University of Science

Report

Lab01

Teacher: *Phạm Minh Hoàng*
Teacher: *Nguyễn Mạnh Hùng*
Teacher: *Lý Quốc Ngọc*

Student: *Huỳnh Cao Minh*

Date: *25/11/2023*

TABLE OF CONTENTS

1. Work progress table:

STT	The name of the job	Implementation progress
Sobel Operator		
1	Using CV2	100%
2	Implementing based on the algorithms learned	100%
Laplace of Gaussian		
3	Using CV2	100%
4	Implementing based on the algorithms learned	100%
Canny method		
5	Using CV2	100%
6	Implementing based on the algorithms learned	100%

2. Implementation content:

- ### Implemented external library:

In the context of edge detection, the external libraries mentioned serve various purposes, providing functionalities that simplify and enhance the implementation of image processing tasks. Here's a brief explanation of the roles of each library:

- **cv2(OpenCV):** OpenCV is a powerful computer vision library that provides a wide range of tools for image and video processing. It is commonly used for loading and displaying images, applying various image processing operations, and working with computer vision algorithms. In the context of edge detection, OpenCV offers functions for convolution, gradient computation, and other related tasks.
- **numpy(np):** NumPy is a fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays. NumPy is used for array manipulation and mathematical operations. In edge detection, it is often used for handling image data as arrays and performing operations such as element-wise arithmetic, normalization, and array slicing.
- **PIL(Pillow):** Pillow is a fork of the Python Imaging Library (PIL), providing easy-to-use methods for opening, manipulating, and saving many different image file formats.
- **scipy.ndimage(convolve):** SciPy is a library for scientific and technical computing in Python, and *'scipy.ndimage'* is a submodule providing functions for multidimensional image processing.
- **matplotlib.pyplot(plt):** Matplotlib is a 2D plotting library for Python, and *'matplotlib.pyplot'* provides a MATLAB-like interface for creating visualizations.

In summary, these libraries collectively offer a comprehensive set of tools for loading, manipulating, and visualizing images, as well as implementing the necessary mathematical operations for edge detection. They help streamline the development process and enable the implementation of complex image processing tasks with relatively concise and readable code.

- ### Sobel Operator:

The Sobel operator performs a convolution operation on an image using a pair of 3x3 kernels (one for horizontal changes and the other for vertical changes). These kernels calculate the gradient of the image intensity in the horizontal and vertical directions. The magnitude of the combined gradients at each pixel represents the strength

of the edge at that location:

In mathematical terms, for an image I, the Sobel operator computes the gradients G_x and G_y using the following kernels:

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

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

The gradient magnitude G is then calculated as $G = \sqrt{G_x^2 + G_y^2}$.

- **Using CV2:**
  - **Reading the Image:** - The function takes the path to an image (*'image_path'*) as an argument.
    - *'Image.open(image_path)'* is used from the Pillow library (*'Image'* module) to open the image in its original form (*'img_origin'*).
    - *'cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)'* from the OpenCV library is used to read the image in grayscale (*'img'*).
  - **Gaussian Blur:** *'cv2.GaussianBlur'* is applied to the grayscale image (img) using a 5x5 kernel to reduce noise and improve the effectiveness of edge detection.
  - **Sobel Operator:**
    - *cv2.Sobel* is applied twice, once for the x-direction (*sobel_x*) and once for the y-direction (*sobel_y*), to compute gradients using the Sobel operator.
    - *cv2.CV_64F* specifies the data type of the output gradients as 64-bit floating-point numbers.
    - *ksize=3* indicates the size of the Sobel kernel as 3x3.
  - **Gradient Magnitude:** The gradient magnitudes in the x and y directions are combined to calculate the overall gradient magnitude using the Euclidean norm.
  - **Normalization and Type Conversion:** The gradient magnitude values are normalized to the range [0, 255] and converted to uint8 for display purposes.
  - **Test:**

- **Implementing based on the algorithms learned:**
  - **Image Conversion:** The input image is converted to the *float64* data type using *image.astype(np.float64)*. Converting the image to a floating-point format is common in image processing operations to ensure that calculations involving gradients are more accurate.
  - **Sobel Operator Kernels:** Two 3x3 kernels (*kernel_x* and *kernel_y*) are defined to represent the Sobel operator for gradient computation in the x and y directions.
  - **Convolution:**
    - The input image is convolved with the Sobel kernels (*kernel_x* and *kernel_y*) using *scipy.ndimage.convolve.*
    - *np.abs* is applied to ensure that the gradients are treated as magnitudes, making them positive.
  - **Gradient Magnitude:** The gradient magnitudes in the x and y directions are combined to compute the overall gradient magnitude using the Euclidean norm.
  - **Normalization and Type Conversion:** The gradient magnitude values are normalized to the range [0, 255] to ensure consistent intensity levels across different images.

