ICE2607 Lab 2: Canny Edge Detection

SJTU-SEIEE cny123222

November 20, 2024

1 Experiment Overview (实验概览)

Lab2 primarily focuses on **Canny edge detection**. Edge detection aims to identify areas in an image where the intensity changes abruptly, often represented as sharp transitions in grayscale values within a small neighborhood. Canny edge detection, being a cornerstone in the field of image processing, involves the following steps:

- **Gaussian Smoothing**: Using a Gaussian filter to smooth the image and reduce noise.
- **Gradient Calculation**: Computing the gradient magnitude and direction for each pixel in the image.
- **Non-Maximum Suppression**: Applying non-maximum suppression to eliminate spurious responses.
- **Double Thresholding**: Employing double thresholding to differentiate between real and potential edges.
- **Edge Tracking by Hysteresis**: Suppressing isolated weak edges to finalize the edge detection.

In the experiment, three images are utilized to compare detection performance by varying double thresholds and gradient magnitude operators. Additionally, we incorporate **adaptive threshold selection based on Otsu's method** as part of the experimentation.

2 Solution Approach (解决思路)

2.1 Step 1: Grayscale Conversion

When working with color images, the primary step in detection involves converting them to grayscale. There are two common methods for grayscale conversion:

- Method 1: Compute grayscale as the average of the red, green, and blue channels using $\text{Gray} = (\text{R} + \text{G} + \text{B})/3$
- Method 2: Calculate grayscale by considering human visual perception with $\text{Gray} = 0.299\text{R} + 0.587\text{G} + 0.114\text{B}$

These methods aim to transform RGB color images into single-channel grayscale representations, a fundamental preprocessing step for edge detection. Taking Method 2 as an example, the implementation pseudocode is as follows:

```
def to_grayscale(img): # img.shape = (H, W, 3)
    weights = [0.299, 0.587, 0.114]
    new_img = np.dot(img, weights).astype(np.uint8) # img.shape = (H, W)
    return new_img
```

2.2 Step 2: Gaussian Blurring

After converting the image to grayscale, the next step in the edge detection process involves applying Gaussian blurring. Gaussian blurring helps to smooth the image and reduce noise, which is crucial for accurate edge detection.

The Gaussian smoothing filter is defined mathematically as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

Here, σ represents the standard deviation of the filter, dictating the extent of smoothing applied to the image. The resulting smoothed image is obtained by convolving the original image $I(x, y)$ with the Gaussian kernel:

$$I'(x, y) = G(x, y) * I(x, y)$$

Through this convolution process, high-frequency noise is attenuated while retaining the crucial structural details within the image. The implementation pseudocode is provided below:

```
def Gaussian_blur(img, kernel_size, sigma):
    kernel = create_kernel(kernel_size, sigma)
    new_img = convolution(img, kernel).astype(np.uint8)
    return new_img
```

2.3 Step 3: Gradient Calculation

In the Gradient Calculation step, we can approximate the gradient of the image's grayscale values using first-order finite differences. This approximation enables us to derive two matrices representing the image's partial derivatives along the x and y axes.

Taking the **Sobel** gradient operator as an example, the convolution kernels for the x-axis and y-axis operators are defined as:

$$s_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad s_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

By convolving the image with these kernels, we obtain the derivative matrices in the x-direction (P_x) and y-direction (P_y). Subsequently, we calculate the gradient magnitude and direction using the following formulas:

$$M[i,j] = \sqrt{P_x[i,j]^2 + P_y[i,j]^2}$$
$$\theta[i,j] = \arctan\left(\frac{P_y[i,j]}{P_x[i,j]}\right)$$

Below is the refined implementation pseudocode for this process:

```
def grad_calc(img, s_x, s_y): # s_x and s_y are convolution kernels
    grad_x = convolution(img, s_x)
    grad_y = convolution(img, s_y)
    grad_mag = np.hypot(grad_x, grad_y) # gradient magnitude
    grad_ang = np.arctan2(grad_y, grad_x) # gradient direction
    return grad_mag, grad_ang
```

2.4 Step 4: Non-Maximum Suppression

Non-maximum suppression is a crucial step in edge detection that involves identifying local maxima and setting the grayscale values of non-maximum points to zero. This process significantly improves the accuracy of edge detection by filtering out non-edge points.

