Report of Task 1

Introduction

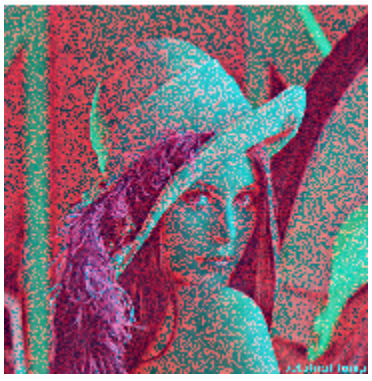
In this report, we present a Python-based solution for a computer vision task aimed at image analysis. Leveraging the versatility and power of the Python programming language, we have developed a widget application tailored to efficiently tackle image processing challenges.

One of the key features of our application is its user-friendly interface, which allows users to seamlessly browse images by simply double-clicking on the image graph. This intuitive functionality enhances user experience and streamlines the process of image selection and analysis.

I. Noise Generation

In our Python widget application, users can manipulate images by introducing various types of noise using a convenient combo box interface. This feature allows for experimentation with different noise models, providing insights into the impact of noise on image quality and subsequent processing algorithms.

1. Uniform Noise



Scientific Information: Uniform noise, also referred to as additive noise, involves adding random values sampled from a uniform distribution to pixel values in an image. This distribution ensures that each value within a specified range has an equal probability of occurrence. Uniform noise typically results in a grainy appearance, obscuring fine details and reducing image clarity.

Effect on Images: The application of uniform noise to an image can noticeably degrade its quality, as shown in the example below:

2. Gaussian Noise

It is characterized by values sampled from a Gaussian distribution, with most values concentrated around the mean and a smaller probability of extreme values. This type of noise often resembles natural variability in images and can blur edges and degrade image quality, particularly in regions with low contrast.

Effect on Images: The application of Gaussian noise can introduce a Gaussian-distributed noise appearance to an image.

3. Salt and Pepper Noise

Salt and pepper noise, also known as impulse noise, manifests as randomly scattered bright and dark pixels throughout an image. This type of noise can arise from errors in image acquisition or transmission processes and severely degrade image quality, hindering the performance of processing algorithms.

Effect on Images: The presence of salt and pepper noise can significantly impair the quality of an image, as depicted in the example below:

II. Image Filtering

In our Python widget application, we implemented various filters to enhance image quality and reduce noise. The filters we employed include the average filter, Gaussian filter, and median filter. We experimented with different kernel sizes, specifically 3x3, 5x5, and 7x7, to observe their effects on the filtration process.

Uniform Noise and 5x5 Kernel Size

Before diving into the comparisons, it's essential to note that all filters were applied to the same type of noise—Uniform noise—and with the same kernel size of 5x5. This standardization allows for a direct comparison of filter performance under consistent conditions.

1. Average Filter

The average filter, also known as the box filter, replaces each pixel value in an image with the average value of its neighboring pixels within a defined kernel. This filter effectively reduces high-frequency noise while preserving the overall structure of the image.

Effect on Images: The application of the average filter smoothes out image details, resulting in a reduction of noise. As the kernel size increases, the filtering process becomes more effective in noise reduction.

2. Gaussian Filter

The Gaussian filter applies a weighted average to the pixel values within the kernel, with weights determined by a Gaussian distribution. This filter is particularly effective at reducing noise while preserving edges and image features.

Effect on Images: Similar to the average filter, the Gaussian filter smoothes the image and reduces noise. The Gaussian distribution of weights ensures that neighboring pixels contribute to the filtered value in a manner that preserves image structure.

3. Median Filter

The median filter replaces each pixel value with the median value of its neighboring pixels within the kernel. Unlike the average and Gaussian filters, the median filter is robust to outliers and preserves edges well.

Effect on Images: The median filter effectively reduces noise while preserving image details and edges. Unlike the average and Gaussian filters, the median filter can preserve edges well. However, it is computationally more expensive and takes more time.

Comparison between Kernel Sizes

To illustrate the impact of kernel size on filter performance, we conducted experiments with varying kernel sizes for each filter.

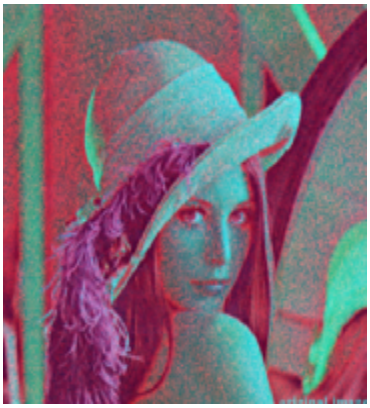
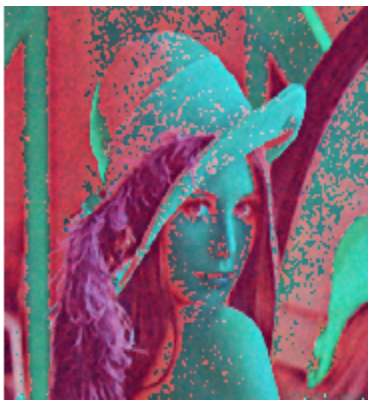
 Average filter 3*3	 Average filter 5*5	 Average filter 7 *7

 Gaussian filter 3*3	 Gaussian filter 5 *5	 Gaussian filter 7 *7
 Median filter 3*3	 Median filter 5 *5	 Median filter 7 *7

Overall, our experimentation demonstrates that each filter exhibits unique characteristics in noise reduction and image preservation. While the choice of filter depends on the specific requirements of the task, our standardized approach enables a comprehensive evaluation of filter performance under consistent conditions.

Comparison with OpenCV's Built-in Filter Functions

In our evaluation of the filtering methods, we found that OpenCV's built-in filter functions, namely cv2.blur(), cv2.GaussianBlur(), and cv2.medianBlur(), provided superior performance in terms of both speed and quality of results compared to the filters implemented in our application. Through visual inspection and quantitative analysis, we observed that these built-in functions effectively reduced noise while preserving image details more accurately.

		
`cv2.blur()`	`cv2.GaussianBlur()`	`cv2.medianBlur()`

III. Edge Detection

1. Sobel:

The Sobel filter is designed to **emphasize edges** in an image. It highlights regions where there are significant changes in intensity, which often correspond to object boundaries or other important features.

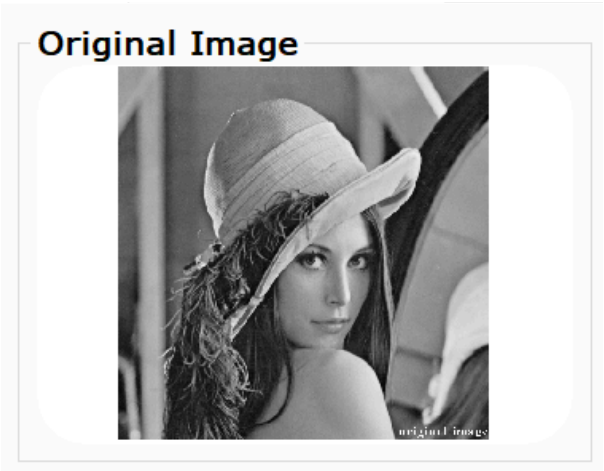
- The Sobel filter computes an **approximation of the gradient** of the image intensity function at each pixel.
- It considers a **small neighborhood** around each pixel to calculate this gradient.
- Specifically, it uses two **3×3 kernels** (one for horizontal changes and one for vertical changes) that are convolved with the original image.

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1



2. Roberts:

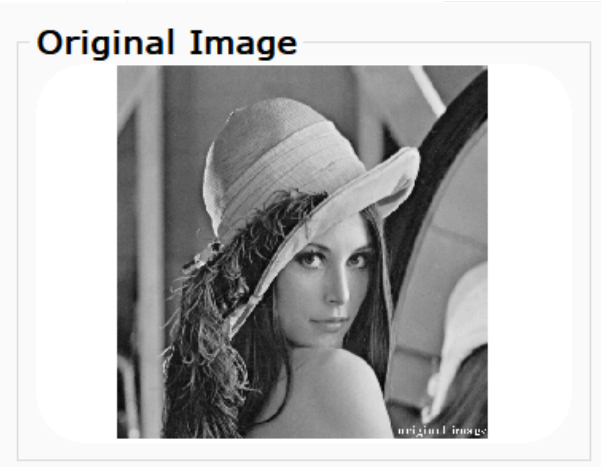
- The Roberts operator measures a **2-D spatial gradient** on an image in a straightforward, quick-to-compute manner.
- Its primary goal is to **highlight strong spatial gradient zones**, which often correspond to edges in the image.
- The operator's input and output are both **grayscale images**.