  The resulting gradient magnitude is converted to uint8 data type for display.

  In summary, the *sobel_operator* takes an input image, applies the Sobel operator for gradient computation, computes the gradient magnitude, normalizes the values for display, and returns the processed gradient magnitude image. This function is commonly used as a building block in edge detection algorithms.
  - **Test:**

- **Laplace of Gaussian**
  - **Using CV2:**
    - **Grayscale Conversion:** The input color image (*image*) is converted to grayscale using *cv2.cvtColor* to simplify the subsequent image processing steps.
    - **Gaussian Smoothing:**
      - Gaussian smoothing is applied to the grayscale image (*gray_image*) using *cv2.GaussianBlur.*
      - The (0, 0) kernel size indicates that the size of the kernel is automatically determined based on the standard deviation (*sigma*) provided as an argument.
    - **Laplacian Filter:**
      - The Laplacian filter is applied to the smoothed image (*blurred_image*) using *cv2.Laplacian.*
      - *cv2.CV_64F* specifies the data type of the output as 64-bit floating-point numbers to handle negative values that may occur during the Laplacian operation.
    - **Conversion and Absolute Values:**
      - The absolute values of the Laplacian are taken to ensure that both positive and negative gradients contribute to the edge information.
      - The result is converted to uint8 for display purposes. The conversion involves scaling the values to the range [0, 255].
    - **Test:**

- **Implementing based on the algorithms learned**
  - **Generate a 2D Gaussian Filter:**
    - This function generates a 2D Gaussian filter of a specified size and sigma.
    - It creates a meshgrid (*xx*, *yy*) over a range defined by ax.
    - The Gaussian kernel is computed using the formula:

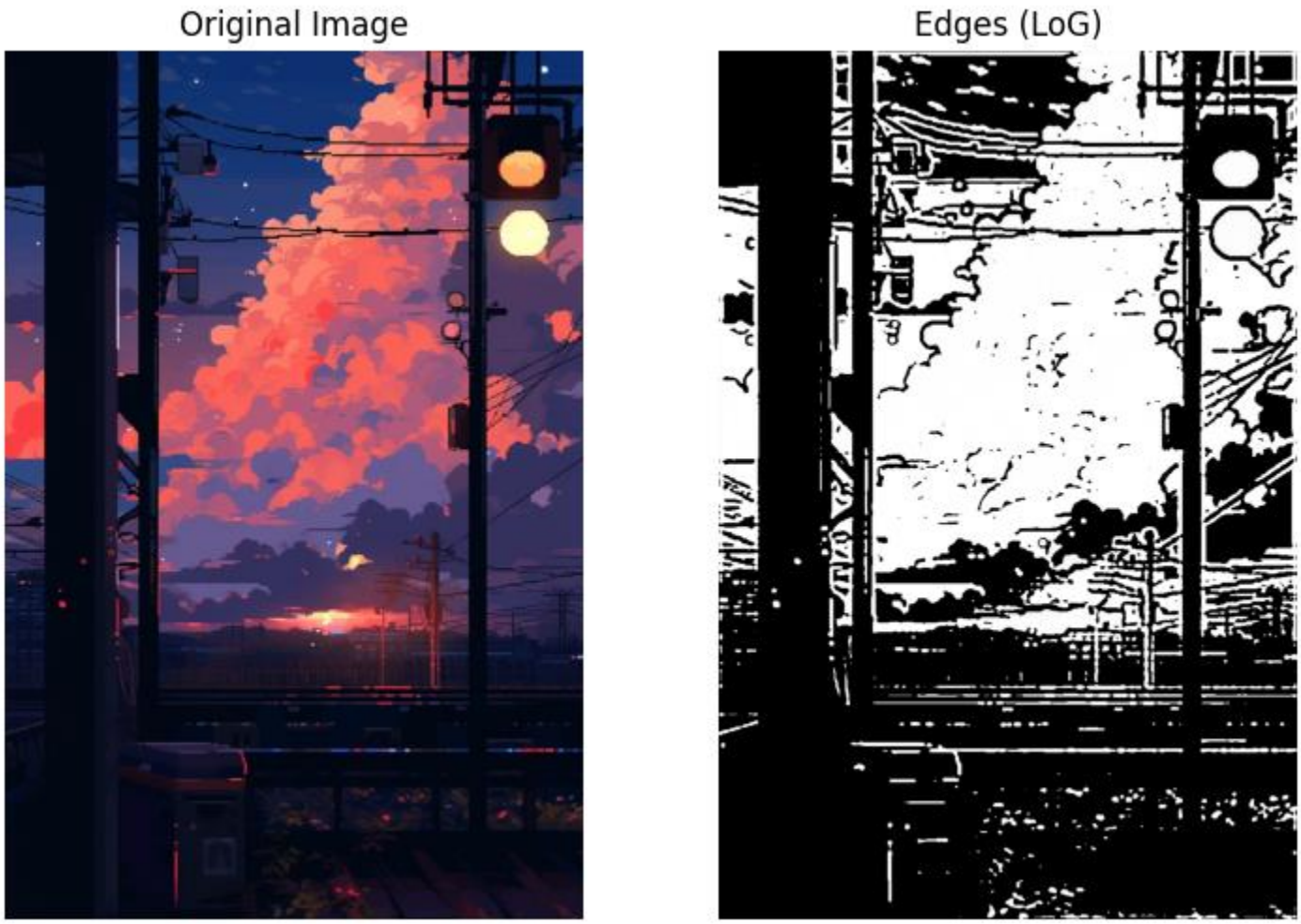
In two dimensions, it is the product of two such Gaussians, one per direction:

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)}$$ [3][4][5]

    - The kernel is then normalized by dividing by the sum of all its values to ensure the sum is 1.
  - **Generate a Gaussian Kernel:**
    - The *laplace_of_gaussian* function first calculates the size of the Gaussian kernel based on the provided sigma.
    - If the calculated size is even, it is incremented to ensure an odd-sized kernel.
    - It then calls the *gaussian_filter* function to generate the Gaussian kernel.
  - **Compute Laplace of Gaussian Filter:**
    - The Laplace of Gaussian (LoG) filter is computed by taking the double gradient of the Gaussian kernel.
    - This is done by applying the gradient along both axes using *np.gradient* twice.
  - **Convolve the Image:**
    - The image is convolved with the Laplace of Gaussian filter using *scipy.ndimage.convolve*.
    - The mode is set to *'constant'*, and the constant value for regions outside the image is set to 0 (*cval=0*).

In summary, these functions work together to generate a 2D Gaussian filter and then apply the Laplace of Gaussian edge detection to an input image. The Laplace of Gaussian method is often used for edge detection to enhance edges while suppressing noise through the initial Gaussian smoothing step.

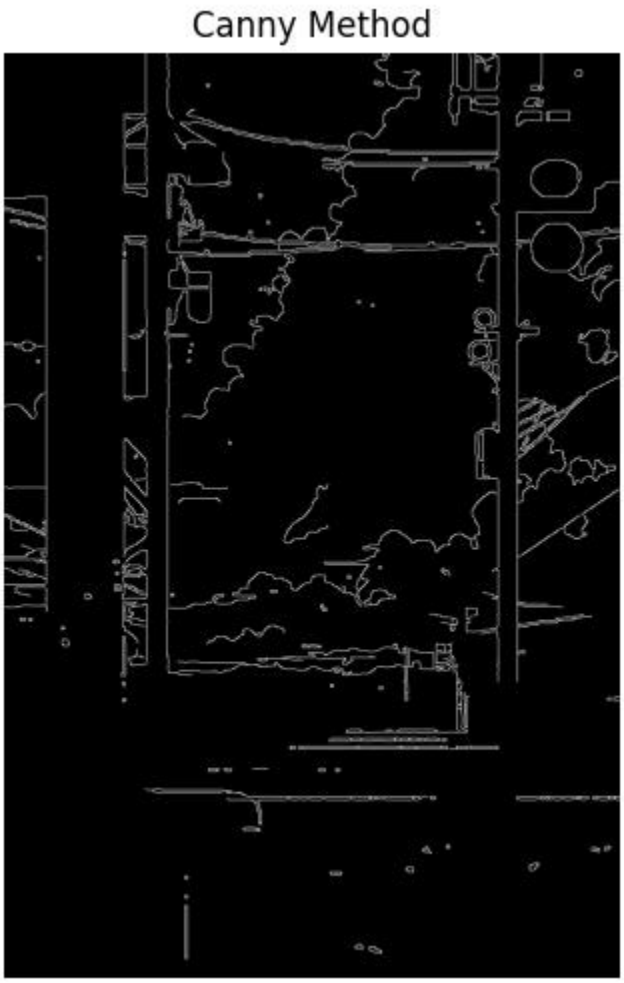
- **Test:**



- **Canny method**
  - **Using CV2**
    - **Gaussian Blur:**
      - Gaussian blur is applied to the grayscale image (*img*) using *cv2.GaussianBlur*.
      - The kernel size is set to (5, 5) to reduce noise and smoothen the image.
    - **Canny Edge Detection:**
      - Canny edge detection is applied to the blurred image (*blurred*) using *cv2.Canny*.
      - *low_threshold* and *high_threshold* are parameters that determine the edges. Pixels with gradient values below *low_threshold* are excluded from edges, and those above *high_threshold* are considered as strong edges.
    - **Displaying Images:**
      - A Matplotlib figure is set up to display two subplots side by side.
      - The left subplot shows the original image (*img_origin*), and the right subplot shows the result of the Canny edge detection (*edges*) with a grayscale colormap.
      - Titles and axis labels are set, and the images are displayed using *plt.show()*.

In summary, the *edge_detection_canny* function reads an image, applies Gaussian blur to reduce noise, performs Canny edge detection, and displays the original and edge-detected images side by side using Matplotlib. The Canny method is popular for its ability to detect edges accurately while minimizing false positives. The user can adjust the *low_threshold* and *high_threshold* parameters to control the sensitivity of edge detection.

