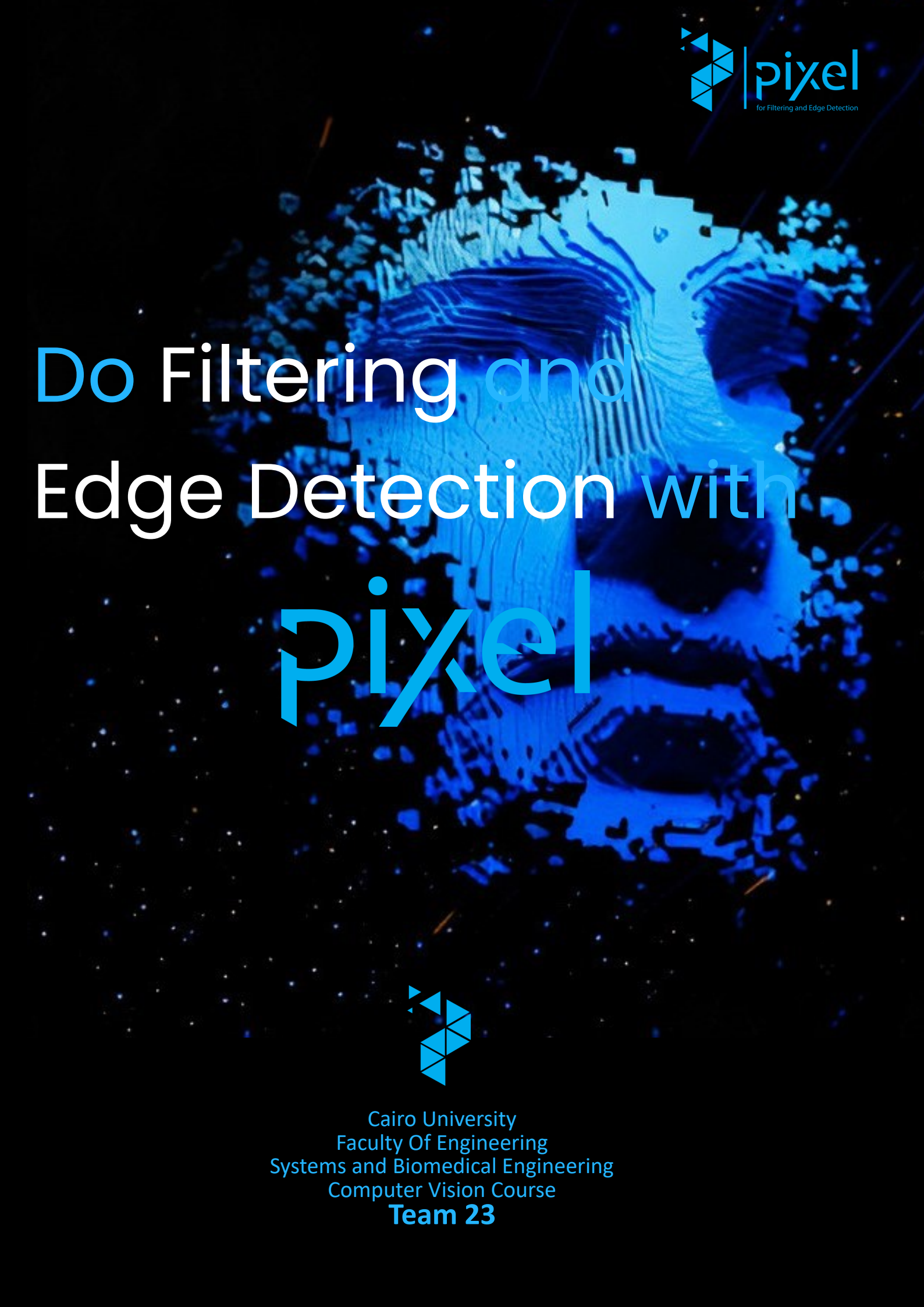
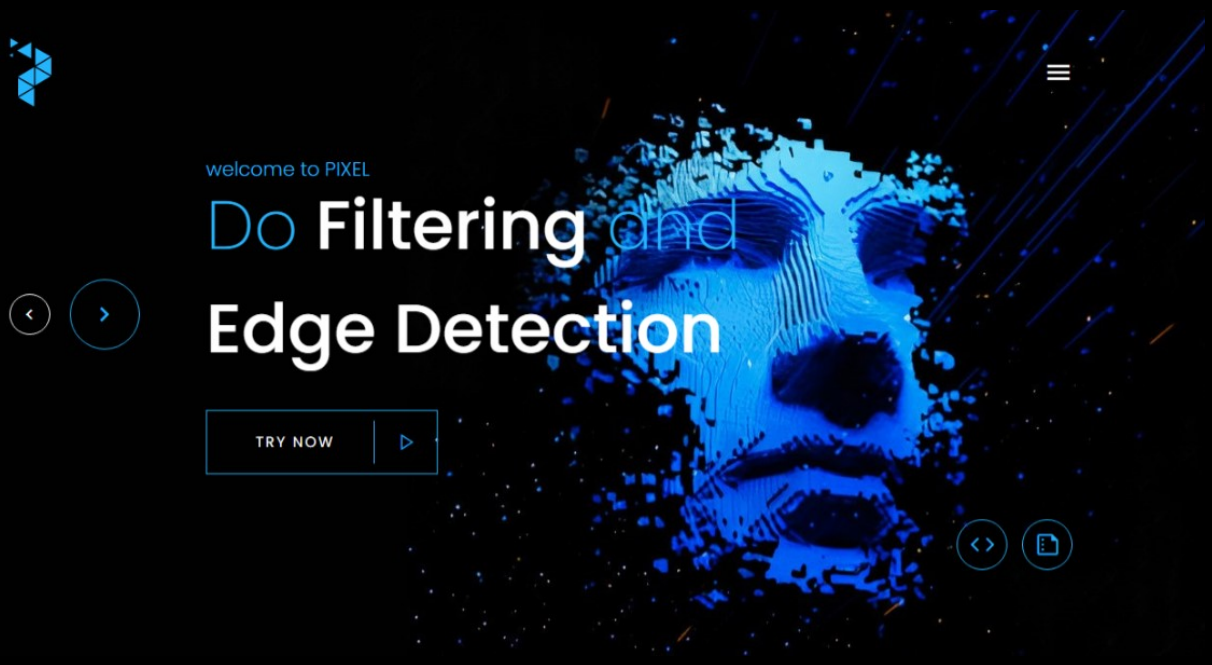


