CS-GY 6643 Computer Vision
Project Report 2

Runze Li
rl50xx@nyu.edu

March 10, 2024

1 Convolution and Derivative Filters

1.1 Correlation/Convolution

Implement 2D convolution (denoted $*$). Your function will take as input two 2D arrays, a filter f and an image I, and return $f * I$. You can handle the boundary of I in any reasonable way of your choice, such as padding with zeros based on the size of the filter f. You are free to make use of any part of your previous indexing implementation.

Make the convolution module flexible to receive 1D line filters (used for separable filtering) and also masks representing image templates (used in the second question below).

Please note that all processing is **done in floating point** and not on byte images. As a first step, convert input images into float arrays, and also use float arrays for all the processing steps. Would you not have a display of floats, you can at the very end convert the results back to byte or integers.

Given an input image I and a convolution kernel f, the convolution operation $f * I$ is defined as:

$$(f * I)(i,j) = \sum_{k,l} f(k,l) I(i-k, j-l)$$

The code A.1 is as shown in the formula. If we assume the input data is:

```
f = np.array([[0.0, 0.1, 0.0],
              [0.1, 0.6, 0.1],
              [0.0, 0.1, 0.0]])
I = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
result = convolution(f, I)
print("Convolution:")
print(result)
```

We can get the result:

```
Convolution:
[[1.3 2.2 3.1 3.5]
 [4.6 6.  7.  7.1]
 [6.9 8.6 9.5 9.1]]
```

1.2 Edge Detection Filtering

All operations have to be performed with floating point images and operation, only for output and report you would convert back to byte [0..255] images.

Denoise the image with a Gaussian filter. Use separability to consecutively filter first with a horizontal 1D Gaussian G_x and second a vertical 1D Gaussian G_y. Reasonable choices are Gaussian width of sigma 2.0 or 3.0, with filter sizes of 3*sigma to the left and the right, which results in filter widths of 6*sigma+1.

Given σ of the Gaussian function, we cab calculate the value of the Gaussian function at each position:

$$g_{1D}(x) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{x^2}{2\sigma^2})$$

$$g_{2D}(x) = \frac{1}{2\pi\sigma^2} exp(-\frac{x^2+y^2}{2\sigma^2})$$

Here we set $sigma = 2.0$. Implement the 1D Gaussian filter as shown in the code A.2, and implement the 2D Gaussian filter as shown in the code A.3. By denoising the two pictures with a Gaussian filter, we can get the denoised picture as shown in the figure 1.

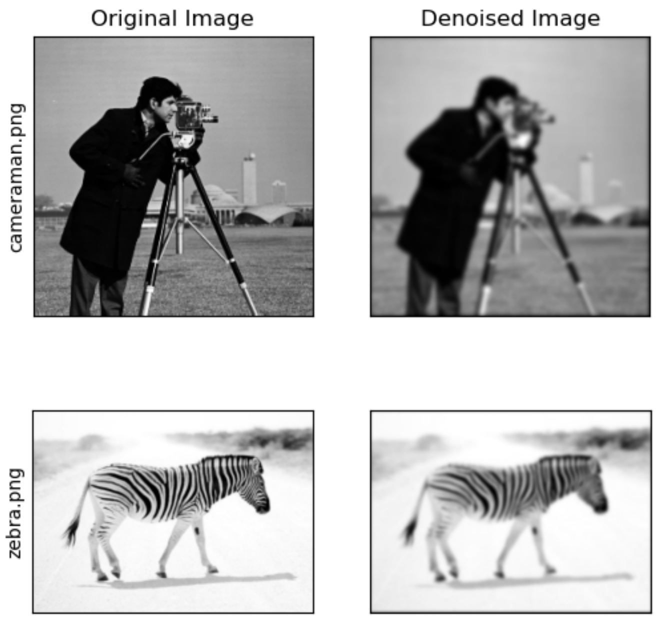

Figure 1: Denoise the image with a Gaussian filter

The result shows the original and denoised versions of an image. The image on the left appears sharper and has more detail, while the denoised image on the right has a blurred effect, which is characteristic of the smoothing done by the Gaussian filter.

1.2.1 Version via first derivatives

Compute derivative images (with respect to x and with respect to y) using the separable derivative filter of your choice, e.g. $[-1\ 0\ 1]$ and $[-1\ 0\ 1]^T$. Here we use the following code to get f_x and f_y:

```
fx = np.array([[-1, 0, 1]], dtype=np.float32)
fy = np.transpose(fx)
```

Based on the code, we can depict x- and y-derivatives as shown in figure 2. The horizontal derivative image indicates the vertical edges in the image. Moreover, vertical derivative images show horizontal edges in the image.

So we can compute the gradient magnitude image that combines x- and y-derivatives, and the code is shown in A.6:

$$G = \sqrt{G_x^2 + G_y^2}$$

Here we choose $threshold = 45$, and we can create binary edge image from the gradient magnitude image 3. The outlines of people and the stripes of zebras are clearly marked in the binary edge image, indicating that the gradient magnitude in those areas exceeds the threshold.

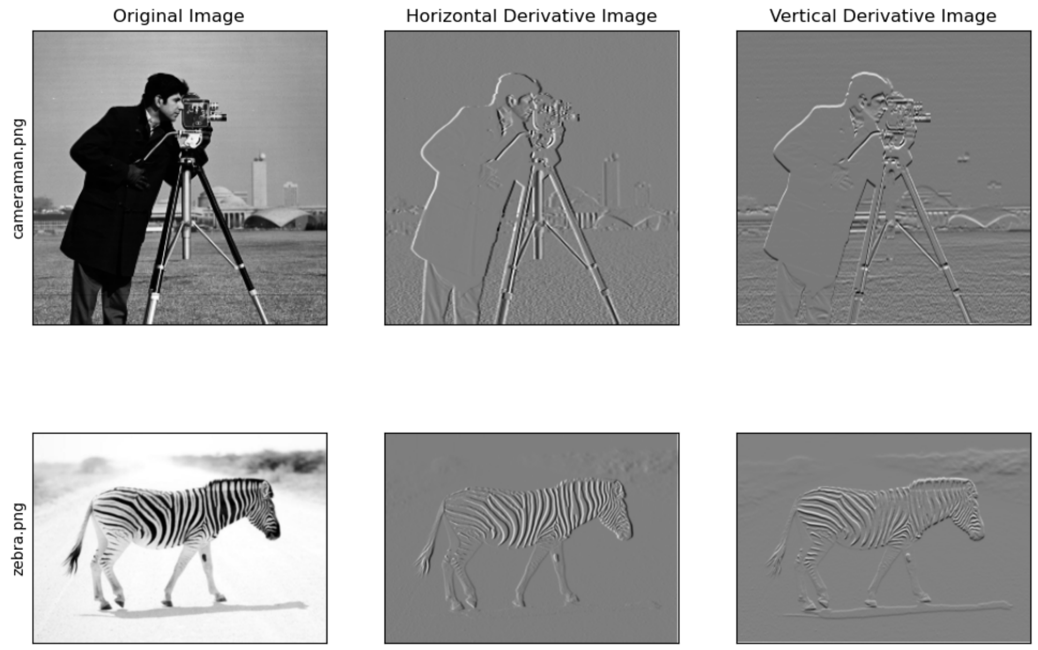

Figure 2: Compute derivative images with respect to x and y

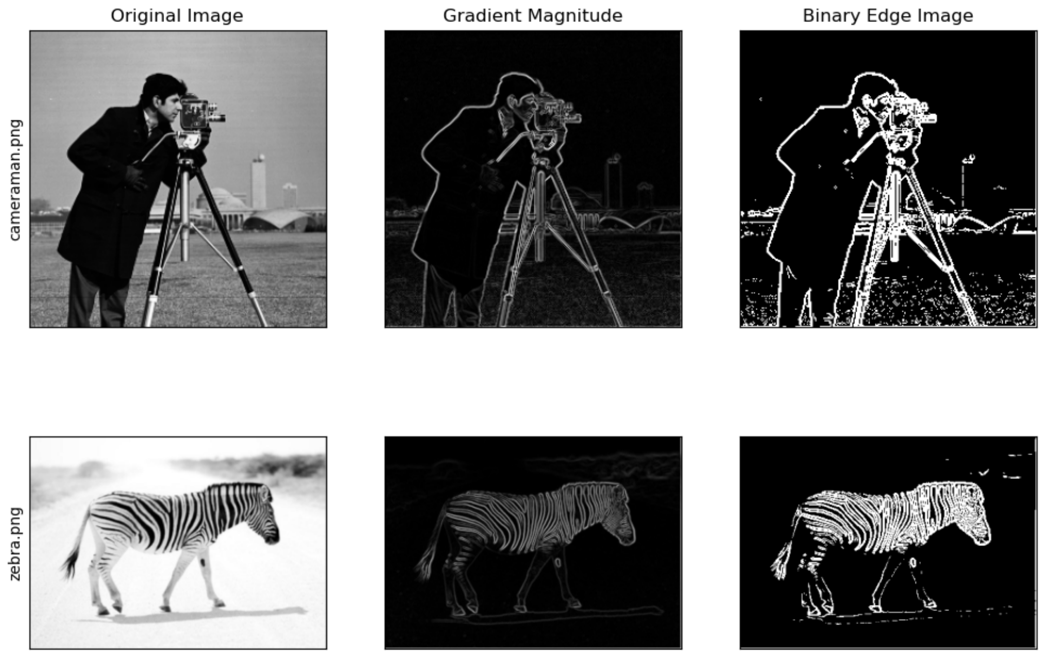

Figure 3: Draw magnitude image and threshold magnitude image

1.2.2 Version via zero-crossing of second derivative

Based on the Gaussian-smoothed image, we apply a $3x3$ Laplacian filter to mimic the 2nd derivative in 2D:

```
laplacian_filter = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
```

We need to write a $3x3$ procedure for detection of zero-crossings following the rule "to find pixels with positive values that have at least one negative neighbor":

First, we scan the Laplacian filtered Gaussian-smoothed image, for each pixel which is positive, check all 8 neighbors for negative pixels. If there is at least one negative pixel, we mark the center pixel as an edge in the output image, which results in an image where all edges are found, even in the noise. The code is shown in A.9. Actually, the output will be an edge image with thin lines of mostly one-pixel width.

If we want to get a better result, we can additionally filter for strong edges if you only select negative pixels which show at least a difference value delta or more to your positive center pixel. Here we set $delta = 5$, and the code is shown in A.10.

Using these steps, we can get the results including Gaussian-smoothed image, Laplacian-filtered image, edge image and improved edge image, as shown in figure 4.

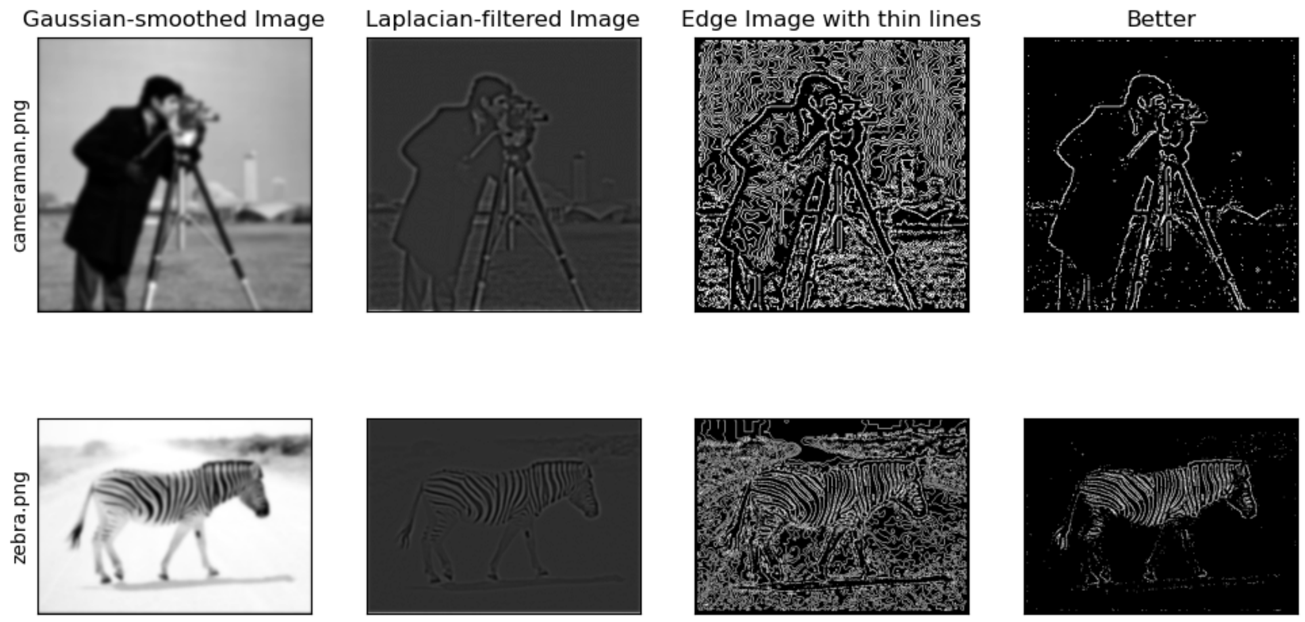

Figure 4: Draw Laplacian-filtered image and edge image

The first column in the figure represents the Gaussian smoothed image, which removes some noise and edges and details become less sharp. The second column is the Laplacian filtered image. The third column is the edge image determined by detecting zero crossing points in the Laplacian filtered image. The result is an image containing a large number of fine lines showing all possible edges in the image, including those caused by noise. The last column is the improved edge image. By introducing a threshold we can further filter out edges with insufficient intensity, resulting in an image with fewer edges but generally more prominent and relevant features.

2 Cross-correlation and Template Matching

The uploaded image 'animal-family-25.jpg' is a single image containing multiple instances of different objects. To detect and count objects, perform the following steps:

First, we use the template image 'animal-family-25-template.jpg' as your filter-mask and convert the original byte image and template to float.

Second, we need to prepare the template image. Concretely, we calculate the min, max and mean value of the template image, and then normalize the template image by subtracting the mean value from all pixel values, so that the mean of the resulting template becomes 0.0. The original image, template image and zero-mean template are shown in figure 5. What's more, we get the result of min, max and mean value of template image:

```
Minimum value: 0
Maximum value: 255
Mean value: 135.27836163836164
```

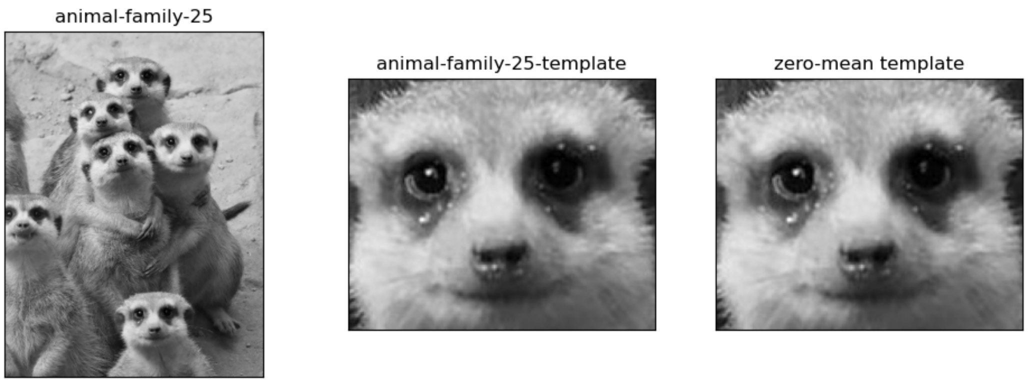

Figure 5: Original image, template image and zero-mean template

The third step is to implement cross-correlation (code A.13) with the original image and the zero-mean template image as mask, and we need to apply the procedure "ImageAdjust" (code A.14) to the correlation image, so that maxima show values closer to 255.

According to the professor's instructions, when the threshold is set to 150, peaks can be generated for the whole family. Finally, we overlay the peaks onto the original image for checking, as shown in figure 6.

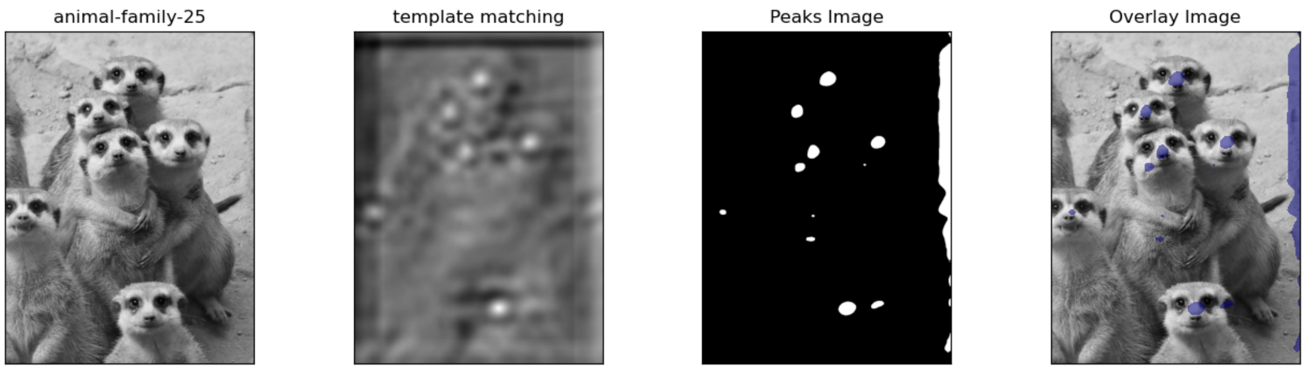

Figure 6: Template matching and peaks image

In addition, we have done some experiment with higher thresholds. Here we set $thresholds = [150, 200, 220, 250]$, and the peaks images are shown in figure 7.

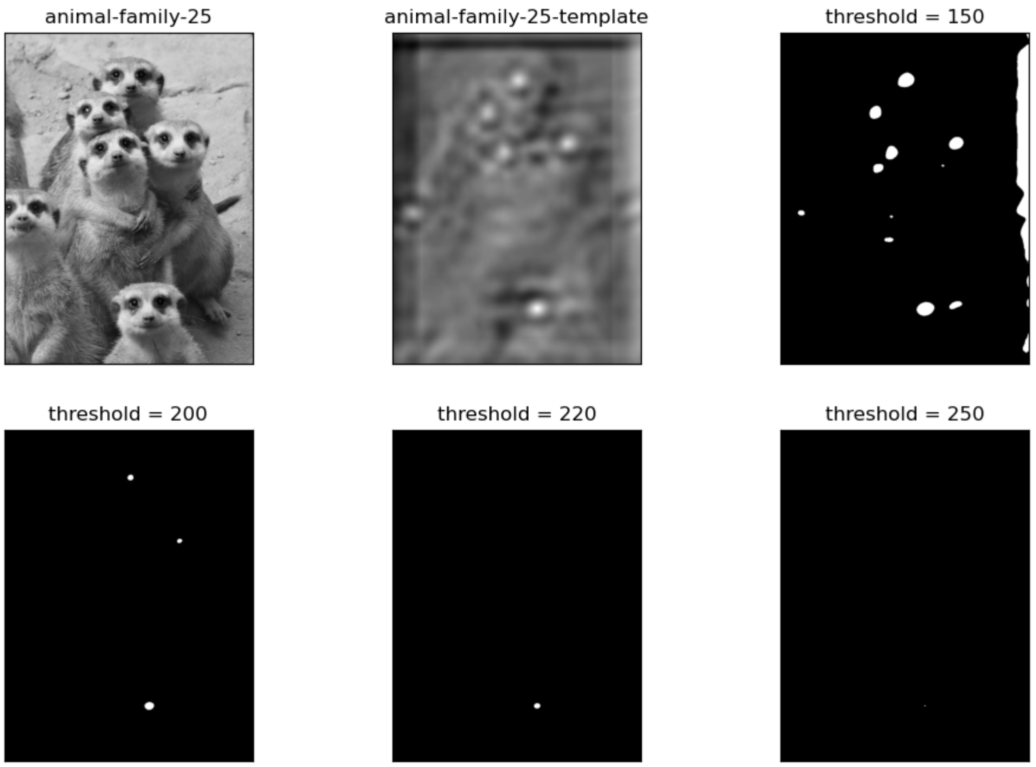

Figure 7: Peaks image with higher thresholds

By varying the threshold, we found that the accuracy of detected matches varied. A lower threshold will produce more matches, but more errors; a higher threshold will reduce the number of matches, resulting in more accurate matches.

3 Create part (Bonus)

Counting cars in a car manufacturer's lot link. A manufacturer may want to know how many cars of a specific type are parked on its parking lot. Here we use template matching algorithm A.17 and get the result:

```
Total cars in the parking lot: 200
```

The image of counting cars is shown in figure 8:

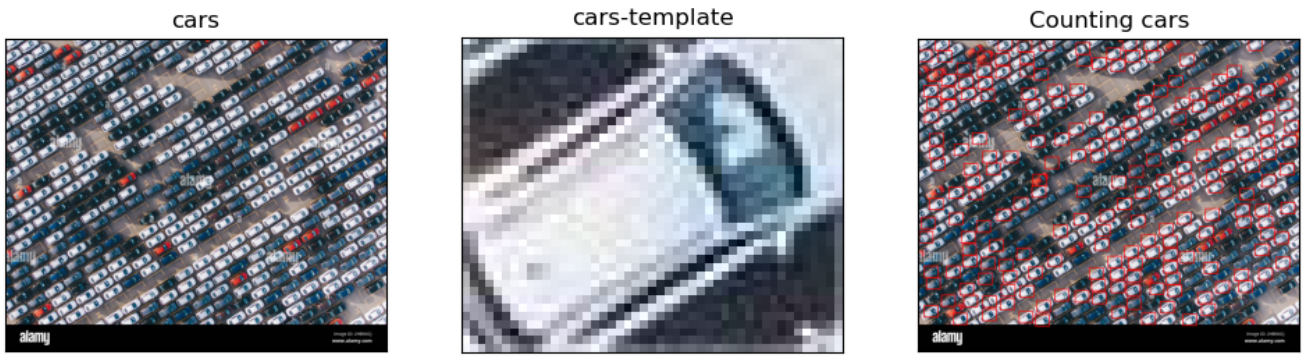

Figure 8: Counting cars

By matching specific white cars, we can find that most of the white cars have been recognized, but there are still some white cars that have not been recognized, and a small number of black cars have been recognized.

A Appendix Code

A.1 Convolution

```
def convolution(f, I):
    # Handle boundary of I, e.g. pad I according to size of f
    im_conv = np.zeros(shape=I.shape)
    if len(f.shape) == 1:
        if f.shape[0] > 1:
            f = np.flipud(f)
        else:
            f = np.fliplr(f)
    else:
        f = np.flipud(np.fliplr(f))
    if len(f.shape) == 1:
        f = f.reshape(1, len(f))
        hori_padding = f.shape[1] // 2
        vert_padding = 0
    else:
        if f.shape[1] > 1:
            hori_padding = f.shape[1] // 2
        else:
            hori_padding = 0
        vert_padding = f.shape[0] // 2
    padded_image = np.zeros(shape=(I.shape[0] + 2*vert_padding,
                                   I.shape[1] + 2*hori_padding))
    padded_image[vert_padding:padded_image.shape[0]-vert_padding,
                 hori_padding:padded_image.shape[1]-hori_padding]=I
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            section = padded_image[i:i+f.shape[0], j:j+f.shape[1]]
            im_conv[i,j] = np.sum(section*f)
    # Compute im_conv = f*I
    return im_conv
```

A.2 Create a 1D Gaussian Filter

```
def gaussian_1d(sigma, width):
    # Generate a 1D Gaussian filter.
    kernel = np.zeros(width)
    center = width // 2
    for x in range(width):
        kernel[x] = np.exp(-((x - center) ** 2) / (2 * sigma ** 2))
    kernel /= np.sum(kernel)
    return kernel
```

A.3 Create a Gaussian Filter

```
def gaussian_filter(image, sigma, size):
    # Apply Gaussian filter to the image using separable convolution.
    # Generate 1D Gaussian kernels
    Gx = gaussian_1d(sigma, size)
    # Transpose for vertical convolution
    Gy = np.transpose([Gx])
    # Apply horizontal convolution
```

```python
    convolved_x = convolution(Gx, image)
    # Apply vertical convolution
    convolved_y = convolution(Gy, convolved_x)
    return convolved_y
```

A.4 Denoise the image with a Gaussian filter

```python
# Load image
images = ['cameraman.png', 'zebra.png']
# Parameters
sigma = 2.0
# Filter size based on sigma
size = int(6 * sigma) + 1

fig, axs = plt.subplots(2, 2, figsize=(6, 6))

for i in range(len(images)):
    image = cv2.imread(images[i], cv2.IMREAD_GRAYSCALE).astype(np.float32)
    axs[i, 0].imshow(image, cmap='gray')
    axs[i, 0].set_xticks([])
    axs[i, 0].set_yticks([])
    axs[i, 0].set_ylabel(images[i])
    # Apply Gaussian filter
    denoised_image = gaussian_filter(image, sigma, size)
    # Convert back to byte [0, 255]
    denoised_image = np.clip(denoised_image, 0, 255).astype(np.uint8)
    axs[i, 1].imshow(denoised_image, cmap='gray')
    axs[i, 1].set_xticks([])
    axs[i, 1].set_yticks([])

axs[0, 0].set_title("Original Image")
axs[0, 1].set_title("Denoised Image")
plt.show()
```

A.5 Compute derivative images with respect to x and y

```python
fx = np.array([[-1, 0, 1]], dtype=np.float32)
fy = np.transpose(fx)

fig, axs = plt.subplots(2, 3, figsize=(12, 8))

for i in range(len(images)):
    image = cv2.imread(images[i], cv2.IMREAD_GRAYSCALE).astype(np.float32)
    axs[i, 0].imshow(image, cmap='gray')
    axs[i, 0].set_xticks([])
    axs[i, 0].set_yticks([])
    axs[i, 0].set_ylabel(images[i])

    Gx = convolution(fx, image)
    axs[i, 1].imshow(Gx, cmap='gray')
    axs[i, 1].set_xticks([])
    axs[i, 1].set_yticks([])

    Gy = convolution(fy, image)
    axs[i, 2].imshow(Gy, cmap='gray')
    axs[i, 2].set_xticks([])
    axs[i, 2].set_yticks([])

axs[0, 0].set_title("Original Image")
axs[0, 1].set_title("Horizontal Derivative Image")
axs[0, 2].set_title("Vertical Derivative Image")
plt.show()
```

A.6 Compute the gradient magnitude

```
def gradient_magnitude(derivative_x, derivative_y):
    # Compute gradient magnitude image.
    gradient_magnitude = np.sqrt(derivative_x**2 + derivative_y**2)
    return gradient_magnitude
```

A.7 Manual threshold

```
def manual_threshold(im_in, threshold):
    # Threshold image with the threshold of your choice
    manual_thresh_img = np.zeros_like(im_in)
    manual_thresh_img[im_in > threshold] = 255
    return manual_thresh_img
```

A.8 Draw magnitude image and threshold magnitude image

```
threshold = 45

fig, axs = plt.subplots(2, 3, figsize=(12, 8))

for i in range(len(images)):
    image = cv2.imread(images[i], cv2.IMREAD_GRAYSCALE).astype(np.float32)
    axs[i, 0].imshow(image, cmap='gray')
    axs[i, 0].set_xticks([])
    axs[i, 0].set_yticks([])
    axs[i, 0].set_ylabel(images[i])

    Gx = convolution(fx, image)
    Gy = convolution(fy, image)
    G = gradient_magnitude(Gx, Gy)
    axs[i, 1].imshow(G, cmap='gray')
    axs[i, 1].set_xticks([])
    axs[i, 1].set_yticks([])

    binary_edge_images = manual_threshold(G, threshold)
    axs[i, 2].imshow(binary_edge_images, cmap='gray')
    axs[i, 2].set_xticks([])
    axs[i, 2].set_yticks([])

axs[0, 0].set_title("Original Image")
axs[0, 1].set_title("Gradient Magnitude")
axs[0, 2].set_title("Binary Edge Image")
plt.show()
```

A.9 zero-crossing without delta

```
def zero_crossing(image):
    edge_image = np.zeros_like(image)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            center_value = image[i, j]
            if center_value > 0:
                # Check the 8 neighbors
                neighbors = image[i-1:i+2, j-1:j+2].flatten()
                # Exclude the center pixel
                neighbors = np.delete(neighbors, 4)
                # If any neighbor is negative, mark as edge.
                if any(n < 0 for n in neighbors):
                    edge_image[i, j] = 255
    return edge_image
```

A.10 zero-crossing with delta

```
def zero_crossing_with_delta(image, delta=0):
    edge_image = np.zeros_like(image)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            center_value = image[i, j]
            if center_value > 0:
                # Check the 8 neighbors
                neighbors = image[i-1:i+2, j-1:j+2].flatten()
                # Exclude the center pixel
                neighbors = np.delete(neighbors, 4)
                if any(n < 0 and center_value - n > delta for n in neighbors):
                    edge_image[i, j] = 255
    return edge_image
```

A.11 Draw edge image via zero-crossings of second derivative

```
# Load image
images = ['cameraman.png', 'zebra.png']

laplacian_filter = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])

# Parameters
sigma = 2.0
# Filter size based on sigma
size = int(6 * sigma) + 1

fig, axs = plt.subplots(2, 4, figsize=(12, 6))

for i in range(len(images)):
    image = cv2.imread(images[i], cv2.IMREAD_GRAYSCALE).astype(np.float32)

    # Apply Gaussian filter
    gaussian_smoothed_image = gaussian_filter(image, sigma, size)
    # Convert back to byte [0, 255]
    gaussian_smoothed_image = np.clip(gaussian_smoothed_image, 0, 255).astype(
       np.uint8)
    axs[i, 0].imshow(gaussian_smoothed_image, cmap='gray')
    axs[i, 0].set_xticks([])
    axs[i, 0].set_yticks([])
    axs[i, 0].set_ylabel(images[i])

    # Apply the Laplacian filter
    laplacian_image = convolution(laplacian_filter, gaussian_smoothed_image) #
       Assuming gaussian_blurred_image is already computed
    axs[i, 1].imshow(laplacian_image, cmap='gray')
    axs[i, 1].set_xticks([])
    axs[i, 1].set_yticks([])

    # Detect edges
    edge_image = zero_crossing(laplacian_image)
    axs[i, 2].imshow(edge_image, cmap='gray')
    axs[i, 2].set_xticks([])
    axs[i, 2].set_yticks([])

    # Detect edges Better!
    edge_image_with_data = zero_crossing_with_delta(laplacian_image, delta=5)
       # Adjust delta as needed
    axs[i, 3].imshow(edge_image_with_data, cmap='gray')
    axs[i, 3].set_xticks([])
    axs[i, 3].set_yticks([])


axs[0, 0].set_title("Gaussian-smoothed Image")
axs[0, 1].set_title("Laplacian-filtered Image")
axs[0, 2].set_title("Edge Image with thin lines")
axs[0, 3].set_title("Better")
plt.show()
```

A.12 Generate original image, template image and zero-mean template

```
image = cv2.imread('animal-family-25.jpg', 0)
image_template = cv2.imread('animal-family-25-template.jpg', 0)

fig, axs = plt.subplots(1, 3, figsize=(12, 4))

axs[0].imshow(image, cmap='gray')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[0].set_title('animal-family-25')
axs[1].imshow(image_template, cmap='gray')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[1].set_title('animal-family-25-template')

# Calculate min, max, and mean values
min_val = np.min(image_template)
max_val = np.max(image_template)
mean_val = np.mean(image_template)
print(f"Minimum value: {min_val}")
print(f"Maximum value: {max_val}")
print(f"Mean value: {mean_val}")
# Normalize the template image to have a zero mean
```

```
zero_mean_template = image_template - mean_val
# Verify the new mean is close to 0.0
new_mean_val = np.mean(zero_mean_template)
print(f"New mean value: {new_mean_val}")
axs[2].imshow(zero_mean_template, cmap='gray')
axs[2].set_xticks([])
axs[2].set_yticks([])
axs[2].set_title('zero-mean template')

plt.show()
```

A.13 Cross correlation

```
def cross_correlation(f, I):
    im_corr = np.zeros(shape=I.shape)

    if len(f.shape) == 1:
        f = f.reshape(1, len(f))
        hori_padding = f.shape[1] // 2
        vert_padding = 0
    else:
        if f.shape[1] > 1:
            hori_padding = f.shape[1] // 2
        else:
            hori_padding = 0
        vert_padding = f.shape[0] // 2

    padded_image = np.zeros(shape=(I.shape[0] + 2*vert_padding,
                                   I.shape[1] + 2*hori_padding))
    padded_image[vert_padding:padded_image.shape[0]-vert_padding,
                 hori_padding:padded_image.shape[1]-hori_padding]=I

    for i in range(I.shape[0]):
        for j in range(I.shape[1]):
            section = padded_image[i:i+f.shape[0], j:j+f.shape[1]]
            im_corr[i,j] = np.sum(section*f)

    return im_corr
```

A.14 Image adjust

```
def image_adjust(image):
    # Calculate the minimum and maximum pixel values of the image
    min_val = np.min(image)
    max_val = np.max(image)
    # Scale the image values to the [0, 255] range
    adjusted_image = 255 * (image - min_val) / (max_val - min_val)
    return adjusted_image.astype(np.uint8)
```

A.15 Template matching

```
fig, axs = plt.subplots(1, 4, figsize=(16, 4))

axs[0].imshow(image, cmap='gray')
```

```
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[0].set_title('animal-family-25')

template_matching = cross_correlation(zero_mean_template, image)
template_matching = image_adjust(template_matching)
axs[1].imshow(template_matching, cmap='gray')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[1].set_title('template matching')

threshold = 150
peaks_image = manual_threshold(template_matching, threshold)
axs[2].imshow(peaks_image, cmap='gray')
axs[2].set_xticks([])
axs[2].set_yticks([])
axs[2].set_title('Peaks Image')

axs[3].imshow(image, cmap='gray')
axs[3].set_xticks([])
axs[3].set_yticks([])
axs[3].set_title('Overlay Image')
axs[3].imshow(np.ma.masked_where(peaks_image == 0, peaks_image), cmap='jet',
    alpha=0.5, interpolation='nearest')

plt.show()
```

A.16 Experiment with high thresholds

```
thresholds = [150, 200, 220, 250]

fig, axs = plt.subplots(2, 3, figsize=(12, 8))

axs[0, 0].imshow(image, cmap='gray')
axs[0, 0].set_xticks([])
axs[0, 0].set_yticks([])
axs[0, 0].set_title('animal-family-25')
axs[0, 1].imshow(template_matching, cmap='gray')
axs[0, 1].set_xticks([])
axs[0, 1].set_yticks([])
axs[0, 1].set_title('animal-family-25-template')

peaks_image_150 = manual_threshold(template_matching, thresholds[0])
axs[0, 2].imshow(peaks_image_150, cmap='gray')
axs[0, 2].set_xticks([])
axs[0, 2].set_yticks([])
axs[0, 2].set_title('threshold = 150')

peaks_image_200 = manual_threshold(template_matching, thresholds[1])
axs[1, 0].imshow(peaks_image_200, cmap='gray')
axs[1, 0].set_xticks([])
axs[1, 0].set_yticks([])
axs[1, 0].set_title('threshold = 200')

peaks_image_220 = manual_threshold(template_matching, thresholds[2])
axs[1, 1].imshow(peaks_image_220, cmap='gray')
```

```
axs[1, 1].set_xticks([])
axs[1, 1].set_yticks([])
axs[1, 1].set_title('threshold = 220')

peaks_image_250 = manual_threshold(template_matching, thresholds[3])
axs[1, 2].imshow(peaks_image_250, cmap='gray')
axs[1, 2].set_xticks([])
axs[1, 2].set_yticks([])
axs[1, 2].set_title('threshold = 250')

plt.show()
```

A.17 Counting cars

```
image = cv2.imread('cars.jpg')
template = cv2.imread('cars-template.jpg')

fig, axs = plt.subplots(1, 3, figsize=(12, 4))
axs[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[0].set_title('cars')
axs[1].imshow(cv2.cvtColor(template, cv2.COLOR_BGR2RGB))
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[1].set_title('cars-template')

res = cv2.matchTemplate(image, template, cv2.TM_CCOEFF_NORMED)
threshold = 0.5
loc = np.where(res >= threshold)
processed_points = set()

(h, w) = template.shape[:2]
for pt in zip(*loc[::-1]):
    # Check if this point is close to others already processed
    if any(abs(pt[0] - processed_pt[0]) <= w and abs(pt[1] - processed_pt[1])
        <= h for processed_pt in processed_points):
         continue
    cv2.rectangle(image, pt, (pt[0] + w, pt[1] + h), (0, 0, 255), 2)
    processed_points.add(pt)

image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
print("Total cars in the parking lot:", len(processed_points))

axs[2].imshow(image_rgb)
axs[2].set_xticks([])
axs[2].set_yticks([])
axs[2].set_title('Counting cars')
plt.show()
```