show()

# -----------------------------

# 7️Inspect Top 5 Good Match Descriptors

# -----------------------------

for i, match in enumerate(top\_matches):

m = match[0]

print(f"Match {i+1}:")

print(" Reference keypoint:")

print(" x, y:", kp\_ref[m.queryIdx].pt)

print(" size:", kp\_ref[m.queryIdx].size)

print(" angle:", kp\_ref[m.queryIdx].angle)

print(" Descriptor vector (128-d):", des\_ref[m.queryIdx])

print(" Target keypoint:")

print(" x, y:", kp\_target[m.trainIdx].pt)

print(" size:", kp\_target[m.trainIdx].size)

print(" angle:", kp\_target[m.trainIdx].angle)

print(" Descriptor vector (128-d):", des\_target[m.trainIdx])

print("-"\*50)

Find the Number of Descriptor, keypoints ??

# -----------------------------

# Inspect Descriptors and Keypoints

# -----------------------------

# Shapes of descriptors

print("Descriptor shape (Reference):", des\_ref.shape)

print("Descriptor shape (Target): ", des\_target.shape)

# Number of keypoints

print("Number of keypoints (Reference):", len(kp\_ref))

print("Number of keypoints (Target): ", len(kp\_target))

# First descriptor vectors (128-d)

print("\nFirst descriptor (Reference):\n", des\_ref[0])

print("\nFirst descriptor (Target):\n", des\_target[0])

# Define a function to match the keypoints the image “get\_similarity\_from\_desc “

def get\_similarity\_from\_desc(desc1, desc2):

bf = cv.BFMatcher()

matches = bf.knnMatch(desc1, desc2, k=2)

good = []

for m, n in matches:

if m.distance < 0.75 \* n.distance: # decreasing means higher precision,reduces match

good.append([m])

similarity\_percentage = (len(good) / len(matches)) \* 100

return len(good), similarity\_percentage

good\_count, similarity\_pct = get\_similarity\_from\_desc(des\_ref, des\_target)

print(f"Good matches: {good\_count}")

print(f"Similarity: {similarity\_pct:.2f}%")

Change a parameter of SIFT\_create() function to increase the number of images in a scale-space to =5, and find keypoints and descriptors in book\_scene.jpeg only. Analyze how the result (keypoints) changes in response to this parameter change. (5 points).

import cv2 as cv

import numpy as np

import matplotlib.pyplot as plt

# Load Images

ref\_image\_path = 'Homography/refDatabricks.jpg'

target\_image\_path = 'Homography/databricks.jpg'

ref\_img = cv.imread(ref\_image\_path)

target\_img = cv.imread(target\_image\_path)

# Convert BGR to RGB for Matplotlib display

ref\_img\_rgb = cv.cvtColor(ref\_img, cv.COLOR\_BGR2RGB)

target\_img\_rgb = cv.cvtColor(target\_img, cv.COLOR\_BGR2RGB)

# Convert to Grayscale

ref\_gray = cv.cvtColor(ref\_img, cv.COLOR\_BGR2GRAY)

target\_gray = cv.cvtColor(target\_img, cv.COLOR\_BGR2GRAY)

# Initialize SIFT and Detect Keypoints

sift = cv.SIFT\_create(nfeatures = 0, nOctaveLayers=5, contrastThreshold=0.04, edgeThreshold = 10, sigma = 1.6)

kp\_ref, des\_ref = sift.detectAndCompute(ref\_gray, None)

kp\_target, des\_target = sift.detectAndCompute(target\_gray, None)

bf = cv.BFMatcher()

matches = bf.knnMatch(des\_ref, des\_target, k=2)

# Apply Lowe's ratio test

good\_matches = []

for m, n in matches:

if m.distance < 0.5 \* n.distance:

good\_matches.append([m])

# -----------------------------

# 6️ Draw Top 5 Matches

# -----------------------------

top\_matches = good\_matches[:5] # Take only the first 5 matches

matches\_img = cv.drawMatchesKnn(

ref\_img\_rgb, kp\_ref,

target\_img\_rgb, kp\_target,

top\_matches, None,

flags=cv.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS

)

plt.figure(figsize=[12,6])

plt.imshow(matches\_img)

plt.axis('off')

plt.title(f"Top 5 Good Matches")

plt.show()

