**CS 6643 - Computer Vision Fall 2021**

**CANNY’S EDGE DETECTOR**

Shravani Rakshe Niharika Krishnan

New York University New York University

spr4123@nyu.edu nk2982@nyu.edu

**ABSTRACT:**

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a range of edges in images. The stages include:

1. Gaussian Smoothing
2. Horizontal Gradient, Vertical Gradient, Gradient Magnitude, Gradient Angle
3. Non-Maxima Suppression
4. Simple Thresholding

**GITHUB LINK:** [**https://github.com/niharikakrishnan/Canny-Edge-Detector**](https://github.com/niharikakrishnan/Canny-Edge-Detector)

**IMPLEMENTATION:**

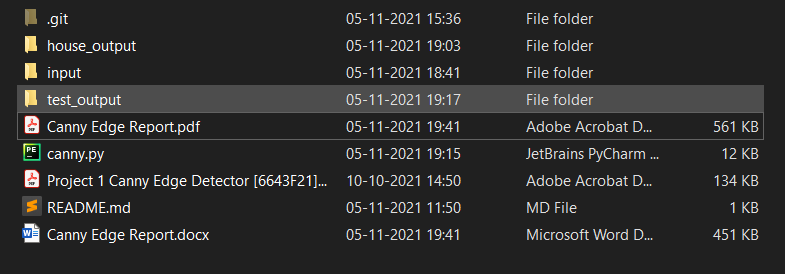
Assumes the pre-requisite environment with Python3 and necessary libraries (opencv & numpy) are already installed. If not, please install using:

| pip install opencv-python pip install numpy |
| --- |

Ensure the input image is in the same path as the source code file, ***canny.py***

**STEPS:**

1. Open terminal window and change directory to where the solution code, ***canny.py*** is located



2. Run the ***Command: python3 canny.py -i “[path\_to\_input\_filename]”***

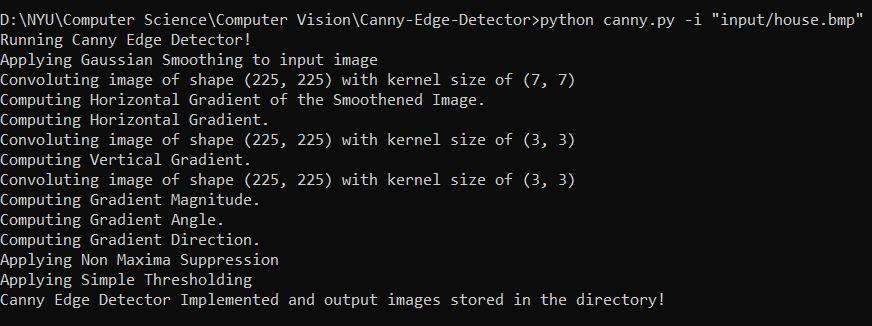
**Note:** Some machines have a different python path setup, in such cases, please use:

***Command: python canny.py -i “[path\_to\_input\_filename]”***

**Example:** python3 canny.py -i "house.bmp"

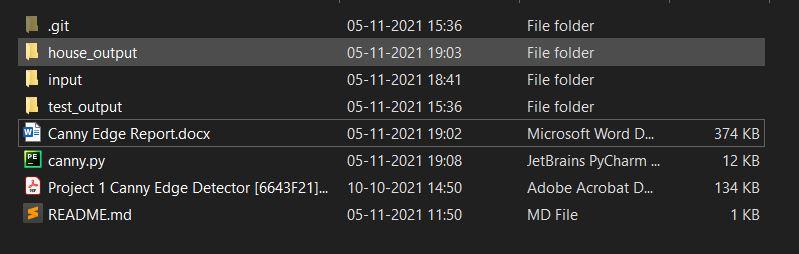


3. Canny Edge Detector Solution is implemented with print statements showing the completion of functions



4. The output images will be stored in a directory named ***filename\_output*** present at the same location

**Example: house\_output**

****

**SOURCE CODE:**

| from math import degrees, pi import numpy as np import argparse import cv2 import os  def convolution(image, mask): ''' Function to perform convolution of an image with a mask using matrix multiplication. In cases where the mask goes outside of the image border, it is considered as undefined and replaced with zeroes. Region of interest surrounding every reference pixel is computed and multiplied with the mask. :param image: a grayscale image of size N X M  :param mask: a mask/kernel of size N X N :return convoluted\_image: image after convolution with size N X M, same as output image '''  #Getting the number of rows and columns of the image and the mask using .shape method  image\_row, image\_col = image.shape  mask\_row, mask\_col = mask.shape  print("Convoluting image of shape {} with kernel size of {}".format(image.shape, mask.shape))  #Initialising the convoluted 2D array with zeros  convoluted\_image = np.zeros(image.shape)   #Defining the number of rows and columns that will be undefined based on the mask dimensions   add\_row = int(mask\_row - 1) // 2  add\_col = int(mask\_col - 1) // 2    #Initialising a 2D array with zeros along with extra rows and columns to handle the undefined values  modified\_image = np.zeros((image\_row + (2 \* add\_row), image\_col + (2 \* add\_col)))   modified\_image\_row, modified\_image\_col = modified\_image.shape    #Defining the region of interest for the input image  modified\_image[add\_row: modified\_image\_row - add\_row, add\_col:modified\_image\_col - add\_col] = image   #Matrix multiplication - Convoluting image with kernel  for row in range(1,image\_row-1):  for col in range(1,image\_col-1):  #Using sliding window concept for maxtrix multiplication of kernel and region of interest of input image  convoluted\_image[row, col] = np.sum(mask \* modified\_image[row : row + mask\_row, col : col + mask\_col])    return convoluted\_image  def gaussian\_smoothing(image, mask): ''' Function to perform Gaussian Smoothing with a 7 x 7 mask. Region of interest surrounding every reference pixel is computed by multiplying with the mask. :param image: a grayscale image of size N X M :param mask: a mask/kernel of size 7 X 7 :return gaussian\_image: image after smoothing '''  print("Applying Gaussian Smoothing to input image")    #Getting the number of rows and columns of the image using .shape method  image\_row, image\_col = image.shape   #Applying Gaussian Smoothing to the input image  gaussian\_image = convolution(image, mask)   #Normalising the gaussian smoothened image by using the sum of total pixels  for row in range(image\_row):  for col in range(image\_col):  if abs(gaussian\_image[row, col]) > 255:  gaussian\_image[row, col] = abs(gaussian\_image[row, col]) / 140    return gaussian\_image  def gradient\_operation(image, edge\_filter): ''' Function that uses Prewitt's operator to compute the horizontal gradient, vertical gradient. This is followed by the computation of gradient magnitude using the square root of the sum of squares of horizontal and vertical gradient. Gradient angle is computed using the tan inverse of vertical and horizontal gradient. :param image: a grayscale image after Gaussian smoothing :param edge\_filter: Prewitt's operator edge filter :return horizontal\_gradient: Normalized horizontal gradient :return vertical\_gradient: Normalized vertical gradient :return gradient\_magnitude: Normalized gradient magnitude image :return gradient\_direction: Gradient angle of the image '''  #Getting the number of rows and columns of the image using .shape method  image\_row, image\_col = image.shape   print("Computing Horizontal Gradient.")  #Computing the horizontal gradient by convoluting the input image with prewitt's horizontal edge filter  horizontal\_gradient = convolution(image, edge\_filter)   print("Computing Vertical Gradient.")  #Transforming and flipping the horizontal gradient edge\_filter to calculate vertical gradient  #Output: [[1,1,1], [0,0,0], [-1,-1,-1]]  vertical\_edge\_filter = np.flip(edge\_filter.T, axis=0)    #Computing the vertical gradient by convoluting the input image with prewitt's vertical edge filter  vertical\_gradient = convolution(image, vertical\_edge\_filter)   print("Computing Gradient Magnitude.")  #Using the formula, gradient magnitude = Square Root of Squares of Horizontal and Vertical Gradient  gradient\_magnitude = np.sqrt(np.square(horizontal\_gradient) + np.square(vertical\_gradient))   #Normalising the Gradient Magnitude by dividing it with (3\*root(2))  for row in range(image\_row):  for col in range(image\_col):  if abs(gradient\_magnitude[row, col]) > 255:  gradient\_magnitude[row, col] = abs(gradient\_magnitude[row, col]) / 3\*(np.sqrt(2))   print("Computing Gradient Angle.")  #Calculating gradient angle -> tan inverse (vertical gradient/horizontal gradient) in radians  gradient\_angle = np.arctan2(vertical\_gradient, horizontal\_gradient)    print("Computing Gradient Direction.")  #Converting gradient angle from radians to degree which returns in the range of -180 to 180. Required to perform non-maxima suppression  gradient\_direction = np.rad2deg(gradient\_angle)   #Adding 180 degrees to the gradient angle for converting the range from 0 360.  gradient\_direction += 180   return horizontal\_gradient, vertical\_gradient, gradient\_magnitude, gradient\_direction   def non\_maxima\_suppression(gradient\_magnitude, gradient\_direction): ''' Function to scan along the image gradient direction, and if pixels are not part of the local maxima they are set to zero. This has the effect of suppressing all image information that is not part of local maxima. :param gradient\_magnitude: square root of sum of squares of horizontal and vertical gradient :param gradient\_direction: the matrix that has the gradient angle at each pixel location :return nms\_output: Normalized gradient magnitude image after non-maxima suppression '''  print("Applying Non Maxima Suppression")   gradient\_row, gradient\_col = gradient\_magnitude.shape  #Getting the number of rows and columns of the gradient magnitude using .shape method  nms\_output = np.zeros(gradient\_magnitude.shape)    #Applying non maxima suppression to all pixels other than the border  for row in range(1, gradient\_row-1):  for col in range(1, gradient\_col-1):  angle = gradient\_direction[row, col]    #Mapping to Sector 0, hence compare with left and right pixels  if (0 <= angle < 22.5) or (157.5 <= angle < 202.5) or (337.5 <= angle <= 360):  before\_pixel = gradient\_magnitude[row, col - 1]  after\_pixel = gradient\_magnitude[row, col + 1]   #Mapping to Sector 1, hence compare with upper right and lower left pixels  elif (22.5 <= angle < 67.5) or (202.5 <= angle < 247.5):  before\_pixel = gradient\_magnitude[row + 1, col - 1]  after\_pixel = gradient\_magnitude[row - 1, col + 1]   #Mapping to Sector 2, hence compare with upper and lower pixels  elif (67.5 <= angle < 112.5) or (247.5 <= angle < 292.5):  before\_pixel = gradient\_magnitude[row - 1, col]  after\_pixel = gradient\_magnitude[row + 1, col]    #Mapping to Sector 3, hence compare with upper left and lower right pixels  else:  before\_pixel = gradient\_magnitude[row - 1, col - 1]  after\_pixel = gradient\_magnitude[row + 1, col + 1]   #If the centre pixel is strictly greater than the neighbouring pixels, we use the gradient magnitude value, else zero  if ((gradient\_magnitude[row, col] > before\_pixel) and (gradient\_magnitude[row, col] > after\_pixel)):  nms\_output[row, col] = gradient\_magnitude[row, col]    return nms\_output  def simple\_thresholding(image): ''' Function to compute binary edge maps using simple thresholding for thresholds chosen at the 25th, 50th and 75th percentiles :param image: the image after non maxima suppression :return tresholding\_output: Binary edge maps using simple thresholding for thresholds chosen at the 25th, 50th and 75th percentiles '''  print("Applying Simple Thresholding")    #Getting the number of rows and columns of the image using .shape method  image\_row, image\_col = image.shape   #Using the percentile function to calculate the threshold value at 25th, 50th and 75th percentile  threshold\_25 = np.percentile(list(set(image.flatten())), 25)  threshold\_50 = np.percentile(list(set(image.flatten())), 50)  threshold\_75 = np.percentile(list(set(image.flatten())), 75)  threshold = {"threshold\_25": threshold\_25, "threshold\_50": threshold\_50, "threshold\_75": threshold\_75}   thresholding\_output = {}   #Applying simple thresholding with threshold values at 25, 50 and 75th percentile  #If pixel intensity is strictly greater than the threshold value, we assign it with a value of 255 (white)  for threshold\_key, threshold\_value in threshold.items():    #Initializing output image array with zeroes  output = np.zeros(image.shape)  for row in range(image\_row):  for col in range(image\_col):  if image[row, col] > threshold\_value:  output[row, col] = 255  thresholding\_output[threshold\_key] = output    return thresholding\_output  if \_\_name\_\_ == '\_\_main\_\_':  print("Running Canny Edge Detector!")  ap = argparse.ArgumentParser()  ap.add\_argument("-i", "--image", required=True, help="Path to the image")  args = vars(ap.parse\_args())   #Reading and opening input image and converting to grayscale  frame = cv2.imread(args['image'])  image = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)   #Creating path to write output images of Canny Edge Detector  folder, fname\_with\_extension = os.path.split(args['image'])  fname , extension = os.path.splitext(fname\_with\_extension)  path = str(fname) + "\_output"  access = 0o755   #Checking whether the specified output path exists or not  isExist = os.path.exists(path)   #Creating a new directory if it does not already exist  if not isExist:  os.makedirs(path,access)  print("Output directory created.")   #Defining 7 x 7 Gaussian mask as mentioned in the Project requirement  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]], dtype='int')    #Applying Gaussian smoothing to input image with given mask  gaussian\_image = gaussian\_smoothing(image, mask)  cv2.imwrite(path + "/"+str(fname)+"\_GaussianSmoothing.bmp", gaussian\_image)   #Defining Prewitt's Edge Operator  edge\_filter = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype='int')   #Applying gradient operation to compute horizontal gradient, vertical gradient, gradient magnitude and gradient angle  horizontal\_gradient, vertical\_gradient, gradient\_magnitude, gradient\_direction = gradient\_operation(image, edge\_filter)  cv2.imwrite(path + "/"+str(fname)+"\_HorizontalGradient.bmp", horizontal\_gradient)  cv2.imwrite(path + "/"+str(fname)+"\_VerticalGradient.bmp", vertical\_gradient)  cv2.imwrite(path + "/"+str(fname)+"\_GradientMagnitude.bmp", gradient\_magnitude)   #Applying Non Maxima Suppression to the gradient magnitude  nonmaxima\_image = non\_maxima\_suppression(gradient\_magnitude, gradient\_direction)  cv2.imwrite(path + "/"+str(fname)+"\_NonMaximaSuppression.bmp", nonmaxima\_image)   #Applying Simple thresholding with thresholds chosen at 25th, 50th and 75th percentile  threshold\_image = simple\_thresholding(nonmaxima\_image)  for threshold\_name, threshold\_value in threshold\_image.items():  cv2.imwrite(path + "/"+str(fname)+ "\_"+threshold\_name+".bmp", threshold\_value)   print("Canny Edge Detector implemented and output images stored in the directory!") |
| --- |
|  |

**INPUT IMAGES:**

****

**house.bmp**

****

**test\_patterns.bmp**

**OUTPUT IMAGE:**

| **GAUSSIAN SMOOTHING** | **HORIZONTAL GRADIENT** |
| --- | --- |
|  |  |
| **VERTICAL GRADIENT** | **GRADIENT MAGNITUDE** |
|  |  |
| **NON MAXIMA SUPPRESSION** | **THRESHOLDING AT 25TH PERCENTILE** |
|  |  |
| **THRESHOLDING AT 50TH PERCENTILE** | **THRESHOLDING AT 75TH PERCENTILE** |
|  |  |

**OUTPUT IMAGES:**

| **GAUSSIAN SMOOTHING** | **HORIZONTAL GRADIENT** |
| --- | --- |
|  |  |
| **VERTICAL GRADIENT** | **GRADIENT MAGNITUDE** |
|  |  |
| **NON MAXIMA SUPPRESSION** | **THRESHOLDING AT 25TH PERCENTILE** |
|  |  |
| **THRESHOLDING AT 50TH PERCENTILE** | **THRESHOLDING AT 75TH PERCENTILE** |
|  |  |