pixel
for Filtering and Edge Detection

DO IMAGE PROCESSING WITH PIXEL

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

Image Thresholding

Optimal Thresholding

optimal_thresholding function

```python
def optimal_thresholding(image):
    (height, width) = image.shape
    # initiate both background pixels and foreground elements
    bckGrnd = [image[0][0], image[0][width-1],
               image[height-1][0], image[height-1][width-1]]
    forGrnd = []
    for i in range(height):
        for j in range(width):
            if not ((i == 0 and j == 0) or (i == 0 and j == width-1) or (i == height-1 and j == 0) or (i == height-1 and j == width-1)):
                forGrnd.append(image[i][j])
    # initiate b background and foreground means
    av_bckGrnd = np.mean(bckGrnd)
    av_forGrnd = np.mean(forGrnd)
    thr = (av_bckGrnd+av_forGrnd)/2
    thr_prev = 0
    print(thr)
    while (not (thr_prev == thr)):
        bckGrnd = []
        forGrnd = []
        thr_prev = thr
        for i in range(height):
            for j in range(width):
                if (image[i][j] < thr):
                    bckGrnd.append(image[i][j])
                else:
                    forGrnd.append(image[i][j])
        av_bckGrnd = np.mean(bckGrnd)
        av_forGrnd = np.mean(forGrnd)
        thr = (av_bckGrnd+av_forGrnd)/2
    print(thr)
    return cv2.threshold(image, thr, 255, cv2.THRESH_BINARY)[1]
```

This function takes 2d array as it represented the image in grayscale, then we specify the corner pixels as the background then the rest pixels are specified as the foreground as initial values of bckGrnd and forGrnd, and calculate the initial values of their mean, after that we calculate the initial value of the threshold thr as it is the average of the mean of bckGrnd and forGrnd.
After that, we do the same procedures unit the thr_prev equals thr as thr is the threshold that will be used on the image.
Then we apply a binary threshold with threshold that we calculated on the image.
Input: image array in grayscale
Output: filtered image using cv2.threshold(image,thr,255,cv2.THRESH_BINARY)[1]
Results

Input

Output

Otsu's Thresholding

otsu_thresholding function

```python
def otsu_thresholding(img):
    hist = cv2.calcHist([img], [0], None, [256], [0, 256])
    hist_norm = hist.ravel() / hist.max()

    q = np.cumsum(hist_norm)
    m = np.cumsum(hist_norm * np.arange(256))

    n = len(hist_norm)
    max_var, threshold = 0, 0
    for i in range(1, n):
        w0, w1 = q[i], q[n-1] - q[i]
        if w0 == 0 or w1 == 0:
            continue
        mu0, mu1 = m[i] / w0, (m[n-1] - m[i]) / w1
        var = w0 * w1 * (mu0 - mu1) ** 2
        if var > max_var:
            max_var = var
            threshold = i

    print(threshold)
    return cv2.threshold(img, threshold, 255, cv2.THRESH_BINARY)[1]
```

This function takes 2d array as it represented the image **img** in grayscale, then we calculate histogram and normalize it, after that we calculate the probabilities of each intensity level, then we calculate the inter-class variance for all possible thresholds.
After we get the threshold, we apply a binary threshold on the image with threshold calculated from the previous step.
Input: image array in grayscale
Output: filtered image using cv2.threshold(img,thrshold,255,cv2.THRESH_BINARY)[1]
Results

Input

Output

Local Thresholding

mean_local_threshold function

```python
def mean_local_threshold(img, block_size, constant_c):
    neighborhood_size = block_size
    C = constant_c
    output = np.zeros_like(img)
    # Iterate over each pixel in the image
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            # Compute the local threshold for the current pixel
            neighborhood = img[max(i-neighborhood_size//2, 0):min(i+neighborhood_size//2+1, img.shape[0]),
                               max(j-neighborhood_size//2, 0):min(j+neighborhood_size//2+1, img.shape[1])]
            threshold = np.mean(neighborhood) - C
            if img[i, j] > threshold:
                output[i, j] = 255
    return output
```

This function takes 2d array as it represented the image **img** in grayscale and the block size of each neighboring pixels for the main pixel and the threshold weight **constant_c**, then we make a zero 2d array with the same size of **img**, after that we iterate over each pixel and calculate the mean of its neighboring pixels after that we calculate the threshold, then check if that pixel is greater than the threshold or not, if it is we convert the same indexed pixel in the zero image from zero to 255.
Input: image array in grayscale, block_size,constant_c
Output: binary filtered image (output)

Results

Apply local threshold with block size of 11 and threshold weight of 2

Input

Output

Spectral Thresholding

spectral_thresholding function

```python
def spectral_threshold(img):
    hist, _ = np.histogram(img, 256, [0, 256])
    mean = np.sum(np.arange(256) * hist) / float(img.size)
    optimal_high = 0
    optimal_low = 0
    max_variance = 0
    for high in range(0, 256):
        for low in range(0, high):
            w0 = np.sum(hist[0:low])
            if w0 == 0:
                continue
            mean0 = np.sum(np.arange(0, low) * hist[0:low]) / float(w0)
            w1 = np.sum(hist[low:high])
            if w1 == 0:
                continue
            mean1 = np.sum(np.arange(low, high) * hist[low:high]) / float(w1)
            w2 = np.sum(hist[high:])
            if w2 == 0:
                continue
            mean2 = np.sum(np.arange(high, 256) * hist[high:]) / float(w2)
            variance = w0 * (mean0 - mean) * 2 + w1 * \
                (mean1 - mean) * 2 + w2 * (mean2 - mean) ** 2
            if variance > max_variance:
                max_variance = variance
                optimal_high = high
                optimal_low = low
    binary = np.zeros(img.shape, dtype=np.uint8)
    binary[img < optimal_low] = 0
    binary[(img >= optimal_low) & (img < optimal_high)] = 128
    binary[img >= optimal_high] = 255
    return binary

```

This function takes 2d array as it represented the image **img** in grayscale, then we compute the histogram of the image after that, we calculate the mean of the entire image.
Starting with initiation of variables for the optimal threshold values and the maximum variance **optimal_high**, **optimal_low** and **max_variance** all set to zero, then we iterate over all possible

threshold values, and select ones with maximum variance between modes, through each iteration we calculate the wight and mean of low pixels and the weight and mean for the high pixels in addition we calculate the variance and update the optimal threshold values if the variance is greater than the maximum variance.
After the iteration finished, we apply the new thresholds all over the image pixels as the pixels which has the values lower than the **optimal_low** are set to zero, the pixels which has the values greater than the **optimal_low** and lower than the **optimal_high** are set to 128, and the pixels which has values greater than the optimal high are set to 255.
Input: image array in grayscale.
Output: binary filtered image (output)

Results

Input

Output

Image Segmentation & Clustering

RGB LUV

This implementation is about mapping the images from the RGB color space to the LUV color space, which is a color space that is designed to be perceptually uniform. This process is done in the following steps:

1) split the image into its r,g,b components. then convert these sections into xyz space by multiplying them with the matrix:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

2) Convert xyz into new u v prime maps using these equations:

$$u' = \frac{4X}{X + 15Y + 3Z} = \frac{4x}{-2x + 12y + 3}$$
$$v' = \frac{9Y}{X + 15Y + 3Z} = \frac{9y}{-2x + 12y + 3}$$

3) Then, we get the l,u,v components using these equations with the outputs we got in steps 1&2:

$$L^* = \begin{cases} \left(\frac{29}{3}\right)^3 \frac{Y}{Y_n}, & \frac{Y}{Y_n} \le \left(\frac{6}{29}\right)^3 \\ 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16, & \frac{Y}{Y_n} > \left(\frac{6}{29}\right)^3 \end{cases}$$
$$u^* = 13L^* \cdot (u' - u'_n)$$
$$v^* = 13L^* \cdot (v' - v'_n)$$

4) Convert the LUV values to an image: The final step is to convert the LUV values back to an image in the LUV color space. This can be done using a library such as OpenCV or Pillow in Python.Overall, mapping RGB to LUV matrices involves converting the RGB values of each pixel in an image to corresponding LUV values using a precomputed transformation matrix and

denormalization constants. The resulting LUV values is used in our implementation of segmentation methods.

```python
def rgb_to_luv(r, g, b):
    # Convert RGB values to XYZ color space
    x = r * 0.412453 + g * 0.357580 + b * 0.180423
    y = r * 0.212671 + g * 0.715160 + b * 0.072169
    z = r * 0.019334 + g * 0.119193 + b * 0.950227

    # Convert XYZ values to LUV color space
    x_ref = 0.95047
    y_ref = 1.00000
    z_ref = 1.08883

    u_ref = (4 * x_ref) / (x_ref + (15 * y_ref) + (3 * z_ref))
    v_ref = (9 * y_ref) / (x_ref + (15 * y_ref) + (3 * z_ref))

    u_prime = (4 * x) / (x + (15 * y) + (3 * z))
    v_prime = (9 * y) / (x + (15 * y) + (3 * z))

    l = (116 * ((y / y_ref) ** (1 / 3))) - 16
    u = 13 * l * (u_prime - u_ref)
    v = 13 * l * (v_prime - v_ref)

    return l, u, v


def rgb_luv(img_path):
    rgb_image = cv2.imread(img_path)
    r, g, b = cv2.split(rgb_image)

    l, u, v = rgb_to_luv(r/50, g/13, b/40)

    luv = cv2.merge((l, u, v))
    img_path = f'./static/download/thresholding/{randint(0,9999999999999999)}_luv.png'
    cv2.imwrite(img_path, luv)
    return img_path
```

Results

Input

Output

Region growing

The function takes three parameters which are image path, seed in x axis and seed in y axis.

Algorithm:

1) we read the image and turn it to grey scale and we store seedx and seedy in list,
2) we call check points function which takes three parameters the image and seeds point list.
3) In the function:
   - We take the row and column of the image and make a black image with same dimensions.