## Class Based SIFT image alignment or Feature Matching:

import cv2

import matplotlib.pyplot as plt

import numpy as np

class SIFTMatcher:

def \_\_init\_\_(self, ref\_image\_path, target\_image\_path, ratio=0.75):

self.ref\_image\_path = ref\_image\_path

self.target\_image\_path = target\_image\_path

self.ratio = ratio # Lowe's ratio for good matches

# Load images

self.ref\_img = cv.imread(ref\_image\_path)

self.target\_img = cv.imread(target\_image\_path)

if self.ref\_img is None or self.target\_img is None:

raise FileNotFoundError("One or both image paths are invalid.")

# Convert to grayscale

self.ref\_gray = cv.cvtColor(self.ref\_img, cv.COLOR\_BGR2GRAY)

self.target\_gray = cv.cvtColor(self.target\_img, cv.COLOR\_BGR2GRAY)

# Convert to RGB for visualization

self.ref\_rgb = cv.cvtColor(self.ref\_img, cv.COLOR\_BGR2RGB)

self.target\_rgb = cv.cvtColor(self.target\_img, cv2.COLOR\_BGR2RGB)

# Initialize SIFT

self.sift = cv.SIFT\_create()

self.kp\_ref, self.des\_ref = None, None

self.kp\_target, self.des\_target = None, None

self.good\_matches = None

# Step 1: Detect keypoints and compute descriptors

def detect\_and\_compute(self):

self.kp\_ref, self.des\_ref = self.sift.detectAndCompute(self.ref\_gray, None)

self.kp\_target, self.des\_target = self.sift.detectAndCompute(self.target\_gray, None)

# Step 2: Match descriptors and compute good matches

def match\_features(self):

bf = cv.BFMatcher()

matches = bf.knnMatch(self.des\_ref, self.des\_target, k=2)

self.good\_matches = [m for m, n in matches if m.distance < self.ratio \* n.distance]

# Step 3: Compute similarity percentage

def similarity\_percentage(self):

if self.good\_matches is None:

raise ValueError("No matches computed. Call match\_features() first.")

return (len(self.good\_matches) / max(len(self.kp\_ref), len(self.kp\_target))) \* 100

# Step 4: Draw top N good matches

def draw\_matches(self, top\_n=20):

if self.good\_matches is None:

raise ValueError("No matches computed. Call match\_features() first.")

top\_matches = self.good\_matches[:top\_n]

matches\_img = cv.drawMatches(

self.ref\_rgb, self.kp\_ref,

self.target\_rgb, self.kp\_target,

top\_matches, None,

flags=cv.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS

)

plt.figure(figsize=(12, 6))

plt.imshow(matches\_img)

plt.axis('off')

plt.title(f"Top {top\_n} Good Matches")

plt.show()

# -----------------------------

# Example Usage

# -----------------------------

if \_\_name\_\_ == '\_\_main\_\_':

ref\_image = 'Homography/refDatabricks.jpg'

target\_image = 'Homography/databricks.jpg'

sift\_matcher = SIFTMatcher(ref\_image, target\_image, ratio=0.75)

sift\_matcher.detect\_and\_compute()

sift\_matcher.match\_features()

print(f"Number of good matches: {len(sift\_matcher.good\_matches)}")

print(f"Similarity: {sift\_matcher.similarity\_percentage():.2f}%")

sift\_matcher.draw\_matches(top\_n=5)

Flow:

 Loads and displays reference and target images.

 Detects SIFT keypoints and descriptors.

 Draws keypoints for each image.

 Matches features using BFMatcher + Lowe’s ratio test.

 Draws **top 5 matched keypoints** visually.

