Coursework 1 - Report

Iva Jorgusheska, UID: 11114620

February 21, 2025

1 Task 1

1.1 Implement python code for image padding

The padding function takes an image as input and adds a specified number of pixels to each side. Given a padding value of n, the function increases both the width and height of the image by 2n, as n pixels are added symmetrically to all four edges. This ensures that the image maintains its central alignment while expanding in size. As required in the task, I pad the image with zeros.

1. Implement python code for image padding

```python
def pad_image(image, pad_width):
    """Pads the image with zeros on all sides."""
    h, w = image.shape
    padded = np.zeros((h + 2 * pad_width, w + 2 * pad_width), dtype=np.uint8)
    padded[pad_width:h + pad_width, pad_width:w + pad_width] = image
    return padded
```

Figure 1: Padding the image with 0 values

This is the result of the image padded with input 1 as padding needed when we use 3x3 kernel.

Since it can be hard to see the padding when the value is 1, I additionally provide here the image padded with 10 pixels on each side just to show the function 'padding' works, and this image is not used for the future processing.

Figure 2: Padding 1 pixel on each side

Figure 3: Padding 10 pixels on each side

1.2 Implement a convolution function between an image and a nxn structural element

I have implemented convolution function that takes image and kernel of any dimensions. The convolution operation slides the kernel over the non padded region of the image. Since we only extract valid n×n regions (3x3 in this case) from the padded image, the final output remains the same size as the original image.

```python
def apply_convolution(image, kernel):
    """Perform convolution using a manually written loop."""
    kernel_size = kernel.shape[0]
    pad = kernel_size // 2  # To keep output size same as input
    padded_image = pad_image(image, pad)
    output = np.zeros_like(image, dtype=np.float32)

    img_height, img_width = image.shape

    for i in range(img_height):
        for j in range(img_width):
            # Extract 3x3 region
            region = padded_image[i:i + kernel_size, j:j + kernel_size]
            # Apply convolution (element-wise multiplication + sum)
            output[i, j] = np.sum(region * kernel)

    return np.clip(output, 0, 255).astype(np.uint8)  # Clip values to valid range
```

Figure 4: Convolution function

I have applied convolution to the image with both average and weighted kernels as requested.

```python
# Define kernels
average_kernel = np.ones((3, 3)) / 9

weighted_kernel = np.array([[1, 2, 1],
                            [2, 4, 2],
                            [1, 2, 1]]) / 16 #Gaussian

# Apply smoothing
smoothed_avg = apply_convolution(image, average_kernel)
smoothed_weighted = apply_convolution(image, weighted_kernel)
```

Figure 5: Filters used to smooth the image

The results I obtained when using these smoothing kernels.

We can notice that the edges are better preserved using the weighted kernel since it gives more significance to the pixel in the middle, and preserves the main characteristics. We can see that the most in the thing the kitten plays with, and also the face of the kitten does not appear that blurred. However, the difference for this weighted kernel with this kernel size does not appear that significant for now.

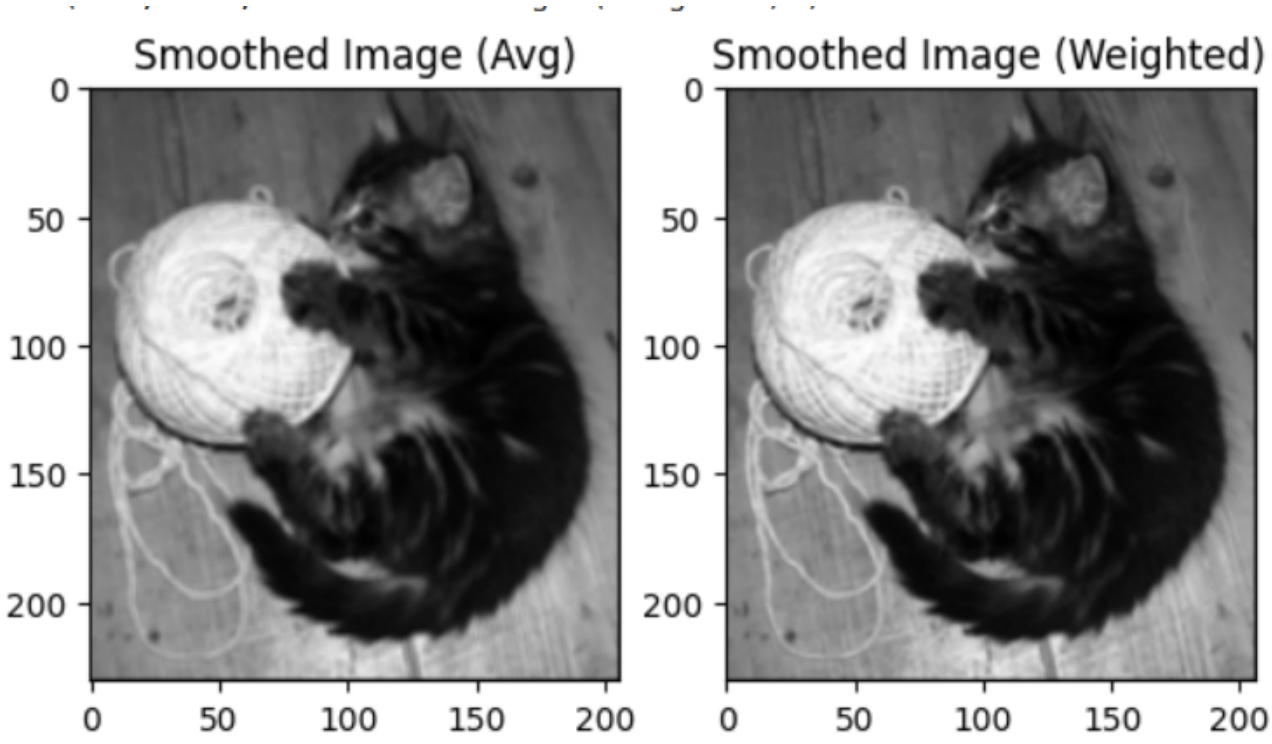


Figure 6: Smoothed images

2 Task 2 - Compute the horizontal and vertical gradient images, and then find the edge strength image given by the gradient magnitude (combined image)

Differentiating before smoothing only amplifies noise, so that is why for calculating the horizontal and vertical gradients I first smooth the image. In this section I will first upload the functions I have written so it is easier for the marker, and then I will upload the results I got for different filtering (smoothing) strategies.

2.1 Calculate Horizontal and Vertical gradient Images and combine to find gradient magnitude image

In this function I use Sobel-like filters to calculate both the x and y gradient images and then combine them together to find the edge strength image. I use the formula provided in the lectures for calculating the edge strength and I then clip the values so they are between 0 and 255. I also tried Prewitt filters, but I use Sobel filters instead since it is more robust to noise and yielded better result.

```python
] def compute_gradients(image):
    """Computes horizontal and vertical gradients using Sobel-like filters."""
    sobel_x = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]])

    sobel_y = np.array([[-1, -2, -1],
                        [ 0,  0,  0],
                        [ 1,  2,  1]])

    gradient_x = apply_convolution(image, sobel_x).astype(np.float32)
    gradient_y = apply_convolution(image, sobel_y).astype(np.float32)

    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
    gradient_magnitude = (gradient_magnitude / np.max(gradient_magnitude)) * 255

    return gradient_x.astype(np.uint8), gradient_y.astype(np.uint8), gradient_magnitude.astype(np.uint8)
```

Figure 7: Calculates Gradient Magnitude

The *edge strength* is given by the gradient magnitude:

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

The gradient direction is given by:

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right)$$

Figure 8: Formula for calculating edge strength and gradient direction

2.2 Calculated gradient images and gradient magnitude image for the image smoothed with 'averaging kernel'

2.3 Calculated gradient images and gradient magnitude image for the image smoothed with 'weighted kernel'

In the following part I have the results for the x-gradient image, y-gradient image and gradient magnitude image for both the one smoothed with 'averaging kernel' and one with 'weighted kernel'. Since the smoothed images differed only a bit, we would expect the same thing for the gradient images. We can notice that same as before, the edge strength image preserve more the edges in the worsted the kitten plays with and the edges in the kitten face are clearer.

