# Trabalho 1

Introdução ao Processamento Digital de Imagem

Ieremies Vieira da Fonseca Romero

## Introdução

O processamento digital de imagens é uma área de estudo fundamental em Ciência da Computação, que visa o tratamento e manipulação de imagens digitais por meio de técnicas e algoritmos computacionais. A análise e processamento de imagens digitais é uma tarefa complexa e multidisciplinar, que envolve conhecimentos em matemática, estatística, computação gráfica, processamento de sinais e outras áreas correlatas.

O objetivo deste trabalho é apresentar uma série de processamentos básicos em imagens digitais, abordando desde a leitura e escrita de imagens até operações de filtragem. Para o desenvolvimento das operações, será empregada a vetorização de comandos, que permite processar as imagens de forma mais eficiente e rápida, através da aplicação de operações vetoriais em vez de operações matriciais.

## O programa

Neste trabalho, utilizamos as bibliotecas `numpy` 1.24.1 e OpenCV 4.7.0.72 utilizado via `cv2`.

```
import numpy as np
import cv2
```

Há duas formas de utilizar o código deste projeto:

**script.py** Executa todas as funções em todas as imagens e salva o seus respectivos resultados.

**iterativo** utilizando `python -i funcs.py`, é possível interagir com cada uma das funções no terminal.

Cada uma das questões do enúnciado é resolvido por uma função contida no arquivo `funcs.py`. Todas as funções possuem como primeiro parâmetro a imagem sob a qual ela irá atuar. Algumas funções possuem o segundo parâmetro, cujos significados são resumidos na tabela .

| Função | Significado do segundo argumento |
|---|---|
| `mosaico` | ordem a ser rearrumada. |
| `combinacao_imagem` | segunda imagem a ser combinada. |
| `ajuste_brilho` | gamma a ser utilizado na operação. |
| `quantizacao` | quantidade de níveis da imagem resultante. |
| `planos_bit` | qual plano queremos. |
| `filtragem` | qual filtro aplicar. |

Para leitura e escrita de imagens, utilizamos a biblioteca `cv2` para ler e salvar imagens, como no código apresentado abaixo.

```
# Read any image to a numpy array
img_colored = cv2.imread('image.png')
# Read a grayscale image
img_gray = cv2.imread('image.png', cv2.IMREAD_GRAYSCALE)
# Save a numpy array to image file
cv.imwrite('out.png', img)
```

## Mosaico

Para essa função recebemos tanto a imagem `img` quanto a ordem `order`. Esta ordem é semelhante a descrita em uma ordenação dos 16 quadrantes, da esquerda para direita, de cima pra baixo. Assim, o $i$-ésimo valor $order_i$ indica que, na nova imagem, o quadrante $i$ aparece no lugar do antigo quadrante $order_i$.

No código, primeiro dividimos a imagem nos 16 quadrantes utilizando a função `np.split` passando como parâmetro 4 partes e eixos diferentes. Depois, é necessário alterar os índices já que, em python, listas são indexadas a partir do zero. Realizamos a conversão de ordem para coordenadas na matriz sabendo que o $i$-ésimo está na linha $i//4$ (divisão inteira) e na coluna $i\%4$ (resto). Por fim, juntamos de volta os quadrantes utilizando a função `np.concatenate` e retornamos a nova imagem.

```
def mosaico(img, order):
    # Divide in 4 horizontal strips (lines)
```

```
lines = np.split(img, 4, axis=0)
# Divide verticaly each line in 4 pieces (squares)
squares = [np.split(line, 4, axis=1) for line in lines]
# Placehold for the new order of squares
new_squares = [[0 for _ in range(4)] for _ in range(4)]
for k in range(16):
    i = order[k] - 1  # make the order start on index zero
    # Convert the order to the coordenate in the matrix
    new_squares[k // 4][k % 4] = squares[i // 4][i % 4]
# Recombine to form the new horizontal lines
new_lines = [np.concatenate(new_square, axis=1) for new_square in new_squares]
# Recombine the horizontal lines into the new image
new_img = np.concatenate(new_lines, axis=0)
return new_img
```