- We swap the x and y seeds point. We make the seed point white. And we make a window of eight pixels which we will check on them around each point the points represent top right, top left, top center, left, right, bottom left, bottom right and bottom center around each point.
- The inside the while loop we make a for loop to check value for points around the desired point if it between the intensity of the desired point -8 & intensity of the desired point +8 then set it to 255 otherwise, we make it zero and then remove that point from list and check the next points. Finally, we return the final image.

```python
def check_points(img, seed):

    row, col = np.shape(img)

    region_grow = np.zeros((row+1, col+1))
    # seeds point should be inverted since seed[1] is the x coordinate and seed[0] is the y coordinate
    swap = [seed[1], seed[0]]
    region_grow[swap[0]][swap[1]] = 255
    region_points = [[swap[0], swap[1]]]
    # the window of 8 pixels that we take around each point
    x_k = [-1, 0, 1, -1, 1, -1, 0, 1]
    y_k = [-1, -1, -1, 0, 0, 1, 1, 1]
    c = 0
    while len(region_points) > 0:

        if c == 0:
            check_point = region_points.pop(0)
            i = check_point[0]
            j = check_point[1]

        intensity = img[i][j]
        low = intensity - 8
        high = intensity + 8

        for k in range(8):
            if region_grow[i + x_k[k]][j + y_k[k]] != 255:
                try:
                    if low < img[i + x_k[k]][j + y_k[k]] < high:
                        region_grow[i + x_k[k]][j + y_k[k]] = 255
                        region_points.append([i + x_k[k], j + y_k[k]])
                    else:
                        region_grow[i + x_k[k]][j + y_k[k]] = 0
                except IndexError:
                    continue
        # we remove the point the was checked and make i and j takes the values for the next point
        check_point = region_points.pop(0)
        i = check_point[0]
        j = check_point[1]
        c = c + 1
    return region_grow


def region_growing(img_path, seedx, seedy):
    print(f'seedX={seedx}, seedY={seedy}')
    img = cv2.imread(img_path, 0)
    seed_points = [seedx, seedy]
    region_grow_image = check_points(img, seed_points)
    img_path = f'./static/download/thresholding/{randint(0,9999999999999999)}_Region_growing.png'
    cv2.imwrite(img_path, region_grow_image)
    return img_path
```

Results

Region Growing with seedX of 50 and seedY of 60

Input **Output using RGB input**

Agglomerative Clustering

- The function takes three parameters which are image path, clusters numbers and initial clusters.
- We used OOP in python because the steps and variables is very related to each other.
- We initialized a class which takes two parameters, clusters numbers and initial clusters.
- In initial clusters function we partition pixels into initial clusters groups based on color similarity. We call the Euclidean distance function and calculate the Euclidean distance between point1 and point2 and we return the distance based on single link method.
- In fit function we assign each point to a distinct cluster inside the while we find the closest pair of clusters, remove the two clusters founded from the clusters list, and then collect the two clusters together and we added it to the list of clusters.
- In predict cluster and predict center functions we want to find cluster number of point and find the center of the cluster respectively.
- Finally, we take the returned image after finding the clusters and turn it to array.

```python
def euclidean_distance(point1, point2):

    return np.linalg.norm(np.array(point1) - np.array(point2))


def clusters_distance(cluster1, cluster2):
    for point1 in cluster1:
        for point2 in cluster2:
            return min(euclidean_distance(point1, point2))


def clusters_distance_2(cluster1, cluster2):

    cluster1_center = np.average(cluster1, axis=0)
    cluster2_center = np.average(cluster2, axis=0)
    return euclidean_distance(cluster1_center, cluster2_center)
```

```python
class AgglomerativeClusteringClass:

    def __init__(self, k, initial_k):
        self.k = k
        self.initial_k = initial_k

    def initial_clusters(self, points):

        groups = {}
        d = int(256 / (self.initial_k))
        for i in range(self.initial_k):
            j = i * d
            groups[(j, j, j)] = []
        for i, p in enumerate(points):
            group = min(groups.keys(), key=lambda c: euclidean_distance(p, c))
            groups[group].append(p)
        return [g for g in groups.values() if len(g) > 0]

    def fit(self, points):

        # initially, assign each point to a distinct cluster
        self.clusters_list = self.initial_clusters(points)

        while len(self.clusters_list) > self.k:

            # Find the closest pair of clusters
            cluster1, cluster2 = min([(c1, c2) for i, c1 in enumerate(self.clusters_list) for c2 in self.clusters_list[:i]],
                                     key=lambda c: clusters_distance_2(c[0], c[1]))

            # Remove the two clusters from the clusters list
            self.clusters_list = [
                c for c in self.clusters_list if c != cluster1 and c != cluster2]

            # collect the two clusters
            merged_cluster = cluster1 + cluster2

            # Add the clusters list
            self.clusters_list.append(merged_cluster)

        self.cluster = {}
        for cl_num, cl in enumerate(self.clusters_list):
            for point in cl:
                self.cluster[tuple(point)] = cl_num

        self.centers = {}
        for cl_num, cl in enumerate(self.clusters_list):
            self.centers[cl_num] = np.average(cl, axis=0)

    def predict_cluster(self, point):

        return self.cluster[tuple(point)]

    def predict_center(self, point):

        point_cluster_num = self.predict_cluster(point)
        center = self.centers[point_cluster_num]
        return center


def agglomerative_clustering(img_path, n_clusters, initial_k):
    img = cv2.imread(img_path)[:, :, ::-1]
    img_shaped = img.reshape((-1, 3))
    print(f'N clust= {n_clusters}, K initial={initial_k}')
    Agglo = AgglomerativeClusteringClass(n_clusters, initial_k)

    Agglo.fit(img_shaped)

    new_img = [[Agglo.predict_center(list(pixel))
                for pixel in row] for row in img]
    new_img = np.array(new_img, np.uint8)
    new_img = new_img[:, :, ::-1]
    img_path = f'./static/download/thresholding/{randint(0,9999999999999999)}_AgglomerativeClustering.png'

    cv2.imwrite(img_path, new_img)
    return img_path

```

Results

Agglomerative Clustering with n_clusters of 4 and initial_k of 20

Input

Output using RGB input

Mean Shift Clustering

The Mean Shift segmentation is a local homogenization technique that is very useful for damping shading or tonality differences in localized objects. In our implementation we made two functions:

```python
def segment(image_path, kernel_bandwidth):
    # read the image path and convert it into the LUV domain
    image = cv.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2LUV)

    # reshape image to a 2D array
    pixels = np.reshape(
        image, (image.shape[0] * image.shape[1], image.shape[2]))

    # run mean shift clustering
    centroids = mean_shift(pixels, kernel_bandwidth)

    # assign each pixel to a cluster
    nbrs = NearestNeighbors(n_neighbors=1).fit(centroids)
    distances, indices = nbrs.kneighbors(pixels)

    # reshape the indices back to the original image shape
    indices = np.reshape(indices, (image.shape[0], image.shape[1]))

    # create a segmented image
    segmented = np.zeros_like(image)
    for i in range(len(centroids)):
        segmented[indices == i] = centroids[i]

    return segmented
```

At the first, you call the segment function with inputs image path and bandwidth size as you like. The steps of this function are:

- We read the image path and convert it into the luv domain.
- We reshape the image array into 2d to do processing on it.
- To get the centroids (points which are the center of each cluster), we call the mean_shift function:

```python
def mean_shift_fast(img_path):
    img = cv.imread(rgb_luv.rgb_luv(img_path))

    # filter to reduce noise
    img = cv.medianBlur(img, 3)

    # flatten the image
    flat_image = img.reshape((-1, 3))
    flat_image = np.float32(flat_image)

    # meanshift
    bandwidth = estimate_bandwidth(flat_image, quantile=.06, n_samples=3000)
    ms = MeanShift(bandwidth=bandwidth, bin_seeding=True, max_iter=800)
    ms.fit(flat_image)
    labeled = ms.labels_

    # get number of segments
    segments = np.unique(labeled)

    # get the average color of each segment
    total = np.zeros((segments.shape[0], 3), dtype=float)
    count = np.zeros(total.shape, dtype=float)
    for i, label in enumerate(labeled):
        total[label] = total[label] + flat_image[i]
        count[label] += 1
    avg = total/count
    avg = np.uint8(avg)

    # cast the labeled image into the corresponding average color
    res = avg[labeled]
    result = res.reshape((img.shape))
    return result

```

In this function we make the centroids by getting between each point and centroid to initialize the centroid values. Then, we loop across the image array to check if the distance we got is zero, so these points are centroids but if it is not, so we need to get the mean value to get new centroids and so on. After that we check for convergence (check if these points are got centroids before or not). Finally, if not all centroids are done, we add the new centroids and loop again to get the new mean centroid.

- After getting the centroids we return to segment function and then, assign each pixel in the image to a cluster of each centroid.
- Then, we reshape the 2d array into the original image shape.
- Finally, we make the segmented image ready to be shown.

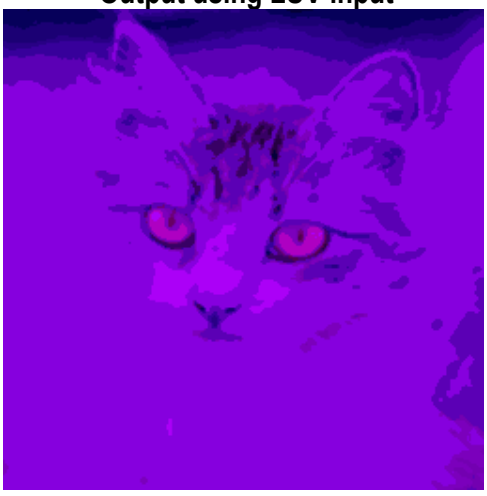
Note:

The implementation of this algorithm may take a long time because it goes throw all the points and does a long processing. So, you can wait or use a fast cv2 code.

Results

Input

Output using LUV input



K-Means Clustering

K-means image segmentation is a technique used to divide an image into multiple segments or regions based on the similarities in pixel color or intensity values.
We implemented this technique in multiple steps, but the main function is:

```python
def kmeans_segmentation(img_path, n_clus, max_iter):
    print("K Means")
    image = cv2.imread(rgb_luv.rgb_luv(img_path))  # Read image
    X = image.reshape((-1, 3))  # Reshape to (Npts, Ndim = 3)
    X = np.float32(X)

    # Call the kmeans class
    km = KMeans(n_clus, max_iter)
    km.fit(X)
    centers = km.getCentroids()
    clusters = km.getClusters()

    segmented_image = centers[clusters]

    segmented_image = segmented_image.reshape((image.shape))
    return segmented_image

```

- We read the image path and convert it into the luv domain.
- We reshape the image array into 2d to do processing on it.
- Then, we call the kmeans class to initialize the values of k and the maximum iteration:

```python
class KMeans():
    def __init__(self, n_clus, max_iter):

        self.n_clus = n_clus  # Number of clusters
        self.centroids = None
        self.X = None
        self.clusters = None
        self.max_iter = max_iter
```

- Then, we call the fit function in this class:

```python
def fit(self, X, init_state=None):
        Npts, Ndim = X.shape
        self.X = X

        if init_state is None:
            X_max, X_min = np.max(X), np.min(X)
            self.centroids = np.random.uniform(
                low=X_min, high=X_max, size=(self.n_clus, Ndim))
        else:
            self.centroids = init_state

        for i in range(self.max_iter):

            diff = cdist(X, self.centroids, metric="euclidean")
            self.clusters = np.argmin(diff, axis=1)

            for i in range(self.n_clus):
                self.centroids[i] = np.mean(
                    X[np.where(self.clusters == i)], axis=0)
```

In this function, we assign pixels to clusters: For each pixel in the image, calculate the distance to each centroid and assign the pixel to the cluster with the nearest centroid. Then, update the centroids: After all pixels have been assigned to clusters, update the centroid values by calculating the mean color or intensity value for each cluster.

- Then, we get the centroids values and their clusters:

```python
def getCentroids(self):
        return self.centroids

    def getClusters(self):
        return self.clusters
```

When we got these now, we can make the segmented image and return to the image shape again to show it.

Results

K-means clustering with K of 3 and max iterations of 20

Input

Output using LUV input

Image Matching

SIFT Implementation

SIFT (scale invariant features transform) algorithm is a highly processing technique with some steps to locate the local features in an image, commonly known as the 'key points of the image. These key points are scale & rotation invariants that can be used for various computer vision applications, like image matching, object detection, scene detection, etc.

