Project 1 - Canny Edge Detector

Project group members: 1. Aniket Bote (N12824308)

2. Sindhu Harish (N19806874)

Instructions to run the code:

1. Download and install python 3.8.6.
2. Clone the project and navigate to the project directory named 'canny-edge-detector'.
3. Create a virtual environment (OPTIONAL).
   - ➔ pip install virtualenv
   - ➔ virtualenv venv
4. Install the required libraries after activating the virtual environment.
   - ➔ pip install -r requirements.txt
5. Run the below command
   - ➔ python canny_edge_detector.py --input_folder input --output_folder output
6. The input_folder is from where the input images are read, and output_folder is where the output images are going to get saved.
7. In the given project structure input_folder is named as input and output_folder is named as output respectively.

Source Code:

1. The main file where the code execution starts:- canny_edge_detector.py

```
# Import the required libraries

import argparse
import glob
import os
import shutil

import cv2

from gaussian_smoothing import perform_gaussian_smoothing
from gradient_operation import perform_gradient_operation
from non_maxima_suppression import perform_non_maxima_suppression
from thresholding import perform_thresholding


parser = argparse.ArgumentParser()
parser.add_argument(
    '--input_folder',
    type=str,
    default='input',
    required=False,
```

```python
    help='input folder with images'
)



parser.add_argument(
    '--output_folder',
    type=str,
    default='output',
    required=False,
    help='output folder to save processed
images'
)

args = parser.parse_args()

if os.path.exists(args.output_folder):
    shutil.rmtree(args.output_folder)

os.makedirs(args.output_folder)

print("Reading images from input folder")
images = glob.glob(os.path.join(args.input_folder, '*.bmp'))
for image_name in images:
      output_image_name = image_name.split('\\')[1].split('.bmp')[0]
      img = cv2.imread(image_name, cv2.IMREAD_GRAYSCALE)

      print("Performing gaussian smoothing for image: " +
      output_image_name)gaussian_smooth_image =
      perform_gaussian_smoothing(args, output_image_name, img)

      print("Performing gradient smoothing for image: " +
      output_image_name)
      M, THETA = perform_gradient_operation(args, output_image_name,
      gaussian_smooth_image)

      print("Performing non-maxima suppression for image: " +
      output_image_name)
      NMS = perform_non_maxima_suppression(args, output_image_name,
      M, THETA)

      print("Performing thresholding for image: " +
      output_image_name, '\n')
      T1, T2, T3 = perform_thresholding(args, output_image_name,
      NMS)
```

2. Source Code for Gaussian Smoothing :- gaussian_smoothing.py

```
import os
import cv2
import numpy as np
from utils import Operator, apply_discrete_convolution


def perform_gaussian_smoothing(args, image_name, image):
    '''
    Args:
        image : An image to on which smoothing will appear
    Returns:
        smoothened image : Smoothened image
    '''
    # Apply discreet convolution with gaussian mask
    image = apply_discrete_convolution(image, Operator.gaussian_mask)

    # Normalize the image
    image = image / np.sum(Operator.gaussian_mask)

    #write image into the output folder after normalization
    cv2.imwrite(os.path.join(args.output_folder, image_name +
'_gaussian_smooth_normalized.bmp'), image)

    # Return the smoothened image
    return image
```

3. Source Code for Gradient Operation:- gradient_operation.py

```
import os
import cv2
import numpy as np
from utils import Operator, apply_discrete_convolution


def perform_gradient_operation(args, image_name, image):
    '''
    Args:
```

```
        image : An image on which gradient operation will happen
    Returns:
        Magnitude : Magnitude of the gradient
        Theta     : Gradient Angle
    '''
    # Compute horizontal gradients
    dfdx = apply_discrete_convolution(image, Operator.gx)

    #Copy Image to Output folder after horizontal gradient
    cv2.imwrite(os.path.join(args.output_folder, image_name +
'_Gx_normalized.bmp'), dfdx)

    # Compute vertical gradients
    dfdy = apply_discrete_convolution(image, Operator.gy)

    #Copy Image to Output folder after vertical gradient
    cv2.imwrite(os.path.join(args.output_folder, image_name +
'_Gy_normalized.bmp'), dfdy)

    # Compute magnitude of the gradient
    m = np.sqrt(np.square(dfdx) + np.square(dfdy))

    # Normalize gradient magnitude
    m = np.absolute(m) / 3

    #Copy Image to Output folder with gradient magnitude value
    cv2.imwrite(os.path.join(args.output_folder, image_name +
'_gradient_magnitude_normalized.bmp'), m)

    # Compute gradient angle
    theta = np.degrees(np.arctan2(dfdy, dfdx))

    return m, theta
```

4. Source Code for non-maxima-supression:- non_maxima_supression.py

```
import os
import numpy as np
import cv2
from utils import get_positive_angle, Sector

def perform_non_maxima_suppression(args, image_name, magnitude,
gradient_angle):
    '''
```

```
    Args:
        magnitude : Magnitude of the gradient
        gradient_angle : Gradient angle
    Returns:
        Magnitude : Magnitude array after non-maxima supression
    '''
    # Compute positive angles
    positive_gradient_angle = get_positive_angle(gradient_angle)

    # Get magnitude array shape
    m_arr, n_arr = magnitude.shape

    # reference pixel location during start of the process
    rpi_m, rpi_n = 1,1

    # Build output array
    output_arr = np.ones((m_arr , n_arr)) * np.nan

    for i in range(m_arr - 2):
        for j in range(n_arr - 2):
            # Compute output pixel location for output array
            op_m, op_n = i + rpi_m, j + rpi_n

            # Get 3 x 3 magnitude slice
            arr_slice = magnitude[i:i+3, j:j+3]

            # Get 3 x 3 angle slice
            angle_slice = positive_gradient_angle[i:i+3, j:j+3]

    # If undefined value at reference pixel in magnitude or angle
put zero in output pixel location
      if np.isnan(arr_slice[rpi_m][rpi_n]) or
      np.isnan(angle_slice[rpi_m][rpi_n]):
                output_arr[op_m][op_n] = 0
      else:
                # Get the sector value
                 sector =
             Sector().get_sector(angle_slice[rpi_m][rpi_n])

      if sector == 0:
# If undefined value at any of sector neighbour put zero in output
pixel location
      if np.isnan(arr_slice[rpi_m][rpi_n+1]) or
np.isnan(arr_slice[rpi_m][rpi_n-1]):
                output_arr[op_m][op_n] = 0
```

```
# If reference pixel is greater than its sector neighbours put reference pixel value at output location
  elif arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m][rpi_n+1] and arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m][rpi_n-1]:
                        output_arr[op_m][op_n] =
                        arr_slice[rpi_m][rpi_n]

# If reference pixel value is less than its sector neighbours put zero in output pixel location
   else:
        output_arr[op_m][op_n] = 0

   elif sector == 1:
# If undefined value at any of sector neighbour put zero in output pixel location
    if np.isnan(arr_slice[rpi_m-1][rpi_n+1]) or np.isnan(arr_slice[rpi_m+1][rpi_n-1]):
                        output_arr[op_m][op_n] = 0

# If reference pixel is greater than its sector neighbours put reference pixel value at output location
   elif arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m-1][rpi_n+1] and arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m+1][rpi_n-1]:
                        output_arr[op_m][op_n] =
arr_slice[rpi_m][rpi_n]

# If reference pixel value is less than its sector neighbours put zero in output pixel location
   else:
                        output_arr[op_m][op_n] = 0

   elif sector == 2:
# If undefined value at any of sector neighbour put zero in output pixel location
    if np.isnan(arr_slice[rpi_m-1][rpi_n]) or np.isnan(arr_slice[rpi_m+1][rpi_n]):
                        output_arr[op_m][op_n] = 0

# If reference pixel is greater than its sector neighbours put reference pixel value at output location
```

```
   elif arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m-1][rpi_n] and
arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m+1][rpi_n]:
                        output_arr[op_m][op_n] =
arr_slice[rpi_m][rpi_n]

# If reference pixel value is less than its sector neighbours put
zero in output pixel location
    else:
                        output_arr[op_m][op_n] = 0

    elif sector == 3:
# If undefined value at any of sector neighbour put zero in output
pixel location
     if np.isnan(arr_slice[rpi_m-1][rpi_n-1]) or
np.isnan(arr_slice[rpi_m+1][rpi_n+1]):
                        output_arr[op_m][op_n] = 0

# If reference pixel is greater than its sector neighbours put
reference pixel value at output location
  elif arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m-1][rpi_n-1] and
arr_slice[rpi_m][rpi_n] > arr_slice[rpi_m+1][rpi_n+1]:
                        output_arr[op_m][op_n] =
arr_slice[rpi_m][rpi_n]

 # If reference pixel value is less than its sector neighbours put
zero in output pixel location
   else:
                        output_arr[op_m][op_n] = 0

 # If sector value is other 0,1,2,3 raise an error.(Not going to
happen its there for correctness)
   else:
    raise f"Undefined sector: {sector}"
    cv2.imwrite(os.path.join(args.output_folder, image_name +
'_non_maxima_supression.bmp'), output_arr)
    return output_arr
```

5. **Source Code for Thresholding:- Thresholding.py**

```
import os
import numpy as np
import cv2

def perform_thresholding(args, image_name, image):
```

```
        '''
        Args:
            image: Non maxima suppressed
        Returns:
            img1 : Image after applying threshold t1
            img2 : Image after applying threshold t2
            img3 : Image after applying threshold t3
        '''

        # Store all the values of image after non- maxima suppression
        which are greater than zero into array
        image_arr = image[image>0].ravel()

        # Get 25th percentile of the array
          t1 = np.percentile(image_arr,25)
          image_1 = (image > t1).astype("int32")
          cv2.imwrite(os.path.join(args.output_folder, image_name +
          f'_threshold_t1_{np.round(t1, 2)}.bmp'), image_1 * 255)
        # Multiplying the image with 255 for contrast

        # Get 50th percentile of the array
          t2 = np.percentile(image_arr,50)
          image_2 = (image > t2).astype("int32")
          cv2.imwrite(os.path.join(args.output_folder, image_name +
          f'_threshold_t2_{np.round(t2, 2)}.bmp'), image_2 * 255)

        # Get 75th percentile of the array
          t3 = np.percentile(image_arr,75)
          image_3 = (image > t3).astype("int32")
          cv2.imwrite(os.path.join(args.output_folder, image_name +
          f'_threshold_t3_{np.round(t3, 2)}.bmp'), image_3 * 255)

       # Apply threshold to the image and convert it into integer array
          return image_1, image_2, image_3
```

6. Common Functions are included as Utility Class :- utility.py

```
import numpy as np


# A class to store all operators
class Operator:
    # Prewitt operator for Gx
    gx = np.array([
        [-1,0,1],
```

```
        [-1,0,1],
        [-1,0,1]])

    # Prewitt operator for Gy
    gy = np.array([
        [1,1,1],
        [0,0,0],
        [-1,-1,-1]])

    # Gaussian mask
    gaussian_mask = np.array([
        [1,1,2,2,2,1,1],
        [1,2,2,4,2,2,1],
        [2,2,4,8,4,2,2],
        [2,4,8,16,8,4,2],
        [2,2,4,8,4,2,2],
        [1,2,2,4,2,2,1],
        [1,1,2,2,2,1,1]])

# A class to store sector angle definitions and method to provide
sector based on angle
class Sector():
    def __init__(self):
        # Dictionary with {sector: sector range}
        self.sector = {0: [(0, 22.5),(337.5,360),(157.5,202.5)], 1:
[(22.5,67.5), (202.5,247.5)], 2:[(67.5,112.5), (247.5, 292.5)],
3:[(112.5, 157.5), (292.5,337.5)]}

    def get_sector(self, angle):
        for key, val in self.sector.items():
            for l,u in val:
                # check if angle lies in the range if yes return key
                if angle >= l and angle < u:
                    return key
        # If angle is not in any range we return -1. (Not going to
happen. Its there for correctness)
        return -1

# A function to apply dicreet convolutions
def apply_discrete_convolution(image, mask):
    '''
    Args:
        image : An image to use for convolution
        mask  : An mask to use for convolution
    Returns:
```

```
        convolved image: An image after convolution
    '''
    # Get the shape of image and mask
    (m_image, n_image), (m_mask, n_mask) = image.shape, mask.shape

    # Compute the reference pixel index from where output array will
start populating
    rpi_m, rpi_n = int(np.floor(m_mask/2)), int(np.floor(n_mask/2))

    # Initialize an output array with nan values
    output_arr = np.ones((m_image, n_image)) * np.nan

    # Iterate through the image
    for i in range(m_image - m_mask + 1):
        for j in range(n_image - n_mask + 1):
            # Isolate the image slice to apply convolution
            img_slice = image[i:i+m_mask, j:j+n_mask]
            # Apply convolution and store the result in output array
in approriate location
            output_arr[i+rpi_m][j+rpi_n] = np.sum(img_slice * mask)

    return output_arr

# A function to convert negative angles to positive angles
def get_positive_angle(angle):
    pos_angle = angle.copy()
    pos_angle[pos_angle<0] += 360
    return pos_angle
```

Output Images

Image 1

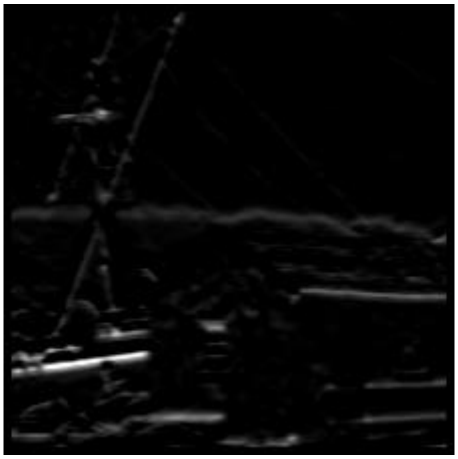
1. **Normalized image result after Gaussian smoothing**
   [input image – house.bmp, output image - House_gaussian_smooth_normalized.bmp]

2. **Normalized horizontal and vertical gradient responses (two separate images.) To generate normalized gradient responses, take the absolute value of the results first and then normalize.**

   Gx:
   [input image – house.bmp, output image – House_Gx_Normalized.bmp]

Gy:

[input image – house.bmp, output image - House_Gy_Normalized.bmp]



3. **Normalized gradient magnitude image.**

[input image – house.bmp, output image - House_gradient_magnitude_normalized.bmp]

4. Normalized gradient magnitude image after non-maxima suppression.
[input image – house.bmp, output image - House_non_maxima_supression.bmp]

(5) Binary edge maps using simple thresholding for thresholds chosen at the 25th, 50th and 75th percentiles

Thresholding T1:
[input image – house.bmp, output image - House_threshold_t1_2.28.bmp]

Thresholding T2:

[input image – house.bmp, output image - House_threshold_t2_5.3.bmp]

Thresholding T3:

[input image – house.bmp, output image - House_threshold_t3_15.56.bmp]

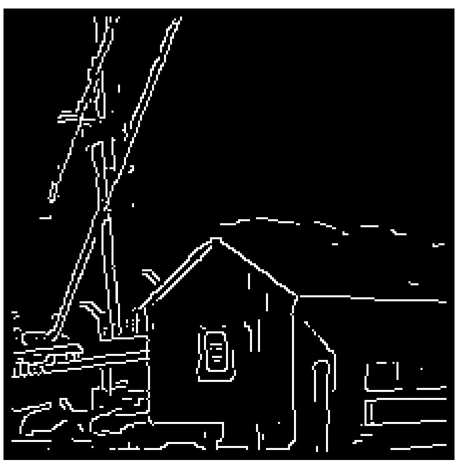


Image 2

(1) Normalized image result after Gaussian smoothing

[input image – Test patterns.bmp, output image - Test patterns_non_maxima_supression.bmp]

(2) Normalized horizontal and vertical gradient responses (two separate images.) To generate normalized gradient responses, take the absolute value of the results first and then normalize.

Gx:

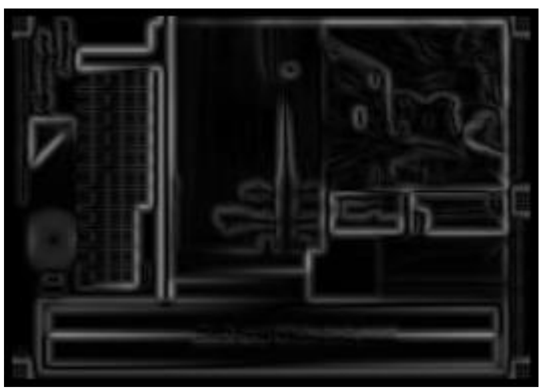
[input image – Test patterns.bmp, output image - Test patterns_non_maxima_supression.bmp]

Gy:

[input image – Test patterns.bmp, output image - Test patterns_non_maxima_supression.bmp]

3. **Normalized gradient magnitude image.**

[input image – Test patterns.bmp, output image - Test patterns_maxima_supression.bmp]



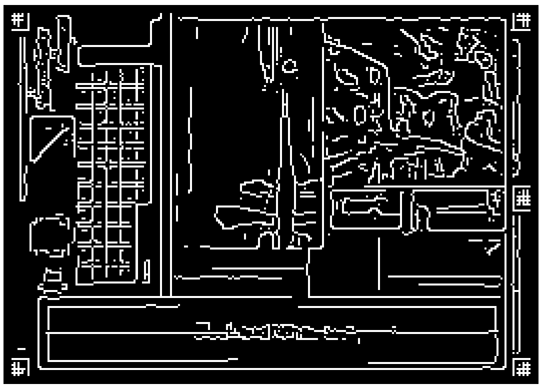
4. **Normalized gradient magnitude image after non-maxima suppression.**

[input image – Test patterns.bmp, output image - Test patterns_non_maxima_supression.bmp]

(5) Binary edge maps using simple thresholding for thresholds chosen at the 25th, 50th and 75th percentiles

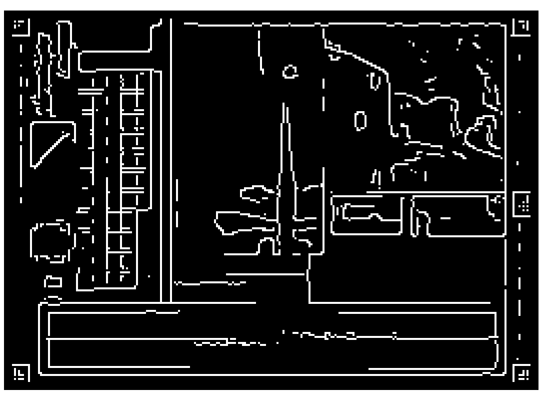
Thresholding T1:

[input image – Test patterns.bmp, output image - Test patterns_threshold_t1_2.28.bmp]



Thresholding T2:

input image – Test patterns.bmp, output image – Test_patterns_threshold_t2_24.45.bmp]



Thresholding T3:

[input image – Test patterns.bmp, output image - Test patterns_threshold_t3_15.56.bmp]