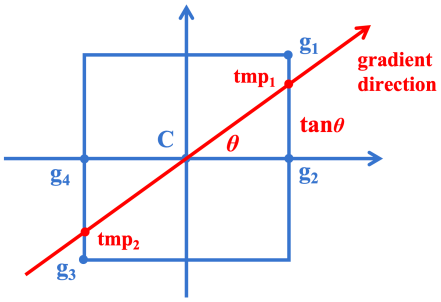


Figure 1 Diagram of Non-Maximum Suppression

The principle of non-maximum suppression can be elucidated by examining the gradient direction at a specific point, as illustrated in Figure 1. The red line in the figure symbolizes the gradient direction at point C, with local maxima surrounding C aligning along this line. Through linear interpolation to determine the gradients at points $tmp1$ and $tmp2$, we ascertain whether the grayscale value at point C is less than either tmp1 or tmp2. If this condition holds true, indicating that C is not on an edge, the grayscale value of C is then suppressed to zero.

For instance, when $0 < \theta < 45°$, the gradient magnitude of tmp1, denoted as $M(tmp1)$, and tmp2, denoted as $M(tmp2)$, can be computed as follows:

$$M(tmp1) = \tan\theta \cdot M(g_1) + (1 - \tan\theta) \cdot M(g_2)$$
$$M(tmp2) = \tan\theta \cdot M(g_3) + (1 - \tan\theta) \cdot M(g_4)$$

Similar calculations can be performed for other gradient directions. Below is a refined pseudocode for the non-maximum suppression process:

```python
def non_max_suppression(img):
    new_img = img.copy() # deepcopy may be needed in real situations
    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            theta = grad_ang[i, j] # get gradient direction
            if 0 < theta < np.pi / 4:
                g1, g2, g3, g4 = img[i-1, j+1], img[i, j+1], img[i+1, j-1], img[i, j-1]
                tmp1 = np.tan(theta) * g1 + (1 - np.tan(theta)) * g2
                tmp2 = np.tan(theta) * g3 + (1 - np.tan(theta)) * g4
            # Handle other gradient directions similarly
            if img[i, j] < tmp1 or img[i, j] < tmp2:
                new_img[i, j] = 0 # set non-maximum points to zero
    return new_img
```

2.5 Step 5: Double Thresholding and Edge Tracking

In the Canny edge detection algorithm, reducing the number of false edges is achieved through the utilization of a double-threshold method.

- If the value of a point exceeds the high threshold, it is classified as a strong edge point and is recognized as part of the edges.
- If the value of a point falls below the low threshold, it is not considered part of the edges.
- If the value of a point is between the low and high thresholds, it is termed a weak edge point. If there exist neighboring points that are strong edge points, it transitions to a strong edge point.

In the double thresholding process, the high threshold helps reduce false edges, while the low threshold aids in closing edge contours. Below is the pseudocode illustrating this process:

```python
def double_threshold(img, th_low, th_high):
    new_img = np.zeros_like(img, dtype=np.uint8)
    edge_x, edges_y = np.where(img > th_high) # Select strong edge points
    edges = list(zip(edge_x, edge_y)) # "edges" stores edge points that require processing
    while len(edges) > 0:
        x, y = edges.pop()
        new_img[x, y] = 255 # Mark as an edge point
        for new_x, new_y in neighbour_points(x, y):
            if th_low < img[new_x, new_y] < th_high and new_img[x, y] != 255:
            # (new_x, new_y) is a weak edge point that we haven't seen
                new_img[new_x, new_y] = 255
                edges.append((new_x, new_y))
    return new_img
```