Steps of implementation:

1) Initializing the main borders of the implementation

SIFT implementation has many steps to work on. So, we used OOP in python to make it easy to understand the steps:

```python
class SIFT:
    def __init__(self, image_path,pre_segma = 0.5) -> None:
        self.float_tolerance = 1e-7
        self.image_path = image_path
        self.pre_segma = pre_segma

    def computeKeypointsAndDescriptors(self,sigma=1.6, num_intervals=3, image_border_width=5):
        image = cv.imread(self.image_path,flags = cv.IMREAD_GRAYSCALE)
        image = image.astype('float32')
        base_image = self.generate_base_image(image, sigma, self.pre_segma)
        num_octaves = self.compute_number_of_octaves(base_image.shape)
        gaussian_segmas = self.generate_gaussian_segmas(sigma, num_intervals)
        gaussian_images = self.generate_gaussian_images(base_image, num_octaves, gaussian_segmas)
        dog_images = self.generate_DoG_images(gaussian_images)
        keypoints = self.findScaleSpaceExtrema(gaussian_images, dog_images, num_intervals, sigma, image_border_width)
        keypoints = self.remove_duplicate_key_points(keypoints)
        keypoints = self.convert_key_points_to_input_image_size(keypoints)
        descriptors = self.generate_descriptors(keypoints, gaussian_images)
        return keypoints,descriptors
```

We initialized a class which takes the image path as the main attribute when calling it. Then, we initialized computeKeypointsAndDescriptors function which has the whole main steps of the algorithm

as it converts the image into greyscale then it calls all the functions of the implementation and then returns the **keypoints** and descriptors which we need from this algorithm.

2) Constructing a Scale Space

we need to ensure that the features are not scale-dependent:

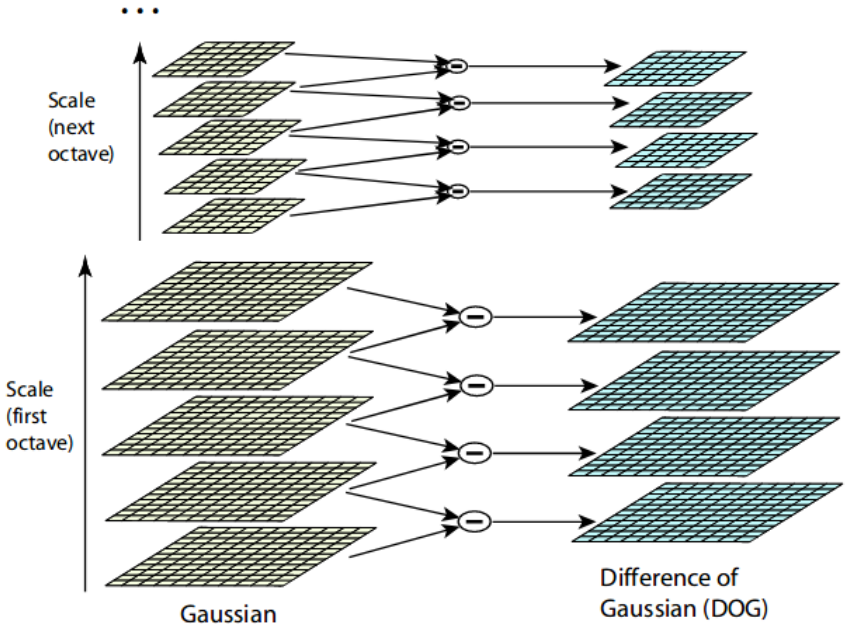
```python
def generate_base_image(self, image, sigma, pre_segma):
        image = cv.resize(image, (0, 0), fx=2, fy=2,
                          interpolation=cv.INTER_LINEAR)
        sigma_diff = np.sqrt(
            max((sigma ** 2) - ((2 * pre_segma) ** 2), 0.01))
        return cv.GaussianBlur(image, (0, 0), sigmaX=sigma_diff, sigmaY=sigma_diff)

    def compute_number_of_octaves(self, image_shape):
        return int(np.round(np.log(min(image_shape)) / np.log(2) - 1))
```

At the first, we need to generate the base image for applying gaussian filter to keep only the relevant information, like the shape and edges. Then, we computed the number of octaves in the image pyramid depending on the shape of input image.

Then, we need to generate the octaves: each octave has some gaussian images with the same size but with different segmas. So, we used generate_gaussian_segmas function to generate an array with different segmas values to apply in each octave. Then generate_gaussian_images function is used to apply these segmas with gaussian filter on the images to generate an octave and store this octave in the gaussian image array.

```python
def generate_gaussian_segmas(self, sigma, num_intervals):
        num_images_per_octave = num_intervals + 3
        k = 2 ** (1. / num_intervals)
        gaussian_segmas = np.zeros(num_images_per_octave)
        gaussian_segmas[0] = sigma

        for image_index in range(1, num_images_per_octave):
            sigma_previous = (k ** (image_index - 1)) * sigma
            sigma_total = k * sigma_previous
            gaussian_segmas[image_index] = np.sqrt(
                sigma_total ** 2 - sigma_previous ** 2)

        return gaussian_segmas

    def generate_gaussian_images(self, image, num_octaves, gaussian_segmas):
        gaussian_images = []
        for octave_index in range(num_octaves):
            gaussian_images_in_octave = []
            gaussian_images_in_octave.append(image)
            for gaussian_segma in gaussian_segmas[1:]:
                image = cv.GaussianBlur(
                    image, (0, 0), sigmaX=gaussian_segma, sigmaY=gaussian_segma)
                gaussian_images_in_octave.append(image)
            gaussian_images.append(gaussian_images_in_octave)
            octave_base = gaussian_images_in_octave[-3]
            image = cv.resize(octave_base, (int(
                octave_base.shape[1] / 2), int(octave_base.shape[0] / 2)), interpolation=cv.INTER_NEAREST)
        return np.array(gaussian_images, dtype=object)

    def generate_DoG_images(self, gaussian_images):
        dog_images = []
        for gaussian_images_in_octave in gaussian_images:
            dog_images_in_octave = []
            for first_image, second_image in zip(gaussian_images_in_octave, gaussian_images_in_octave[1:]):
                # ordinary subtraction will not work because the images are unsigned integers
                dog_images_in_octave.append(
                    np.subtract(second_image, first_image))
            dog_images.append(dog_images_in_octave)
        return np.array(dog_images, dtype=object)
```

Then, we use generate_DoG_images function to substrate each pair of images in each octave array of images in the gaussian_images array as shown in the figure below:

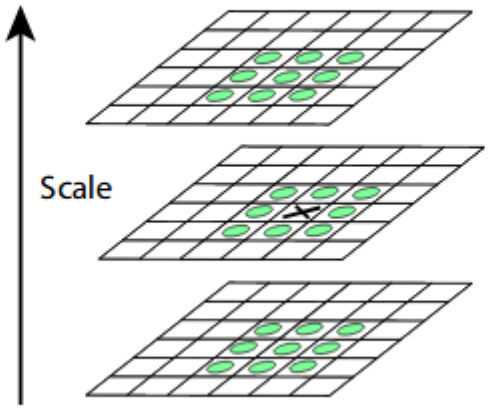


3) Keypoint Localization

The next step is to find the important keypoints from the image that can be used for feature matching:

```python
def findScaleSpaceExtrema(self, gaussian_images, dog_images, num_intervals, sigma, image_border_width, contrast_threshold=0.04
):
        # from OpenCV implementation
        threshold = np.floor(0.5 * contrast_threshold / num_intervals * 255)
        keypoints = []
        for octave_index, dog_images_in_octave in enumerate(dog_images):
            for image_index, (first_image, second_image, third_image) in enumerate(zip(dog_images_in_octave, dog_images_in_oct
ave[1:], dog_images_in_octave[2:])):
                # (i, j) is the center of the 3x3 array
                for i in range(image_border_width, first_image.shape[0] - image_border_width):
                    for j in range(image_border_width, first_image.shape[1] - image_border_width):
                        if self.is_pixel_an_extremum(first_image[i-1:i+2, j-1:j+2], second_image[i-1:i+2, j-1:j+2], third_imag
e[i-1:i+2, j-1:j+2], threshold):
                            localization_result = self.localize_extremum_via_quadratic_fit(
                                i, j, image_index + 1, octave_index, num_intervals, dog_images_in_octave, sigma, contrast_thre
shold, image_border_width)
                            if localization_result is not None:
                                keypoint, localized_image_index = localization_result
                                keypoints_with_orientations = self.compute_key_points_with_orientations(
                                    keypoint, octave_index, gaussian_images[octave_index][localized_image_index])
                                for keypoint_with_orientation in keypoints_with_orientations:
                                    keypoints.append(keypoint_with_orientation)
        return keypoints
```

Here is the function findScaleSpaceExtrema which returns the keypoints of the image as we define the threshold which we will take the keypoints depending on it. Then, we loop on the DOG images and take three consecutive images to process on them as shown in the figure:



Then, we loop on i and j coordinates which mean 3x3 shape on each image.
Then, we make sure that the pixels chosen are above or below threshold using is_pixel_an_extremum function:

```python
def is_pixel_an_extremum(self, first_subimage, second_subimage, third_subimage, threshold):
        center_pixel_value = second_subimage[1, 1]
        if abs(center_pixel_value) > threshold:
            if center_pixel_value > 0:
                return np.all(center_pixel_value >= first_subimage) and \
                    np.all(center_pixel_value >= third_subimage) and \
                    np.all(center_pixel_value >= second_subimage[0, :]) and \
                    np.all(center_pixel_value >= second_subimage[2, :]) and \
                    center_pixel_value >= second_subimage[1, 0] and \
                    center_pixel_value >= second_subimage[1, 2]
            elif center_pixel_value < 0:
                return np.all(center_pixel_value <= first_subimage) and \
                    np.all(center_pixel_value <= third_subimage) and \
                    np.all(center_pixel_value <= second_subimage[0, :]) and \
                    np.all(center_pixel_value <= second_subimage[2, :]) and \
                    center_pixel_value <= second_subimage[1, 0] and \
                    center_pixel_value <= second_subimage[1, 2]
        return False
```

Then, we make sure that there is no keypoints with noise and eliminate them using Hessain matrix implementation in math and reduce the key points to just important ones using localize_extremum_via_quadratic_fit function:

```python
def localize_extremum_via_quadratic_fit(self, i, j, image_index, octave_index, num_intervals, dog_images_in_octave, sigma, contrast_threshold, image_border_width, eigenvalue_ratio=10, num_attempts_until_convergence=5):
        extremum_is_outside_image = False
        image_shape = dog_images_in_octave[0].shape
        for attempt_index in range(num_attempts_until_convergence):
            # need to convert from uint8 to float32 to compute derivatives and need to rescale pixel values to [0, 1] to apply Lowe's thresholds
            first_image, second_image, third_image = dog_images_in_octave[
                image_index-1:image_index+2]
            pixel_cube = np.stack([first_image[i-1:i+2, j-1:j+2],
                                   second_image[i-1:i+2, j-1:j+2],
                                   third_image[i-1:i+2, j-1:j+2]]).astype('float32') / 255.
            gradient = self.compute_gradient_at_center_pixel(pixel_cube)
            hessian = self.compute_hessian_at_center_pixel(pixel_cube)
            extremum_update = - \
                np.linalg.lstsq(hessian, gradient, rcond=None)[0]
            if abs(extremum_update[0]) < 0.5 and abs(extremum_update[1]) < 0.5 and abs(extremum_update[2]) < 0.5:
                break
            j += int(np.round(extremum_update[0]))
            i += int(np.round(extremum_update[1]))
            image_index += int(np.round(extremum_update[2]))
            # make sure the new pixel_cube will lie entirely within the image
            if i < image_border_width or i >= image_shape[0] - image_border_width or j < image_border_width or j >= image_shape[1] - image_border_width or image_index < 1 or image_index > num_intervals:
                extremum_is_outside_image = True
                break
        if extremum_is_outside_image:
            return None
        if attempt_index >= num_attempts_until_convergence - 1:
            return None
        functionValueAtUpdatedExtremum = pixel_cube[1,
                                                    1, 1] + 0.5 * np.dot(gradient, extremum_update)
        if abs(functionValueAtUpdatedExtremum) * num_intervals >= contrast_threshold:
            xy_hessian = hessian[:2, :2]
            xy_hessian_trace = np.trace(xy_hessian)
            xy_hessian_det = np.linalg.det(xy_hessian)
            if xy_hessian_det > 0 and eigenvalue_ratio * (xy_hessian_trace ** 2) < ((eigenvalue_ratio + 1) ** 2) * xy_hessian_det:
                # Contrast check passed -- construct and return OpenCV KeyPoint object
                keypoint = cv.KeyPoint()
                keypoint.pt = (
                    (j + extremum_update[0]) * (2 ** octave_index), (i + extremum_update[1]) * (2 ** octave_index))
                keypoint.octave = octave_index + image_index * \
                    (2 ** 8) + \
                    int(np.round((extremum_update[2] + 0.5) * 255)) * (2 ** 16)
                keypoint.size = sigma * (2 ** ((image_index + extremum_update[2]) / np.float32(num_intervals))) * (
                    2 ** (octave_index + 1))  # octave_index + 1 because the input image was doubled
                keypoint.response = abs(functionValueAtUpdatedExtremum)
                return keypoint, image_index
        return None
```

4) Orientation Assignment

We will now assign an orientation to each of these keypoints so that they are invariant to rotation:

```python
def compute_key_points_with_orientations(self, keypoint, octave_index, gaussian_image, radius_factor=3, num_bins=36, peak_ratio=0.8, scale_factor=1.5):

        keypoints_with_orientations = []
        image_shape = gaussian_image.shape

        # compare with keypoint.size computation in localizeExtremumViaQuadraticFit()
        scale = scale_factor * keypoint.size / \
            np.float32(2 ** (octave_index + 1))
        radius = int(np.round(radius_factor * scale))
        weight_factor = -0.5 / (scale ** 2)
        raw_histogram = np.zeros(num_bins)
        smooth_histogram = np.zeros(num_bins)

        for i in range(-radius, radius + 1):
            region_y = int(
                np.round(keypoint.pt[1] / np.float32(2 ** octave_index))) + i
            if region_y > 0 and region_y < image_shape[0] - 1:
                for j in range(-radius, radius + 1):
                    region_x = int(
                        np.round(keypoint.pt[0] / np.float32(2 ** octave_index))) + j
                    if region_x > 0 and region_x < image_shape[1] - 1:
                        dx = gaussian_image[region_y, region_x + 1] - \
                            gaussian_image[region_y, region_x - 1]
                        dy = gaussian_image[region_y - 1, region_x] - \
                            gaussian_image[region_y + 1, region_x]
                        gradient_magnitude = np.sqrt(dx * dx + dy * dy)
                        gradient_orientation = np.rad2deg(np.arctan2(dy, dx))
                        # constant in front of exponential can be dropped because we will find peaks later
                        weight = np.exp(weight_factor * (i ** 2 + j ** 2))
                        histogram_index = int(
                            np.round(gradient_orientation * num_bins / 360.))
                        raw_histogram[histogram_index %
                                      num_bins] += weight * gradient_magnitude

        for n in range(num_bins):
            smooth_histogram[n] = (6 * raw_histogram[n] + 4 * (raw_histogram[n - 1] + raw_histogram[(
                n + 1) % num_bins]) + raw_histogram[n - 2] + raw_histogram[(n + 2) % num_bins]) / 16.
        orientation_max = max(smooth_histogram)
        orientation_peaks = np.where(np.logical_and(smooth_histogram > np.roll(
            smooth_histogram, 1), smooth_histogram > np.roll(smooth_histogram, -1)))[0]
        for peak_index in orientation_peaks:
            peak_value = smooth_histogram[peak_index]
            if peak_value >= peak_ratio * orientation_max:
                # Quadratic peak interpolation
                # The interpolation update is given by equation (6.30) in https://ccrma.stanford.edu/~jos/sasp/Quadratic_Interpolation_Spectral_Peaks.html
                left_value = smooth_histogram[(peak_index - 1) % num_bins]
                right_value = smooth_histogram[(peak_index + 1) % num_bins]
                interpolated_peak_index = (peak_index + 0.5 * (left_value - right_value) / (
                    left_value - 2 * peak_value + right_value)) % num_bins
                orientation = 360. - interpolated_peak_index * 360. / num_bins
                if abs(orientation - 360.) < self.float_tolerance:
                    orientation = 0
                new_keypoint = cv.KeyPoint(
                    *keypoint.pt, keypoint.size, orientation, keypoint.response, keypoint.octave)
                keypoints_with_orientations.append(new_keypoint)
        return keypoints_with_orientations

```

5) Keypoint Descriptor

This is the final step for SIFT. So far, we have stable keypoints that are scale-invariant and rotation-invariant. In this section, we will use the neighboring pixels, their orientations, and their magnitude to generate a unique fingerprint for this keypoint called a ‘descriptor’.

```python
def compare_key_points(self, keypoint1, keypoint2):
        if keypoint1.pt[0] != keypoint2.pt[0]:
            return keypoint1.pt[0] - keypoint2.pt[0]
        if keypoint1.pt[1] != keypoint2.pt[1]:
            return keypoint1.pt[1] - keypoint2.pt[1]
        if keypoint1.size != keypoint2.size:
            return keypoint2.size - keypoint1.size
        if keypoint1.angle != keypoint2.angle:
            return keypoint1.angle - keypoint2.angle
        if keypoint1.response != keypoint2.response:
            return keypoint2.response - keypoint1.response
        if keypoint1.octave != keypoint2.octave:
            return keypoint2.octave - keypoint1.octave
        return keypoint2.class_id - keypoint1.class_id

    def convert_key_points_to_input_image_size(self, keypoints):
        converted_keypoints = []
        for keypoint in keypoints:
            keypoint.pt = tuple(0.5 * np.array(keypoint.pt))
            keypoint.size *= 0.5
            keypoint.octave = (keypoint.octave & ~255) | (
                (keypoint.octave - 1) & 255)
            converted_keypoints.append(keypoint)
        return converted_keypoints

    def remove_duplicate_key_points(self, keypoints):

        if len(keypoints) < 2:
            return keypoints
        keypoints.sort(key=cmp_to_key(self.compare_key_points))
        unique_keypoints = [keypoints[0]]

        for next_keypoint in keypoints[1:]:
            last_unique_keypoint = unique_keypoints[-1]
            if last_unique_keypoint.pt[0] != next_keypoint.pt[0] or \
                    last_unique_keypoint.pt[1] != next_keypoint.pt[1] or \
                    last_unique_keypoint.size != next_keypoint.size or \
                    last_unique_keypoint.angle != next_keypoint.angle:
                unique_keypoints.append(next_keypoint)

        return unique_keypoints

    def unpack_octave(self, keypoint):
        octave = keypoint.octave & 255
        layer = (keypoint.octave >> 8) & 255
        if octave >= 128:
            octave = octave | -128
        scale = 1 / \
            np.float32(1 << octave) if octave >= 0 else np.float32(
                1 << -octave)
        return octave, layer, scale
```

At first, we do some steps before descriptor generation:

- We use compareKeypoints function to compare between pairs of keypoints and return true if keypoint1 is less than keypoint2.
- We use removeDuplicateKeypoints function to sort keypoints and remove duplicate keypoints.
- We use convertKeypointsToInputImageSize function to convert keypoint point, size, and octave to input image size.
- We use unpackOctave function to compute octave, layer, and scale from a keypoint.

Then, we implement the descriptor function:

```python
def generate_descriptors(self, keypoints, gaussian_images, window_width=4, num_bins=8, scale_multiplier=3, descriptor_max_valu
e=0.2):
       descriptors = []
       for keypoint in keypoints:
           octave, layer, scale = self.unpack_octave(keypoint)
           gaussian_image = gaussian_images[octave + 1, layer]
           num_rows, num_cols = gaussian_image.shape
           point = np.round(scale * np.array(keypoint.pt)).astype('int')
           bins_per_degree = num_bins / 360.
           angle = 360. - keypoint.angle
           cos_angle = np.cos(np.deg2rad(angle))
           sin_angle = np.sin(np.deg2rad(angle))
           weight_multiplier = -0.5 / ((0.5 * window_width) ** 2)
           row_bin_list = []
           col_bin_list = []
           magnitude_list = []
           orientation_bin_list = []
           # first two dimensions are increased by 2 to account for border effects
           histogram_tensor = np.zeros(
               (window_width + 2, window_width + 2, num_bins))
```

```
# Descriptor window size (described by half_width) follows OpenCV convention
            hist_width = scale_multiplier * 0.5 * scale * keypoint.size
            # sqrt(2) corresponds to diagonal length of a pixel
            half_width = int(np.round(hist_width * np.sqrt(2)
                                      * (window_width + 1) * 0.5))
            # ensure half_width lies within image
            half_width = int(
                min(half_width, np.sqrt(num_rows ** 2 + num_cols ** 2)))

            for row in range(-half_width, half_width + 1):
                for col in range(-half_width, half_width + 1):
                    row_rot = col * sin_angle + row * cos_angle
                    col_rot = col * cos_angle - row * sin_angle
                    row_bin = (row_rot / hist_width) + 0.5 * window_width - 0.5
                    col_bin = (col_rot / hist_width) + 0.5 * window_width - 0.5
                    if row_bin > -1 and row_bin < window_width and col_bin > -1 and col_bin < window_width:
                        window_row = int(np.round(point[1] + row))
                        window_col = int(np.round(point[0] + col))
                        if window_row > 0 and window_row < num_rows - 1 and window_col > 0 and window_col < num_cols - 1:
                            dx = gaussian_image[window_row, window_col + 1] - \
                                gaussian_image[window_row, window_col - 1]
                            dy = gaussian_image[window_row - 1, window_col] - \
                                gaussian_image[window_row + 1, window_col]
                            gradient_magnitude = np.sqrt(dx * dx + dy * dy)
                            gradient_orientation = np.rad2deg(
                                np.arctan2(dy, dx)) % 360
                            weight = np.exp(
                                weight_multiplier * ((row_rot / hist_width) ** 2 + (col_rot / hist_width) ** 2))
                            row_bin_list.append(row_bin)
                            col_bin_list.append(col_bin)
                            magnitude_list.append(weight * gradient_magnitude)
                            orientation_bin_list.append(
                                (gradient_orientation - angle) * bins_per_degree)

            for row_bin, col_bin, magnitude, orientation_bin in zip(row_bin_list, col_bin_list, magnitude_list, orientation_bi
n_list):
                # Smoothing via trilinear interpolation
                # Notations follows https://en.wikipedia.org/wiki/Trilinear_interpolation
                # Note that we are really doing the inverse of trilinear interpolation here (we take the center value of the c
ube and distribute it among its eight neighbors)
                row_bin_floor, col_bin_floor, orientation_bin_floor = np.floor(
                    [row_bin, col_bin, orientation_bin]).astype(int)
                row_fraction, col_fraction, orientation_fraction = row_bin - \
                    row_bin_floor, col_bin - col_bin_floor, orientation_bin - orientation_bin_floor
                if orientation_bin_floor < 0:
                    orientation_bin_floor += num_bins
                if orientation_bin_floor >= num_bins:
                    orientation_bin_floor -= num_bins

                c1 = magnitude * row_fraction
                c0 = magnitude * (1 - row_fraction)
                c11 = c1 * col_fraction
                c10 = c1 * (1 - col_fraction)
                c01 = c0 * col_fraction
                c00 = c0 * (1 - col_fraction)
                c111 = c11 * orientation_fraction
                c110 = c11 * (1 - orientation_fraction)
                c101 = c10 * orientation_fraction
                c100 = c10 * (1 - orientation_fraction)
                c011 = c01 * orientation_fraction
                c010 = c01 * (1 - orientation_fraction)
                c001 = c00 * orientation_fraction
                c000 = c00 * (1 - orientation_fraction)

                histogram_tensor[row_bin_floor + 1,
                                 col_bin_floor + 1, orientation_bin_floor] += c000
                histogram_tensor[row_bin_floor + 1, col_bin_floor +
                                 1, (orientation_bin_floor + 1) % num_bins] += c001
                histogram_tensor[row_bin_floor + 1,
                                 col_bin_floor + 2, orientation_bin_floor] += c010
                histogram_tensor[row_bin_floor + 1, col_bin_floor +
                                 2, (orientation_bin_floor + 1) % num_bins] += c011
                histogram_tensor[row_bin_floor + 2,
                                 col_bin_floor + 1, orientation_bin_floor] += c100
                histogram_tensor[row_bin_floor + 2, col_bin_floor +
                                 1, (orientation_bin_floor + 1) % num_bins] += c101
                histogram_tensor[row_bin_floor + 2,
                                 col_bin_floor + 2, orientation_bin_floor] += c110
                histogram_tensor[row_bin_floor + 2, col_bin_floor +
                                 2, (orientation_bin_floor + 1) % num_bins] += c111

```

```
# Remove histogram borders
            descriptor_vector = histogram_tensor[1:-1, 1:-1, :].flatten()
            # Threshold and normalize descriptor_vector
            threshold = np.linalg.norm(
                descriptor_vector) * descriptor_max_value
            descriptor_vector[descriptor_vector > threshold] = threshold
            descriptor_vector /= max(np.linalg.norm(descriptor_vector), self.float_tolerance)
            # Multiply by 512, np.round, and saturate between 0 and 255 to convert from float32 to unsigned char (OpenCV convention)
            descriptor_vector = np.round(512 * descriptor_vector)
            descriptor_vector[descriptor_vector < 0] = 0
            descriptor_vector[descriptor_vector > 255] = 255
            descriptors.append(descriptor_vector)

        return np.array(descriptors, dtype='float32')
```

feature_matching function

```
def feature_matching(img_path1, img_path2, threshold, mode):

    # read images
    img1 = cv2.imread(img_path1, 0)
    img2 = cv2.imread(img_path2, 0)

    sift1 = SIFTimplementation.SIFT(img_path1)
    keypoints_1, descriptors_1 = sift1.computeKeypointsAndDescriptors()
    sift_time_st = time.time()
    sift2 = SIFTimplementation.SIFT(img_path2)
    keypoints_2, descriptors_2 = sift2.computeKeypointsAndDescriptors()
    sift_time_end = time.time()
    sift_total_time = sift_time_end-sift_time_st
    if mode == 'ssd':
        print("ssd")
        start = time.time()
        matches = ssd_matching(keypoints_1, keypoints_2,
                               descriptors_1, descriptors_2, threshold)
        end = time.time()
        total_time = end-start

    elif mode == 'ncc':
        print("ncc")
        start = time.time()
        matches = ncc_matching(keypoints_1, keypoints_2,
                               descriptors_1, descriptors_2, threshold)
        end = time.time()
        total_time = end-start
    new_path = drawMatches(img1, keypoints_1, img2, keypoints_2, matches[:10])
    print(total_time)
    return total_time, new_path, sift_total_time

```

Description

It is the main function of SSD or NCC methods for image matching as it takes four parameters img_path1, img_path2, threshold and mode (“ncc”, “ssd”).

The values of threshold depend on the method used, for example; if NCC used the values of threshold are preferred to be in range of]0,1[, and if SSD used the values of threshold are preferred to be in range [1000,10000].

The output is the path of the result image and total time computed while using NCC or SSD methods, and the total time of computed while using SIFT method.

SSD matching function

```python
def ssd_matching(keypoints_1, keypoints_2, desc1, desc2, threshold):

    matches = []
    for i in range(len(desc1)):
        for j in range(len(desc2)):
            ssd = np.sum(np.square(desc1[i]-desc2[j]))
            if ssd < threshold:
                matches.append([i, j, ssd])
    final = []
    for i in range(len(matches)):
        dis = np.linalg.norm(np.array(
            keypoints_1[matches[i][0]].pt) - np.array(keypoints_2[matches[i][1]].pt))
        final.append(cv2.DMatch(matches[i][0], matches[i][1], dis))
    return final
```

Description

The function takes five parameters key point and descriptors for both images and threshold taking from the user.
The first loop is used to take every descriptor in image1 and subtract it from image2 and sum the result if the result is smaller than the threshold, we append the result in matches list because in sum of square difference the low score means high similarity.in second loop we calculate the distance between the key points for image1 and key points for image2 based on the match that we got and make vector normalization. Finally, we use the matching function to get the final result between points and append it in list.

Draw_matches function:

```python
def drawMatches(img1, kp1, img2, kp2, matches):

    rows1 = img1.shape[0]
    cols1 = img1.shape[1]
    rows2 = img2.shape[0]
    cols2 = img2.shape[1]

    out = np.zeros((max([rows1, rows2]), cols1+cols2, 3), dtype='uint8')

    out[:rows1, :cols1, :] = np.dstack([img1, img1, img1])

    out[:rows2, cols1:cols1+cols2, :] = np.dstack([img2, img2, img2])

    for count, mat in enumerate(matches):

        img1_idx = mat.queryIdx
        img2_idx = mat.trainIdx

        (x1, y1) = kp1[img1_idx].pt
        (x2, y2) = kp2[img2_idx].pt

        cv2.circle(out, (int(x1), int(y1)), 4,
                   ((count+2)*10, count*25, count*30), 1)
        cv2.circle(out, (int(x2)+cols1, int(y2)), 4,
                   ((2+count)*10, count*25, count*30), 1)

        cv2.line(out, (int(x1), int(y1)), (int(x2)+cols1, int(y2)),
                 ((2+count)*10, count*25, count*30), 1)

    img_path = f'./static/download/ssd/{randint(0,9999999999999999)}_feature.png'
    cv2.imwrite(img_path, out)
    return img_path
```

Description

The function takes five parameters' images, key point for both images and the matching points. we Create a new output image that concatenates the two images together. Place the first image to the left and place the next image to the right of it. For each pair of points, we have between both images we draw circles, then connect a line between them. Then we get the matching key points for each of the images. We draw a small circle at both coordinates and connect between them with a line. Finally, we save the image in path and return it

NCC matching function

```python
def ncc_matching(keypoints_1, keypoints_2, desc1, desc2, threshold):

    matches = []

    for i in range(len(desc1)):
        for j in range(len(desc2)):
            out1_norm = (desc1[i] - np.mean(desc1[i])) / (np.std(desc1[i]))
            out2_norm = (desc2[j] - np.mean(desc2[j])) / (np.std(desc2[j]))
            corr_vector = np.multiply(out1_norm, out2_norm)
            corr = float(np.mean(corr_vector))
            if corr > threshold:
                matches.append([i, j, corr])

    final = []
    for i in range(len(matches)):
        dis = np.linalg.norm(np.array(
            keypoints_1[matches[i][0]].pt) - np.array(keypoints_2[matches[i][1]].pt))
        final.append(cv2.DMatch(matches[i][0], matches[i][1], dis))
    return final
```

Description

The function takes five parameters key point and descriptors for both images and threshold taking from the user.

The first loop is used to take every descriptor in image1 and subtract it from the meaning of it and divide it by its. Then we multiply the two results and take the mean standard deviation and do the same for image 2. The normalized cross correlation is opposite to the sum of square difference high score means high similiarty.in second loop we calculate the distance between the key points for image1 and key points for image2 based on the match that we got and make vector normalization. Finally, we use the matching function to get the final result between points and append it in list.

Results

Applying NCC with threshold of 0.99

Input image 1

Input image 2

Output

Applying SSD with threshold of 1999

Input image 1

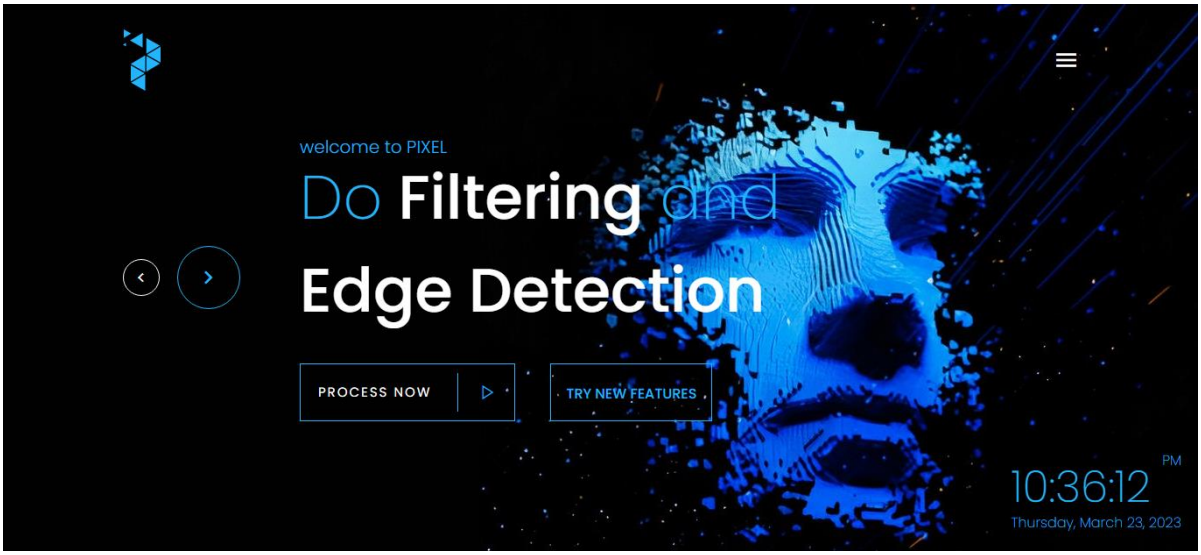
Input image 2

Output

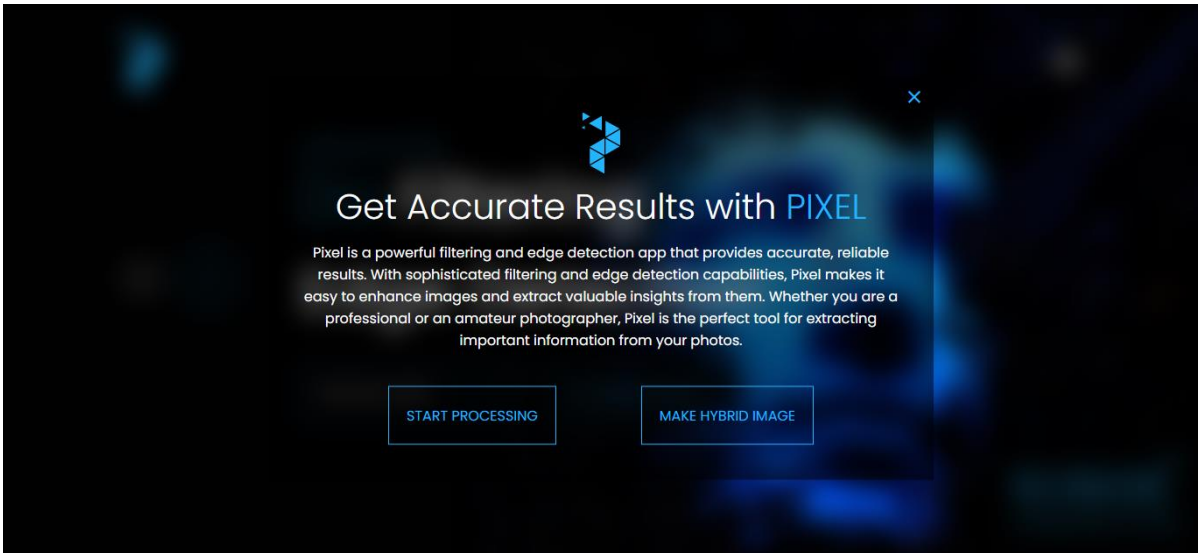
How to apply some new updates

Harris Corner Detection

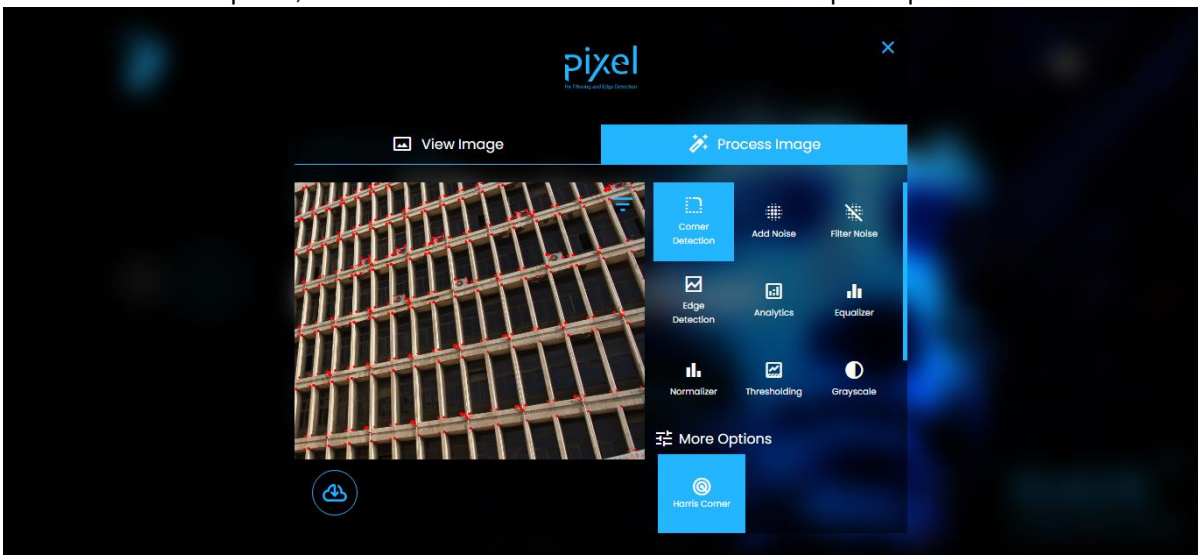
Press **PROCESS NOW** button



Choose **START PROCESSING**



After that choose an image to be processed, then choose **process image** and choose **corner detection** from the panel, after that choose **Harris Corner** from more options panel.



Algorithms of New Features

Corner Detection

apply_harris_corner function

```python
def apply_harris_corner(img_dir, window_size, k, threshold):
    img = cv2.imread(img_dir)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_gaussian = cv2.GaussianBlur(gray, (3, 3), 0)
    height = img.shape[0]
    width = img.shape[1]
    matrix_R = np.zeros((height, width))
    dx = cv2.Sobel(img_gaussian, cv2.CV_64F, 1, 0, ksize=3)
    dy = cv2.Sobel(img_gaussian, cv2.CV_64F, 0, 1, ksize=3)
    dx2 = np.square(dx)
    dy2 = np.square(dy)
    dxy = dx*dy
    offset = int(window_size / 2)
    for y in range(offset, height-offset):
        for x in range(offset, width-offset):
            Sx2 = np.sum(dx2[y-offset:y+1+offset, x-offset:x+1+offset])
            Sy2 = np.sum(dy2[y-offset:y+1+offset, x-offset:x+1+offset])
            Sxy = np.sum(dxy[y-offset:y+1+offset, x-offset:x+1+offset])
            H = np.array([[Sx2, Sxy], [Sxy, Sy2]])
            det = np.linalg.det(H)
            tr = np.matrix.trace(H)
            R = det-k*(tr**2)
            matrix_R[y-offset, x-offset] = R

    cv2.normalize(matrix_R, matrix_R, 0, 1, cv2.NORM_MINMAX)
    for y in range(offset, height-offset):
        for x in range(offset, width-offset):
            value = matrix_R[y, x]
            if value > threshold:
                cv2.circle(img, (x, y), 1, (0, 0, 255))
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_harris_corner_detection_thr_{threshold}_img.png'
    cv2.imwrite(img_path, img)
    return img_path
```

Description

We first specify **window_size**, and the constant **k** and the **threshold**, then we read the image input with **cv2.imread**, then convert it to gray scale and apply gaussian filter on the image with **cv2.GaussinaBlur**, then calculate the image derivatives dx and dy with applying Sobel edge detection with **cv2.Sobel**, after that we calculate the product and second derivatives $(dx)^2$, $(dy)^2$ and dxy with **np.square**, then we calculate the sum of two products derivatives for each pixel by iterate over each pixel and define the matrix $H(x,y)=\begin{bmatrix} Sx2 & Sxy \\ Sxy & Sy2 \end{bmatrix}$ with **np.array**, then calculate the response function $R= det(H) - k\,(Trace(H))^2$ with **np.linalg.det** to get determent of matrix and **np.matrix.trace** to get trace of matrix. After that, we apply the threshold, if the value is greater than threshold, a circle will be drawn in around that pixel containing this value with **cv2.circle**, after that we save the image with **cv2.imwrite** and return the image path

Results

Harris corner detection with k 0.04

Input

Output

Object Detection

active_contour function

Functions Used

```python
def active_contour(image_path, alpha, beta, iterations, w_line, w_edge, radius):
    neighbors = np.array([[i, j] for i in range(-1, 2) for j in range(-1, 2)])
    def find_center():
        im = Image.open(image_path)
        immat = im.load()
        (X, Y) = im.size
        m = np.zeros((X, Y))
        for x in range(X):
            for y in range(Y):
                m[x, y] = immat[(x, y)] != (255, 255, 255)
        m = m / np.sum(np.sum(m))
        dx = np.sum(m, 1)
        dy = np.sum(m, 0)

        cx = np.sum(dx * np.arange(X))
        cy = np.sum(dy * np.arange(Y))

        return cx, cy
    def display(image, changedPoint=None, snake=None):
        if snake is not None:
            for s in snake:
                if (changedPoint is not None and (s[0] == changedPoint[0] and s[1] == changedPoint[1])):
                    plt.plot(s[0], s[1], '.r-', markersize=5.0)

                else:
                    plb.plot(s[0], s[1], 'g.', markersize=5.0)
        plt.imshow(image, cmap=cm.Greys_r)
    def imageGradient(gradient, snake):
        sum = 0
        snaxels_Len = len(snake)
        for index in range(snaxels_Len-1):
            point = snake[index]
            sum = sum+((gradient[point[1]][point[0]]))
        return sum
    def point_inside(img, point):
        return np.all(point < np.shape(img)) and np.all(point > 0)
    def img_gradient(img):
        gauss = cv2.GaussianBlur(img, (21, 21), 0)
        ix = cv2.Sobel(gauss, cv2.CV_64F, 1, 0, ksize=21)
        iy = cv2.Sobel(gauss, cv2.CV_64F, 0, 1, ksize=21)
        sobel = np.sqrt(np.square(ix) + np.square(iy))
        return sobel
    def total_energy(gradient, image, snake):
        inter_Energy = internal_energy(snake)
        ext_Energy = external_energy(gradient, image, snake)
        tEnergy = inter_Energy+ext_Energy
        return tEnergy
    def internal_energy(snake):
        inter_Energy = 0
        snakeLength = len(snake)
        for index in range(snakeLength-1, -1, -1):
            nextPoint = (index+1) % snakeLength
            currentPoint = index % snakeLength
            previousePoint = (index - 1) % snakeLength
            inter_Energy = inter_Energy + (alpha * (np.linalg.norm(snake[nextPoint] - snake[currentPoint])**2))\
                + (beta * (np.linalg.norm(snake[nextPoint] - 2 *
                   snake[currentPoint] + snake[previousePoint])**2))
        return inter_Energy
    def external_energy(gradient, image, snake):
        sum = 0
        snaxels_Len = len(snake)
        for index in range(snaxels_Len - 1):
            point = snake[index]
            sum = +(image[point[1]][point[0]])
            pixel = 255 * sum
        ext_Energy = w_line*pixel - w_edge*imageGradient(gradient, snake)
        return ext_Energy
    def draw_circle(center, radius, num_points):
        points = np.zeros((num_points, 2), dtype=np.int32)
        for i in range(num_points):
            theta = float(i)/num_points * (2 * np.pi)
            x = center[0] + radius * np.cos(theta)
            y = center[1] + radius * np.sin(theta)
            p = [x, y]
            points[i] = p
        return points

```

Code Cont.

```
    img = cv2.imread(image_path, 0)
    img_grad = img_gradient(img)
    cx, cy = find_center()
    snake = draw_circle((cx, cy), radius, 40)
    snake_copy = copy.deepcopy(snake)
    for i in range(iterations):
        for index, point in enumerate(snake):
            min_energy = float("inf")
            for index2, move in enumerate(neighbors):
                next_point = (point + move)
                if not point_inside(img, next_point):
                    continue
                if not point_inside( img, point):
                        continue
                snake_copy[index] = next_point
                totalEnergyNext = total_energy(img_grad, img, snake_copy)
                if (totalEnergyNext < min_energy):
                    min_energy = copy.deepcopy(totalEnergyNext)
                    indexOFlessEnergy = copy.deepcopy(index2)
            snake[index] = (snake[index]+neighbors[indexOFlessEnergy])
    display(img, None, snake)
    img_path = f'./static/download/ac_hf/{randint(0,9999999999999999)}'
    plt.savefig(os.path.splitext(img_path)[0] + "active.png")
    return (img_path+"active.png")
```

Description

The active contour or snake is used for object tracking, shape recognition and edge detection. The function takes seven parameters which are image path, alpha, beta, iteration, wline, wedge and radius.

The Algorithm steps

- Read the image a turn I to gray scale.
- We pass the image to img_gradient to make gaussian blurring and Sobel edge detection in ix and iy and return the Sobel after square them.
- We call find center function to find the center of gravity of the object and return the center in x and y axis.
- We call draw circle and pass to it the center of the object in x and y axis, the radius of the circle and the number of points wanted in the contour.
- The draw circle makes array of zeros and make a for loop to find a center for every point, save it in the array and return the points.
- We make a for loop for the snake points and make in its variable act as unbounded upper value for find the min energy.
- The neighbor variable makes a window for every point to find passed on the window the min energy in it.
- The loop of neighbor finds if it in the image or not.
- The total energy function return the total energy and it is the summation of the internal and external energy so it calls the two functions. The internal energy function takes the snake parameter because the internal energy related to the snake itself (smoothness and elasticity) it makes for loop to find the equation $E_{internal}= \sum_{0}^{n-1} \alpha(\text{nextpoint-currentpoint})^2+\beta(\text{nextpoint-2*currentpoint+previouspoint})^2$ the function returns the internal energy. The external energy related to power in the object itself the function take the gradient; image and the snake points it the equation that we try to achieve is $E_{external}=w_{line}*E_{line}+w_{edge}*E_{edge}$ but in the code we subtract the Eedge because we want it be small so we can find the min energy then we return the external energy.
- We compare the total energy with the min energy variable that we made, and we make the snake move to the min energy points.
- Finally, we call display image to draw the points of the snake .and save the new image in path and return the path.

Results

Applying active contour model with β=0.3, α=20 and radius of 350px with 500 iterations.

Input

Output

Line Detection

This process consists of three steps which are:

houghLine function

```python
def houghLine(img, peaksnum):
    # reding the photo and applying canny edge detection on it
    image = cv2.imread(img)
    grey = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(grey, (5, 5), 0)
    edge = cv2.Canny(blur, 50, 150)
    # Get image dimensions
    # y for rows and x for columns
    Ny = edge.shape[0]
    Nx = edge.shape[1]

    # Max diatance is diagonal one
    Maxdist = int(np.round(np.sqrt(Nx**2 + Ny ** 2)))
    # Theta in range from -90 to 90 degrees
    thetas = np.deg2rad(np.arange(-90, 90))
    # Range of radius
    rhos = np.linspace(-Maxdist, Maxdist, 2*Maxdist)
    accumulator = np.zeros((2 * Maxdist, len(thetas)))
    for y in range(Ny):
        for x in range(Nx):
            # Check if it is an edge pixel
            #  NB: y -> rows , x -> columns
            if edge[y, x] > 0:
                # Map edge pixel to hough space
                for k in range(len(thetas)):
                    # Calculate space parameter
                    r = x*np.cos(thetas[k]) + y * np.sin(thetas[k])
                    # Update the accumulator
                    # N.B: r has value -max to max
                    # map r to its idx 0 : 2*max
                    accumulator[int(r) + Maxdist, k] += 1
    # getting the indicies of peaks to draw it on the photo
    indicies, acci = hough_peaks(accumulator, peaksnum)
    print('done')
    img_path = hough_lines_draw(image, indicies, rhos, thetas)
    return img_path

```

Description

This function is considered the core of applying line detection. At first, we make the image to grey scale color, apply blurring, apply canny edge detection to get all the edges in our photo. Then, we apply hough transform analysis:

- Getting the edge photo dimensions.
- Getting the polar variables of the image: theta & rho.

- Defining the accumulator that is used to detect the existence of a particular line.
- Then, we do for loop all over the pixels of the image to apply line equation and check if there is a line and updating the accumulator values.
- Then, we call the hough peaks function.

hough_peaks function

```python
def hough_peaks(H, num_peaks, nhood_size=3):
    # loop through number of peaks to identify
    indices = []
    H1 = np.copy(H)
    for i in range(num_peaks):
        idx = np.argmax(H1)  # find argmax in flattened array
        H1_idx = np.unravel_index(idx, H1.shape)  # remap to shape of H
        indices.append(H1_idx)

        # surpass indices in neighborhood
        idx_y, idx_x = H1_idx  # first separate x, y indexes from argmax(H)
        # if idx_x is too close to the edges choose appropriate values
        if (idx_x - (nhood_size / 2)) < 0:
            min_x = 0
        else:
            min_x = idx_x - (nhood_size / 2)
        if (idx_x + (nhood_size / 2) + 1) > H.shape[1]:
            max_x = H.shape[1]
        else:
            max_x = idx_x + (nhood_size / 2) + 1

        # if idx_y is too close to the edges choose appropriate values
        if (idx_y - (nhood_size / 2)) < 0:
            min_y = 0
        else:
            min_y = idx_y - (nhood_size / 2)
        if (idx_y + (nhood_size / 2) + 1) > H.shape[0]:
            max_y = H.shape[0]
        else:
            max_y = idx_y + (nhood_size / 2) + 1

        # bound each index by the neighborhood size and set all values to 0
        for x in range(int(min_x), int(max_x)):
            for y in range(int(min_y), int(max_y)):
                # remove neighborhoods in H1
                H1[y, x] = 0

                # highlight peaks in original H
                if x == min_x or x == (max_x - 1):
                    H[y, x] = 255
                if y == min_y or y == (max_y - 1):
                    H[y, x] = 255

    # return the indices and the original Hough space with selected points
    return indices, H

```

Description

The hough (line) transform's accumulator contains positions where values are at their highest ("peaks," according to the hough_peaks function). Its columns are angle coordinates and its rows are distance coordinates (rho) of lines in a picture (theta). So, this function gets the indices of these peaks that are repeated mostly and means that there is a line. Then return these indices to draw them on the lines of the image.

hough_line_draw function

```python
def hough_lines_draw(img, indices, rhos, thetas):
    rho = []
    for i in range(len(indices)):
        # reverse engineer lines from rhos and thetas
        # if(indices[i][0]<783):
        rho = rhos[int(indices[i][0])]
        # if(indices[i][1]<180):
        theta = thetas[int(indices[i][1])]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        # these are then scaled so that the lines go off the edges of the image
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * (a))
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * (a))

        cv2.line(img, (x1, y1), (x2, y2), (0, 255, 0), 2)

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

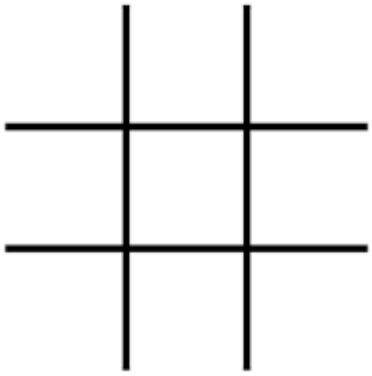
Description

This function is used for applying this hough transform on our photo by getting the edges indices of lines and looping on it and then transforming the polar coordinates into normal variables that are scaled to apply on image dimensions and then draw the lines we got with different colors to show it to the user.

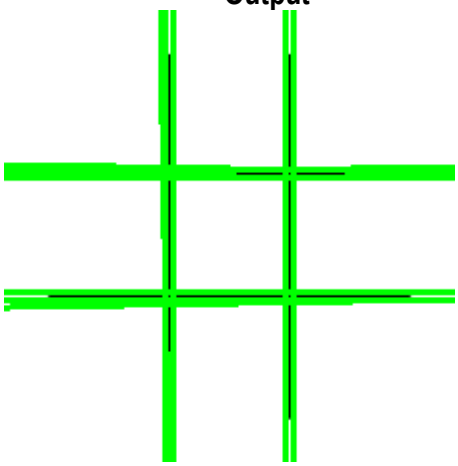
Results

Line detection with threshold of 12

Input



Output



Circle Detection

detectCircles function & **displayCircles** function

```python
def detectCircles(input_img, threshold, region, radius=None):
    imgread = cv2.imread(input_img)
    img = cv2.cvtColor(imgread, cv2.COLOR_BGR2GRAY)
    img = cv2.GaussianBlur(img, (5, 5), 1.5)
    img = cv2.Canny(img, 100, 200)
    (M, N) = img.shape
    if radius == None:
        R_max = np.max((M, N))
        R_min = 3
    else:
        [R_max, R_min] = radius

    print(radius)
    print(input_img)
    R = R_max - R_min
    # Initializing accumulator array.
    # Accumulator array is a 3 dimensional array with the dimensions representing
    # the radius, X coordinate and Y coordinate resectively.
    # Also appending a padding of 2 times R_max to overcome the problems of overflow
    A = np.zeros((R_max, M+2*R_max, N+2*R_max))
    B = np.zeros((R_max, M+2*R_max, N+2*R_max))

    # Precomputing all angles to increase the speed of the algorithm
    theta = np.arange(0, 360)*np.pi/180
    edges = np.argwhere(img[:, :])  # Extracting all edge coordinates
    for val in range(R):
        r = R_min+val
        # Creating a Circle Blueprint
        bprint = np.zeros((2*(r+1), 2*(r+1)))
        (m, n) = (r+1, r+1)  # Finding out the center of the blueprint
        for angle in theta:
            x = int(np.round(r*np.cos(angle)))
            y = int(np.round(r*np.sin(angle)))
            bprint[m+x, n+y] = 1
        constant = np.argwhere(bprint).shape[0]
        for x, y in edges:  # For each edge coordinates
            # Centering the blueprint circle over the edges
            # and updating the accumulator array
            X = [x-m+R_max, x+m+R_max]  # Computing the extreme X values
            Y = [y-n+R_max, y+n+R_max]  # Computing the extreme Y values
            A[r, X[0]:X[1], Y[0]:Y[1]] += bprint
        A[r][A[r] < threshold*constant/r] = 0

    for r, x, y in np.argwhere(A):
        temp = A[r-region:r+region, x-region:x+region, y-region:y+region]
        try:
            p, a, b = np.unravel_index(np.argmax(temp), temp.shape)
        except:
            continue
        B[r+(p-region), x+(a-region), y+(b-region)] = 1
    print('done')
    img_path = displayCircles(B[:, R_max:-R_max, R_max:-R_max], imgread)
    return img_path


def displayCircles(A, img):
    circleCoordinates = np.argwhere(A)  # Extracting the circle information
    for r, x, y in circleCoordinates:
        cv2.circle(img, (y, x), r, color=(0, 255, 0), thickness=2)
    img_path = f'./static/download/edit/{randint(0,9999999999999999)}_circle_detection.png'
    # img_path = f'./static/download/edit/{randint(0,10000000000000000000)}_circle_detection.png'
    cv2.imwrite(img_path, img)
    return img_path
```

Description

Circle detection function is the same concept as line detection function with just differences in the equation of each one of them as we:

- Getting the edge photo dimensions.
- Getting the polar variables of the image: theta & rho.

- Initializing accumulator array (Accumulator array is a 3-dimensional array with the dimensions representing the radius, X coordinate and Y coordinate respectively).
- Also appending a padding of 2 times R_max to overcome the problems of overflow Then, we do for loop all over the pixels of the image to apply line equation and check if there is a line and updating the accumulator values.
- Then, recomputing all angles to increase the speed of the algorithm.
- Then, Centering the blueprint circle over the edges and updating the accumulator array.
- Finally, we apply the polar coordinates into image circles directions and save the figures.

Results

Circle detection with threshold of 15, maximum radius of 100px and minimum radius of 40px.

Input **Output**

Ellipse Detection

adaptiveThreshold function (useful function)

```python
def adaptiveThreshold(img,  sub_thresh=0.10):
    image = img.copy()
    if image.shape[-1] == 3:
        gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    else:
        gray = image
    integralimage = cv2.integral(gray, cv2.CV_32F)

    width = gray.shape[1]
    height = gray.shape[0]
    win_length = int(width / 10)
    image_thresh = np.zeros((height, width, 1), dtype=np.uint8)
    for j in range(height):
        for i in range(width):
            x1 = i - win_length
            x2 = i + win_length
            y1 = j - win_length
            y2 = j + win_length
            if (x1 < 0):
                x1 = 0
            if (y1 < 0):
                y1 = 0
            if (x2 > width):
                x2 = width - 1
            if (y2 > height):
                y2 = height - 1
            count = (x2 - x1) * (y2 - y1)

            sum = integralimage[y2, x2] - integralimage[y1, x2] - \
                integralimage[y2, x1] + integralimage[y1, x1]
            if (int)(gray[j, i] * count) < (int)(sum * (1.0 - sub_thresh)):
                image_thresh[j, i] = 0
            else:
                image_thresh[j, i] = 255

    return image_thresh
```

Ellipse_detection function

```python
def ellipse_detection(img_path):
    image = cv2.imread(img_path)
    image = cv2.resize(image, (700, 700), cv2.INTER_CUBIC)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    r = (17, 51, 618, 611)

    image = image[int(r[1]):int(r[1]+r[3]), int(r[0]):int(r[0]+r[2])]
    pixel_vals = image.reshape((-1, 3))

    pixel_vals = np.float32(pixel_vals)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.75)

    k = 3
    retval, labels, centers = cv2.kmeans(
        pixel_vals, k, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)

    centers = np.uint8(centers)
    segmented_data = centers[labels.flatten()]

    segmented_image = segmented_data.reshape((image.shape))

    mask = np.zeros(segmented_image.shape[:2], dtype=np.uint8)
    segmented_image = cv2.cvtColor(segmented_image, cv2.COLOR_BGR2GRAY)
    segmented_image = adaptiveThreshold(segmented_image)
    plt.imshow(segmented_image)
    contours, hierarchy = cv2.findContours(
        segmented_image, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    approx = []
    for cnt in contours[1:]:
        epsilon = 0.0001*cv2.arcLength(cnt, True)
        approx.append(cv2.approxPolyDP(cnt, epsilon, True))
        try:
            ellipse = cv2.fitEllipse(cnt)
            (x, y), (MA, ma), angle = ellipse
            MA = max(MA, ma)
            area = cv2.contourArea(cnt)
            equi_diameter = np.sqrt(4*area/np.pi)
            # print(MA/equi_diameter)
            if MA/equi_diameter < 1.5 and MA < max(mask.shape)/1.7:
                img = cv2.ellipse(mask, ellipse, (255, 255, 255), 3)
        except:
            pass
    img_path = f'images/{randint(0,9999999999999999)}_ellipse.png'
    cv2.imwrite(img_path, img)
    return img_path
```

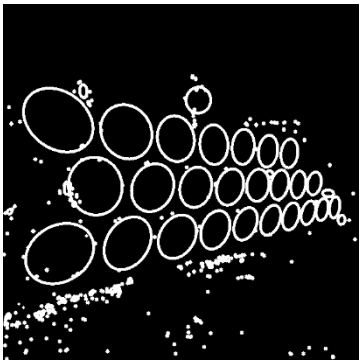
Description

- The function takes the path of the image.
- We read the image then resize it and turn it to gray scale.
- We find the part we want to detect the ellipse on it.
- We segment the image by k-means segmentation and then adaptive thresholding.
- Then we try to find the contour and fit the ellipse by some accepting criterion.
- The criteria are put based on ratio of the major axis to the minor axis.
- The function returns the path of the image.

Results

Ellipse detection with sub threshold of 0.1.

Input **Output**



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.

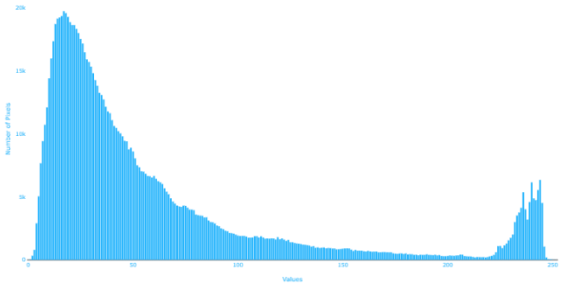
Histograms and Distribution Curves

Histograms

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

Results

Input	Output



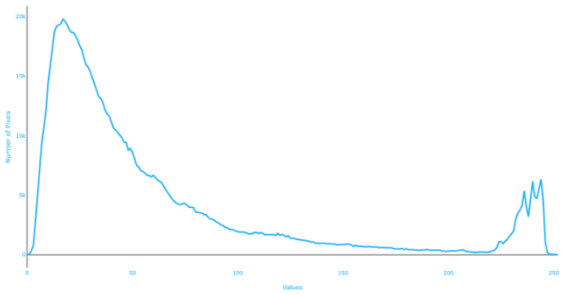
Distribution Curves

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

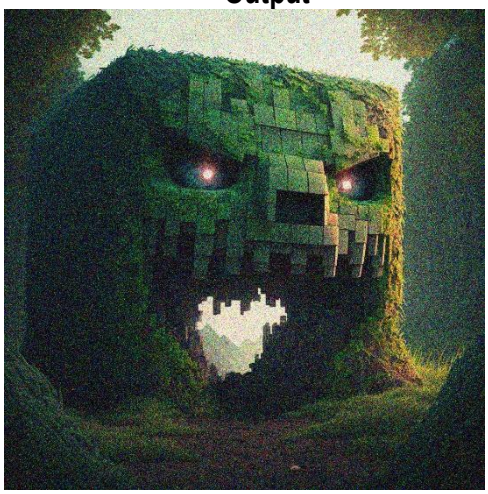
The function takes two parameters the image and the variance value.
First, we read the image and change it to gray scale. Second, we normalize the image to make max value equal to 1 and minimum equal to zero. Third, we take the number of rows and columns. Forth, we set the mean value to zero and calculate 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
def 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

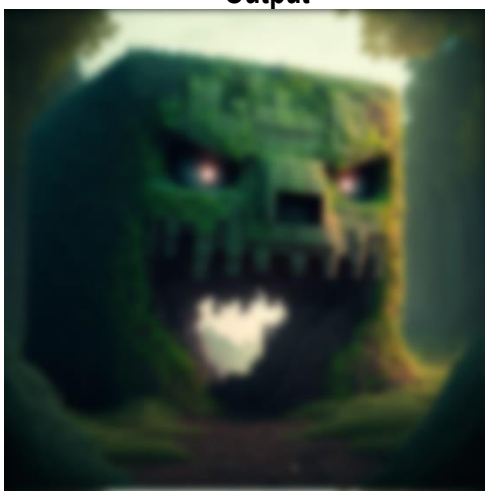
For every function it takes the image.
First, we read the image and try to fix its colors and resize the image. second, we call the Fourier transform function. the Fourier function make a list to the transform channels for red, green and blue channels then make Fourier transform for every channel and 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

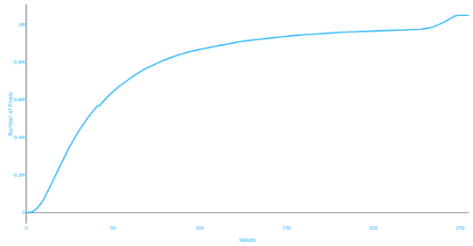
```python
def get_cumulative_curve(img_path):
    list_x = []
    for i in range(256):
        list_x.append(i)
    image = cv2.imread(img_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    flat = image.flatten()
    # array with size of bins, set to zeros
    hist = np.zeros(256)
    # loop 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

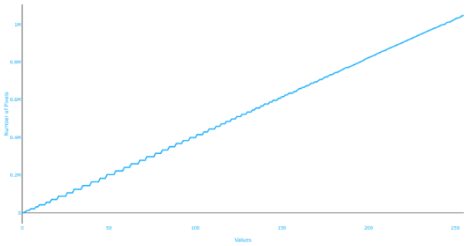
This feature is applied in the equalization and normalization of images. As you can see the curve is applied to detect how the distribution changes. The algorithm of drawing cumulative curve is the same as the algorithm of equalization of the image as it is one of the steps to get equalized image.

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**.

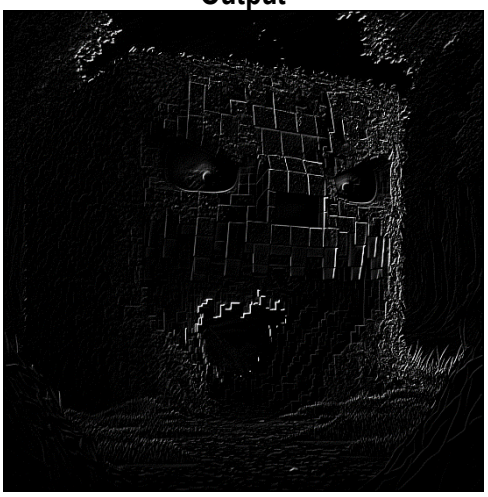
This outputs **prewitt_img_path** refer to the path which the new image has been saved.

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**.

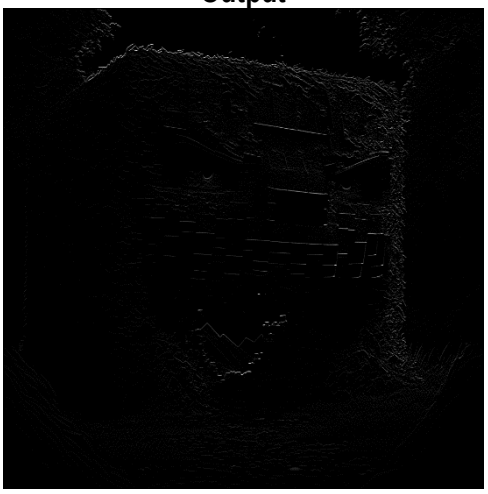
This outputs **roberts_img_path** refer to the path which the new image has been saved.

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**.

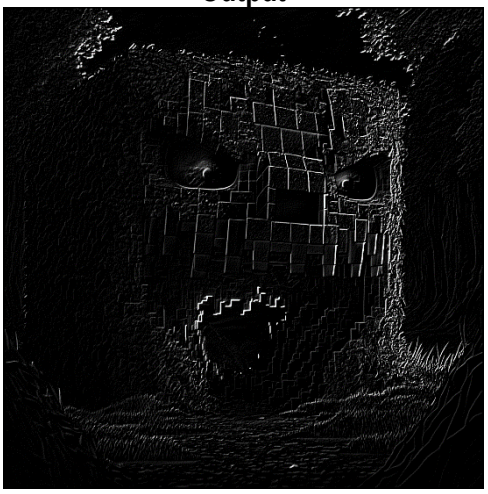
This outputs **sobel_img_path** refer to the path which the new image has been saved.

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