COMP37212 Lab1 Report

Rui Xu

1. Abstract

This report delves into the application of convolution operations in image processing, focusing on tasks such as image smoothing, gradient computation, and edge detection. Using the provided image (kitty.bmp, Appendix1), we investigate the effects of convolving with average and weighted average smoothing kernels. Subsequently, we compute gradient images and derive edge strength images to detect major edges while mitigating unwanted texture patterns. Comparative analysis between convolving the original image and its weighted mean counterpart sheds light on the impact of smoothing on edge detection. Our findings underscore the significance of convolution techniques in enhancing edge detection algorithms for various computer vision applications.

2. Preparation

For the purpose of this assignment, Python 3.9.6 has been selected as the implementation language, with a majority of functions developed from scratch. Throughout the development process, the manipulation of matrices has been facilitated through the utilization of Numpy arrays, while OpenCV has been employed mainly for input-output image operations. Convolution and thresholding functions are **all implemented from scratch without external libraries**.

3. Implementation

Please tap the headings to see code snippets⬇️

Convolution

The convolution function facilitates a 2D convolution operation between an input image and a 3x3 kernel. It ensures the kernel dimensions are odd, stores the original image data type, and applies zero-padding for edge and corner handling. After converting the image to a floating-point type and normalizing it, the function initializes a result matrix. Through explicit pixel-wise iteration(looping over the image pixels), it executes the convolution operation, summing the products of

pixel and neighboring values with corresponding kernel elements. The results undergo absolute normalization and conversion back to the original data type before being returned.

Padding

When a filter (or kernel) is applied to an image, the spatial dimensions of the output can shrink depending on the size of the filter and the stride used. Padding helps to maintain the spatial dimensions of the input and output volumes. The padding function takes the input image and adds extra rows and columns of zeros around the edges of the image. The number of rows and columns to be added depends on the size of the filter and the desired padding.

Mean filter

The mean filter serves as a basic smoothing kernel, where each pixel value is substituted with the average of itself and its neighboring pixels. However, a drawback lies in its inability to consider the spatial context of pixels. Consequently, neighboring pixels, irrespective of whether they represent the same object as the current pixel, exert equal influence on the resultant pixel.

Weighted-mean(Gaussian) filter

The weighted-mean filter addresses the previously mentioned concern by assigning weights to each element within the kernel. To determine these values, a two-dimensional Gaussian distribution function has been incorporated, yielding the characteristic bell-shaped curve. Consequently, a distribution of values is generated, which undergoes normalization before convolution.

The function code reveals two parameters within the Gaussian kernel: size and sigma value. The kernel size refers to the dimensions of the kernel matrix, typically represented as an N×N square matrix. Meanwhile, the sigma value signifies the standard deviation of the Gaussian distribution.

Thresholding

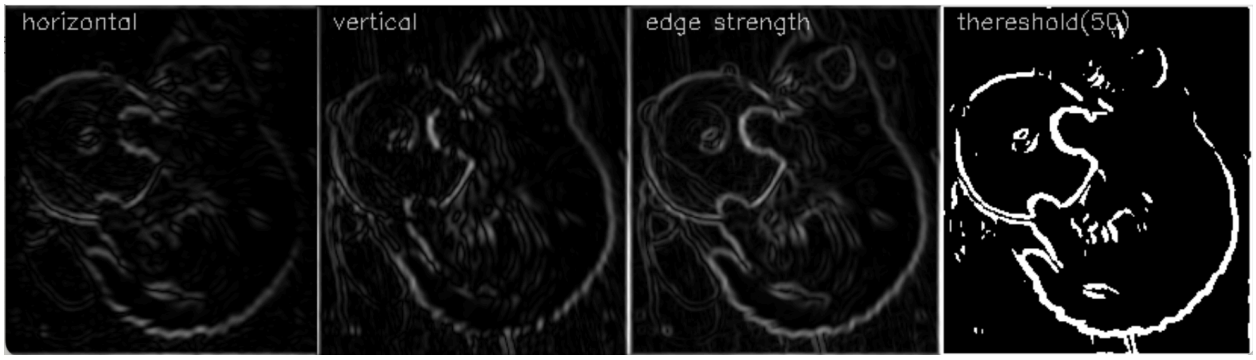
Thresholding helps in accentuating edges by categorizing pixels into foreground (edges) and background based on their intensity values. By setting a threshold

value, pixels above it are considered part of the edge, while those below it are not. This enhances the visibility of edges in the image.

4. Results

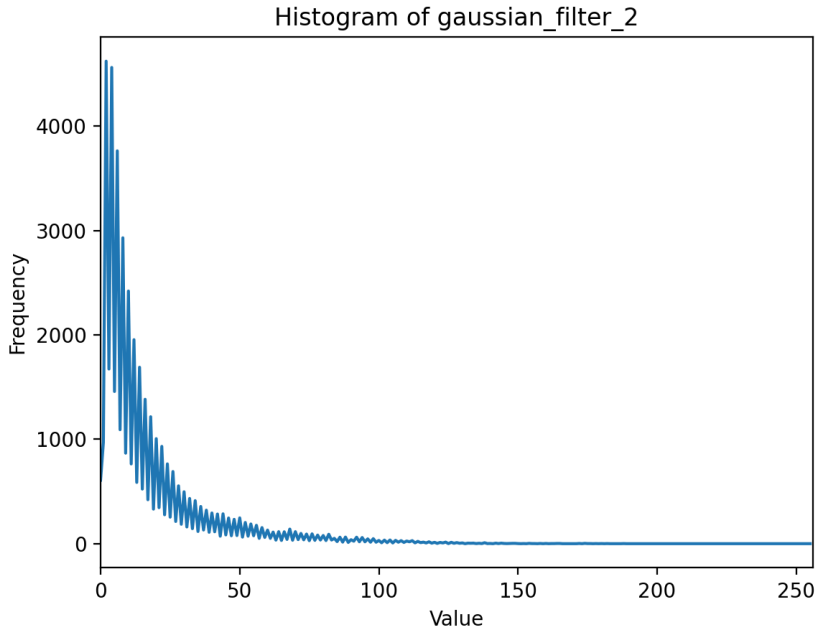
4.1 Gradient Images and Thresholding

As per the coursework requirements, I've successfully implemented the **horizontal, and vertical(using Sobel Operator), and combined gradient images** along with **thresholding**. The outcomes are displayed below.



The resulting image above is generated using a Gaussian kernel size of 7x7 and a sigma value of 1.5 for Gaussian filtering. Normalizing was utilized for extra processing. The thresholding value is 50(The 4th image).

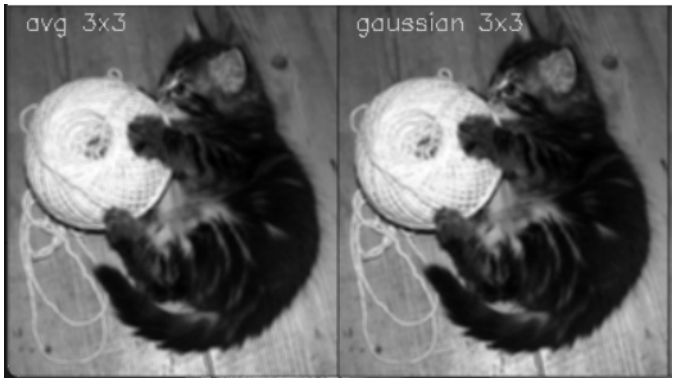
The histogram below illustrates the distribution of the edge strength image.



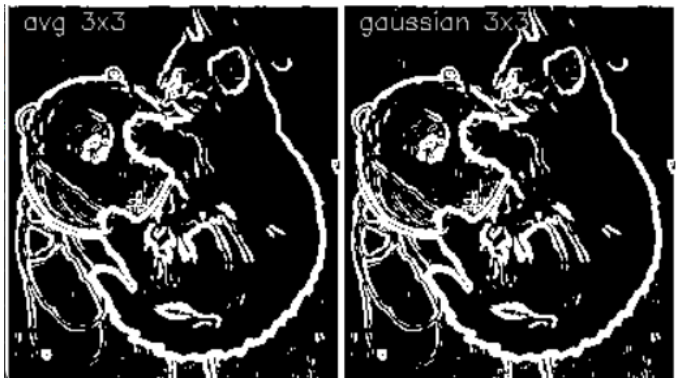
4.2 Compare Average Filtering / Weighted Average (Gaussian) Filtering

Below is a comparison of the outcomes of weighted average (Here we use Gaussian) filtering versus average filtering.

Blur image

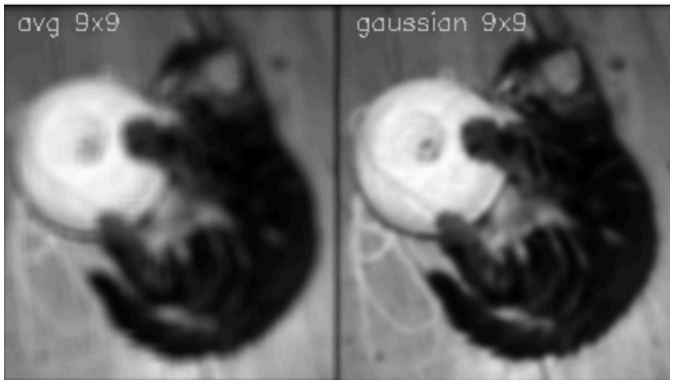


Thresholding image



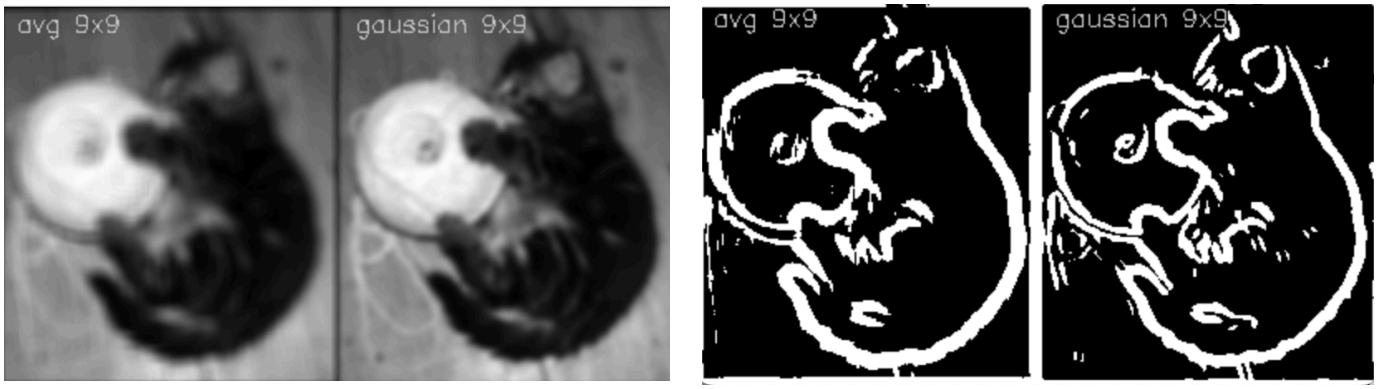
The outcomes of Gaussian filtering were generated employing a 3x3 Gaussian kernel size, a sigma value of 1.5, and absolute value normalization for post-processing. However, because of the small kernel size and sigma value, there doesn't appear to be much variation in the results.

Therefore, a fresh comparison is presented between the outcomes of average filtering(9x9) and weighted average (Gaussian) filtering. This time, a larger kernel



size (9x9) and sigma value (2) are employed for the Gaussian filtering process.

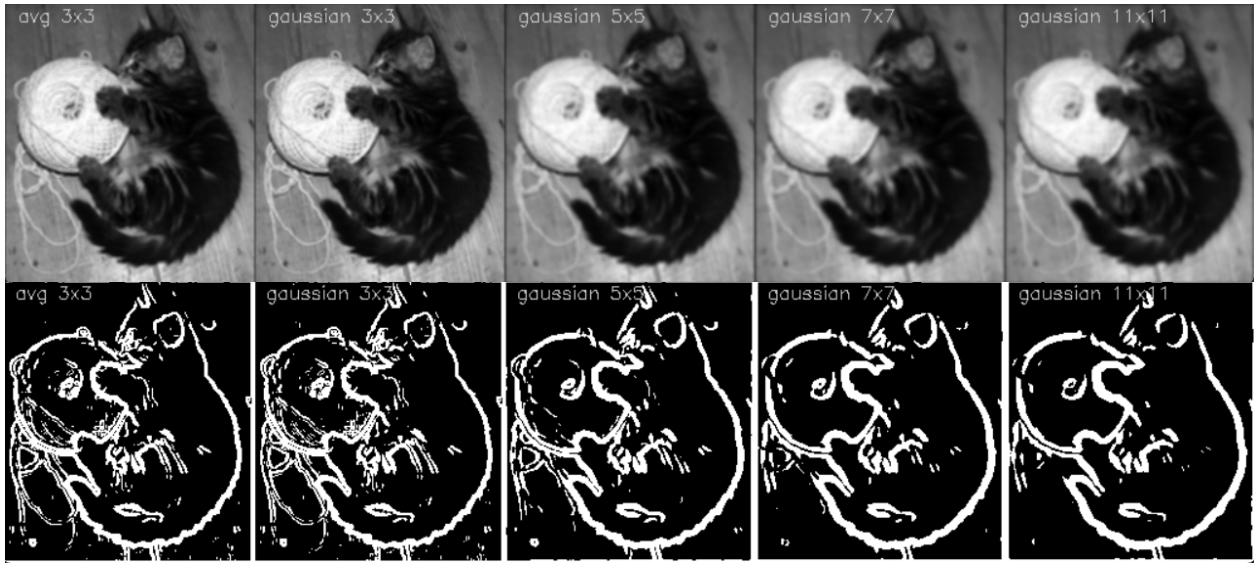
Analysis Conclusion



Both of the kernels in the image above blur the image, however, the Gaussian kernel **retains more of the image's details**. The thresholding findings demonstrate that the edges are more visible and the Gaussian kernel performs **better** at edge detection.

4.3 Kernel Size

The results of the experiments of the Gaussian filter on different kernel sizes are shown below.



Based on the provided images, it is evident that increasing the kernel size leads to greater blurring or smoothing of the filtered image. Additionally, the thresholding results indicate that with larger kernel sizes, the edge image becomes **clearer**, and the edges appear **thicker.**

Additionally, to obtain the best results, several thresholding values are used for images with varying kernel sizes applied, as shown in the images. The Gaussian kernel appears as follows for various kernel sizes: (All photos have a sigma value of 2; intensity is rescaled.)

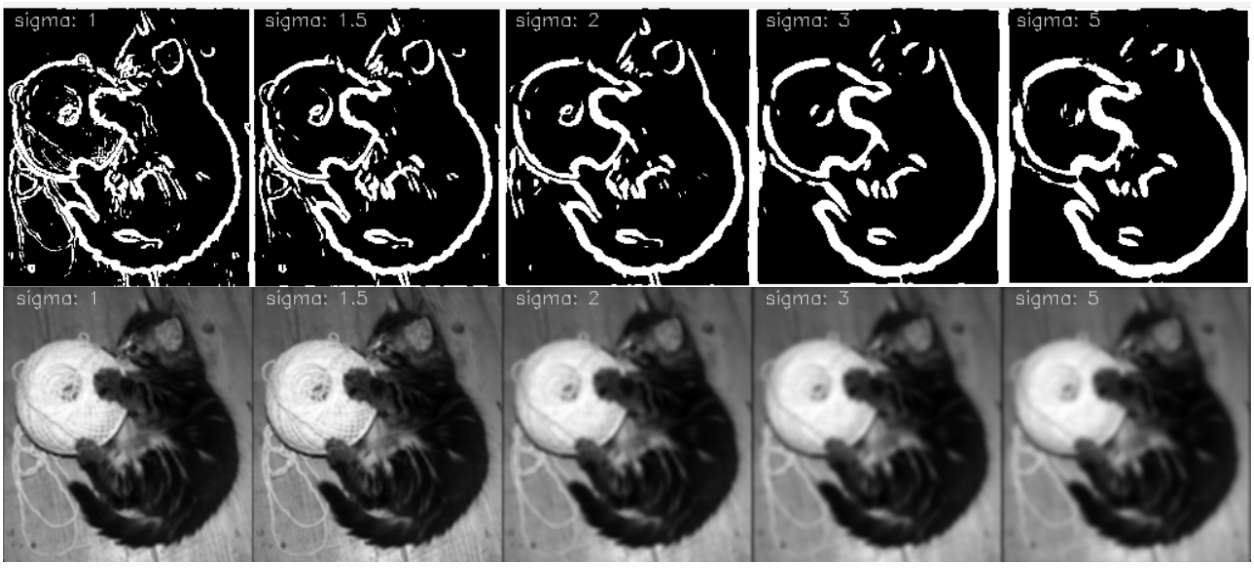
Analysis Conclusion

The kernel matrix, which is a square matrix with dimension NxN, is the same size as the kernel. The number of pixels used in the computation of the new pixel value increases with increasing kernel size. Therefore, as the kernel size **increases,** the image becomes increasingly **blurry**.

4.4 Sigma Value

Below are the findings from the Gaussian filtering studies performed on various sigma values.

The sigma value influences the image's blurring effect in a similar way to the kernel size. With an **increase** in sigma value, the image gets progressively **blurred**.



Analysis Conclusion

The sigma value determines the breadth of the Gaussian distribution, specifying the extent of values it covers. A higher sigma value yields a wider distribution, leading to a more even averaging of pixels within the kernel matrix (flattening and smoothing the Gaussian curve). Conversely, a lower sigma value concentrates the kernel matrix more around the central pixel, enhancing the contrast between it and neighboring pixels (yielding a sharper Gaussian distribution).

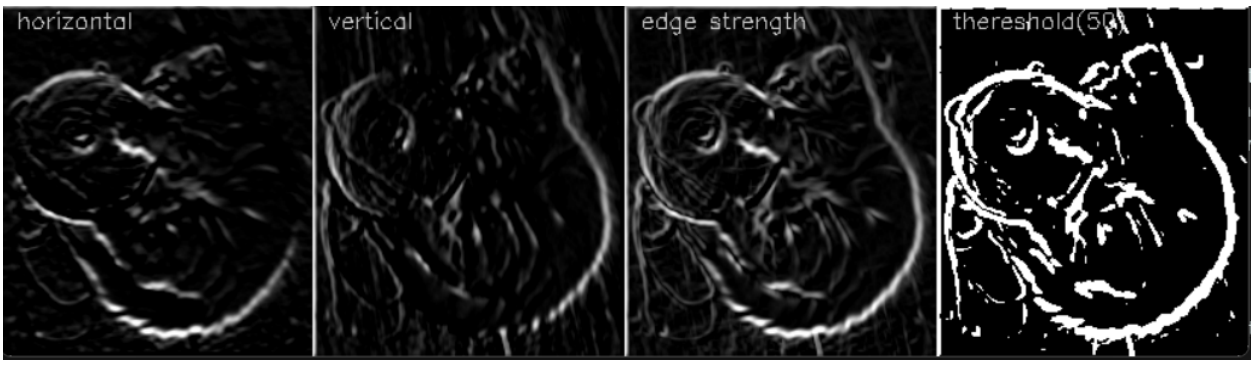
When applying a kernel with a smaller sigma value to an image, neighboring pixels exert less influence on the calculation of the new pixel value, with the central pixel predominantly determining it. Consequently, the filtration has a lesser impact on the image, rendering edges more pronounced. Conversely, in a kernel with a **larger sigma value**, surrounding pixels are more evenly involved in the calculation, leading to **greater blurring of the image and less distinct edges**.

4.5 Compare normalization and clipping

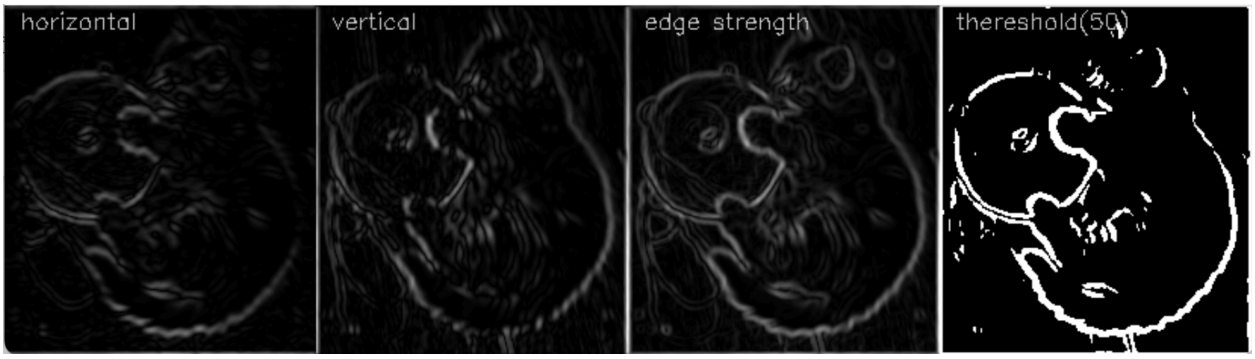
The process of normalization involves mapping the gradient image's values to a new range of values. It maps the gradient image's values to the new range by using the image's minimum and maximum values as the new values' range.

In contrast to re-mapping, clipping is a technique for limiting or discarding all excess values—that is, pixel values that exceed 0-255—obtained from convolution and other computations. As a result, some of the gradient photos' details have been lost.

Clipping



Normalization



The clipped images exhibit slightly clearer edges but also contain more noise. Moreover, upon examining the edge strength image, it becomes evident that the clipped image displays higher contrast(brightness) along the edges, facilitating easier detection by human observers.

Conclusion

This report has provided a comprehensive exploration of convolution operations in image processing, specifically focusing on tasks such as image smoothing, gradient computation, and edge detection. The implementation of various convolution techniques and analysis of their effects on edge detection, underscores the significance of convolution techniques, particularly Gaussian filtering, in enhancing edge detection algorithms for various computer vision applications. By understanding the nuances of kernel sizes and sigma values, practitioners can optimize image processing pipelines to achieve desired outcomes, balancing between image smoothing and edge preservation effectively.

5.Appendix

5.1 Image

5.2 Code snippet

Convolution function

```
onvolution(m, k):
'"Performs 2D convolution on an image.
rgs:
   m: input image
   k: kernel"""
k, ky = k.shape[0], k.shape[1]
ssert kx % 2 == 1 and ky % 2 == 1, "convolution error: kernel dimensions must be odd numbers; got {} and {}".form
   kx, ky)

rig_type, alpha, beta = np_to_cv_type(m.dtype)    # Remember the original type
k, py = kx // 2, ky // 2   # Compute what padding is required

 Pad matrix to deal with edges and corners and convert it to float64
added = padding_matrix(m, 0, px, py)

 Initialize result matrix
esult64 = np.empty(m.shape, dtype=np.float64)

 Loop through the image and perform the convolution
k, ny = m.shape
or i in range(0, nx):
   for j in range(0, ny):
       region = padded[i:i+kx, j:j+ky]
       result64[i, j] = np.sum(region * k)

esult = post_process(result64, orig_type, method="clip", alpha=alpha, beta=beta)

eturn result
```

Padding function

```python
def padding_matrix(m, v=0, width=1, height=1):
    """Pads a matrix with a given value for image convolution.
    Args:
        m: matrix
        v: value
        width: padding width
        height: padding height"""
    dx = m.shape[0] + 2 * width
    dy = m.shape[1] + 2 * height
    result = np.full((dx, dy), v, dtype=m.dtype)
    result[width:m.shape[0]+width, height:m.shape[1]+height] = m
    return result
```

Mean Kernel

```python
def mean_distribution(row, col, ktype=np.float32):
    """Generates an average distribution mxn matrix, where all elements are 1/(row*col).
    Used for mean filtering."""
    return np.ones((row, col), dtype=ktype) / (row * col)
```

Gaussian Kernel

```python
def gaussian_distribution(row, col, amp, sigma,
                          cx=0, cy=0,
                          ktype=np.float32,
                          normalize=True):
    """Generates a Gaussian distribution mxn matrix.
    Args:
        row: number of rows
        col: number of columns
        amp: amplitude
        sigma: standard deviation
        cx/cy: center on X/Y axis
        ktype: type of the matrix
        normalize: normalize"""
    kernel = np.empty((row, col), dtype=np.float64)

    # Compute anchor point
    ax, ay = row // 2, col // 2
    # Fill matrix with results from the 2D Gaussian function
    total = 0.0
    for i in range(0, row):
        for j in range(0, col):
            x = i - ax
            y = j - ay
            gx = ((x-cx)**2) / (2 * sigma**2)
            gy = ((y-cy)**2) / (2 * sigma**2)
            value = amp * np.exp(-(gx + gy)) / (2 * np.pi * sigma**2)
            kernel[i, j] = value
            total = total + value

    if normalize:    # Normalize numbers by default so they add up to 1
        kernel = kernel / total

    return kernel.astype(ktype)
```

Thresholding

```python
def thresholding_binary(img, threshold):
    """Performs binary thresholding on an image.
    Args:
        img: input image
        threshold: threshold"""
    new_img = np.zeros(img.shape, np.uint8)
    height, width = img.shape[:2]
    for i in range(height):
        for j in range(width):
            if img[i,j] > threshold:
                new_img[i,j] = 255
            else:
                new_img[i,j] = 0
    return new_img
```

Normalization

```python
def custom_normalize(image, alpha=0, beta=1, norm_type='minmax', dtype=np.float64):
    """
    Custom implementation of normalization similar to cv2.normalize function.

    Parameters:
        image: numpy.ndarray
            The input image.
        alpha: float, optional
            The lower bound of the normalization range.
        beta: float, optional
            The upper bound of the normalization range.
        norm_type: str, optional
            The type of normalization. Options are 'minmax' for min-max normalization
            and 'meanstd' for mean and standard deviation normalization.
        dtype: type, optional
            The data type of the output array.

    Returns:
        numpy.ndarray
            The normalized image."""
    if norm_type == 'minmax':
        min_val = np.min(image)
        max_val = np.max(image)
        normalized_image = (image - min_val) / (max_val -
                                                min_val) * (beta - alpha) + alpha
    elif norm_type == 'meanstd':
        mean_val = np.mean(image)
        std_val = np.std(image)
        normalized_image = (image - mean_val) / std_val
        normalized_image = np.clip(normalized_image, alpha, beta)
    else:
        raise ValueError(
            "Invalid normalization type. Choose 'minmax' or 'meanstd'.")

    return normalized_image.astype(dtype)
```

Clipping

```python
def post_process(img, orig_type, method="clip", alpha=0, beta=1):
    if method == "clip":
        return np.clip(img, 0, 255).astype("uint8")
    elif method == "normalize":
        new_img = custom_normalize(np.absolute(img), alpha=alpha, beta=beta, norm_type='minmax', dtype=orig_type)
        return new_img
    else:
        return img
```