 Prints detailed descriptor info for these 5 matches

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Parameter** | **Description** | **Effect on Matching** |
| **SIFT Detector** | nFeatures | Maximum number of keypoints to detect | Fewer → faster but may miss keypoints; More → slower but more keypoints |
|  | contrastThreshold | Minimum contrast for a keypoint | Higher → fewer keypoints in low-contrast areas; Lower → more keypoints but noisy |
|  | edgeThreshold | Threshold to filter edge-like keypoints | Higher → keeps more edge keypoints; Lower → removes edge-like points |
|  | sigma | Gaussian smoothing for scale-space | Changes the scale of detected keypoints |
| **BFMatcher** | normType | Distance metric (cv2.NORM\_L2 for SIFT) | Determines how descriptor distance is calculated |
|  | crossCheck | If True, only mutual matches are kept | True → fewer but more reliable matches; False → more matches, may include false ones |
| **k-NN Matching** | k | Number of nearest neighbors per descriptor | k=2 is standard for Lowe’s ratio test; larger k may give more options for filtering |
| **Lowe’s Ratio Test** | ratio threshold | Threshold in if m.distance < ratio\*n.distance | Lower → stricter, fewer matches; Higher → more matches, may include false positives |
| **Image Preprocessing** | Grayscale, Resize, Blur/Noise | Preprocessing before detection | Good preprocessing → better keypoints and matches; Noise or extreme scale differences → poor matches |

KNN-

Import library:

#Import Necessary Library

import os

import torch

import numpy as np

import matplotlib.pyplot as plt

from torch.utils.data import DataLoader, Subset

from torchvision import datasets, transforms

# from torchvision.datasets import CIFAR10

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, confusion\_matrix

from scipy.spatial import distance

# from sklearn.datasets import fetch\_openml

# Using the default libraries function

from sklearn.neighbors import KNeighborsClassifier

from sklearn.linear\_model import LogisticRegression

Generating Sysnthetic Dataset:

# Generating synthetic dataset

X\_syn, y\_syn = make\_classification(n\_samples=200, n\_features=2, n\_informative=2, n\_redundant=0, n\_classes=2, random\_state=42)

X\_train\_syn, X\_test\_syn, y\_train\_syn, y\_test\_syn = train\_test\_split(X\_syn, y\_syn, test\_size=0.2, random\_state=42)

# To get unique values in y

unique\_train\_classes = np.unique(y\_train\_syn)

unique\_test\_classes = np.unique(y\_test\_syn)

print(unique\_train\_classes)

# Check if the unique classes in both arrays are equal

assert np.array\_equal(unique\_train\_classes, unique\_test\_classes), "Unique classes in train and test sets are different."

# NumPy arrays do not support direct comparison for equality! Instead we should use array\_equal function!

# Plot the data for training set

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

plt.scatter(X\_train\_syn[:, 0], X\_train\_syn[:, 1], c=y\_train\_syn, cmap='coolwarm', edgecolor='k', s=100)

plt.title("Scatter Plot of Training Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

Custom KNN

# Custom KNN Classifier

class CustomKNNClassifierCPU():

def \_\_init\_\_(self, k=3):

self.k = k

def fit(self, X\_train, y\_train):

self.X\_train = X\_train

self.y\_train = y\_train

def predict(self, X\_test):

y\_pred = []

for x in X\_test:

# Calculate the Euclidean Distance with each training sample

distances = [distance.euclidean(x, x\_train) for x\_train in self.X\_train]

# Sort the distances and get k nearest neighbors

k\_indices = np.argsort(distances)[:self.k]

# Find the labels of these nearest samples

# k\_nearest\_labels = [self.y\_train[i] for i in k\_indices]

k\_nearest\_labels = self.y\_train[k\_indices]

# Find the majority of repeated labels

most\_common = np.bincount(k\_nearest\_labels).argmax()

# Append the prediction

y\_pred.append(most\_common)

return np.array(y\_pred)

Training

# Training the KNN

knn = CustomKNNClassifierCPU(k=3)

knn.fit(X\_train\_syn, y\_train\_syn)

# Predicting

y\_pred = knn.predict(X\_test\_syn)