3 Experimental Results (实验结果)

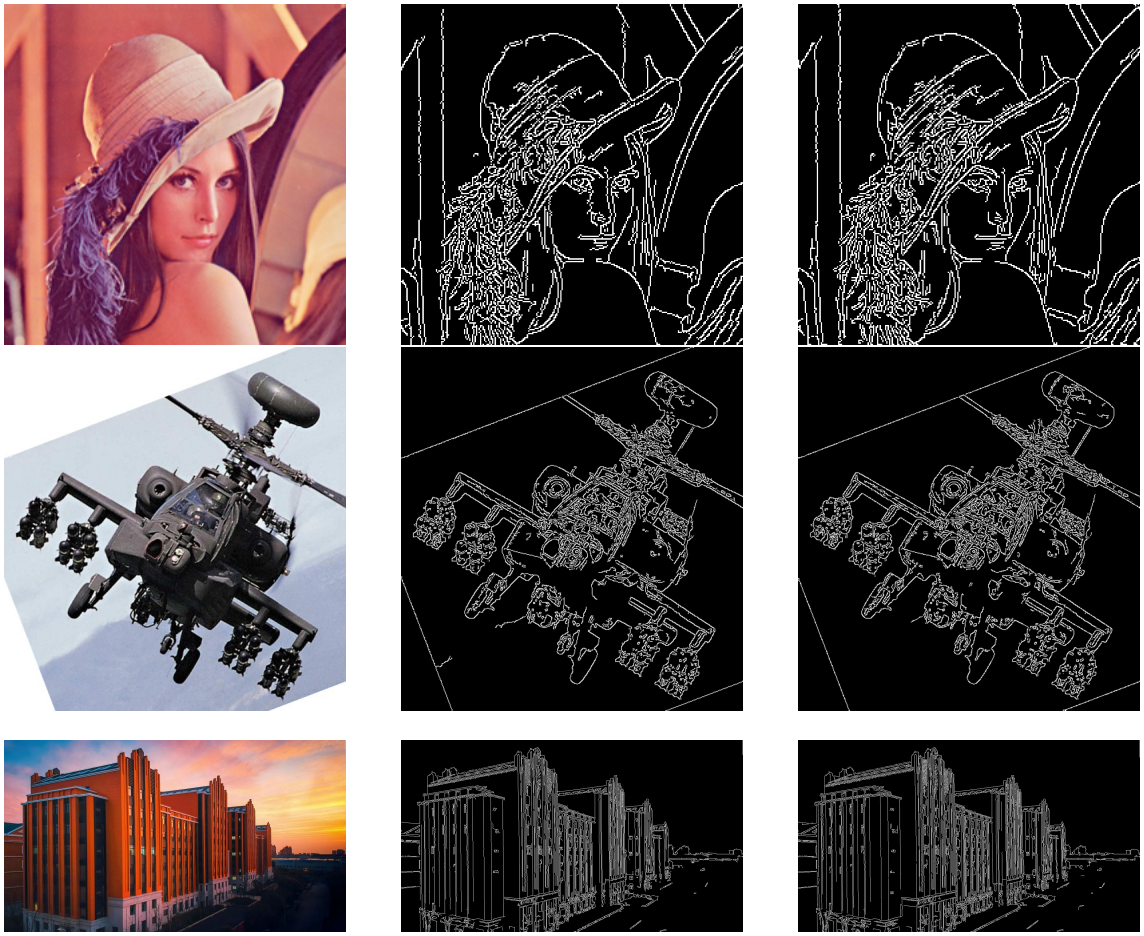
We conducted experiments on three images rich in edge features, utilizing varied thresholds and gradient operators to assess the efficacy of edge detection.

3.1 Experimental Environment

The experiments were conducted on macOS Sonoma 14.6.1 with Python 3.13.0. Key libraries included OpenCV-Python 4.10.0.84, Numpy 2.1.3 and Matplotlib 3.9.2.

3.2 Detection Results

Figure 2 displays the edge detection outcomes from both OpenCV's Canny algorithm and our customized Canny implementation. The similarity between the results is apparent. (Our Canny utilizes a sigma of 0.2, the Sobel operator, a low threshold of 40, and a high threshold of 100.) In the subsequent section, we will delve into the implications of employing different operators and thresholds.



