|  |  |
| --- | --- |
| Name: | Jason Rich Darmawan |
| Student ID: | 2024280038 |

# Introduction

Edges in an image appear where there is a sudden change in pixel intensity. To illustrate this, consider a grayscale image, which consists of two dimensions—rows and columns. An RGB image, on the other hand, has an extra third dimension for color channels. In this example, we analyze an image of a dog to observe the difference in pixel intensity between the dog and the background (see Figure 1.1). As shown in the figure, the intensity changes quickly between the background and the dog, but it is still difficult to pinpoint the exact edges.

A collage of images and graphs

Description automatically generated

Figure 1.1 Pixel Intensity

To detect edges, we calculate the derivative of the image. The derivative shows how quickly the pixel values are changing, and edges occur where the change is the greatest (see Figure 1.2). One simple way to do this is by using the forward difference, backward difference and central difference, which measure changes in pixel values by comparing neighboring pixels in horizontal or vertical directions.

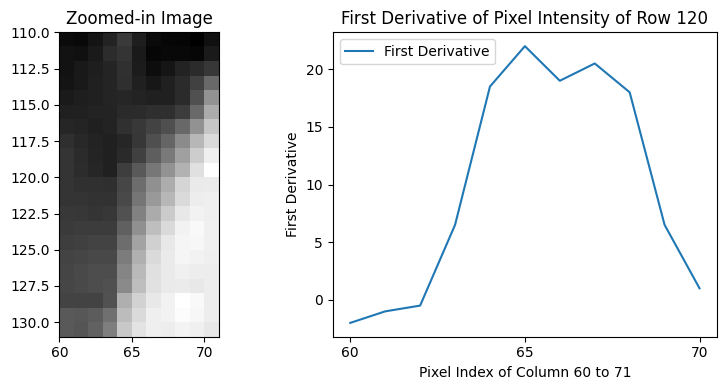


Figure 1.2 First Derivative of Pixel Intensity

To calculate the change of pixel intensity between neighboring pixels at the borders, we use the forward/backward difference method as shown in Eqs. (1.1) and (1.2):

|  |  |  |
| --- | --- | --- |
|  |  | (1.1) |

|  |  |  |
| --- | --- | --- |
|  |  | (1.2) |

For interior pixels, we use the central difference method as shown in Eqs. (1.3) and (1.4):

|  |  |  |
| --- | --- | --- |
|  |  | (1.3) |

|  |  |  |
| --- | --- | --- |
|  |  | (1.4) |

For brevity, Eq. (1.1) computes the difference between the intensity of the pixel at and , where represents the pixel intensity at position .

By combining the gradients from both directions, we calculate the magnitude of intensity changes, which highlights the edges using the following equation: . In images with low noise, such as our example of a dog in an open field, these simple methods can work well and produce results similar to the Sobel operator (see Figure 1.3). However, the Sobel operator includes a smoothing effect (see Figure 1.4), which makes it more robust in images with more noise or complex textures, where small random variations in pixel intensity could interfere with edge detection.

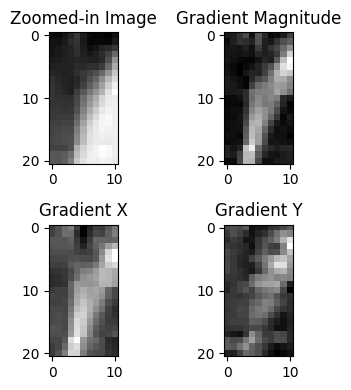


Figure 1.3 With First Derivatives

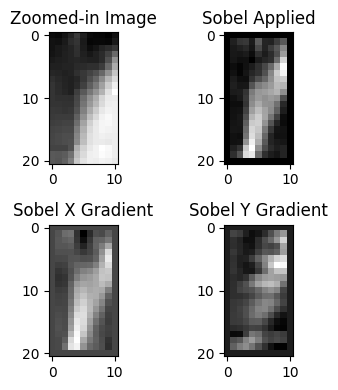


Figure 1.4 With Sobel Derivatives

To improve edge detection, we apply the Sobel operator, which is specifically designed to detect edges more reliably. The Sobel operator combines two key ideas: it approximates the derivative by measuring intensity changes, but it also smooths out noise by using a weighted 3x3 kernel. Additionally, it uses two separate kernels for detecting edges in both the horizontal and vertical directions. This combination of smoothing and directional sensitivity makes the Sobel operator more effective at detecting edges, even in noisy images (see Figure 1.5).

