Tram Pham
EE 568
03/12/2021

Final Project Report:

Implementing a real-time Image Processing Demo with a GUI

EE 568 Winter 2021 Final Project Due: 11:59 pm, March 12 Friday, 2021

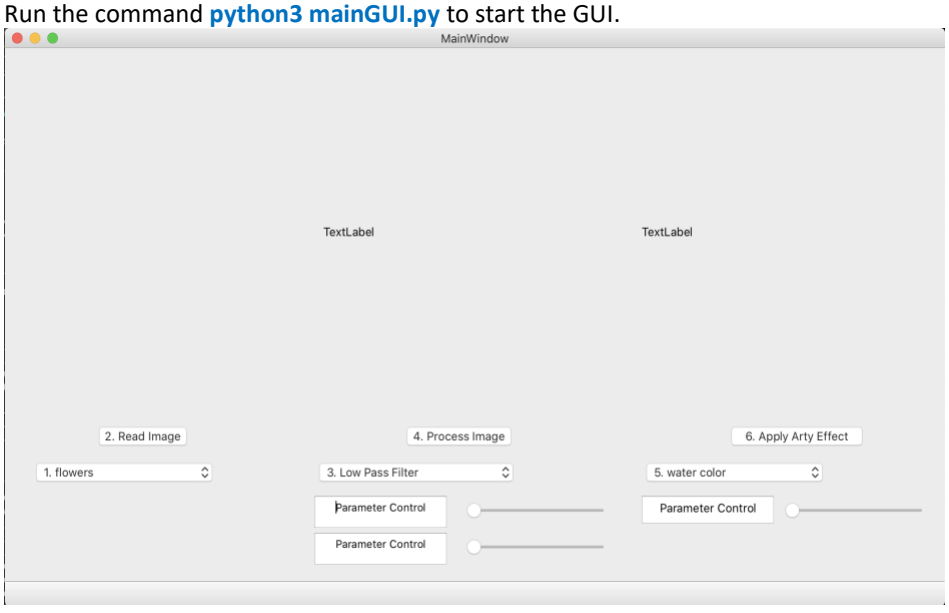
I. Introduction/Objective:

The aim of this project is to demonstrate how image processing techniques are utilized to solve image processing problems and perform a real-time demonstration of an image processing example via a simple graphic user interface (GUI). This demo is built based on my knowledge learned from class and external research. In this project, I will implement several of the image processing techniques, including Specification Histogram Equalization, Image Pyramid (downsize), Gaussian Blurring, Low Pass Filtering, and Edge Detection. Besides, I also implemented the water coloring effect to produce image that looks like a water painting. I intend to use the GUI to analyze the image, see how each parameter affects the result, and produce artist effect without arty work. Users can load an image and perform real-time processing with a small overhead to produce a better image or enhance its quality.

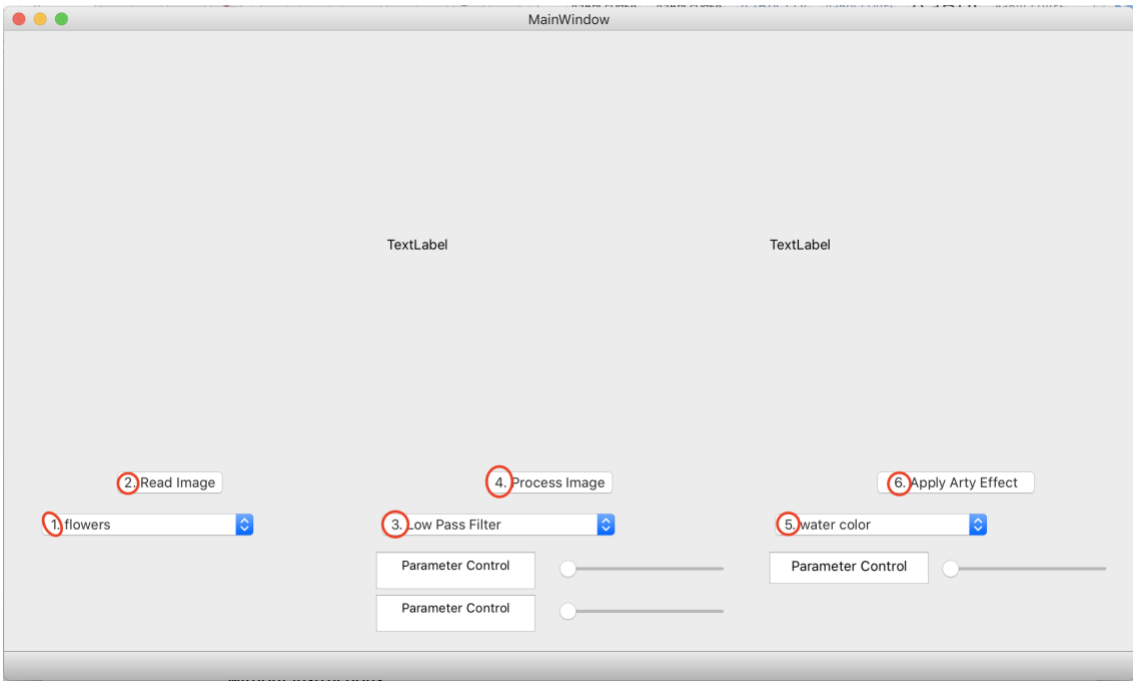
II. Instructions on how to run your program

The GUI can be run by following steps:

1. Download the gui.zip file and unzip it.
2. Navigate to the **terminal** and go to the path contain the file
3. Run the command **python3 mainGUI.py** to start the GUI.



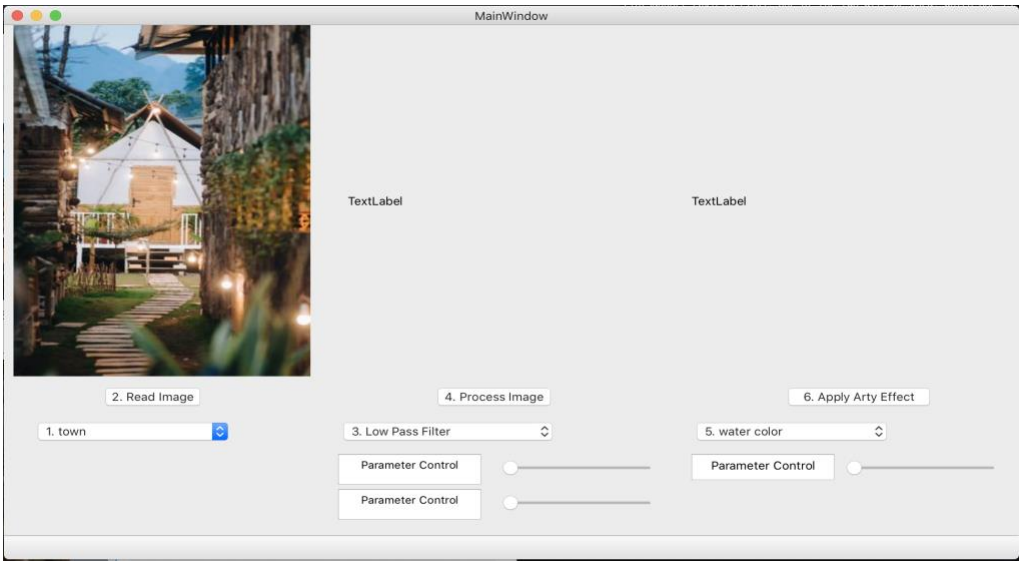
4. You can start using the GUI by following the step label indicated in the GUI



5. Example.

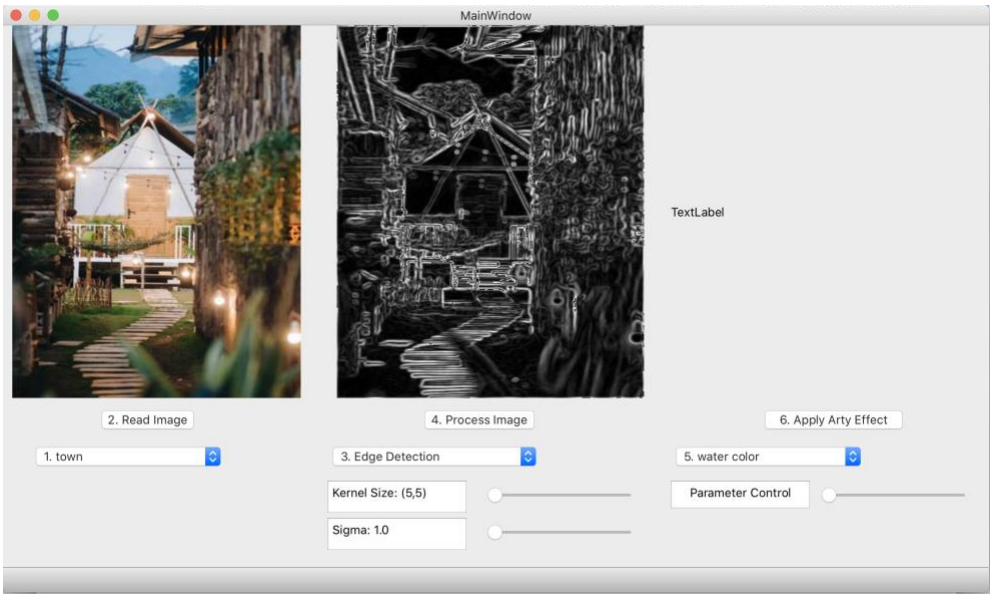
Step 1: Choose “1. town”

Step 2: Click the Read Image button



Step 3: Choose (a processing technique) “3. Edge Detection”

Step 4: Click the Process Image button

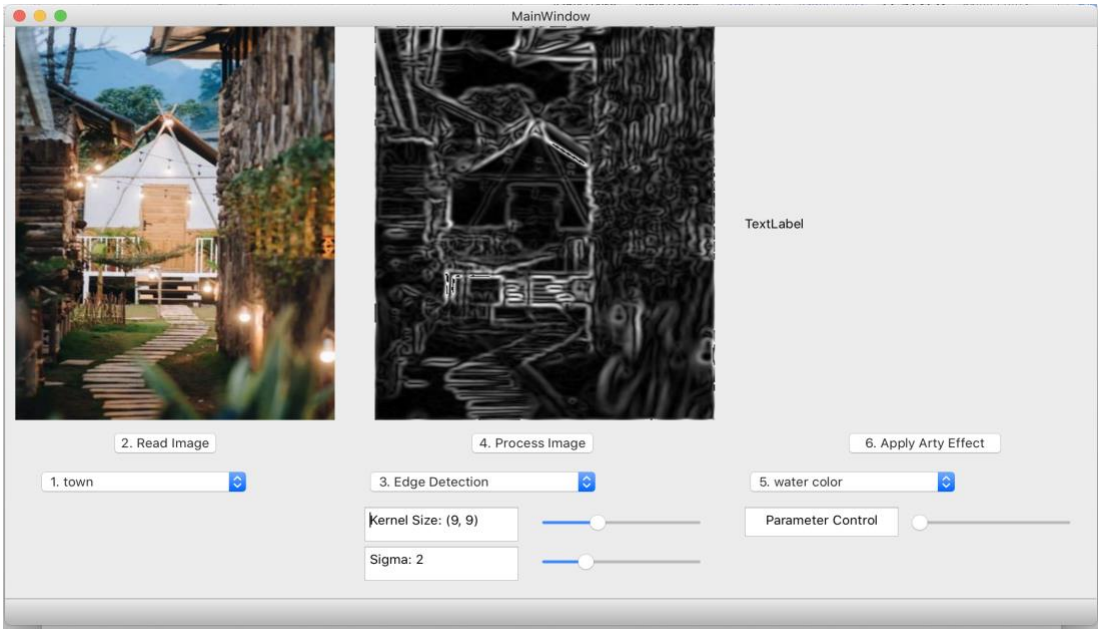


NOTE: The parameters control bars won't become active until you hit Process Image button

NOTE: Don't slide the control bar; CLICK instead

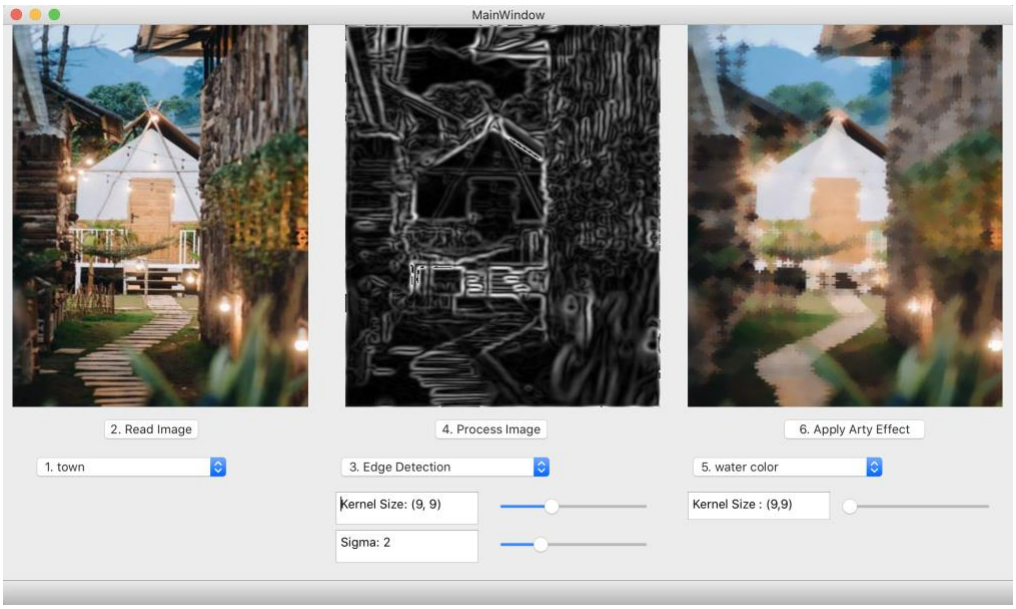
The two parameters control bar will show up.

CLICK on a different place on the control bar to see a real-time effect on the result.



Step 5: Choose an arty effect "5. Water color"

Step 6: Click Apply Arty Effect button



NOTE: The parameters control bars won't work until you hit Apply Arty Effect button

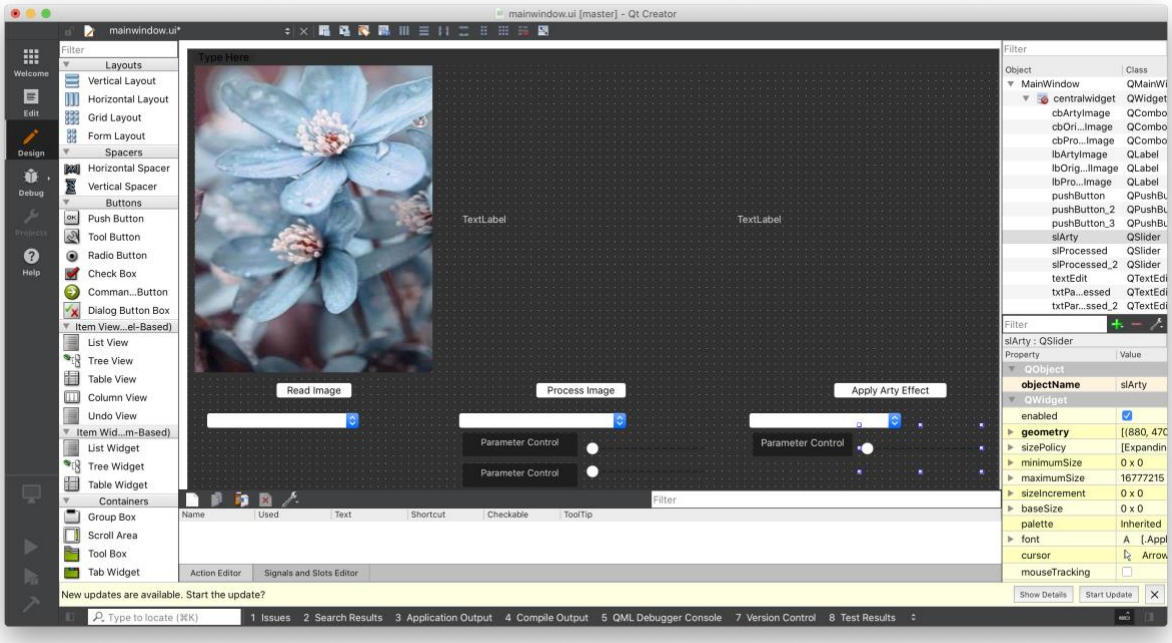
NOTE: Don't slide the control bar; CLICK instead

The parameter control bar will show up.

CLICK on different places on the control bar to see a real-time effect on the result.

III. Implementation descriptions

THIS GUI is built using PyQt, which is an open-source software for GUI design. I created my GUI in Qt Designer Form and then installed a python tool to convert Qt's GUI design to python code to manipulate and implement image processing techniques. Below is a screenshot of how the Qt designer looks like.



Six key components construct the GUI:

1. *MainWindown/Central Widget:* hold other smaller components.
2. *QComboBox*: contains various options of images, processing techniques, and arty effects for users to choose from.
3. QLabel: placeholder to display the produced images.
4. QPushButton: the buttons (labeled as Read Image, Processing Image, Apply Arty Effect) that users can click on to execute actions.
5. Qslider: the parameter control bar.
6. QTextEdit: shows the current parameter values obtained from the control bar.

Overview of each method

a) Water Coloring Effect:

This effect is one of the most challenging techniques to implement while building this GUI. I used the trick in homework 5 to implement the erosion and dilation. Erosion can be implemented faster by vectoring the kernel and perform matrix multiplication instead of performing traditional convolution on every pixel. The code that implements the method is attached below. We first create the latten matrix, which is the core of this implementation, then multiply with the chosen structuring element, then take min/max value depending on erosion or dilation.

```
#### creating latten matrix which is the core of the algorithm
for i in range(0, ksize):
    for j in np.arange(s, -1, -1):
        latten_mat.append(im[i:h-(s-i), j:w-(s-j)])
matrix = np.stack(latten_mat)

rmatrix = matrix.reshape(matrix.shape[0], matrix.shape[1]*matrix.shape[2]).T
kernel = cv2.getStructuringElement(cv2.cv2.MORPH_ELLIPSE, (ksize,ksize))
ker = np.repeat(kernel.reshape(1,-1), rmatrix.shape[0], axis = 0)
result = np.multiply(ker, rmatrix)

if op == "erosion":
    filtered_image = np.min(np.where(result==0, result.max(), result), axis=1)
else:
    filtered_image = np.max(np.where(result==0, result.min(), result), axis=1)
```

b) Pyramid Image:

This technique is done by first applying Gaussian filtering on the image to remove noise and smooth out the appearance, then remove every other 2, 4, or 6 pixels to the downscale image. The Gaussian kernel can be computed via

$$g(x,y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Although filtering will leave the image the same size, downscale the image will accelerate the computation. Pyramid image is implemented in the function gaussianBlur, which will be executed when the user hits the button “Process Image” in the GUI with an appropriate choice of Gaussian Blur method in the combo box.

```
def gaussianBlur(self, kernel_size = 3, sigma = 1.0):
```

c) Specify Histogram Equalization (SHE):

Specify Histogram Equalization is done by first generate the desired distribution and then transform the image’s distribution to that desired one. We will use Gaussian distribution, which depends on the mu and standard deviation of the pixel value to generate new distribution. This method has high overhead and typically takes longer to finish; hence, the user may experience a short wait while running this real-time performance method.

To implement this method, we will first transform the RGB color image to HSV color space and apply to specify histogram equalization on the V value only,

```
### get hsv
image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
H, S, V = cv2.split(image_hsv)
v_pdf, v_cdf = self.compute_pdf_cdf(V)
row, column = V.shape
### generate gaussian distribution
gauss = np.round(np.random.normal(mu, sigma, [V.shape[0], V.shape[1]])).astype(int)
g_pdf, g_cdf = self.compute_pdf_cdf(gauss)
```

then transfer it back to RGB space. This algorithm's core is to map the pixel value corresponding to the new distribution, which is done as below.

```
### transform image
transformed_V = V
for r in range(row):
    for c in range(column):
        pixel_val = V[r,c]
        cdf_pixel = v_cdf[pixel_val]
        ## mapping
        index = np.argmin(abs(g_cdf - cdf_pixel))
        transformed_V[r,c] = index
```

d) Gaussian Blur and Low Pass Filter:

These two methods have the same “role,” in which they help to remove noise and smooth out the image. They are computed by applying the kernel, convolving with the image, and taking the weighted average as the new value. The Gaussian kernel is the same as in the Pyramid method, and the filter for Low Pass Filter is the average kernel.

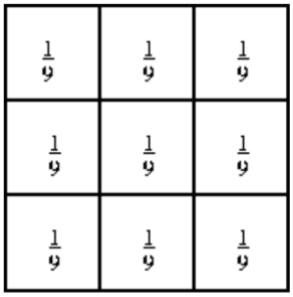


Figure 1: 3x3 averaging kernel used in low pass filtering.

The low pass filter is implemented in the function

```
def LPF(self, kernel_size = 3):
```

e) Edge Detection:

Edge is detected using gradient analysis via Sobel kernel. In addition to the simple use of the Sobel kernel, we will first smooth out the image by applying a Gaussian filter and see how the filter may affect the detected edges. The edge is computed by taking the magnitude of horizontal and vertical edges

```
h1 = np.array([[1, 0, -1], [2, 0 , -2], [1, 0, -1]])
h2 = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

m1 = signal.convolve2d(image, h1, mode = "same")
m2 = signal.convolve2d(image, h2, mode = "same")

m = np.sqrt(m1**2 + m2**2)
```

Overview of GUI's components:

Users will interact with the GUI components, including the combo box, press button, slide the parameter control bar.

Action 1: After the user chose the exciting image, the user clicks on the button "Read Image" (or QPushButton), which will execute the code below

```
self.pbOriginalImage.clicked.connect(self.pressedOriginalImage)
```

This button will link to the pressedOriginalImage, which will read the image in and display the image in the QLabel object, which is lbOriginalImage in the code attached below.

```
def pressedOriginalImage(self):
    global fname
    image_name = str(self.cbOriginalImage.currentText())
    if image_name == "1. flowers":
        fname = "blue_flowers.jpg"
    elif image_name == "1. town":
        fname = "mocchau.jpg"
    else:
        fname = "dragonfly.jpg"

    self.lbOriginalImage.setText("")
    self.lbOriginalImage.setPixmap(QtGui.QPixmap(fname))
    self.lbOriginalImage.setScaledContents(True)
```

Action 2: The user will choose a processing method, then click the "Process Image" button, which will execute the code

```
self.pbProcessedImage.clicked.connect(self.pressedProcessEffect)
```

This will link to the pressedProcessEffect function, which will check which method is chosen, process the image, and display it.

Action 3: The user then will have more options to change the parameter to experience a real-time effect. This will execute the code

```
self.slProcessed.valueChanged[int].connect(self.sliderProcess)
```

The sliderProcess will receive the slider's value and pass it to the method to modify the technique.

```
def sliderProcess(self, value):
    processMethod = str(self.cbProcessedImage.currentText())
    if processMethod == "3. Low Pass Filter":
        self.txtParamProcessed1.setPlainText(" ")
        self.slProcessed.setRange(3, 15)
        self.txtParamProcessed.setPlainText("Kernel Size: (" + str(value) + ", " + str(value) + ")")
        self.LPF(kernel_size = value)
    elif processMethod == "3. Edge Detection":
        s = self.slProcessed1.value()
        self.txtParamProcessed.setPlainText("Kernel Size: (" + str(value) + ", " + str(value) + ")")
        self.sobel(kernel_size = value, sigma = s)
    elif processMethod == "3. Gaussian Blur":
        s = self.slProcessed1.value()
        self.txtParamProcessed.setPlainText("Kernel Size: (" + str(value) + ", " + str(value) + ")")
        self.gaussianBlur(kernel_size = value, sigma = s)
    elif processMethod == "3. Specify Histogram Equalizer":
        s = self.slProcessed1.value()
        self.txtParamProcessed.setPlainText("mu: " + str(value))
        self.histogram_specification(mu=value, sigma = s)
    elif processMethod == "3. Pyramid Down":
        self.txtParamProcessed.setPlainText("Scale factor: " + str(value))
        self.pyrDown(s = value)
```

Action 4: Same as action 3, but for Arty Effect instead of Processing image.

IV. Results discussion and output images

1. Processing Effect

a) Pyramid Image (Downsize imaging):

In this method, the only parameter we consider is how much we want to downscale the image. It can be a factor of 2, 4, 8, etc., of the original image. As illustrated in figure 2, the image quality degrades as we keep downscaling the image, which we expected since we basically reduce the resolution, making the image less detailed.

Figure 2: (a) Original Image (b) Downscale by 2 (c) Downscale by 5

b) Specify Histogram Equalization (SHE):

This method will use Gaussian distribution to generate the desired distribution we want to use to transform the image. The two parameters that affect the distribution are mu *m* and sigma *s.* Figure 3 shows how changing mu and sigma leads to dramatically different results. As we increase mu or sigma, we make the distribution move towards large pixel values i.e., making the picture brighter. On the other hand, reducing mu and sigma will shift the distribution to lower pixel values; hence darker the image as illustrated.

Figure 3: (a) Original Image (b) SHE: mu = 180, s = 20 (c) SHE: mu = 150, s = 2

c) Low Pass Filter (LPF):

The main parameter used in the method is the kernel size *k*. As we increase the size, we will take more surrounding pixels into account, resulting in a blurrier picture. Figure 4 illustrates the effect.

Figure 4: (a) Original Image (b) LPF with k = (3,3). (c) LPF with k = (7,7)

d) *Gaussian Blur:*

Like LPF, Gaussian filtering filters the high-frequency component and removes noise in the image, making the image smoother. There are two main parameters in the Gaussian Blur method: sigma *s* and kernel size *k*. An increase in any of the two will cause the image to blurrier.

Figure 5: (a) Original Image (b) Gaussian blur: s = 2, k = (5,5)

e) *Edge Detection:*

Since we are using gradient analysis to detect the edge, I used the Sobel kernel of size three as the fixed kernel. However, we will focus on exploring how smoothing the image will affect edge detection. I used Gaussian Blur to smooth out the image then perform edge detection on the smooth image. As we increase the sigma *s* and size of the Gaussian kernel *k*, the image starts losing details and becomes blurry, hence reducing the number of detected edges.

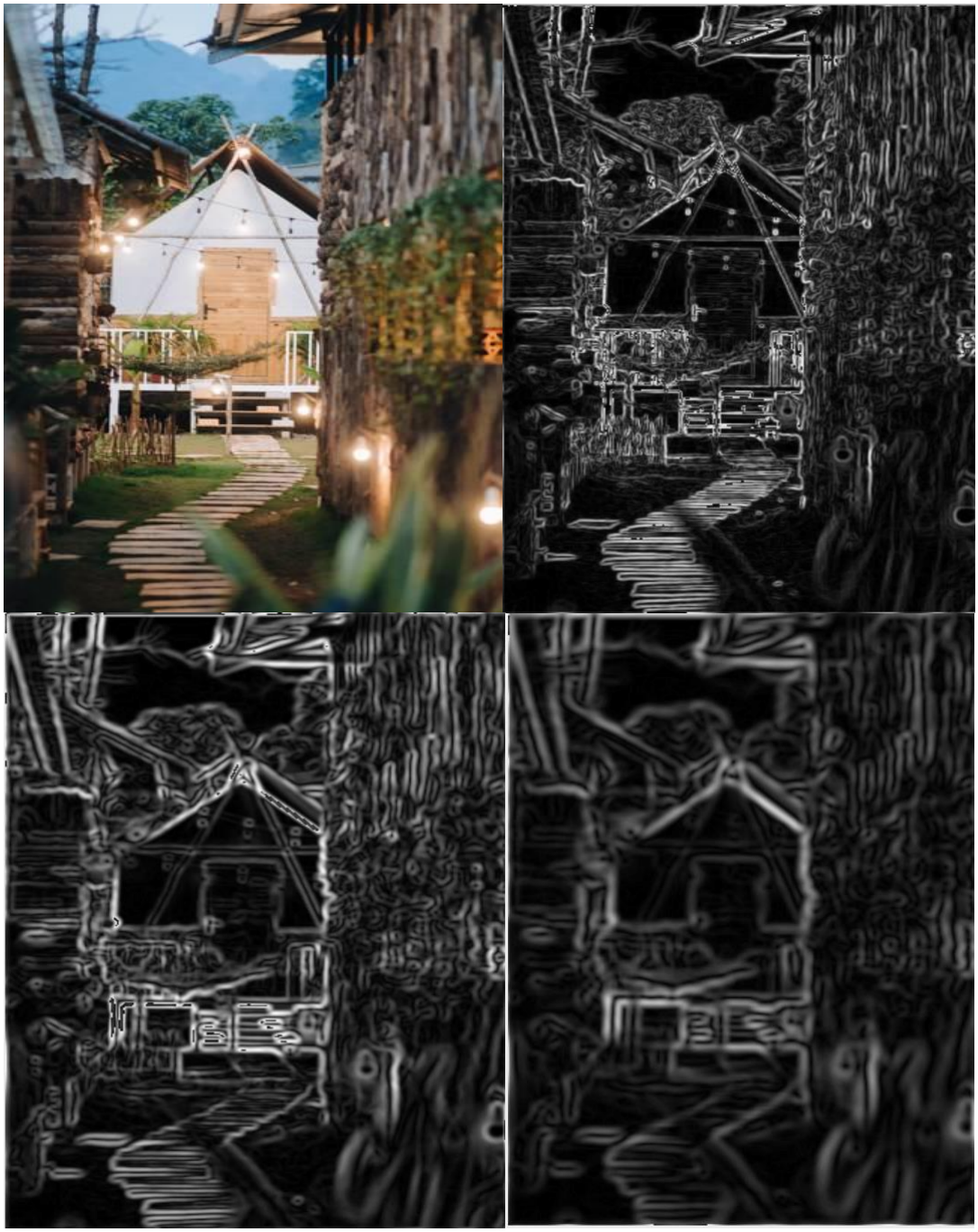
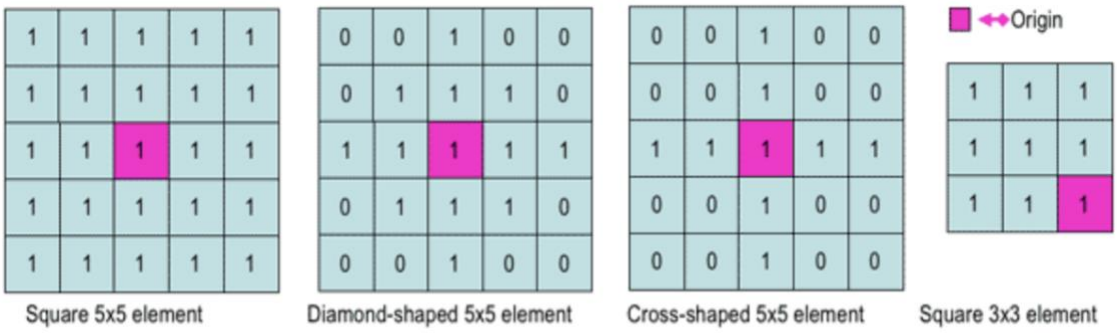


Figure 6: (a) (b) : (a) Original Image (b) Edge Detection with k = (1,1) and s = 1.0

(c) (d): Edge Detection with k = (5,5), k = (11, 11) and s = 2, s = 3; respectively

2. Arty Effect: Water Coloring

There are two main parameters in the water coloring effect: the structuring element and its size. The ***structuring element (SE)*** acts as a kernel but perform simple test/operation on pixel values. The size determines how large the SE is. Below are several commonly used SE.



Examples of simple structuring elements.

First, we will compare the results as different structuring elements are used. Users are not able to change the type of structuring element in real-time. The ellipse SE produces a nice and smooth effect compared to other structuring elements. As we can see in figure 7 and figure 8, ellipse SE's result looks smoother and waterier than cross SE. Hence, the GUI sticks with ellipse SE and emphasizes the SE's size for the real-time effect.

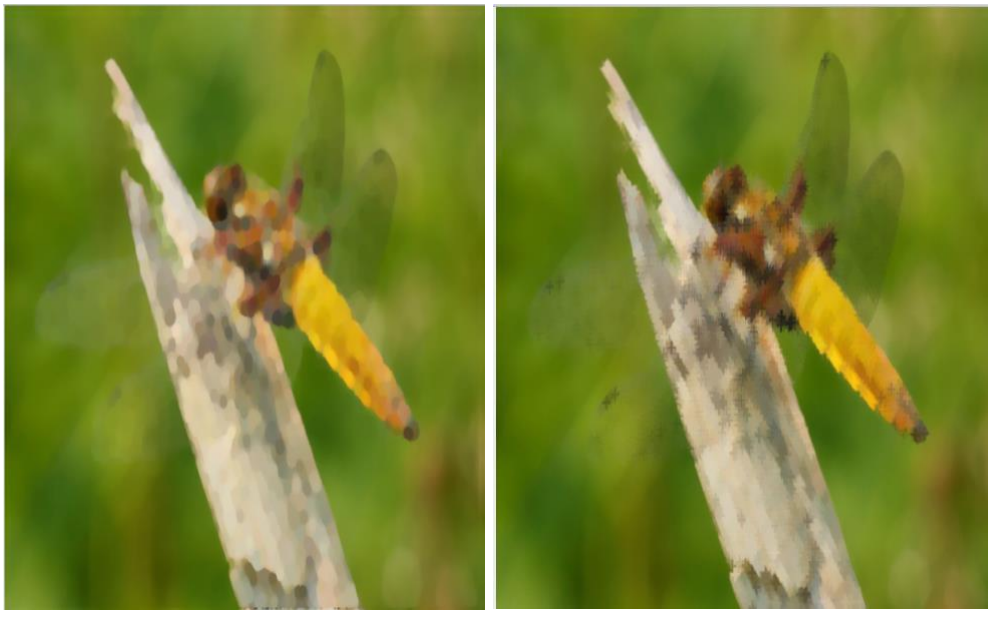


Figure 7: (a) Ellipse SE with size (7,7) (b) Cross SE with size (7,7)

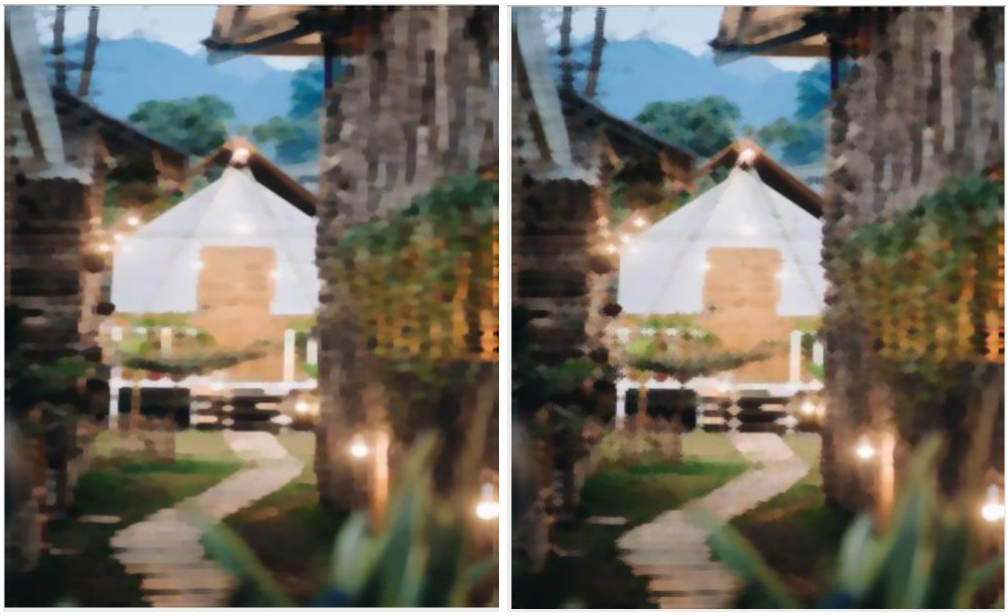
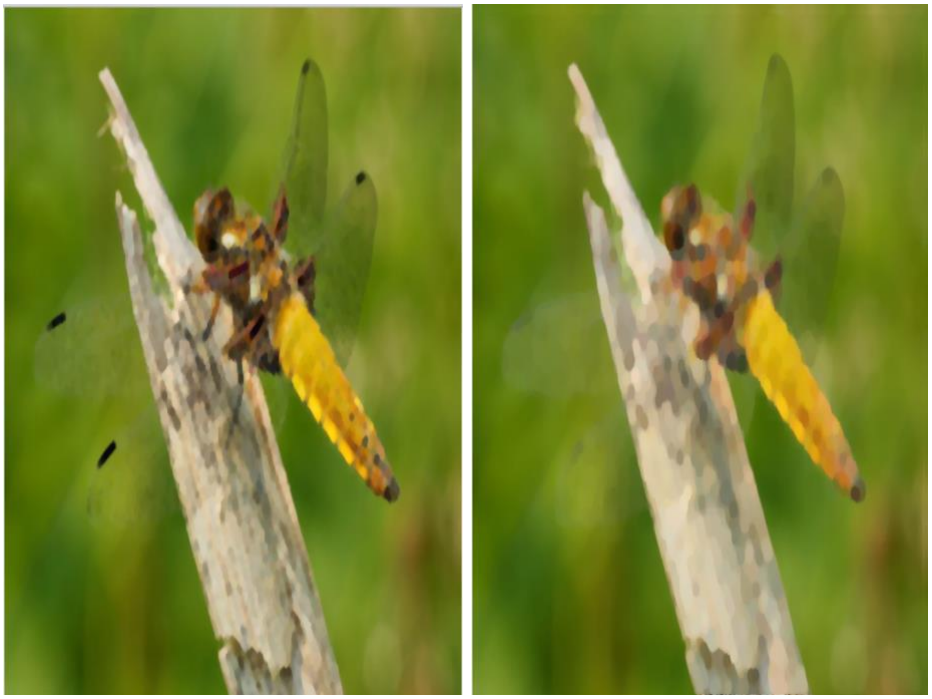


Figure 8: (a) Ellipse SE with size (5,5) (b) Cross SE with size (5,5)

Figure 9 displays how the kernel (i.e. structuring element) size plays a vital role in the outcome. As we increase the structuring element's size, the result becomes more realistic and demonstrates the expected water coloring effect. The small kernel tends to produce sharp details in the image.



V. Future work/what you learned

What I learned:

- How to implement image processing techniques to run in real-time (the trick to do convolution without looping through every pixel) for arbitrary kernel size and image size
- Get a better understand of how each technique works through the implementation
- Get Hand-on experiences in image processing
- How to implement a GUI using PyQt, which is most commonly used in the industry
- Some algorithms are hard to scale (like Specify histogram equalizer)

Future work:

- Implement other arty effects such as cartoon effect, segmentation, etc.
- Re-produce some image processing techniques based on the foundation learned in the class
- Make Specify Histogram Equalization and other techniques scale faster.

-

VI. References/resources

- http://jamesgregson.ca/bilateral-filtering-in-python.html
- https://stackoverflow.com/questions/29731726/how-to-calculate-a-gaussian-kernel-matrix-efficiently-in-numpy
- https://realpython.com/python-pyqt-gui-calculator/
- https://www.askaswiss.com/2016/01/how-to-create-cartoon-effect-opencv-python.html
- https://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm
- https://customers.pyimagesearch.com/lessons/morphological-operations/
- http://www.cyto.purdue.edu/cdroms/micro2/content/education/wirth07.pdf