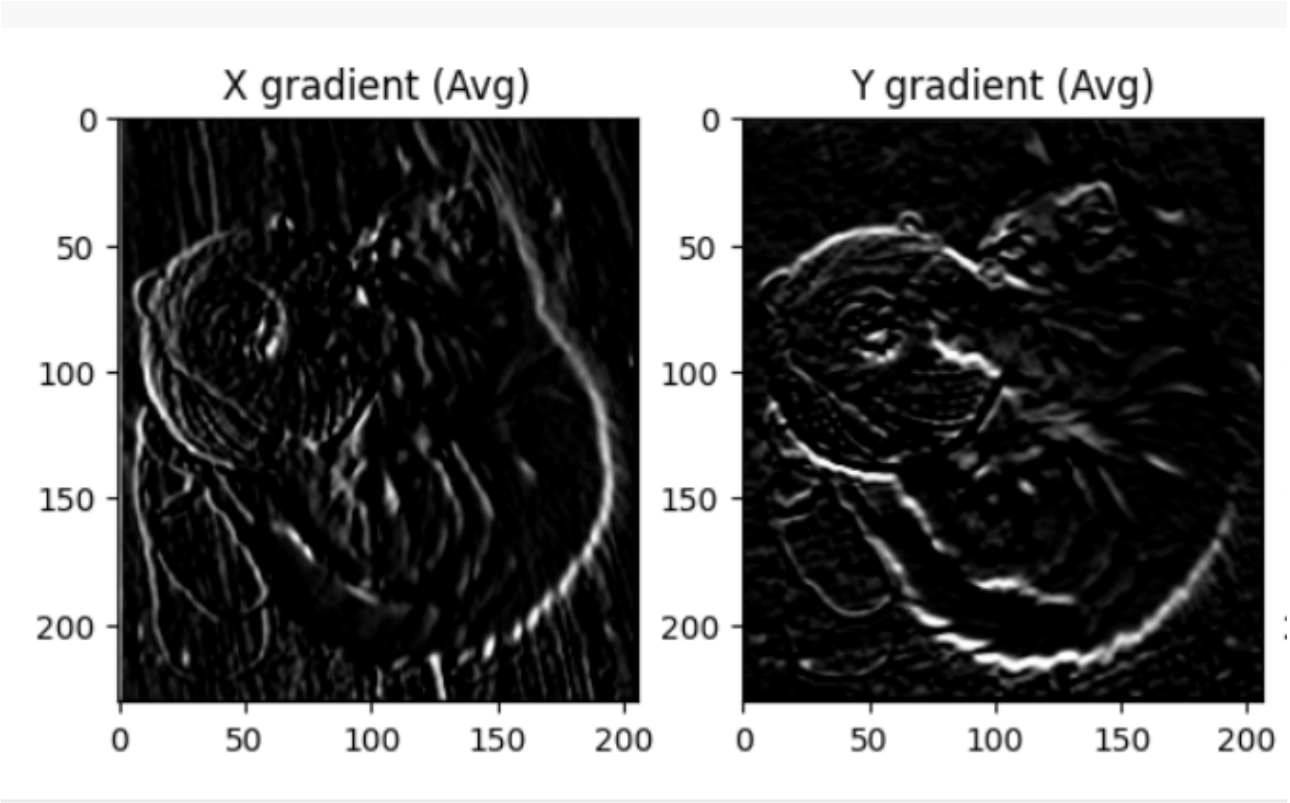


Figure 9: Calculating gradient images and edge strength image for the average smoothed image

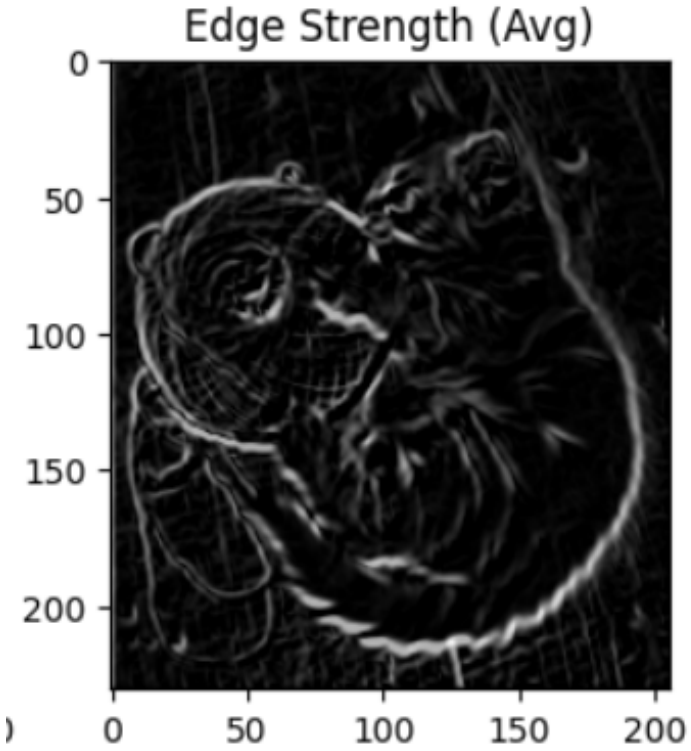


Figure 10: Gradient Magnitude Image (Average Kernel)

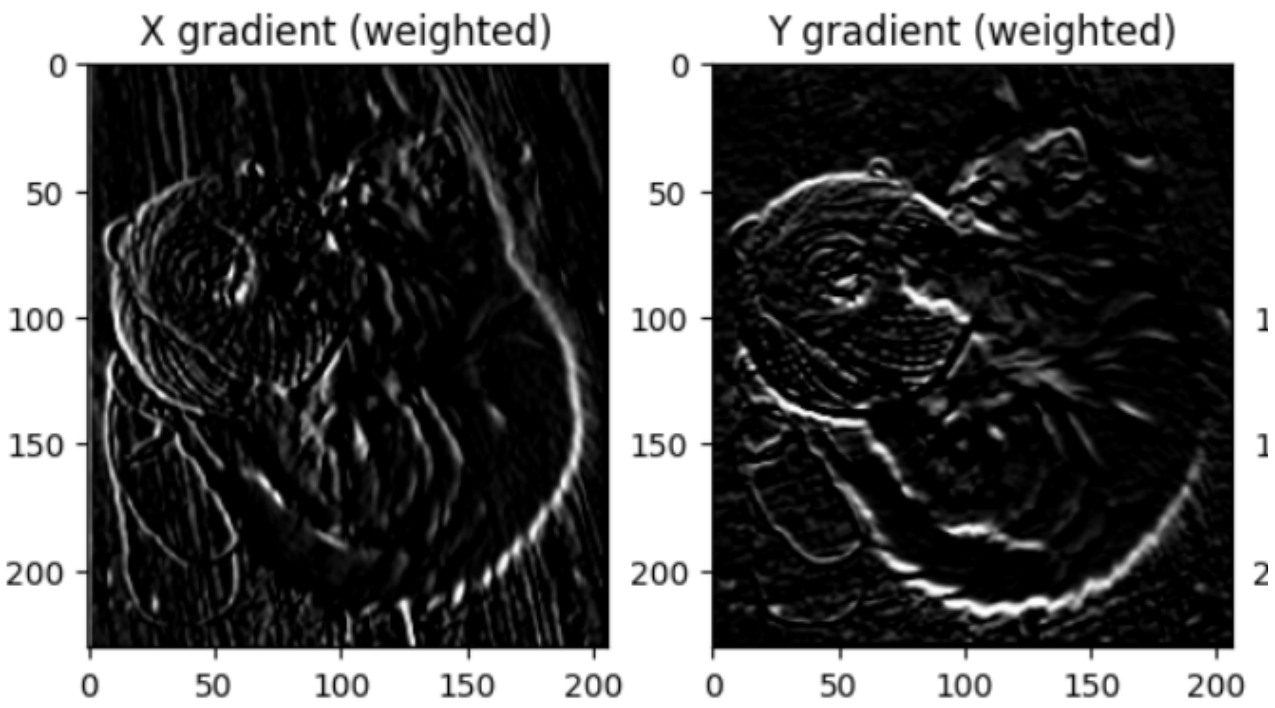


Figure 11: Calculating gradient images and edge strength image for the weighted smoothed image

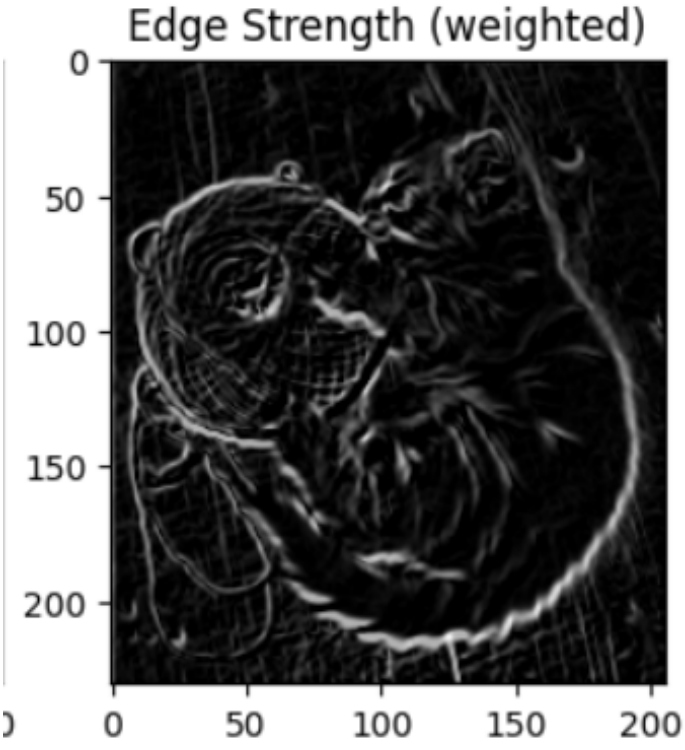


Figure 12: Gradient Magnitude Image (Weighted Kernel)

3 Task 3 - Thresholding

3.1 Implemented function that performs thresholding to find image edges

I will first show you the function I used to perform thresholding to find the edges of the image. I also plotted histogram to see the number of pixels at each intensity from 0 to 255, so I can choose a threshold value easier. However, it was not easy to determine a threshold value, and I think in this case there is not a one correct threshold value. Choosing higher one we may not get the full border of the outline of the cat, when choosing lower one we can get the outline but also additionally other not that significant edges we may not want.

```python
def threshold_edges(edge_image, threshold):
    """Applies thresholding to the edge strength image"""
    thresholded = np.zeros_like(edge_image, dtype=np.uint8)
    thresholded[edge_image >= threshold] = 255
    return thresholded

def plot_histogram(image):
    """Plots the histogram of the image to help determine the threshold value."""
    plt.figure()
    plt.hist(image.ravel(), bins=256, range=[0, 256], color='blue', alpha=0.7)
    plt.title("Histogram of Edge Strength Image")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
    plt.show()

# Plot histogram and determine threshold
plot_histogram(edge_strength)
threshold_value = 50 # Adjusted based on histogram
edge_detected = threshold_edges(edge_strength, threshold_value)
```

Figure 13: Thresholding function

I will now present the results obtained by applying thresholding to both the gradient images derived from the average and weighted smoothing kernels, using the same threshold value - 45. I chose this value since it is the one that has the best trade-off between preserving the kitten's outline and not showing the edges in the fur or the wood backgorund, as asked.

We can see that when we use the weighted kernels the edges of the worsted the cat is playing with, or the ears of the cat are better preserved. This is because the weighted kernel places more emphasis on the central pixel while still considering surrounding pixels, preserving important structural details while reducing noise. However, with this kernel size, and sigma value, we do not really achieve much different results than when we simply average the pixel values. We do not achieve a clean outline of the cat alone; instead, finer details, such as fur texture, the play structure, and the wood in the background, also appear as detected edges. This occurs because edge detection responds to all intensity variations, and without sufficient smoothing, high-frequency details (such as fur patterns) are also detected alongside the main object contours.

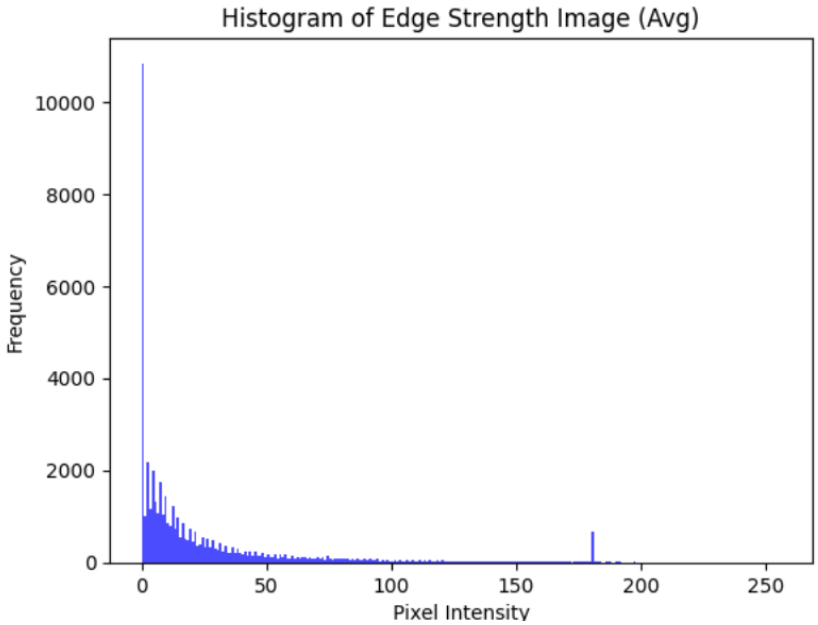


Figure 14: Avg image - histogram

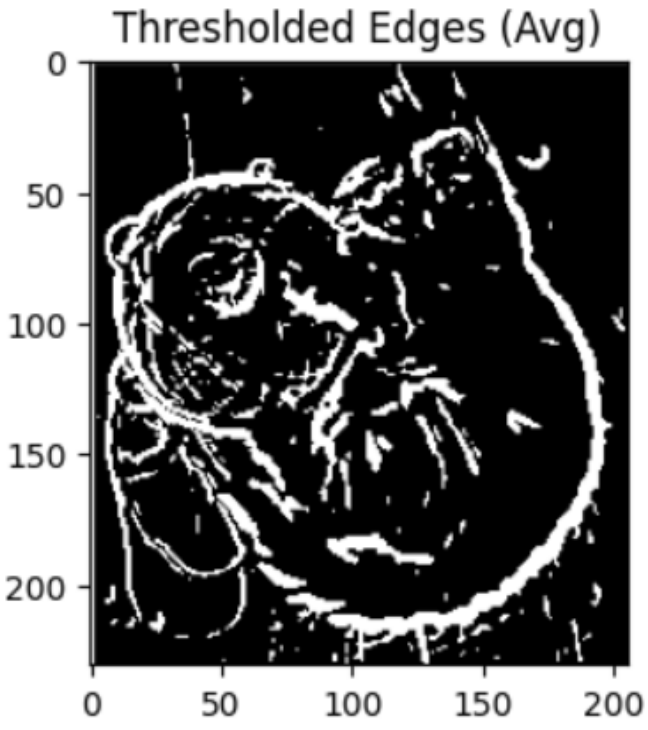


Figure 15: Avg - edges with threshold 45

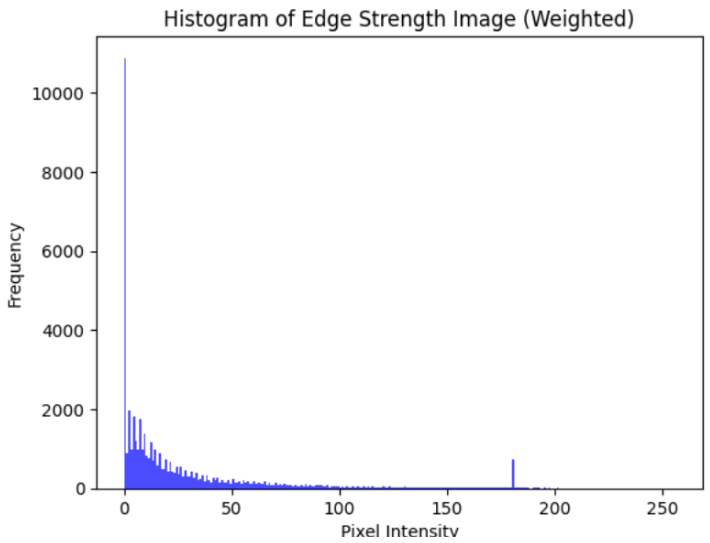


Figure 16: Weighted image - histogram

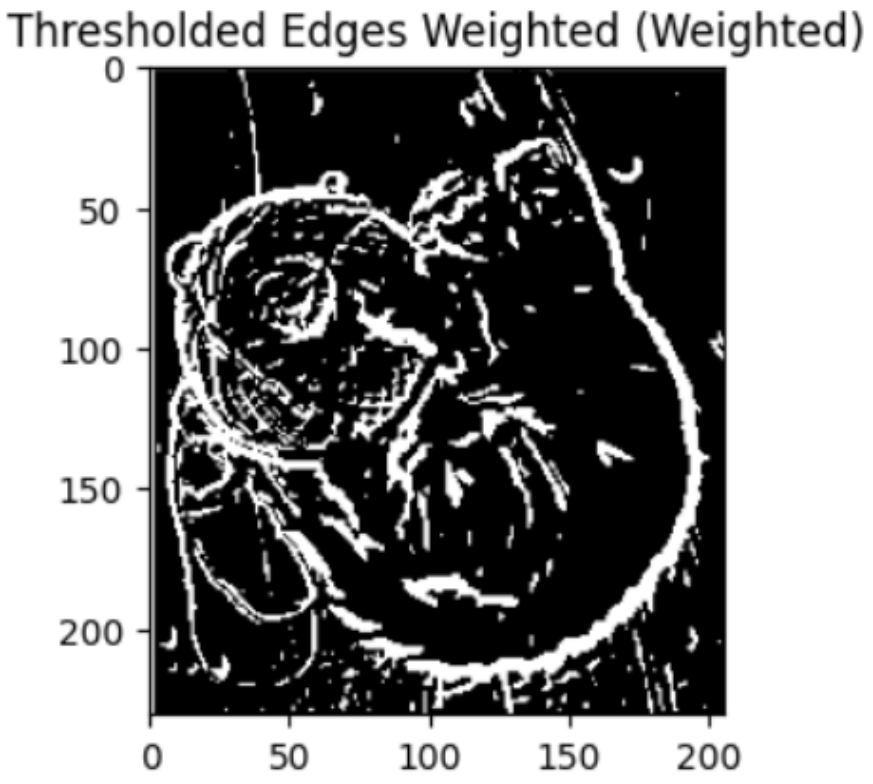


Figure 17: Weighted - edges with threshold 45

In order to see if we can get the edges we want depending on our task, for example, only the outline of the body, or fine detailed edges on the face, I will try different kernels.

I will explore the effect of Gaussian blurring with different window sizes (3, 5, 7, 11) and sigma values (0.5, 1, 2, 3) to see the effect. I used the following function to create Gaussian kernels with different size and sigma values.

```
# Experiment with different Gaussian kernels
kernel_sizes = [3, 5, 7, 11]
sigma_values = [0.5, 1, 2, 3]

fig, axes = plt.subplots(len(kernel_sizes), len(sigma_values), figsize=(12, 10))
for i, size in enumerate(kernel_sizes):
    for j, sigma in enumerate(sigma_values):
        kernel = create_gaussian_kernel(size, sigma)
        smoothed_image = apply_convolution(image, kernel)
        _, _, edge_strength = compute_gradients(smoothed_image)
        edge_output = apply_threshold(edge_strength, threshold=45)
        axes[i, j].imshow(edge_output, cmap='gray')
        axes[i, j].set_title(f"Size={size}, Sigma={sigma}")
        axes[i, j].axis('off')
plt.suptitle("Edge Detection with Threshold 55 after Gaussian Smoothing")
plt.tight_layout()
plt.show()
```

Figure 18: Weighted image - threshold 45

I got these results for the weighted gradient image.

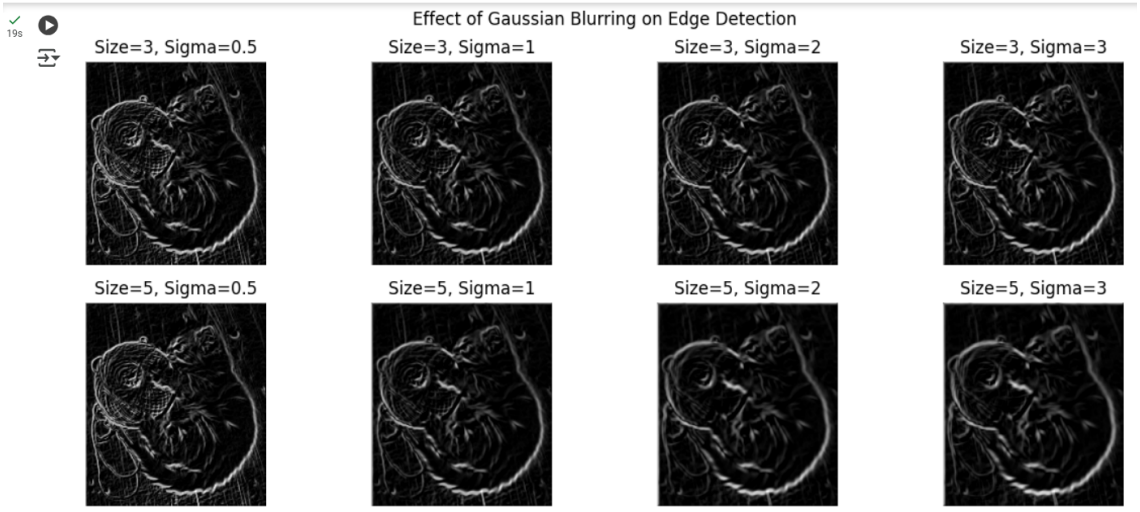


Figure 19: Weighted gradients with different sigma values and kernel size

.

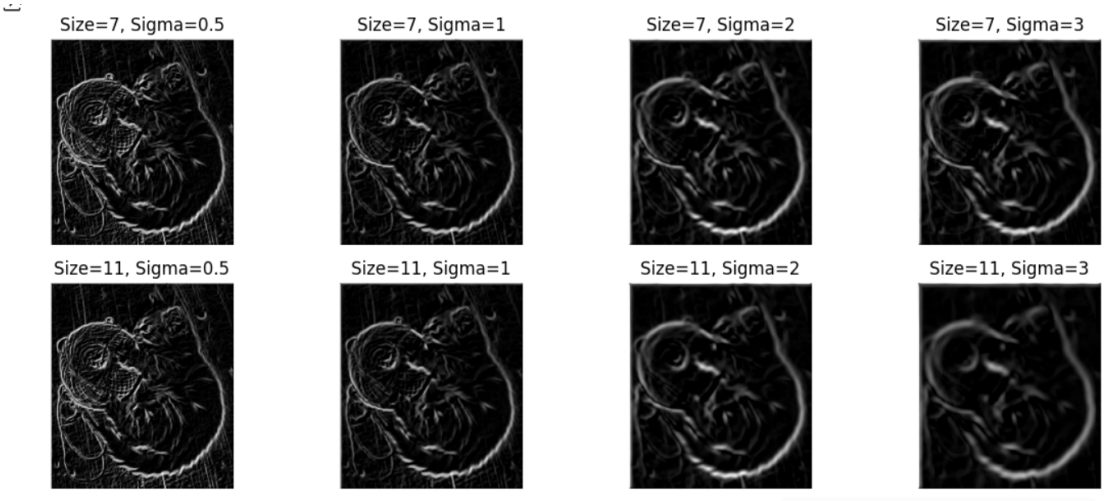


Figure 20: Weighted gradients with different sigma values and kernel size

As we can observe in the obtained gradient images, smaller window sizes and lower sigma values result in thinner edges, effectively capturing finer details in the image. This happens because a smaller Gaussian kernel applies less smoothing, preserving high-frequency components and allowing sharp transitions between intensity values to remain prominent. Conversely, with larger sigma values and bigger window sizes, the image becomes more blurred, leading to thicker edges. This is due to the fact that a larger Gaussian kernel averages pixel intensities over a wider neighborhood, reducing local variations and merging closely spaced edges. As a result, fine details are suppressed, and only the most dominant edges remain visible, appearing broader due to the gradual transition in intensity introduced by the increased smoothing effect. We can see that the sigma value plays a crucial role, since for the same window size with sigma = 0.5 we get really fine-grained thin edges, while for sigma=3, we get only the really thick outline of the cat's body. It is also important to notice that if the sigma value is big, then the window size should also be bigger otherwise the Gaussian filter may not 'fit' in the window size and we will not achieve the desired results.

Then for the weighted images I obtained, I thresholded them with value 45 as before.

.

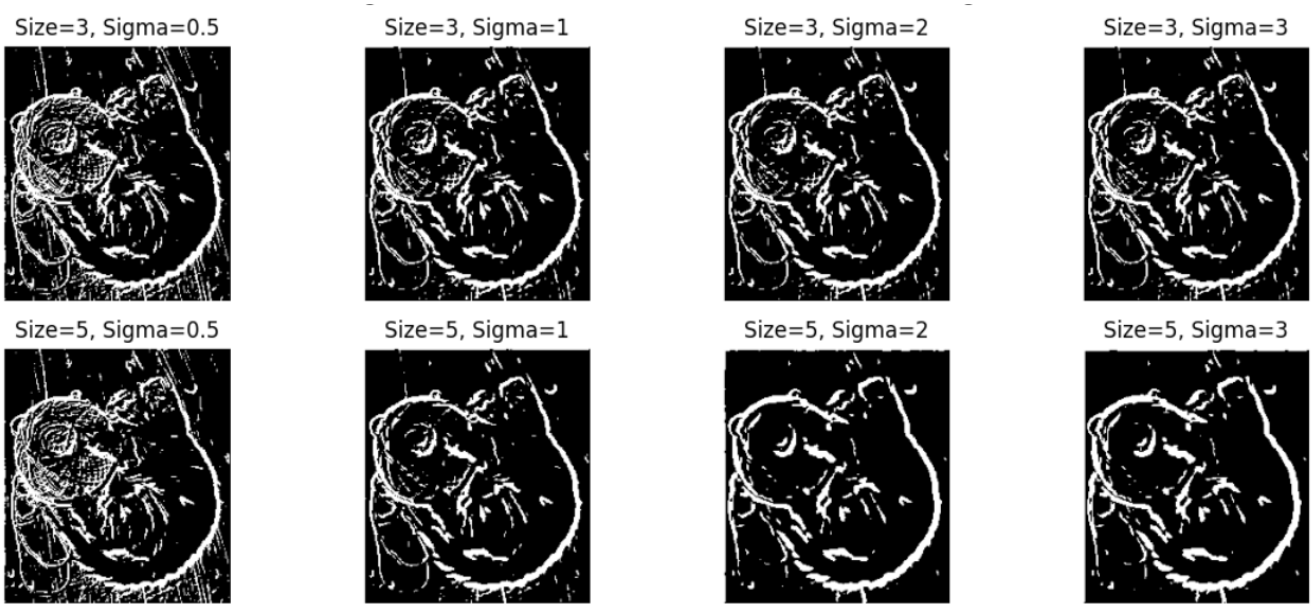


Figure 21: Thresholded at 45

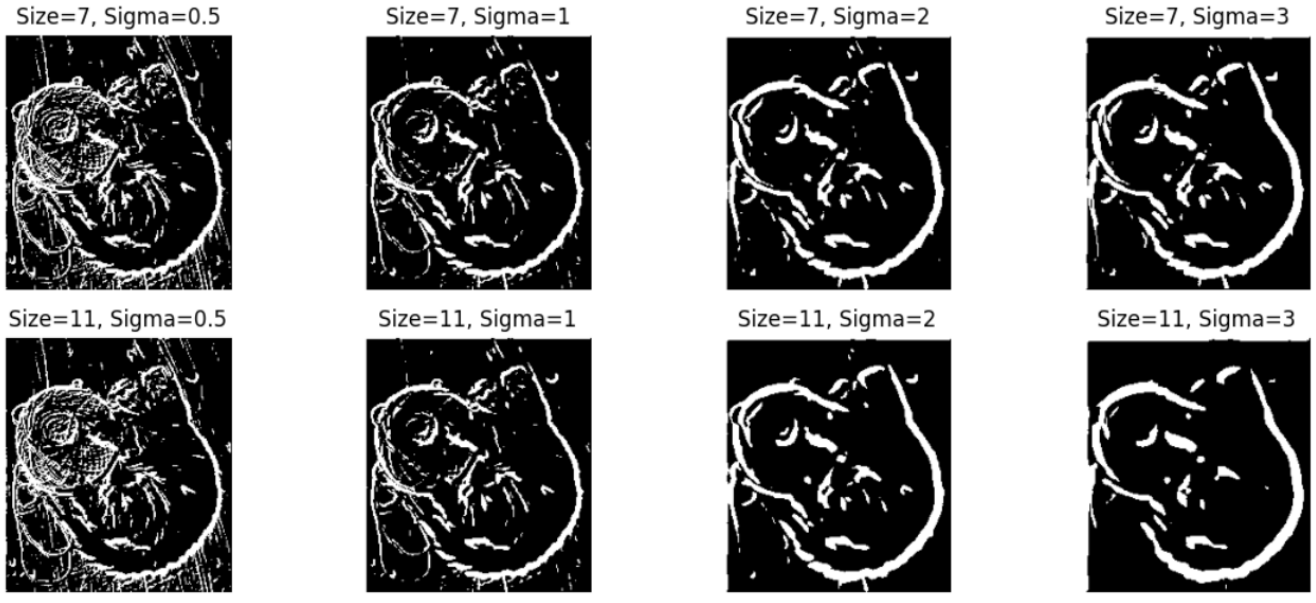


Figure 22: Thresholded at 45

When applying tresholding to the weighted gradient images, we observe that with smaller sigma values and smaller window sizes, the resulting edges capture finer details of the image. This occurs because less smoothing retains high-frequency components, preserving subtle textures such as the cat's fur. On the other hand, when using larger sigma values and bigger window sizes, the smoothing effect is more pronounced, reducing noise and minor details while emphasizing dominant edges. This allows us to extract the overall outline of the cat, which can be beneficial if our goal is to focus on its primary shape rather than finer textures.

In this case we can see that with the threshold values of 45, with size = 7 and sigma = 1, or with size=5 and sigma = 3, are some of the ones that extract only the cat's outline and not the fur or the wood.

With small window size, 3 in this case, we get almost the same result no matter the sigma value, since the window size is not big enough to be able to accommodate the sigma value.

3.2 Second derivative (local optima) - Laplacian of Gaussian (LoG) with Different Sigma Values and Kernel Sizes

We notice that the edges get thicker, we do not want to find the neighbours of the edge pixel,we want the exact location of the edge. We get this problem when using the first derivative. That problem we may get when we apply the first derivative, since the peak may be flat and we cannot find the exact location of the edge. That is why we can use the second derivative because it has 0-crossing. With 2D Laplacian kernels we directly get the second derivatives. Laplacian Kernel: negative center and positive surrounding, or reverse, but the sum of all values should be 0 so we do not have to normalize since it is differentiating kernel.

Convolving with this kernel will result in getting the second derivatives in which the edge positions are located by the 0 crossing. It is isotropic, giving same result to edges in any direction so we lose info about edge direction. We may be able to localize edges better, but we lose info about direction.

Results with Laplacian of Gaussian:

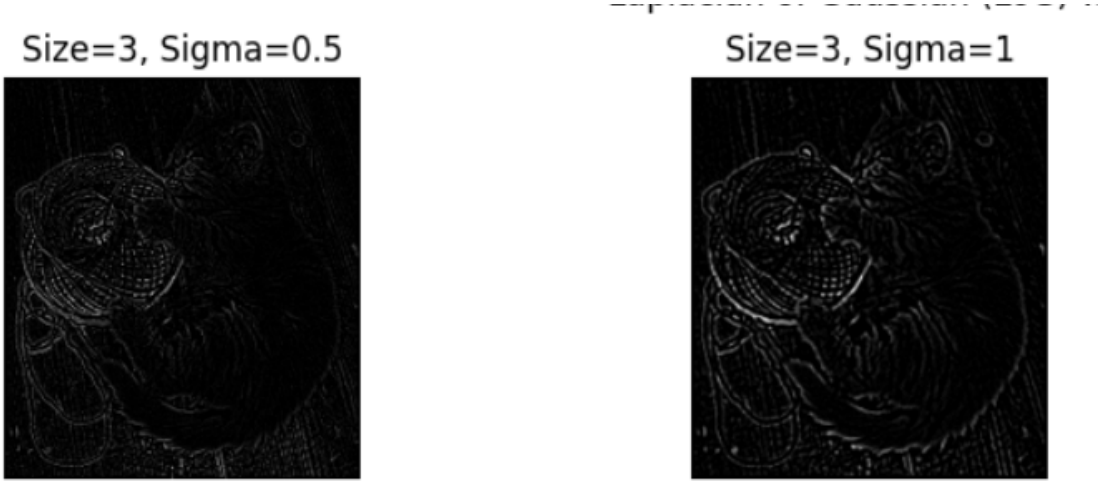


Figure 23:

.

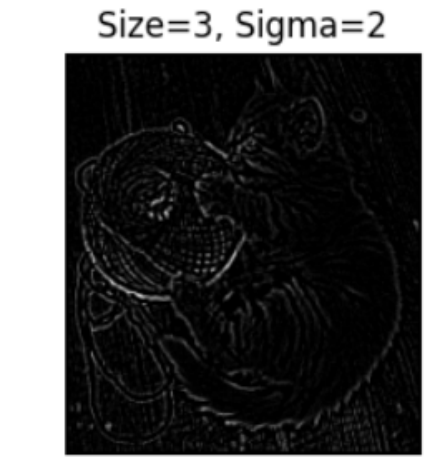


Figure 24:

Figure 25:

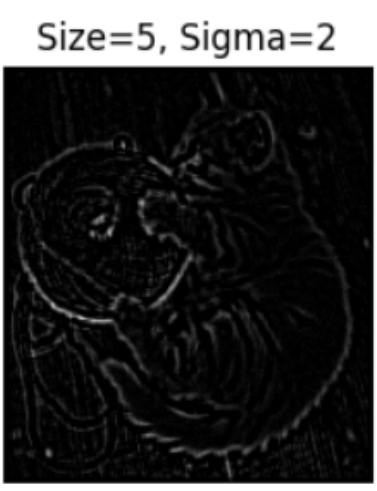


Figure 26:

Figure 27:

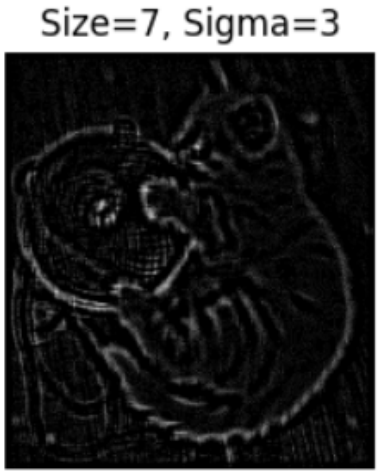
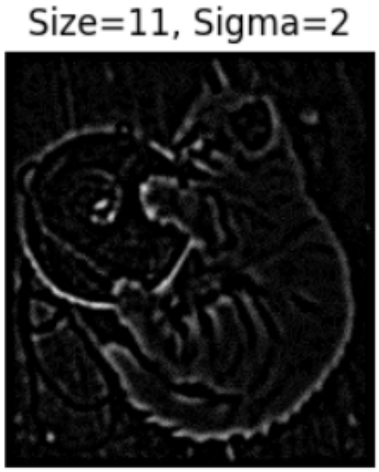


Figure 28:

Figure 29:



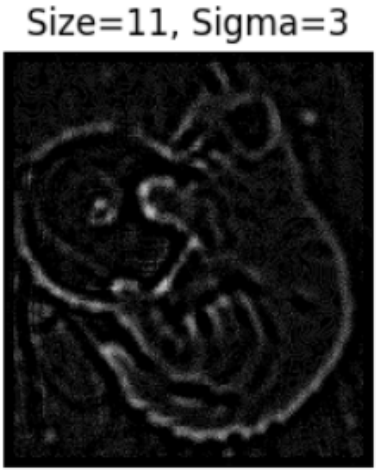


Figure 30:

We can see that with Laplacian of Gaussian we achieved the goal to get thinner edges, i.e. to get only the exact edge without the neighbouring pixels. In this way, with sigma values 2 and 3 we get a thinner outline of the cat's body, but it is a bit more blurred. The choice of sigma in the Laplacian of Gaussian (LoG) filter significantly impacts edge detection, particularly in how fine details and thin edges are preserved or lost. As we observed, smaller sigma value results in a Gaussian function with a narrow spread, meaning only slight blurring is applied before the Laplacian filter is used. This makes the filter highly sensitive to fine details and noise, but weak edges may not produce a strong enough response to be clearly detected, leading to some edges appearing incomplete, fragmented, or even missing. In contrast, a larger sigma value results in stronger blurring, which smooths out small details and noise, allowing only prominent and well-defined edges to remain visible. This makes edges appear thicker and more continuous, enhancing overall edge detection while reducing sensitivity to noise. As a result, choosing a small sigma is beneficial for capturing fine textures, while a larger sigma is better suited for detecting strong, well-defined edges. We see that the effect of bigger sigma values may not be visible in small window sizes, since they do not fit. In order to see their effect we need bigger window value as shown.

Summary

- **Smoothing with Average vs. Weighted Kernels:**
  - Weighted kernels preserve edges better because they assign more significance to the central pixel, reducing unnecessary blurring of important details.
  - The average kernel produces a more uniform blur, leading to loss of finer details, while the weighted kernel maintains structural integrity, especially in areas with high contrast.
  - The difference is evident in edge detection results—while both methods smooth the image, the weighted kernel retains sharper transitions, particularly in detailed regions.
- **Thresholding for Edge Extraction:**
  - Identifying the cat's outline is challenging due to similar gradient values in the fur and wooden floor, making edges hard to separate.

  - Adjusting sigma and kernel size in Gaussian filtering allows control over detail retention. Smaller values result in finer edges but retain noise, while larger values smooth out textures, highlighting dominant structures.
  - The right balance between sigma and kernel size determines whether subtle textures (like fur) are preserved or only the main object's shape is extracted.

- **Effect of Sigma and Window Size:**
  - A small sigma applies minimal blurring, capturing fine details but making the image more susceptible to noise and fragmented edges.
  - A large sigma smooths the image more aggressively, reducing noise and merging closely spaced edges, resulting in thicker and more continuous outlines.
  - The choice of sigma depends on whether fine textures or only strong edges are needed for analysis.

- **Second Derivative for Improved Edge Localization:**
  - The first derivative detects edge strength but may fail to pinpoint exact locations due to broad intensity peaks.
  - The Laplacian operator (second derivative) overcomes this by detecting zero-crossings, allowing precise localization of edges.
  - However, the Laplacian is isotropic, meaning it captures edges equally in all directions but does not retain orientation information, making it less effective for directional edge analysis.

Each method and parameter choice impacts the trade-off between detail preservation and noise reduction, highlighting the importance of fine-tuning for specific image processing goals.

Throughout the report I have provided more comments for each step that I have done.

A Source Code

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def pad_image(image, pad_width):
    """Pads the image with zeros on all sides."""
    h, w = image.shape
    padded = np.zeros((h + 2 * pad_width, w + 2 * pad_width), dtype=np.uint8)
    padded[pad_width:h + pad_width, pad_width:w + pad_width] = image
    return padded
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('kitty.bmp', cv2.IMREAD_GRAYSCALE)

# Apply padding (just to show it's working)
padded_image = pad_image(image, 1)  # Pad by 1 pixel for 3x3 kernel
padded_image2 = pad_image(image, 10)  # Pad by 10 pixels to makew it more obvious that
    the paddign function works

# Display the padded image
plt.figure(figsize=(5, 5))
plt.imshow(padded_image, cmap='gray')
plt.title("Padded Image (1 pixel on each side)")
plt.axis('off')

plt.figure(figsize=(5, 5))
plt.imshow(padded_image2, cmap='gray')
plt.title("Padded Image (10 pixels on each side)")
plt.axis('off')

plt.show()
```

```
32
33  def apply_convolution(image, kernel):
34      """Perform convolution using a manually written loop."""
35      kernel_size = kernel.shape[0]
36      pad = kernel_size // 2  # To keep output size same as input
37      padded_image = pad_image(image, pad)
38      output = np.zeros_like(image, dtype=np.float32)
39
40      img_height, img_width = image.shape
41
42      for i in range(img_height):
43          for j in range(img_width):
44              # Extract 3x3 region
45              region = padded_image[i:i + kernel_size, j:j + kernel_size]
46              # Apply convolution (element-wise multiplication + sum)
47              output[i, j] = np.sum(region * kernel)
48
49      return np.clip(output, 0, 255).astype(np.uint8)  # Clip values to valid range
50
51
52  # Define kernels
53  average_kernel = np.ones((3, 3)) / 9
54
55  weighted_kernel = np.array([[1, 2, 1],
56                              [2, 4, 2],
57                              [1, 2, 1]]) / 16 #Gaussian
58
59  # Apply smoothing
60  smoothed_avg = apply_convolution(image, average_kernel)
61  smoothed_weighted = apply_convolution(image, weighted_kernel)
62
63  # Save results
64  cv2.imwrite("smoothed_avg.png", smoothed_avg)
65  cv2.imwrite("smoothed_weighted.png", smoothed_weighted)
66
67
68   # Display results
69  plt.figure(figsize=(10, 5))
70  plt.subplot(1, 3, 1)
71  plt.imshow(smoothed_avg, cmap='gray')
72  plt.title("Smoothed␣Image␣(Avg)")
73
74  plt.subplot(1, 3, 2)
75  plt.imshow(smoothed_weighted, cmap='gray')
76  plt.title("Smoothed␣Image␣(Weighted)")
77
78
79  def compute_gradients(image):
80      """Computes horizontal and vertical gradients using Sobel-like filters."""
81      sobel_x = np.array([[-1, 0, 1],
82                          [-2, 0, 2],
83                          [-1, 0, 1]])
84
85      sobel_y = np.array([[-1, -2, -1],
86                          [ 0,  0,  0],
87                          [ 1,  2,  1]])
88
89      gradient_x = apply_convolution(image, sobel_x).astype(np.float32)
90      gradient_y = apply_convolution(image, sobel_y).astype(np.float32)
91
92      gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
93      gradient_magnitude = (gradient_magnitude / np.max(gradient_magnitude)) * 255
94
95      return gradient_x.astype(np.uint8), gradient_y.astype(np.uint8),
            gradient_magnitude.astype(np.uint8)
96
97
98
99  # Compute gradients
100 gradient_x, gradient_y, edge_strength = compute_gradients(smoothed_avg)
101
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 1)
plt.imshow(gradient_x, cmap='gray')
plt.title("X gradient (Avg)")

plt.subplot(1, 3, 2)
plt.imshow(gradient_y, cmap='gray')
plt.title("Y gradient (Avg)")

plt.subplot(1, 3, 3)
plt.imshow(edge_strength, cmap='gray')
plt.title("Edge Strength (Avg)")
plt.show()

def threshold_edges(edge_image, threshold):
    """Applies thresholding to the edge strength image"""
    thresholded = np.zeros_like(edge_image, dtype=np.uint8)
    thresholded[edge_image >= threshold] = 255
    return thresholded

def plot_histogram(image):
    """Plots the histogram of the image to help determine the threshold value."""
    plt.figure()
    plt.hist(image.ravel(), bins=256, range=[0, 256], color='blue', alpha=0.7)
    plt.title("Histogram of Edge Strength Image (Avg)")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
    plt.show()

# Plot histogram and determine threshold
plot_histogram(edge_strength)
threshold_value = 45 # Adjusted based on histogram
edge_detected = threshold_edges(edge_strength, threshold_value)


cv2.imwrite("edge_detected.png", edge_detected)

plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 3)
plt.imshow(edge_detected, cmap='gray')
plt.title("Thresholded Edges (Avg)")

plt.show()


gradient_x_weighted, gradient_y_weighted, edge_strength_weighted = compute_gradients(
    smoothed_weighted)

plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 1)
plt.imshow(gradient_x_weighted, cmap='gray')
plt.title("X gradient (weighted)")

plt.subplot(1, 3, 2)
plt.imshow(gradient_y_weighted, cmap='gray')
plt.title("Y gradient (weighted)")

plt.subplot(1, 3, 3)
plt.imshow(edge_strength_weighted, cmap='gray')
plt.title("Edge Strength (weighted)")
plt.show()


def threshold_edges(edge_image, threshold):
    """Applies thresholding to the edge strength image without OpenCV."""
    thresholded = np.zeros_like(edge_image, dtype=np.uint8)
    thresholded[edge_image >= threshold] = 255
    return thresholded

def plot_histogram(image):
    """Plots the histogram of the image to help determine the threshold value."""
```

```
    plt.figure()
    plt.hist(image.ravel(), bins=256, range=[0, 256], color='blue', alpha=0.7)
    plt.title("Histogram of Edge Strength Image (Weighted)")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
    plt.show()

# Plot histogram and determine threshold
plot_histogram(edge_strength_weighted)
threshold_value_weighted = 45  # Adjust based on histogram
edge_detected_weighted = threshold_edges(edge_strength_weighted,
    threshold_value_weighted)


cv2.imwrite("edge_detected.png", edge_detected_weighted)

plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 3)
plt.imshow(edge_detected_weighted, cmap='gray')
plt.title("Thresholded Edges Weighted (Weighted)")

plt.show()

import cv2
import numpy as np
import matplotlib.pyplot as plt

def create_gaussian_kernel(size, sigma):
    """Generates a Gaussian kernel of given size and sigma."""
    ax = np.linspace(-(size // 2), size // 2, size)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-(xx**2 + yy**2) / (2.0 * sigma**2))
    return kernel / np.sum(kernel)

def apply_convolution(image, kernel):
    """Performs convolution using a manually written loop."""
    kernel_size = kernel.shape[0]
    pad = kernel_size // 2  # Keep output size same as input
    padded_image = pad_image(image, pad)
    output = np.zeros_like(image, dtype=np.float32)

    img_height, img_width = image.shape

    for i in range(img_height):
        for j in range(img_width):
            # Extract region and apply convolution
            region = padded_image[i:i + kernel_size, j:j + kernel_size]
            output[i, j] = np.sum(region * kernel)

    return np.clip(output, 0, 255).astype(np.uint8)

def compute_gradients(image):
    """Computes horizontal and vertical gradients using Sobel filters."""
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    gradient_x = apply_convolution(image, sobel_x).astype(np.float32)
    gradient_y = apply_convolution(image, sobel_y).astype(np.float32)

    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
    gradient_magnitude = (gradient_magnitude / np.max(gradient_magnitude)) * 255

    return gradient_x.astype(np.uint8), gradient_y.astype(np.uint8),
        gradient_magnitude.astype(np.uint8)

# Load the image
image = cv2.imread('kitty.bmp', cv2.IMREAD_GRAYSCALE)

# Experiment with different Gaussian kernels
kernel_sizes = [3, 5, 7, 11]  # Increasing window sizes
sigma_values = [0.5, 1, 2, 3]  # Different blurring intensities
```

```python

fig, axes = plt.subplots(len(kernel_sizes), len(sigma_values), figsize=(12, 10))

for i, size in enumerate(kernel_sizes):
    for j, sigma in enumerate(sigma_values):
        kernel = create_gaussian_kernel(size, sigma)
        smoothed_image = apply_convolution(image, kernel)
        _, _, edge_strength = compute_gradients(smoothed_image)

        axes[i, j].imshow(edge_strength, cmap='gray')
        axes[i, j].set_title(f"Size={size}, Sigma={sigma}")
        axes[i, j].axis('off')

plt.suptitle("Effect of Gaussian Blurring on Edge Detection")
plt.tight_layout()
plt.show()


import cv2
import numpy as np
import matplotlib.pyplot as plt

def create_gaussian_kernel(size, sigma):
    """Generates a Gaussian kernel of given size and sigma."""
    ax = np.linspace(-(size // 2), size // 2, size)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-(xx**2 + yy**2) / (2.0 * sigma**2))
    return kernel / np.sum(kernel)

def apply_convolution(image, kernel):
    """Performs convolution using a manually written loop."""
    kernel_size = kernel.shape[0]
    pad = kernel_size // 2
    padded_image = pad_image(image, pad)
    output = np.zeros_like(image, dtype=np.float32)
    img_height, img_width = image.shape
    for i in range(img_height):
        for j in range(img_width):
            region = padded_image[i:i + kernel_size, j:j + kernel_size]
            output[i, j] = np.sum(region * kernel)
    return np.clip(output, 0, 255).astype(np.uint8)

def compute_gradients(image):
    """Computes horizontal and vertical gradients using Sobel filters."""
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    gradient_x = apply_convolution(image, sobel_x).astype(np.float32)
    gradient_y = apply_convolution(image, sobel_y).astype(np.float32)
    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
    gradient_magnitude = (gradient_magnitude / np.max(gradient_magnitude)) * 255
    return gradient_x.astype(np.uint8), gradient_y.astype(np.uint8),
        gradient_magnitude.astype(np.uint8)

def apply_threshold(image, threshold=55):
    """Applies a threshold to highlight strong edges."""
    return (image > threshold).astype(np.uint8) * 255

# Load the image
image = cv2.imread('kitty.bmp', cv2.IMREAD_GRAYSCALE)

# Experiment with different Gaussian kernels
kernel_sizes = [3, 5, 7, 11]
sigma_values = [0.5, 1, 2, 3]

fig, axes = plt.subplots(len(kernel_sizes), len(sigma_values), figsize=(12, 10))
for i, size in enumerate(kernel_sizes):
    for j, sigma in enumerate(sigma_values):
        kernel = create_gaussian_kernel(size, sigma)
        smoothed_image = apply_convolution(image, kernel)
        _, _, edge_strength = compute_gradients(smoothed_image)
        edge_output = apply_threshold(edge_strength, threshold=45)
```

```
        axes[i, j].imshow(edge_output, cmap='gray')
        axes[i, j].set_title(f"Size={size}, Sigma={sigma}")
        axes[i, j].axis('off')
plt.suptitle("Edge Detection with Threshold 55 after Gaussian Smoothing")
plt.tight_layout()
plt.show()


def laplacian_of_gaussian(image, size, sigma):
    """Applies Laplacian of Gaussian (LoG) edge detection."""
    gaussian_kernel = create_gaussian_kernel(size, sigma)
    smoothed_image = apply_convolution(image, gaussian_kernel)

    laplacian_kernel = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    log_image = apply_convolution(smoothed_image, laplacian_kernel)

    return log_image

# Load the image
image = cv2.imread('kitty.bmp', cv2.IMREAD_GRAYSCALE)

# Experiment with different sigma values and kernel sizes
sigma_values = [0.5, 1, 2, 3]
kernel_sizes = [3, 5, 7, 11]

fig, axes = plt.subplots(len(kernel_sizes), len(sigma_values), figsize=(15, 10))

for i, size in enumerate(kernel_sizes):
    for j, sigma in enumerate(sigma_values):
        log_result = laplacian_of_gaussian(image, size, sigma)
        axes[i, j].imshow(log_result, cmap='gray')
        axes[i, j].set_title(f'Size={size}, Sigma={sigma}')
        axes[i, j].axis('off')

plt.suptitle("Laplacian of Gaussian (LoG) with Different Sigma Values and Kernel Sizes
    ")
plt.tight_layout()
plt.show()
```