- **Test:**

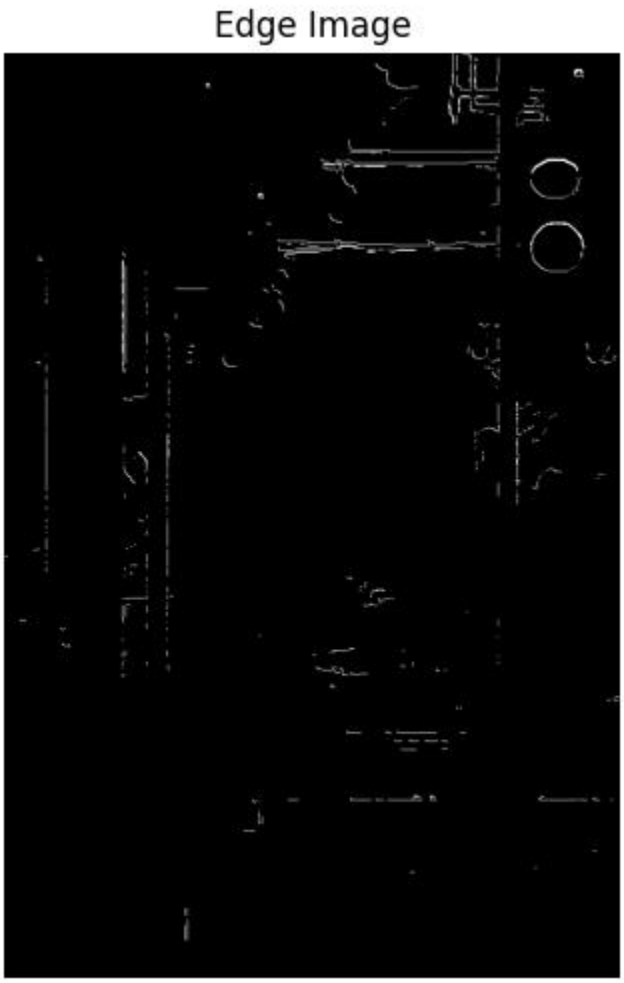


- **Implementing based on the algorithms learned**
  - **Generate a Gaussian Kernel:**
    - This function generates a 2D Gaussian kernel of a specified size and sigma.
    - It uses the *np.fromfunction* function to create the kernel based on a mathematical expression representing a 2D Gaussian distribution.
    - The resulting kernel is normalized by dividing it by the sum of its elements to ensure that the sum is equal to 1.
  - **Image Conversion to Grayscale:** If the input image has three channels (e.g., RGB), it is converted to grayscale by taking the mean along the last axis.
  - **Gaussian Blur:**
    - Gaussian blur is applied to the grayscale image using the previously defined *gaussian_kernel*.
    - The convolve function from *scipy.ndimage* is used for convolution.
  - **Compute Gradients:** Sobel filters are applied to the smoothed image to compute gradient components in the x and y directions.
  - **Compute Gradient Magnitude and Direction:** The gradient magnitude and direction are computed using the Sobel gradients.
  - **Non-maximum Suppression and Hysteresis Thresholding:**
    - Non-maximum suppression is performed to keep only the local maxima along the gradient direction.
    - Hysteresis thresholding is applied to identify strong and weak edges based on the gradient magnitude.
    - Edge tracking by hysteresis is performed to connect weak edges to strong edges.

In summary, the *canny_edge_detection* function applies the Canny edge detection algorithm to an input image. It involves steps such as Gaussian blur, gradient computation, non-maximum suppression, and

hysteresis thresholding to identify and track edges in the image. The function returns the final edge-detected image.

- **Test:**



3. Compare:

About Sobel Operator:

- **Input:**
  - The first snippet (*edge_detection*) takes an image path, reads the image, and performs edge detection. It is more focused on the end-to-end process and visualization.
  - The second snippet (*sobel_operator*) takes an image array directly and returns the gradient magnitude. It is more modular and suitable for integration into larger systems.
- **Processing:** Both snippets use the Sobel operator to compute gradients, but the first snippet includes additional steps such as Gaussian blur and visualization.
- **Output:**
  - The first snippet visually displays the original and edge-detected images using Matplotlib.
