HCMUS - VNUHCM / FIT / Computer Vision & Cognitive Cybernetics Department

Digital image & video processing - LQN

**Student ID:** 21127690

**Student name:** Ngo Nguyen Thanh Thanh

**Report: EDGE DETECTION**

**I. Evaluation summary:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | | | **Requirement Met(%)** | **Notes** |
| **Implementation** | Gradient Operator | Robert Operator | 100% |  |
| Sobel Operator | 100% |  |
| Frei-Chen Operator | 100% |  |
| Prewitt Operator | 100% |  |
| Laplace Operator | | 100% |  |
| Laplace of Gaussian | | 100% |  |
|  |
|  |
|  | Canny | | 100% |  |
| **Total:** | |  | 100% |  |

1. **List of features:**

**List of Functions:** a summary of the key functions in the program:

**The program with proof images:**

**1. Image Handling**

* **read\_image(image\_path)**: Loads and converts the image to grayscale.
* **gaussian\_blur(image, kernel\_size, sigma)**: Applies Gaussian blur to reduce noise.
* **measure\_time(function, image)**: Measures the execution time of a function.

**2. Gradient Operators**

* **sobel\_operator(image):** Custom Sobel operator for edge detection.
* **prewitt\_operator(image):** Custom Prewitt operator for edge detection.
* **robert\_operator(image):** Custom Robert operator for edge detection.
* **frei\_chen\_operator(image):** Custom Frei-Chen operator for edge detection.

**3. Laplace Operators**

* **laplace\_operator(image)**: Custom Laplace operator for edge detection.
* **laplace\_of\_gaussian(image)**: Combines Gaussian blur with Laplace for edge detection.

**4. Canny Edge Detection**

canny\_custom(image, sigma\_values): Full 7-step Canny edge detection implementation.

* **sobel\_operator\_forcanny(image):** Computes gradient for Canny edge detection.
* **non\_maximum\_suppression(magnitude, angle):** Thins edges using NMS.
* **double\_threshold(nms, low\_threshold, high\_threshold): Classifies** edges as strong, weak, or non-edge.
* **edge\_tracking\_by\_hysteresis(result):** Tracks edges to connect weak edges to strong ones.
* **feature\_synthesis(edges\_list):** Combines edge maps from different sigma values.
* **select\_thresholds(image):** Allows manual selection of threshold values for Canny.

**5. Menu and Control**

* **menu()**: Displays the main menu to select edge detection methods.
* **gradient\_menu()**: Allows selection of Gradient Operators (Sobel, Prewitt, Robert, Frei-Chen).
* **main()**: Main program loop to run edge detection and display results.

**Image proof:**

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A computer screen shot of text

Description automatically generated

A computer screen shot of a program code

Description automatically generated

A screen shot of a computer code

Description automatically generated

A computer screen shot of a program code

Description automatically generated

A computer screen shot of text

Description automatically generated

A computer screen shot of text

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

**III. Summarization of the usage**

1. **Image Handling Functions**
   * **read\_image(image\_path)**: Used to load and convert an input image to grayscale for edge detection. Essential for preprocessing the image before applying edge detection algorithms.
   * **gaussian\_blur(image, kernel\_size, sigma)**: Reduces noise by smoothing the image, which is crucial for all edge detection methods, especially Canny.
   * **measure\_time(function, image)**: Used to calculate the runtime of each edge detection function to analyze performance.
2. **Gradient Operator Functions**
   * **sobel\_operator(image)**: Detects edges by calculating gradients along x and y axes. Used to highlight vertical and horizontal edges.
   * **prewitt\_operator(image)**: Similar to Sobel, but simpler and less sensitive to noise. Used for detecting vertical and horizontal edges.
   * **robert\_operator(image)**: Detects edges using a smaller 2x2 kernel, providing sharper edge detection but more sensitive to noise.
   * **frei\_chen\_operator(image)**: An advanced version of Prewitt with better diagonal edge detection. Used for detecting edges at multiple angles.
3. **Laplace and LoG Functions**
   * **laplace\_operator(image)**: Detects edges by calculating second-order derivatives, identifying regions with rapid intensity change.
   * **laplace\_of\_gaussian(image)**: Combines Gaussian smoothing with Laplacian edge detection to reduce noise before detecting edges.
4. **Canny Edge Detection Functions**
   * **canny\_custom(image, sigma\_values)**: Full implementation of the 7-step Canny edge detection process, which includes Gaussian blur, gradient calculation, NMS, double thresholding, edge tracking, and synthesis of multiple scales.
   * **sobel\_operator\_forcanny(image)**: Calculates gradient magnitude and direction as part of the Canny process.
   * **non\_maximum\_suppression(magnitude, angle)**: Thins edges by suppressing non-maximum points, making edges thinner and cleaner.
   * **double\_threshold(nms, low\_threshold, high\_threshold)**: Classifies edges as strong, weak, or non-edge for edge tracking.
   * **edge\_tracking\_by\_hysteresis(result)**: Tracks weak edges and connects them to strong edges, ensuring that connected edges remain visible.
   * **feature\_synthesis(edges\_list)**: Aggregates edge maps at multiple scales into a single final edge map.
   * **select\_thresholds(image)**: Used to manually select low and high threshold values, giving users control over Canny's edge-detection sensitivity.
5. **Menu and Control Functions**
   * **menu()**: Displays a menu to select the type of edge detection method (Gradient, Laplace, LoG, or Canny) and exits the program.
   * **gradient\_menu()**: Allows users to select one of four gradient-based edge detection operators (Sobel, Prewitt, Robert, or Frei-Chen).
   * **main()**: The main control loop for running the program. It displays menus, captures user input, and calls the appropriate edge detection functions.