# Evaluation

print("Custom KNN Accuracy:", accuracy\_score(y\_test\_syn, y\_pred))

print("Confusion Matrix:\n", confusion\_matrix(y\_test\_syn, y\_pred))

#plotting

def plot\_decision\_boundary(clf, X, y):

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.5)

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k', marker='o')

plt.show()

# Plotting decision boundary for KNN

Now lets work on CIFAR Dataset:

Loading the GPU

# Check if GPU is available

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

Normailizing

# Desired mean and standard deviation for the normalization of inputs!

mean = 0.0

stddev = 1.0

# Define Transformation for input image. You may be able to use many more transform using this.

transform=transforms.Compose([transforms.ToTensor(),

transforms.Normalize((mean), (stddev))])

# This is equivalent to standard scalar funtion from sklearn.preprocessing But you can define own

Loading Data

cifar\_train = datasets.CIFAR10('./data', train=True, download=True ,transform=transforms.ToTensor())

cifar\_test = datasets.CIFAR10('./data', train=False, download=True ,transform=transforms.ToTensor())

# Converting the format of Data

print(f"Training data: {len(cifar\_train)}")

print(f"Test data: {len(cifar\_test)}")

image, label = cifar\_train[1]

# Now you can check the shape of the image

print(f"Image shape: {image.shape}")

# If the image is in [C, H, W] format, we need to permute it to [H, W, C] for displaying

image\_np = image.permute(1, 2, 0).cpu().numpy()

print(f"Image shape after transformation: {image\_np.shape}")

# print(image\_np) -- You will able to see that image is in range [0,1]

# Ensure it's in the right range [0, 255] for displaying

image\_np = (image\_np \* 255).astype('uint8')

# Display the image

plt.figure(figsize=(1,1))

plt.imshow(image\_np)

plt.axis('off') # Turn off axis labels

plt.show()

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

print(f"label: {classes[label]}")

# Function to subsample CIFAR-10 dataset

def subsample\_dataset(dataset, sample\_size=10000):

indices = np.random.choice(len(dataset), sample\_size, replace=False)

subset = Subset(dataset, indices)

return subset

# Subsample the training and test datasets

sample\_size = 10000

train\_subset = subsample\_dataset(cifar\_train, sample\_size=sample\_size)

test\_subset = subsample\_dataset(cifar\_test, sample\_size=int(sample\_size \* 0.4))

# Load data into PyTorch DataLoader

train\_loader = DataLoader(train\_subset, batch\_size=sample\_size, shuffle=True)

test\_loader = DataLoader(test\_subset, batch\_size=int(sample\_size \* 0.4), shuffle=False)

# Fetch all data and labels for easier handling

X\_train, y\_train = next(iter(train\_loader))

X\_test, y\_test = next(iter(test\_loader))

print("Before Flattening")

print(f"Training data shape: {X\_train.shape}")

print(f"Test data shape: {X\_test.shape}")

# Reshape the images to 2D for the KNN algorithm

X\_train = X\_train.view(X\_train.size(0), -1).to(device) # Flatten

X\_test = X\_test.view(X\_test.size(0), -1).to(device)

y\_train = y\_train.to(device)

y\_test = y\_test.to(device)

print("After Flattening")

print(f"Training data shape: {X\_train.shape}")

print(f"Test data shape: {X\_test.shape}")

Creating a class:

class CustomKNNClassifierGPU:

def \_\_init\_\_(self, k=3, num\_classes=10, chunk\_size=256):

self.k = k

self.num\_classes = num\_classes

self.chunk\_size = chunk\_size # Splits test data into chunks to control memory.

self.X\_train = None

self.y\_train = None

def fit(self, X\_train: torch.Tensor, y\_train: torch.Tensor):

# X\_train: [N, D] float tensor on device

# y\_train: [N] long tensor on same device

assert isinstance(X\_train, torch.Tensor)

assert isinstance(y\_train, torch.Tensor)

assert X\_train.device == y\_train.device

self.X\_train = X\_train

self.y\_train = y\_train.long()

@torch.no\_grad() # Saves Memory

def predict(self, X\_test: torch.Tensor) -> torch.Tensor:

assert self.X\_train is not None and self.y\_train is not None

assert X\_test.device == self.X\_train.device

preds = [] # to collect batch predictions.

