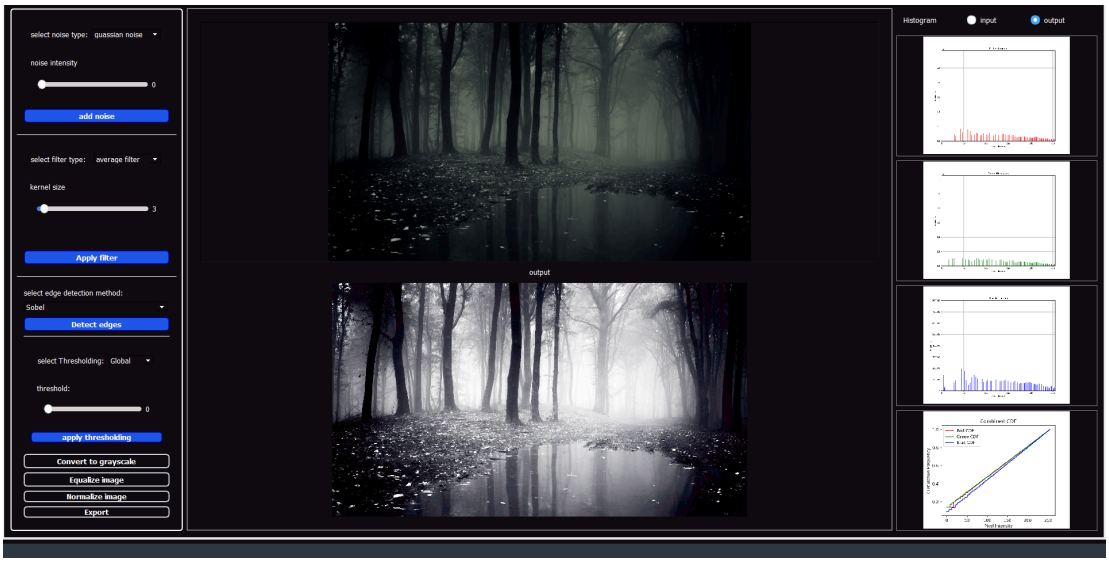
Task 1 - Team 12

Report

Main Interface



Requirement 1: Add additive noise to the image

1. Gaussian Noise:

In [1]:
```python
def add_gaussian_noise(image, mean=0, sigma=25):
    sigma = sigma * 2.55  # Convert to [0, 255] range
    noisy = image + np.random.normal(mean, sigma, image.shape) # guassian = norm
    return np.clip(noisy, 0, 255).astype(np.uint8)
```

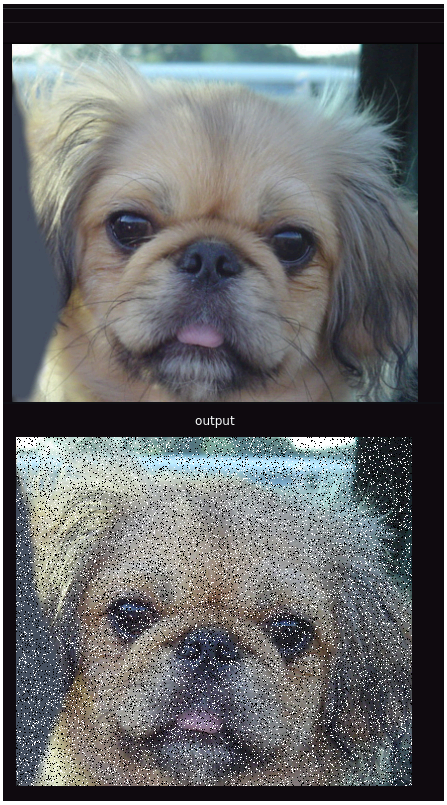
2. Salt & Pepper Noise:

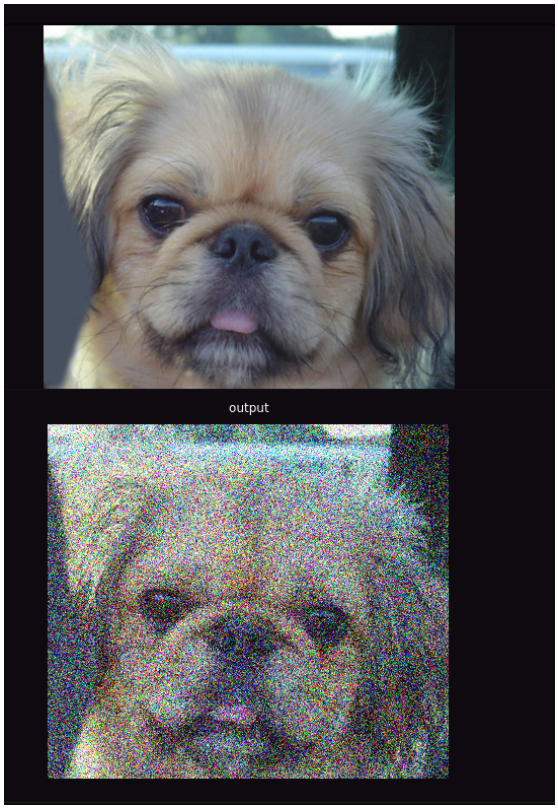
In [2]:
```python
def add_salt_pepper_noise(image, salt_prob=0.02, pepper_prob=0.02):
    noisy = np.copy(image)
    total_pixels = image.size
    num_salt = int(total_pixels * salt_prob)
    num_pepper = int(total_pixels * pepper_prob)

    # Add salt (white) noise
    salt_coords = [np.random.randint(0, i, num_salt) for i in image.shape]
    noisy[salt_coords[0], salt_coords[1]] = 255

    # Add pepper (black) noise
    pepper_coords = [np.random.randint(0, i, num_pepper) for i in image.shape]
    noisy[pepper_coords[0], pepper_coords[1]] = 0

    return noisy
```

3. Uniform Noise

In [3]:
```python
def add_uniform_noise(image, intensity=50):
    intensity = intensity * 2.55  # Convert to [0, 255] range
    noisy = image + np.random.uniform(-intensity, intensity, image.shape)
    return np.clip(noisy, 0, 255).astype(np.uint8)
```

screen shots

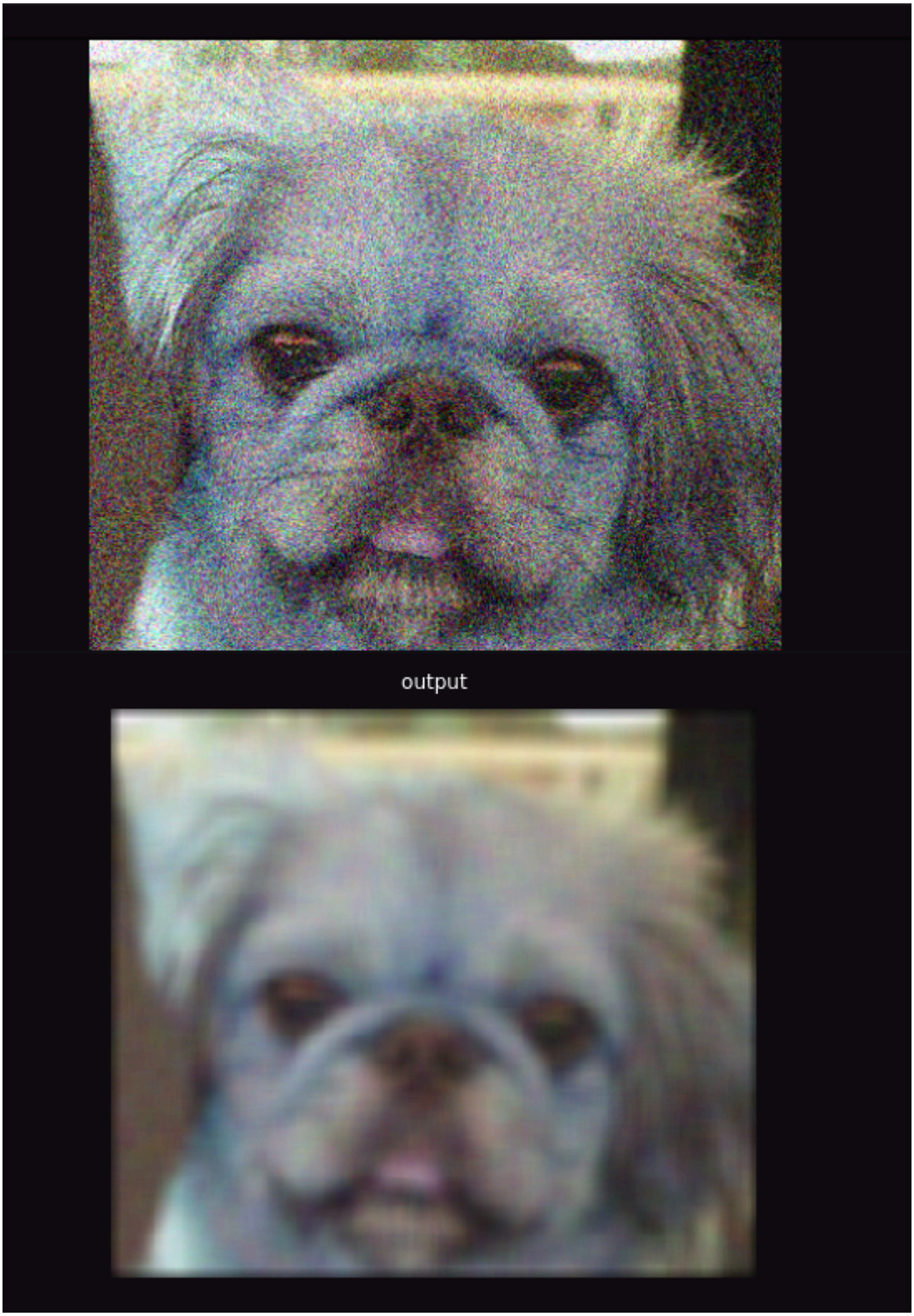




Requirement 2: Filters

1. Average Filter

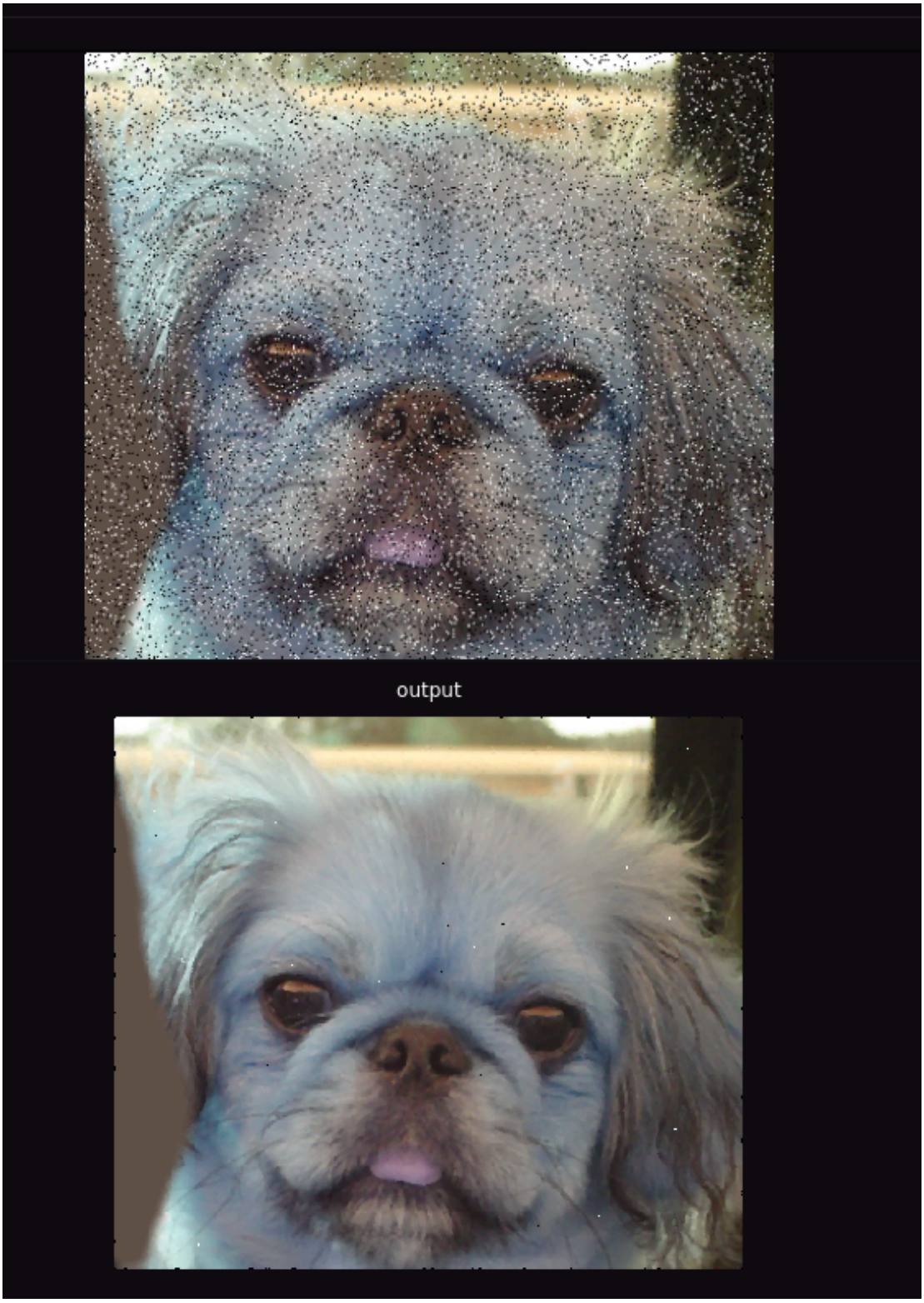
In [4]:
```python
def apply_average_filter(image, kernel_size=30):
    kernel = np.ones((kernel_size, kernel_size)) / (kernel_size ** 2)
    # pad image to ensure that filter is applied to edges
    padded_image = np.pad(image, ((kernel_size//2, kernel_size//2), (kernel_size
                          2, kernel_size//2), (0, 0)), mode='constant', constant
    filtered_image = np.zeros_like(image)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            for k in range(image.shape[2]):
                window = padded_image[i:i+kernel_size, j:j+kernel_size, k]
                filtered_image[i, j, k] = np.sum(window * kernel)

    return np.clip(filtered_image, 0, 255).astype(np.uint8)
```



2. Median Filter

In [5]:
```
def apply_median_filter(image, kernel_size=3):
    padded_image = np.pad(image, ((kernel_size//2, kernel_size//2), (kernel_size
                          2, kernel_size//2), (0, 0)), mode='constant', constant
    filtered_image = np.zeros_like(image)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            for k in range(image.shape[2]):
                region = padded_image[i:i+kernel_size, j:j+kernel_size, k]
```

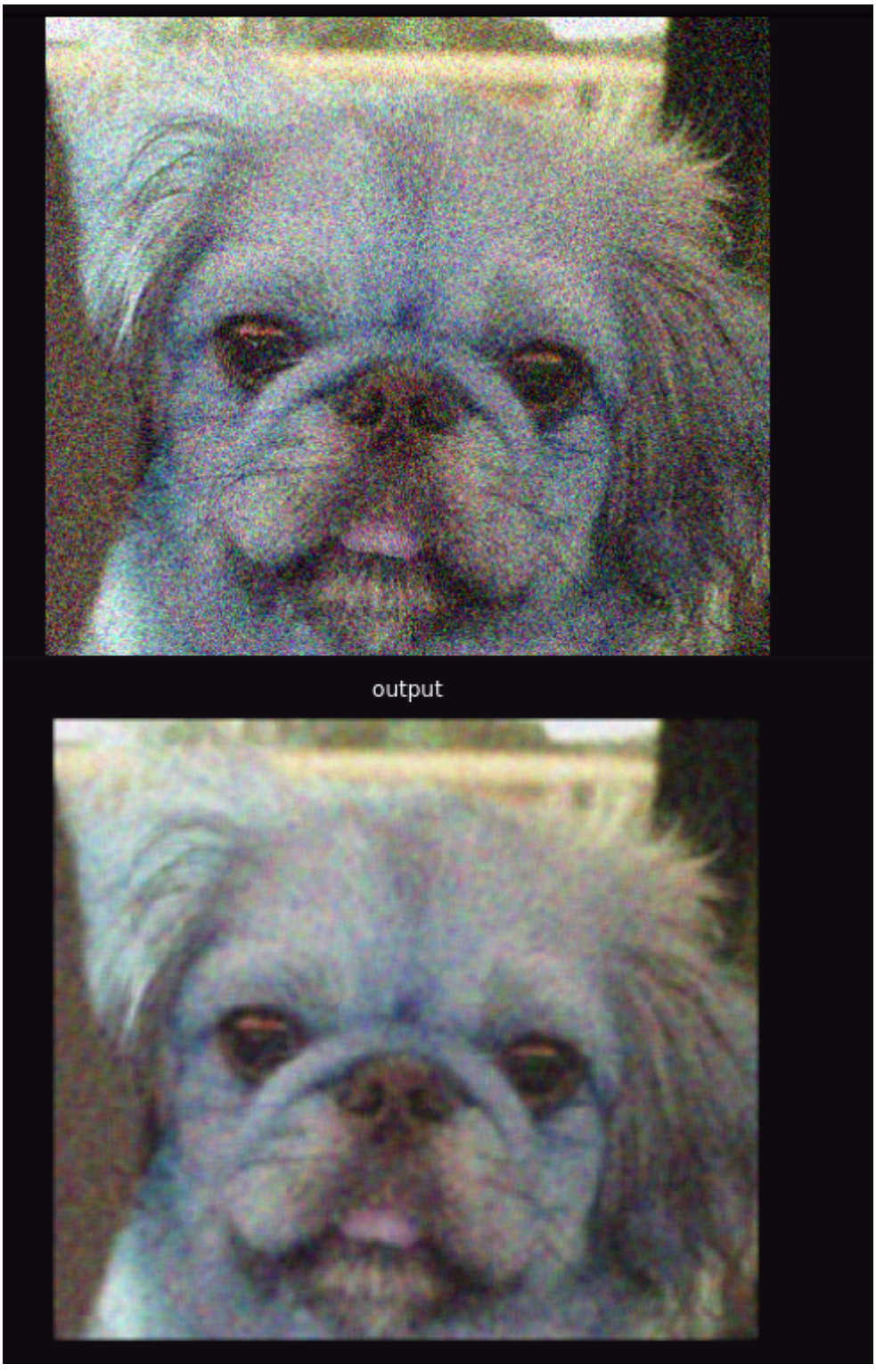
```
                filtered_image[i, j, k] = np.median(region)

    return filtered_image
```



3. Gaussian Filter

In [6]:
```
def apply_gaussian_filter(image, kernel_size=3, sigma=1):

    ax = np.linspace(-(kernel_size // 2), kernel_size // 2,
                     kernel_size)  # Generate a Gaussian kernel
    gauss = np.exp(-0.5 * np.square(ax) / np.square(sigma))
    kernel = np.outer(gauss, gauss)
```

```
    kernel /= np.sum(kernel)

    # Apply convolution
    padded_image = np.pad(image, ((kernel_size//2, kernel_size//2), (kernel_size
                          2, kernel_size//2), (0, 0)), mode='constant', constant
    filtered_image = np.zeros_like(image)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            for k in range(image.shape[2]):
                region = padded_image[i:i+kernel_size, j:j+kernel_size, k]
                filtered_image[i, j, k] = np.sum(region * kernel)

    return np.clip(filtered_image, 0, 255).astype(np.uint8)
```



Requirement 3: Edge Detection

1. Sobel filter:

In [7]:
```
def sobel_kernel(self):
    self.sobel_x = np.array([[-1, 0, 1],
                             [-2, 0, 2],
                             [-1, 0, 1]])

    self.sobel_y = np.array([[-1, -2, -1],
                             [0, 0, 0],
                             [1, 2, 1]])
    output_sobel_x = self.apply_kernel(self.sobel_x)
    output_sobel_y = self.apply_kernel(self.sobel_y)
    return output_sobel_x, output_sobel_y
```

2. Prewitt filter

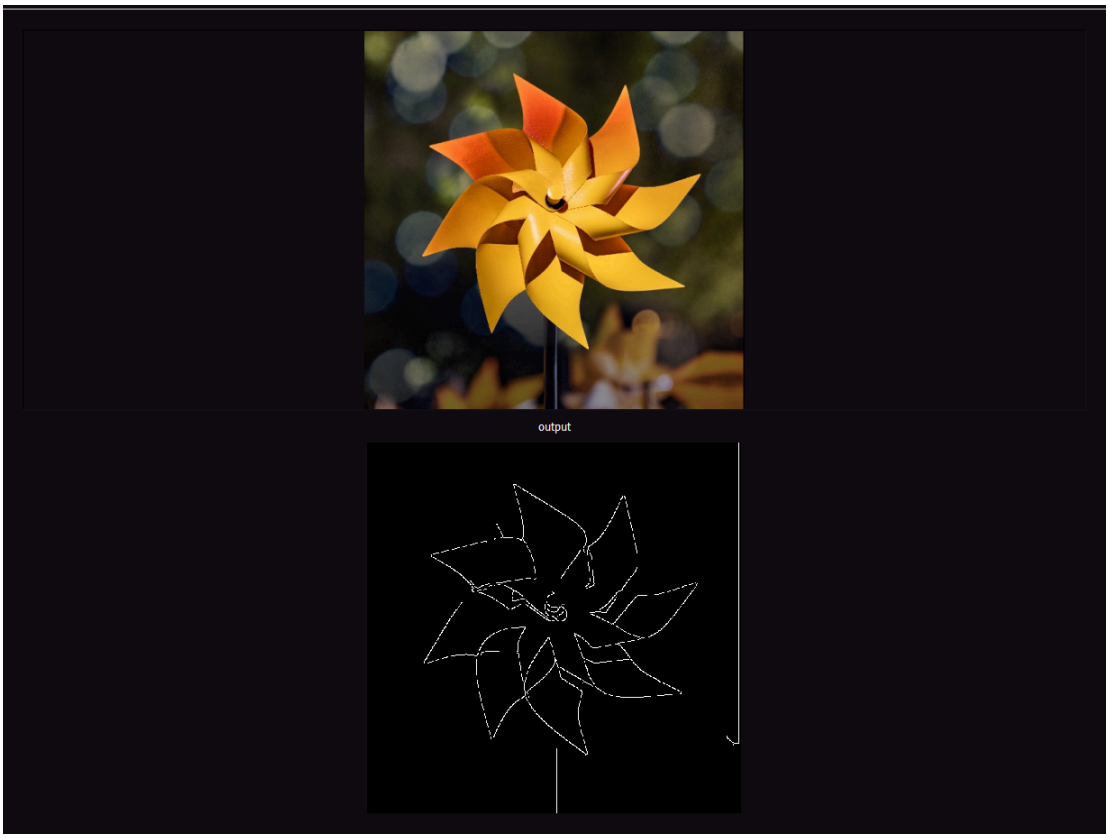
In [8]:
```
def prewitt_kernel(self):
    self.prewitt_x = np.array([[-1, 0, 1],
                               [-1, 0, 1],
                               [-1, 0, 1]])

    self.prewitt_y = np.array([[-1, -1, -1],
                               [0, 0, 0],
                               [1, 1, 1]])
    output_prewitt_x = self.apply_kernel(self.prewitt_x)
    output_prewitt_y = self.apply_kernel(self.prewitt_y)
    return output_prewitt_x, output_prewitt_y
```

3. Roberts filter

In [9]:
```
def roberts_kernel(self):
    self.roberts_x = np.array([[1, 0],
                               [0, -1]])

    self.roberts_y = np.array([[0, 1],
                               [-1, 0]])

    output_roberts_x = self.apply_kernel(self.roberts_x)
    output_roberts_y = self.apply_kernel(self.roberts_y)
    return output_roberts_x, output_roberts_y
```

4. Canny filter:

In [10]:
```
def canny_kernel(self, image):
    #Apply Gaussian blur
    # Compute gradients using Sobel filters.
    # Apply Non-Maximum Suppression.
    # Implement Double Thresholding.
    # Apply Edge Tracking by Hysteresis.
    blurred = cv2.GaussianBlur(image, (5, 5), 1.4)
    output_image = cv2.Canny(blurred, 20, 200)
    return output_image
```



Output Calculation:

In [11]:
```
def detect_edges(self):
    self.edge_detector.mask_selection = self.main_window.edge_detection_method_c
    self.edge_detector.image = self.gray_image
    gradient_magnitude = [[]]

    if self.edge_detector.mask_selection == "Sobel":
        Gx, Gy = self.edge_detector.sobel_kernel()
        gradient_magnitude = np.sqrt(Gx ** 2 + Gy ** 2)

    elif self.edge_detector.mask_selection == "Prewitt":
        Gx, Gy = self.edge_detector.prewitt_kernel()
        gradient_magnitude = np.sqrt(Gx ** 2 + Gy ** 2)

    elif self.edge_detector.mask_selection == "Roberts":
        Gx, Gy = self.edge_detector.roberts_kernel()
        gradient_magnitude = np.sqrt(Gx ** 2 + Gy ** 2)

    elif self.edge_detector.mask_selection == "Canny":
        gradient_magnitude = self.edge_detector.canny_kernel(
            self.edge_detector.image)


    gradient_magnitude = (gradient_magnitude / gradient_magnitude.max()) * 255
    gradient_magnitude = gradient_magnitude.astype(np.uint8)
    self.output_image = gradient_magnitude
    self.display_image(self.output_image_view, gradient_magnitude)
```

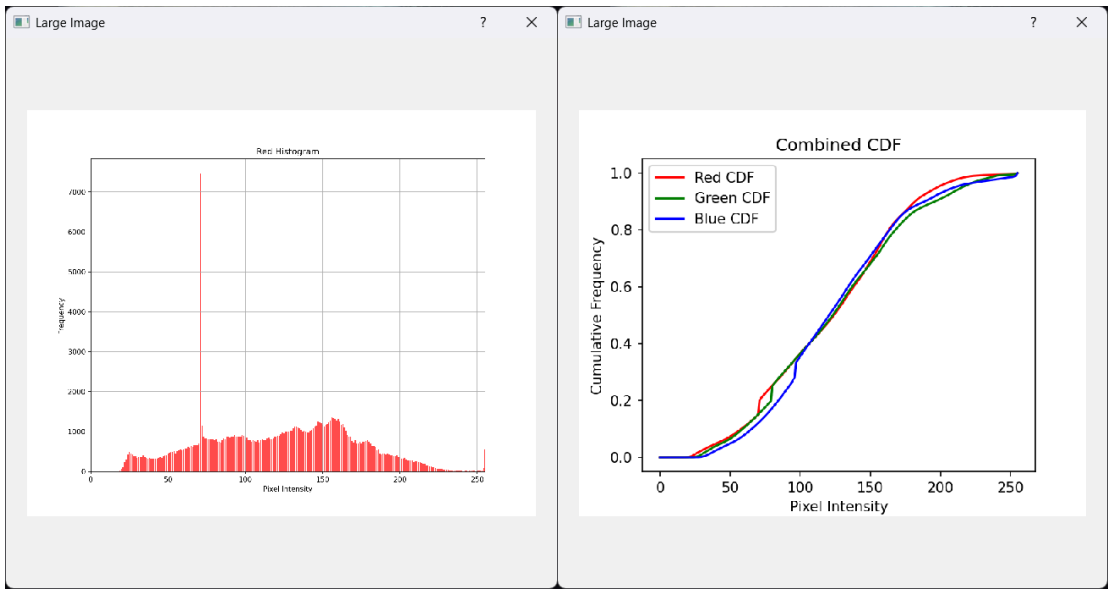
Requirement 4: Draw Histogram & CDF

1. Histogram (clear details)

In [12]:
```
def compute_histogram(image):
    histogram = np.zeros(256, dtype=int)
    for pixel_value in image.flatten():
        histogram[pixel_value] += 1
    return histogram
```

2. CDF

In [13]:
```
def compute_CDF(histogram, grey_levels=256):
    cdf = np.cumsum(histogram)
    cdf_norm = np.round(cdf / cdf[-1] * (grey_levels - 1)).astype(int)
    return cdf_norm
```



Requirement 5: Image Equalization

In [14]:
```
def equalize(image, grey_levels=256):
    if image is None: return
    image_shape = image.shape
    histogram = HistogramEqualization.compute_histogram(image)
    cdf_norm = HistogramEqualization.compute_CDF(histogram, grey_levels)
    equalized_hist = np.zeros(grey_levels, dtype=int)
    for i in range(grey_levels):
        equalized_hist[cdf_norm[i]] += histogram[i]
    new_flattened_image = cdf_norm[image]
    new_image = new_flattened_image.reshape(image_shape)
    return histogram, equalized_hist, new_image.astype(np.uint8)
```

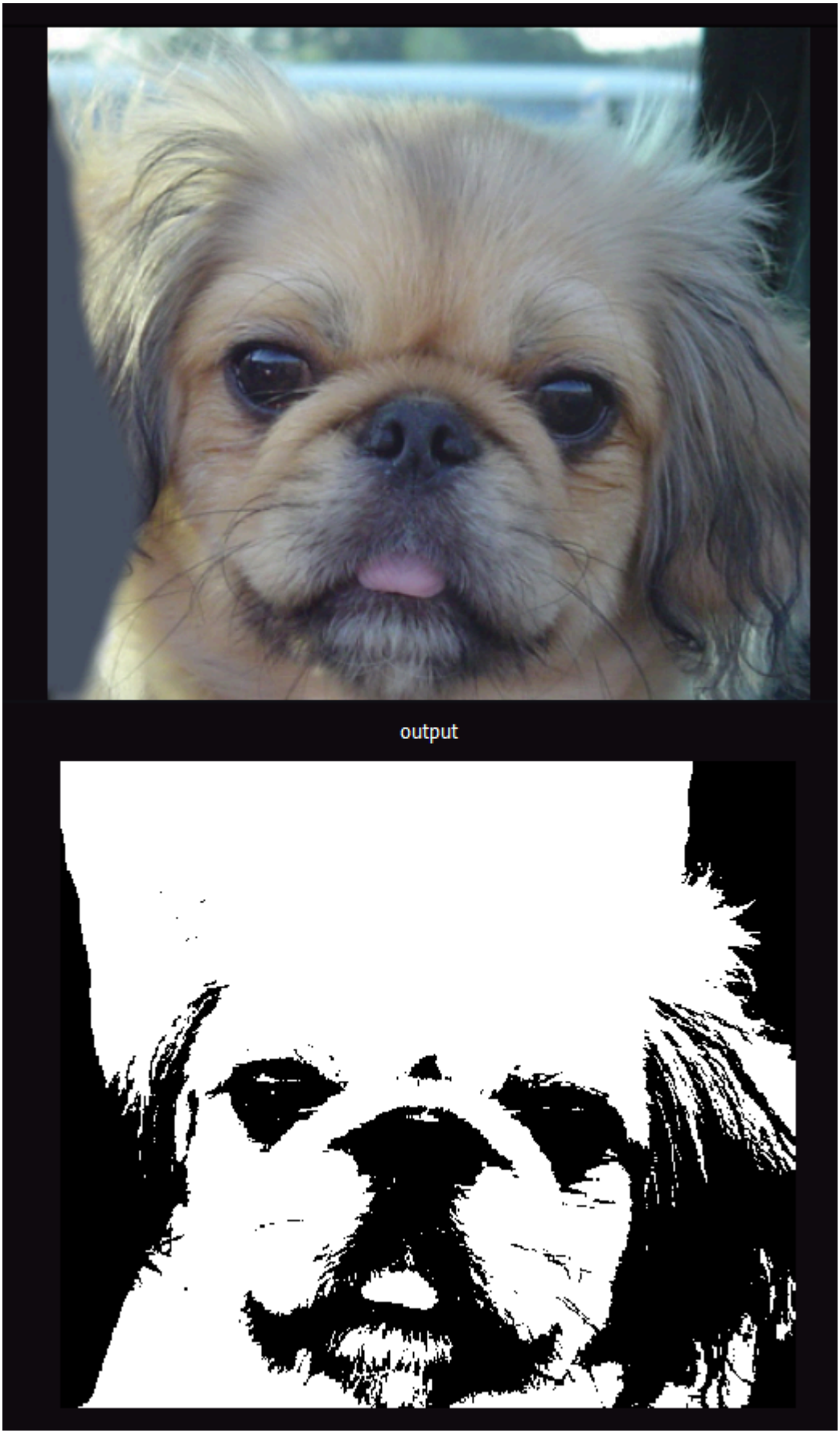
Requirement 6: Normalization

In [15]:
```python
def normalize_image(image):
    # Convert the image to a numpy array if it's not already
    image = np.array(image)

    # Get the minimum and maximum values from the image
    min_val = np.min(image)
    max_val = np.max(image)

    # Normalize the image using vectorized operations
    normalized = ((image - min_val) / (max_val - min_val)) * 255

    # Return the normalized image
    return normalized.astype(np.uint8)
```

Requirement 7: Thresholding

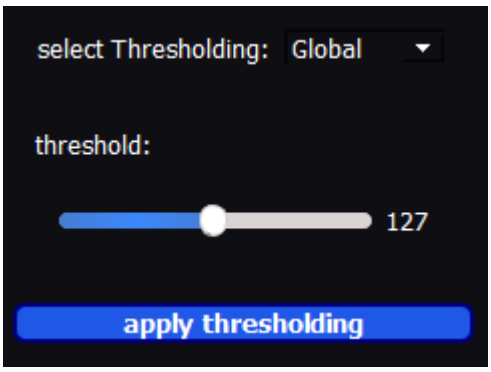
1. Global

- Parameters used: Threshold value



In [16]:
```
def global_thresholding(image, threshold):
    # Apply thresholding using NumPy vectorization
    thresholded_image = (image > threshold) * 255
    # ensures the image has the correct data type for saving and displaying
```

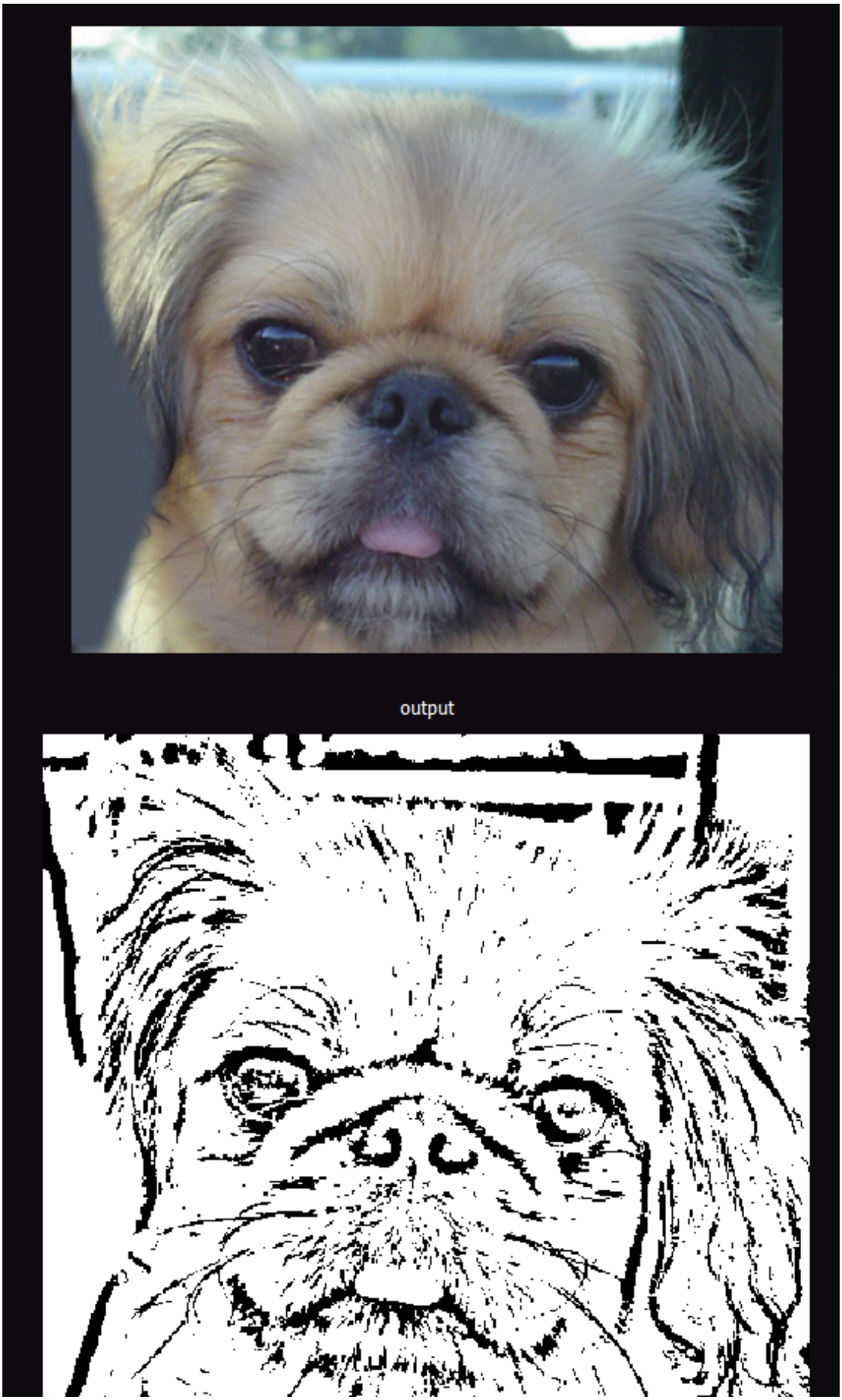
```
    thresholded_image = thresholded_image.astype('uint8')
    return thresholded_image
```



2. Local

- Type used: Local mean thresholding
- Parameters used: window size & Sensitivity(c)

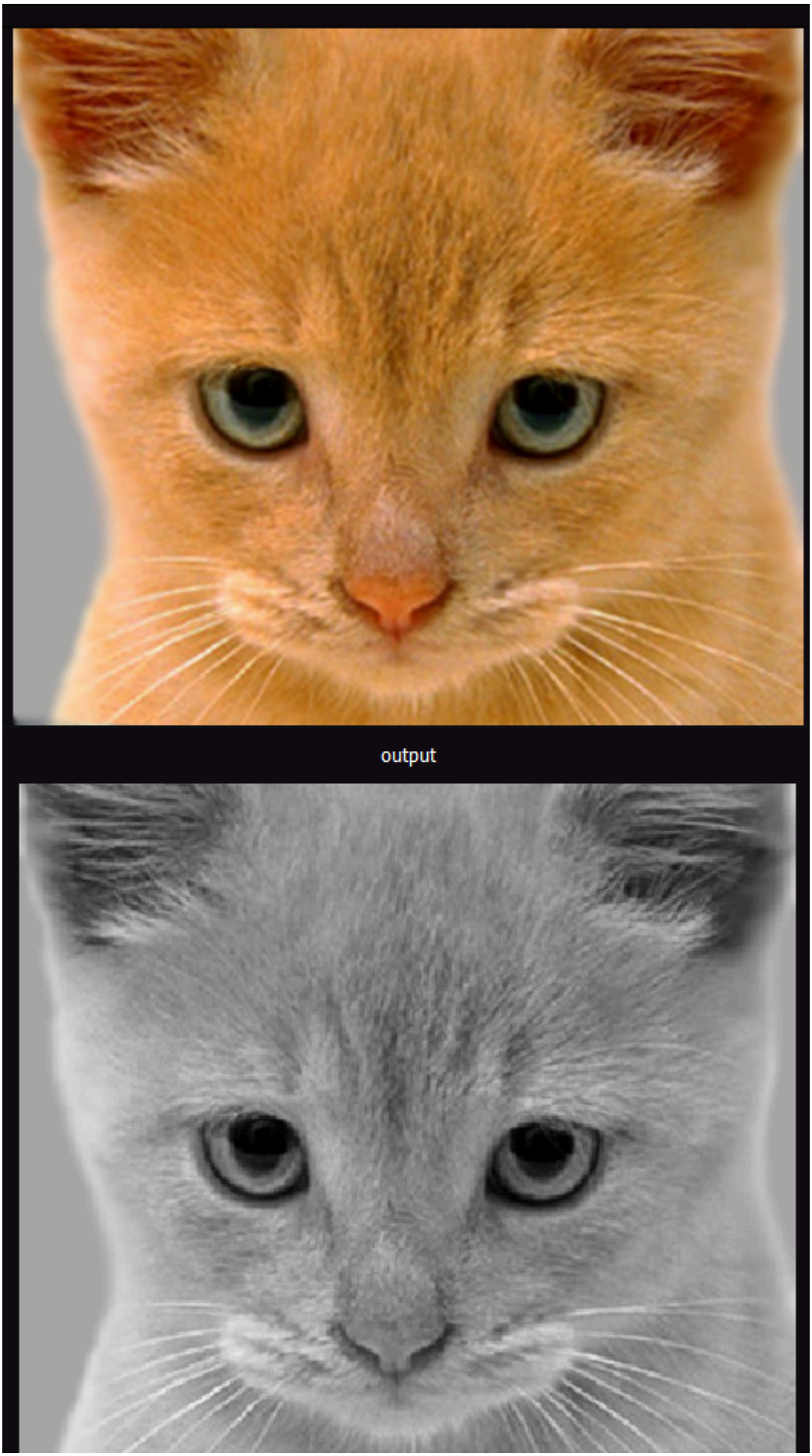
In [17]:
```
def local_thresholding(image, window_size, C):
    # Ensure window size is odd
    # if window_size % 2 == 0:
    #     raise ValueError("window_size must be an odd integer.")

    # Pad the image to handle borders
    pad_size = window_size // 2
    padded_image = np.pad(image, pad_size, mode='reflect')

    # Create an empty array for the thresholded image
    thresholded_image = np.zeros_like(image, dtype=np.uint8)

    # Iterate over each pixel in the original image
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            # Extract the local window
            y0, y1 = y, y + window_size
            x0, x1 = x, x + window_size
            window = padded_image[y0:y1, x0:x1]

            # Calculate the local mean
            local_mean = np.mean(window)

            # Calculate local threshold and apply it
            T_local = local_mean - C
            thresholded_image[y, x] = 255 if image[y, x] > T_local else 0

    return thresholded_image
```



Requirement 8: Converting from RGB to Grayscale

In [18]:

```python
def convert_to_grayscale(image):
    # Ensure the image has three channels (BGR)
    if len(image.shape) != 3 or image.shape[2] != 3:
        raise ValueError("Input image must be a BGR image with 3 channels.")

    # Get image dimensions
    height, width, channels = image.shape

    # Initialize a new array for the grayscale image
    grayscale_image = np.zeros((height, width), dtype=np.uint8)

    # Convert to grayscale using the luminance formula
    for y in range(height):
        for x in range(width):
            # Get BGR values (OpenCV uses BGR format)
            B, G, R = image[y, x]

            # Calculate grayscale value
            gray = int(0.299 * R + 0.587 * G + 0.114 * B)

            # Assign the grayscale value to the new image
            grayscale_image[y, x] = gray

    return grayscale_image
```



- RGB Histogram

- By extracting the R, G and B values from the image, We plot each channel's histogram individually
- Computed Histogram: By counting the frequencies of each intensity value

In [19]:
```python
def compute_histogram(channel_values):
    # Initialize histogram with zeros for all possible intensity values (0-255)
    histogram = [0] * 256

    # Count the frequency of each intensity value
    for value in channel_values:
        histogram[value] += 1

    return histogram
```

- Compute CDF: By getting the probabilities of each intensity and sum them till it reaches 1

In [20]:
```python
def compute_cdf(histogram):
    cdf = []
    cumulative = 0
    total_pixels = sum(histogram)
    for count in histogram:
        cumulative += count
        cdf_value = cumulative / total_pixels  # Normalize to [0,1]
        cdf.append(cdf_value)
    return cdf
```

Requirement 9: Frequency Filters

- Parameters used: Cut Off Frequency
- Steps:

1. Getting fourier transform
2. Creating filter by computing Euclidean distance and whether the filter is low or high, the intensity value is changed according to the cut off frequency and the filter type
   Euclidean distance equation:

```
D = np.sqrt((U - M // 2) ** 2 + (V - N // 2) ** 2)
```

3. Applying filter by element wise multiplication
4. Getting inverse fourier transform
5. Normalizing the image

Requirement 10: Hybrid Image

- By taking low frequencies from one image and high frequencies from the other image and adding both we can get the hybrid image Gaussian Kernel is generated by **gaussian function**

In [21]:
```
def generate_gaussian_kernel(size, sigma):
    ax = np.arange(-size // 2 + 1., size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-(xx**2 + yy**2) / (2. * sigma**2))
    kernel = kernel / np.sum(kernel)
    return kernel
```

- Parameters used: Kernel size & Sigma