Cairo University
Faculty Of Engineering
Systems and Biomedical Engineering
Computer Vision Course
Team 23

It is a website that makes some processing on images. There are many features on this page. You can upload or download an image. You can apply three types of noises to the image. Then, choose the appropriate filter that can be applied to it to release this noise. You can apply edge detection to the image. You can see all the curves and histograms that describe the image. Finally, there is a special feature to apply hybrid image.



TEAM 23

Name	Section	Bench Number
Mahmoud Zakaria Seyam	2	26
Aisha Amr Hassan	1	51
Yasmine Yasser Ali	2	52
Ahmed Abd El Moneim Radwan	1	5
Sara Tarek Galal	1	40

Algorithms of Features

Useful functions used in features

conv function

```python
def conv(img, krnl):
    krnl_h, krnl_w = len(krnl), len(krnl[0])
    img_h, img_w = img.shape
    img_conv = np.zeros(img.shape)
    for i in range(krnl_h, img_h-krnl_h):
        for j in range(krnl_w, img_w-krnl_w):
            sum = 0
            for m in range(krnl_h):
                for n in range(krnl_w):
                    sum += krnl[m][n]*img[i-krnl_h+m][j-krnl_w+n]
            img_conv[i][j] = sum
    return img_conv
```

Description

We first specify the kernel input **krnl_h** with **len(krnl)** refers to kernel height and **krnl_w** with **len(krnl[0])** refers to kernel width to be used in the iteration, then, we specify the image input dimensions with **img.shape** as **img_h** refers to image height and **img_w** refers to image width to be used in iteration as well.

Then, we make **img_conv** with same dimension of the original image but all elements are zeros.

Then, we pass over each pixel in the original image and multiply it by the kernel elements and sum them.

This iteration lasts until we reach the elements at **img_h-krnl_h+1** and **img_w-krnl_w-1**, as after these positions the all elements still zero, that's why the generated image will be padded at all sides with number of pixels **(krnl_h-1)/2** and **(krnl_w-1)/2**.

This output **img_conv** is filtered in grayscale.

generate_av_kernel function

```python
def generate_av_kernel(krnl_size):
    kernel = []
    for i in range(krnl_size):
        row = []
        for j in range(krnl_size):
            row.append(1/krnl_size**2)
        kernel.append(row)
    return kernel
```

Description

This function is generated an 2d dimensional array with equal sides, as we first make **kernel** the array which saves the kernel values.

We iterate vertically with **krnl_size** iterations, in each iteration we make **row** which save the row value even the vertical iteration finishes, and clear it with each start of new vertical iteration.

Then we iterate horizontally with **krnl_size** iterations and save value **1/krnl_size^2** in row array, then, after the horizontal iteration finishes, we save **row** list in kernel.