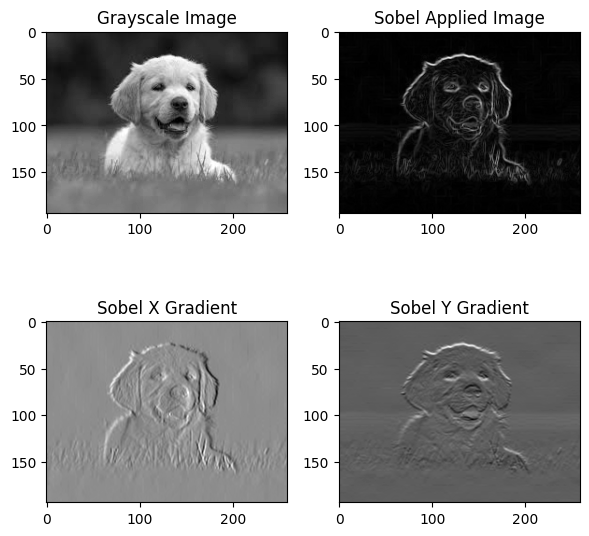


Figure 1.5 Sobel Applied

To calculate the horizontal changes, we convolve the pixel intensity with a kernel as shown in Eq. (1.5):

|  |  |  |
| --- | --- | --- |
|  |  | (1.5) |

To calculate the vertical changes, we convolve the pixel intensity with a kernel as shown in Eq. (1.6):

|  |  |  |
| --- | --- | --- |
|  |  | (1.6) |

For brevity, Eq. (1.5) applies a convolution kernel to the image to compute the gradient in the horizontal direction, where refers to the region of the image used in the convolution.

# Python Scripting with Explanation

The code is available at <https://github.com/jasonrichdarmawan/npu-Python-Programming-Fundamentals-and-Abaqus-Scripting-report>

# Import the required Python packages

import numpy as np

import numpy.typing as npt

import matplotlib.pyplot as plt

def rgb2Gray(src: npt.ArrayLike) -> tuple[npt.ArrayLike, int]:

# Convert the image to grayscale

## Extract the Red, Green, and Blue channels

R = img[:, :, 0]

G = img[:, :, 1]

B = img[:, :, 2]

## Apply the grayscale conversion formula

## Based on https://docs.opencv.org/3.4/de/d25/imgproc\_color\_conversions.html#color\_convert\_rgb\_gray

gray\_img = (0.299 \* R + 0.587 \* G + 0.114 \* B).astype(np.uint8)

# Check if image is loaded fine

if src is None:

print ('Error opening image')

return np.empty((0,0), dtype=np.uint8), -1

return gray\_img, 0

# kernel $G\_x$

SOBEL\_X = np.array([[-1, 0, 1],

[-2, 0, 2],

[-1, 0, 1]])

# kernel $G\_y$

SOBEL\_Y = np.array([[-1, -2, -1],

[ 0, 0, 0],

[ 1, 2, 1]])

def sobel(src: npt.ArrayLike) -> tuple[dict[str, npt.ArrayLike], int]:

# Check if image is loaded fine

if src is None:

print ('Error opening image')

return {}, -1

gray\_img, err = rgb2Gray(src) # matrix

if err != 0:

print('Error converting image to grayscale')

return {}, -1

# Initialize the result matrix

sobel\_applied = np.zeros\_like(gray\_img)

# Apply the Sobel operator to each 2D slice

grad\_x = np.zeros\_like(gray\_img, dtype=np.float64) # To avoid overflow

grad\_y = np.zeros\_like(gray\_img, dtype=np.float64)

# Convolve with Sobel x and y operators

# To avoid boundary issues, we skip the first and last rows and columns

# The Sobel operator requires a 3x3 region around each pixel to compute the gradient,

# and this cannot be done for the pixels at the edges of the image.

for row in range(1, gray\_img.shape[0] - 1):

for col in range(1, gray\_img.shape[1] - 1):

region = gray\_img[row-1:row+2, col-1:col+2]

grad\_x[row, col] = np.sum(region \* SOBEL\_X) # Element-wise multiplication and summation

grad\_y[row, col] = np.sum(region \* SOBEL\_Y)

# Compute the gradient magnitude

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

return {

'gray\_img': gray\_img,

'grad\_x': grad\_x,

'grad\_y': grad\_y,

'sobel\_applied': sobel\_applied,

}, 0

ok, err = sobel(img)

# Plot the original and Sobel-applied matrices

fig, axes = plt.subplots(2, 2, figsize=(6, 6))

axes[0, 0].imshow(ok['gray\_img'], cmap='gray')

axes[0, 0].set\_title(f'Grayscale Image')

axes[0, 1].imshow(ok['sobel\_applied'], cmap='gray')

axes[0, 1].set\_title(f'Sobel Applied Image')

axes[1, 0].imshow(ok['grad\_x'], cmap='gray')

axes[1, 0].set\_title(f'Sobel X Gradient')

axes[1, 1].imshow(ok['grad\_y'], cmap='gray')

axes[1, 1].set\_title(f'Sobel Y Gradient')

plt.tight\_layout()

plt.show()

# Results and Discussion

As illustrated in Figure 1.3 and Figure 1.4, the first derivative method directly captures the intensity changes between neighboring pixels but does not apply any smoothing. This lack of smoothing can result in sharper but potentially noisier edge detection. In contrast, the Sobel derivative introduces a smoothing effect due to the weighted averaging in its kernel. This results in edges that are less sensitive to noise, providing a cleaner and more refined edge detection, as shown in the figures.

# Conclusion

In this report, we compared the performance of the first derivative method and the Sobel operator for edge detection in digital images. The result demonstrated that while the first derivative method effectively detects edges, it lacks a smoothing effect, making it more sensitive to noise. In contrast, the Sobel operator incorporates a smoothing effect due to its weighted kernel, producing cleaner and more refined edges. This makes Sobel more suitable for edge detection in noisy or complex images.