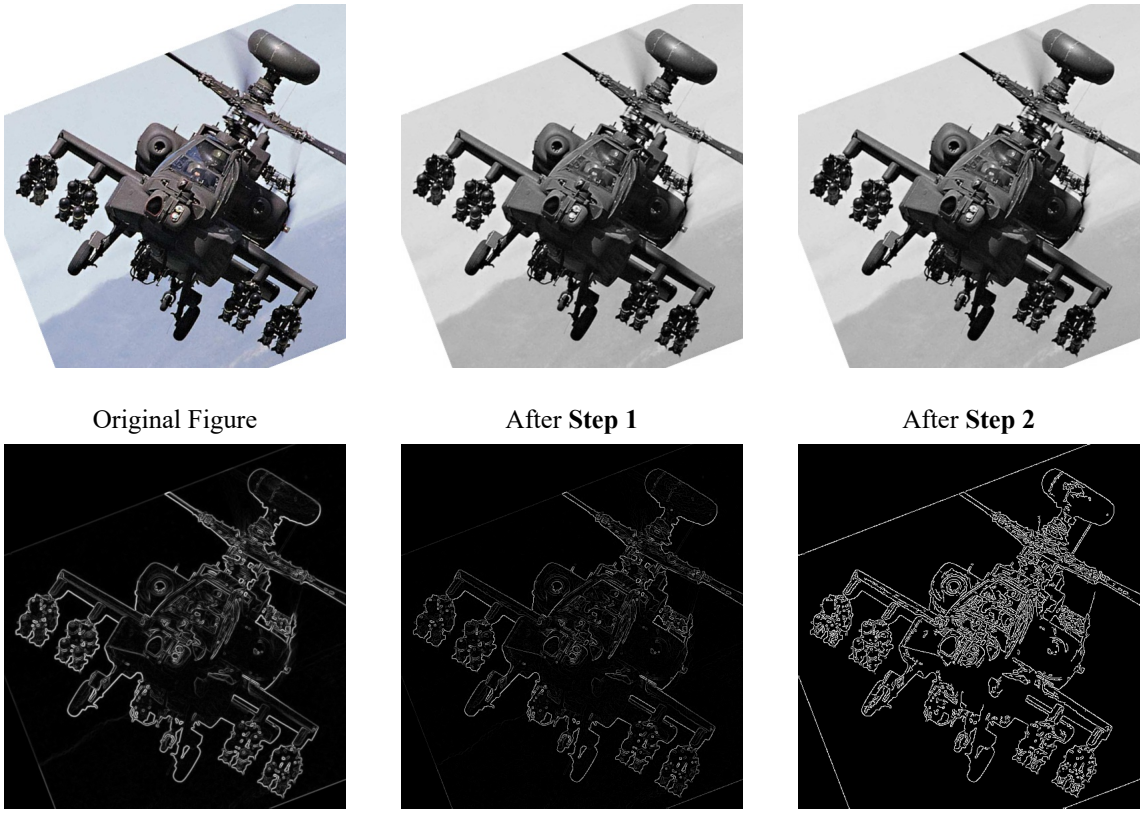
Original OpenCV-Canny My-Canny

Figure 2 Resulting Images of Canny Edge Detection

4 Analysis and Discussion (分析与思考)

4.1 What Happened behind each Step?

To gain a deeper insight into the processes underlying the Canny algorithm, we showcase the appearance of the second image after each operational step in Figure 3.