This outputs an array with values 1/krnl_size^2.

search_index function

```python
def search_index(index, img):
    count = 0
    for i in range(len(img)):
        for j in range(len(img[0])):
            if img[i][j] == index:
                count += 1
    return count
```

Description

This function counts the value input in img, as it takes the index which is the value required to be counted in the img which is the second parameter input.
Then, we iterate over each pixel and check if the index is equal to the value of the pixel, if true, the count is incremented, after iteration finish, count is returned.

unique_img function

```python
def unique_img(img):
    unique_arr = np.zeros(256)
    for i in range(len(unique_arr)):
        unique_arr[i] = search_index(i, img)
    return unique_arr
```

Description

We first make **unique_arr** which is 1d array with dimension of 256, as the range of values in each pixel is from 0 to 255, then, we use **seach_index** to get the count of the value which is the index, and save the count at the index which refers the value as the count returned is the count of this value, then, after the iteration finishes, it returns one hashed array.

get_unique function

```python
def get_unique(arr):
    unique, count = np.unique(arr, return_counts=True)
    return [unique.tolist(), count.tolist()]

```

Description

This function has the same concept of **unique_img**, but we use **np.unique** which returns an array of unique values and an array of counts which is corresponding its values in **unique** array, then return an array contains both **unique** array and **count** but we use **tolist** with both of those array as it cannot be used in json object.

Histograms and Distribution Curves

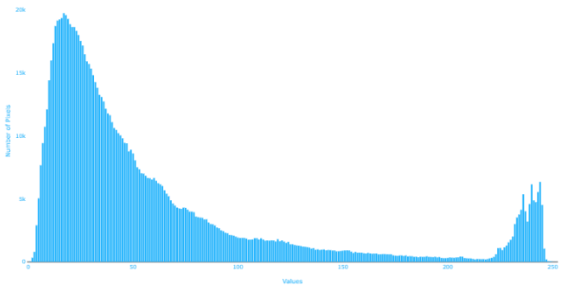
Histograms

We use **unique_img** (mentioned before) to take unique values with its counted pixels, then plot it using **plotly js**

Results

Input

Output

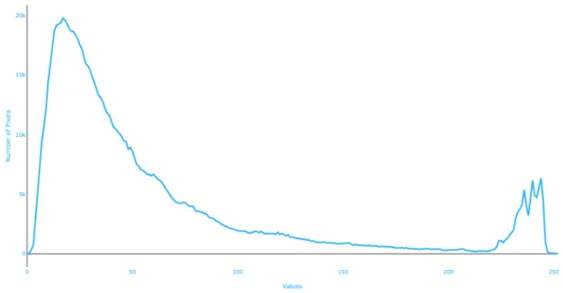


Distribution Curves

We use **unique_img** (mentioned before) to take unique values with its counted pixels, then plot it using **plotly js**

Results

Input **Output**



Noises

add_uniform_noise function

```python
def add_uniform_noise(img_path, type):
    '''This function takes two variable img_path => path of image and type => type of the image if it is gray or rgb
    and return the path of the generated image'''
    img = cv2.imread(img_path)
    img = img/255
    if type == 'rgb':
        x, y, z = img.shape
    elif type == 'gray':
        x, y, _ = img.shape
    a = 0
    b = 1.1
    noise = np.zeros(img.shape, dtype=np.uint8)
    for i in range(x):
        for j in range(y):
            noise[i][j] = np.random.uniform(a, b)
    noise_img = img + noise
    noise_img = noise_img*255
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_uniform_noise_img.png'
    cv2.imwrite(img_path, noise_img)
    return img_path
```

Description

The uniform noise follows a uniform distribution.
First, we read the image and change it to gray scale. Second, we normalize the image with maximum pixel value (255). Third, we create the uniform noise by setting the minimum number to variable an equal to zero and maximum b equal to 1.1. Fourth, we create matrix of zeros and then change it with uniform distribution. Finally, we add the noise to our image.

Results

Applying uniform noise

Input

Output

Add_salt_pepper_noise function

```python
def add_salt_pepper_noise(image, pepper):
    '''This function takes two variable img_path => path of image and pepper => the distribution of balck pixels in filter
    and return the path of the generated image'''
    image = cv2.imread(image)
    output = np.zeros(image.shape, dtype=np.uint8)
    salt = 1 - pepper
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            rdn = random()
            if rdn < pepper:
                output[i][j] = 0
            elif rdn > salt:
                output[i][j] = 255
            else:
                output[i][j] = image[i][j]
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_salt_pepper_noise_img.png'
    cv2.imwrite(img_path, output)
    return img_path
```

Description

The function takes to parameters the image and the amount or probability of pepper.
First, we read the image and ten change it to gray scale. Second, we create matrix of zero that will change it latter to salt and pepper noise or the pixel value of the image. third, inside the for loop we go thought every row and column and generate a random noise number goes from 0 to 1. finally, we compare the random number with the salt and pepper value If less than pepper value fills it with zero if it less then salt value fill it with 255 otherwise fill it with the original image pixel value.