**IV. Implementation**:

**Description of Methods and Pseudo code**

**1. read\_image(image\_path)**

* **Purpose**: Load and convert an input image to grayscale for edge detection.
* **Details**: This function reads the image from the provided path and converts it to grayscale using OpenCV's cv2.imread() with cv2.IMREAD\_GRAYSCALE flag. Grayscale images are necessary for most edge detection algorithms.

**Pseudo code**:

FUNCTION read\_image(image\_path):

image = READ image from image\_path as grayscale

RETURN image

**2. gaussian\_blur(image, kernel\_size, sigma)**

* **Purpose**: Reduce image noise and smooth the image to prepare it for edge detection.
* **Details**: This function applies a Gaussian filter to the image using a manually calculated Gaussian kernel. It pads the image to avoid boundary issues and convolves the image with the kernel.

**Pseudo code**:

FUNCTION gaussian\_blur(image, kernel\_size, sigma):

Calculate Gaussian kernel based on kernel\_size and sigma

Pad image to prevent boundary issues

FOR each pixel in image:

Extract local region around pixel

Convolve local region with Gaussian kernel

RETURN blurred image

**3. sobel\_operator(image)**

* **Purpose**: Detect vertical and horizontal edges using the Sobel operator.
* **Details**: Applies two 3x3 Sobel kernels to compute the gradients in the x and y directions. The gradient magnitude is calculated as the Euclidean norm of the x and y components.

**Pseudo code**:

FUNCTION sobel\_operator(image):

Apply Sobel kernel in X direction to get Gx

Apply Sobel kernel in Y direction to get Gy

Calculate gradient magnitude = sqrt(Gx^2 + Gy^2)

Normalize gradient to range [0, 255]

RETURN edges

**4. prewitt\_operator(image)**

* **Purpose**: Detect edges in vertical and horizontal directions.
* **Details**: Similar to Sobel, but uses a simpler 3x3 kernel. The gradient magnitude is computed using Prewitt kernels in the x and y directions.

**Pseudo code**:

FUNCTION prewitt\_operator(image):

Apply Prewitt kernel in X direction to get Gx

Apply Prewitt kernel in Y direction to get Gy

Calculate gradient magnitude = sqrt(Gx^2 + Gy^2)

RETURN edges

**5. robert\_operator(image)**

* **Purpose**: Detect edges using Robert's cross operator (2x2 kernel).
* **Details**: Applies Robert's cross kernels to detect diagonal edges. This operator is simple but sensitive to noise.

**Pseudo code**:

FUNCTION robert\_operator(image):

FOR each pixel in image (except the last row and column):

Calculate Gx using Robert's kernel

Calculate Gy using Robert's kernel

Calculate gradient magnitude = sqrt(Gx^2 + Gy^2)

RETURN edges

**6. frei\_chen\_operator(image)**

* **Purpose**: Detect edges using the Frei-Chen operator for diagonal and vertical edges.
* **Details**: Similar to Prewitt but uses square root of 2 in the kernel to enhance diagonal edge detection.

**Pseudo code**:

FUNCTION frei\_chen\_operator(image):

Apply Frei-Chen kernel in X direction to get Gx

Apply Frei-Chen kernel in Y direction to get Gy

Calculate gradient magnitude = sqrt(Gx^2 + Gy^2)

RETURN edges

**7. laplace\_operator(image)**

* **Purpose**: Detect regions of rapid intensity change using the Laplace operator.
* **Details**: Computes the second-order derivative of the image using the Laplace kernel. It detects edge-like regions without considering edge direction.

**Pseudo code**:

FUNCTION laplace\_operator(image):

Apply Laplace kernel to image

RETURN edges

**8. laplace\_of\_gaussian(image)**

* **Purpose**: Combine Gaussian smoothing with Laplace edge detection.
* **Details**: Blurs the image using a Gaussian filter and then applies the Laplace operator to detect edges.

**Pseudo code**:

FUNCTION laplace\_of\_gaussian(image):

Blur the image using Gaussian kernel

Apply Laplace operator to the blurred image

RETURN edges

**9. canny\_custom(image, sigma\_values)**

* **Purpose**: Detect edges using the 7-step Canny edge detection algorithm.
* **Details**: Performs all 7 steps of Canny edge detection, including Gaussian blur, gradient calculation, NMS, double thresholding, and edge tracking.