  - The second snippet returns the gradient magnitude as an array, which can be further processed or displayed separately.
- **Normalization:** The second snippet includes explicit normalization of the gradient magnitude to the range [0, 255], which is not present in the first snippet.

About Laplace of Gaussian:

- **Image Conversion:**
  - Snippet 1 assumes the input image is in color (BGR) and converts it to grayscale using *cv2.cvtColor*.

- Snippet 2 does not assume color and directly operates on the input image.

- **Gaussian Kernel Generation:**
  - Both snippets generate a 2D Gaussian filter, but Snippet 1 uses a standard function (*cv2.GaussianBlur*) to apply the Gaussian smoothing.
  - Snippet 2 generates the Gaussian kernel separately using a custom function (*gaussian_filter*) and then convolves it with the image.
- **Convolution:**
  - Both snippets perform convolution with the Laplace of Gaussian (LoG) filter, but Snippet 2 explicitly computes the second derivative of the Gaussian kernel using *np.gradient* before convolution.
  - Snippet 1 relies on the standard cv2.Laplacian function for the convolution.
- **Input Type:**
  - Snippet 1 assumes the input image is in BGR format (*color*).
  - Snippet 2 is more general and can handle grayscale or color images.
- **Conclusion:**
  - Snippet 1 is more concise and relies on the standard OpenCV functions for image processing.
  - Snippet 2 provides more flexibility and transparency, allowing customization of the Gaussian kernel and its convolution with the image.

About Canny method:

- **Input:**
  - Snippet 1 reads the image from a file path using OpenCV (*cv2.imread*).
  - Snippet 2 expects the image to be passed directly as an array.
- **Gaussian Blur:**
  - Snippet 1 applies Gaussian blur using OpenCV's *cv2.GaussianBlur*.
  - Snippet 2 defines a Gaussian kernel and applies convolution for smoothing.
- **Gradient Computation:**
  - Snippet 1 relies on the Canny edge detection function (*cv2.Canny*) for gradient computation.
  - Snippet 2 explicitly computes gradients using Sobel filters and performs additional steps like non-maximum suppression and hysteresis thresholding.
- **Display:**
  - Snippet 1 includes a Matplotlib plot for visualizing the original and Canny edge-detected images.
  - Snippet 2 returns the final edge-detected image as an array.
- **Conclusion:**
  - Snippet 1 is more concise and suitable for quick visualization and testing.
  - Snippet 2 provides more control over the edge detection process, allowing customization of Gaussian smoothing and gradient computation.