Results

Applying salt and pepper noise

Input

Output

add_gaussian_noise function

```python
def add_gaussian_noise(img_path, var):
    '''This function takes two variable img_path => path of image and var => variance
    and return the path of the generated image'''
    img = cv2.imread(img_path)  # convert image into grayscale
    img = img/255   # normalize the image
    x, y, z = img.shape
    mean = 0
    sigma = np.sqrt(var)
    noise = np.random.normal(loc=mean, scale=sigma, size=(x, y, z))
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_gaussian_noise_var_{var}_img.png'
    cv2.imwrite(img_path, (noise+img)*255)
    return img_path
```

Description

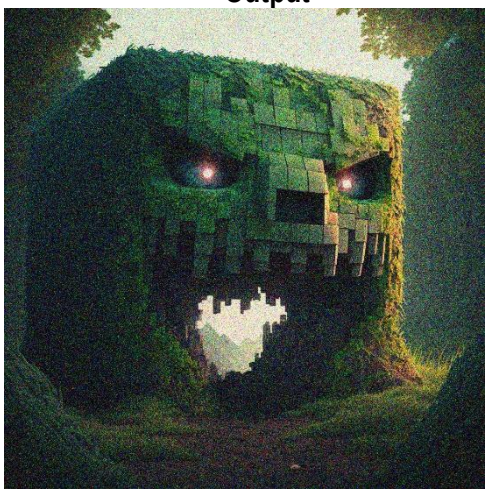
The function takes two parameters the image and the variance value.
First, we read the image and change it to gray scale. Second, we normalize the image to make max value equal to 1 and minimum equal to zero. Third, we take the number of rows and columns. Forth, we set the mean value to zero and calculate the standard deviation or sigma. Finally, we create our gaussian noise using NumPy normal and add the noise to the original image.

Results

Applying gaussian noise

Input

Output



Thresholding

Global_thresholding function

```python
def global_threshold(img):
    img = cv2.imread(img, 0)

    h = img.shape[0]
    w = img.shape[1]

    img_thres = np.zeros((h, w))
    n_pix = 0
    # loop over the image, pixel by pixel
    for y in range(0, h):
        for x in range(0, w):
            # threshold the pixel
            pixel = img[y, x]
            if pixel < 128:  # because pixel value will be between 0-255.
                n_pix = 0
            else:
                n_pix = pixel
            img_thres[y, x] = n_pix
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_gloabal_img.png'
    cv2.imwrite(img_path, img_thres)
    return img_path
```

Description

First, we read the image. second, we take the height and width of the image and make a zero matrix of the same size. Third, we make a variable that will take the new value of the pixel. forth, inside the for loop we go thought the width and the height and compare the value of the pixel of image to half of the pixels range that goes from 0 to 255 the half is 127, if the pixel is smaller set, it to zero else it takes the original value.

Results

Applying global thresholding

Input

Output

Local_thresholding function

```python
def local_threshold(img):
    img = cv2.imread(img, 0)
    windowsize_r = 3
    windowsize_c = 3
    sub_img = []
    for r in range(0, img.shape[0] - windowsize_r, windowsize_r):
        for c in range(0, img.shape[1] - windowsize_c, windowsize_c):
            window = img[r:r+windowsize_r, c:c+windowsize_c]
            sub_img.append(window)
    average_list = []
    for iter1 in sub_img:
        height = iter1.shape[0]
        width = iter1.shape[1]
        sum = 0
        average = 0
        for i in range(0, width):
            for j in range(0, height):
                sum += iter1[i][j]
        average = sum/(width * height)
        average_list.append(average)

    new_image = []
    it = 0
    for iter in sub_img:
        height = iter.shape[0]
        width = iter.shape[1]
        for i in range(0, height):
            for j in range(0, width):
                if iter[i][j] > (average_list[it]-6):
                    iter[i][j] = 255
                else:
                    iter[i][j] = 0

        it = it+1
        new_image.append(iter)

    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_local_img.png'
    cv2.imwrite(img_path, img)
    return img_path

```

Description

First, we read the image and then change it to gray scale. Second, we make two variables for the row and columns that we will make window and their size. third, the for loop split the image into too many small windows depend on the size of the window (rows and columns). forth, we take average list that we will append the average of every pixel's window and then make a for loop to calculate every window average. Finally, and the last for loop we compare every window pixel to the average of its pixels set it to zero if it is smaller and one if it is bigger and increment the iterator of the average list.