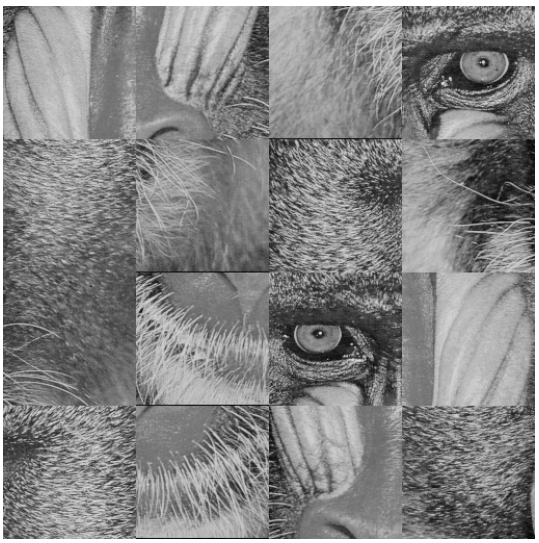

Figura 1: Reorganização das partes da imagem “baboon.png” na mesma ordenação proposta no enunciado.

## Combinação de imagem

Utilizando a vetorização, podemos combinar as imagens `img1` e `img2`, cada uma sendo multiplicada por 0.5.

```
def combinacao_imagem(img1, img2):
    return 0.5 * img1 + 0.5 * img2
```

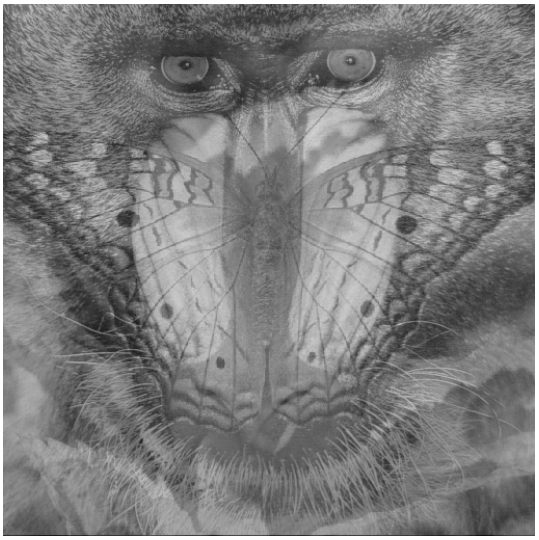

Figura 2: Combinação da imagem “baboon.png” e “butterfly.png”.

## Transformação de intensidade

### Negativo

Podemos atingir o resultado desejado utilizando a operação `np.invert`.

```
def negativo(img):
    return np.invert(img)
```

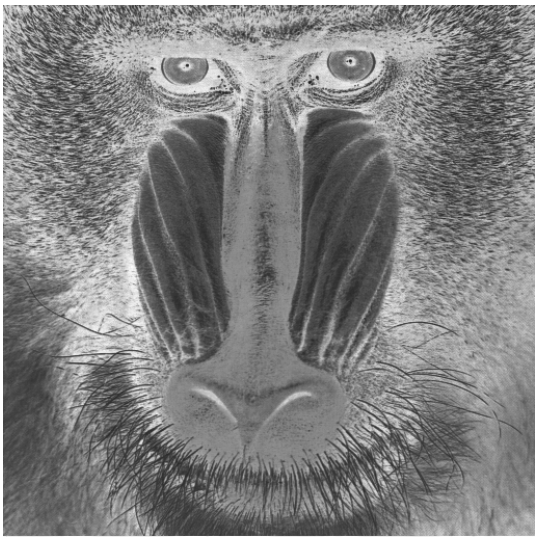

Figura 3: Negativo da imagem "baboon.png".

### Converter para o intervalo $[100, 200]$.

Para isso, entendemos que cada valor da imagem original está a uma proporção de 0 a 255. Calculamos tal proporção e transpomos a mesma para o intervalo 100 a 200.

```
def converter_para_intervalo(img):
    return 100 + (img / 255) * 100
```

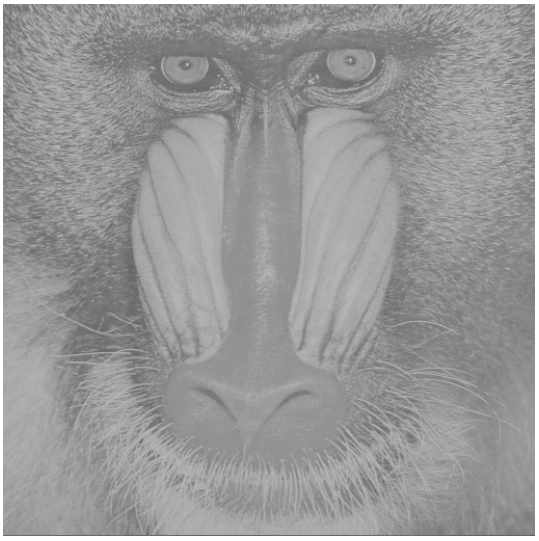

Figura 4: Converção proporcional dos valores da imagem "baboon.png" ao intervalo $[100, 200]$.

### Linhas pares invertidas

Nesse caso, primeiro selecionamos as linhas pares, utilizando o parâmetro `step` de listas e, de forma similar, invertemo-nas utilizando `-1`.

```
def linhas_pares_invertidas(img):
    linhas_pares = img[::2]         # seleciona as linhas pares
    img[::2] = linhas_pares[:,::-1] # inverte as linhas
    return img
```

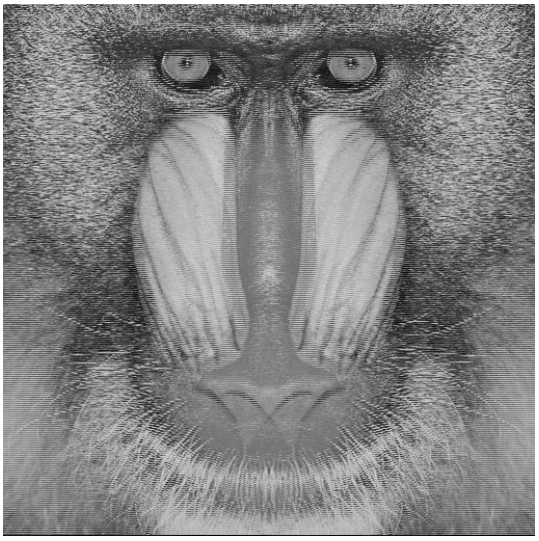

Figura 5: Linhas pares da imagem “baboon.png” invertidas.

### Reflexão de linhas

Nesse, cortamos a imagem na metade, copiamos a imagem e invertemos essa. Por fim, salvamos a concatenação na vertical das imagens.

```
def reflexao_linhas(img):
    n, m = img.shape
    part1 = img[:m//2:]
    part2 = part1[::-1]
    return np.concatenate((part1, part2), axis=0)
```

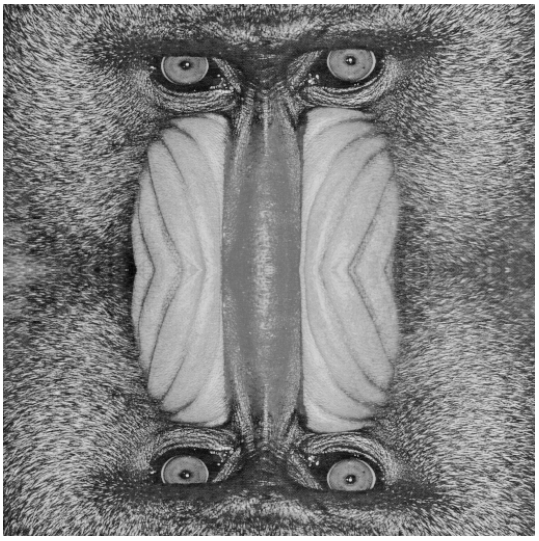

Figura 6: Reflexão das linhas da imagem “baboon.png”

### Espelhamento vertical

Utilizando o `step` do parâmetro de listas em python, podemos inverter a matriz.

```
def espelhamento_vertical(img):
    return img[::-1]
```

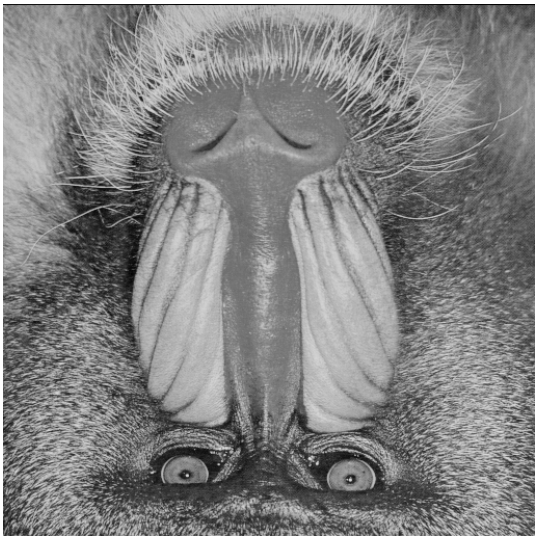

Figura 7: Espelhamento vertical da imagem "baboon.png"

# Imagens coloridas

## Colorida para colorida

Cada pixel é representado por um vetor de três dimensões, cada uma para cada valor dos canais RGB. Para aplicar a tranformação proposta, podemos utilizar de uma matriz para representa-la, como descrita no código pela variável `A`. Assim, aplicamos o produto de matrizes via `np.dot` e cada pixel e substituimos o novo vetor de três dimensões como novo pixel. Por fim, garantimos que todos os valores de canais de todos os pixels estão no intervalo de $[0, 255]$.

```
def colorida_colorida(img):
    # Set up the transformation matrix
    A = np.array([[0.393, 0.769, 0.189],
                  [0.349, 0.686, 0.168],
                  [0.272, 0.534, 0.131]])
    # Multiply each pixel by the transformation matrix
    img = np.dot(img, A)
    # Limit pixel values to the range [0, 255]
    return np.clip(img, 0, 255)
```

Figura 8: Transformação descrita aplicada à imagem "lena.png".

### Coloridas para cinza

Cada pixel é representado por um vetor de três dimensões, cada uma para cada valor dos canais RGB. Para aplicar a tranformação proposta, podemos utilizar um vetor para representa-la, como descrita no código pela variável `A`. Assim, aplicamos o produto de matrizes via `np.dot` e cada pixel e substituimos o novo pixel pelo escalar resultante. Por fim, garantimos que os valores de todos os pixels estão no intervalo de $[0, 255]$.

```
def colorida_cinza(img):
    # Set up the transformation matrix
    A = np.array([0.2989, 0.5870, 0.1140])
    # Multiply each pixel by the transformation matrix
    img = np.dot(img, A)
    # Limit pixel values to the range [0, 255]
    return np.clip(img, 0, 255)
```

Figura 9: Transformação à escala de cinza aplicada à imagem "lena.png".