**Pseudo code**:

FUNCTION canny\_custom(image, sigma\_values):

FOR each sigma in sigma\_values:

Step 1: Blur image using Gaussian filter

Step 2: Calculate gradient magnitude and direction using Sobel

Step 3: Apply Non-Maximum Suppression (NMS) to thin edges

Step 4: Apply double thresholding to classify strong, weak, and non-edges

Step 5: Perform edge tracking by hysteresis

Save edges for this scale

Step 6 & 7: Aggregate edges from multiple scales

RETURN combined edges

**10. sobel\_operator\_forcanny(image)**

* **Purpose**: Calculate gradient magnitude and direction for Canny edge detection.
* **Details**: Similar to Sobel but returns the x and y gradients separately for use in Canny.

**Pseudo code**:

FUNCTION sobel\_operator\_forcanny(image):

Apply Sobel kernel in X direction to get Gx

Apply Sobel kernel in Y direction to get Gy

Calculate gradient magnitude = sqrt(Gx^2 + Gy^2)

RETURN magnitude, Gx, Gy

**11. non\_maximum\_suppression(magnitude, angle)**

* **Purpose**: Thin edges by suppressing non-maximum pixels.
* **Details**: Uses the gradient direction to keep only the local maximum values in the gradient image.

**Pseudo code**:

FUNCTION non\_maximum\_suppression(magnitude, angle):

FOR each pixel in magnitude (ignore edges of the image):

Check if pixel is maximum along gradient direction

If not, set pixel to 0

RETURN thinned edges

**12. double\_threshold(nms, low\_threshold, high\_threshold)**

* **Purpose**: Classify pixels as strong, weak, or non-edges.
* **Details**: Pixels are categorized as strong, weak, or suppressed based on two threshold values.

**Pseudo code**:

FUNCTION double\_threshold(nms, low\_threshold, high\_threshold):

Strong pixels = pixels >= high\_threshold

Weak pixels = pixels between low\_threshold and high\_threshold

RETURN image with strong, weak, and non-edges marked

**13. edge\_tracking\_by\_hysteresis(result)**

* **Purpose**: Track edges by connecting weak edges to strong edges.
* **Details**: Connects weak edges to strong edges if they are connected by a neighboring pixel.

**Pseudo code**:

FUNCTION edge\_tracking\_by\_hysteresis(result):

FOR each weak pixel in image:

If connected to a strong edge, convert to a strong edge

RETURN final edge image

**14. feature\_synthesis(edges\_list)**

* **Purpose**: Combine edges from multiple scales.
* **Details**: Takes the maximum edge intensity from multiple edge maps to create a final edge map.

**Pseudo code**:

FUNCTION feature\_synthesis(edges\_list):

Combine edges from multiple scales using max value at each pixel

RETURN final edge map

**15. select\_thresholds(image)**

* **Purpose**: Manually select low and high thresholds for Canny edge detection.
* **Details**: Displays a histogram of pixel intensities and allows users to manually input thresholds.

**Pseudo code**:

FUNCTION select\_thresholds(image):

Display histogram of pixel intensities

Prompt user to enter low and high threshold

RETURN low\_threshold, high\_threshold

**16. menu()**

* **Purpose**: Display main menu for user to select an edge detection method.
* **Details**: Provides options for Gradient, Laplace, LoG, and Canny edge detection.

**Pseudo code**:

FUNCTION menu():

Display menu with edge detection options

RETURN user choice

**V. RESULTS:**

**A close-up of a person's face

Description automatically generated**

**A person wearing a hat and a picture of a tiger

Description automatically generated**

**A person with a hat

Description automatically generated**

**A person wearing a hat and a white hat

Description automatically generated**

**A black and white image of a person

Description automatically generated**

**A close-up of a person's face

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**A collage of images of two people

Description automatically generated**

**VI. COMPARISON**

**Comparison of Custom-Coded Algorithms vs OpenCV Implementations**

**1. Accuracy**

* **Custom Code**: Prone to slight inaccuracies due to manual normalization and kernel design.
* **OpenCV**: Highly accurate, with robust handling of edge detection using pre-tested methods.

**2. Speed**

* **Custom Code**: Slow, especially for large images due to loops and manual convolution.
* **OpenCV**: Fast, leveraging SIMD instructions and parallel processing.

**3. Customization**

* **Custom Code**: Full control to modify kernels, edge thresholds, and gradient methods.
* **OpenCV**: Limited customization but offers configurable parameters.

**4. Robustness**

* **Custom Code**: Less robust, errors often occur in gradient, NMS, and edge tracking.
* **OpenCV**: More robust, with precise border handling and cleaner edge detection.

**5. Ease of Use**

* **Custom Code**: Hard to debug and maintain, requires deep knowledge of image processing.
* **OpenCV**: Simple and reusable with one-liner functions like cv2.Canny.

**6. Error-Prone**

* **Custom Code**: Prone to logic errors, especially in NMS, border padding, and kernel issues.
* **OpenCV**: Rarely encounters errors unless used incorrectly.

**7. Maintainability**

* **Custom Code**: Harder to maintain as every part is explicitly coded.
* **OpenCV**: Easy to maintain and update with one-liner methods.