Original Figure | After **Step 1** | After **Step 2**

After **Step 3** | After **Step 4** | After **Step 5**

Figure 3 Images after each Step

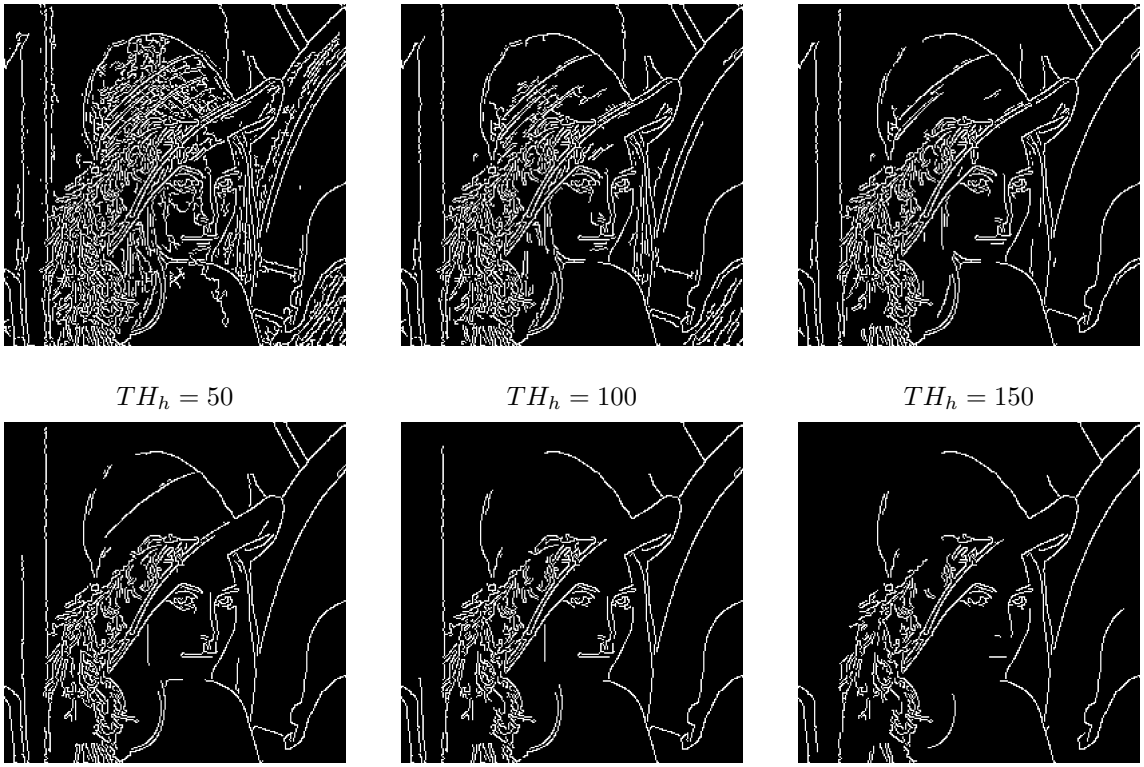
The transformations of the image align with our expectations. Following **Step 1** (Grayscale Conversion), the image transitions into grayscale. Subsequently, after **Step 2** (Gaussian Blurring), the image exhibits a slight blur. **Step 3** (Gradient Calculation) results in lighter areas near edges. After **Step 4** (Non-Maximum Suppression), the image dims, with edges becoming a single pixel wide. Finally, after **Step 5** (Double Thresholding and Edge Tracking), the detected edges are prominently displayed.

4.2 Changing Thresholds

To investigate the impact of varying thresholds on the effectiveness of Canny edge detection, we utilized the first image (Lenna) as a benchmark, varied the high threshold values to 50, 100, 150, 200, 250, and 300, while keeping the low threshold at 0.4 times the high threshold.

By examining the resulting images in Figure 4, it becomes evident that the choice of threshold significantly impacts the edge detection outcome. Lower thresholds lead to an increase in the number of displayed edges, but also introduce more false edges. Conversely, higher thresholds reduce the number of detected edges, but also lead to an increase in undetected edges and incomplete edge closures. Therefore, finding an

optimal threshold is crucial for achieving accurate edge detection results.



$TH_h = 50$ $TH_h = 100$ $TH_h = 150$

$TH_h = 200$ $TH_h = 250$ $TH_h = 300$

Figure 4 Effects of Different Thresholds

4.3 Changing Gradient Operators

After assessing the impact of various thresholds on Canny edge detection, we proceeded to evaluate the effects of different operators on the detection outcomes. Initially employing the Sobel operator, we further tested the Roberts, Prewitt, and Canny operators on the Lenna image. The results are shown in Figure 5. Our observations indicated that the results across these operators were relatively similar.