N = X\_test.size(0) # num of test samples

for start in range(0, N, self.chunk\_size):

end = min(start + self.chunk\_size, N) # End index for this chunk.

xb = X\_test[start:end] # [B, D] Slice a batch of B test vectors with D features.

dists = torch.cdist(xb, self.X\_train, p=2) # [B, N] B rows (test items) by N columns (train items).

\_, idx = torch.topk(dists, k=self.k, largest=False) # [B, k] For each test item, get indices of the k smallest distances.

neigh\_labels = self.y\_train[idx] # [B, k] Gather labels of those neighbors using indices.

# Majority vote per row

batch\_pred = []

for row in neigh\_labels: # row: [k] Loop over each test item’s k labels.

counts = torch.bincount(row, minlength=self.num\_classes) # Count label frequency across the k neighbors.

batch\_pred.append(torch.argmax(counts)) # Pick the label with the highest count.

preds.append(torch.stack(batch\_pred))

return torch.cat(preds).to(X\_test.device) # [N]

# Training the model

# Initialize and train custom KNN classifier

if device.type == 'cuda':

knn = CustomKNNClassifierGPU(k=5)

else:

knn = CustomKNNClassifierCPU(k=5)

knn.fit(X\_train, y\_train)

# Predict and evaluate

y\_pred = knn.predict(X\_test)

accuracy = accuracy\_score(y\_test.cpu().numpy() if isinstance(y\_test, torch.Tensor) else y\_test, y\_pred.cpu().numpy() if isinstance(y\_pred, torch.Tensor) else y\_pred)

print(f"Accuracy of Custom KNN Classifier: {accuracy:.2f}")

# Predicting the model

# SKLEARN needs the data in cpu

X\_train\_np = X\_train.detach().cpu().numpy()

y\_train\_np = y\_train.detach().cpu().numpy()

X\_test\_np = X\_test.detach().cpu().numpy()

y\_test\_np = y\_test.detach().cpu().numpy()

knn\_sklearn = KNeighborsClassifier(n\_neighbors=3)

knn\_sklearn.fit(X\_train\_np, y\_train\_np)

# Predicting

y\_pred\_sklearn = knn\_sklearn.predict(X\_test\_np)

# Evaluation

print("sklearn KNN Accuracy:", accuracy\_score(y\_test\_np, y\_pred\_sklearn))

print("Confusion Matrix (sklearn KNN):\n", confusion\_matrix(y\_test\_np, y\_pred\_sklearn))

Training the model

# Initialize and train custom KNN classifier

if device.type == 'cuda':

knn = CustomKNNClassifierGPU(k=5)

else:

knn = CustomKNNClassifierCPU(k=5)

knn.fit(X\_train, y\_train)

# Predict and evaluate

y\_pred = knn.predict(X\_test)

accuracy = accuracy\_score(y\_test.cpu().numpy() if isinstance(y\_test, torch.Tensor) else y\_test, y\_pred.cpu().numpy() if isinstance(y\_pred, torch.Tensor) else y\_pred)

print(f"Accuracy of Custom KNN Classifier: {accuracy:.2f}")

# predicting

# SKLEARN needs the data in cpu

X\_train\_np = X\_train.detach().cpu().numpy()

y\_train\_np = y\_train.detach().cpu().numpy()

X\_test\_np = X\_test.detach().cpu().numpy()

y\_test\_np = y\_test.detach().cpu().numpy()

knn\_sklearn = KNeighborsClassifier(n\_neighbors=3)

knn\_sklearn.fit(X\_train\_np, y\_train\_np)

