PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

UE21MA141B- LINEAR ALGEBRA AND ITS APPLICATIONS

LAA- PROJECT

Session: Jan-May 2023

Branch : BTECH CSE

Semester & Section : 4th SEM, 'C' SEC

Sl No.	Name of the Student	SRN	Marks Allotted (Out of 5)
1.	DHIRURAM B	PES1UG21CS918	
2.	CHANDANA S M	PES1UG21CS149	
3.	DEEKSHA KRISHNAPPA	PES1UG21CS169	

Name of the Course Instructor : **Dr. UMA D**

Signature of the Course Instructor

(with Date) : ______________________________

Table of Contents:

EDGE DETECTION using IMAGE PROCESSING

AS AN APPLICATION OF LINEAR ALGEBRA

Abstract - What is Edge Detection?

Edge detection using image processing can be seen as a **prominent application of linear algebra**. In image processing, **edge detection algorithms** are commonly used to extract features from images, and these algorithms typically **involve the use of mathematical operations** on image data.

In our project, we have opted to go with the **Canny Edge Detection Algorithm**. This algorithm involves convolving the image with a **Gaussian filter**, computing the **gradient of the image**, and applying **non-maximum suppression** and **hysteresis thresholding** to obtain the final edge map. These operations can be formulated **using linear algebra**, with the Gaussian filter and gradient operators represented as **matrices** and the convolution operation represented as **matrix multiplication**.

Other edge detection techniques such as the **Sobel, Prewitt, and Roberts** operators also involve the use of convolution with matrices, which can be represented as linear algebra operations.

Procedure:

Here is a brief overview of the steps involved in Canny Edge Detection:

1. **Gaussian smoothing or Noise Reduction:** The image is convolved with a Gaussian kernel to reduce noise.
2. **Gradient calculation:** The gradient magnitude and direction are calculated for each pixel using the Sobel operator or another edge detection operator.
3. **Non-maximum suppression:** The gradient magnitude is thinned by suppressing non-maximum pixels along the direction perpendicular to the edge.
4. **Double thresholding:** The edges are separated into strong and weak edges using two thresholds. Pixels with gradient magnitude above the high threshold are considered strong edges, while those below the low threshold are discarded. Pixels with gradient magnitude between the two thresholds are considered weak edges.
5. **Edge tracking by hysteresis:** The weak edges are connected to the strong edges using a process called edge tracking. Starting from a strong edge pixel, the algorithm follows the connected weak edge pixels until it reaches a weak edge that is not connected to any other strong edges.

Linear Algebra Methods and Concepts used:

1. **Gaussian smoothing or Noise Reduction:**

   One way to get rid of the noise on the image, is by applying Gaussian blur to smooth it. To do so, image convolution technique is applied with a Gaussian Kernel (3x3, 5x5, 7x7 etc…). The kernel size depends on the expected blurring effect. Basically, the smallest the kernel, the less visible is the blur. In our example, we will use a 5 by 5 Gaussian kernel.

   The equation for a Gaussian filter kernel of size (2k+1)×(2k+1) is given by:

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right); 1 \le i, j \le (2k+1)$$

2. **Gradient calculation:**

   The Gradient calculation step detects the edge intensity and direction by calculating the gradient of the image using edge detection operators.

   Edges correspond to a change of pixels' intensity. To detect it, the easiest way is to apply filters that highlight this intensity change in both directions: horizontal (x) and vertical (y)

   When the image is smoothed, the derivatives Ix and Iy w.r.t. x and y are calculated. It can be implemented by convolving I with Sobel kernels Kx and Ky, respectively:

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

   Then, the magnitude ***G*** and the slope ***θ*** of the gradient are calculated as follow:

$$|G| = \sqrt{I_x^2 + I_y^2},$$
$$\theta(x,y) = arctan\left(\frac{I_y}{I_x}\right)$$

   The result is almost the expected one, but we can see that some of the edges are thick and others are thin. Non-Max Suppression step will help us mitigate the thick ones.

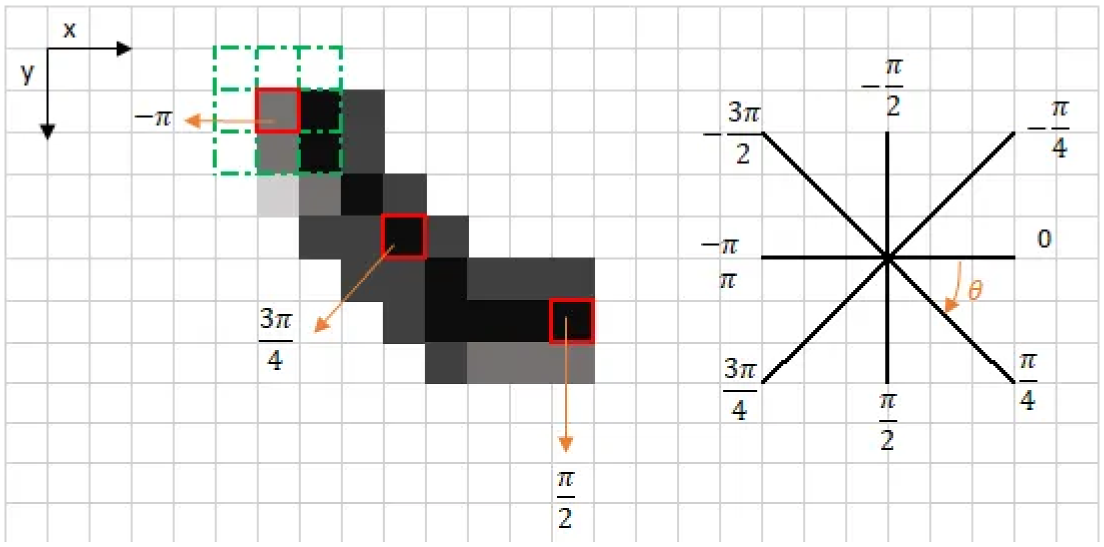
   Moreover, the gradient intensity level is between 0 and 255 which is not uniform. The edges on the final result should have the same intensity (i-e. white pixel = 255).

3. **Non-maximum suppression:**

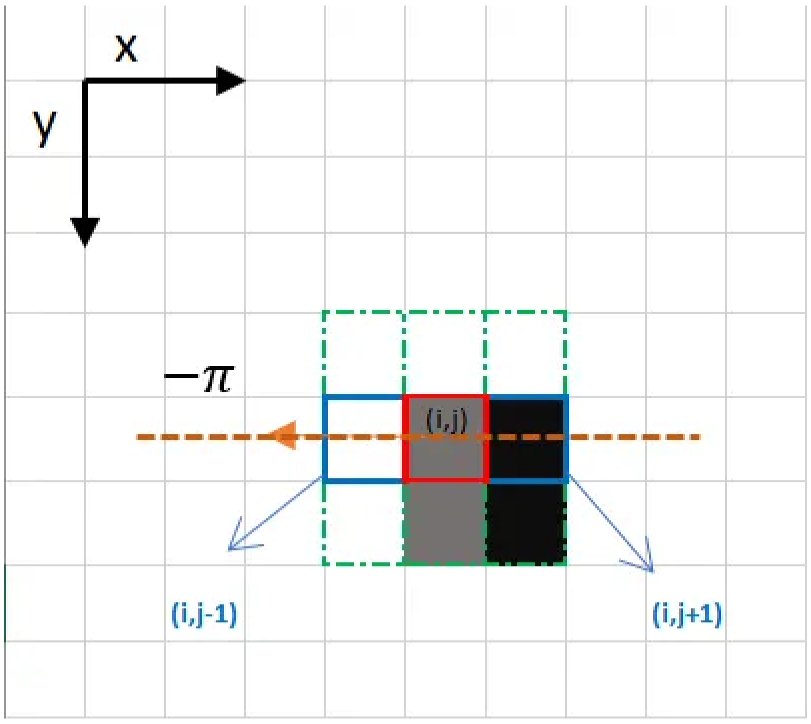
Ideally, the final image should have thin edges. Thus, we must perform non-maximum suppression to thin out the edges.

The principle is simple: the algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions.

Let's take an easy example:



The upper left corner red box present on the above image, represents an intensity pixel of the Gradient Intensity matrix being processed. The corresponding edge direction is represented by the orange arrow with an angle of -pi radians (+/-180 degrees).



Focus on the upper left corner red box pixel

The edge direction is the orange dotted line (horizontal from left to right). The purpose of the algorithm is to check if the pixels on the same direction are more or less intense than the ones being processed. In the example above, the pixel (i, j) is being processed, and the pixels on the same direction are highlighted in blue (i, j-1) and (i, j+1). If one those two pixels are more intense than the one being processed, then only the more intense one is kept. Pixel (i, j-1) seems to be more intense, because it is white (value of 255). Hence, the intensity value of the current pixel (i, j) is set to 0. If there are no pixels in the edge direction having more intense values, then the value of the current pixel is kept.

Let's sum this up. Each pixel has 2 main criteria (edge direction in radians, and pixel intensity (between 0–255)). Based on these inputs the non-max-suppression steps are:

- Create a matrix initialized to 0 of the same size of the original gradient intensity matrix;
- Identify the edge direction based on the angle value from the angle matrix;
- Check if the pixel in the same direction has a higher intensity than the pixel that is currently processed;
- Return the image processed with the non-max suppression algorithm.

The result is the same image with thinner edges. We can however still notice some variation regarding the edges' intensity: some pixels seem to be brighter than others, and we will try to cover this shortcoming with the two final steps.

4. **Double thresholding:**

The double threshold step aims at identifying 3 kinds of pixels: strong, weak, and non-relevant:

- Strong pixels are pixels that have an intensity so high that we are sure they contribute to the final edge.
- Weak pixels are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as non-relevant for the edge detection.
- Other pixels are considered as non-relevant for the edge.

Now you can see what the double thresholds holds for:

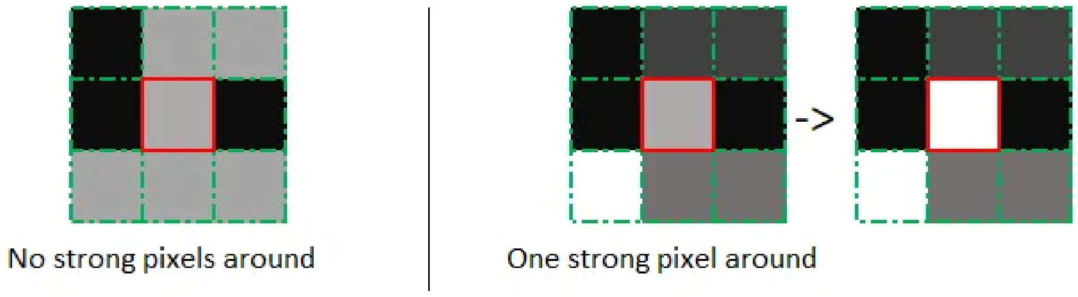
- High threshold is used to identify the strong pixels (intensity higher than the high threshold)
- Low threshold is used to identify the non-relevant pixels (intensity lower than the low threshold)

- All pixels having intensity between both thresholds are flagged as weak and the Hysteresis mechanism (next step) will help us identify the ones that could be considered as strong and the ones that are considered as non-relevant.

The result of this step is an image with only 2 pixel intensity values (strong and weak).

5. **<u>Edge tracking by hysteresis</u>:**

   Based on the threshold results, the hysteresis consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one, as described below:



<u>Code and Implementation</u>:

Code:

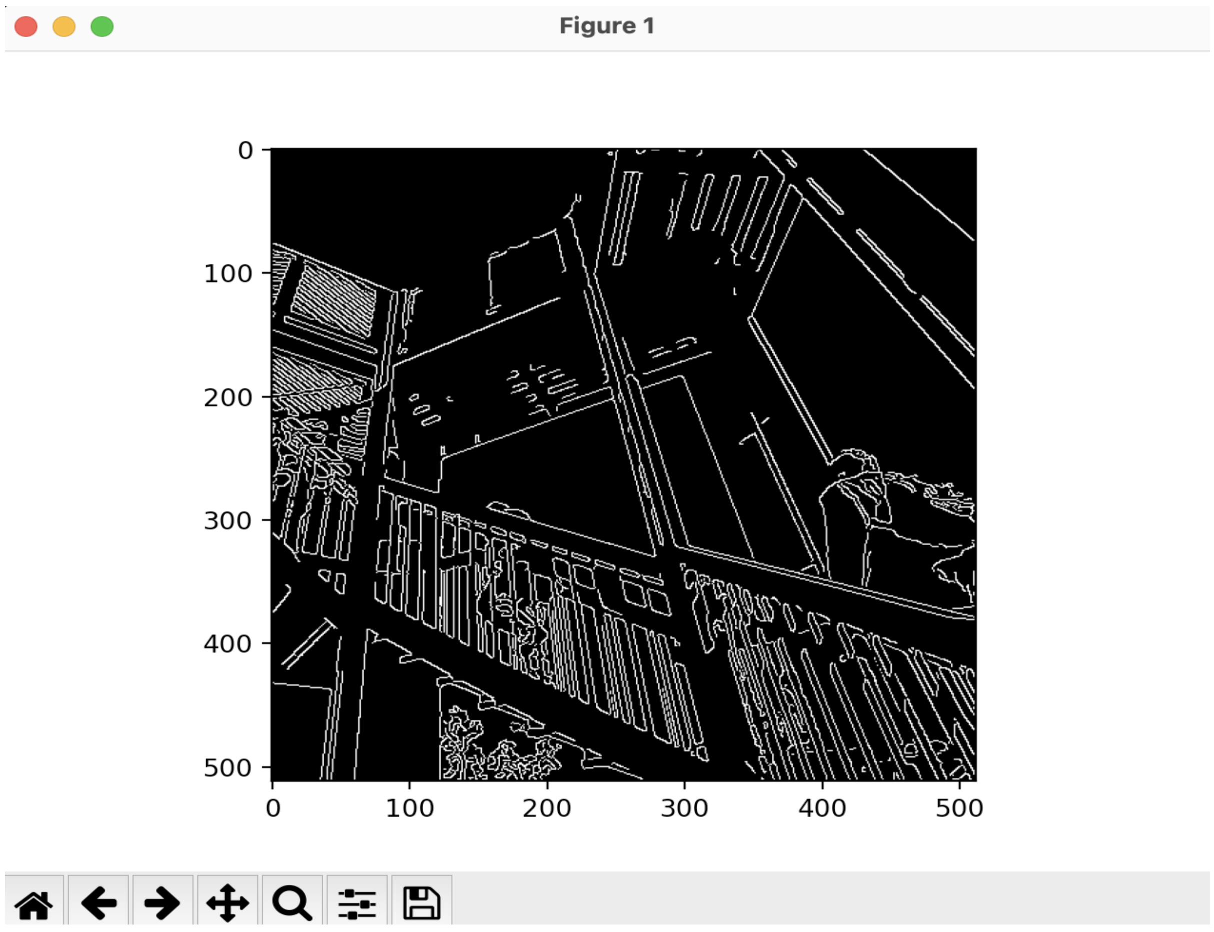
```
from scipy import *
import imageio
from scipy import ndimage
from scipy.ndimage.filters import convolve

from scipy import misc
import numpy as np

class cannyEdgeDetector:
   def __init__(self, imgs, sigma=1, kernel_size=5, weak_pixel=75,
strong_pixel=255, lowthreshold=0.05, highthreshold=0.15):
       self.imgs = imgs
       self.imgs_final = []
       self.img_smoothed = None
```

```
    self.gradientMat = None
    self.thetaMat = None
    self.nonMaxImg = None
    self.thresholdImg = None
    self.weak_pixel = weak_pixel
    self.strong_pixel = strong_pixel
    self.sigma = sigma
    self.kernel_size = kernel_size
    self.lowThreshold = lowthreshold
    self.highThreshold = highthreshold
    return

def gaussian_kernel(self, size, sigma=1):
    size = int(size) // 2
    x, y = np.mgrid[-size:size+1, -size:size+1]
    normal = 1 / (2.0 * np.pi * sigma**2)
    g =  np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
    return g

def sobel_filters(self, img):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)

    Ix = ndimage.filters.convolve(img, Kx)
    Iy = ndimage.filters.convolve(img, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)
    return (G, theta)


def non_max_suppression(self, img, D):
    M, N = img.shape
    Z = np.zeros((M,N), dtype=np.int32)
    angle = D * 180. / np.pi
    angle[angle < 0] += 180


    for i in range(1,M-1):
        for j in range(1,N-1):
            try:
                q = 255
                r = 255

               #angle 0
```

```
                if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                    q = img[i, j+1]
                    r = img[i, j-1]
                #angle 45
                elif (22.5 <= angle[i,j] < 67.5):
                    q = img[i+1, j-1]
                    r = img[i-1, j+1]
                #angle 90
                elif (67.5 <= angle[i,j] < 112.5):
                    q = img[i+1, j]
                    r = img[i-1, j]
                #angle 135
                elif (112.5 <= angle[i,j] < 157.5):
                    q = img[i-1, j-1]
                    r = img[i+1, j+1]

                if (img[i,j] >= q) and (img[i,j] >= r):
                    Z[i,j] = img[i,j]
                else:
                    Z[i,j] = 0


            except IndexError as e:
                pass

        return Z

    def threshold(self, img):

        highThreshold = img.max() * self.highThreshold;
        lowThreshold = highThreshold * self.lowThreshold;

        M, N = img.shape
        res = np.zeros((M,N), dtype=np.int32)

        weak = np.int32(self.weak_pixel)
        strong = np.int32(self.strong_pixel)

        strong_i, strong_j = np.where(img >= highThreshold)
        zeros_i, zeros_j = np.where(img < lowThreshold)

        weak_i, weak_j = np.where((img <= highThreshold) & (img >=
lowThreshold))

        res[strong_i, strong_j] = strong
        res[weak_i, weak_j] = weak
        return (res)

    def hysteresis(self, img):

        M, N = img.shape
        weak = self.weak_pixel
        strong = self.strong_pixel

        for i in range(1, M-1):
            for j in range(1, N-1):
                if (img[i,j] == weak):
                    try:
                        if ((img[i+1, j-1] == strong) or (img[i+1, j] == strong) or
(img[i+1, j+1] == strong)
                            or (img[i, j-1] == strong) or (img[i, j+1] == strong)
                            or (img[i-1, j-1] == strong) or (img[i-1, j] == strong) or
(img[i-1, j+1] == strong)):
                            img[i, j] = strong
                        else:
                            img[i, j] = 0
                    except IndexError as e:
                        pass

        return img

    def detect(self):
        imgs_final = []
        for i, img in enumerate(self.imgs):
            self.img_smoothed = convolve(img,
self.gaussian_kernel(self.kernel_size, self.sigma))
            self.gradientMat, self.thetaMat = self.sobel_filters(self.img_smoothed)
            self.nonMaxImg = self.non_max_suppression(self.gradientMat,
self.thetaMat)
            self.thresholdImg = self.threshold(self.nonMaxImg)
            img_final = self.hysteresis(self.thresholdImg)
            self.imgs_final.append(img_final)

        return self.imgs_final



# Load the image
img = misc.ascent()

# Create an instance of the cannyEdgeDetector class
```

```
detector = cannyEdgeDetector([img])

# Detect the edges in the image
detector.detect()

# Access the final output image(s) from the detector object
final_img = detector.imgs_final[0]

# Display the final output image
import matplotlib.pyplot as plt

plt.imshow(final_img, cmap=plt.cm.gray)
plt.show()
```

Output:



Real World Applications:

Edge detection is a fundamental and widely used technique in image processing and computer vision. It has numerous applications in various fields such as:

- **Object detection and recognition**: Edge detection is used to identify the boundaries of objects in an image, which is essential for object detection and recognition.
- **Medical imaging**: Edge detection is used to detect tumors and other abnormalities in medical images such as X-rays, CT scans, and MRIs.
- **Robotics**: Edge detection is used in robotics for object detection and tracking, as well as for obstacle detection.
- **Surveillance**: Edge detection is used in video surveillance to detect suspicious behavior, track moving objects, and identify people and vehicles.
- **Autonomous vehicles**: Edge detection is used in autonomous vehicles for obstacle detection, lane detection, and object tracking.
- **Quality control**: Edge detection is used in quality control to detect defects and flaws in manufactured products.
- **Agriculture**: Edge detection is used in agriculture for crop monitoring and yield estimation.

Overall, edge detection is a versatile technique that has numerous applications in various fields.

Advantages of Canny Edge Algorithm:

1. **Good accuracy**: The Canny algorithm is known for its high accuracy in detecting edges in images, and it can detect even subtle edges.
2. **Low error rate**: It has a low error rate compared to other edge detection algorithms, making it useful in applications where accuracy is crucial.
3. **Handles noise well**: The Canny algorithm has a noise reduction step that makes it less susceptible to noise in the image.
4. **Single edge detection**: The algorithm can detect only single, well-defined edges, making it useful for applications that require clear and distinct edges.
5. **Tunable parameters**: The algorithm allows for tuning of parameters like the threshold value, making it flexible for different use cases.

Disadvantages of Canny Edge Algorithm:

1. **Computationally expensive:** The Canny algorithm involves several computationally intensive steps, making it slower than other edge detection algorithms.
2. **Not suitable for all images:** The algorithm is not suitable for all types of images, particularly those with complex backgrounds or multiple edges.

3. **Parameters tuning:** Despite the advantage of being tunable, selecting the optimal threshold values for the algorithm can be a challenge.
4. **Missing low-contrast edges:** The Canny algorithm may not detect low-contrast edges in an image that have a similar intensity level to the surrounding areas.
5. **Sensitivity to lighting changes:** The algorithm may be affected by lighting changes, particularly if the image has shadows or other variations in lighting.

Future enhancements:

1. The current code loads an image from the scipy library using the misc.ascent() function. In the future, we would like to implement the functionality of loading our own images instead of the auto-generated image from scipy library.
2. The current code only displays the resultant picture, after all the filter functions have been applied on the original image. In the future, we would like to be able to display both, the original image and the resultant image side by side for easy comparison.
3. We would like to run this algorithm on several images to test its accuracy, and fine-tune it to ensure it meets our desired accuracy levels.

References:

- https://en.wikipedia.org/wiki/Canny_edge_detector
- https://analyticsindiamag.com/different-edge-detection-techniques-with-implementation-in-opencv/
- https://medium.com/swlh/applications-of-linear-algebra-in-image-filters-part-i-operations-aeb64f236845#:~:text=2.2%20Edge%20Detection%3A&text=The%20main%20idea%20of%20the,by%20the%20absolute%20value%2C%20i.e.
- https://github.com/hemanth-nag/basic_image_processing/blob/master/Linear_Algebra_filters.ipynb
- https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123
- https://github.com/FienSoP/canny_edge_detector
- https://www.slideshare.net/NareshBiloniya/application-of-edge-detection-96942725
- https://www.youtube.com/watch?v=sRFM5IEqR2w&t=195s&ab_channel=Computerphile
- Further questioning on various related topics on Google and ChatGPT for in-depth understanding and clarity.