FILTERS AND EDGE DETECTION REPORT

Ziad Essam

1. Sobel Filter

Introduction to the Sobel filter

The Sobel filter is a widely used technique in image processing for edge detection. It is particularly effective at highlighting sudden changes in pixel intensity, which correspond to edges within an image. By detecting these edges, the Sobel filter helps in identifying boundaries and structures within an image, making it a fundamental tool in computer vision.

How the Sobel filter works

The Sobel filter uses convolution with specific kernels to detect edges in an image. It operates with two distinct kernels:

- One kernel approximates the intensity change in the horizontal (x) direction.
- The other kernel approximates the intensity change in the vertical (y) direction.

In simple terms, these kernels estimate the gradient of the intensity values at each pixel, which is a way of measuring how quickly the pixel values are changing. The gradient is essentially the multi-dimensional equivalent of the derivative or slope discussed earlier.

These two kernels are applied to each pixel in the image through the convolution operation, which helps identify regions where the intensity change is greatest in both the x and y directions. This process effectively highlights the edges within the image, making the Sobel filter a powerful tool for detecting boundaries and features in various computer vision applications.

Applications of the Sobel Filter

The applications of the Sobel filter are vast and encompass fields ranging from computer vision to medical imaging. It is often employed in object detection tasks, image segmentation, and even in facial recognition algorithms.

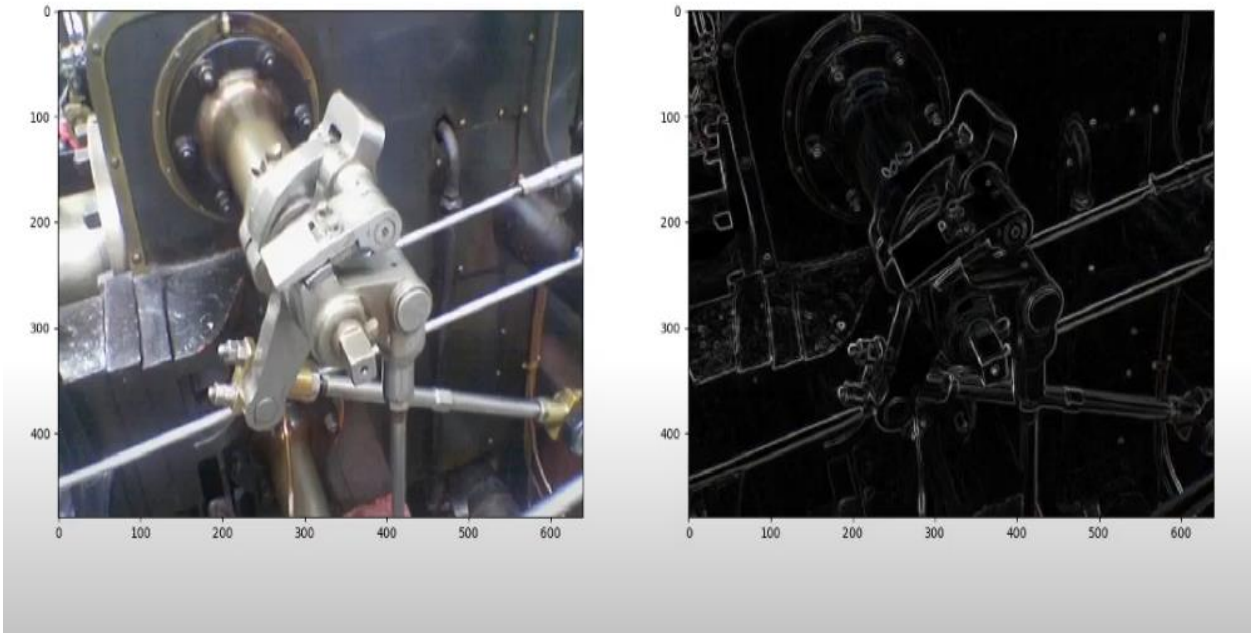
Sobel filter is mainly used to detect edges, as finding edges is a fundamental problem in image processing as edges define object boundaries and represent important structural properties in an image.

Example Code for Implementing the Sobel Filter

Code**:**

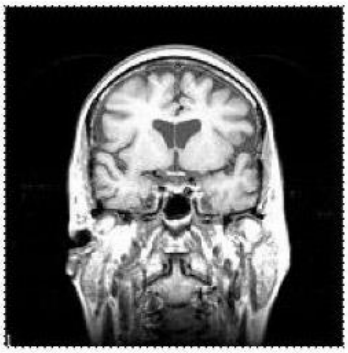
```
import numpy as np
from matplotlib.image import imread
from scipy import ndimage
import matplotlib.pyplot as plt

# Here we read the image and bring it as an array
original_image = imread('Images/original_image.PNG')

# Next we apply the Sobel filter in the x and y directions to then calculate the output image
dx, dy = ndimage.sobel(original_image, axis=0), ndimage.sobel(original_image, axis=1)
sobel_filtered_image = np.hypot(dx, dy)  # is equal to ( dx ^ 2 + dy ^ 2 ) ^ 0.5
sobel_filtered_image = sobel_filtered_image / np.max(sobel_filtered_image)  # normalization step

# Display and compare input and output images
fig = plt.figure(1)
ax1, ax2 = fig.add_subplot(121), fig.add_subplot(122)
ax1.imshow(original_image)
ax2.imshow(sobel_filtered_image, cmap=plt.get_cmap('gray'))
plt.show()
```

Results:


0
100
200
300
400
0
100
200
300
400
500
600
0
100
200
300
400
0
100
200
300
400
500
600

2. Laplacian Filter

Introduction to the Laplacian filter



(A) Original MR image

(B) Laplacian results

(C) Extraction of the zero crossing of the Laplacian (object edges)

The Laplacian filter is a second-order derivative filter used in digital image processing for edge detection. It measures the rate of change of intensity in an image, making it particularly effective at highlighting regions with rapid intensity shifts, such as edges. Unlike first-order derivative filters that detect edges in horizontal and vertical directions separately, the Laplacian filter detects edges across the entire image simultaneously. This makes it valuable for extracting fine details and enhancing features in an image, which is crucial for tasks like training machine learning models.

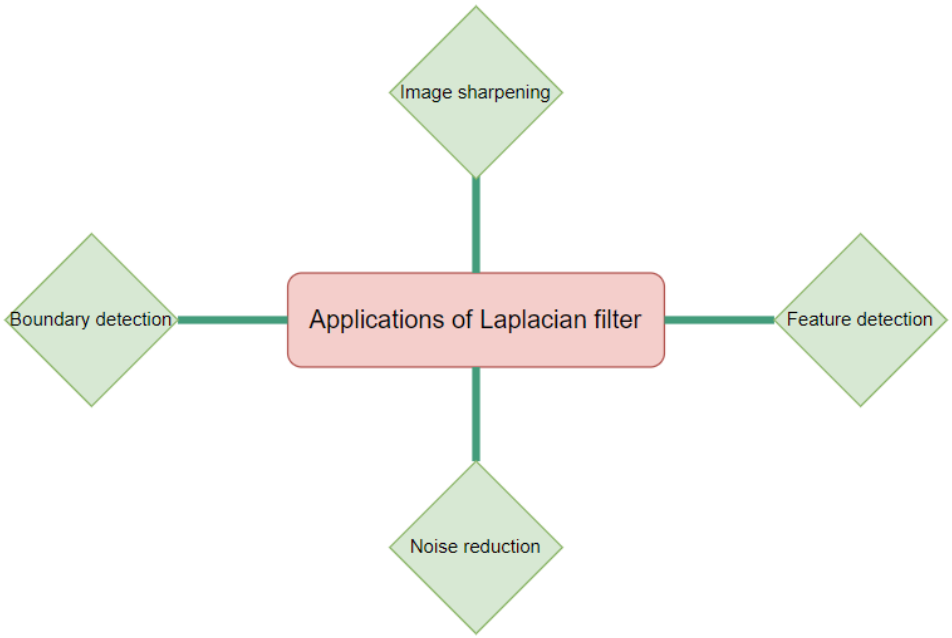
How the Laplacian filter works

Laplacian filter kernels usually contain negative values in a cross pattern, centered within the array. The corners are either zero or positive values. The center value can be either negative or positive. The following array is an example of a 3x3 kernel for a Laplacian filter.

then the kernel moves across the image, like sliding a small window from one side to the other. At each spot, the kernel covers a small part of the image. It multiplies the numbers in the kernel with the numbers in the image it’s covering and adds up the results. This gives a new number that replaces the old one in the image. This process helps to highlight important details, like edges.

Applications of the Laplacian Filter

We use the Laplacian filter in various tasks. Besides detecting edges, many other wide-range applications of Laplacian filters, some of which are illustrated below.



Applications of Laplacian filter

Example Code for Implementing the Laplacian Filter

Code:

```
; Select the file.
file = FILEPATH('nyny.dat', SUBDIRECTORY = ['examples', 'data'])
orig_imageSize = [768, 512]

; Use READ_BINARY to read the image as a binary file.
orig_image = READ_BINARY(file, DATA_DIMS = orig_imageSize)

; Crop the image to focus on the bridges.
croppedSize = [256, 256]
croppedImage = orig_image[200:(croppedSize[0] - 1) + 200, $
  180:(croppedSize[1] - 1) + 180]

; Display the cropped image.
im01 = IMAGE(croppedImage, $
   TITLE = 'Original cropped image')

; Create a kernel of a Laplacian filter.
kernelSize = [3, 3]
kernel = FLTARR(kernelSize[0], kernelSize[1])
kernel[1, *] = -1.
kernel[*, 1] = -1.
kernel[1, 1] = 4.

; Apply the filter to the image.
filteredImage = CONVOL(FLOAT(croppedImage), kernel, /CENTER)

; Display the resulting filtered image:
im02 = IMAGE(filteredImage, $
   TITLE = 'Laplacian-filtered image')
```

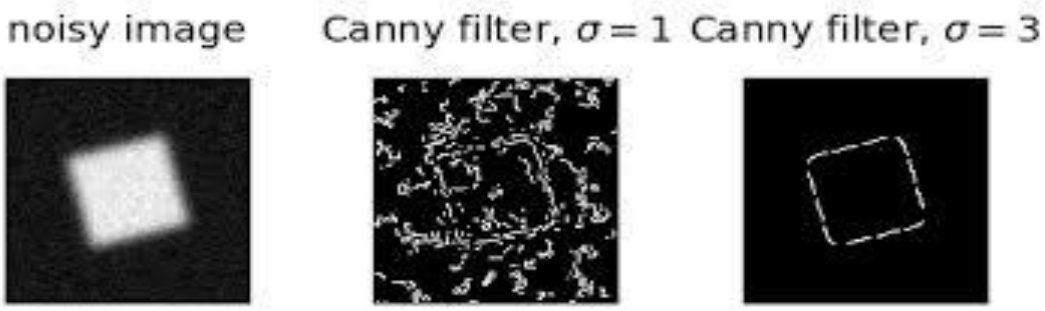
Results:

Original cropped image

Laplacian-filtered image

3. Canny Edge Detector

Introduction to the Canny Edge Detector



Canny edge detection is a popular and widely used edge detection technique that aims to identify and extract the edges of objects within an image. It was developed by John F. Canny in 1986 and has since become a fundamental tool in computer vision and image analysis.

Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations.

How the Canny Edge Detection works

The Canny operator works in a multi-stage process. First of all, the image is smoothed by Gaussian convolution. Then a simple 2-D first derivative operator is applied to the smoothed image to highlight regions of the image with high first spatial derivatives. Edges give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as *non-maximal suppression*.

The tracking process exhibits hysteresis controlled by two thresholds: *T1* and *T2*, with *T1* > *T2*. Tracking can only begin at a point on a ridge higher than *T1*. Tracking then continues in both directions out from that point until the height of the ridge falls below *T2*. This hysteresis helps to ensure that noisy edges are not broken up into multiple edge fragments.

Applications of the Canny Edge Detection:

1. Object Detection and Recognition

- **Autonomous Vehicles:** Canny edge detection helps in identifying lanes, road edges, and other vehicles by detecting edges in images captured by cameras.
- **Facial Recognition:** Used to detect the contours of a face and key facial features, which can then be used for recognition purposes.

2. Barcode and QR Code Scanning

- **Edge Detection for Scanning:** Canny edge detection helps in identifying the edges of barcodes and QR codes, making it easier to read and decode them.

3. Video Surveillance

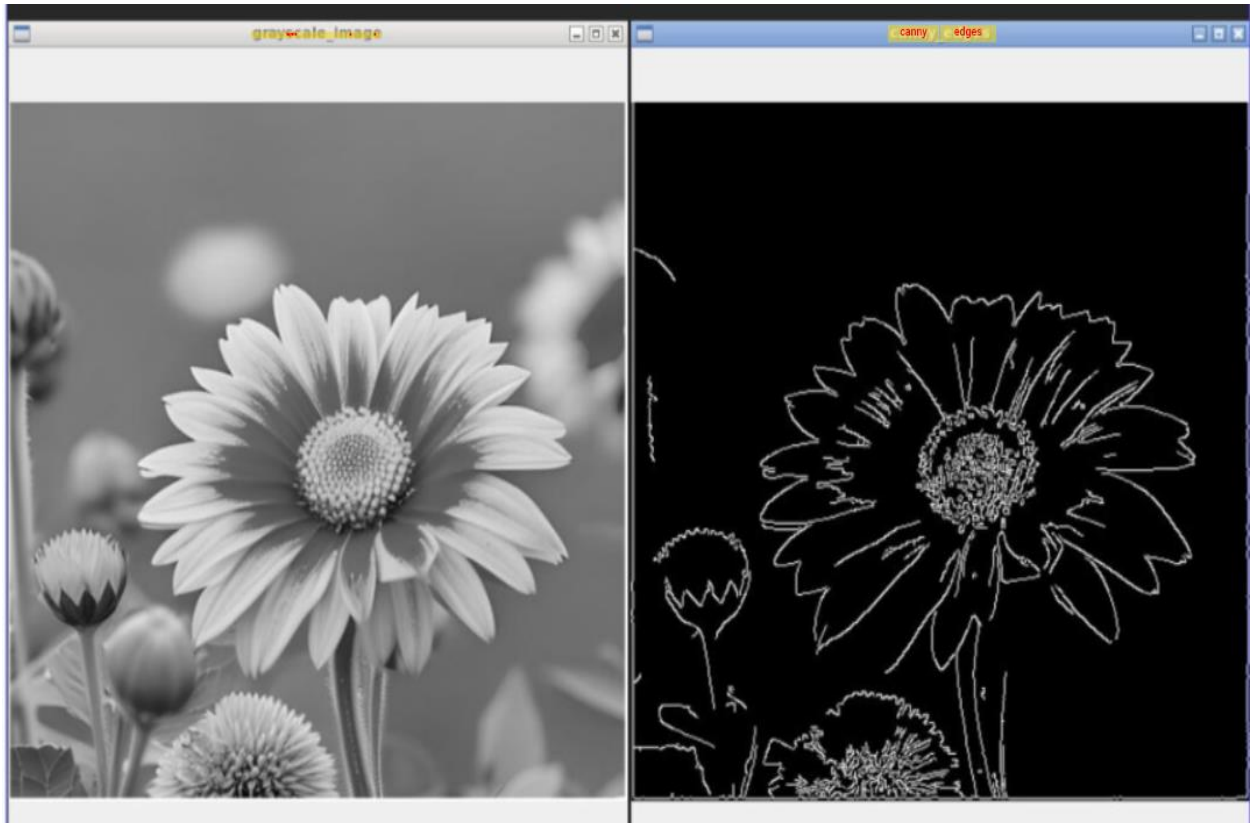
- **Motion Detection:** By detecting edges in successive frames, it helps in identifying moving objects, which is crucial for video surveillance systems.

Example Code for Implementing the Canny Edge Detection

Code:

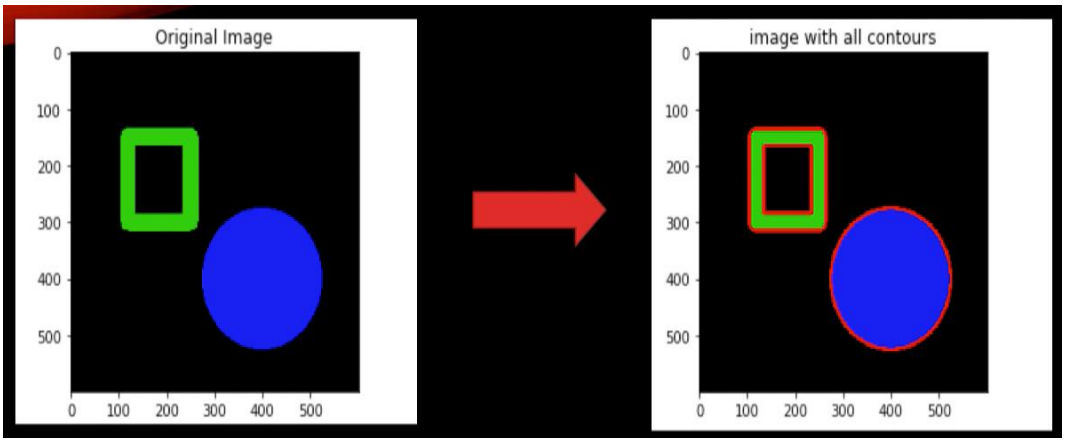
```
import cv2

# Load an image
image = cv2.imread('flower.png', cv2.IMREAD_GRAYSCALE)

#create a named window to display image (optional)
cv2.namedWindow('grayscale_image')
cv2.moveWindow('grayscale_image', 0,0)
image_display = cv2.resize(image, (510, 510))
cv2.imshow('grayscale_image', image_display)

# Apply Canny edge detection
canny_edges = cv2.Canny(image, threshold1=150, threshold2=400)

#create another named window to display output image (optional)
cv2.namedWindow('canny_edges')
cv2.moveWindow('canny_edges', 513,0)
image = cv2.resize(canny_edges, (510, 510))
cv2.imshow('canny_edges', image)
```

Results:



4. Contours in Image processing

Introduction to the Sobel filter



Contours can be defined as the curves or outlines that represent the boundaries of objects or shapes within an image. These curves join all the continuous points (along the boundary) having the same color or intensity, highlighting the structural properties of objects and providing a way to extract and represent shape information from images.

Contours are useful when working with grayscale or binary images where objects are clearly distinguished from the background based on variations in brightness or color. Even with simple images we transform it into grayscale or binary first to detect contours in a simple way.

How the Contours works

To represent contours, methods like chain codes are used. Chain codes trace the outline of an object by following the direction from one pixel to the next, either in four or eight directions. These codes help describe the shape of the object in a simple way. Sometimes, the contour might not be continuous due to changes in the image, so edge detection techniques are used to connect the gaps and complete the contour.

The process of finding contours usually starts with binarization, where the image is converted to black and white. This makes it easier to spot the boundary pixels. Techniques like Otsu's thresholding is often used to achieve this. After binarization, contours are extracted, but the initial results might have breaks in the boundary. Methods like the Canny or Roberts operators help fill in these gaps.

Applications of Contours:

Contours have many practical uses in computer vision. They are crucial for recognizing objects, compressing images by focusing on important pixels, and helping algorithms find key features in images.

1. Object Recognition:
    - Contour detection plays a pivotal role in recognizing and distinguishing objects within an image.
2. Shape Analysis:
    - It enables the analysis of shapes, facilitating tasks like character recognition and geometric measurements.
3. Image Segmentation:
    - Contours aid in segmenting an image into meaningful regions, enhancing the efficiency of subsequent processing steps.
4. Object Tracking:
    - Dynamic applications such as object tracking leverage contours to monitor and follow the movement of objects over time.

Example Code for Implementing Contour

Code:

```
/import cv2

# read the image
image = cv2.imread('input/image_1.jpg')

# convert the image to grayscale format
img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# apply binary thresholding
ret, thresh = cv2.threshold(img_gray, 150, 255, cv2.THRESH_BINARY)
# visualize the binary image
cv2.imshow('Binary image', thresh)
cv2.waitKey(0)
cv2.imwrite('image_thres1.jpg', thresh)
cv2.destroyAllWindows()

# detect the contours on the binary image using cv2.CHAIN_APPROX_NONE
contours, hierarchy = cv2.findContours(image=thresh, mode=cv2.RETR_TREE, method=cv2.CHAIN_APPROX_NONE)

# draw contours on the original image
image_copy = image.copy()
cv2.drawContours(image=image_copy, contours=contours, contourIdx=-1, color=(0, 255, 0), thickness=2, lineType=cv2.LINE_AA)

# see the results
cv2.imshow('None approximation', image_copy)
cv2.waitKey(0)
cv2.imwrite('contours_none_image1.jpg', image_copy)
cv2.destroyAllWindows()
```

Results: