Computer Vision 2025 Assignment 1: Image filtering.

In this assignment, you will research, implement and test some image filtering operations. Image filtering by convolution is a fundamental step in many computer vision tasks and you will find it useful to have a firm grasp of how it works. For example, later in the course we will come across Convolutional Neural Networks (CNNs) which are built from convolutional image filters.

The main aims of the assignment are:

- to understand the basics of how images are stored and processed in memory;
- to gain exposure to several common image filters, and understand how they work;
- to get practical experience implementing convolutional image filters;
- to test your intuition about image filtering by running some experiments;
- to report your results in a clear and concise manner.

This assignment relates to the following ACS CBOK areas: abstraction, design, hardware and software, data and information, HCI and programming.

General instructions

Follow the instructions in this Python notebook and the accompanying file *a1code.py* to answer each question. It's your responsibility to make sure your answer to each question is clearly labelled and easy to understand. Note that most questions require some combination of Python code, graphical output, and text analysing or describing your results. Although we will check your code as needed, marks will be assigned based on the quality of your write up rather than for code correctness! This is not a programming test - we are more interested in your understanding of the topic.

Only a small amount of code is required to answer each question. We will make extensive use of the Python libraries

- numpy for mathematical functions
- skimage for image loading and processing
- matplotlib for displaying graphical results
- jupyter for Jupyter Notebooks

You should get familiar with the documentation for these libraries so that you can use them effectively.

The Questions

To get started, below is some setup code to import the libraries we need. You should not need to edit it.

In [1]:
```
# Numpy is the main package for scientific computing with Python.
import numpy as np

#from skimage import io

# Imports all the methods we define in the file a1code.py
from a1code import *

# Matplotlib is a useful plotting library for python
import matplotlib.pyplot as plt
# This code is to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python module
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in
%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

Question 0: Numpy warm up! (Not Assesed. This part is for you to understand the basic of numpy)

Before starting the assignment, make sure you have a working Python 3 installation, with up to date versions of the libraries mentioned above. If this is all new to you, I'd suggest downloading an all in one Python installation such as Anaconda. Alternatively you can use a Python package manager such as pip or conda, to get the libraries you need. If you're struggling with this please ask a question on the MyUni discussion forum.

For this assignment, you need some familiarity with numpy syntax. The numpy QuickStart should be enough to get you started:

https://numpy.org/doc/stable/user/quickstart.html

Here are a few warm up exercises to make sure you understand the basics. Answer them in the space below. Be sure to print the output of each question so we can see it!

1. Create a 1D numpy array Z with 12 elements. Fill with values 1 to 12.
2. Reshape Z into a 2D numpy array A with 3 rows and 4 columns.
3. Reshape Z into a 2D numpy array B with 4 rows and 3 columns.
4. Calculate the *matrix* product of A and B.
5. Calculate the *element wise* product of A and B^T (B transpose).

In [2]:
```
# array from 1 to 12
z = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
print(z)

# reshape z into a 3x4 matrix
a = z.reshape(3,4)
print(a)

# reshape z into a 4x3 matrix
b = z.reshape(4,3)
print(b)

# product of a and b
c = np.dot(a,b)
print(c)

# element-wise product of a and b
b_transpose = b.T
d = a*b_transpose
print(d)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 70  80  90]
 [158 184 210]
 [246 288 330]]
[[  1   8  21  40]
 [ 10  30  56  88]
 [ 27  60  99 144]]
```

You need to be comfortable with numpy arrays because that is how we store images. Let's do that next!

Question 1: Loading and displaying an image (10%)

Below is a function to display an image using the pyplot module in matplotlib. Implement the `load()` and `print_stats()` functions in a1code.py so that the following code loads the mandrill image, displays it and prints its height, width and channel.

In [3]:
```
def display(img, caption=''):
    # Show image using pyplot
    plt.figure()
    plt.imshow(img)
    plt.title(caption)
    plt.axis('off')
    plt.show()
```

In [4]:
```
image1 = load('images/whipbird.jpg')
```

```
display(image1, 'whipbird')

print_stats(image1)
```

whipbird

```
Height:  667
Width:  1000
Channels:  3
```

Return to this question after reading through the rest of the assignment. Find **at least 2 more images** to use as test cases in this assignment for all the following questions and display them below. Use your print_stats() function to display their height, width and number of channels. Explain *why* you have chosen each image.

In [5]:
```
### Your code to load and display your images here
image2 = load('images/cafe.jpg')
display(image2, 'cafe')
print_stats(image2)
```

cafe


```
Height:  2160
Width:  3840
Channels:  3
```

In [6]:
```
image3 = load('images/grayscale.jpg')
display(image3, 'grayscale')
print_stats(image3)
```

grayscale

```
Height:  1356
Width:  2048
Channels: 1
```

In [7]:
```
image4 = load('images/mandrill.jpg')
display(image4, 'mandrill')
print_stats(image4)
```

mandrill

```
Height:  300
Width:  300
Channels:  3
```

In [8]:
```
# clipping test image
image_clipped_1 = image1.copy()
image_clipped_2 = image1.copy()

image_clipped_1[:333,:333] = -1
image_clipped_2[:333,:333] = 256
display(image_clipped_1, 'whipbird with 1 pixel edited to -1')
display(image_clipped_2, 'whipbird with 1 pixel edited to 256')

# apply point process to pixels on single channel
image_processed = image1.copy()
image_processed[:333,:333,0] = 0

display(image_processed, 'whipbird with red channel turned to 0')
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-1.0..1.0].
```

whipbird with 1 pixel edited to -1

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..256.0].
```

whipbird with 1 pixel edited to 256

whipbird with red channel turned to 0

Your explanation of images here

Image choices explanation:

1. Whipbird

- This image has both sharp details and blurred areas, allow me to test how filters handle edges and smooth backgrounds. The bird and plants are in focus and sharp, while the soft background helps analyze effects of filters on blurred areas.

2. Cafe

- This image has bright neon lights and dark shadows, making it good for testing contrast adjustments and thresholding. The mix of bright and dark areas helps see how filters work with different colors and exposure.
- Additionally, this image contains many straight and horizontal edges, such as window frames, neon signs, and the building structure. These features make it ideal for testing edge detection filters.

3. Grayscale Fisherman

- This is a grayscale image, which have no colour (black and white only) and focuses only on textures and brightness levels. It's useful for testing edge detection and seeing how grayscale images behave with different filters.

4. Mandrill

- This image has strong colours and contrast, which makes it useful for testing colour filters and brightness adjustments.

- The detailed textures on the fur and face are helpful for seeing how edge detection and blurring filters affect fine details.

Experiment with clipping and single channel editing

- Negative Value Clipping: Setting pixel values to -1 demonstrated clipping; the value was clamped to the minimum valid intensity (0), visually resulting in a black pixel.
- Positive Value Clipping: Setting pixel values to 256, exceeding the standard maximum (255), also resulted in clipping; the value was clamped to the maximum, visually resulting in a white pixel.
- Channel Manipulation (RGB): Zeroing out the Red channel removed all red components from the image section pixels.
- Resulting Color Shift: Consequently, the image section appearance shifted towards greenish tones, as no red colors appears.

Question 2: Image processing (30%)

Now that you have an image stored as a numpy array, let's try some operations on it.

1. Implement the `crop()` function in a1code.py. Use array slicing to crop the image.
2. Implement the `resize()` function in a1code.py.
3. Implement the `change_contrast()` function in a1code.py.
4. Implement the `greyscale()` function in a1code.py.
5. Implement the `binary()` function in a1code.py.

What do you observe when you change the threshold of the binary function?

Apply all these functions with different parameters on your own test images.

In [9]:
```
# This should crop the bird from the  image; you will need to adjust the
crop_img = crop(image1, 75, 125, 400, 400)
display(crop_img, 'crop')
print_stats(crop_img)

resize_img = resize(crop_img, 500, 900)
display(resize_img, 'resize')
print_stats(resize_img)

contrast_img = change_contrast(image1, 0.5)
display(contrast_img, 'contrast 0.5')
print_stats(contrast_img)

contrast_img = change_contrast(image1, 1.5)
display(contrast_img, 'contrast 1.5')
print_stats(contrast_img)

grey_img = greyscale(image1)
display(grey_img, 'greyscale')
print_stats(grey_img)
```

```
binary_img = binary(grey_img, 0.3)
display(binary_img, 'binary 0.3')
print_stats(binary_img)

binary_img = binary(grey_img, 0.7)
display(binary_img, 'binary 0.7')
print_stats(binary_img)
```


```
Height:  400
Width:  400
Channels:  3
```

resize

```
Height:  500
Width:  900
Channels:  3
```

contrast 0.5

```
Height:  667
Width:  1000
Channels:  3
```

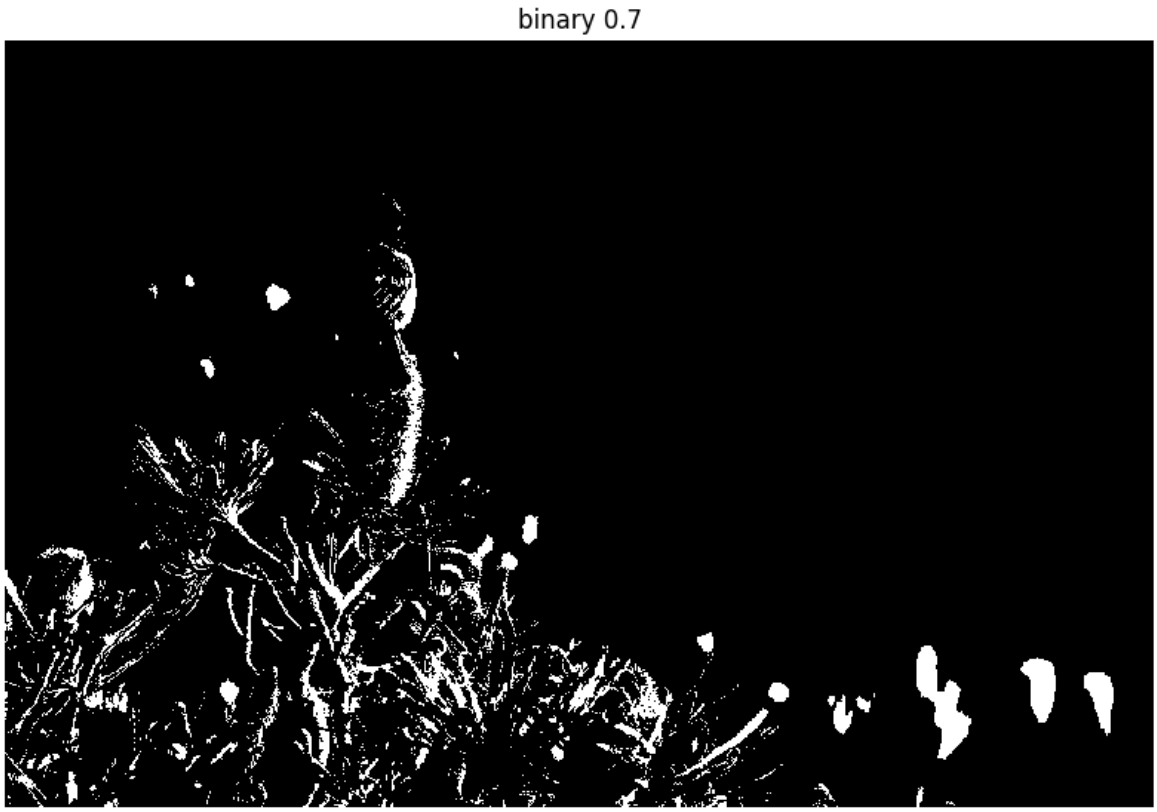
contrast 1.5

```
Height:  667
Width:  1000
Channels:  3
```

greyscale

```
Height:  667
Width:  1000
Channels: 1
```


```
Height:  667
Width:  1000
Channels: 1
```




```
Height:  667
Width:  1000
Channels: 1
```

In [10]:
```
# Cafe image
crop_img = crop(image2, 500, 2400, 800, 800)
display(crop_img, 'crop')
print_stats(crop_img)
```

```
resize_img = resize(crop_img, 1000, 400)
display(resize_img, 'resize')
print_stats(resize_img)

contrast_img = change_contrast(image2, 0.5)
display(contrast_img, 'contrast 0.5')
print_stats(contrast_img)

contrast_img = change_contrast(image2, 1.5)
display(contrast_img, 'contrast 1.5')
print_stats(contrast_img)

grey_img = greyscale(image2)
display(grey_img, 'greyscale')
print_stats(grey_img)

binary_img = binary(grey_img, 0.3)
display(binary_img, 'binary 0.3')
print_stats(binary_img)

binary_img = binary(grey_img, 0.7)
display(binary_img, 'binary 0.7')
print_stats(binary_img)
```

crop


```
Height:  800
Width:  800
Channels:  3
```

resize


```
Height:  1000
Width:  400
Channels:  3
```

contrast 0.5


```
Height:  2160
Width:  3840
Channels:  3
```

contrast 1.5


```
Height:  2160
Width:  3840
Channels:  3
```

greyscale


```
Height:  2160
Width:  3840
Channels: 1
```

binary 0.3

```
Height:  2160
Width:  3840
Channels: 1
```

binary 0.7


```
Height:  2160
Width:  3840
Channels: 1
```

In [11]:

```python
# Fisherman image
crop_img = crop(image3, 0, 0, 1000, 1000)
display(crop_img, 'crop')
print_stats(crop_img)

resize_img = resize(crop_img, 100, 100)
display(resize_img, 'resize')
print_stats(resize_img)

contrast_img = change_contrast(image3, 0.5)
display(contrast_img, 'contrast 0.5')
print_stats(contrast_img)

contrast_img = change_contrast(image3, 1.5)
display(contrast_img, 'contrast 1.5')
print_stats(contrast_img)

grey_img = greyscale(image3)
display(grey_img, 'greyscale')
print_stats(grey_img)

binary_img = binary(grey_img, 0.3)
display(binary_img, 'binary 0.3')
print_stats(binary_img)

binary_img = binary(grey_img, 0.7)
display(binary_img, 'binary 0.7')
print_stats(binary_img)
```

crop

```
Height:  1000
Width:  1000
Channels: 1
```


```
Height:  100
Width:  100
Channels:  3
```

contrast 0.5

```
Height:  1356
Width:  2048
Channels: 1
```

contrast 1.5

```
Height:  1356
Width:  2048
Channels: 1
```

greyscale

Height: 1356
Width: 2048
Channels: 1

binary 0.3

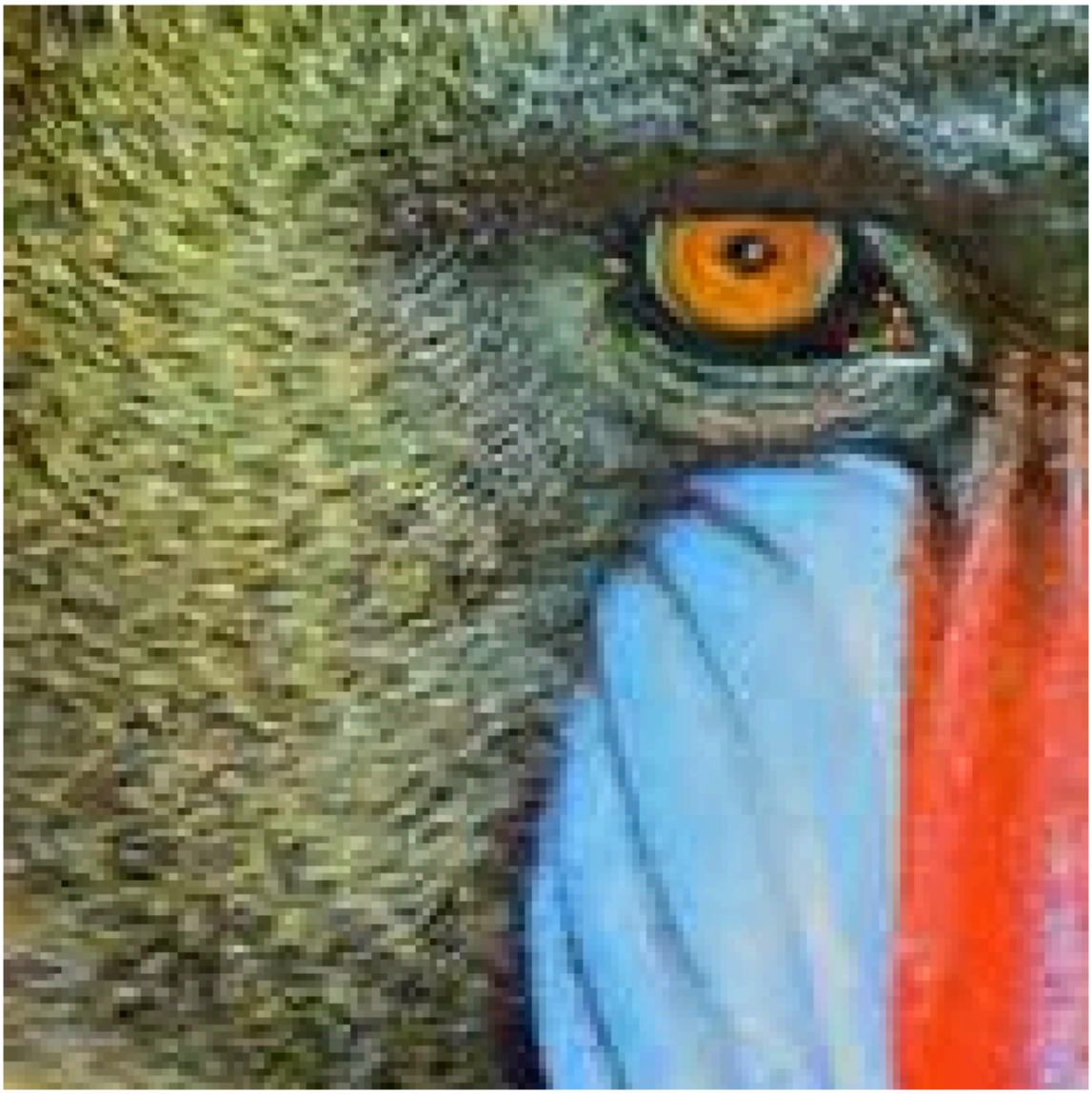
Height: 1356
Width: 2048
Channels: 1


```
Height:  1356
Width:  2048
Channels: 1
```

In [12]:

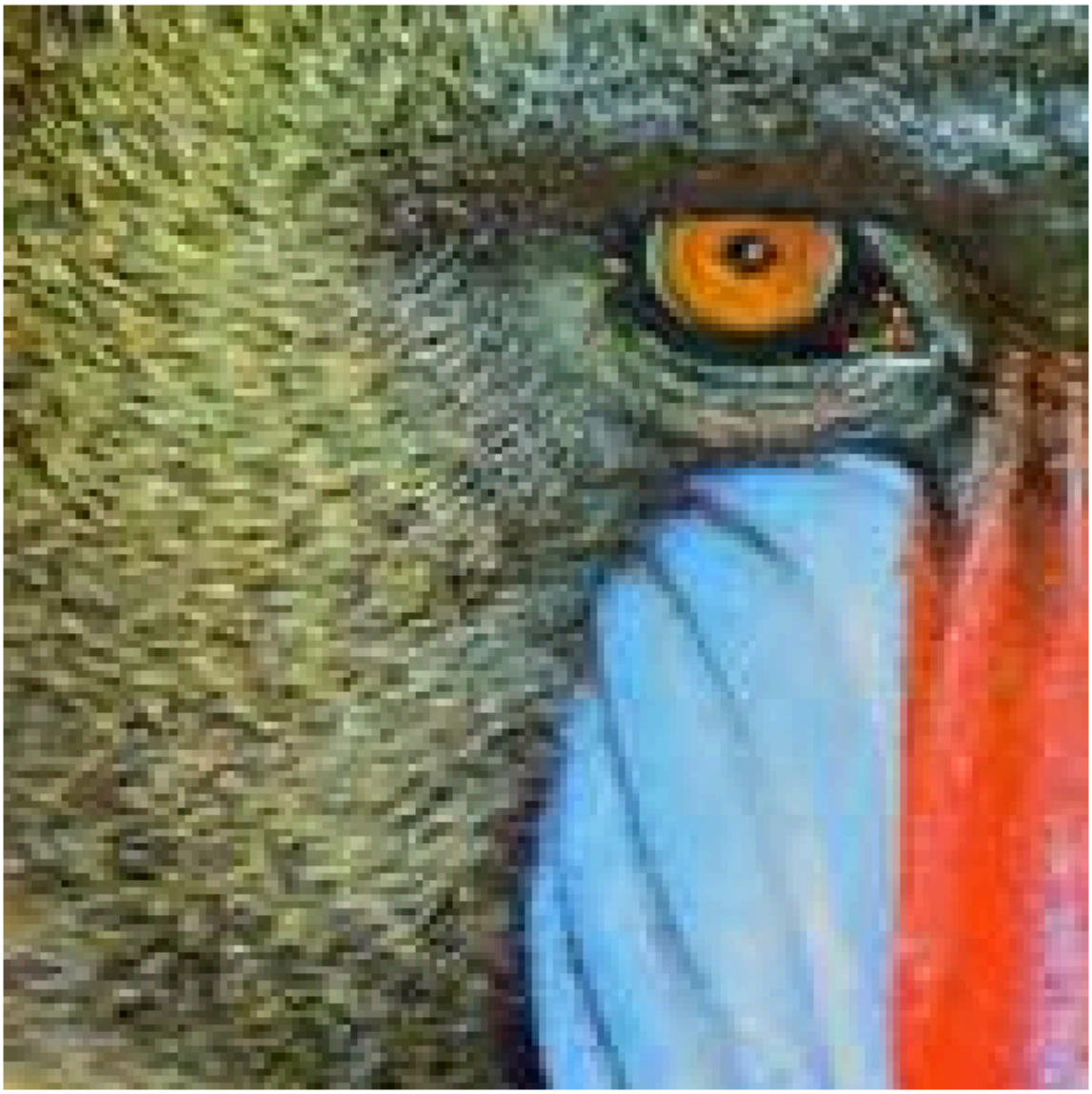
```
# Mandrill image
crop_img = crop(image4, 0, 0, 150, 150)
display(crop_img, 'crop')
print_stats(crop_img)

resize_img = resize(crop_img, 1500, 1500)
display(resize_img, 'resize')
print_stats(resize_img)

contrast_img = change_contrast(image4, 0.5)
display(contrast_img, 'contrast 0.5')
print_stats(contrast_img)

contrast_img = change_contrast(image4, 1.5)
display(contrast_img, 'contrast 1.5')
print_stats(contrast_img)

grey_img = greyscale(image4)
display(grey_img, 'greyscale')
print_stats(grey_img)

binary_img = binary(grey_img, 0.3)
display(binary_img, 'binary 0.3')
print_stats(binary_img)

binary_img = binary(grey_img, 0.7)
display(binary_img, 'binary 0.7')
print_stats(binary_img)
```

crop



```
Height:  150
Width:  150
Channels:  3
```

resize

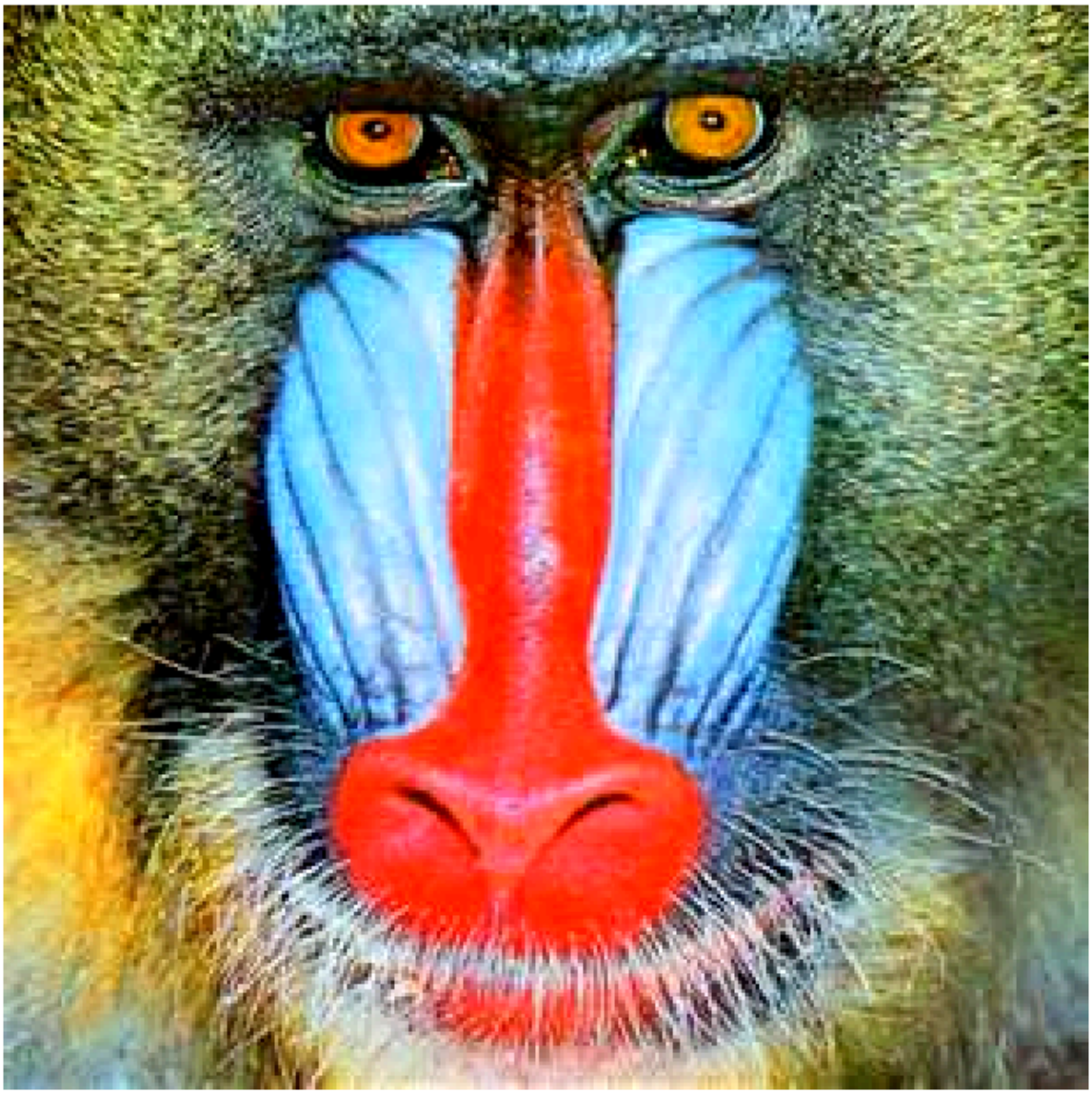


```
Height:  1500
Width:  1500
Channels:  3
```

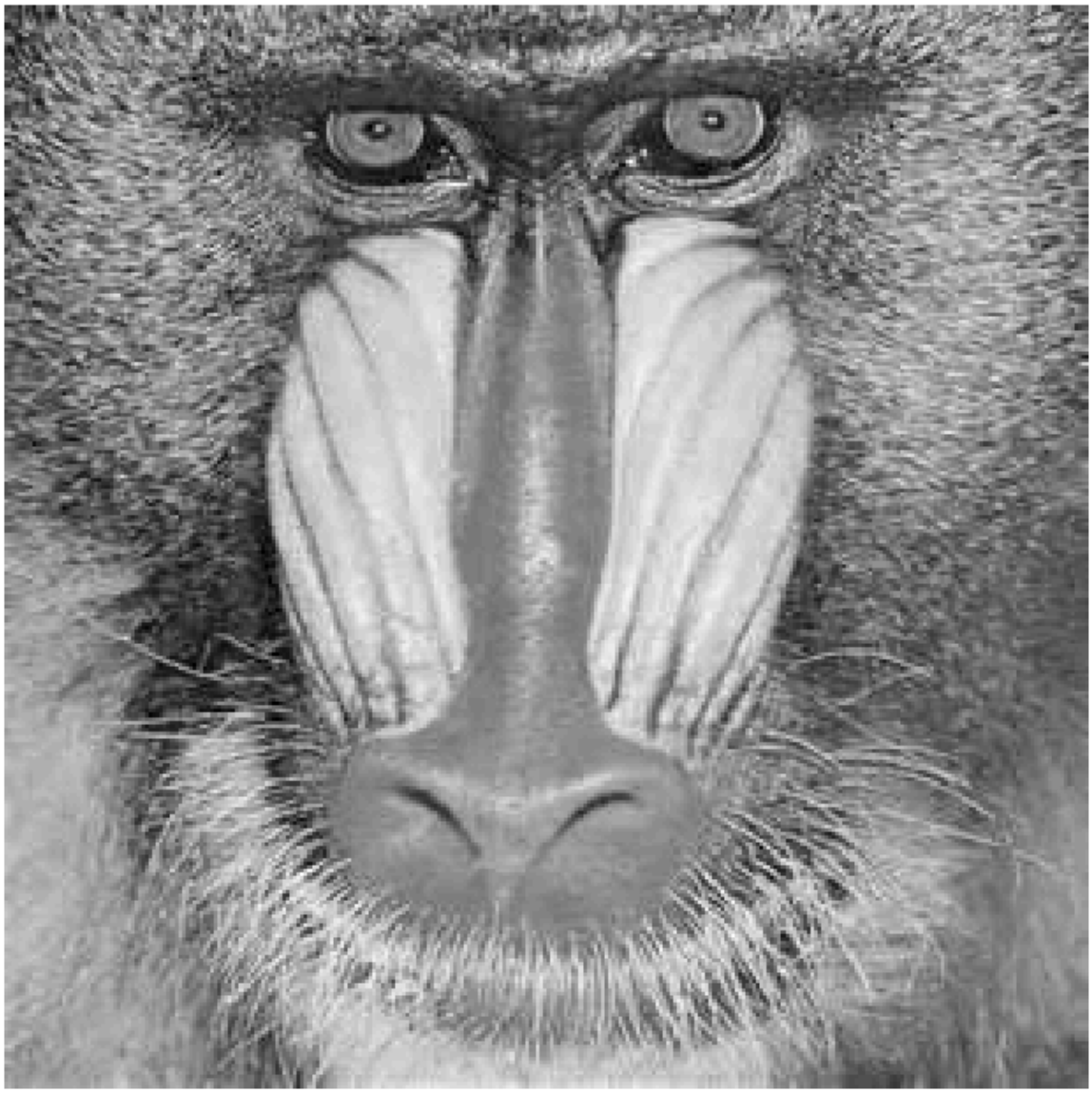
contrast 0.5

```
Height:  300
Width:  300
Channels:  3
```

contrast 1.5

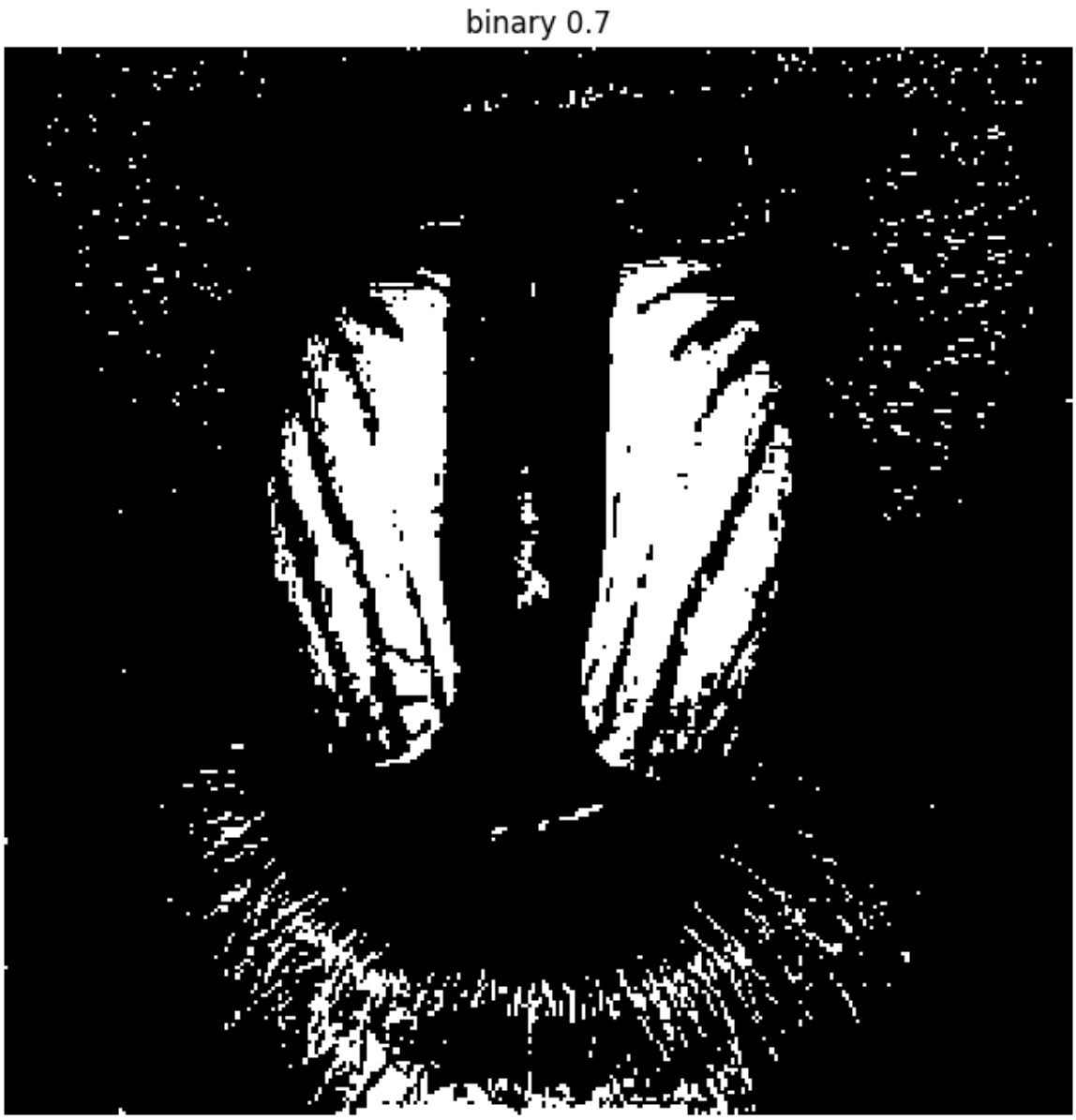


```
Height:  300
Width:  300
Channels:  3
```




```
Height:  300
Width:  300
Channels: 1
```


```
Height:  300
Width:  300
Channels: 1
```




```
Height:  300
Width:  300
Channels: 1
```

Discussion

Cropping:

- Preserved full color detail and enabled focused analysis on specific regions.

Resizing (nearest-neighbor):

- Upscaling caused visible pixelation due to lack of interpolation.
- Downsampling introduced aliasing; fine details were lost, edges became jagged.

Contrast Adjustment:

- Low contrast (0.5) flattened details; image looked washed out.
- High contrast (1.5) enhanced edges but clipped highlights (example: café's neon lights).

Grayscale Conversion:

- Averaging RGB channels preserved structure but lost color cues.
- Patterns remained visible, but color-based features became indistinct
- No difference in result when applying grayscale conversion to an already grayscale image

Binary Thresholding:

- Low threshold (0.3) retained most bright areas; image mostly white.
- High threshold (0.7) removed midtones; only the brightest regions remained.
- Effective for separating regions by intensity.

Question 3: Convolution (30%)

3.1(a) 2D convolution

Using the definition of 2D convolution from week 1, implement the convolution operation in the function `conv2D()` in a1code.py.

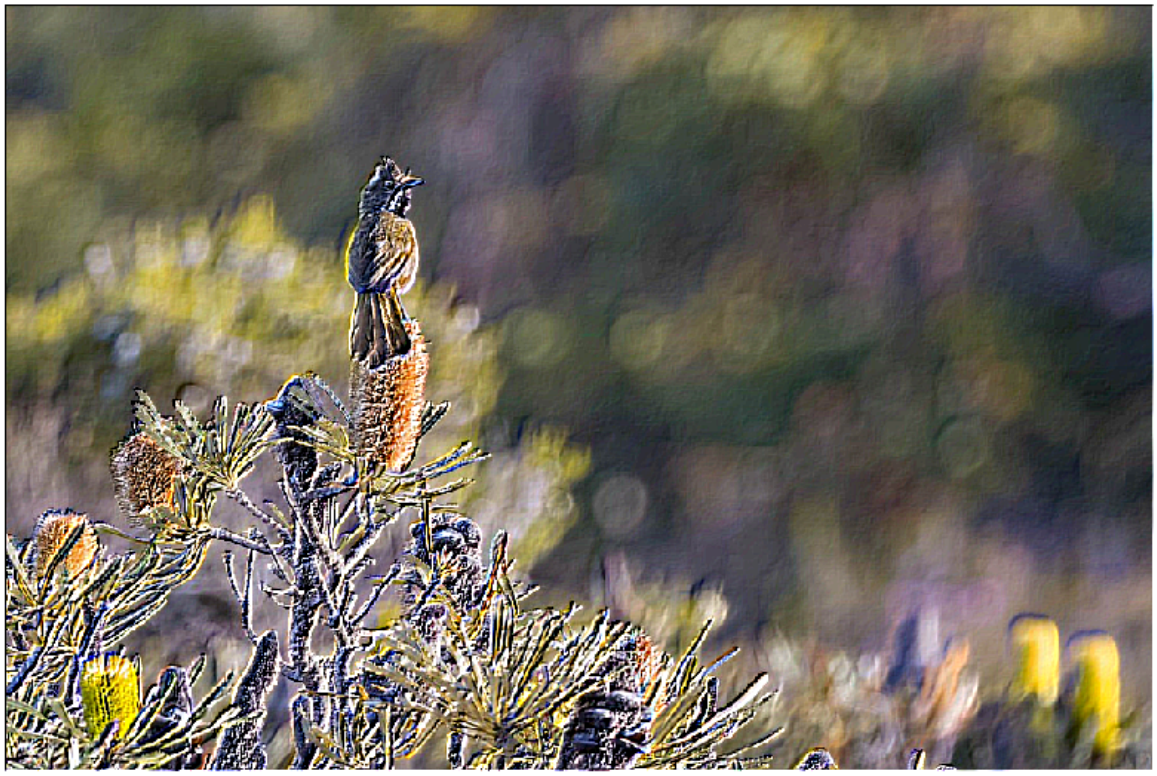
In [13]:
```
test_conv2D()
```

3.1(b) RGB convolution

In the function `conv` in a1code.py, extend your function `conv2D` to work on RGB images, by applying the 2D convolution to each channel independently.

In [14]:
```
# sample kernel (Emboss filter)
kernel = np.array([[-2, -1, 0],
                   [-1, 1, 1],
                   [0, 1, 2]])

image1 = load('images/whipbird.jpg')

test_output = conv(image1, kernel)

display(test_output, 'convolution output')
print_stats(test_output)
```

convolution output



```
Height:  667
Width:  1000
Channels:  3
```

3.2 Gaussian filter convolution

Use the `gauss2D` function provided in a1code.py to create a Gaussian kernel, and apply it to your images with convolution. You will obtain marks for trying different tests and analysing the results, for example:

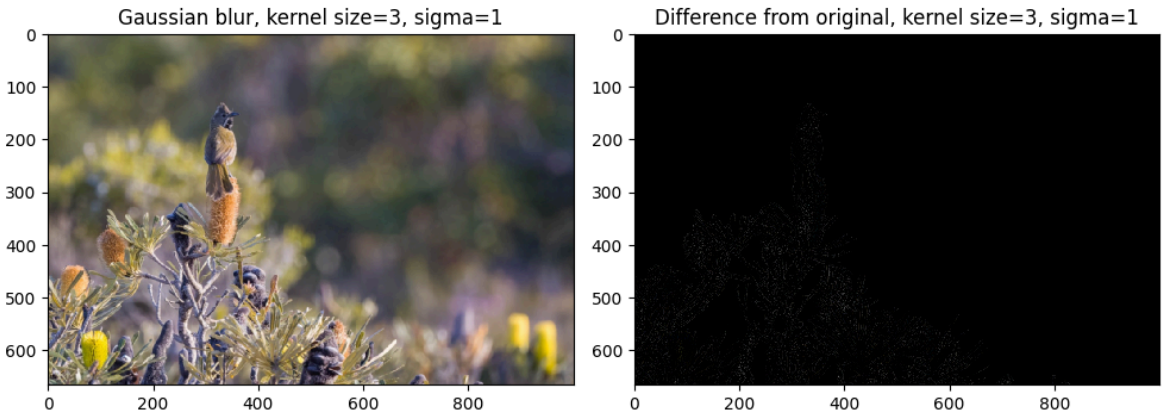
- try varying the image size, and the size and variance of the filter
- subtract the filtered image from the original - this gives you an idea of what information is lost when filtering

What do you observe and why?

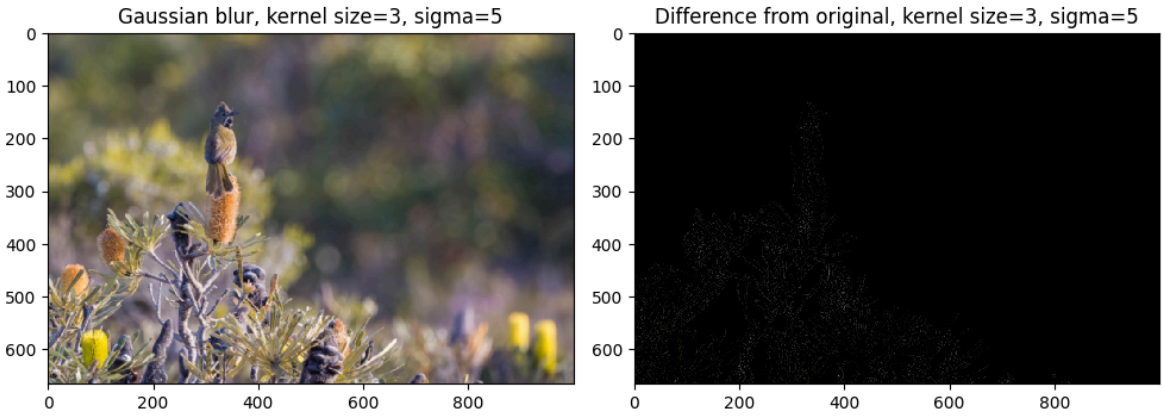
In [15]:
```
# Varying filter size and variance

import matplotlib.pyplot as plt

image1 = load('images/whipbird.jpg')

filter_sizes = [3, 7, 11]
sigmas = [1, 5, 10]
scale = [0.5, 1, 2]

# Test for sigma and filter size
for size in filter_sizes:
    for sigma in sigmas:
        gaussian_kernel = gauss2D(size, sigma)
        test_output = conv(image1, gaussian_kernel)
        diff = image1 - test_output
```

```
        plt.figure(figsize=(10, 5))

        plt.subplot(1, 2, 1)
        plt.imshow(test_output)
        plt.title(f'Gaussian blur, kernel size={size}, sigma={sigma}')

        plt.subplot(1, 2, 2)
        plt.imshow(diff)
        plt.title(f'Difference from original, kernel size={size}, sigma={

        plt.tight_layout()
        plt.show()

# Test for scale
for s in scale:
    gaussian_kernel = gauss2D(7, 5)
    scaled_image = resize(image1, int(image1.shape[0]*s), int(image1.shap
    test_output = conv(scaled_image, gaussian_kernel)
    diff = scaled_image - test_output

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(test_output)
    plt.title(f'Gaussian blur, resize scale={s}, kernel size=7, sigma=5')

    plt.subplot(1, 2, 2)
    plt.imshow(diff)
    plt.title(f'Difference, resize scale={s}, kernel size=7, sigma=5')

    plt.tight_layout()
    plt.show()
```

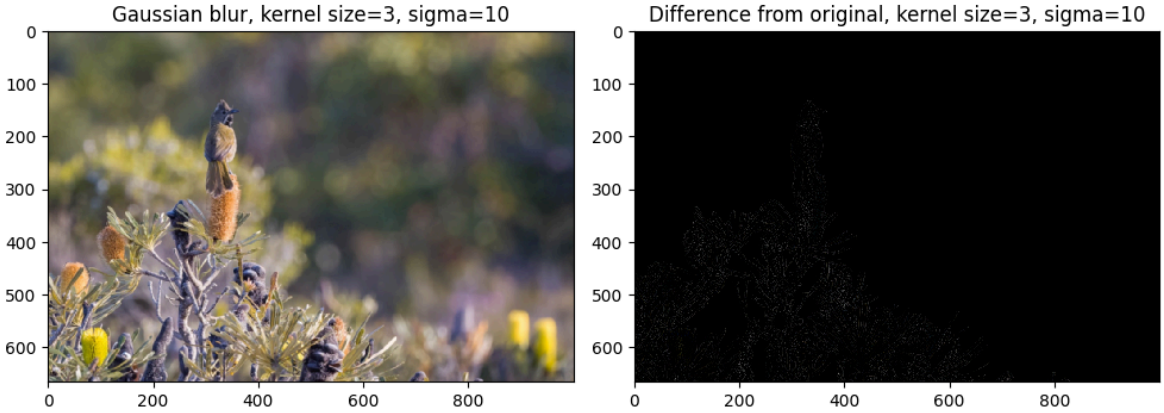
```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.30839969416562923..0.421
77638069358264].
```



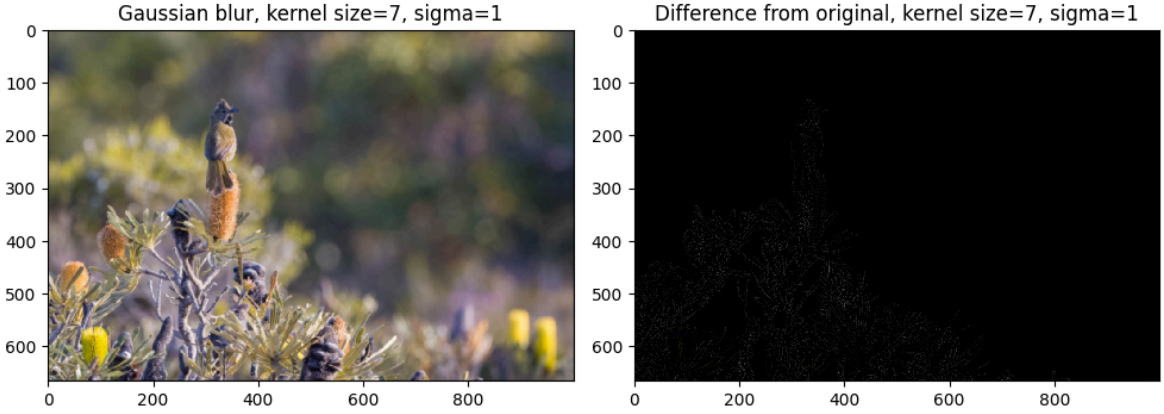

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.34932539467587814..0.493
1836769060888].
```



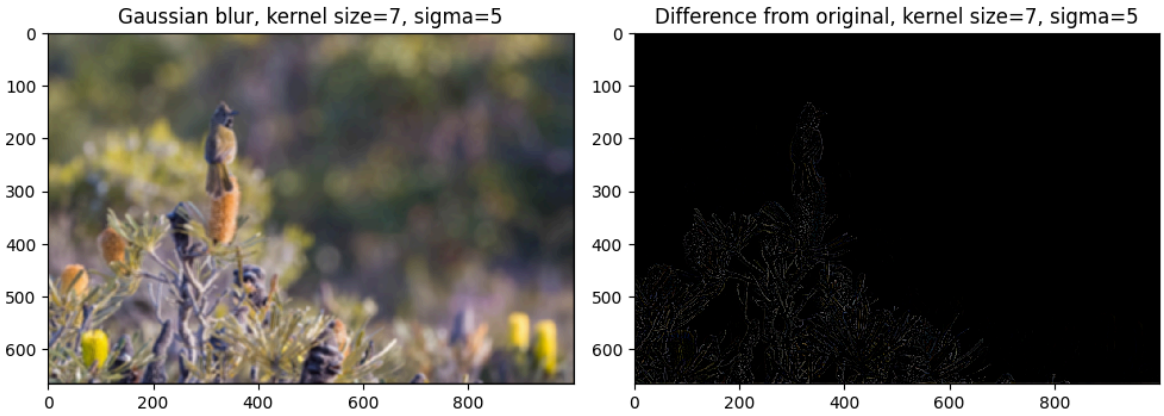

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.3504052776414211..0.4951
938040400469].
```



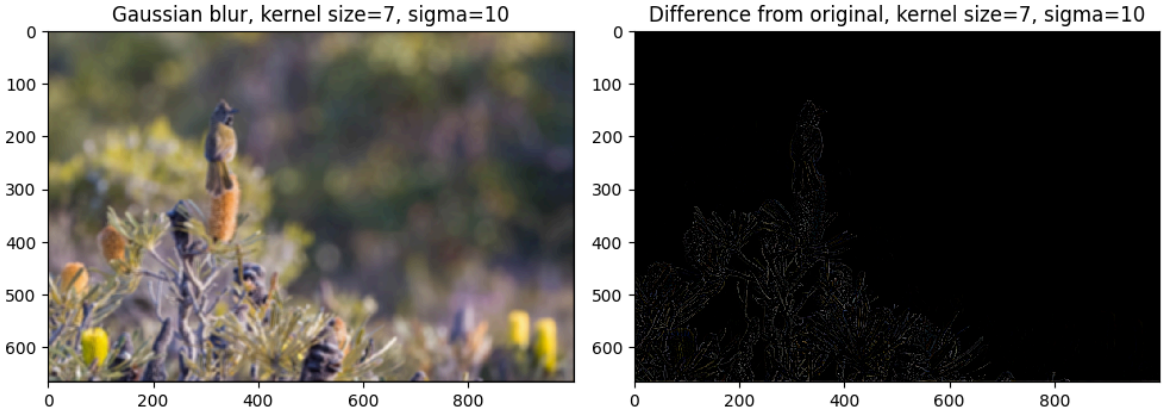

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.31124945670683696..0.469
2814708269352].
```



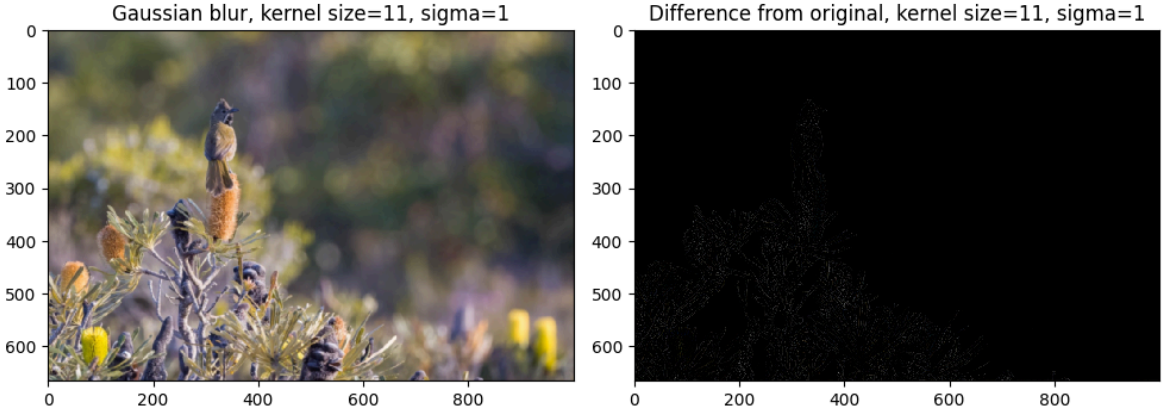

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.4777915557971515..0.6078
51458803965].
```



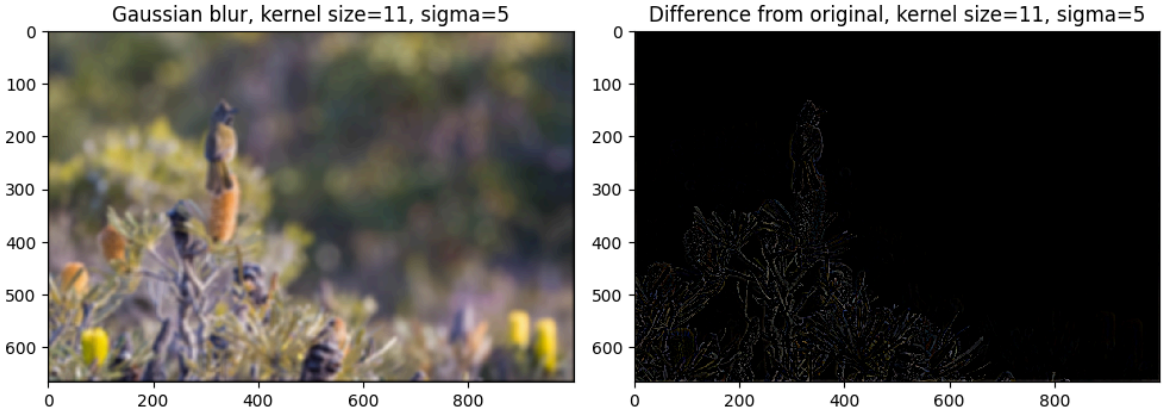

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.4844961336856064..0.6142
291156157147].
```



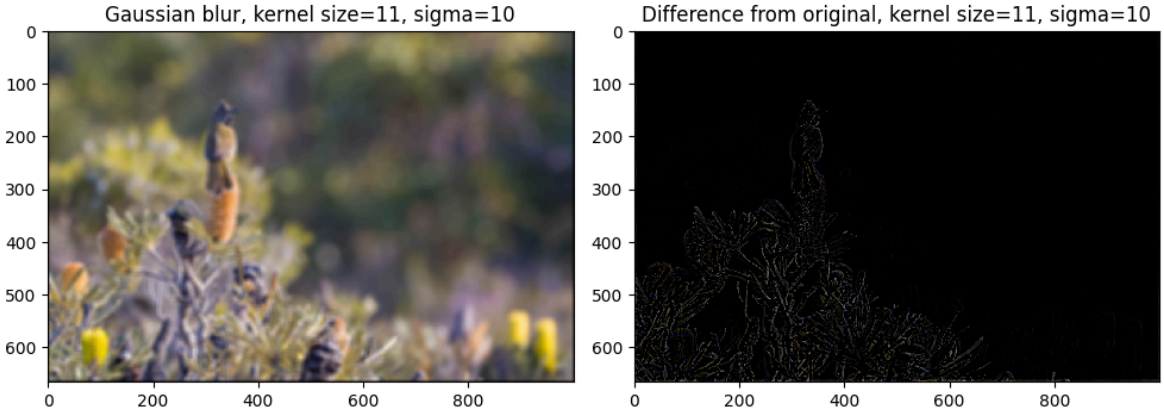

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.31128186187256596..0.469
323530061634].
```



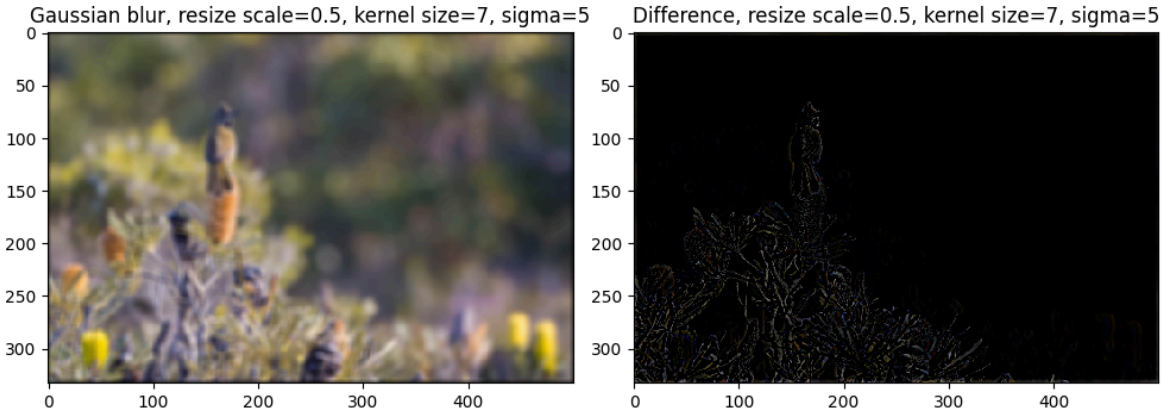

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.52883440191087..0.656266
5754384541].
```



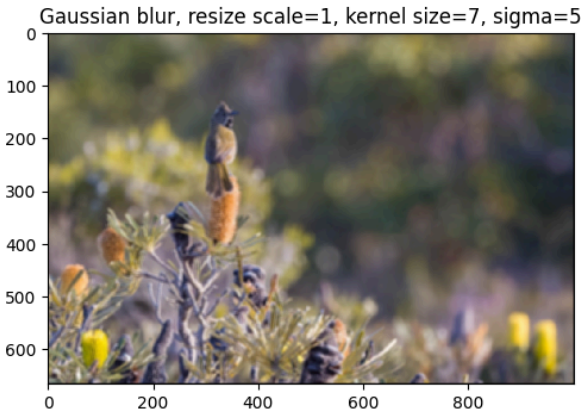

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.5420543267414696..0.6815
513811856776].
```

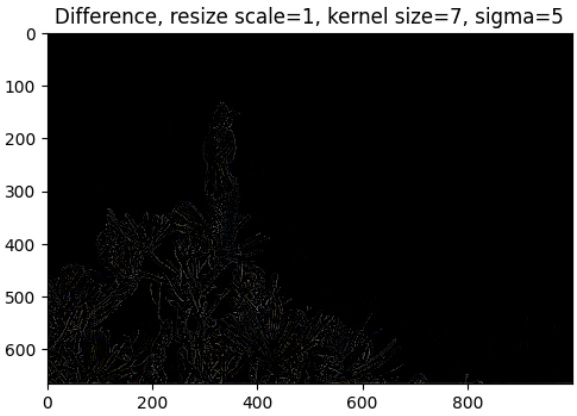



```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.5342519615096166..0.6547
057053652843].
```

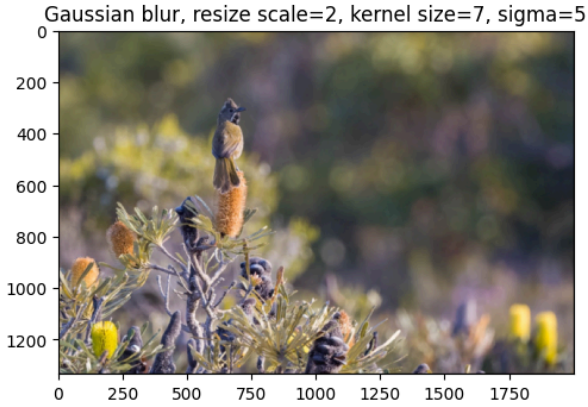


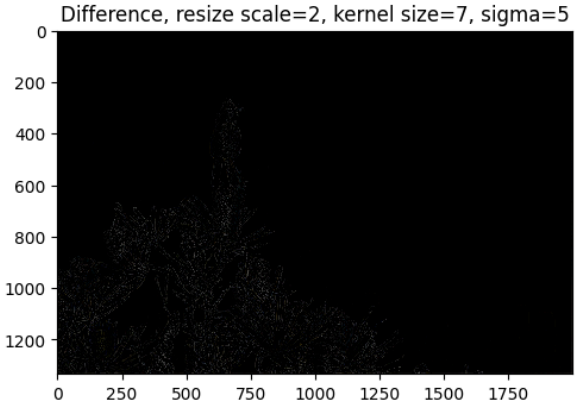

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.4777915557971515..0.6078
51458803965].
```






```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [-0.3753563943036882..0.5442
30346378419].
```





Discussion

For the same sigma, increasing the kernel size results in a stronger blurring effect:

- Larger kernels capture more of the Gaussian distribution, allowing the blur to spread further.
- Small kernels (e.g., 3×3) truncate the Gaussian, reducing the intended effect even if sigma is large.

For the same kernel size, increasing sigma also increases the blur:

- A higher sigma spreads the weight of the Gaussian, reducing the emphasis on the center pixel and smoothing out fine details more aggressively.
- This is visible in how edges and textures become progressively softer as σ increases.

Combined effect:

- Maximum blurring occurs when both kernel size and sigma are large.
- This was especially noticeable in the whipbird and mandrill image with large (11×11) kernels and sigma=10, almost all fine textures disappeared (have the most differences).

Effect of Image Size (Fixed Kernel Size, Fixed Sigma)

- Applying a fixed filter (size=7x7, sigma=5) to differently scaled images changes its relative effect.
- Downsampled image (Scale = 0.5): The fixed kernel covers larger image features proportionally. Difference image shows significantly more fine detail removal.
- Upsampled image (Scale = 2): The fixed kernel covers smaller image features/pixels proportionally. Difference image shows significantly less fine detail removal.
- Conclusion: The perceived smoothing strength of a fixed filter depends on the size of the image it interacts with.

Implication:

- Effective blurring depends on both kernel size and sigma working together.
- A mismatch (e.g., small kernel with large sigma) underrepresents the Gaussian and weakens the filter's effectiveness.

3.3 Sobel filters

Define a horizontal and vertical Sobel edge filter kernel and test them on your images. You will obtain marks for testing them and displaying results in interesting ways, for example:

- apply them to an image at different scales
- considering how to display positive and negative gradients
- apply different combinations of horizontal and vertical filters as asked in the Assignment sheet.

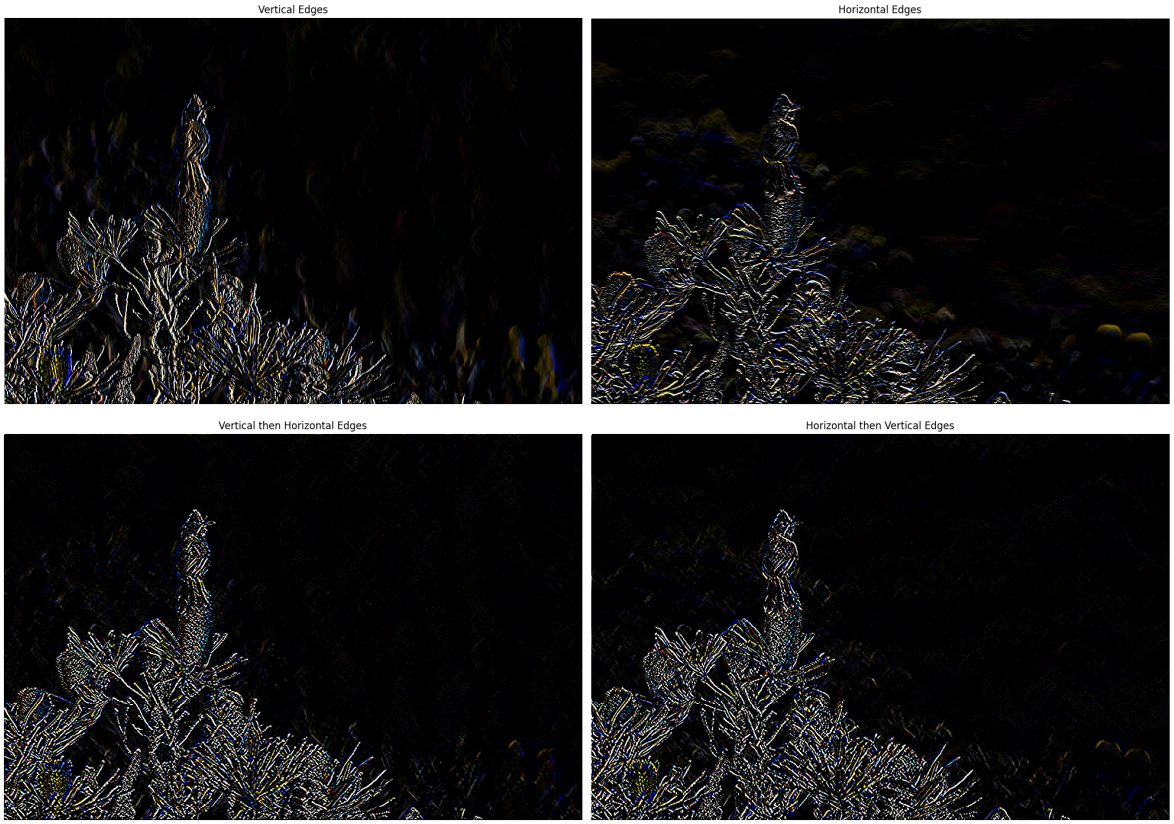
In [16]:
```
# Sobel edge detection kernels
sobel_vertical = np.array([[1, 0, -1],
                           [2, 0, -2],
                           [1, 0, -1]])

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

In [17]:
```
# Apply sobel filters to the original image and display
image1 = load('images/whipbird.jpg')
vertical_edges = conv(image1, sobel_vertical)
horizontal_edges = conv(image1, sobel_horizontal)
vertical_horizontal = conv(vertical_edges, sobel_horizontal)
horizontal_vertical = conv(horizontal_edges, sobel_vertical)

# Create a figure with 4 subplots side by side
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_edges, cmap='gray')
```

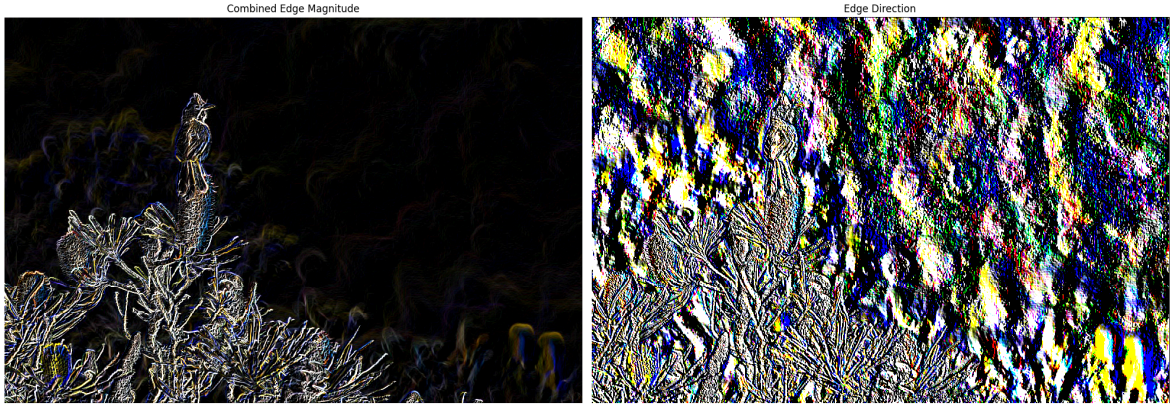
```
plt.title('Vertical Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_edges, cmap='gray')
plt.title('Horizontal Edges')
plt.axis('off')

plt.tight_layout()
plt.show()

# Figure for vertical-horizontal edges
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_horizontal, cmap='gray')
plt.title('Vertical then Horizontal Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_vertical, cmap='gray')
plt.title('Horizontal then Vertical Edges')
plt.axis('off')

plt.tight_layout()
plt.show()


# Combine responses to compute edge magnitude and direction
edge_magnitude = np.sqrt(vertical_edges**2 + horizontal_edges**2)
edge_direction = np.arctan2(vertical_edges, horizontal_edges)

plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(edge_magnitude, cmap='gray')
plt.title('Combined Edge Magnitude')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(edge_direction, cmap='gray')
plt.title('Edge Direction')
plt.axis('off')

plt.tight_layout()
plt.show()

# Positive and negative gradient
positive_vertical = np.maximum(vertical_edges, 0)
negative_vertical = np.minimum(vertical_edges, 0)


plt.figure(figsize=(20, 12))

# Plot edge magnitude
```

```
plt.subplot(121)
plt.imshow(positive_vertical, cmap='gray')
plt.title('Positive Vertical Gradient Response')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(negative_vertical, cmap='gray')
plt.title('Negative Vertical Gradient Response')
plt.axis('off')

plt.tight_layout()
plt.show()


# Blurr first and then apply Sobel filters
gaussian_kernel_5_10 = gauss2D(5, 10)
blurred_image = conv(image1, gaussian_kernel_5_10)

# Apply the Sobel filters to the blurred image
scaled_vertical = conv(blurred_image, sobel_vertical)
scaled_horizontal = conv(blurred_image, sobel_horizontal)

plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(scaled_vertical, cmap='gray')
plt.title('Vertical Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(scaled_horizontal, cmap='gray')
plt.title('Horizontal Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

plt.tight_layout()
plt.show()

# Resizing first and then apply Sobel filters
scales = [0.5, 1.0, 2.0]
for scale in scales:
    # Resize the image
    new_rows = int(image1.shape[0] * scale)
    new_cols = int(image1.shape[1] * scale)
    scaled_img = resize(image1, new_rows, new_cols)

    # Apply the Sobel filters on the resized image
    scaled_vert = conv(scaled_img, sobel_vertical)
    scaled_horiz = conv(scaled_img, sobel_horizontal)

    # Display all edge responses in one figure
    plt.figure(figsize=(20,12))

    plt.subplot(231)
    plt.imshow(scaled_vert, cmap='gray')
    plt.title(f'Vertical Edge Response (Scale {scale})')
    plt.axis('off')
```

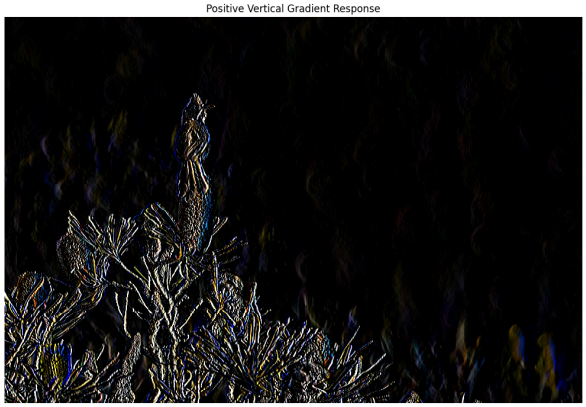
```
    plt.subplot(232)
    plt.imshow(scaled_horiz, cmap='gray')
    plt.title(f'Horizontal Edge Response (Scale {scale})')
    plt.axis('off')

    plt.tight_layout()
    plt.show()
```



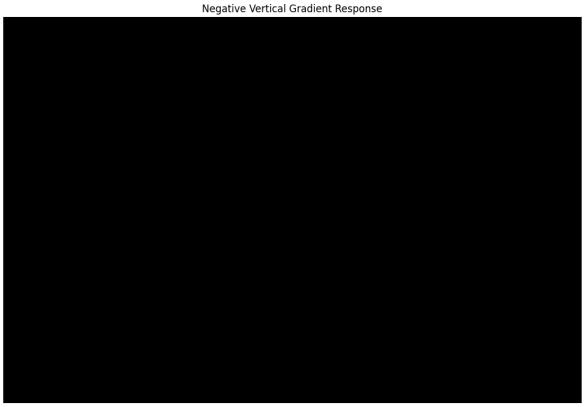

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..1.4142135623730951].
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..1.5707963267948966].
```



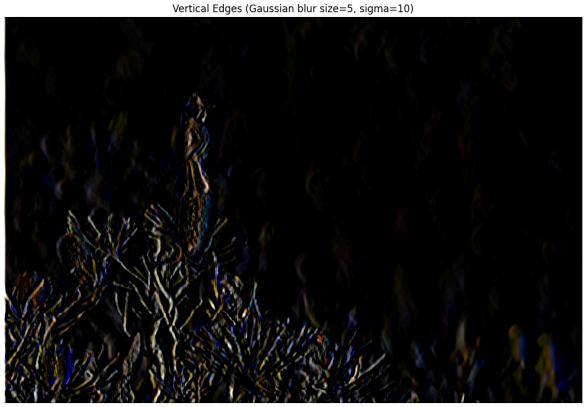
Positive Vertical Gradient Response



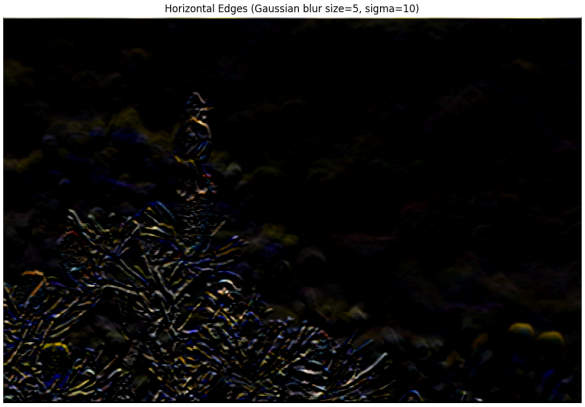
Negative Vertical Gradient Response



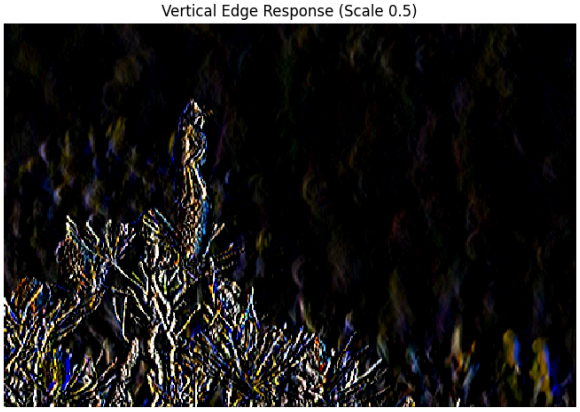
Vertical Edges (Gaussian blur size=5, sigma=10)



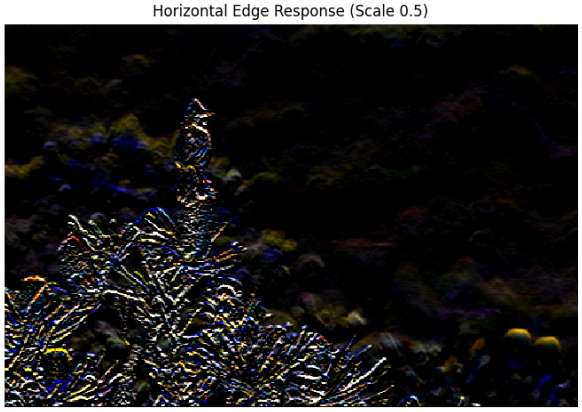
Horizontal Edges (Gaussian blur size=5, sigma=10)



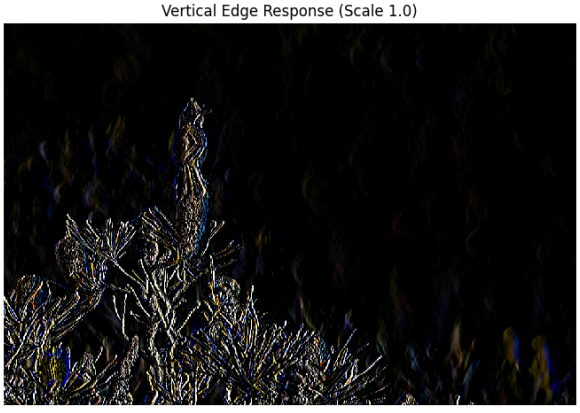
Vertical Edge Response (Scale 0.5)



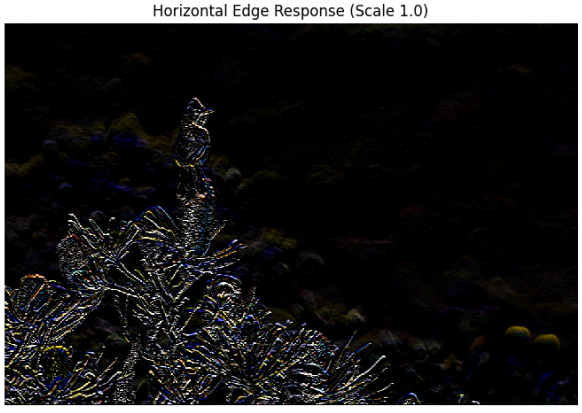
Horizontal Edge Response (Scale 0.5)

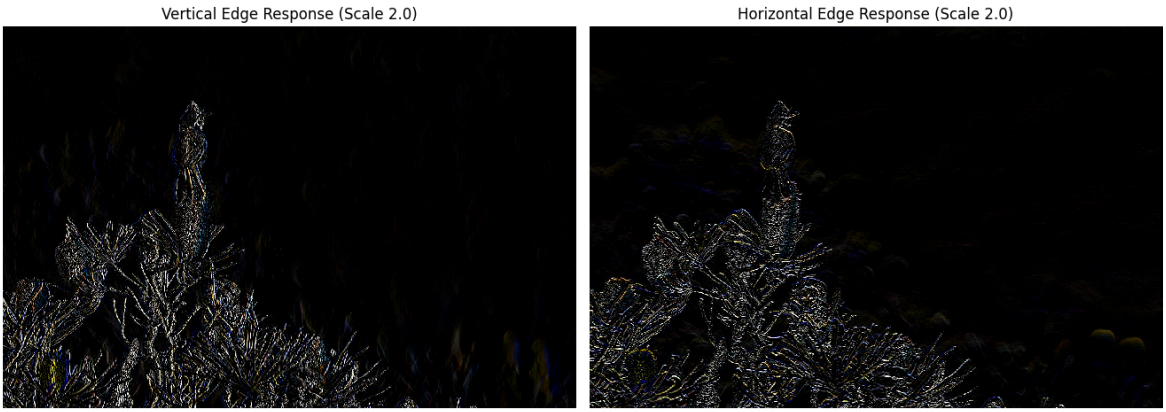


Vertical Edge Response (Scale 1.0)



Horizontal Edge Response (Scale 1.0)

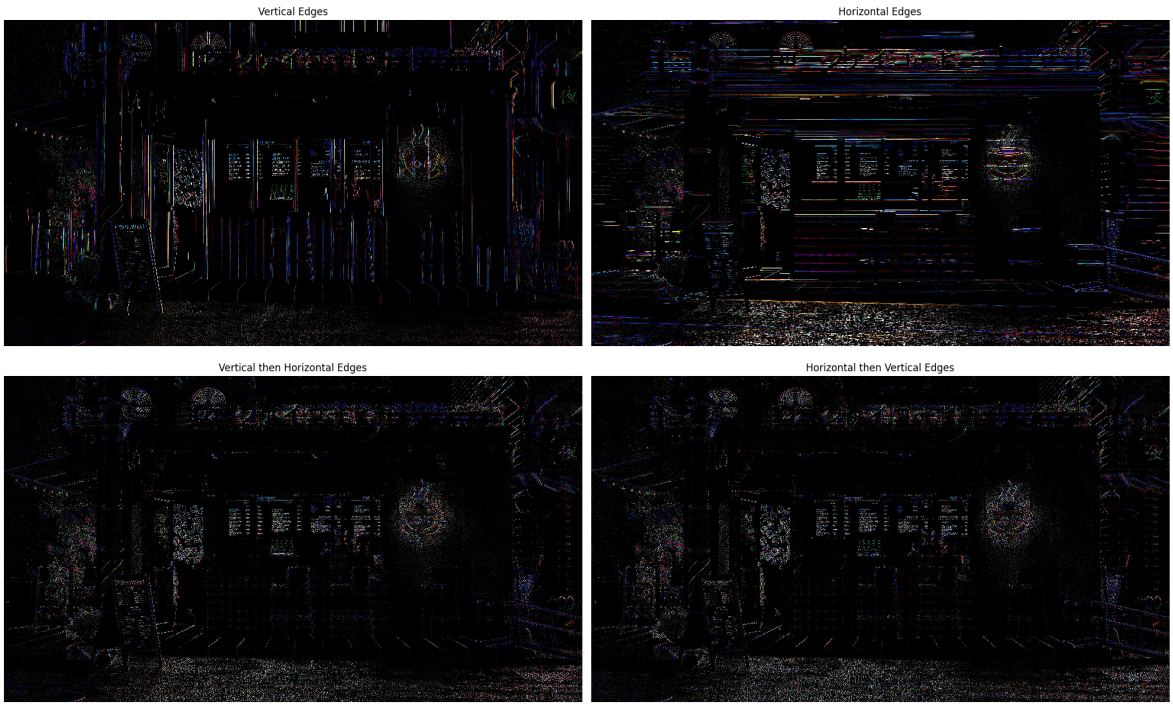




In [18]:
```
# Apply sobel filters to the original image and display
image2 = load('images/cafe.jpg')
vertical_edges = conv(image2, sobel_vertical)
horizontal_edges = conv(image2, sobel_horizontal)
vertical_horizontal = conv(vertical_edges, sobel_horizontal)
horizontal_vertical = conv(horizontal_edges, sobel_vertical)

# Create a figure with 4 subplots side by side
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_edges, cmap='gray')
plt.title('Vertical Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_edges, cmap='gray')
plt.title('Horizontal Edges')
plt.axis('off')

plt.tight_layout()
plt.show()

# Figure for vertical-horizontal edges
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_horizontal, cmap='gray')
plt.title('Vertical then Horizontal Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_vertical, cmap='gray')
plt.title('Horizontal then Vertical Edges')
plt.axis('off')

plt.tight_layout()
plt.show()



# Combine responses to compute edge magnitude and direction
```

```
edge_magnitude = np.sqrt(vertical_edges**2 + horizontal_edges**2)
edge_direction = np.arctan2(vertical_edges, horizontal_edges)

plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(edge_magnitude, cmap='gray')
plt.title('Combined Edge Magnitude')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(edge_direction, cmap='gray')
plt.title('Edge Direction')
plt.axis('off')

plt.tight_layout()
plt.show()

# Positive and negative gradient
positive_vertical = np.maximum(vertical_edges, 0)
negative_vertical = np.minimum(vertical_edges, 0)


plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(positive_vertical, cmap='gray')
plt.title('Positive Vertical Gradient Response')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(negative_vertical, cmap='gray')
plt.title('Negative Vertical Gradient Response')
plt.axis('off')

plt.tight_layout()
plt.show()


# Blurr first and then apply Sobel filters
gaussian_kernel_5_10 = gauss2D(5, 10)
blurred_image = conv(image2, gaussian_kernel_5_10)

# Apply the Sobel filters to the blurred image
scaled_vertical = conv(blurred_image, sobel_vertical)
scaled_horizontal = conv(blurred_image, sobel_horizontal)

plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(scaled_vertical, cmap='gray')
plt.title('Vertical Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

# Plot horizontal edges
```

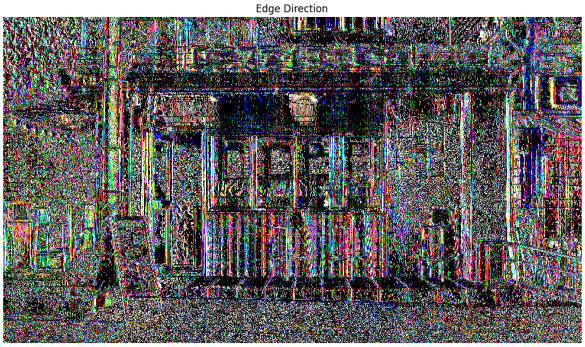
```
plt.subplot(122)
plt.imshow(scaled_horizontal, cmap='gray')
plt.title('Horizontal Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

plt.tight_layout()
plt.show()

# Resizing first and then apply Sobel filters
scales = [0.5, 1.0, 2.0]
for scale in scales:
    # Resize the image
    new_rows = int(image2.shape[0] * scale)
    new_cols = int(image2.shape[1] * scale)
    scaled_img = resize(image2, new_rows, new_cols)

    # Apply the Sobel filters on the resized image
    scaled_vert = conv(scaled_img, sobel_vertical)
    scaled_horiz = conv(scaled_img, sobel_horizontal)

    # Display all edge responses in one figure
    plt.figure(figsize=(20,12))

    plt.subplot(231)
    plt.imshow(scaled_vert, cmap='gray')
    plt.title(f'Vertical Edge Response (Scale {scale})')
    plt.axis('off')

    plt.subplot(232)
    plt.imshow(scaled_horiz, cmap='gray')
    plt.title(f'Horizontal Edge Response (Scale {scale})')
    plt.axis('off')

    plt.tight_layout()
    plt.show()
```



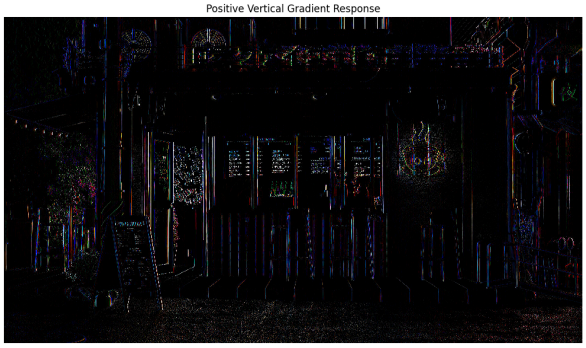
```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..1.4142135623730951].
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..1.5707963267948966].
```

Combined Edge Magnitude

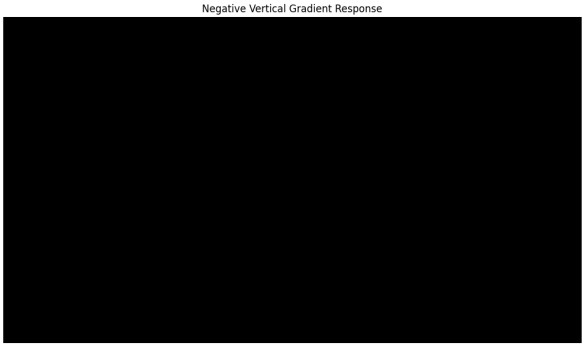
Edge Direction



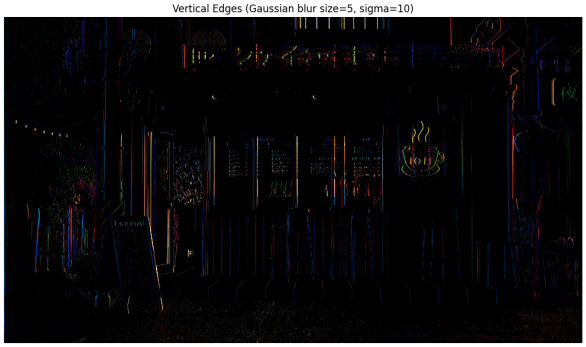
Positive Vertical Gradient Response



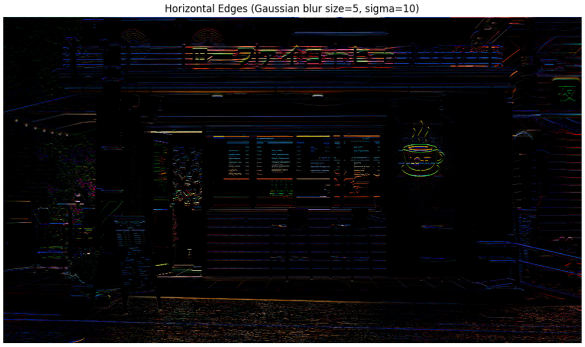
Negative Vertical Gradient Response



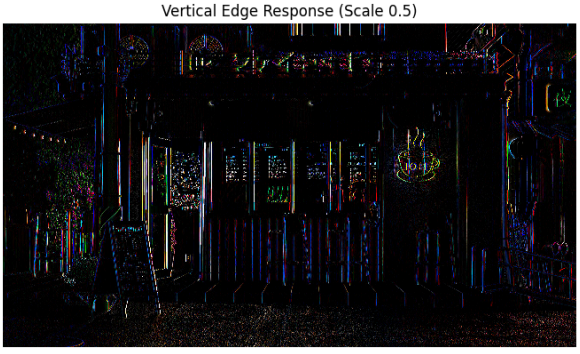
Vertical Edges (Gaussian blur size=5, sigma=10)



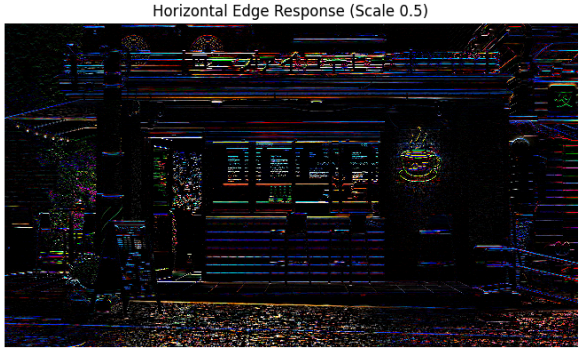
Horizontal Edges (Gaussian blur size=5, sigma=10)



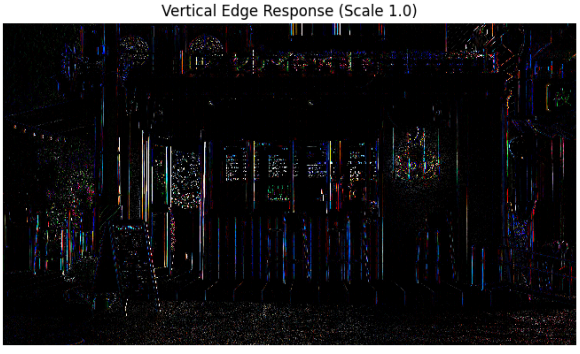
Vertical Edge Response (Scale 0.5)



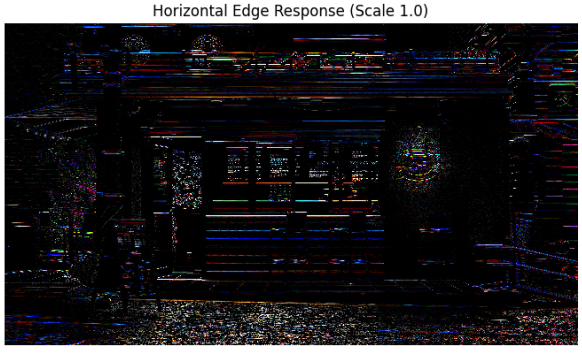
Horizontal Edge Response (Scale 0.5)

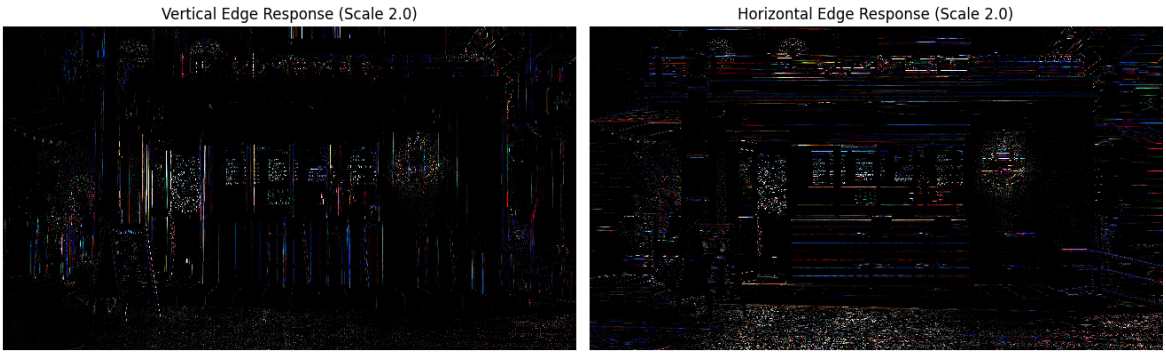


Vertical Edge Response (Scale 1.0)



Horizontal Edge Response (Scale 1.0)



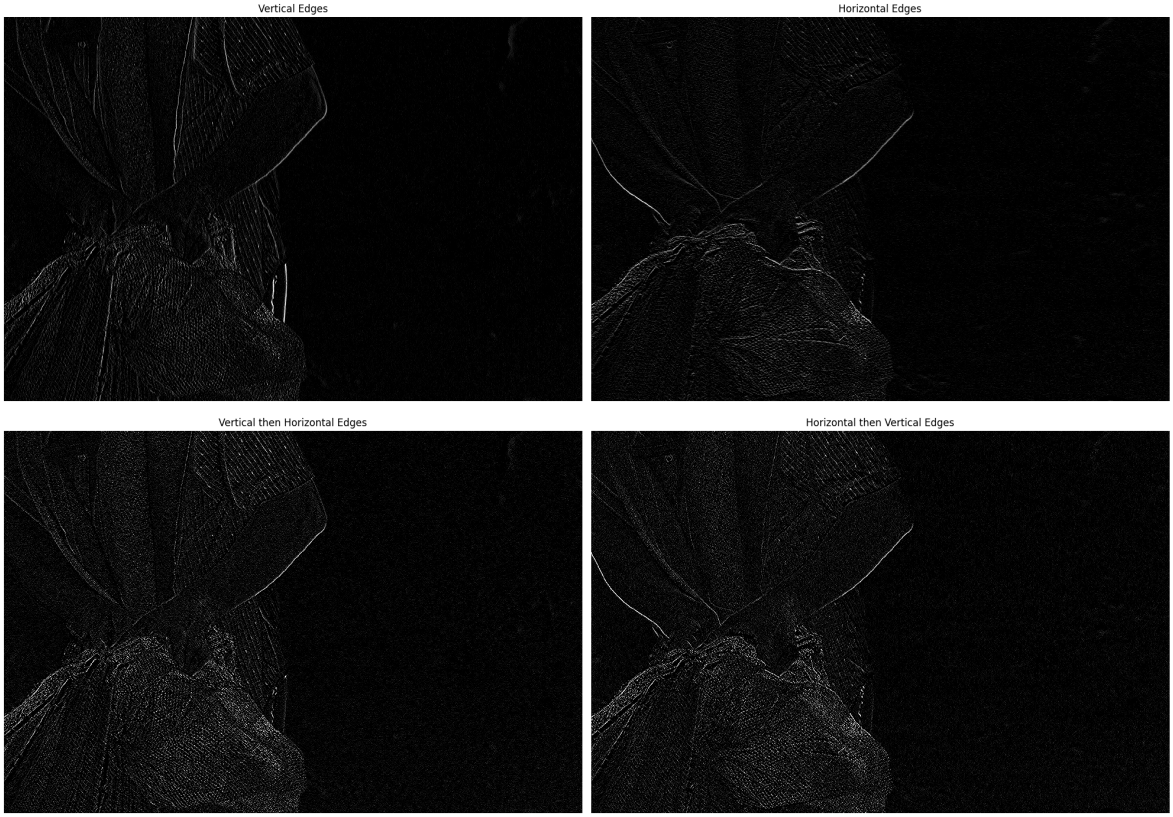


In [19]:

```python
# Apply sobel filters to the original image and display
image3 = load('images/grayscale.jpg')
vertical_edges = conv(image3, sobel_vertical)
horizontal_edges = conv(image3, sobel_horizontal)
vertical_horizontal = conv(vertical_edges, sobel_horizontal)
horizontal_vertical = conv(horizontal_edges, sobel_vertical)

# Create a figure with 4 subplots side by side
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_edges, cmap='gray')
plt.title('Vertical Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_edges, cmap='gray')
plt.title('Horizontal Edges')
plt.axis('off')

plt.tight_layout()
plt.show()

# Figure for vertical-horizontal edges
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_horizontal, cmap='gray')
plt.title('Vertical then Horizontal Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_vertical, cmap='gray')
plt.title('Horizontal then Vertical Edges')
plt.axis('off')

plt.tight_layout()
plt.show()



# Combine responses to compute edge magnitude and direction
edge_magnitude = np.sqrt(vertical_edges**2 + horizontal_edges**2)
edge_direction = np.arctan2(vertical_edges, horizontal_edges)
```

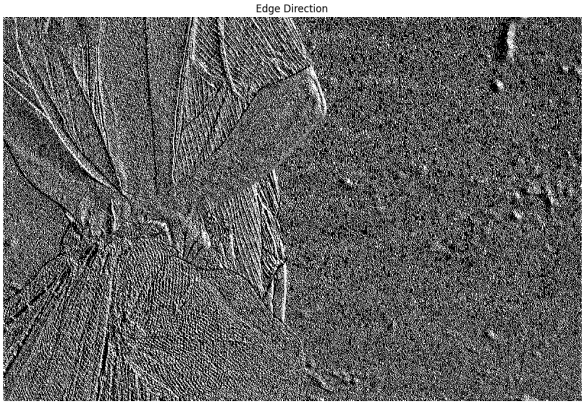
```
plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(edge_magnitude, cmap='gray')
plt.title('Combined Edge Magnitude')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(edge_direction, cmap='gray')
plt.title('Edge Direction')
plt.axis('off')

plt.tight_layout()
plt.show()

# Positive and negative gradient
positive_vertical = np.maximum(vertical_edges, 0)
negative_vertical = np.minimum(vertical_edges, 0)


plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(positive_vertical, cmap='gray')
plt.title('Positive Vertical Gradient Response')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(negative_vertical, cmap='gray')
plt.title('Negative Vertical Gradient Response')
plt.axis('off')

plt.tight_layout()
plt.show()


# Blurr first and then apply Sobel filters
gaussian_kernel_5_10 = gauss2D(5, 10)
blurred_image = conv(image3, gaussian_kernel_5_10)

# Apply the Sobel filters to the blurred image
scaled_vertical = conv(blurred_image, sobel_vertical)
scaled_horizontal = conv(blurred_image, sobel_horizontal)

plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(scaled_vertical, cmap='gray')
plt.title('Vertical Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(scaled_horizontal, cmap='gray')
```

```
plt.title('Horizontal Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

plt.tight_layout()
plt.show()

# Resizing first and then apply Sobel filters
scales = [0.5, 1.0, 2.0]
for scale in scales:
    # Resize the image
    new_rows = int(image3.shape[0] * scale)
    new_cols = int(image3.shape[1] * scale)
    scaled_img = resize(image3, new_rows, new_cols)

    # Apply the Sobel filters on the resized image
    scaled_vert = conv(scaled_img, sobel_vertical)
    scaled_horiz = conv(scaled_img, sobel_horizontal)

    # Display all edge responses in one figure
    plt.figure(figsize=(20,12))

    plt.subplot(231)
    plt.imshow(scaled_vert, cmap='gray')
    plt.title(f'Vertical Edge Response (Scale {scale})')
    plt.axis('off')

    plt.subplot(232)
    plt.imshow(scaled_horiz, cmap='gray')
    plt.title(f'Horizontal Edge Response (Scale {scale})')
    plt.axis('off')

    plt.tight_layout()
    plt.show()
```

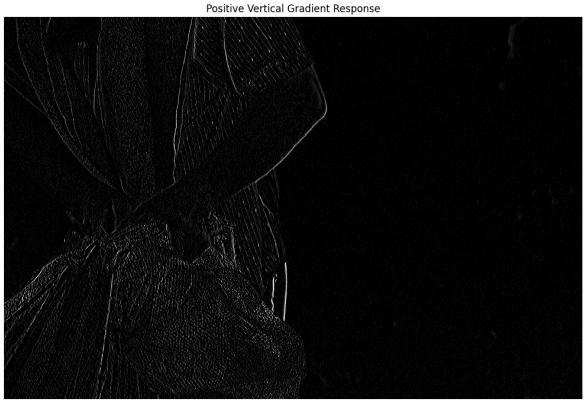


Combined Edge Magnitude

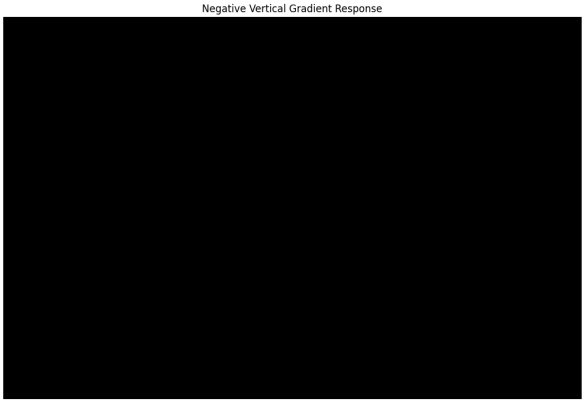
Edge Direction



Positive Vertical Gradient Response

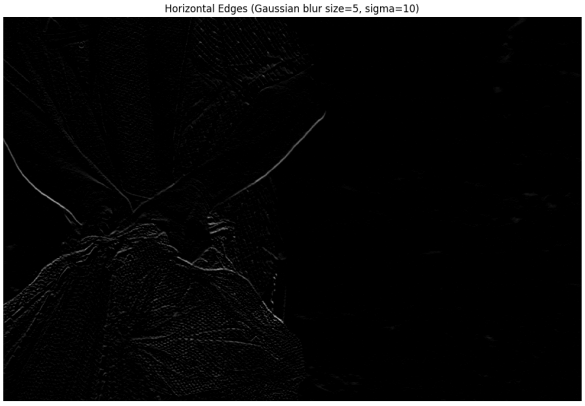


Negative Vertical Gradient Response



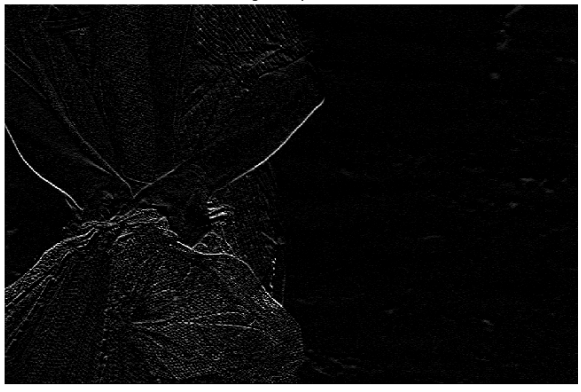
Vertical Edges (Gaussian blur size=5, sigma=10)

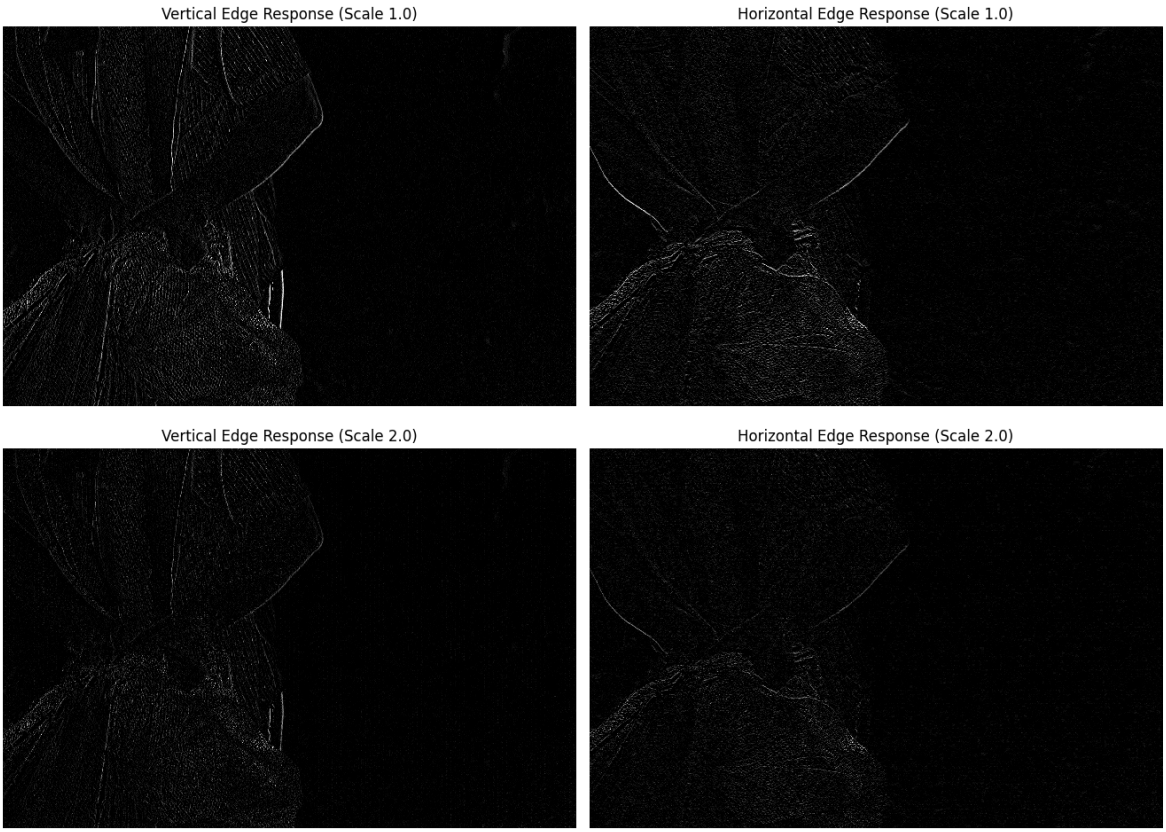
Horizontal Edges (Gaussian blur size=5, sigma=10)



Vertical Edge Response (Scale 0.5)

Horizontal Edge Response (Scale 0.5)





In [20]:
```
# Apply sobel filters to the original image and display
image4 = load('images/mandrill.jpg')
vertical_edges = conv(image4, sobel_vertical)
horizontal_edges = conv(image4, sobel_horizontal)
vertical_horizontal = conv(vertical_edges, sobel_horizontal)
horizontal_vertical = conv(horizontal_edges, sobel_vertical)

# Create a figure with 4 subplots side by side
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_edges, cmap='gray')
plt.title('Vertical Edges')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_edges, cmap='gray')
plt.title('Horizontal Edges')
plt.axis('off')

plt.tight_layout()
plt.show()

# Figure for vertical-horizontal edges
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(vertical_horizontal, cmap='gray')
plt.title('Vertical then Horizontal Edges')
plt.axis('off')
```

```
# Plot horizontal edges
plt.subplot(122)
plt.imshow(horizontal_vertical, cmap='gray')
plt.title('Horizontal then Vertical Edges')
plt.axis('off')

plt.tight_layout()
plt.show()

# Combine responses to compute edge magnitude and direction
edge_magnitude = np.sqrt(vertical_edges**2 + horizontal_edges**2)
edge_direction = np.arctan2(vertical_edges, horizontal_edges)

plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(edge_magnitude, cmap='gray')
plt.title('Combined Edge Magnitude')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(edge_direction, cmap='gray')
plt.title('Edge Direction')
plt.axis('off')

plt.tight_layout()
plt.show()

# Positive and negative gradient
positive_vertical = np.maximum(vertical_edges, 0)
negative_vertical = np.minimum(vertical_edges, 0)


plt.figure(figsize=(20, 12))

# Plot edge magnitude
plt.subplot(121)
plt.imshow(positive_vertical, cmap='gray')
plt.title('Positive Vertical Gradient Response')
plt.axis('off')

# Plot edge direction
plt.subplot(122)
plt.imshow(negative_vertical, cmap='gray')
plt.title('Negative Vertical Gradient Response')
plt.axis('off')

plt.tight_layout()
plt.show()


# Blurr first and then apply Sobel filters
gaussian_kernel_5_10 = gauss2D(5, 10)
blurred_image = conv(image4, gaussian_kernel_5_10)

# Apply the Sobel filters to the blurred image
scaled_vertical = conv(blurred_image, sobel_vertical)
scaled_horizontal = conv(blurred_image, sobel_horizontal)
```

```
plt.figure(figsize=(20, 12))

# Plot vertical edges
plt.subplot(121)
plt.imshow(scaled_vertical, cmap='gray')
plt.title('Vertical Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

# Plot horizontal edges
plt.subplot(122)
plt.imshow(scaled_horizontal, cmap='gray')
plt.title('Horizontal Edges (Gaussian blur size=5, sigma=10)')
plt.axis('off')

plt.tight_layout()
plt.show()

# Resizing first and then apply Sobel filters
scales = [0.5, 1.0, 2.0]
for scale in scales:
    # Resize the image
    new_rows = int(image4.shape[0] * scale)
    new_cols = int(image4.shape[1] * scale)
    scaled_img = resize(image4, new_rows, new_cols)

    # Apply the Sobel filters on the resized image
    scaled_vert = conv(scaled_img, sobel_vertical)
    scaled_horiz = conv(scaled_img, sobel_horizontal)

    # Display all edge responses in one figure
    plt.figure(figsize=(20,12))

    plt.subplot(231)
    plt.imshow(scaled_vert, cmap='gray')
    plt.title(f'Vertical Edge Response (Scale {scale})')
    plt.axis('off')

    plt.subplot(232)
    plt.imshow(scaled_horiz, cmap='gray')
    plt.title(f'Horizontal Edge Response (Scale {scale})')
    plt.axis('off')

    plt.tight_layout()
    plt.show()
```

Vertical Edges

Horizontal Edges

Vertical then Horizontal Edges

Horizontal then Vertical Edges

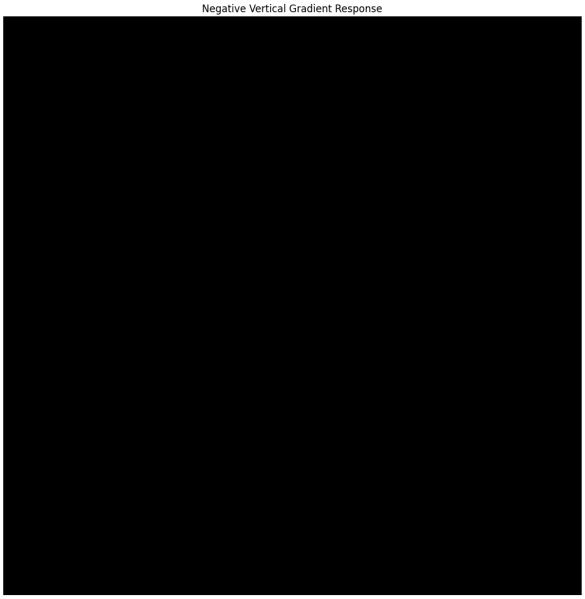
```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..1.4142135623730951].
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers). Got range [0.0..1.5707963267948966].
```

Combined Edge Magnitude

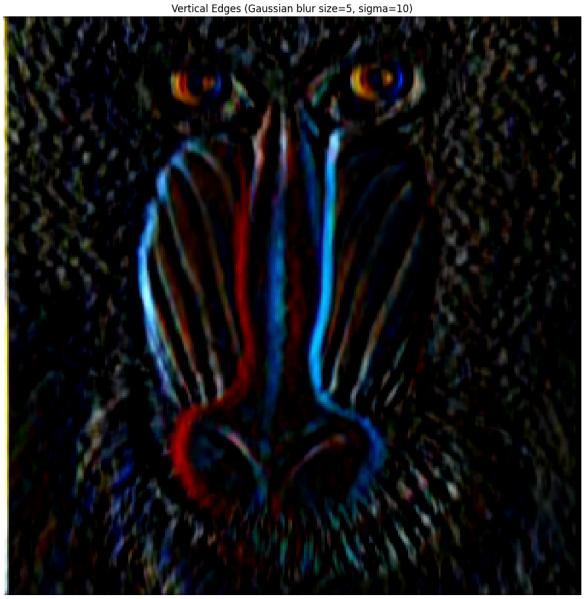
Edge Direction

Positive Vertical Gradient Response

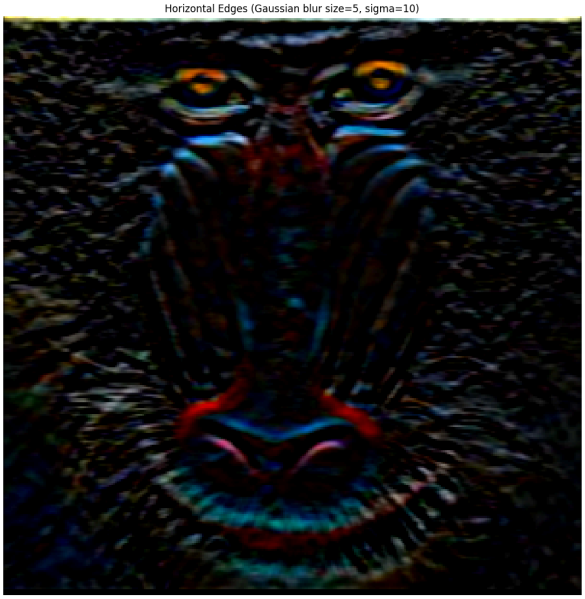
Negative Vertical Gradient Response



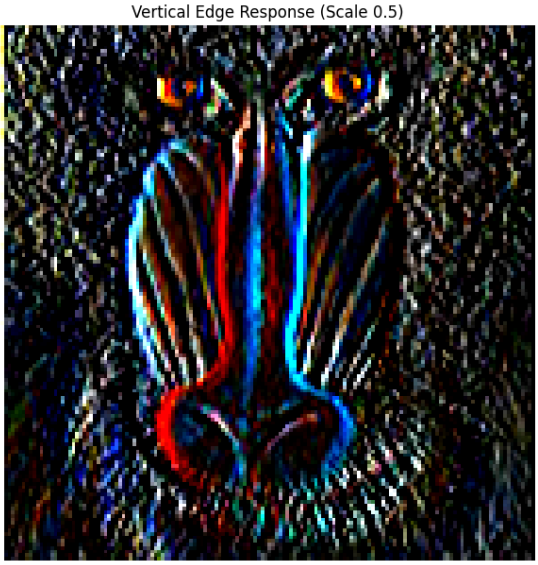
Vertical Edges (Gaussian blur size=5, sigma=10)



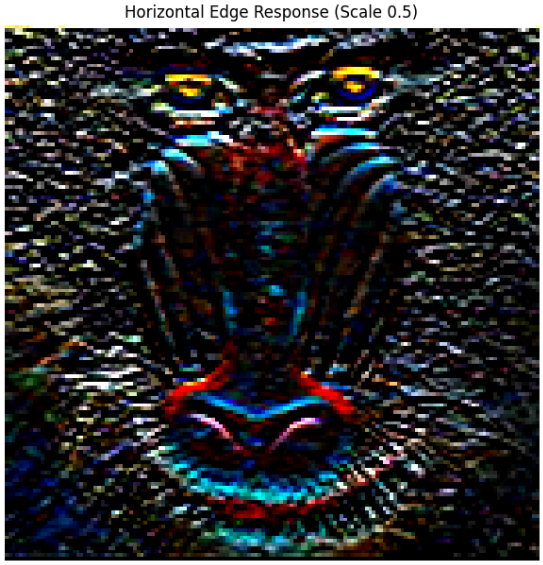
Horizontal Edges (Gaussian blur size=5, sigma=10)

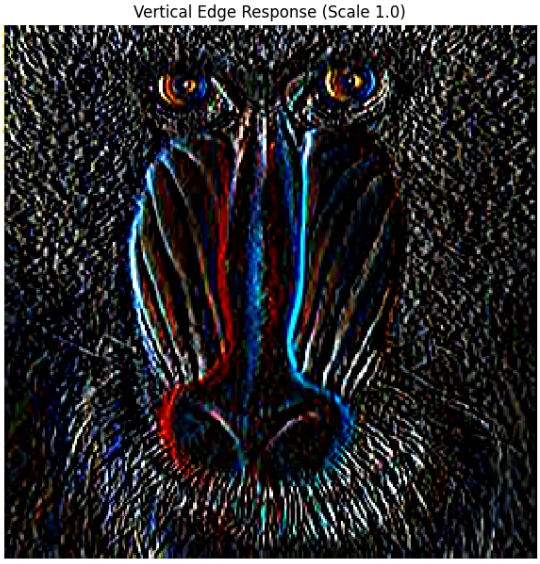


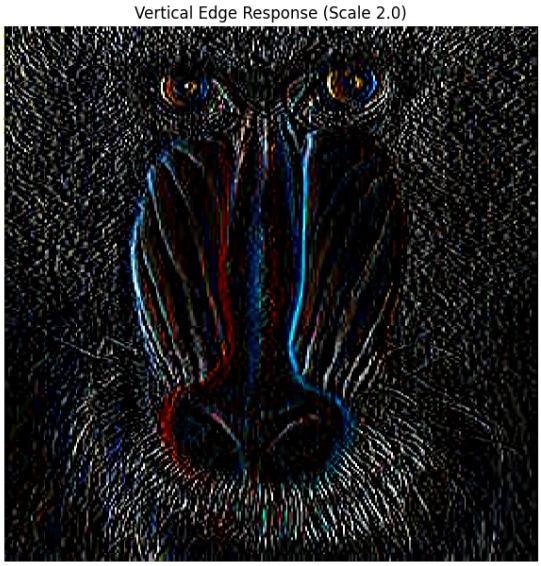
Vertical Edge Response (Scale 0.5)

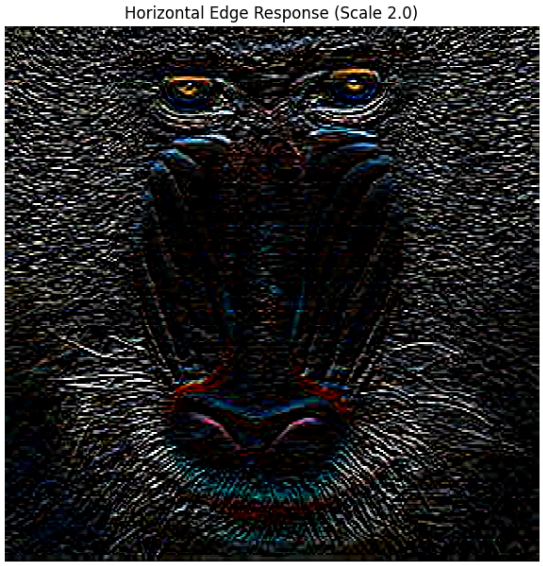


Horizontal Edge Response (Scale 0.5)









Your comments/analysis of your results here...

Discussion

Vertical and Horizontal Edge Detection

- Whipbird: Vertical edges strongly defined the bird and tree trunk, while horizontal edges highlighted layered foliage and finer leaf textures. The natural setting produced rich, complex edge maps but also introduced some clutter in textured areas.

- Cafe:Produced exceptionally clean and crisp vertical and horizontal edges. Architectural lines, windows, and signage appeared with high clarity. This image showcased ideal conditions for Sobel — as originally expected at question 1.

- Grayscale Fisherman: Vertical edges revealed the vertical garment folds, while horizontal edges captured the horizontal lines of the net. Most edges are observe on the fisherman with a clean background.

- Mandrill: Vertical gradients emphasized the central nose ridge and facial symmetry. Horizontal edges picked up intricate fur textures and color striping. The density of texture produced intense, high-frequency edge maps.

Combined Vertical and Horizontal Filtering

- Applying Sobel filtering in sequence behaves like a second-order gradient operation, highlighting regions where changes occur in both directions. This emphasizes corners, junctions, and other areas with strong perpendicular structure.
- Observation:
  - Whipbird: Some intersecting features like branch junctions and the bird's contour were preserved. However, background foliage detail was significantly lost, likely due to the soft, curved, and irregular nature of natural textures.
  - Cafe: Preserved corner-like features at doors, windows, and signs. Structured layout with clear perpendicular lines made the output relatively sharp and interpretable.
  - Grayscale fisherman: Enhanced the body figure due to intersecting gradients in clothing, limbs, and the net. Edge flow was clean and focused.
  - Mandrill: Produced dense, high-frequency noise from fur texture.

Combined Edge Magnitude and Direction

- Magnitude aggregates edge strength from both orientations, producing a more holistic edge map.
  - Mandrill: Strongest magnitude observed along nose ridge and fur contours, highlighting its high texture complexity.
  - Cafe: Clear outlines along structural features (windows, signs), though slightly less precise than individual directional maps.
  - Whipbird: Emphasized the bird and nearby branches but lost detail in soft background textures.
  - Grayscale Fisherman: Produced sharp, interpretable outlines of the figure with minimal background interference.
- Direction encodes the angle of edge orientation per pixel.
  - Whipbird and Mandrill: Edge direction appeared noisy and disordered reflecting complex textures and irregular features.
  - Cafe and Fisherman: More structured responses — dominated by horizontal/vertical or diagonal patterns tied to image geometry.

Positive and Negative Gradient Separation

- Positive vertical gradients enhance light-to-dark transitions (e.g., edges of shadows).
- Negative gradients were minimal or visually suppressed in all examples, with little interpretable structure compared to positive gradients.

Gaussian Blur Before Sobel

- Blurring smooths texture, reducing noise and isolating larger structural edges.
  - Mandrill: Fine fur detail suppressed; only major facial contours remained.

    - Cafe: Retained sharp architecture lines while eliminating small lighting variations.
    - Whipbird: Background clutter reduced; bird outline slightly more distinct.
    - Grayscale Fisherman: Reduced visibility of net threads and garment details.

Rescaling Before Sobel

- Downsampling (scale = 0.5): Loss of detail; only strong, large-scale edges detected.
- Original scale: Balanced detection of structural and fine features.
- Upscaling (scale = 2.0): Increased sensitivity to noise and texture; Sobel detected more fine details but with clutter.
- For each image:
    - Mandrill: At larger scale, fur textures dominated; smaller scales suppressed them.
    - Cafe: Architecture remained clear at all scales.
    - Whipbird: Strong edges persisted across scales, but fine foliage vanished when downsized.
    - Fisherman: human figure and net remained consistent; larger scales have more effect on fine texture.

Question 4: Image sampling and pyramids (30%)

4.1 Image Sampling

- Apply your `resize()` function to reduce an image (I) to 0.5*height and 0.5*width

- Repeat the above procedure, but apply a Gaussian blur filter to your original image before downsampling it. How does the result compare to your previous output, and to the original image? Why?

In [21]:
```
# Your answers to question 4 here

# 4.1 image sampling
image1 = load('images/whipbird.jpg')
image2 = load('images/cafe.jpg')
image3 = load('images/grayscale.jpg')
image4 = load('images/mandrill.jpg')

image1_size = image1.shape
image2_size = image2.shape
image3_size = image3.shape
image4_size = image4.shape

desired_size_image1 = (image1_size[0]//2, image1_size[1]//2)
desired_size_image2 = (image2_size[0]//2, image2_size[1]//2)
desired_size_image3 = (image3_size[0]//2, image3_size[1]//2)
desired_size_image4 = (image4_size[0]//2, image4_size[1]//2)

gaussian_kernel_5_1 = gauss2D(5, 1)
```

```
# Process whipbird image
plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.imshow(image1, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2, 2, 2)
image1_resized = resize(image1, desired_size_image1[0], desired_size_imag
plt.imshow(image1_resized, cmap='gray')
plt.title('Resized Image')
plt.axis('off')

plt.subplot(2, 2, 3)
image1_blurred = conv(image1, gaussian_kernel_5_1)
plt.imshow(image1_blurred, cmap='gray')
plt.title('Blurred Image')
plt.axis('off')

plt.subplot(2, 2, 4)
image1_resized_blurred = resize(image1_blurred, desired_size_image1[0], d
plt.imshow(image1_resized_blurred, cmap='gray')
plt.title('Blurred and Resized Image')
plt.axis('off')

plt.tight_layout()
plt.show()

# Process cafe image
plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.imshow(image2, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2, 2, 2)
image2_resized = resize(image2, desired_size_image2[0], desired_size_imag
plt.imshow(image2_resized, cmap='gray')
plt.title('Resized Image')
plt.axis('off')

plt.subplot(2, 2, 3)
image2_blurred = conv(image2, gaussian_kernel_5_1)
plt.imshow(image2_blurred, cmap='gray')
plt.title('Blurred Image')
plt.axis('off')

plt.subplot(2, 2, 4)
image2_resized_blurred = resize(image2_blurred, desired_size_image2[0], d
plt.imshow(image2_resized_blurred, cmap='gray')
plt.title('Blurred and Resized Image')
plt.axis('off')

plt.tight_layout()
plt.show()

# Process grayscale image
```

```
plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.imshow(image3, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2, 2, 2)
image3_resized = resize(image3, desired_size_image3[0], desired_size_imag
plt.imshow(image3_resized, cmap='gray')
plt.title('Resized Image')
plt.axis('off')

plt.subplot(2, 2, 3)
image3_blurred = conv(image3, gaussian_kernel_5_1)
plt.imshow(image3_blurred, cmap='gray')
plt.title('Blurred Image')
plt.axis('off')

plt.subplot(2, 2, 4)
image3_resized_blurred = resize(image3_blurred, desired_size_image3[0], d
plt.imshow(image3_resized_blurred, cmap='gray')
plt.title('Blurred and Resized Image')
plt.axis('off')

plt.tight_layout()
plt.show()

# Process mandrill image
plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.imshow(image4, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2, 2, 2)
image4_resized = resize(image4, desired_size_image4[0], desired_size_imag
plt.imshow(image4_resized, cmap='gray')
plt.title('Resized Image')
plt.axis('off')

plt.subplot(2, 2, 3)
image4_blurred = conv(image4, gaussian_kernel_5_1)
plt.imshow(image4_blurred, cmap='gray')
plt.title('Blurred Image')
plt.axis('off')

plt.subplot(2, 2, 4)
image4_resized_blurred = resize(image4_blurred, desired_size_image4[0], d
plt.imshow(image4_resized_blurred, cmap='gray')
plt.title('Blurred and Resized Image')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Original Image

Resized Image

Blurred Image

Blurred and Resized Image

Original Image

Resized Image

Blurred Image

Blurred and Resized Image

Original Image

Resized Image

Blurred Image

Blurred and Resized Image

Original Image

Resized Image

Blurred Image

Blurred and Resized Image

Discussion (for all test images)

- Resized Image (without blurring): The direct downsample retained sharp transitions but introduced aliasing—this is most evidence in smaller size images like whipbird and mandrill.
- Blurred and Resized Image: Smoother, more natural appearance with fewer artifacts. The blur effectively suppressed high-frequency noise before resizing, resulting in a more visually coherent image at lower resolution.
- Reasons: When downsampling without pre-blur, high-frequency details that can't be represented at the smaller scale fold into lower frequencies, creating aliasing artifacts. Pre-blurring acts as a low-pass filter, removing those frequencies beforehand (also known as anti-aliasing filter).
- Visual Comparison:
  - Resized: retains crispness/sharpness but appears harsher, especially around sharp features.
  - Blurred + Resized: looks softer but cleaner, with improved structural consistency in complex regions.

4.2 Image Pyramids

- Create a Gaussian pyramid as described in week2's lecture on an image.
- Apply a Gaussian kernel to an image I, and resize it with ratio 0.5, to get I_1. Repeat this step to get I_2, I_3 and I_4.
- Display these four images in a manner analogus to the example shown in the lectures.

In [22]:
```
## 4.2 pyramid
# Create Gaussian pyramid for image1 (whipbird)
gaussian_kernel = gauss2D(5, 1)

# Original image I
I = load('images/whipbird.jpg')
height, width = I.shape[:2]

# I1 - First level (1/2 size)
I1 = conv(I, gaussian_kernel)
I1 = resize(I1, height//2, width//2)

# I2 - Second level (1/4 size)
I2 = conv(I1, gaussian_kernel)
I2 = resize(I2, height//4, width//4)

# I3 - Third level (1/8 size)
I3 = conv(I2, gaussian_kernel)
I3 = resize(I3, height//8, width//8)
```

```python
# I4 - Fourth level (1/16 size)
I4 = conv(I3, gaussian_kernel)
I4 = resize(I4, height//16, width//16)

display(I, 'Original Image')

# Display pyramid
pyramid = [I1, I2, I3, I4]

# Calculate canvas size
# Create canvas with spacing between images
spacing = 10
total_width = sum(img.shape[1] for img in pyramid) + spacing * (len(pyram
max_height = max(img.shape[0] for img in pyramid)
canvas = np.ones((max_height, total_width, 3), dtype=np.float32)

# Paste images with spacing
x = 0
for img in pyramid:
    h, w = img.shape[:2]
    canvas[0:h, x:x+w] = img
    x += w + spacing

# Display pyramid
plt.figure(figsize=(16, 6))
plt.imshow(canvas)
plt.axis('off')
plt.title("Gaussian Pyramid")
plt.show()
```

Original Image

Your comments/analysis of your results here...

Discussion

- Each level in the pyramid is both blurred and downsampled, reducing resolution and detail progressively.
- Gaussian blur at each step prevents aliasing, preserving structural consistency across scales.
- As levels go up, fine textures fade, and only large-scale structures remain visible.
- This multi-scale representation is useful for detecting features at different resolutions (example: blobs or edges).

Question 5: (optional, assesed for granting up to 20% bonus marks for the A1)

Image filtering lectures, particularly Lecture 2, have covered the details related to this question. This is a bonus question for the students to get opportunities to recover lost marks in the other parts of the assignment. **Note that the overall marks will be capped at 100%**.

5.1 Apply and analyse a blob detector

- Create a Laplacian of Gaussian (LoG) filter in the function `LoG2D()` and visualize its response on your images. You can use the template function (and hints therein) for the task if you wish.

- Modify parameters of the LoG filters and apply them to an image of your choice. Show how these variations are manifested in the output.

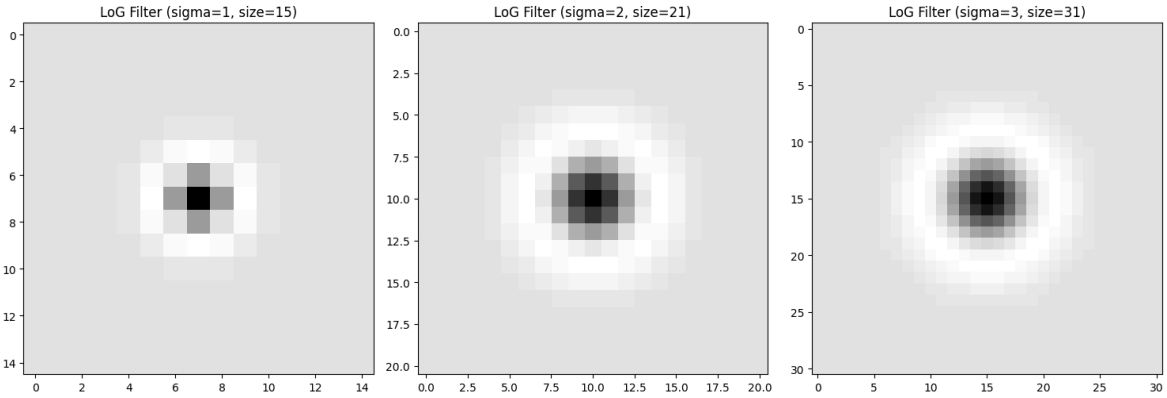
- Repeat the experiment by rescaling the image with a combination of appropriate filters designed by you for these assignment. What correlations do you find when changing the scale or modifying the filters?

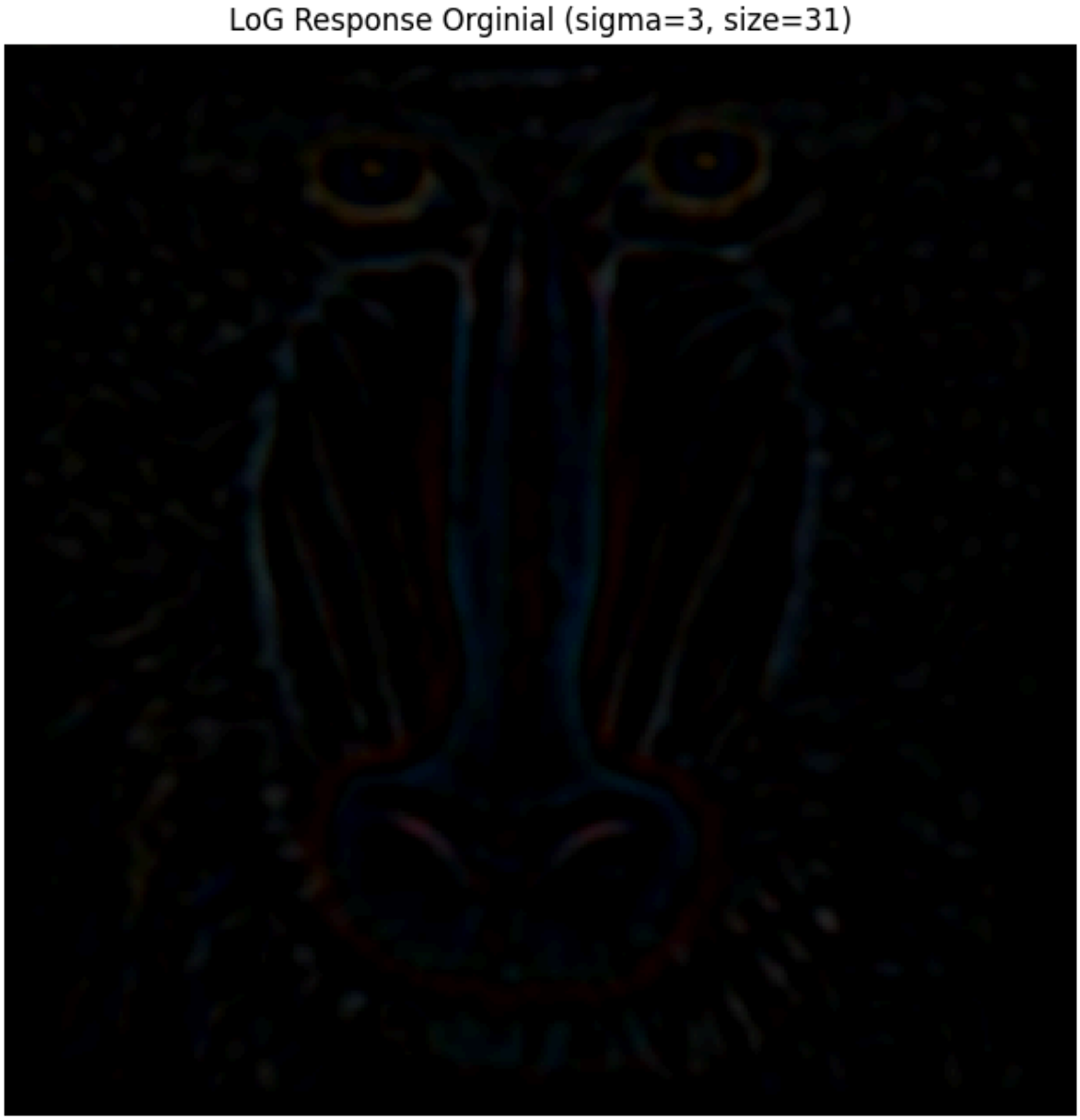
- How does the response of LoG filter change when you rotate the image by 90 degrees? You can write a function to rotate the image or use an externally rotated image for this task.

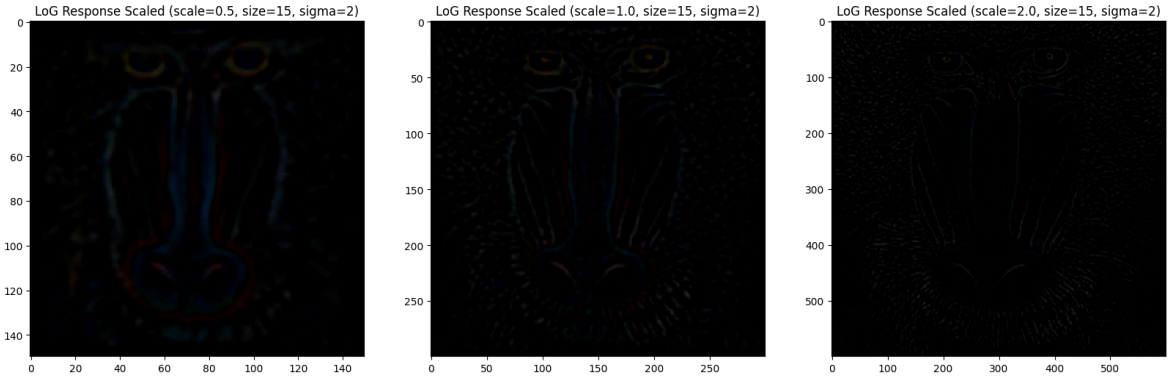
In [23]:

```
# Your code to answer question 5 and display results here

sigmas = [1, 2, 3]
sizes = [15, 21, 31]

# Visualize LoG filters with different sizes and sigmas
plt.figure(figsize=(15, 5))
for i, (sigma, size) in enumerate(zip(sigmas, sizes)):
    log_filter = LoG2D(size, sigma)
    plt.subplot(1, 3, i+1)
    plt.imshow(log_filter, cmap='gray')
    plt.title(f'LoG Filter (sigma={sigma}, size={size})')
plt.tight_layout()

# Apply LoG filter
image = load('images/mandrill.jpg')
log_response = conv(image, LoG2D(31, 3))
display(log_response, 'LoG Response Orginial (sigma=3, size=31)')

# Test different scales
scales = [0.5, 1.0, 2.0]
plt.figure(figsize=(20, 12))
for i, scale in enumerate(scales):
    # Resize image
    new_rows = int(image.shape[0] * scale)
    new_cols = int(image.shape[1] * scale)
    scaled_image = resize(image, new_rows, new_cols)

    # Apply LoG filter with fixed parameters (for consistency)
    size = 15
    sigma = 2
    log_response_scaled = conv(scaled_image, LoG2D(size, sigma))
    plt.subplot(1, 3, i+1)
    plt.imshow(log_response_scaled, cmap='gray')
    plt.title(f'LoG Response Scaled (scale={scale}, size={size}, sigma={s

# Test rotation
def rotate_image_90(image):
    """Rotate image 90 degrees clockwise"""
    return np.rot90(image, k=-1)

# Original and rotated responses
plt.figure(figsize=(15, 5))

# Original
log_response_original = conv(image, LoG2D(15, 2))
rotated_image = rotate_image_90(image)
log_response_rotated = conv(rotated_image, LoG2D(15, 2))

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(log_response_original, cmap='hot')
axes[0].set_title('Original')

axes[1].imshow(log_response_rotated, cmap='hot')
axes[1].set_title('Rotated')

fig.suptitle('LoG Response Original and Rotated (size=15, sigma=2)')
```

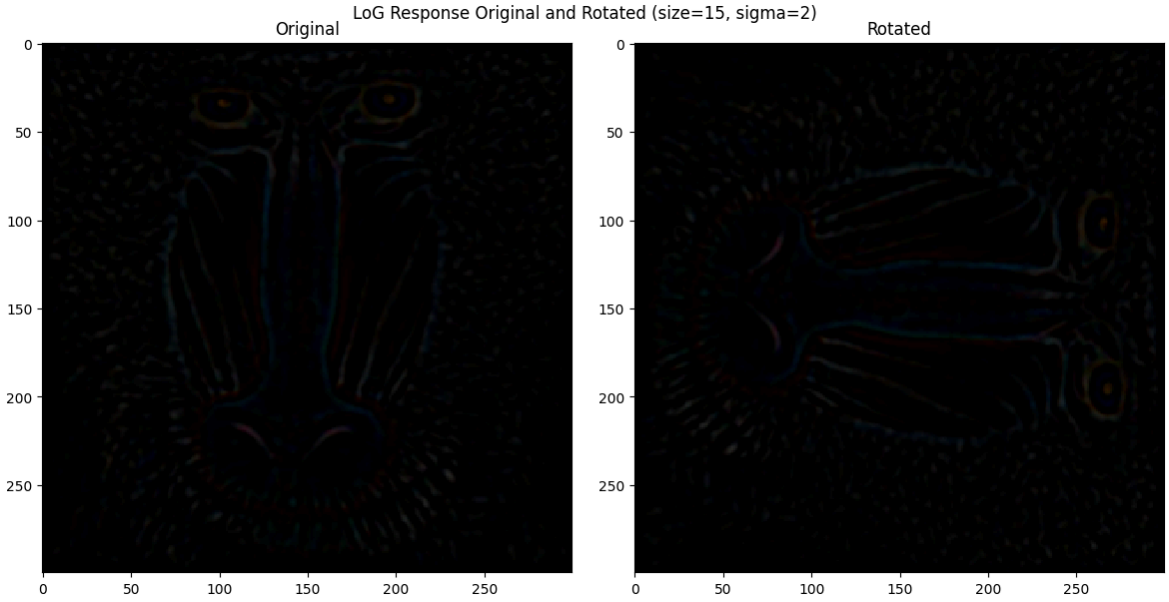
```
plt.tight_layout()
plt.show()
```








```
<Figure size 1500x500 with 0 Axes>
```



Discussion:

LoG filter visualization

- LoG filters resemble a hat with a central dip and surrounding ring.
- Larger sigma creates broader, smoother filters for detecting larger structures.
- Larger kernel size provides more context and prevents edge truncation for large sigma.

Respone on the image

- Applying a LoG filter (sigma=3, size=31) highlights circular blob-like textures such as the nostrils, fur patterns, and eye edges.

LoG Filter Response Across Scales

- Scale 0.5×:The image is downsampled, so image features appear larger relative to the fixed-size LoG filter. As a result, the filter detects larger blobs and coarse structures, but finer details are suppressed.Edges like the nostrils and eyes are more prominent, while fur texture is lost.
- Scale 1.0×: Filter and feature sizes are balanced. Both medium-sized blobs and finer texture elements are visible. This produces the clearest and most informative response — ideal scale for this filter configuration.

- Scale 2.0×: Image is enlarged, and features become smaller relative to the fixed LoG kernel. The filter now responds more to small-scale textures, especially the dense fur patterns. Larger blobs (example: nose structure) are less emphasized.

Effect of 90 degree rotation

- Rotating the image had minimal impact on the LoG response pattern.
- This confirms that the LoG filter is rotation-invariant — its circularly symmetric structure ensures similar response regardless of orientation.
- Feature positions shift due to image rotation, but their shape and contrast strength in the response remain consistent.