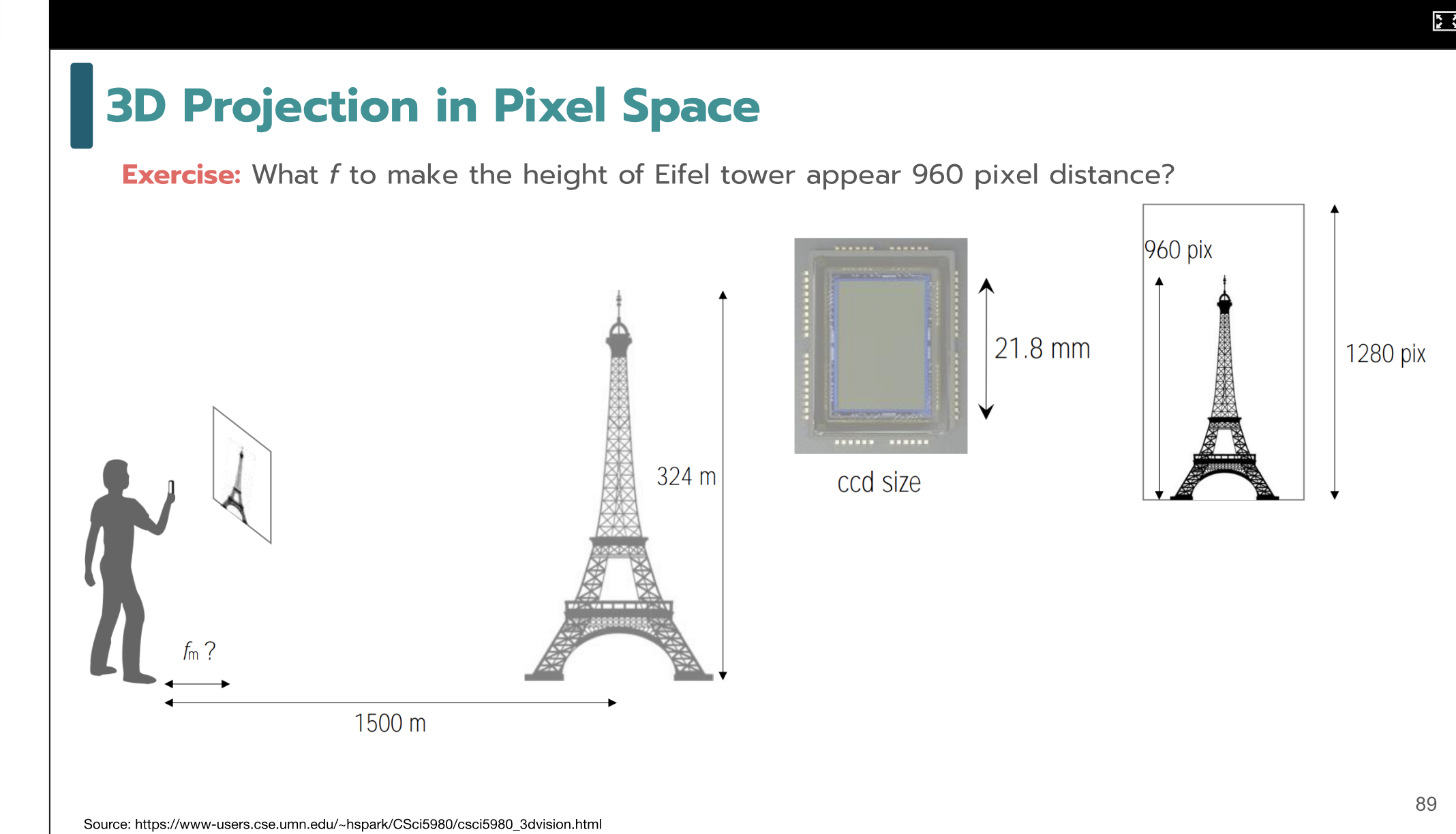
# Predicting

y\_pred\_sklearn = knn\_sklearn.predict(X\_test\_np)

# Evaluation

print("sklearn KNN Accuracy:", accuracy\_score(y\_test\_np, y\_pred\_sklearn))

print("Confusion Matrix (sklearn KNN):\n", confusion\_matrix(y\_test\_np, y\_pred\_sklearn))



Anser:



