Assignment 1 : Basic Image Editor

AKELLA VENKATA SURYA SRINIDHI
Roll Number - 213079003
EE1 - Communication Engineering
Department of Electrical Engineering
Indian Institute of Technology Bombay

***Abstract*—This report aims to demonstrate some basic image processing algorithms introduced in class and create a GUI-based platform that integrates several image processing functionalities. Using Python, a simple user interface is designed and is accessible on all devices.**

I. INTRODUCTION

Image processing entails applying some operations to images to extract their features and hidden details. Implementing an algorithm becomes effective if the result of the process is known. This assignment aims to create an efficient GUI-based platform that includes a comprehensive set of tools to explore various techniques for image processing.

PyQt5, a Python library, is used for designing the GUI interface. The editor consists of two types of features - primary features supporting the image processing techniques, and secondary features needed for image manipulation.

The primary features involved are:

1) Histogram equalization
2) Gamma correction
3) Log transformation
4) Image blurring
5) Image sharpening

The secondary features involved are:

1) Load the image
2) Undo the last operation
3) RESET (undo all changes)
4) Save the image
5) Close and exit

The additional features added are:

1) Image negative
2) Edge detection using the Sobel operator.

The image editor supports both Gray-scale and colour images. Colour images are transformed into Value(V) channel, equivalent to the colour image in HSV space. Upon completion of the processing, the V channel is converted back to RGB space for subsequent applications.

The report consists of the following sections:

- Section-2 includes a glossary of requirements for tackling the problem based on the background readings and findings.
- Section-3 describes the development process of the editor and briefly summarizes the techniques used to create it.
- Section-4 presents the results of each technique.
- Section-5 highlights the challenges faced and potential solutions possible if given more time.

II. BACKGROUND READING

In addition to the various online resources, the class notes, lecture slides, and the book "Digital image processing" by Rafael C. Gonzalez and Richard E. Woods were extremely helpful in providing all the related concepts pertaining to each technique.
The main areas of focus are:

- GUI
  As someone who has no prior knowledge of GUI programming but is good at Python programming, I looked for various modules like Tkinter and PyQt used for GUI programming. Due to PyQt's advanced features and versatility, I have chosen to build the editor using it due to its ease of use, smooth coding, and better layout.
- COLOR MODELS
  Aside from the various colour models available, RGB and HSV are primarily focused. The RGB model of colour displays colours as a collection of primary spectral components of RED (R), GREEN (G), and BLUE (B). It is a 3D colour model based on a cartesian coordinate system. HSV (hue, saturation, value) is a cylinder-shaped colour model that remaps RGB primary colours into human-understanding dimensions.

III. APPROACH

The packages and modules used are:

- OpenCV for reading/writing of images and color space conversion from RGB to HSV.
- NumPy for array operations and vectorization.
- PyQt5 (version 5) for GUI.
- Matplotlib for plotting the histogram of images.

A. GUI Designing

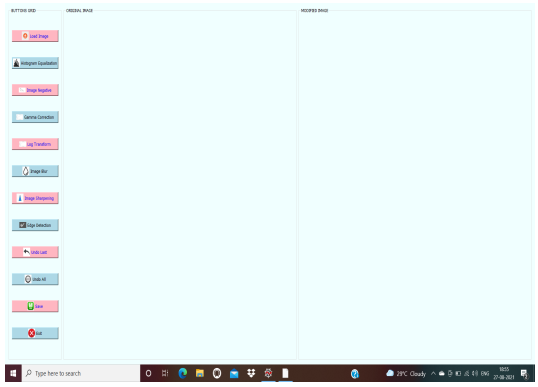
The editor window has been allowed to take the entire desktop. The GUI layout consists of three sections:

1) **Buttons grid** - provides the list of buttons used to perform the operations on images.

2) **Original image** - always displays the input image on which the manipulations are to be done.

3) **Modified image** - displays the processed image.

A screenshot of the application is provided in Figure 1.



(a)

Fig. 1: Image Editor Layout

The Buttons grid consists of the following buttons:

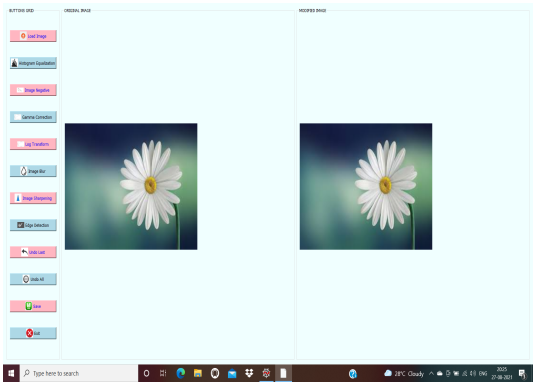
1) Load Image: opens an input dialog box for image selection.
2) Histogram Equalization: performs histogram equalization on the image.
3) Image Negative: displays the image negative.
4) Gamma Correction: An option is provided for the user to input the gamma value and perform gamma correction.
5) Log Transformation: performs log transformation.
6) Image Blurring: uses the Gaussian kernel for blurring the image after asking the user for the filter size and sigma.
7) Image Sharpening: sharpens the image using “Unsharp Masking”.
8) Edge Detection: performs edge detection using Sobel operator.
9) Undo last: undo the last operation performed on image.
10) Undo all: undo all the operations performed on the image.
11) Save: allows the user to save the modified image.
12) Exit: closes and exits the Image Editor.

B. *Detailing each operation*

*1) **Loading an image**:* Upon clicking the ”Load Image” button, a file selector dialog opens, prompting the user to select the input file.

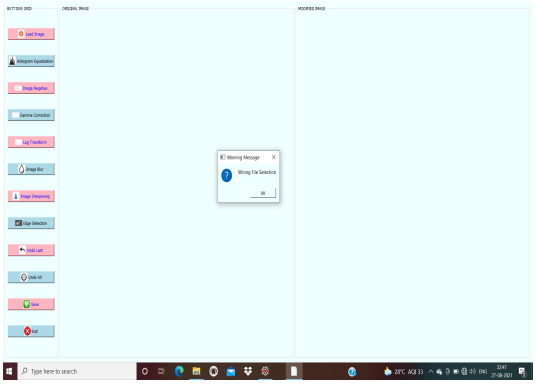
- Files with the extensions ***.jpg, *.png, *.jpeg, *.bmp** are considered for processing. It displays a warning if the file is not compatible with these extensions.
- After taking the image file, it converts the RGB image to an HSV image and only uses the V channel for the processing techniques, with H, S channels untouched.
- After loading the image into the program, it displays the original image in the ”Original Image” grid and the processed image in the ”Modified Image” grid.(Initially, when the image is selected, both the grids display the loaded image since no manipulations have yet been performed.)

Figure 2 and Figure 3 depict screenshots of the application.



(a)

Fig. 2: Loaded Image



(a)

Fig. 3: Wrong File Selection

*2) **Histogram Equalization**:* The histogram equalization technique adjusts the contrast of an image by evenly distributing its pixels over its entire intensity range (0-255, considering the loaded image is an 8-bit image).

- The histogram equalization is performed in this editor using a cumulative distribution function (CDF). The histogram of an equalized image will have a CDF of a straight line.
- Thus, this technique aims to map the image intensities carefully to have an approximate linear CDF for the output / processed image.

The following steps are followed:

(i) Find the histogram of the input image in the intensity range of [0-255] (total intensity levels L = 256) by counting the number of pixels of each intensity.
(ii) Compute the probability distribution by dividing each intensity count by the total number of pixels.

$$p_r(r_k) = \frac{1}{MN}\sum_{j=0}^{k} n_j$$

where:

r_k = kth intensity of input image
MN = total number of pixels of input image
n_j = number of pixels

(iii) Calculate the CDF from the probability distribution.

$$CDF = \sum_{j=0}^{k} p_r(r_j)$$

(iv) Take the output image as a zero-filled image with the dimensions of the input image.
(v) Update the intensities of output image with the rounded values of CDF scaled by 255(i.e., L-1=255). Rounding is performed as the intensities need to be discrete.

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j) = (L-1) \times CDF$$

where:

s_k = kth intensity of output image
$T(r_k)$ = performing transformation(T) on kth intensity of input image

(vi) Find the equalized histogram by computing the CDF of output image and rounding off the CDF scaled with 255.
(vii) Both the original and equalized histograms are plotted and the resulting image is displayed in the "Modified Image" grid.

*3) **Image Negative**:* Each pixel of the image is subtracted from the maximum intensity value to produce the image negative. Negative images feature the darkening of lighter elements and the lightening of darker ones.
In this editor, the following steps are performed:

(i) The input image is initially normalized so that the intensity range is [0-1]. Hence, the maximum intensity is one.
(ii) The output intensities are then calculated using the following mathematical expression:

$$s = c(1-r)$$

where:

c = correction factor used to maintain the output image intensities in the range [0-255]
s = output intensity
r = input intensity

(iii) The output image intensities are then normalized by dividing it by its maximum intensity value (making the output intensity range of [0-1]) and then scaled by 255 (making the output intensity range in [0-255]).
(iv) The image negative is then displayed in the "Modified Image" grid.

*4) **Gamma Correction**:* In gamma correction, pixels are enhanced selectively according to their gamma value, controlling the brightness of the overall image.
For $\gamma < 1$, the values of darker pixels are expanded and for $\gamma > 1$, the values of brighter pixels are expanded.
The following steps are performed:

(i) Clicking the "Gamma Correction" push-button opens a dialog box asking for the gamma value.
(ii) It then normalizes the input image so that its intensities fall between [0-1].
(iii) Next, the intensities of this normalized image are raised to the power of gamma (determined by the user) to obtain the output image intensities by the following mathematical expression,

$$s = cr^{\gamma}$$

where:

c = gamma correction factor used to maintain the output image intensities in the range [0-255]
s = output intensity
r = input intensity

(iv) The output image intensities are then normalized by dividing it by its maximum intensity value (making the output intensity range of [0-1]) and then scaled by 255 (making the output intensity range in [0-255]).
(v) The gamma-corrected image is then displayed in the "Modified Image" grid.

*5) **Log Transformation**:* A log transform enhances the contrast of dark pixels in the image and compresses the brighter ones.
In this editor, the following steps are performed:

(i) The input image is initially normalized and thus has a range of intensity [0-1].
(ii) The output intensities are then calculated using the following mathematical expression:

$$s = c\log(1+r)$$

where:

c = correction factor used to maintain the output image intensities in the range [0-255]
s = output intensity
r = input intensity

(iii) One is added to the log since a pixel intensity of zero raises a problem of log(0).
(iv) The output image intensities are then normalized by dividing it by its maximum intensity value (making the output intensity range of [0-1]) and then scaled by 255 (making the output intensity range in [0-255]).
(v) The log-transformed image is then displayed in the "Modified Image" grid.

*6) **Image Blurring**:* It is generally possible to analyse an image if all of its objects are identified. We can obtain a sharp and clear image by reducing the contents of edges and allowing smooth transitions between colours. Blurring can help achieve this result.

- An image can be blurred by spatial filtering techniques such as mean filters, weighted average filters (box filters), Gaussian filters, etc., and by frequency domain operations such as low pass filters.
- Blurring is implemented in this application using a Gaussian filter in the spatial domain.

Following are the steps:

(i) Clicking on the "Image Blur" push-button opens a dialog box asking for the size of the filter.
(ii) The filter size must be odd so that the neighbourhood around the centre pixel is symmetric. As a result, if the user enters an even value for the filter size, a warning appears asking for an odd value.
(iii) Then another dialog box is opened ,when given an odd value, asking for the value of sigma.
(iv) A Gaussian kernel is created based on the values of filter size and sigma using the following expression,

$$G(x,y) = \frac{1}{2\pi\sigma^2}\exp(-\frac{x^2+y^2}{2\sigma^2})$$

where:
(x, y) = location of pixel of the image.
(v) The Gaussian kernel is normalized by dividing the intensities of all pixels by their sums.
(vi) The output image is created as a zero-filled image with the same dimensions as the input image.
(vii) A zero padding is applied to the input image based on the filter size.
(viii) Convolution of the padded image and the Gaussian kernel yields the output image intensities.
(viii) "Modified Image" grid then displays the blurred image output.

*7) **Image Sharpening**:* Sharpening an image increases its contrast between dark and bright pixels to reveal its hidden details.Sharpening generally refers to a high pass filter. Thus a digital image can be sharpened by digital differentiation. Sharpening of the image is done by **"Unsharp Masking"**. The following steps are performed:

(i) When the "Image Sharpening" push-button is clicked by the user, a dialog box appears asking the user for the filter size of gaussian kernel.
(ii) Blurring is a basic image processing technique and the method of sharpening using "Unsharp masking" primarily involves blurring of the image.
(iii) If the given filter size is odd, then two dialog boxes are opened asking the user for sigma and scaling constant. Otherwise, it raises an error asking for odd size of filter.
(iv) The blurring is then performed by taking these inputs, forming a Gaussian kernel, performing convolution of the input image and Gaussian kernel and producing a blurred image.
(v) The blurred image is subtracted from the input image to get the mask. This is represented mathematically as follows:

$$g_{mask}(x,y) = f(x,y) - (f(x,y) * h(x,y))$$

where:
$f(x, y)$ = input image
$h(x, y)$ = blurred image
$g_{mask}(x, y)$ = mask
$*$ = convolution operator
(vi) This mask image is scaled by the scaling constant "k" given by the user and added to the input image. This leads to a sharpened image.
(vii) This is given mathematically as:

$$g(x,y) = f(x,y) + k \times (g_{mask}(x,y))$$

where:
$g(x, y)$ = sharpened image
(viii) The scaling constant
k = 1 refers to "Unsharp Masking"
$k > 1$ refers to "High boost filtering"

*8) **Edge Detection**:* The edges of a picture are areas where there is a sudden change in the pixel intensity. The edge detection process reduces the information, preserving the relevant structural characteristics while removing unnecessary details.

- Edge detection using Sobel Edge detection is widely used in image processing. Edges are detected independently in both the X and the Y directions by the Sobel Edge Detection technique, and the results are combined to represent the sum of edges in both directions.

A Sobel operator-based algorithm for edge detection is presented below.

(i) Initially, the image is transformed to grayscale.
(ii) With a fixed filter size of 3 and a standard deviation of 1.6, a Gaussian kernel is created.
(iii) Convolution of the grayscale image and the Gaussian kernel is performed next to produce the blurred image.
(iv) Following is the matrix of the Sobel kernel for measuring the image in the x-direction.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

(v) The following matrix describes how the Sobel kernel measures the image in the y-direction.

$$K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

(vi) After the blurred image is obtained, it is convolved with the x-direction kernel to calculate the x gradient.

(vii) Similarly, the blurred images are then convolved with a y-direction kernel to provide the y gradient.
(viii) Combining both x and y gradients yields the resulting gradient image.

$$G = \sqrt{G_x^2 + G_y^2}$$

where:
G_x = x gradient
G_y = y gradient
G = resulting gradient

(ix) Scaling the normalized gradient image with 255 yields the edge detected image. This is displayed on the "Modified Image" grid.

*9) **Undo the Last Operation**:* By clicking this button, you are undoing the previous operation and viewing the previous image. As a result, the previous and current images are swapped.
Since all the operations are performed on the V-channel with H,S channel unchanged, following steps are taken for implementing this feature :

(i) The current V-channel image is combined with its H, S channels to produce the HSV image.
(ii) This HSV image is converted to RGB and stored in a temporary variable.
(iii) The previous image is displayed in the "Modified Image" grid.
(iv) The previous image is then converted from RGB to HSV, and its V-channel becomes the current image..
(v) The temporary variable is assigned to the previous image variable.

*10) **Undo All changes**:* This operation reverts the image to its original form by resetting all operations performed on the image.
This feature is implemented as follows:

(i) The present V-channel image is combined with its corresponding H, S channels to form the HSV image.
(ii) This HSV image is converted to RGB and stored as the previous image.
(iii) The original image is then displayed in the "Modified Image" grid.
(iv) The original loaded image is converted to HSV and its V-channel stored as present image.

*11) **Saving the image**:* The user can select the location of the image to save by clicking the "Save" button. For this operation, first, the image is converted to HSV and then to RGB, then saved using OpenCV.

IV. SELECTION & RESULTS ON TEST IMAGES

A. *Histogram Equalization*

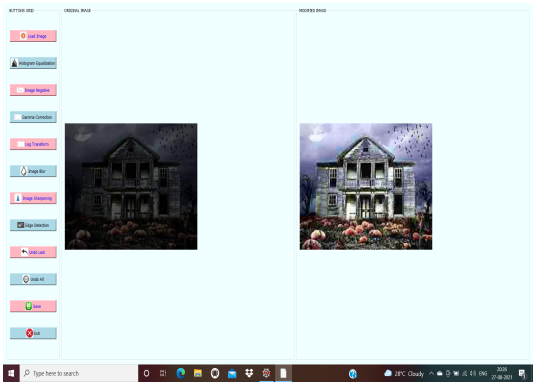
Typically, it is more prevalent on images whose histogram distribution is non-uniform, for example, images where the majority of pixels are dark (such as a dark image) or the ones in which are bright (such as a bright image). Thus, the test images were either very dark or very bright, such that histogram equalization in output renders more information

(a) Input Image

(b) Histogram Equalized Image

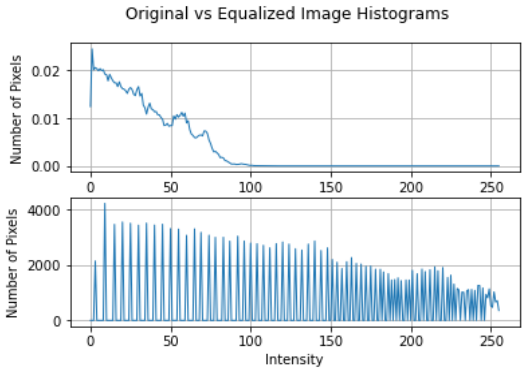
Fig. 4: Comparison of input and histogram equalized images

(visually) than in input.
In this figure we can observe that the input image has some hidden details. Applying histogram equalisation revealed the hidden details.
The histograms of input and equalised images are shown below:



(a)

Fig. 5: Screenshot of Application



(a)

Fig. 6: Comparision of histograms

B. *Image Negative*

A photographic negative is created by reversing the intensities of the pixels. When used with a gray-scale image, this processing emphasizes the details in darker regions within it. Thus the following image is tested:

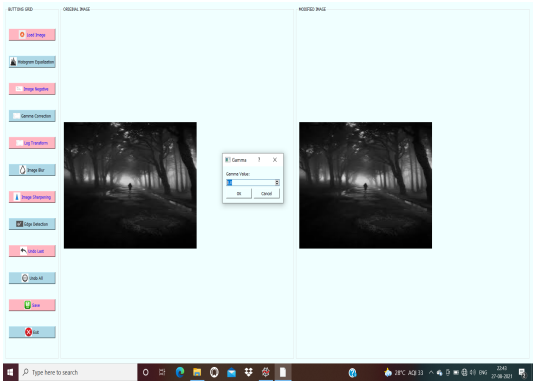
(a) Input Image (b) Image Negative

Fig. 7: Image Negative

C. *Gamma Transformation*

In gamma transform, choosing $\gamma < 1$ results in darker intensity pixels stretched, making the overall image look brighter than the input image, and $\gamma > 1$, on the other hand, causes higher intensities to stretch, making the overall image appear darker.

The screenshot of the application after clicking "Gamma Correction" button and the resulting tested images are as follows:



(a)

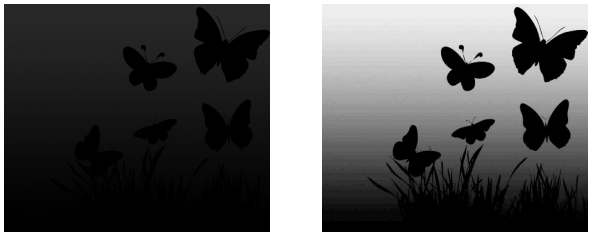
Fig. 8: Screenshot of Application Prompting Gamma value

(a) Input Image (b) Gamma Corrected Image (gamma = 0.5)

Fig. 9: Results of Gamma Correction

D. *Log Transformation*

During this process, darker pixels become more intense, making the overall image appear brighter than the input image. Therefore, a dark image with some hidden details is tested and becomes more appealing after transformation.

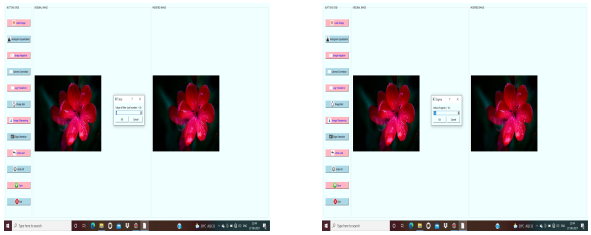


(a) Input Image (b) Log Transformed Image

Fig. 10: Results of Log Transformation

E. *Image Blurring*

Blurring will be most noticeable in sharp images with a lot of texture detail, edge information, and noisy images. Hence, the following pictures portray the degree of blurring.



(a) Prompting kernel size (b) Prompting Sigma Value

Fig. 11: Screenshots of Application

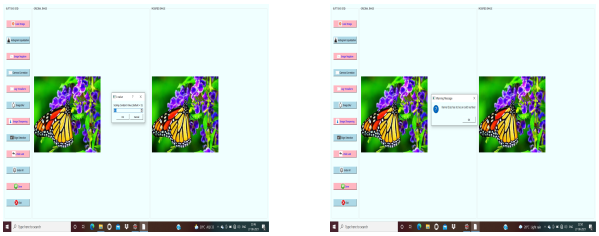
(a) Input Image (b) Blurred Image(sigma = 1.5, filter size = 3)

(c) Blurred Image(sigma = 3, filter size = 5) (d) Blurred Image(sigma = 4, filter size = 7)

Fig. 12: Results of Blurring

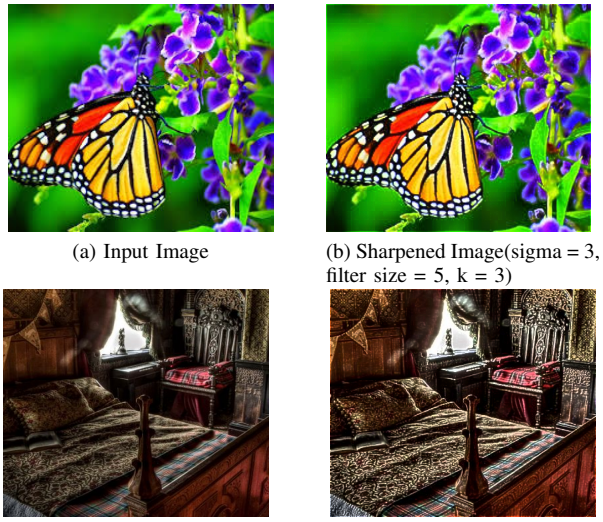
F. *Image Sharpening*

Adding sharpening to images that are out of focus or blurry yields the best results. Therefore, the figures below serve as examples of the sharpening process. Both these images are blurred, resulting in images that are more visually stimulating.



(a) Prompting Scaling Constant (b) Giving Even filter size

Fig. 13: Screenshots of Application



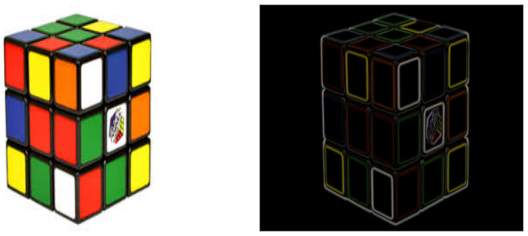
(a) Input Image (b) Sharpened Image(sigma = 3, filter size = 5, k = 3)

(c) Input Image (d) Sharpened Image(sigma = 3, filter size = 5, k = 5)

Fig. 14: Results of Sharpening

G. *Edge Detection*

As compared to other features, this feature provided a more intuitive method of selecting testing images. Images with abrupt intensity transitions are best for demonstrating edge detection.



(a) Input Image (b) Edge Detected Image

Fig. 15: Results of Edge Detection

V. DISCUSSION AND CHALLENGES FACED

A. *CHALLENGES FACED*

The major challenges encountered during its implementation were:

- **GUI Programming**: Having no hands-on experience with GUI programming and having a great deal of difficulty grasping the complicated features available in PyQt5, the time required for implementation decreased.
- **Convolution processing time**: In this case, the processing time for convolution was quite long, which resulted in the application being less responsive. By using the window, instead of looping over the image, the number of ”for” loops reduced. With large windows, however, the performance still is slow.
- **Unwanted dark pixels in sharpened images**: The reasons for dark spots appearing on sharpened, especially color images, took a lot of time and effort to understand. Out-of-range pixels were found to be the cause (values < 0 and values > 255). The problem was resolved by setting the pixel intensities < 0 to zero and those > 255 to 255.
- **Displaying images**: Displaying images with QPixmap was the most challenging task.Having no detailed documentation caused many problems with QPixmap. To use the QPixmap, the image had to be converted to Qimage using OpenCV.

B. *FUTURE IMPROVEMENTS POSSIBLE*

If I had more time, I would like to do the following:

- Display the original and equalized histograms inside the main editor window.
- Experiment with the implementation of DFT and FFT algorithms.
- Analyse the effects in the output of frequency domain and spatial domain filtering.
- Control the extent of blurring and sharpening using a slider and observe the variations at a stretch.

REFERENCES

[1] R. C. Gonzalez, R. E. Woods, Digital Image Processing, 3rd ed. Prentice Hall

[2] A.Sethi, Intensity Transform and Spatial Filtering, Lecture Notes

[3] PyQt5 Official Website - https://pythonspot.com/pyqt5/

[4] PyQt5 Tutorials Point - https://www.tutorialspoint.com/pyqt/index.htm

[5] PyQt5 Tutorial Videos - https://www.youtube.com/watch?v=Fk1TBoBcrR4

[6] Convert openCV Image to QImage - https://gist.github.com/smex/5287589

[7] Writing in Latex - http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf

VI. Appendix

A. *main.py*

```

'''import all the necessary libraries
1) OpenCV for reading,writing of images and color space conversion from RGB to HSV.
2) NumPy for array operations and vectorization.
3) PyQt5 (version 5) for GUI.
4) Matplotlib.pyplot for plotting the histogram of images.
'''
import os
import sys,cv2
import numpy as np
import matplotlib.pyplot as plt
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *
cwd = os.getcwd()

'''
Any image is represented in the form of RGB format with each colour having red,green
and blue components(red,green,blue).
This RGB triplet (red,green,blue) is represented by QRgb.
qRgb function creates and return the QRgb triplet values.
'''
gray_color_table = [qRgb(i, i, i) for i in range(256)]

class Editor(QWidget):
    def __init__(self):
        super().__init__()
        self.title = '213079003-Image-Editor'  # declares the editor-window's title
        self.number_grids = 3
        self.initialise_User_Interface()   # initialises the User Interface


    def initialise_User_Interface(self):
        self.setWindowTitle(self.title)
     # sets the editor-window's title as defined
        geometry = app.desktop().availableGeometry()
     # the window is set to the size of the screen i.e. full-screen mode.
        self.setGeometry(geometry)
        self.setStyleSheet("background-color : Azure")
        windowLayout = QHBoxLayout()       # sets the geometry of the window

# The entire layout is divided into three grids.
# 1) First grid - provides the list of buttons
# 2) Second grid - for displaying the original image
# 3) Third grid - for displaying the processed image.

# The entire layout is taken as an empty list initially and three grids are appended to it.
        self.layout = []
        number_layouts = 3
        for i in range(number_layouts):
            self.layout.append(QGridLayout())

# adjusts the grid dimensions
        self.layout[1].setColumnStretch(1, 2)
        self.layout[2].setColumnStretch(1, 2)

# defining the push buttons using QPushButton widget
        button_1 = QPushButton('Load Image', self)
        button_2 = QPushButton('Histogram Equalization',self)
        button_3 = QPushButton('Image Negative',self)
        button_4 = QPushButton('Gamma Correction',self)
        button_5 = QPushButton('Log Transform', self)
        button_6 = QPushButton('Image Blur', self)
        button_7 = QPushButton('Image Sharpening', self)
        button_8 = QPushButton('Edge Detection', self)
        button_9 = QPushButton('Undo Last', self)
        button_10 = QPushButton('Undo All', self)
        button_11 = QPushButton('Save', self)
        button_12 = QPushButton('Exit', self)

```

```
# setting colours for the push buttons
        button_1.setStyleSheet("color : blue; background-color : lightpink;")
        button_2.setStyleSheet("color : black; background-color : lightblue;")
        button_3.setStyleSheet("color : blue; background-color : lightpink;")
        button_4.setStyleSheet("color : black; background-color : lightblue;")
        button_5.setStyleSheet("color : blue; background-color : lightpink;")
        button_6.setStyleSheet("color : black; background-color : lightblue;")
        button_7.setStyleSheet("color : blue; background-color : lightpink;")
        button_8.setStyleSheet("color : black; background-color : lightblue;")
        button_9.setStyleSheet("color : blue; background-color : lightpink;")
        button_10.setStyleSheet("color : black; background-color : lightblue;")
        button_11.setStyleSheet("color : blue; background-color : lightpink;")
        button_12.setStyleSheet("color : black; background-color : lightblue;")

# setting icons for the push buttons
        button_1.setIcon(QIcon(cwd+'\\load.PNG'))
        button_2.setIcon(QIcon(cwd+'\\histogram.PNG'))
        button_3.setIcon(QIcon(cwd+'\\negative.PNG'))
        button_4.setIcon(QIcon(cwd+'\\gamma.PNG'))
        button_5.setIcon(QIcon(cwd+'\\log.PNG'))
        button_6.setIcon(QIcon(cwd+'\\blurring.PNG'))
        button_7.setIcon(QIcon(cwd+'\\sharpening.JPG'))
        button_8.setIcon(QIcon(cwd+'\edge.PNG'))
        button_9.setIcon(QIcon(cwd+'\\Undo.PNG'))
        button_10.setIcon(QIcon(cwd+'\\reset1.JPG'))
        button_11.setIcon(QIcon(cwd+'\\save.JPG'))
        button_12.setIcon(QIcon(cwd+'\\exit.PNG'))

# setting dimensions for the push buttons
        button_1.setIconSize(QSize(25,25))
        button_2.setIconSize(QSize(25,25))
        button_3.setIconSize(QSize(25,25))
        button_4.setIconSize(QSize(25,25))
        button_5.setIconSize(QSize(25,25))
        button_6.setIconSize(QSize(25,25))
        button_7.setIconSize(QSize(25,25))
        button_8.setIconSize(QSize(25,25))
        button_9.setIconSize(QSize(25,25))
        button_10.setIconSize(QSize(25,25))
        button_11.setIconSize(QSize(25,25))
        button_12.setIconSize(QSize(25,25))

# each pushbutton upon clicking performs its respective function.
        button_1.clicked.connect(self.load_Image)
        button_2.clicked.connect(self.histogram_Equalization)
        button_3.clicked.connect(self.image_Negative)
        button_4.clicked.connect(self.gamma_Correction)
        button_5.clicked.connect(self.log_Transformation)
        button_6.clicked.connect(self.image_Blurring)
        button_7.clicked.connect(self.image_Sharpening)
        button_8.clicked.connect(self.edge_Detection)
        button_9.clicked.connect(self.undo_Last)
        button_10.clicked.connect(self.undo_All)
        button_11.clicked.connect(self.save)
        button_12.clicked.connect(self.exit)

#adding each button in the first grid of the layout.
#This is performed using the fuction addWidget(button,row,column)
#since all the buttons are added in the first grid, the column= 0
        self.layout[0].addWidget(button_1,0,0),
        self.layout[0].addWidget(button_2,1,0)
        self.layout[0].addWidget(button_3,2,0)
        self.layout[0].addWidget(button_4,3,0)
        self.layout[0].addWidget(button_5,4,0)
        self.layout[0].addWidget(button_6,5,0)
        self.layout[0].addWidget(button_7,6,0)
        self.layout[0].addWidget(button_8,7,0)
        self.layout[0].addWidget(button_9,8,0)
        self.layout[0].addWidget(button_10,9,0)
        self.layout[0].addWidget(button_11,10,0)
        self.layout[0].addWidget(button_12,11,0)

#defining the names for each grid.
# 1)First layout is named as  Buttons Grid.
```

```
# 2)Second layout is named as  Original Image.
# 3)Third layout is named as  Modified Image.

# The labels is taken as an empty list initially and three labels names are appended to it.
        grid_labels = ["BUTTONS GRID", "ORIGINAL IMAGE", "MODIFIED IMAGE"]
        self.labels=[]
        for i in range(self.number_grids):
            self.labels.append(QGroupBox(grid_labels[i]))
            self.labels[i].setLayout(self.layout[i])

#adding the defined labels to the grids.
        for i in range(self.number_grids):
            windowLayout.addWidget(self.labels[i])

        self.setLayout(windowLayout)  # sets the layout of editor-window to be a windowLayout
        self.show()                   # displays the editor-window on screen

# initialize the pyqt QLabel widget for always displaying the original Image
        self.label = QLabel(self)

# the pyqt QLabel widget for always displaying the modified Image.
# Initialized to original image at the beginning.
# So both the Original and Modified image grids display the original image initially.
        self.label_result = QLabel(self)

# both the declared label widgets are added to 2nd & 3rd grids respectively.
        self.layout[1].addWidget(self.label)
        self.layout[2].addWidget(self.label_result)

# function to display the modified (output) image.

    def displayOutput(self, output):

         # combine V-channel of input image(self.inputImage) with the H,S channels.
        self.hsvImage[:,:,2] = self.inputImage
         # storing the input image(in RGB format) as previous image
        self.previousImage = cv2.cvtColor(self.hsvImage, cv2.COLOR_HSV2RGB)
         # combine V-channel of output image (output) with the H,S channels.
        self.hsvImage[:,:,2] = output
         # update the image display with the modified RGB image
        qImage = self.CV_to_QImage(cv2.cvtColor(self.hsvImage, cv2.COLOR_HSV2RGB))
         # display the output image
        self.label_result.setPixmap(QPixmap(qImage))
         # set the output as input image.
        self.inputImage = output

# function for selecting the image.
    def openFile(self):
        fileLoadOptions = QFileDialog.Options()
        fileLoadOptions |= QFileDialog.DontUseNativeDialog
        fileName, _ = QFileDialog.getOpenFileName(self, "Load Image File")
# if the user selects a file with extensions .jpg, .png, .jpeg, .bmp , .JPG, .PNG', .JPEG, .BMP , then it
    is an allowed file.
# if user selects an allowed file
        if fileName.split('.')[1].lower() in ['jpg', 'png', 'jpeg', 'bmp']:
            # reads the selected file(i.e.image) and resizes to (image_width, image_height)=(480, 360)
            self.originalImage = cv2.resize(cv2.imread(fileName, 1), (480, 360))

            # OpenCV reads an image in BGR format..and thus convert into RGB for further processing
            self.originalImage = cv2.cvtColor(self.originalImage, cv2.COLOR_BGR2RGB)

            # previous image variable for "Undo" Operation and currently set to the original image.
            self.previousImage = self.originalImage

            # convert image format from RGB to HSV for operations on V channel
            self.hsvImage = cv2.cvtColor(self.originalImage, cv2.COLOR_RGB2HSV)

            # inputimage = V channel of original HSV image
            self.inputImage = self.hsvImage[:, :, 2].astype(np.uint8)

            # convert OpenCV image into PyQt QImage for displaying in Pixmap
            qImage = self.CV_to_QImage(self.originalImage)

            # set pixmap with the original loaded image in both original and modified grids
```

```
            self.label.setPixmap(QPixmap(qImage))
            self.label_result.setPixmap(QPixmap(qImage))

#  self.originalImage is the original selected image in RGB format
#  self.previousImage is the variable used to store the previous image in RGB format
#  self.hsvImage is variable used to store the image in HSV format
#  self.inputImage is the selected image having V-channel

# case when user selects a file with other extensions and raises an error asking for proper file
        elif (fileName):
            buttonReply = QMessageBox.question(self, 'Warning Message', "Wrong File Selection", QMessageBox
    .Ok, QMessageBox.Ok)

# converts an OpenCV image into  a QImage.
    def CV_to_QImage(self, img, copy=False):
    #input image must be in OpenCV image format (np.uint8)

      # if input image format is OpenCV image format np.uint8
        if img.dtype == np.uint8:
           # grayscale images or the images having two dimensions [height, width]
            if len(img.shape) == 2:
                qim = QImage(img.data, img.shape[1], img.shape[0], img.strides[0], QImage.Format_Indexed8)
                qim.setColorTable(gray_color_table)
                return qim.copy() if copy else qim
            # images having three dimensions [height, width, number of Channels]
            elif len(img.shape) == 3:
                # if the image has three channels
                if img.shape[2] == 3:
                    qim = QImage(img.data, img.shape[1], img.shape[0], img.strides[0], QImage.Format_RGB888
    )
                    return qim.copy() if copy else qim
                # if the image has four channels
                elif img.shape[2] == 4:
                    qim = QImage(img.data, img.shape[1], img.shape[0], img.strides[0], QImage.Format_ARGB32
    )
                    return qim.copy() if copy else qim


    def correlation(self, image, window):
        output = np.zeros(image.shape,dtype=np.uint8)
        # find no of rows and columns in the ndarray of image passed
        img_height = image.shape[0]
        img_width = image.shape[1]

        # compute window size from window passed
        window_size = window.shape[0]
        # compute zero padding requirements and offset to be used during convolution
        zero_padding = window_size - 1
        offset = int(zero_padding / 2)

        # create the zero padded image
        paddedImage =  np.zeros((img_height + zero_padding, img_width+ zero_padding),dtype=np.uint8)
        paddedImage[offset: img_height + offset, offset:img_width +offset] = image

        # loop through the window elements
        # computes correlation as shifted sum of image elements by keeping  window stationary

        for i in range(img_height):
                for j in range(img_width):
                    product = paddedImage[i:i + window_size, j:j + window_size]*window
                    output[i,j] = np.round(np.sum(product))

        #  return the computed image
        return output

    def convolution(self, image, window):
        # to perform convolution, flip the window and compute correlation
        window = np.flipud(np.fliplr(window))
        output = self.correlation(image, window)
        #  return the computed image
        return output
```

```
 # opens a dialog box for the user to select an Image
    @pyqtSlot()
    def load_Image(self):
        self.openFile()

# function performing histogram equalization
    @pyqtSlot()
    def histogram_Equalization(self):
     # Histrogram Equalization Function has the following algorithm:
     # 1. Finds the histogram of input image.
     # 2. Computes the probability distribution of each intensity.
     # 3. Computes the cumulative distribution of input image.
     # 4. Assign the new intensity values to pixels by scaling the cdf by 255.
     # 5. Computes the equalised histogram.
     # 6. Plots the input image histogram and equalized image histogram.

# considering a 8-bit image, the total number of discrete intensity levels are 256
        L = 256
        img_height, img_width = self.inputImage.shape # variables storing the image dimensions
        outputImage = np.zeros(self.inputImage.shape)
        # numpy array of zeros for output "histogram equalized" image
        original_img_hist = np.zeros((L, 1))    # variable to store the input image histogram
        equalized_img_hist = np.zeros((L, 1))   # variable to store the equalized histogram

        # Compute the histogram of input image
        for i in range(L):
            original_img_hist[i, 0] = np.sum(self.inputImage == i) # counting the number of pixels of each
    intensity.

        # Compute the probability distribution by dividing each intensity count by the total number of
    pixels
        original_img_hist = original_img_hist/ (img_height*img_width)

        # Initialize the cdf of input image as zero
        cdf = np.zeros(original_img_hist.shape)
        sum_Hist = 0 # variable to store the sum of previous probabilities.
        for i in range(L):
            sum_Hist+= original_img_hist[i,0]
            cdf[i,0] = sum_Hist # updates the cdf with the sum of probabilities

        ''' compute the transform values for each intensity from [0-255] and assign it
       to the pixels locations where that intensity is present in input image
       and the output image is found.'''

        for i in range(L):
            outputImage[np.where(self.inputImage == i)] = np.round(((L-1))*cdf[i])

        # compute the equalized histogram of output image
        for i in range(L):
            equalized_img_hist[i, 0] = np.sum(outputImage == i)

        outputImage = outputImage.astype(np.uint8)  # change output image type for display purposes

        # plotting the input and ouput histograms

        plt.subplot(2,1,1)
        plt.plot(original_img_hist, linewidth=1)
        plt.xlabel('Intensity')
        plt.ylabel('Number of Pixels')
        plt.grid(True)
        plt.subplot(2,1,2)
        plt.plot(equalized_img_hist, linewidth=1)
        plt.xlabel('Intensity')
        plt.ylabel('Number of Pixels')
        plt.grid(True)
        plt.suptitle('Original vs Equalized Image Histograms')
        plt.show()

        self.displayOutput(outputImage) # display the output image

# function performing image negative
    @pyqtSlot()
    def image_Negative(self):
        # convert image from intensity range [0-255] to [0-1]
```

```
        normalised_Image = cv2.normalize(self.inputImage.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)
        outputImage = 1 - normalised_Image     # finding output intensities
        max_intensity = np.max(outputImage)    # finding the maximum intensity value  of output image
        outputImage = (outputImage/max_intensity)*255.0 # bring the output image to [0-255] range
        outputImage = outputImage.astype(np.uint8)   # change output image type for display purposes

        self.displayOutput(outputImage)     # display the output image

# function performing gamma correction
    @pyqtSlot()
    def gamma_Correction(self):

     # opens a dialog asking for gamma value and stores the gamma value in a variable "gamma"
        gamma, ok_is_Pressed = QInputDialog.getDouble(self, "Gamma", "Gamma Value:")
        if ok_is_Pressed:      # compute only if the user presses "OK"
            # convert image from intensity range [0-255] to [0-1]
            normalised_Image = cv2.normalize(self.inputImage.astype('float'), None, 0.0, 1.0, cv2.
    NORM_MINMAX)
            outputImage = np.power(normalised_Image, gamma)     # finding output intensities
            max_intensity = np.max(outputImage)   # finding the maximum intensity value  of output image
            outputImage = (outputImage/max_intensity)*255.0    # bring the output image to [0-255] range
            outputImage = outputImage.astype(np.uint8)     # change output image type for display purposes

            self.displayOutput(outputImage)   # display the output image

# function performing log transformation
    @pyqtSlot()
    def log_Transformation(self):
        # convert image from intensity range [0-255] to [0-1]
        normalised_Image = cv2.normalize(self.inputImage.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)
        outputImage = np.log(1+normalised_Image)    # finding output intensities
        max_intensity = np.max(outputImage)       # finding the maximum intensity value  of output image
        outputImage = (outputImage/max_intensity)*255.0    # bring the output image to [0-255] range
        outputImage = outputImage.astype(np.uint8)   # change output image type for display purposes

        self.displayOutput(outputImage)        # display the output image

# function performing blurring operation
    @pyqtSlot()
    def image_Blurring(self):
        """
            Blur Image using Gaussian Kernel with kernel size and gaussian variance
            chosen by initUserInterface
        """

        #opens a dialog prompting user for filter size
        filter_size, ok_1_pressed = QInputDialog.getInt(self, "Size", "Value of filter (odd number > 0):")
        #if user gives odd size of filter
        if (ok_1_pressed  and filter_size%2!=0):
        # opens a dialog prompting user for sigma
            sigma, ok_2_pressed = QInputDialog.getDouble(self, "Sigma", "Value of sigma (> 0):")
            if (ok_2_pressed):
                # creates a gaussian kernel of given window size and sigma
                kernel = GaussianKernel(filter_size, sigma)
                # perfomes convolution of image and kernel to form the blurred image
                blurredImage = self.convolution(self.inputImage, kernel)
                blurredImage = blurredImage/(kernel.sum())
                blurredImage = blurredImage.astype(np.uint8)# change output image type for display purposes

                self.displayOutput(blurredImage)    # display the output image

        #if user gives even size of filter
        elif (ok_1_pressed  and filter_size%2==0):
            # displays a warning asking for odd value
            buttonReply = QMessageBox.question(self, 'Warning Message', "Kernel Size has to be an odd
    number", QMessageBox.Ok, QMessageBox.Ok)
            # again opens the dialog for filter size and produces the blurred image
            filter_size, ok_1_pressed = QInputDialog.getInt(self, "Size", "Value of filter (odd number > 0)
    :")
            if (ok_1_pressed  and filter_size%2!=0):
                sigma, ok_2_pressed = QInputDialog.getDouble(self, "Sigma", "Value of sigma (> 0):")
                if (ok_2_pressed):

                    kernel = GaussianKernel(filter_size, sigma)
```

```
                    blurredImage = self.convolution(self.inputImage, kernel)
                    blurredImage = blurredImage/(kernel.sum())
                    blurredImage = blurredImage.astype(np.uint8)
                    self.displayOutput(blurredImage)

# function performing sharpening operation
    @pyqtSlot()
    def image_Sharpening(self):
        """
            Sharpen Image using "Unsharp Masking" by using a Gaussian Kernel with kernel size and gaussian
    variance
             and a scaling constant k chosen by initUserInterface
        """
        #opens a dialog prompting user for filter size
        filter_size, ok_1_pressed = QInputDialog.getInt(self, "Size", "Value of filter (odd number > 0):")
        #if user gives odd size of filter
