Project Title: Image Filtering and Edge Detection using OpenCV

Submission Date: 24/11/25

Submitted By: Aadira Rajeev (23BHI10136)

Course/Context: COMPUTER VISION PROJECT1

INTRODUCTION

Image processing : Image processing, in the context of computer vision, is a core field that deals with **manipulating and analyzing digital images** using computer algorithms. Its primary goal is to either **improve the visual quality** of an image for human viewing or to **extract key features and information** to prepare the image for further analysis by a machine.

Image filtering in computer vision is the process of modifying or enhancing an image by applying a **kernel** (also called a convolution matrix) to every pixel in the image. It's a fundamental technique used to achieve specific visual effects, such as blurring, sharpening, or detecting edges.

GOAL: To implement and demonstrate fundamental image processing techniques (blurring, sharpening, and edge detection) using Python and OpenCV.

Problem Statement

The objective is to apply and visually compare different types of **convolution kernels** (for blurring/smoothing and sharpening) and the **Canny edge detection algorithm** to an input image. Specifically, the project addresses:

- How to smooth/blur an image using averaging (box) and Gaussian kernels.
- How to enhance image details using a sharpening kernel.
- How to detect structural boundaries (edges) in an image, highlighting the importance of pre-processing (blurring) for cleaner results.

Functional Requirements

- **Image Loading:** Must be able to load an image from a specified file path.

- **Image Blurring:** Must implement and demonstrate **Averaging Filter** (via cv2.filter2D and cv2.blur) and **Gaussian Blur** (cv2.GaussianBlur).
- **Image Sharpening:** Must implement and apply a **Sharpening Kernel** using cv2.filter2D.
- **Edge Detection:** Must implement the **Canny Edge Detection** algorithm (cv2.Canny).
- **Color Conversion:** Must convert the image to **Grayscale** for optimal edge detection.
- **Image Display:** Must provide a helper function to correctly display OpenCV's BGR images in Matplotlib (RGB).

Non-functional Requirements

- **Performance:** Image filtering operations should complete quickly for typical image sizes (achieved via OpenCV's optimized C/C++ backend).
- **Maintainability:** The code should be clear, modular, and well-commented (e.g., using helper functions like display_image).
- **Readability:** The convolution kernels should be clearly defined using NumPy arrays.

System Architecture

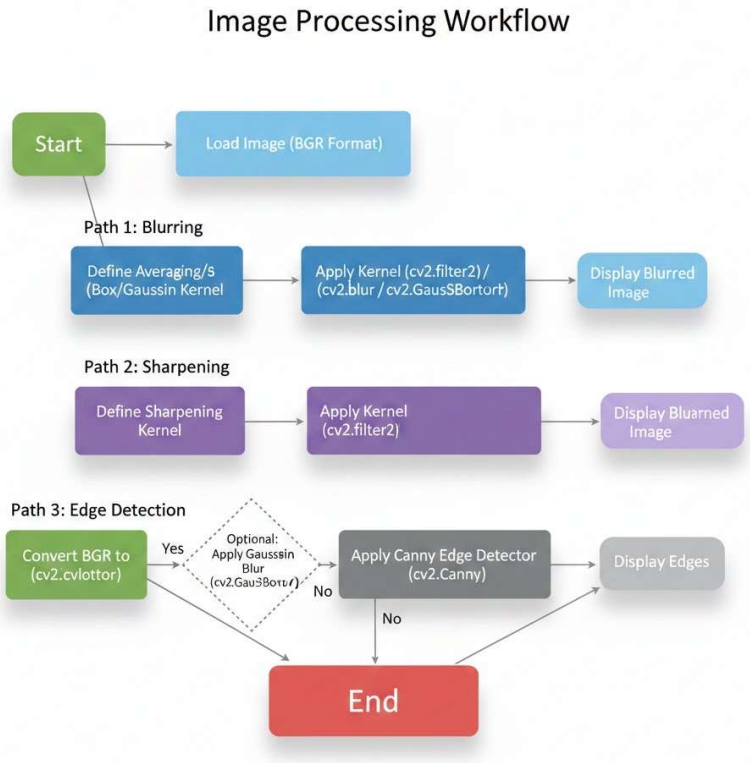
The system is a **standalone script** based on a **Client-Server Architecture** metaphor where:

- **Client (User/Jupyter Notebook):** Provides the input image path and executes the processing steps.
- **Server (OpenCV/NumPy Libraries):** Performs the heavy-lifting image processing tasks (reading, convolution, color space conversion, display).

Key Components:

- **Input Layer:** cv2.imread()
- **Processing Layer (Core):** numpy for kernel definition, cv2.filter2D, cv2.blur, cv2.GaussianBlur, cv2.Canny, cv2.cvtColor.
- **Output/Visualization Layer:** matplotlib.pyplot and the display_image helper function.