## Ajuste de brilho

Neste caso, recebemos a imagem e um certo valor de gamma como parâmetros. Utilizando vetorização, aplicamos diretamente a fórmula $B = A^{\frac{1}{\gamma}}$. Por fim, apenas garantimos que os valores de cada pixel estão proporcionamente no intervalo $[0, 255]$.

```
def ajuste_brilho(img, gamma):
    img = (img / 255) ** (1 / gamma)
    factor = 255 / np.max(img)
    return img * factor
```

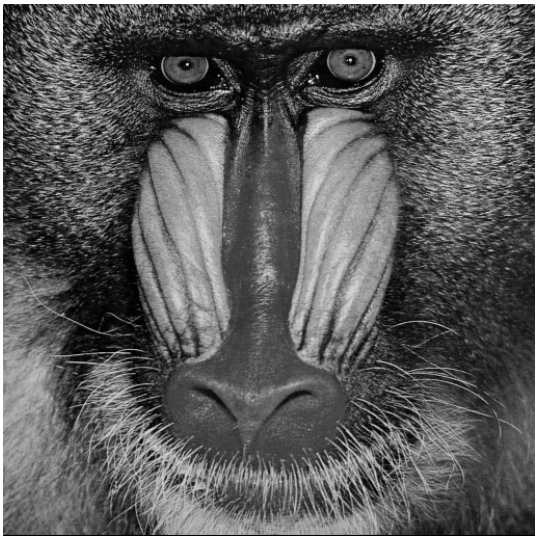

Figura 10: Ajuste de brilho com gamma 0.5 aplicado à imagem "baboon.png".

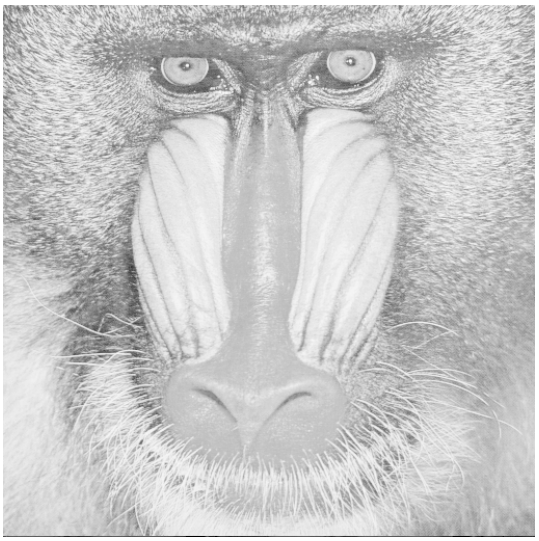

Figura 11: Ajuste de brilho com gamma 2.5 aplicado à imagem "baboon.png".

## Quantização de imagens

Para reduzir a quantidade de níveis de cinza à $L$, definimos que cada valor está a uma certa proporção do intervalo de 0 a 255. Mantemos a mesma proporção no intervalo $[0, L-1]$ e arredondamos ao nível inteiro mais próximo. Por fim, retornamos ao formato de 256 mas agora com, na prática, apenas $L$ níveis de cinza

```
def quantizacao(img, l):
    img = (img / 255) * (l - 1)
    img = np.around(img)
    return img * (256/(l-1))
```

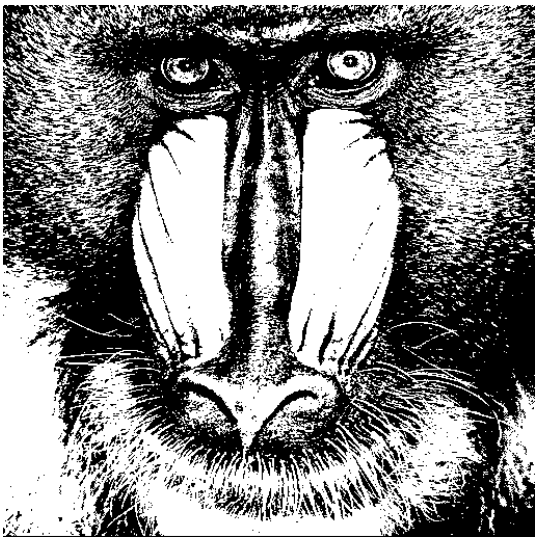

Figura 12: Quantização da imagem "baboon.png" em 2 níveis.