4. Limitations and errors:

About Sobel Operator:

- **Sensitivity to Noise:** The Sobel operator is sensitive to noise in the image. Since it calculates derivatives, noise can be amplified, leading to false edges.
- **Edge Thickness:** The Sobel operator tends to produce thick edges. The gradient magnitude may represent a range of edge strengths, and the thresholding step might result in thicker edges.
- **Fixed Kernel Size:** The Sobel operator uses a fixed 3x3 kernel size. While this is common, it might not be suitable for all images or applications. Some edges may be better detected with a larger or smaller kernel.
- **Limited Directional Sensitivity:** Sobel operator is designed to detect edges in the horizontal and vertical directions. It might not perform as well for edges that are not aligned with these directions.
- **Computationally Intensive:** The Sobel operator involves convolutions and square root calculations, which can be computationally intensive, especially for large images.
- **Noisy Gradient Direction:** The Sobel operator does not explicitly provide information about the direction of the edges. Additional steps are needed if information about edge orientation is required.

About Laplace of Gaussian:

- **Kernel Size Selection:** The size of the Gaussian kernel is determined based on the provided sigma. While a factor of 6 is commonly used, the choice of this factor may not be optimal for all images or applications.
- **Gaussian Smoothing:** The code relies on a 2D Gaussian filter for smoothing the image before applying the Laplacian. Gaussian smoothing can blur the image, and the extent of blurring is influenced by the choice of sigma.
- **Double Gradient Calculation:** The code calculates the double gradient of the Gaussian filter along both axes. While this can enhance the response to edges, it also makes the computation more complex.
- **Convolution Operation:** The convolution operation involves a double gradient operation on the Gaussian filter, followed by convolution with the input image. This can be computationally expensive, especially for large images.
- **Boundary Handling:** The code uses 'constant' mode for convolution, filling values outside the image boundary with a constant value. This approach may not be suitable for all images, and alternative boundary-handling methods might be considered.
- **Normalization:** The code does not normalize the Laplacian of Gaussian filter. Normalization is essential to ensure that the filter response is not affected by the filter size.
- **Fixed Cutoff Frequency:** The Laplacian of Gaussian filter has a fixed cutoff frequency determined by the choice of sigma. This may limit its ability to adapt to different image content and features.
- **Sensitivity to Noise:** Like any edge detection method, the Laplacian of Gaussian operator is sensitive to noise in the image. Noise can be amplified, leading to false positives.
- **No Non-maximum Suppression:** The code does not include a non-maximum suppression step, which is commonly used in edge detection methods to thin down the edges to one-pixel thickness.

About Canny method:

- **Fixed Kernel Sizes:** The code uses fixed kernel sizes for both the Gaussian blur and the Sobel filters. While these values are common starting points, they may not be optimal for all images or applications.
- **Fixed Thresholds:** The code requires manual setting of low and high thresholds for edge detection. Choosing appropriate thresholds can be challenging and may require experimentation or additional techniques.
- **Hardcoded Parameters:** Parameters such as kernel sizes and thresholds are hardcoded in the function, limiting the adaptability of the algorithm to different types of images or scenarios.
- **Sensitivity to Noise:** Like any edge detection method, the Canny edge detector is sensitive to noise. Gaussian smoothing is applied to mitigate this, but the choice of the smoothing kernel and the sigma parameter can affect performance.
- **Computationally Intensive:** The Canny edge detection algorithm involves multiple convolution operations, gradient computations, and thresholding steps, making it computationally intensive, especially for large images.
- **Noisy Gradient Direction:** The code does not explicitly provide information about the direction of the edges. If edge orientation information is needed, additional steps may be required.
- **Edge Thickness:** The Canny algorithm may produce thick edges, especially if there is a range of edge strengths in the image. Additional post-processing steps may be needed to thin down the edges.
- **Limited for Non-Photographic Images:** The algorithm is primarily designed for photographic images. It may not perform as well on non-photographic images or images with unique characteristics.
- **Edge Tracking Complexity:** The hysteresis thresholding and edge tracking by hysteresis involve additional complexity. The effectiveness of these steps can be sensitive to the choice of parameters.

5. References:

Gaussian Filter
Sobel Edge Detector
Sobel Filter Tutorial (with sample code and implementation)
Laplace of Gaussian (Theoretical)
Laplace of Gaussian (Implementation)
Canny Edge Detection (Tutorial, using OpenCV)
Canny Edge Detection (Implementation)