Roberts Operator ($TH_h = 25$) Prewitt Operator ($TH_h = 80$) Canny Operator ($TH_h = 40$)

Figure 5 Effects of Different Gradient Operators

However, it is crucial to recognize the theoretical distinctions among these operators. They differ in their noise reduction capabilities, sensitivity to various types of edges, and other key characteristics.

4.4 Adaptive Thresholds (自适应阈值)

Since threshold settings are crucial for accurate edge identification in the Canny algorithm, the adoption of adaptive thresholds becomes essential. These thresholds adjust dynamically, eliminating the need for manual intervention and streamlining the detection process.

One effective method for determining adaptive thresholds is the **Otsu method**(大津法). This technique automatically selects the optimal threshold value in image processing by maximizing the inter-class variance between foreground and background pixels.

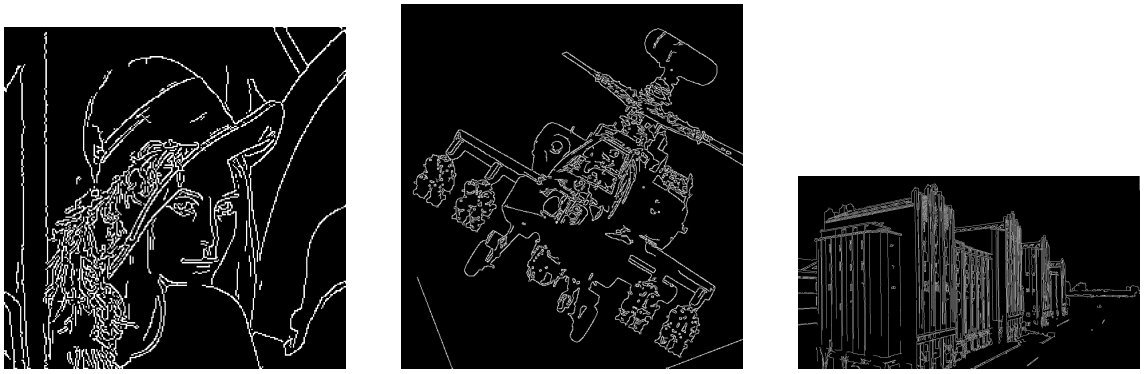


Figure 6 Canny Edge Detection Using Adaptive Thresholds

Due to the complexity of the Otsu method, its implementation has been omitted in this paper. The images produced in Figure 6 demonstrate the effectiveness of adaptive thresholds. Integration of the Otsu method within the Canny algorithm enables automatic threshold adjustments based on image characteristics, simplifying the process and enhancing adaptability to diverse image complexities.

4.5 Highlights of Codes (代码创新点)

- **Object-Oriented Approach:** Utilizing an object-oriented design to enhance code readability and maintainability.
- **Modularity:** Embracing a modular approach inspired by **PyTorch** for streamlined construction and parameter selection.
- **Command-Line Interface:** Implementation of a command-line interface allowing for the input of various parameters, enhancing versatility, usability, and cross-system compatibility.

5 Reflection and Conclusion (实验感想)

5.1 Learning from Experiments

The experiments have equipped us with a fundamental understanding of the Canny edge detection process, parameter selection, and enhanced LaTeX proficiency, thereby enriching our skill set and nurturing creativity.

5.2 Challenges Encountered

During the image processing phase, one significant challenge was the necessity to carefully consider data types to avoid unexpected issues. Another obstacle involved determining edge closure during edge linking, a task both complex and computationally intensive. To mitigate this, omitting closure checks can be a viable solution.

5.3 Conclusion

During the experiment, we employed Canny edge detection to identify edges in three images, examining the effects of varying thresholds and gradient operators. Furthermore, we integrated the Otsu method for adaptive threshold selection, thereby improving convenience in edge detection.

6 Reference

https://github.com/khushitejwani/Canny-Edge-Detection-Using-Otsu-Thresholding