        if (ok_1_pressed  and filter_size%2!=0):
        # opens dialog boxes  prompting user for sigma and k
            sigma, ok_2_pressed = QInputDialog.getDouble(self, "Sigma", "Value of sigma (> 0):")
            k, ok_3_pressed = QInputDialog.getDouble(self, "k value", "Scaling Constant Value (default = 5)
    :")
            if (ok_2_pressed and ok_3_pressed ):
                # creates a gaussian kernel of given window size and sigma
                kernel = GaussianKernel(filter_size, sigma)
                # perfomes convolution of image and kernel to form the blurred image
                blurredImage = self.convolution(self.inputImage, kernel)
                blurredImage = blurredImage/(kernel.sum())
                # converting both blurred and input images into same format
                blurredImage = blurredImage.astype(np.uint8)
                inputimg = self.inputImage.astype(np.uint8)
 # scaled version of mask computed by subtracting the blurred image from original image and scaling by k
                maskImage = k*cv2.subtract(inputimg, blurredImage)
                maskImage = maskImage.astype(np.uint8)
                # sharp image computed by adding scaled mask with original image
                sharpenedImage = cv2.add(self.inputImage, maskImage)
                for i in range(256):
                    sharpenedImage[np.where(i <0)] = 0
                    sharpenedImage[np.where(i >255)] = 255
                sharpenedImage = sharpenedImage.astype(np.uint8) # change output image type for display
    purposes

                self.displayOutput(sharpenedImage) # display the output image
        #if user gives even size of filter
        elif (ok_1_pressed  and filter_size%2==0):
            # displays a warning asking for odd value
            buttonReply = QMessageBox.question(self, 'Warning Message', "Kernel Size has to be an odd
    number", QMessageBox.Ok, QMessageBox.Ok)
            # again opens the dialog for filter size and produces the sharpened image
            filter_size, ok_1_pressed = QInputDialog.getInt(self, "Size", "Value of filter (odd number > 0)
    :")

            if (ok_1_pressed  and filter_size%2!=0):

                sigma, ok_2_pressed = QInputDialog.getDouble(self, "Sigma", "Value of sigma (> 0):")
                k, ok_3_pressed = QInputDialog.getDouble(self, "k value", "Scaling Constant Value (default
    = 5):")
                if (ok_2_pressed and ok_3_pressed ):
                    kernel = GaussianKernel(filter_size, sigma)
                    blurredImage = self.convolution(self.inputImage, kernel)
                    blurredImage = blurredImage/(kernel.sum())
                    blurredImage = blurredImage.astype(np.uint8)
                    inputimg = self.inputImage.astype(np.uint8)
                    maskImage = k*cv2.subtract(self.inputImage, blurredImage)
                    maskImage = maskImage.astype(np.uint8)
                    sharpenedImage = cv2.add(self.inputImage, maskImage)
                    for i in range(256):
                        sharpenedImage[np.where(i <0)] = 0
                        sharpenedImage[np.where(i >255)] = 255
                    sharpenedImage = sharpenedImage.astype(np.uint8)

                    self.displayOutput(sharpenedImage)

# function performing edge detection
    @pyqtSlot()
```

```
    def edge_Detection(self):
        """
            Edge detection is performed using Fixed 3x3 Gaussian kernel with a
            standard deviation of 1.6 and using a 3x3 Sobel Operator

        """


        sigma = 1.6       # standard deviation for gaussian blurring
        filter_size = 3    # kernel size for gaussian blurring

        # combine V-channel (self.image) with H,S channel
        self.hsvImage[:,:,2] = self.inputImage
        # convert into rgb
        img = cv2.cvtColor(self.hsvImage, cv2.COLOR_HSV2RGB)
        # convert into gray-scale
        img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        img = img.astype(np.int32)
        # storing the dimensions of input image
        img_height, img_width = img.shape
        # creates a gaussian kernel of given window size and sigma
        kernel = GaussianKernel(filter_size, sigma)
        # perform convolution of image and kernel to form the blurred image
        zero_padding = filter_size - 1
        offset = int(zero_padding / 2)
        # zero padded image for convolution with gaussian kernel
        paddedImage =  np.zeros((img_height + zero_padding, img_width+ zero_padding),dtype=np.uint32)
        paddedImage[offset: img_height + offset, offset:img_width +offset] = img
        blurredImage = np.zeros(self.inputImage.shape,dtype=np.uint32)

        # loop through the window elements
        # computes correlation as shifted sum of image elements by keeping  window stationary
        for i in range(img_height):
          for j in range(img_width):
            product = paddedImage[i:i + filter_size, j:j + filter_size]*kernel
            blurredImage[i,j] = np.round(np.sum(product))

        # kernel for finding gradient in X & Y direction
        kernel_Gradient_X = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.int32)    # vertical edges
        kernel_Gradient_Y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.int32)    # horizontal edges

        paddedImage[offset: img_height + offset, offset:img_width +offset] = blurredImage
        gradientX = np.zeros(img.shape, dtype=np.int32)
        gradientY = np.zeros(img.shape, dtype=np.int32)

     # convolution for finding gradients
        for i in range(img_height):
            for j in range(img_width):
              horizontal_gradient = paddedImage[i:i + filter_size, j:j + filter_size]*(-1*kernel_Gradient_X
    )
              vertical_gradient = paddedImage[i:i + filter_size, j:j + filter_size]*(-1*kernel_Gradient_Y)
              gradientX[i,j] = np.round(np.sum(horizontal_gradient))
              gradientY[i,j] = np.round(np.sum(vertical_gradient))

        sobel_operated_img = np.sqrt((np.power(gradientX,2))+ (np.power(gradientY,2)))
        for i in range(256):
            sobel_operated_img[np.where(i <0)] = 0
            sobel_operated_img[np.where(i >255)] = 255
        edgeDetectedImage = 255 * cv2.normalize(sobel_operated_img.astype('float'), None, 0.0, 1.0, cv2.
    NORM_MINMAX)

        self.displayOutput(edgeDetectedImage)

# function performing undo operation
    @pyqtSlot()
    def undo_Last(self):
      # combine V-channel (self.image) with H,S channel
      self.hsvImage[:,:,2] = self.inputImage

      # storing the image in a temporary variable
      temp = cv2.cvtColor(self.hsvImage, cv2.COLOR_HSV2RGB)

      #display the previous image
      qImage = self.CV_to_QImage(self.previousImage)
```

```
      self.label_result.setPixmap(QPixmap(qImage))

      # assign the V-channel of previous image to present image after converting it to HSV
      self.inputImage = cv2.cvtColor(self.previousImage, cv2.COLOR_RGB2HSV)[:,:,2]

      # assign present image to previous image variable
      self.previousImage = temp

# function performing reset operation
    @pyqtSlot()
    def undo_All(self):
      # display original image
      qImage = self.CV_to_QImage(self.originalImage)
      self.label_result.setPixmap(QPixmap(qImage))
      # combine V-channel (self.image) with H,S channel
      self.hsvImage[:,:,2] = self.inputImage
      # store the present image as previous image
      self.previousImage = cv2.cvtColor(self.hsvImage, cv2.COLOR_HSV2RGB)
      # assign the V-channel of original image to present image after converting it to HSV
      self.inputImage = cv2.cvtColor(self.originalImage, cv2.COLOR_RGB2HSV)[:,:,2]

# function for saving the image
    @pyqtSlot()
    def save(self):
        fileSaveOptions = QFileDialog.Options()
        fileSaveOptions |= QFileDialog.DontUseNativeDialog
        # open a dialog box for choosing destination location for saving image
        fileName, _ = QFileDialog.getSaveFileName(self,"Save Image File")
        # combine V-channel with H,S channel
        self.hsvImage[:,:,2] = self.inputImage
        #convert HSV to BGR since OpenCV default color format is BGR
        # save image at location specified by fileName
        cv2.imwrite(fileName, cv2.cvtColor(self.hsvImage, cv2.COLOR_HSV2BGR))

# function for closing the application
    @pyqtSlot()
    def exit(self):
        self.close()    # in-built function of QWidget class


#function gor creating gaussian kernel
def GaussianKernel(winSize, sigma):
    """
        generates a gaussian kernel taking window size = winSize
        and standard deviation = sigma as input/control parameters.
        Returns the gaussian kernel
    """
    kernel = np.zeros((winSize, winSize))     # generate a zero numpy kernel
    # finding each window coefficient
    for i in range(winSize):
        for j in range(winSize):
            temp = pow(i-winSize//2,2)+pow(j-winSize//2,2)
            kernel[i,j] = np.exp(-1*temp/(2*pow(sigma,2)))
    kernel = kernel/(2*np.pi*pow(sigma,2))
    norm_factor = np.sum(kernel)               # finding the sum of the kernel generated
    kernel = kernel/norm_factor                # dividing by total sum to make the kernel matrix unit-sum
    return kernel

if __name__ == '__main__':
    app = QApplication(sys.argv)    # defines pyqt application object
    ex = Editor()
    sys.exit(app.exec_())
```