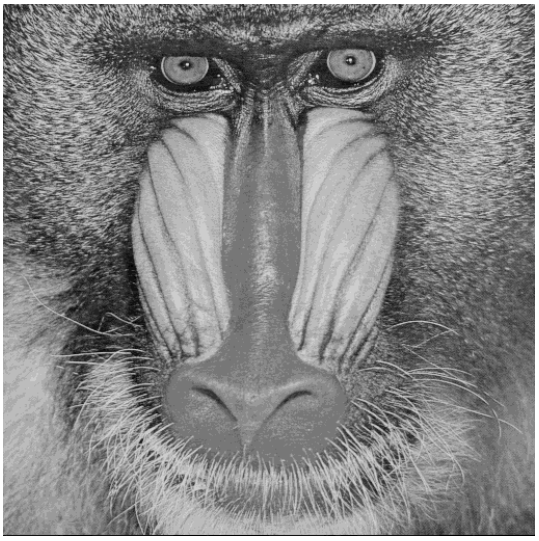

Figura 13: Quantização da imagem “baboon.png” em 16 níveis.

## Planos de bits

Para extrairmos cada plano de bit, como definido no enunciado, precisamos pegar o $i - textsimo$ bit da esquerda para direita. Assim, o $i$ plano, fazemos `(img » i) & 1` e salvamos cada um desses planos numa lista. Na hora de salvar a imagem, precisamos multiplicar por 255, já que cada plano de bit é uma array de 0 e 1.

```
def planos_bit(img, plano):
    bit_planes = [np.uint8((img >> bit) & 1) for bit in range(8)]
    return bit_planes[plano] * 255
```

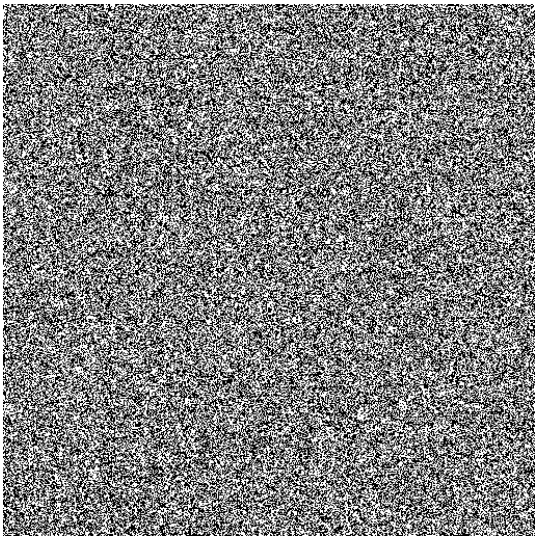

Figura 14: Primeiro plano de bits da imagem “baboon.png”.

Figura 15: Quinto plano de bits da imagem “baboon.png”.

## Filtragem de imagens

```
# Definição das matrizes
h1 = np.asarray([[0, 0, -1, 0, 0],
                 [0, -1, -2, -1, 0],
                 [-1, -2, 16, -2, -1],
                 [0, -1, -2, -1, 0],
                 [0, 0, -1, 0, 0]])
h2 = np.asarray([[1, 4, 6, 4, 1],
                 [4, 16, 24, 16, 4],
                 [6, 24, 36, 24, 6],
                 [4, 16, 24, 16, 4],
                 [1, 4, 6, 4, 1]])
h2 = h2 / 256
h3 = np.asarray([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
h4 = np.asarray([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
h5 = np.asarray([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
h6 = np.asarray([[1, 1, 1], [1, 1, 1], [1, 1, 1]]) / 9
h7 = np.asarray([[-1, -1, 2], [-1, 2, -1], [2, -1, -1]])
h8 = np.asarray([[2, -1, -1], [-1, 2, -1], [-1, -1, 2]])
h9 = np.identity(9) / 9
h10 = np.asarray([[-1, -1, -1, -1, -1],
                  [-1, 2, 2, 2, -1],
                  [-1, 2, 8, 2, -1],
                  [-1, 2, 2, 2, -1],
                  [-1, -1, -1, -1, -1]])
h10 = h10 / 8
h11 = np.asarray([[1, -1, 0], [-1, 0, 1], [0, 1, 1]])
```

Para cada filtro, definimos um kernel, uma matriz que representa nosso filtro. Para aplicarmos o filtro, passamos a imagem bem como o filtro para a função `cv2.filter2D`. Vale a pena ressaltar o argumento `-1` da função, este especifica que a saída deve ter a mesma profundidade que a entrada.

Outro fato interessante de comentar é sobre as decisões de "padding". Quando aplicamos um filtro como os descritos aqui, precisamos decidir o que fazer nos pixels de borda, já que estes não possuem todos os vizinhos como os demais pixels. A função `cv2.filter2D` aplica o chamado "zero-padding", na qual são adicionados zeros às bordas conforme necessário.

```
def filtragem(img, h):
    return cv2.filter2D(img, -1, h)
```

Por questões de espaço, neste relatório iremos reproduzir os resultados de alguns filtros. Todos os resultados estão disponíveis na pasta `out`.

Figura 16: Filtro $h_3$ aplicado à imagem "baboon.png".

Figura 17: Filtro $h_4$ aplicado à imagem "baboon.png".

Figura 18: Filtro $h_9$ aplicado à imagem "baboon.png".

Após executar os filtros observamos os seguintes efeitos de cada um:

**h1** passa-alta que realça as bordas.

**h2** de suavização (blur) que reduz o ruído.

**h3** detecta bordas no sentido horizontal.

**h4** detecta bordas no sentido vertical.

**h5** realça as bordas.

**h6** de suavização (blur) que reduz o ruído e as características.

**h7** detecta bordas diagonais.

**h8** detecta bordas diagonais.

**h9** de suavização (blur) diagonalmente.

**h10** de nitidez que realça as características.

**h11** aumenta o brilho (efeito de gloom).

Por fim, como sugerido no enunciado, podemos combinar o kernel $h3$ com o $h4$. Utilizamos as funções `np.square` e `np.sqrt` para realizar as

operações aritméticas vetorizadas. Vemos que a combinação de dois filtros que detectavam bordas em sentidos diferentes resultam em um bom filtro para detectar bordas em ambos.

```
def filtragem_h3_h4(img):
    img = img.astype(np.float32)
    res = np.square(filtragem(img, h3))
    res += np.square(filtragem(img, h4))
    res = np.sqrt(res)
    return res
```

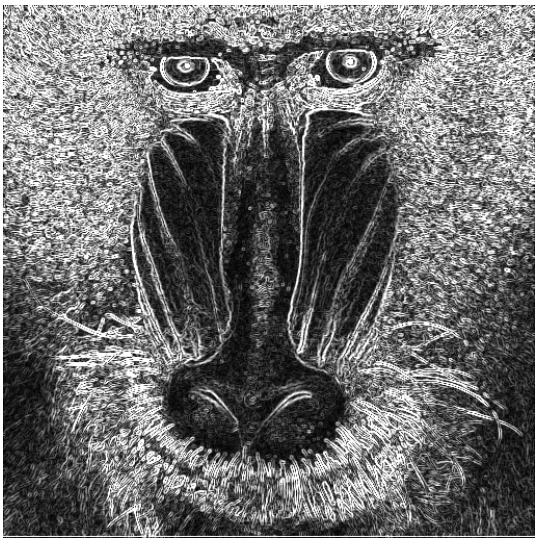

Figura 19: Resultado do filtro $h_3$ combinado com o resultado do filtro $h_4$ na imagem "baboon.png"