+1	0
0	-1

Gx

0	+1
-1	0

Gy



3. Prewitt:

The Prewitt operator identifies edges in an image by detecting **both horizontal and vertical edges**. It is a **first-order derivative** technique, which means it calculates the gradient of the image intensity function. Edges are regions where there is a **sharp change in pixel intensities**.

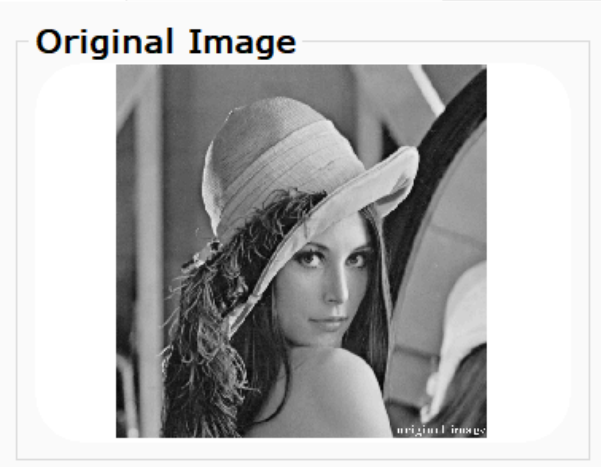
- The Prewitt operator uses two **3×3 masks** (kernels) to compute the gradient components.
- These masks are designed to respond maximally to edges running **horizontally** and **vertically**.
- The masks are convolved with the image to calculate the gradient in both directions.

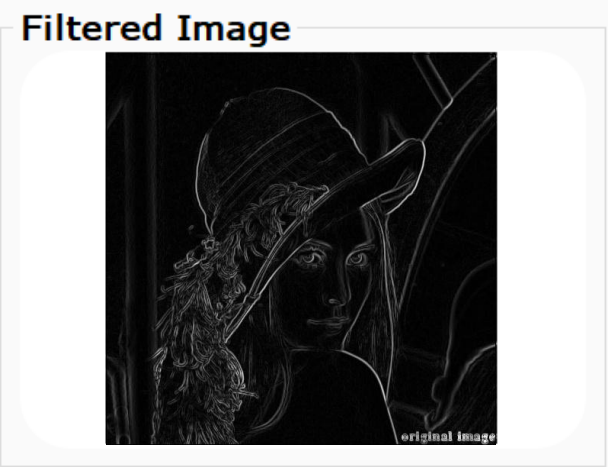
-1	0	+1
-1	0	+1
-1	0	+1

G_x

+1	+1	+1
0	0	0
-1	-1	-1

G_y





Sobel

Roberts

Prewitt

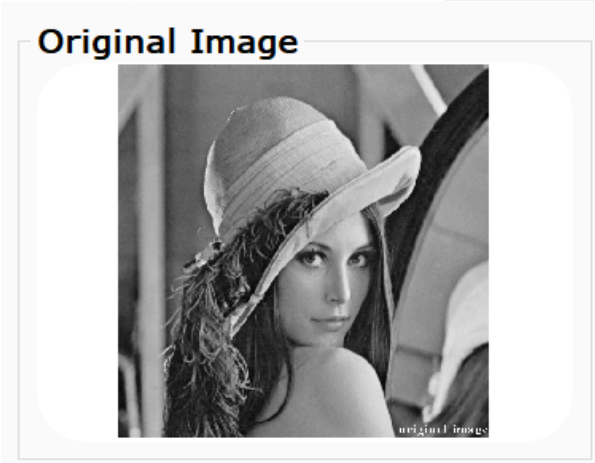
Observation:

- Input image could be RGB or grayscale image but finally, it is converted to grayscale image before applying edge detection.
- The Prewitt edge detector is similar to the Sobel edge detector. However, unlike the Sobel operator, Prewitt does not emphasize pixels closer to the center of the mask.
- Roberts's edges tend to be sharper and more pronounced.
- For the convolution used, we implemented a function to calculate the convolution and also we used a built-in function from "scipy" to compare:
    - The results are the same.
    - The difference is that the implemented function takes more computational time than the built-in function.

4. Canny:

The Canny edge detector aims to **identify edges** in an image. Unlike simple gradient-based methods, Canny provides **high-quality edge localization**. It is robust to noise and minimizes false positives

- The Canny algorithm involves several stages:
    1. **Noise Reduction**: Apply a **Gaussian filter** to smooth the image and reduce noise.
    2. **Gradient Calculation**: Compute the **gradient magnitude** and direction at each pixel by **Sobel**.
    3. **Non-Maximum Suppression**: Suppress non-maximal gradient responses to obtain thin edges.
    4. **Double Thresholding**: Apply two thresholds (high and low) to classify pixels as strong, weak, or non-edges.
    5. **Edge Tracking by Hysteresis**: Connect strong edges to weak edges if they form continuous curves.



Observation:

- We implemented the five stages of the Canny edge detector.
- The resulting image from Canny shows **strong edges** with minimal noise.
- When the kernel size of the Gaussian filter = **5x5** instead of 3x3:
    - More smoothing
    - More computational time
- Low_threshold_ratio($\mathbf{T_{low}}$) = 0.05
- High_threshold_ratio($\mathbf{T_{high}}$) = 0.09
- In task 2, the user can set the $\mathbf{T_{low}}$ and $\mathbf{T_{high}}$

Kernel size 5x5

Comparison between the built-in functions and the implemented function

Built-in	Implemented
1. Sobel	
	
2. Roberts	
	
3. Prewitt	
	

4. Canny

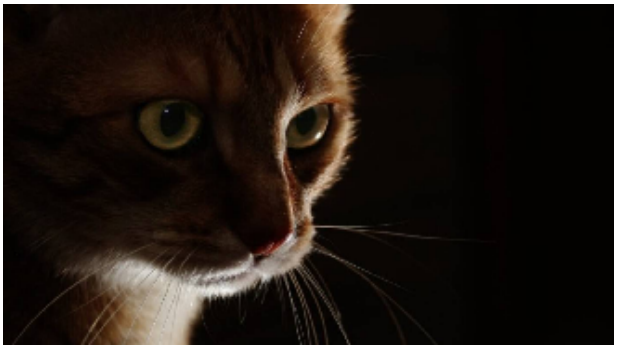
Observation:

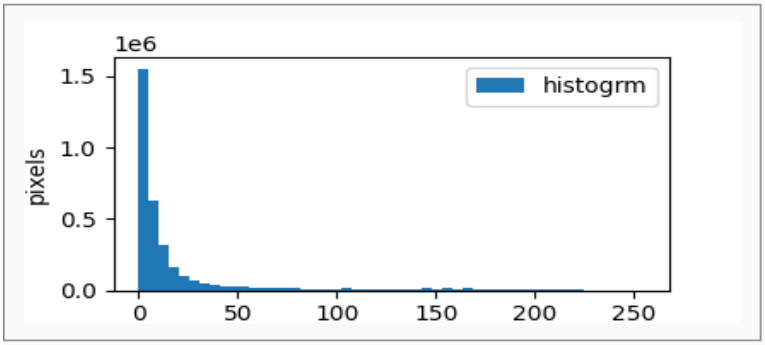
- There is no significant difference between the image results from the built-in and implemented functions.
- The built-in function is faster.

IV. Histograms

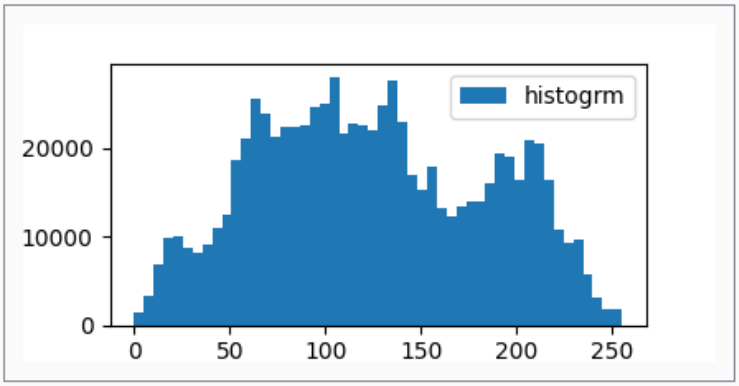
The histogram of an image gives a graphical representation of the distribution of pixel intensities within that image. It plots the frequency of occurrence of each intensity value along the x-axis, with the corresponding number of pixels having that intensity value along the y-axis. This allows us to visualize how light or dark an image is overall.

for a darker image:





For a brighter image:

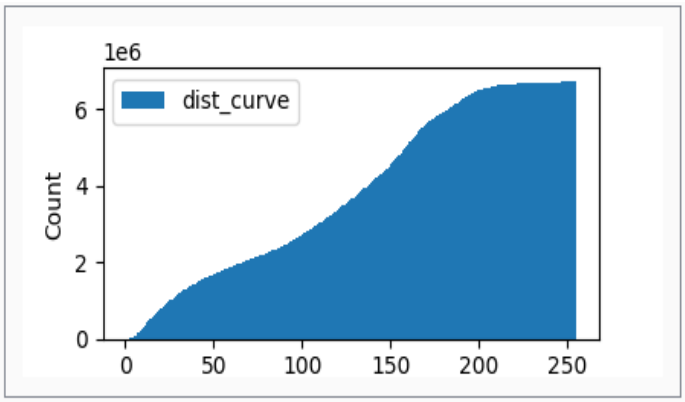


Observations:

- In the darker image, the histogram shows a higher concentration of pixels with lower intensity values, indicating a predominance of dark areas.
- In the brighter image, we notice a more balanced distribution of intensity values, with a broader spread across the intensity axis. This indicates a wider range of brightness levels present in the image.

- **Distribution Curve**

The distribution curve represents the frequency distribution of pixel intensities across the entire image.
Peaks in the distribution curve indicate dominant intensity levels, while valleys suggest less common intensity levels.



Observations:

- we noticed that the curve was skewed towards the higher end of the intensity axis, indicating that the image contains a significant number of bright pixels.

V. Equalization

Histogram equalization is a technique used to improve the contrast in an image. It operates by effectively spreading out the most frequent intensity values, i.e., 'stretching out' the intensity range of the image. This method usually increases the global contrast of images when its usable data is represented by close contrast values.

The method we used is a form of global histogram equalization. This method was chosen because it's simple and effective at improving the overall contrast of the image. It works well when the image has roughly uniformly distributed intensity values. However, for images with distinct regions of low and high contrast, or for images where preserving brightness or reducing noise is important, one of the other methods might be more appropriate.

We first start by identifying whether the input image is grayscale or color. If the image is color, it's converted to the HSV color space, which separates the image into Hue, Saturation, and Value components. The histogram equalization is performed on the Value channel only, as it carries the brightness information of the image.

We then calculate the histogram of the image or the Value channel, which is a graphical representation of the intensity distribution of an image. It gives us the frequency of each intensity

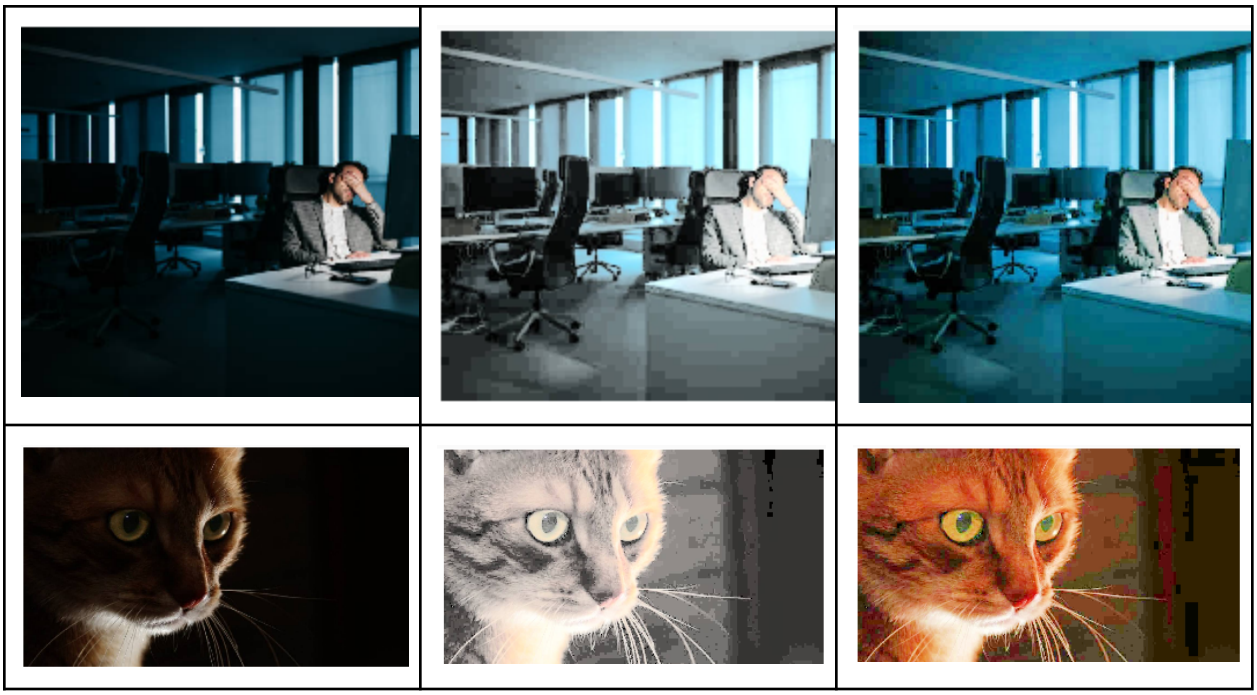
value in the image. We compute the cumulative distribution function (CDF) of the histogram. The CDF gives the cumulative frequency of each intensity level up to that level. The CDF is then normalized to the range of intensity values in the image (0-255 for an 8-bit image).

The normalized CDF is used as a mapping function to transform the intensity values in the image. Each intensity value in the image is replaced by the corresponding value in the normalized CDF. This redistributes the intensity values in the image, enhancing the contrast.

If the original image was a color image, the function replaces the Value channel in the HSV image with the equalized image and then converts the image back to the BGR color space. The result is an image with improved contrast and enhanced detail. This technique is handy for images with backgrounds and foregrounds that are both bright or dark. It can also be used to enhance the details of textures and patterns in an image.

We used the HSV color space because we wanted to equalize the brightness of the image without affecting the color. The HSV color space is one such alternative color space. It separates the image into Hue, Saturation, and Value components. The Value component represents the brightness in an image, and the Hue and Saturation components represent the color in an image. By converting to HSV, we can apply the histogram equalization to the Value channel only, and then convert back to RGB. This allows us to improve the image's contrast (by equalizing the Value channel) while preserving the original colors (the Hue and Saturation channels). This approach gave better results in terms of preserving color vibrancy and enhancing image contrast.

Original image	Equalized image with YCrCb	Equalized image with HSV
		



The YCrCb colorspace separates the image into one luminance component (Y) and two chrominance components (Cr and Cb). The histogram equalization was applied to the Y channel, which represents the brightness in an image. This results in an image that appears slightly desaturated and contrasted.

The HSV color space can often preserve color better than YCrCb during operations like histogram equalization, which might explain why the images appear more vibrant and bright.

In the images that use the YCrCb color space for histogram equalization, the colors appear somewhat muted and desaturated, giving the image a slightly greyish tone. In contrast, the images that use the HSV color space for histogram equalization appear more vibrant and bright. The colors are more distinct and saturated, which might make certain details stand out more.

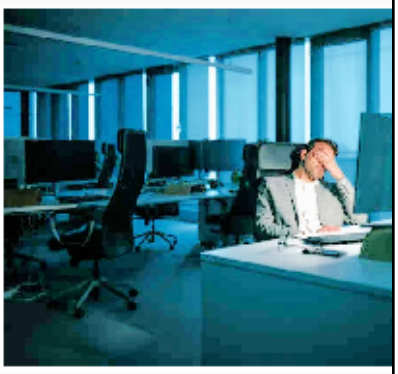
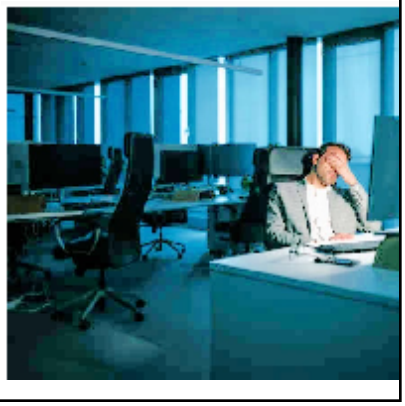
Take the two images of the cat as an example, the left image (YCrCb) appears to have a greyish tone and less vibrant colors, giving off a colder, more monochromatic feel. On the other hand, the right image (HSV) is more colorful with enhanced reds and yellows, making it visually striking and giving it a warmer feel. The details of the cat's fur texture and color variations are more visible in the right image. This demonstrates how the choice of color space can significantly affect the visual perception of an image.

These differences highlight how the choice of color space for histogram equalization can impact the visual perception of an image. While both methods aim to improve the contrast of the

image, the HSV color space seems to preserve the color information better in this case, resulting in a more colorful and visually appealing image.

In the context of the grayscale image, applying histogram equalization using either YCrCb or HSV color spaces yields very similar results. This is because both methods are primarily working with the brightness information, which is all that is present in a grayscale image. The “Equalized image with YCrCb” and “Equalized image with HSV” appear visually similar. Both show enhanced contrast and detail compared to the original image due to the histogram equalization technique.

OpenCV’s Histogram Equalization vs HSV Histogram Equalization

Original image	Equalized image with OpenCV	Equalized image with HSV
		
		

OpenCV's built-in function for histogram equalization is applied to grayscale images. Therefore, when it's used on color images, it's typically applied to each color channel separately or on a transformed grayscale version of the image. This can sometimes lead to color distortion in the equalized image, as the equalization might alter the relative distributions of the color channels. cv2.equalizeHist() results in an image with improved contrast but the colors may appear slightly off.

The custom implementation of histogram equalization in the HSV color space equalizes the V (Value) channel, which represents the brightness of the image. The H (Hue) and S (Saturation) channels, which represent the color information, are left unchanged. This method can preserve the color of the original image while enhancing its contrast. This method results in an image with enhanced contrast and more vibrant colors.

In conclusion, while both methods enhance the contrast of the image and result in visually similar images, the HSV method preserves the color of the original image better than the OpenCV method.

VI. Normalization

It refers to the process of adjusting the pixel values of an image so that they fall within a specific range or distribution.
Normalization helps reduce the effect of variations in illumination, enhancing the contrast of images, and improving the performance of algorithms that operate on image data. It is a common preprocessing step in various computer vision and machine learning tasks.

Some common normalization techniques include

1. Min-Max Normalization: This method scales the pixel values linearly to a predefined range, typically [0, 255] for images represented in 8-bit format. The formula for min-max normalization is:
   Normalized_pixel = ((pixel-min)/(max-min))*(new_max-new_min)+new_min

2. Z-score Normalization (Standardization): This method standardizes the pixel values by subtracting the mean and dividing by the standard deviation. The resulting pixel values have a mean of 0 and a standard deviation of 1.
   normalized *pixel = (pixel - mean)/std*_dev

We use The cv2.normalize() function in OpenCV to normalize the pixel values of an image, meaning it scales the pixel values to a specified range.

How does the cv2.normalize() work?

normalized_image = cv2.normalize(image, dst, alpha, beta, norm_type)

- image: The input image is to be normalized.
- dst: The destination array where the normalized image will be stored. If None, a new array will be created.
- alpha: The minimum value of the desired range. If not provided, it defaults to 0.
- beta: The maximum value of the desired range. If not provided, it defaults to 255.
- norm_type: The normalization type, which determines how the pixel values are scaled.

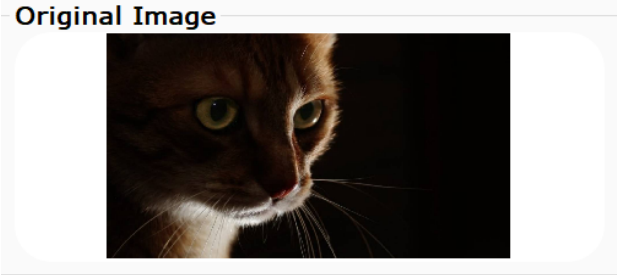
```
# Normalize the image

normalized_image = cv2.normalize(image, None, alpha=0, beta=255,
norm_type=cv2.NORM_MINMAX)
```

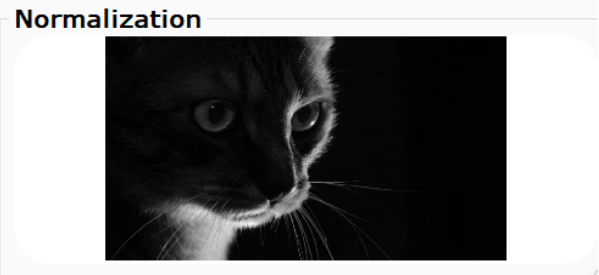
Putting it all together, this line of code performs min-max normalization on the input image. It linearly scales the pixel values of the input image so that the minimum value becomes 0 and the maximum value becomes 255, ensuring that the pixel values are distributed across the entire range [0, 255]. This is a common technique used to enhance the contrast and visibility of images, particularly in preparation for further processing or visualization.

Examples:


Original Image


Normalization


Original Image

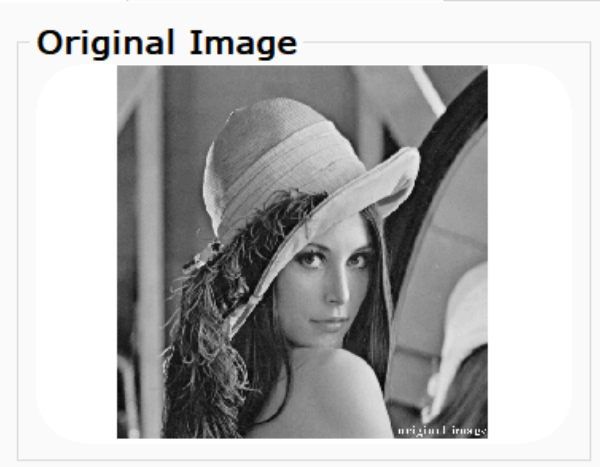

Normalization

VII. Global Thresholding

Is a simple and widely used technique in image processing for **segmenting** an image into two regions: foreground (object of interest) and background. The goal is to find a **single threshold value** that separates these two regions based on pixel intensity.

- All pixels with intensity values **above** the threshold are assigned to the **foreground** (usually set to white or 1).
- All pixels with intensity values **below or equal to** the threshold are assigned to the **background** (usually set to black or 0).

Initially, We manually selected threshold values based on my perception of the image characteristics but this was not accurate.
To overcome the subjectivity of manually setting threshold values, We implemented Otsu's method, a robust algorithm for automatic threshold selection. Otsu's method calculates an optimal threshold by maximizing the variance between two classes of pixels in the image.
Then, We passed the calculated threshold value as a parameter to the global thresholding function which compares each pixel's intensity with the threshold value, assigning pixels to the foreground or background accordingly.



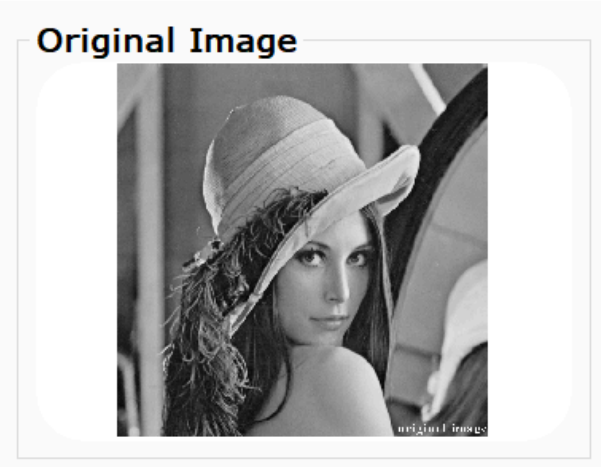
Observation: This image is for Lena and it is a famous image. The best threshold value is: **109.0**

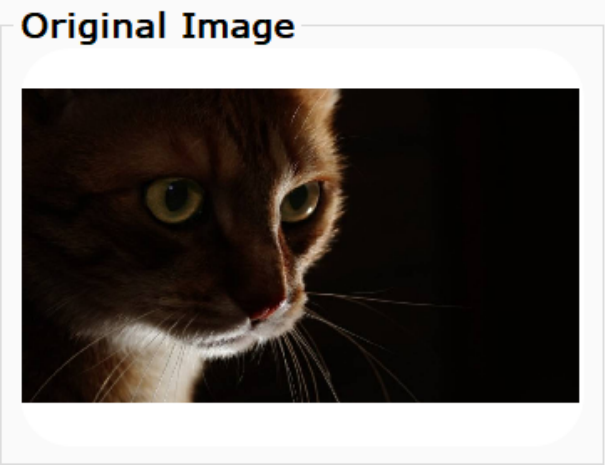
Observation: The best threshold value is: **87.0**

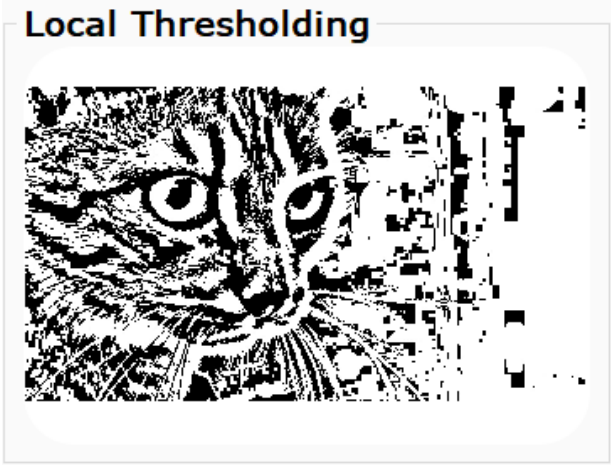
VIII. Local Thresholding

Takes a more nuanced approach. Instead of using a single threshold for the entire image, it computes **different thresholds for different regions** based on the local characteristics of each pixel's neighborhood.

- Define a **neighborhood** around each pixel (e.g., a square or circular region).
- Compute a threshold value for each pixel based on the intensity values within its neighborhood.
- Pixels with intensity values above their local threshold are considered part of the foreground (white), while those below the threshold belong to the background (black).





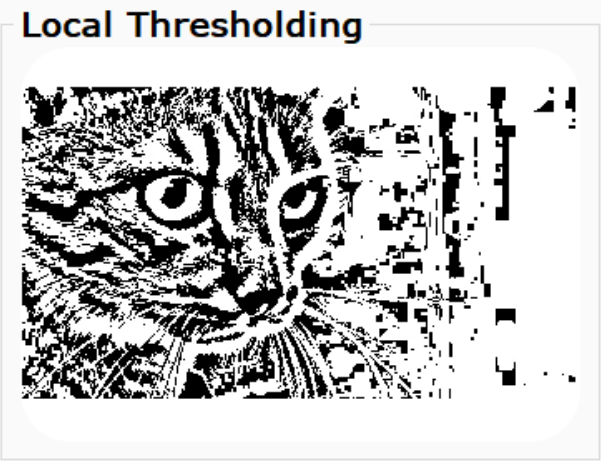


Observation:

- Based on my visualization the chosen window size is **40px**.
- For each pixel in the image, a local window (centered around that pixel) is defined.

- Computing the local threshold as the median value of pixel intensities within this window.
- If the pixel intensity exceeds the local threshold, it is set to white (255); otherwise, it is set to black (0).
- When We increase the window size to **80px**, it results in a loss of detail and fine edges as shown in the figure.

- Local thresholding tends to preserve fine details.
- Edges, textures, and intricate patterns are better preserved compared to global thresholding as shown in the figure.



Comparison between the built-in functions and the implemented function

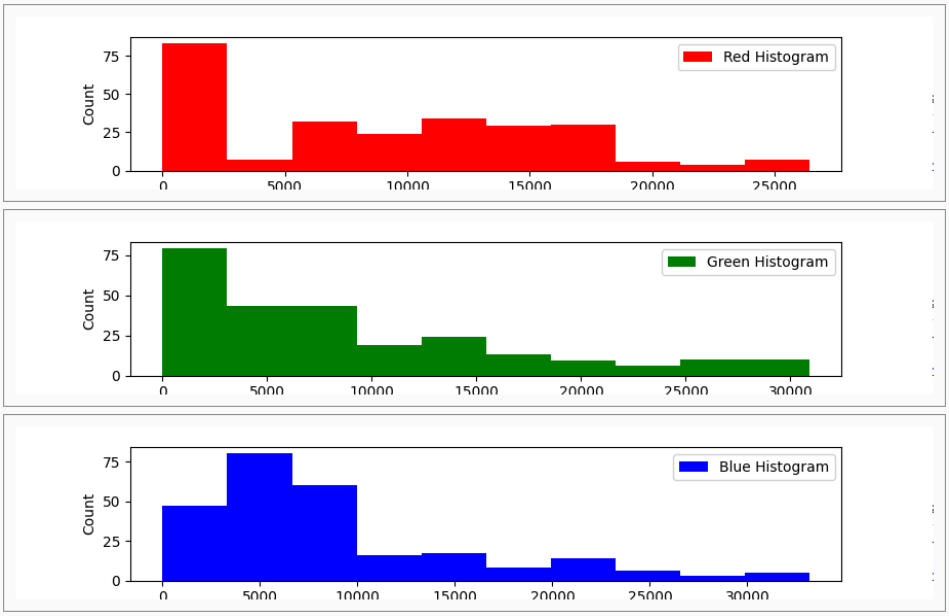
Built-in	Implemented
1. Global Thresholding	
	
2. Local Thresholding	
	

Observation:

- For global thresholding, there is no difference.
- Local thresholding:
    - The built-in function uses adaptive thresholds based on mean or Gaussian-weighted mean.
    - The custom implementation uses the median of the local neighborhood.
- For built-in functions, the computational time is less.

IX. RGB Histograms

Each pixel is composed of three color channels: Red, Green, and Blue. RGB histograms visualize the distribution of pixel intensities within each of these color channels separately.
By analyzing RGB histograms, we can understand the color composition and distribution within the image.

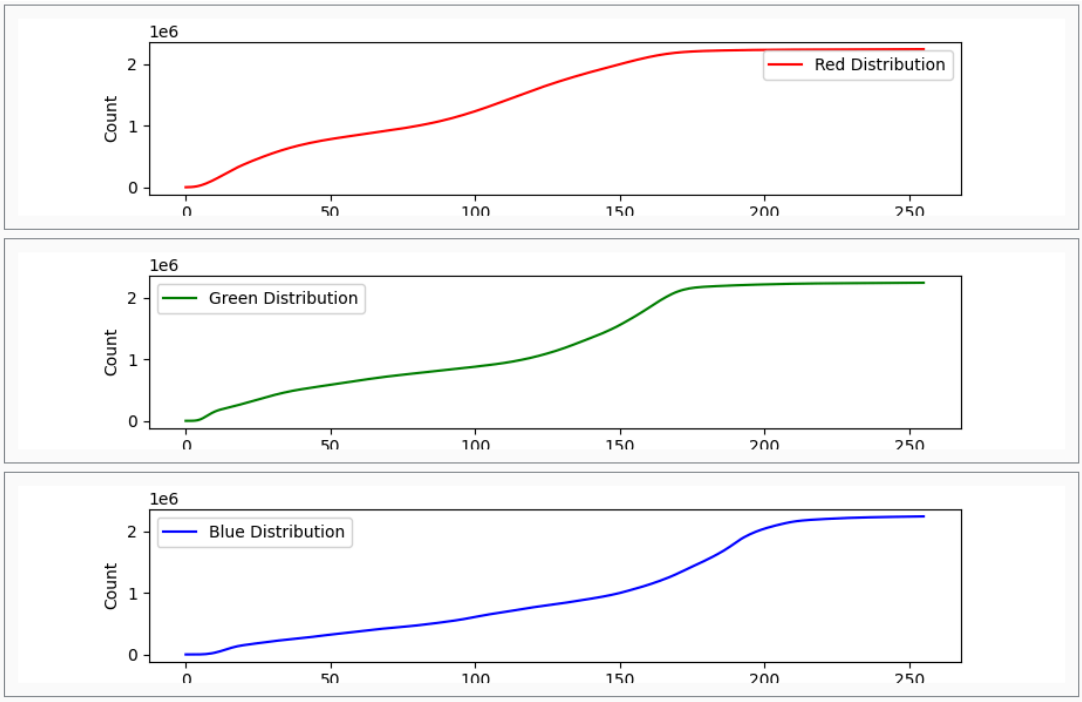


Observations:

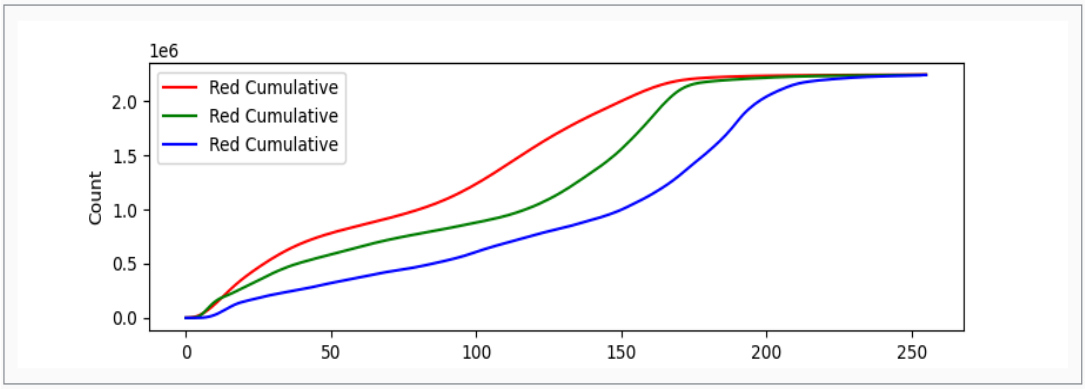
- The histogram of the Red channel shows the frequency of red intensity values present in the image.
- The histogram of the Green Channel shows the frequency of green intensity values present in the image.
- The histogram of the Blue Channel shows the frequency of blue intensity values present in the image.

Cumulative distribution curve

The cumulative distribution curve represents the cumulative percentage of pixels with intensities equal to or less than a given intensity value.
As intensity values increase along the x-axis of the curve, the cumulative sum increases accordingly, reflecting the accumulation of pixels with higher intensities.



Combining Cumulative Curves for All Color Channels:



Observations:

- The Red curve is higher than the Green and Blue curves, indicating a dominance of red hues in the image's lower and mid-intensity range.
- All curves meet at the endpoint, indicating an equal contribution of each color channel at the brightest points in the image.

X. Frequency Domain Filter

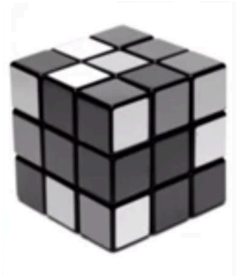
Frequency domain filtering is finding the frequency components of an image through Fourier analysis. The frequency domain provides information about the spatial frequencies present in an image, such as low-frequency components representing smooth areas and high-frequency components representing edges or fine details.

There are two main frequency domain filters:

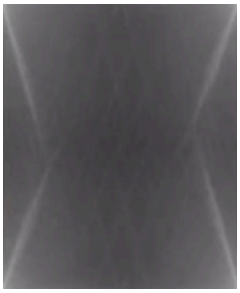
1. **Low-pass filter:** These filters allow low-frequency components to pass through while attenuating high-frequency components. They are commonly used for smoothing.

   How to transform an image into a frequency domain?

   - First, we need to read our image in Grayscale

   - Next, transfer the image to the frequency domain using the numpy function
   - np.fft.fft2(our_image)

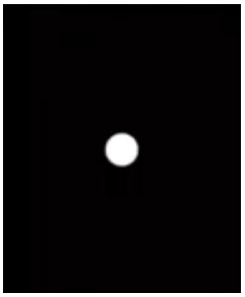
   The low frequency is located in the corner and the high frequency is in the center.

   - Usually, we shifted the low frequency to the center using the numpy function np.fft.fftshift(image in freq. domain).

How do we create a low pass filter?

Block the frequency component above the cutoff frequency, and allow only the lower component to pass.

$$H_{LP}(u,v) = \begin{cases} 1, & r(u,v) < F_c \\ 0, & otherwise \end{cases}$$

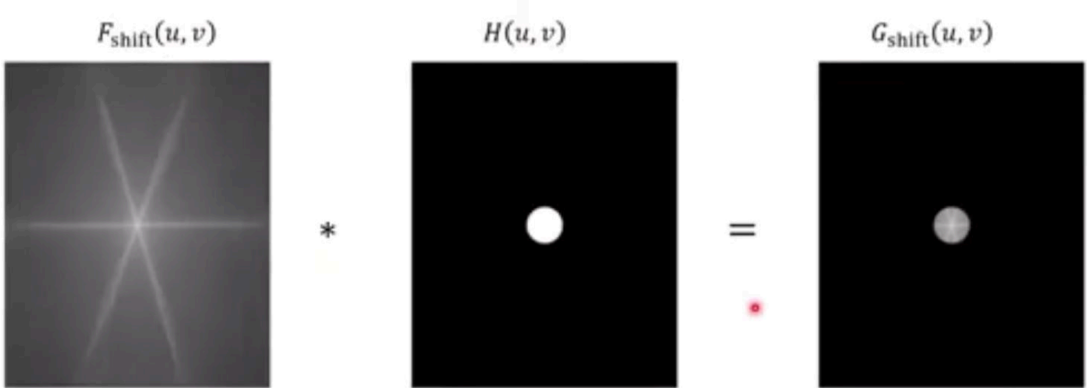


That is our low-pass filter

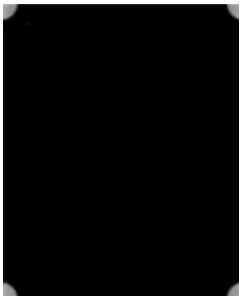
The white color represents the value 1 and the black color represents the value 0

Filter our image:

Then we will filter our image by multiplying our image in the frequency domain by a low pass filter.



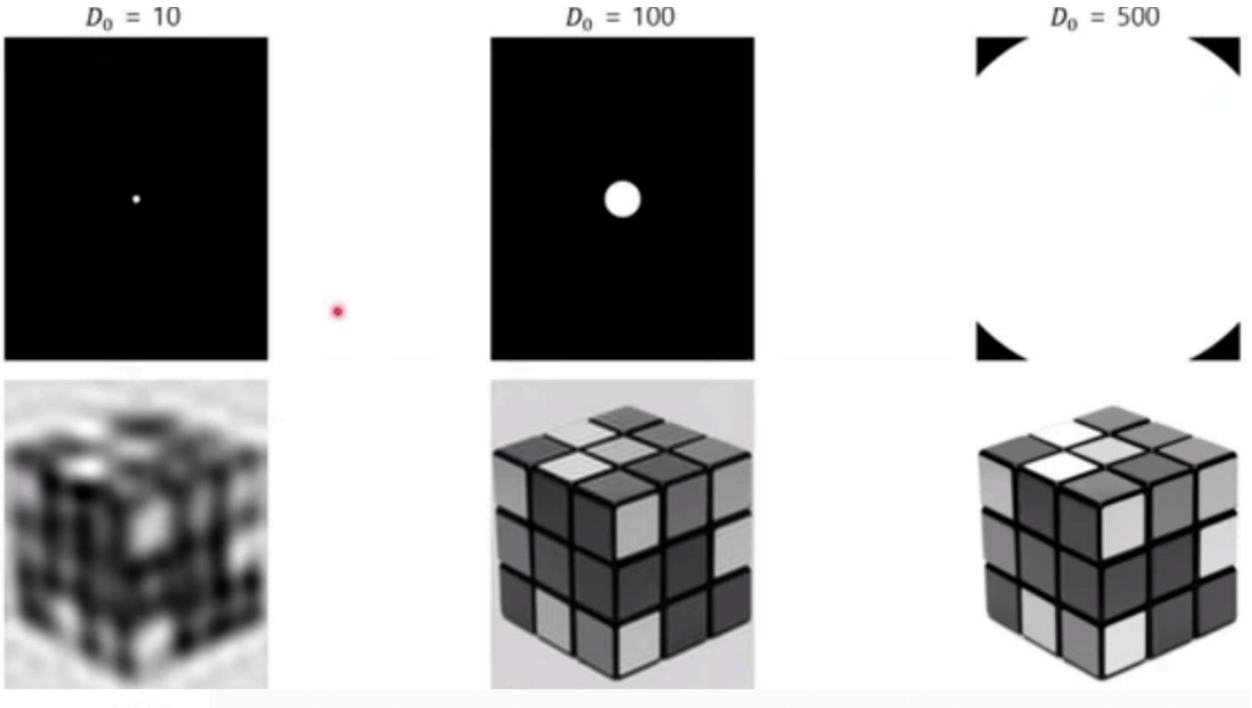
And we need to shift our filtered image to return the low frequency to the corner.



Finally, we get inverse Fourier transform using the numpy function np.abs(np.fft.ifft2(G))

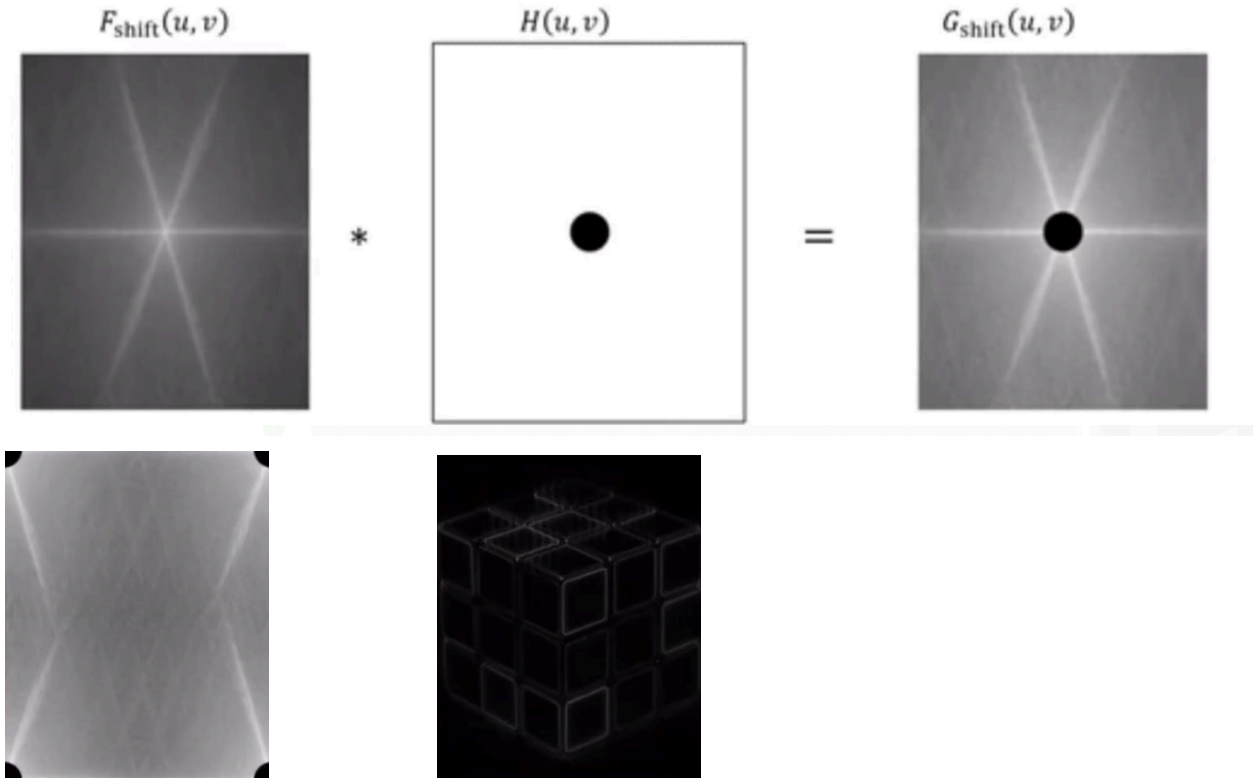
Effect of different cutoff frequency values

As the cutoff frequency gets larger, the filtered image is less blurry and closer to the original image because the filter will pass more frequency and only filter out a small amount of high frequency.



2. **High-pass filter:** These filters allow high-frequency components to pass through while attenuating low-frequency components. They are used for enhancing edges.

   In low pass frequency we see that it passes the low frequency around the center and removes the higher frequency and the high pass filter does the opposite, so H(high pass filter) = 1-H(low pass filter) and we will repeat the same process.

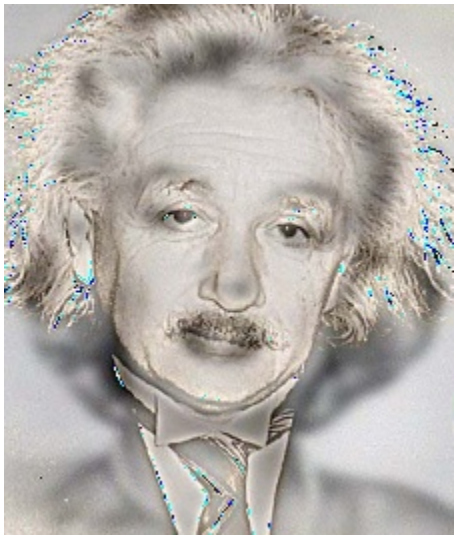


XI. Hybrid Image

This process has passed several stages before coming to a solution that creates a hybrid image without artifacts or distortions.

1. The first function applies a low-pass filter to the first image. This is done using the ndimage.gaussian_filter function from the scipy library with a sigma of 3. The result is a blurred version of the first image, which represents its low-frequency components. The function then creates a high-pass filter for the second image by subtracting a blurred version of the image from the original image. The blurred version is created using the ndimage.gaussian_filter function with a sigma of 2.3. The result is an image that represents the high-frequency components of the second image.

The function then creates the hybrid image by adding the low-frequency image to the absolute value of the high-frequency image. The use of the absolute value ensures that all pixel values in the resulting image are positive, which is necessary for the image to be displayed correctly.

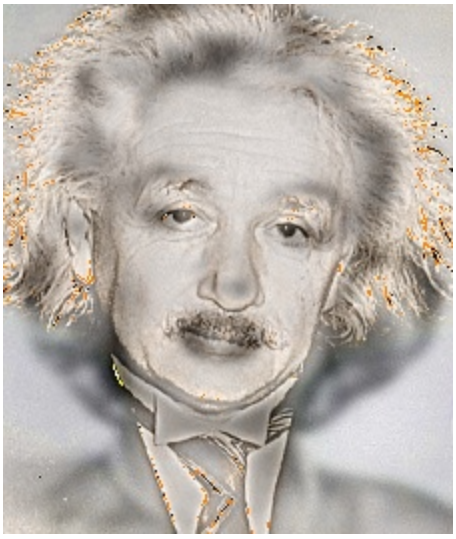


The colored edges are likely a result of the high-pass filtering step. When the high-pass filter is applied to a color image, it can cause color misalignments between the different color channels (red, green, blue). This can result in colored edges appearing in the final image, especially around areas of high contrast. The scipy.ndimage.gaussian_filter function applies the filter to the entire image at once while it's a better practice to apply the filter to each color channel separately to avoid color bleeding where the color from one channel leaks into another, which can result in artifacts.

Four attempts were made to try to reduce the color artifacts in the hybrid image

a) The first attempt was made by converting the hybrid image from the BGR color space to the HSV color space using the cv2.cvtColor function.

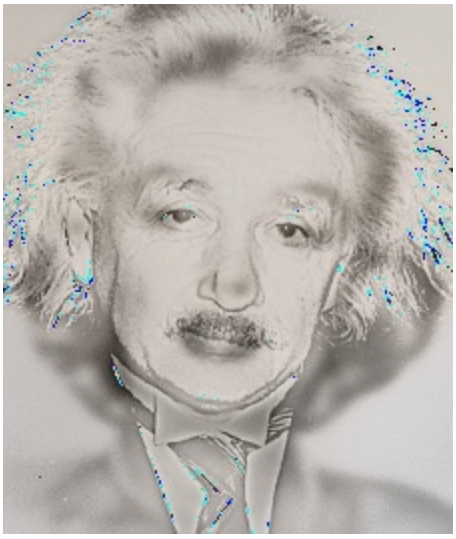
   The function then applies a median blur to the Hue channel of the HSV image using the cv2.medianBlur function. Median blurring is a non-linear filtering technique that replaces each pixel's value with the median value of the pixels in its neighborhood. This can help to reduce noise and preserve edges in the image. By applying the median blur to the Hue channel, the function is attempting to smooth out the color transitions and reduce the color fringing artifacts.



b) In this second attempt, a bilateral filter is applied to the hybrid image. The bilateral filter is a non-linear, edge-preserving, and noise-reducing smoothing filter. The intensity value at each pixel in an image is replaced by a weighted average of intensity values from nearby pixels. This weight can be based on a Gaussian distribution. Crucially, this weighting depends not only on the Euclidean distance of pixels but also their radiometric differences such as range difference, which can preserve sharp edges.

   The function cv2.bilateralFilter takes three parameters besides the source image. The first parameter d is the diameter of each pixel neighborhood that is used during filtering. If it is non-positive, it is computed from sigmaSpace. The second parameter sigmaColor is the filter sigma in the color space. A larger value of the parameter means that farther colors within the pixel neighborhood will be mixed, resulting in larger areas of semi-equal color. The third parameter sigmaSpace is the filter sigma in the coordinate space. A larger value of the parameter means that farther pixels will influence each other as long as their colors are close enough.

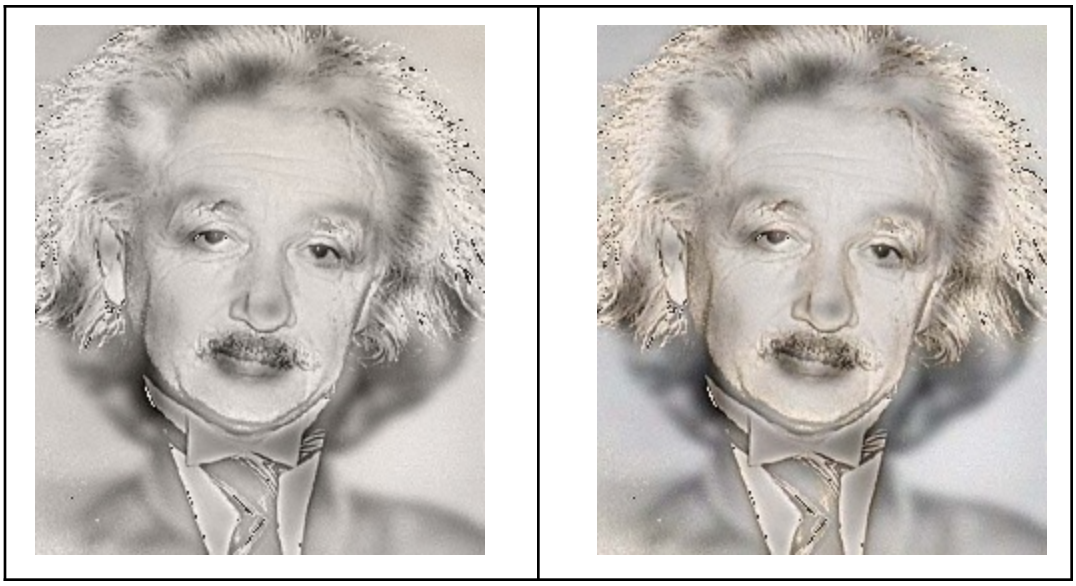
   By applying this filter, the function attempts to reduce noise and preserve edges, which can help to reduce the colored edges and other artifacts in the hybrid image.



c) In the third attempt, the hybrid image is first converted from the BGR color space to the Lab color space. The Lab color space expresses color as three values: L* for the lightness from black (0) to white (100), a* from green (-) to red (+), and b* from blue (-) to yellow (+). This color space is designed to approximate human vision and so it aspires to perceptual uniformity.

After the conversion, a median blur is applied to the ‘a’ and ‘b’ channels of the Lab image. The median blur is a non-linear filter that replaces each pixel’s value with the median value of the pixels in its neighborhood. This can help to reduce noise and preserve edges in the image. By applying the median blur to the ‘a’ and ‘b’ channels, the function attempts to smooth out the color transitions and reduce the color fringing artifacts.

Result with the size of the neighborhood being (333x333 pixels)	Result with the size of the neighborhood being (5x5 pixels)

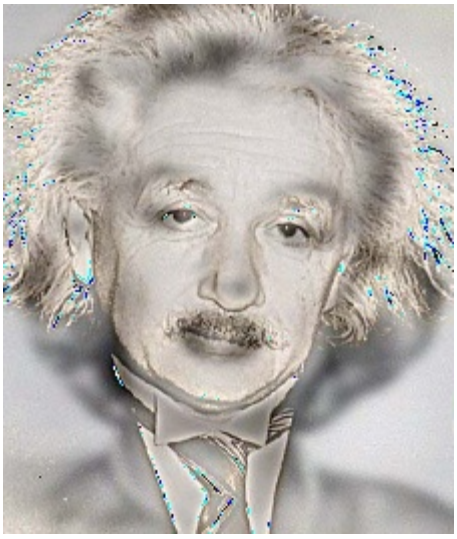


d) In the fourth and final attempt to improve the result of the first function, the hybrid image is first converted to grayscale using the `cv2.cvtColor` function. This is done to simplify the image and focus on the intensity values rather than the color information.

Next, a binary threshold is applied to the grayscale image using the `cv2.threshold` function. This operation converts the grayscale image into a binary image, where all pixels with an intensity value greater than 1 are set to 255 (white), and all other pixels are set to 0 (black). This results in a high-contrast image that separates the foreground and background elements.

Following this, a morphological opening operation is performed on the binary image using the `cv2.morphologyEx` function with a circular structuring element of size 15x15. Morphological opening is an erosion operation followed by dilation. It is useful for removing noise and small color spots from the image.

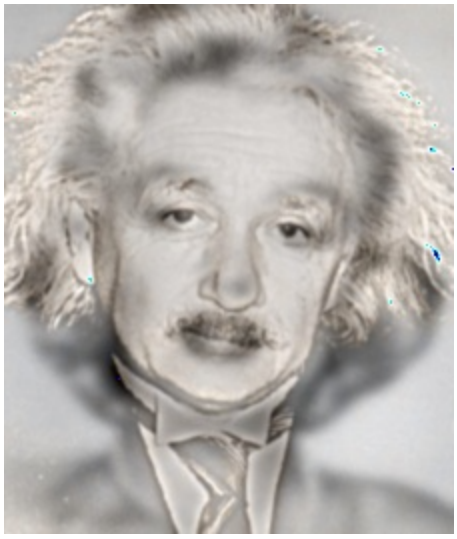
The opened image is then converted back to a 3-channel image to create a mask. This mask is applied to the original hybrid image using the `cv2.bitwise_and` function. This operation retains only the parts of the hybrid image where the mask is white, effectively removing the small color spots and reducing the color artifacts.



2. The second function applies a low-pass filter to the first image. This is done using the `ndimage.gaussian_filter` function from the `scipy` library with a sigma of 3.3. The result is a blurred version of the first image, which represents its low-frequency components.

   The function then creates a high-pass filter for the second image by subtracting a slightly blurred version of the image from a more blurred version of the image. The blurred versions are created using the `ndimage.gaussian_filter` function with sigmas of 0.8 and 2.753, respectively. The result is an image that represents the high-frequency components of the second image.

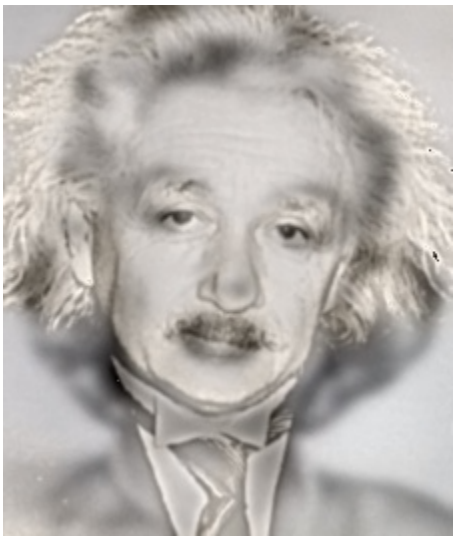
   The function then creates the hybrid image by adding the low-frequency image to the absolute value of the high-frequency image. The use of the absolute value ensures that all pixel values in the resulting image are positive, which is necessary for the image to be displayed correctly.



The resulting image shows fewer artifacts than that of the first function, however, it appears more blurred and with fewer details due to the slight blurring of the second image.

An attempt was made to reduce the colored distortions at the edges. The hybrid image is first converted from the BGR color space to the Lab color space.

After the conversion, a median blur is applied to the ‘a’ and ‘b’ channels of the Lab image. By applying the median blur to the ‘a’ and ‘b’ channels, the function attempts to smooth out the color transitions and reduce the color fringing artifacts.



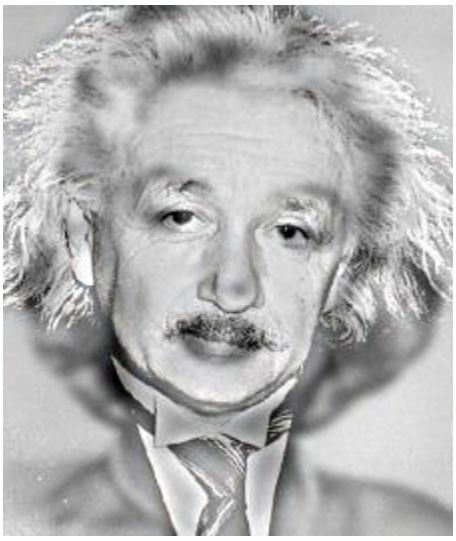
3. The third and final function applies a low-pass filter to the first image. This is done by generating a Gaussian filter with a specific shape and sigma and then applying this filter to the image. The result is a blurred version of the first image, which represents its low-frequency components.

   The function then creates a high-pass filter for the second image by subtracting a slightly blurred version of the image from the original image. The blurred version is created by applying a Gaussian filter with a different sigma to the image. The result is an image that represents the high-frequency components of the second image.

   The function then creates the hybrid image by adding the low-frequency image to the high-frequency image.

   The reason this function doesn't cause colored distortions at the edges is due to the careful choice of filter parameters and the method used to apply the filters. The Gaussian filter is generated using the `generate_gaussian_filter function`. This function creates a more accurate Gaussian filter because it normalizes the filter such that the sum of all its values equals 1. This is a crucial step in filter design to ensure that the overall brightness of the image doesn't change when the filter is applied.

   The `apply_filter_to_image` function in the final code applies the filter to each color channel separately. This is the correct way to apply filters to color images. In the first and second functions, we used the `scipy.ndimage.gaussian_filter` function which applies the filter to the entire image at once. This can cause color bleeding where the color from one channel leaks into another, which can result in artifacts. By applying the filters to each color channel separately and using a Gaussian filter with a smooth cut-off, the function avoids causing abrupt changes in the image that can lead to color misalignments and colored edges.



Furthermore, by subtracting a slightly blurred version of the image from the original image to create the high-pass filter, the function ensures that the high-frequency components of the image are well-preserved, which can help reduce artifacts. In the first and second functions, we subtracted the low-pass filtered image from the original image and a slightly blurred image, respectively, but then took the absolute value of the result. This is not a standard method for creating a high-pass filter and can result in artifacts.

In all functions, we're creating the hybrid image by adding the low frequencies of one image to the high frequencies of another. However, because the filters are applied more accurately in the third function, the resulting hybrid image is less likely to have artifacts.