Design Diagrams



Design Decisions & Rationale

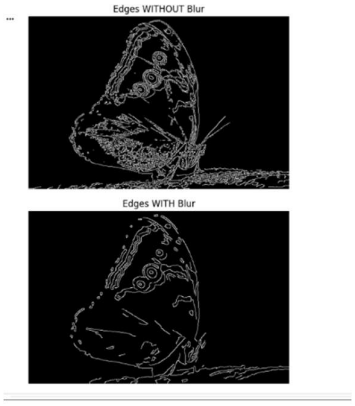
Design Decision	Rationale
Using cv2.filter2D for Averaging/Sharpening	This is the generic function in OpenCV for 2D convolution, allowing for the direct application of custom NumPy kernels (like the $\mathbf{5 \times 5}$ averaging or the $\mathbf{3 \times 3}$ sharpening matrix).
Using cv2.blur and cv2.GaussianBlur	To demonstrate and compare specialized, optimized functions for common filtering tasks against the generic cv2.filter2D. Gaussian blur is generally preferred for pre-processing due to its weighted average, which better preserves overall image structure.
Grayscale Conversion for Canny	Canny algorithm is most effective and computationally efficient when run on a single channel (grayscale) image, as edge detection is fundamentally based on intensity gradients.
Pre-blurring before Canny	**Crucial Rationale:** Applying a small Gaussian blur *before* Canny reduces **texture noise** and irrelevant small-scale details, ensuring the edge detector focuses on significant structural boundaries. This is explicitly demonstrated in the code.
Sharpening Kernel $\begin{pmatrix}-1 & -1 & -1	This kernel works by subtracting the sum of the surrounding pixels from a multiple of the center pixel (9 times the center). This is a

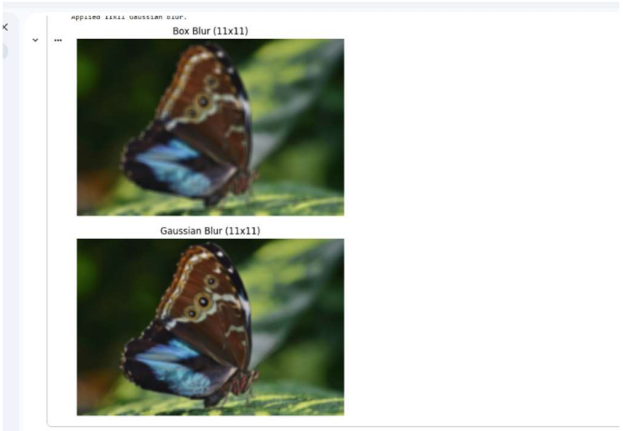
Design Decision	**Rationale**
\\ -1 & 9 & -1 \\ -1 & -1 & -1\end{pmatrix}$	common implementation of the **Unsharp Masking** technique, which enhances high-frequency components (details).

Implementation Details :

- **Libraries:** OpenCV (cv2), NumPy (np), Matplotlib (plt).
- **Kernel Definition:** Kernels are defined as **NumPy arrays** (np.ones for averaging, np.array for sharpening).
- **Averaging Kernel:** $\text{kernel} = \frac{1}{\text{size}^2} \times \mathbf{1}_{\text{size} \times \text{size}}$. This ensures the sum of kernel elements is $\mathbf{1}$, maintaining the overall image brightness.
- **display_image Function:** Handles the necessary **BGR to RGB conversion** (cv2.cvtColor) because OpenCV reads images in BGR format, but Matplotlib plots them in RGB. It also handles single-channel (grayscale/edge) images.

SCREENSHOTS :





Testing Approach

- **Unit Testing:** The focus was on ensuring the OpenCV functions were called with correct parameters (e.g., kernel size, threshold values for Canny, correct color space).
- **Visual Testing (Primary Method):** Output images were visually inspected to confirm the expected effect of each filter (e.g., blur looks blurred, sharpened looks sharper, Canny output contains prominent object boundaries).
- **Comparison Testing:** Specifically compared the results of Canny edge detection with and without a pre-processing blur to validate the effect of the Gaussian filter.

Challenges Faced

- **BGR vs. RGB:** Ensuring correct display in Matplotlib required explicitly converting the BGR image loaded by OpenCV to RGB format in the display_image function. Failing to do this results in incorrect colors.
- **Kernel Normalization:** Remembering to **normalize** the averaging kernel (divide by size^2) was essential to prevent the blurred image from becoming too bright or too dark.
- **Finding Optimal Canny Thresholds:** Determining the best low_threshold and high_threshold values for the Canny detector is often empirical and image-dependent. A slight change can drastically alter the final edge map.

Learnings & Key Takeaways

- **Convolution Principle:** Understood how a small kernel is slid across an image (convolution) to produce a filtered image.
- **Filter Types:** Learned the difference between **Low-Pass Filters** (Blurring/Smoothing, which remove high-frequency noise) and **High-Pass Filters** (Sharpening, which enhance high-frequency details).
- **Canny Algorithm:** Gained insight into the multi-stage Canny algorithm and the critical role of the initial Gaussian smoothing step to manage noise before gradient computation and hysteresis thresholding.
- **OpenCV Functionality:** Proficiently used core OpenCV functions like filter2D, blur, GaussianBlur, Canny, and color space conversion.

Future Enhancements

- **Interactive Parameter Tuning:** Implement sliders or input fields to allow the user to interactively change kernel sizes, Canny thresholds, and σ for Gaussian blur, seeing the results in real-time.
- **Advanced Filters:** Implement other convolution-based filters such as **Sobel** or **Laplacian** for gradient and detail detection.
- **Color-Space Filtering:** Explore filtering techniques in different color spaces (e.g., HSV or LAB) to target specific color components.

References

- OpenCV Documentation for cv2.filter2D, cv2.blur, cv2.GaussianBlur, cv2.Canny.
- Matplotlib and NumPy Documentation.
- [Any specific tutorials or textbooks used.]