Results

Applying local thresholding

Input

Output

High and Low Pass Filters

Fourier_domain_rgb function

```python
ef fourier_domain_rgb(image, sigma):
    transformed_channels = []
    for i in range(3):
        input_ = np.fft.fft2((image[:, :, i]))
        result = ndimage.fourier_gaussian(input_, sigma)
        transformed_channels.append(np.fft.ifft2(result))

    final_image = np.dstack([transformed_channels[0].astype(int),
                             transformed_channels[1].astype(int),
                             transformed_channels[2].astype(int)])
    return final_image.real
```

HPF_rgb function

```python
def HPF_rgb(path):
    img = cv2.imread(path)[:, :, ::-1]
    img = cv2.resize(img, (1000, 1000))
    img = img-fourier_domain_rgb(img, 9)
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_HPF_img.png'
    cv2.imwrite(img_path, img)
    return img_path
```

LPF_rgb function

```python
def LPF_rgb(path):
    img = cv2.imread(path)[:, :, ::-1]
    img = cv2.resize(img, (1000, 1000))
    img = fourier_domain_rgb(img, 9)
    img = img[:, :, ::-1]
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_LPF_img.png'
    cv2.imwrite(img_path, img)
    return img_path
```

Description for the previous three functions

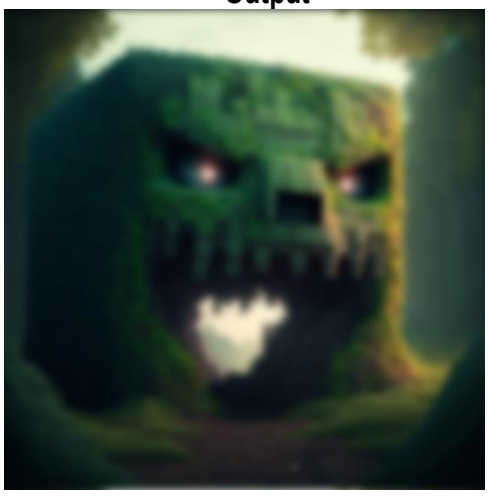
For every function it takes the image.
First, we read the image and try to fix its colors and resize the image. second, we call the Fourier transform function. the Fourier function make a list to the transform channels for red, green and blue channels then make Fourier transform for every channel and Fourier gaussian blurring and return the final transformed image. finally in low pass filleter we take the output as it but in the high pass filter, we subtract the image from the return from the function.

Results

Applying low pass filter

Input

Output



The low pass filter is making blurring to image

Results

Applying high pass filter

Input

Output

The high pass filter is making edge detection.

Hybrid Image

Hybrid_rgb function

```python
def hybrid_rgb(img1_path, img2_path):
    # prepare img1
    img1 = cv2.imread(img1_path)
    img1 = img1[:, :, ::-1]
    img1 = cv2.resize(img1, (1000, 1000))
    # prepare img2
    img2 = cv2.imread(img2_path)
    img2 = img2[:, :, ::-1]
    img2 = cv2.resize(img2, (1000, 1000))

    img1 = fourier_domain_rgb(img1, 9)  # low pass filter img1
    img2 = img2 - fourier_domain_rgb(img2,  1000)  # high pass filter img1

    hybrid_image = img2+img1
    hybrid_image = hybrid_image[:, :, ::-1]

    img_path = f'./static/download/hybrid/{randint(0,9999999999999999)}_hybrid_image_img.png'

    cv2.imwrite(img_path, hybrid_image)

    return img_path

```

Description

First, we read the two images then try to fix the color of the them. Second, we resize both images to be the same size. Third, we call the same function used in low and high pass that make high pass filter for one image and low pass filter for the other filter. Finally, we add both images.

Results

Applying high pass filter

Input image 1 (LPF)

Input image 2 (HPF)

Output

Filters

Median Filter

```python
def apply_median_filter(img_path, window_size):
    img = cv2.imread(img_path, 0)
    filtered_img = np.zeros_like(img)
    padding_size = window_size // 2
    padded_img = np.pad(img, padding_size, mode='symmetric')
    for i in range(padding_size, len(img) + padding_size):
        for j in range(padding_size, len(img[0]) + padding_size):
            window = padded_img[i-padding_size:i+padding_size +
                                1, j-padding_size:j+padding_size+1].flatten()
            median = np.median(window)
            filtered_img[i-padding_size, j-padding_size] = median
    img_path = f'./static/download/edit/{randint(0,999999999999999)}_median_filtered_krnl_{window_size}_img.png'
    cv2.imwrite(img_path, filtered_img)
    return img_path
```

Description

We first read the image with **cv2.imread** in grayscale, then, we pad the image with the **np.pad** function, using symmetric padding (which copies the border pixels). Then, we iterate over each pixel of the original image, and for each pixel we extract a window of size **window_size** around it. We flatten the window to a 1-dimensional array, and find the median of the values in the window using the **np.median** function. Finally, we assign the median to the corresponding pixel in the **filtered_img** array.

Note that we use **np.zeros_like** to create the **filtered_img** array, which has the same shape and data type as the input image. This ensures that the filtered image has the same properties as the original image.

To use **apply_median_filter**, we pass two parameters input img_path that refers to the path of the image and the **window_size** that refer to the size of kernel.

This will output the path of **image_filtered** grayscale image saved in **img_path**.

Results

Applying median filter with kernel size of 9*9

Input **Output**

Average Filter

```python
def add_average_filter(img_path, krnl_size):
    img = cv2.imread(img_path, 0)  # read image in grayscale
    kernel = generate_av_kernel(krnl_size)
    new_img_path = f'./static/download/edit/{randint(0,9999999999999999)}_average_filter_krnl_{krnl_size}_img.png'
    cv2.imwrite(new_img_path, conv(img, kernel))
    return new_img_path
```

Description

We first read the image with **cv2.imread** in grayscale, then, we generate the kernel window of size **krnl_size** with **generate_av_kernel (mention before)**, then we apply convolution with **conv (mention before)** function to return filtered image in grayscale, then we save it with **cv.imread** in **new_img_path.**

To use **add_average_filter**, we pass two parameters input **img_path** refers to the path of the image input and **krnl_size** refers to the window size.

This will output the path of filtered image as a grayscale image saved in **new_img_path**.

Results

Applying average filter with kernel size of 9*9

Input

Output

Gaussian Filter

```python
def add_gaussian_filter(img_path):
    img = cv2.imread(img_path, 0)  # read image in grayscale
    kernel = [[1/16, 2/16, 1/16],
              [2/16, 4/16, 2/16],
              [1/16, 2/16, 1/16]]
    new_img_path = f'./static/download/edit/{randint(0,9999999999999999)}_gaussian_filter_img.png'
    cv2.imwrite(new_img_path, conv(img, kernel))
    return new_img_path
```

Description

We first read the image with **cv2.imread** in grayscale, then we apply convolution with **conv (mention before)** function to return filtered image in grayscale, then we save it with **cv.imread** in **new_img_path.**

To **add_gaussian_filter**, we pass only one parameter input **img_path** refers to the path of the image input as the kernel is constant **krnl_size** refers to the window size.

This will output the path of filtered image as a grayscale image saved in **new_img_path**

Results

Applying gaussian filter with kernel size of 3*3

Input

Output

Equalization

equalization function

```python
def equalization(img, bin):
    image = cv2.imread(img)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    flat = image.flatten()
    # array with size of bins, set to zeros
    hist = np.zeros(bin)
    # loop through pixels and sum up counts of pixels
    for pixel in flat:
        hist[pixel] += 1
    # create our cumulative sum
    hist = iter(hist)
    b = [next(hist)]
    for i in hist:
        b.append(b[-1] + i)
    cs = np.array(b)
    img_new = cs[flat]
    img_new = np.reshape(img_new, image.shape)
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_equalized_img.png'
    imageio.imwrite(img_path, img_new)
    return img_path

```

Description

For image equalization we transform the image to grey scale level. Then, apply pixel counting for the given image array using for loop and save counting values in an array called hist. Then, we apply cumulative sum algorithm by getting pdf & cdf with the other for loop and save

cumulative values in an array called cs. Then, we upgrade histogram values to a new level. So, we return our array to a 2d image and save this image in a certain path to show it to the user.

Results

Applying equalization on the image

Input

Output

Analytics

get_cumulative_curve function

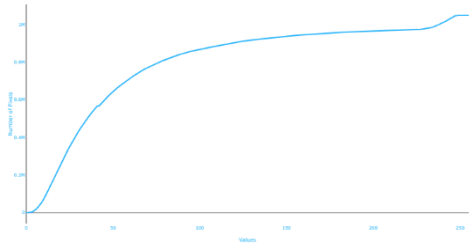
```python
def get_cumulative_curve(img_path):
    list_x = []
    for i in range(256):
        list_x.append(i)
    image = cv2.imread(img_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    flat = image.flatten()
    # array with size of bins, set to zeros
    hist = np.zeros(256)
    # loop through pixels and sum up counts of pixels
    for pixel in flat:
        hist[pixel] += 1
    # create our cumulative sum
    hist = iter(hist)
    b = [next(hist)]
    for i in hist:
        b.append(b[-1] + i)
    cs = np.array(b)

    return [list_x, cs.tolist()]
```

Description

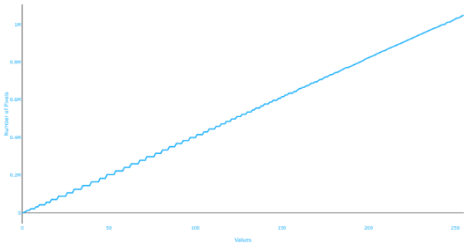
This feature is applied in the equalization and normalization of images. As you can see the curve is applied to detect how the distribution changes. The algorithm of drawing cumulative curve is the same as the algorithm of equalization of the image as it is one of the steps to get equalized image.

Results

Normalization cumulative curve



Equalization cumulative curve



Normalization

normalization function

```python
def normalization(img):
    img = cv2.imread(img)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ar = np.array(img).astype(np.float32)
    for i in range(1000):
        mn = np.min(ar)
        mx = np.max(ar)
        norm = (ar - mn) * (1.0 / (mx-mn))
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_normalized_img.png'
    # cv2.imwrite(img_path,norm )
    imageio.imwrite(img_path, norm)
    return img_path
```

Description

Image normalization is a simple algorithm. It doesn't make so many changes to the image. At first, we transfer the image into a grey scale level. Then, we apply for loop to the image array in a certain range to get its maximum and minimum values. Then, the normalized image array is given by dividing the image array with elimination of minimum values by subtraction of maximum with minimum values. So, the image array values are normalized into another level in an array called norm. Finally, we save the image in a certain path to show the output to the user.

Results

Applying normalization on the image

Input

Output

Edge Detection

Prewitt Edge Detection Filter

```python
def add_prewitt_filter(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img_x = conv(img, [[-1, 0, 1],
                       [-1, 0, 1],
                       [-1, 0, 1]])
    img_y = conv(img, [[-1, -1, -1],
                       [0, 0, 0],
                       [1, 1, 1]])
    prewitt_img_path = f'./static/download/edit/{randint(0,3333333333655)}_prewitt_img.png'
    cv2.imwrite(prewitt_img_path, img_x+img_y)
    return prewitt_img_path

```

Description

We first read the image as save it in **img** in grayscale with **cv2.imread**, then we make apply convolution of **img** with a constant Prewitt's x-direction kernel and save it in **img_x** with **conv (mention before)** and do the same but with Prewitt's y-direction kernel and save it in **img_y**. Then, we sum **img_x** with **img_y** to get total edges in the image, then, we save it with **cv2.imwrite** in **prewitt_img_path**.

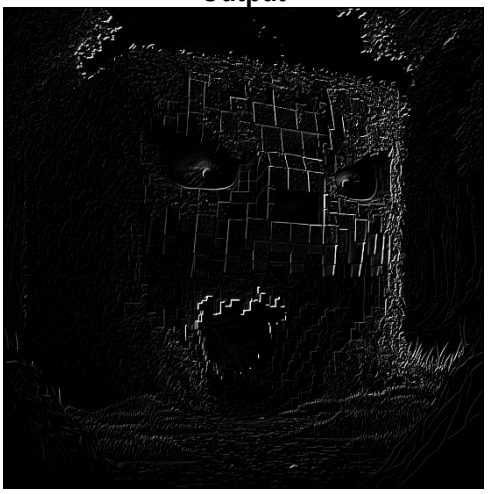
This outputs **prewitt_img_path** refer to the path which the new image has been saved.

Results

Applying Prewitt filter with kernel size of 3*3

Input

Output



Roberts Filter

```python
def add_roberts_filter(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img_x = conv(img, [[1, 0],
                       [0, -1]])
    img_y = conv(img, [[0, 1],
                       [-1, 0]])
    roberts_img_path = f'./static/download/edit/{randint(0,3333333333655)}_roberts_img.png'
    cv2.imwrite(roberts_img_path, img_x+img_y)
    return roberts_img_path
```

Description

We first read the image as save it in **img** in grayscale with **cv2.imread**, then we make apply convolution of **img** with a constant Robert's x-direction kernel and save it in **img_x** with **conv (mention before)** and do the same but with Roberts y-direction kernel and save it in **img_y**. Then, we sum **img_x** with **img_y** to get total edges in the image, then, we save it with **cv2.imwrite** in **roberts_img_path**.

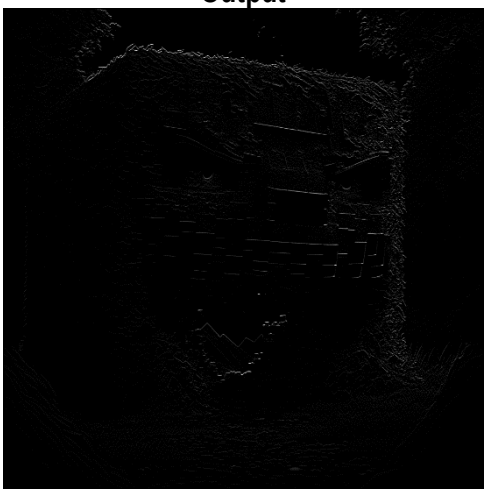
This outputs **roberts_img_path** refer to the path which the new image has been saved.

Results

Applying Prewitt filter with kernel size of 2*2

Input

Output



Sobel Filter

```python
def add_sobel_filter(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img_x = conv(img, [[-1, 0, 1],
                       [-2, 0, 2],
                       [-1, 0, 1]])
    img_y = conv(img, [[-1, -2, -1],
                       [0, 0, 0],
                       [1, 2, 1]])
    sobel_img_path = f'./static/download/edit/{randint(0,3333333333655)}_sobel_img.png'
    cv2.imwrite(sobel_img_path, img_x+img_y)
    return sobel_img_path


```

Description

We first read the image as save it in **img** in grayscale with **cv2.imread**, then we make apply convolution of **img** with a constant Robert's x-direction kernel and save it in **img_x** with **conv (mention before)** and do the same but with Roberts y-direction kernel and save it in **img_y**. Then, we sum **img_x** with **img_y** to get total edges in the image, then, we save it with **cv2.imwrite** in **sobel_img_path**.

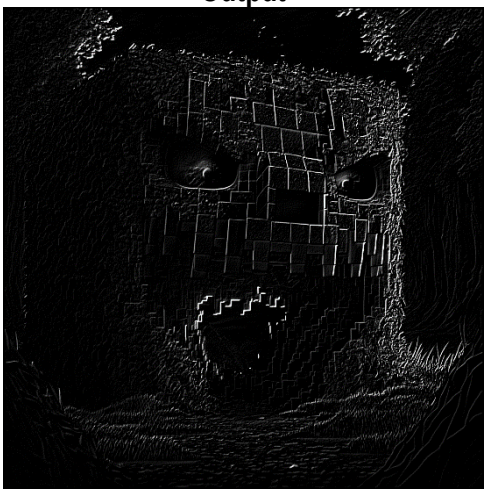
This outputs **sobel_img_path** refer to the path which the new image has been saved.

Results

Applying Prewitt filter with kernel size of 3*3

Input

Output



Canny Edge Detection Filter

```python
def add_canny_filter(img_path, high_threshold, low_threshold):
    img = cv2.imread(img_path, 0)
    gauss = cv2.GaussianBlur(img, (3, 3), 0)

    Ix = cv2.Sobel(gauss, cv2.CV_64F, 1, 0)
    Iy = cv2.Sobel(gauss, cv2.CV_64F, 0, 1)
    sobel = np.sqrt(np.square(Ix) + np.square(Iy))
    theta = np.arctan2(Iy, Ix)

    M, N = sobel.shape
    Z = np.zeros((M, N), dtype=np.int32)
    angle = theta * 180. / np.pi
    angle[angle < 0] += 180

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

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

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

    highThreshold = high_threshold
    lowThreshold = low_threshold

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

    weak = np.int32(25)
    strong = np.int32(255)

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

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

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak

    strong = 255
    M, N = res.shape
    for i in range(1, M-1):
        for j in range(1, N-1):
            if (res[i, j] == weak):
                try:
                    if ((res[i+1, j-1] == strong) or (res[i+1, j] == strong) or (res[i+1, j+1] == strong)
                        or (res[i, j-1] == strong) or (res[i, j+1] == strong)
                            or (res[i-1, j-1] == strong) or (res[i-1, j] == strong) or (res[i-1, j+1] == strong)):
                        res[i, j] = strong
                    else:
                        res[i, j] = 0
                except IndexError as e:
                    pass

    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_canny_img.png'
    cv2.imwrite(img_path, res)

    return img_path

```

Description

First, we read the image and change it to gray scale. Second, we make noise reduction to the image by applying gaussian blurring. Third, we detect the edges intensity by using Sobel edge detection and get both the edge intensity and the edge direction. Fourth, the final image should have thin edges so we create a matrix of zeros of the same size of the original gradient intensity matrix, Identify the edge direction based on the angle value from the angle matrix, check if the pixel in the same direction has a higher intensity than the pixel that is currently processed, and get the processed image with non-max suppression. Fifth, we take the low and

high threshold from the user. Finally, we make edge tracking by hysteresis by transforming weak edges to strong ones.

Results

Input

Output