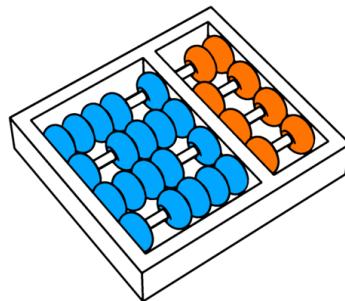

Manipulação de imagens com Python

UNIVERSIDADE ESTADUAL DE CAMPINAS

ENGENHARIA DE COMPUTAÇÃO



Autor:
José Eduardo Santos Rabelo (260551)

Data: April 7, 2025

Trabalho de Processamento de Imagens

José Eduardo Santos Rabelo

April 7, 2025

Contents

1 Questão 1 - Esboço a Lápis	2
1.1 Código em Python	2
1.2 Explicação do código	2
1.2.1 Por que funciona a divisão?	3
1.3 Resultado obtido no código	4
2 Questão 2 - Ajuste de Brilho	5
2.1 Código em Python	5
2.2 Explicação do ajuste	5
2.3 Resultado obtido no código	6
3 Questão 3 - Mosaico	7
3.1 Código em Python	7
3.2 Explicação do código	7
3.2.1 Entendendo o reshape	7
3.2.2 Entendendo o transpose	8
3.2.3 Entendendo a formação do mosaico	8
3.3 Resultado obtido no código	8
4 Questão 4 - Alteração de Cores	9
4.1 Código em Python	9
4.2 Explicação do código	10
4.2.1 Operação dot da biblioteca Numpy	10
4.2.2 Por que o peso do campo G é maior?	10
4.3 Resultado obtido no código	11
5 Questão 5 - Transformação de Imagens Coloridas	12
5.1 Código em Python	12
5.2 Explicação do código	12
5.2.1 Transformação do efeito	12
5.2.2 Transformação em tons de cinza	13
5.3 Resultado obtido no código	13

6 Questão 6 - Plano de Bits	14
6.1 Código em Python	14
6.2 Explicação do código	15
6.2.1 Operação bitwise_and da biblioteca Numpy	15
6.2.2 Mas e o quê é obtido no final?	16
6.3 Resultado obtido no código	16
7 Questão 7 - Combinação de Imagens	17
7.1 Código em Python	17
7.2 Explicação do código	17
7.2.1 Como foi feita a combinação	17
7.3 Resultado obtido no código	18
8 Questão 8 - Transformação de Intensidade	19
8.1 Explicação do código	19
8.1.1 Imagem Negativa	19
8.1.2 Transformação de Intensidade para o Intervalo [100, 200]	20
8.1.3 Inversão das Linhas Pares da Imagem	20
8.1.4 Espelhamento da Metade Superior da Imagem na Parte Inferior .	21
8.1.5 Espelhamento Vertical da Imagem	21
8.2 Resultado obtido no código	22
9 Questão 9 - Quantização de Imagens	23
9.1 Código em Python	23
9.2 Explicação do código	23
9.2.1 Quantização de Imagens Monocromáticas	23
9.3 Resultado obtido no código	24
10 Questão 10 - Filtragem das Imagens	25
10.1 Código em Python	25
10.2 Explicação do código	27
10.2.1 Função convolve da biblioteca <code>scipy.ndimage</code>	27
10.2.2 Função <code>aplicar_filtro</code>	27
10.2.3 Filtro h_1	28
10.2.4 Filtro h_2	28
10.2.5 Filtro h_3	28
10.2.6 Filtro h_4	29
10.2.7 Filtro h_5	29
10.2.8 Filtro h_6	29
10.2.9 Filtro h_7	30
10.2.10 Filtro h_8	30
10.2.11 Filtro h_9	30
10.2.12 Filtro h_{10}	31
10.2.13 Filtro h_{11}	31
10.2.14 Combinação dos filtros h_3 e h_4	32
10.3 Resultado obtido no código	32

1 Questão 1 - Esboço a Lápis

1.1 Código em Python

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.ndimage import gaussian_filter
4 from PIL import Image
5
6 img = np.array(Image.open('watch.png').convert('RGB'))
7
8 def rgb2gray(rgb):
9     rgb_img = np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])
10    rgb_img = np.interp(rgb_img, (rgb_img.min(), rgb_img.max()), (0, 255))
11    rgb_img = rgb_img.astype(np.uint8)
12    return rgb_img
13
14 gray = rgb2gray(img)
15 blurred = gaussian_filter(gray, sigma=21/6)
16
17 fig, axs = plt.subplots(1, 3, figsize=(15, 5))
18
19 # Subplot 1: Imagem em tons de cinza
20 axs[0].imshow(gray, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
21 axs[0].set_title('Tons de Cinza')
22 axs[0].axis('off')
23
24 # Subplot 2: Imagem desfocada
25 axs[1].imshow(blurred, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
26 axs[1].set_title('Imagen Desfocada')
27 axs[1].axis('off')
28
29 # Subplot 3: Imagem em tons de cinza dividida pela imagem desfocada
30 result = np.clip((gray / blurred) * 255, 0, 255).astype(np.uint8)
31
32 axs[2].imshow(result, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
33 axs[2].set_title('Imagen em Tons de Cinza Dividida pela Imagen Desfocada')
34 axs[2].axis('off')
35
36 plt.tight_layout()
37 plt.show()

```

1.2 Explicação do código

Função gaussian_filter do scipy.ndimage

Documentação: `gaussian_filter(input, sigma, order=0, output=None, mode='reflect', cval=0.0, truncate=4.0, *, radius=None, axes=None)`

- `sigma`: Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

A distribuição gaussiana tem 99.7% de sua área dentro de $\pm\sigma$ do centro. Dessa forma, um kernel de largura 6σ cobre praticamente toda influência do filtro. Adicionando +1 para ter um pixel central. Como foi sugerido que utilizássemos um kernel 21×21 , temos que:

$$6 \cdot \sigma + 1 = 21 \Rightarrow \sigma = \frac{21}{6}$$

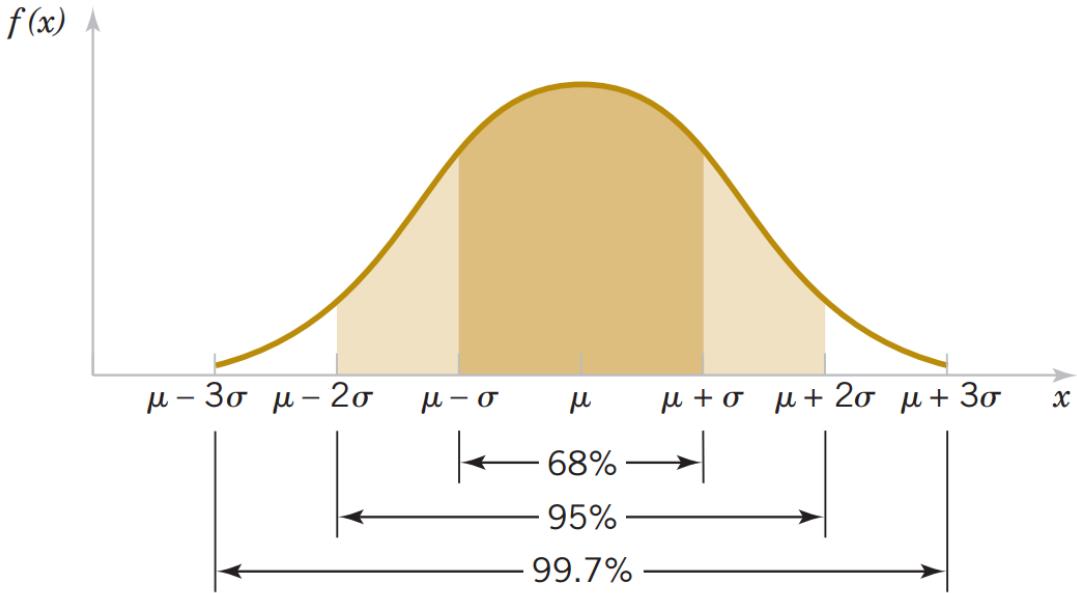


Figure 1: Gaussiana

A função **rgb2gray** foi desenvolvida para converter a imagem RGB (Red, Green, Blue) para uma imagem com escala de cinza usando uma combinação linear ponderada dos sinais de cores. A conversão segue uma recomendação ITU-R BT.601 para transformação RGB para luma (Y):

$$Y = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

1.2.1 Por que funciona a divisão?

O filtro gaussiano suaviza as bordas e remove detalhes considerados finos. Dessa forma, ele gera uma imagem com frequência mais baixa que a original. Portanto:

- Em regiões onde $gray \approx blurred$ (regiões mais uniformes), a razão é aproximadamente 1. Ao retomarmos para o intervalo $[0, 255]$ as regiões uniformes se tornam brancas (ou perto disso).
- Em regiões de borda e detalhes, teremos $gray \neq blurred$, pois o filtro gaussiano suaviza mais intensamente essas transições abruptas. Dessa forma:
 - Nas bordas **escuras** (onde $gray < blurred$), a razão será menor que 1, resultando em tons mais escuros após a normalização
 - Nas bordas **claras** (onde $gray > blurred$), a razão será maior que 1, mas é limitada a 255 pelo clipping
- O efeito combinado produz:
 1. Fundo branco (áreas uniformes)
 2. Linhas escuras nas bordas (transições de intensidade)
 3. Textura semelhante a traços de lápis devido à variação local da razão



Figure 2: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

1.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-1.py](#)

2 Questão 2 - Ajuste de Brilho

2.1 Código em Python

```

1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Função para corrigir o gamma de uma imagem
6 def correct_gamma(img, gamma):
7     # Aplica a correção de gamma na imagem
8     img_corrected = np.power(np.divide(img, 255), np.reciprocal(gamma))
9     return np.multiply(img_corrected, 255).astype(np.uint8)
10
11 # Carrega a imagem em escala de cinza
12 img = np.array(Image.open('baboon_monocromatica.png').convert('L'))
13
14 # Cria uma figura com 1 linha e 5 colunas para exibir as imagens
15 fig, axs = plt.subplots(1, 5, figsize=(25, 5))
16
17 # Exibe a imagem original
18 axs[0].imshow(img, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
19 axs[0].set_title('Original')
20 axs[0].axis('off')
21
22 # Aplica a correção de gamma com diferentes valores
23 img1 = correct_gamma(img, 0.5)
24 img2 = correct_gamma(img, 1.0)
25 img3 = correct_gamma(img, 1.5)
26 img4 = correct_gamma(img, 2.0)
27
28 # Exibe a imagem com gamma = 0.5
29 axs[1].imshow(img1, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
30 axs[1].set_title('γ = 0.5')
31 axs[1].axis('off')
32
33 # Exibe a imagem com gamma = 1.0
34 axs[2].imshow(img2, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
35 axs[2].set_title('γ = 1.0')
36 axs[2].axis('off')
37
38 # Exibe a imagem com gamma = 1.5
39 axs[3].imshow(img3, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
40 axs[3].set_title('γ = 1.5')
41 axs[3].axis('off')
42
43 # Exibe a imagem com gamma = 2.0
44 axs[4].imshow(img4, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
45 axs[4].set_title('γ = 2.0')
46 axs[4].axis('off')
47
48 # Ajusta o layout para evitar sobreposição
49 plt.tight_layout()
50 plt.show()

```

2.2 Explicação do ajuste

O código desenvolvido segue a seguinte fórmula para realizar a correção gamma:

$$B = A^{1/\gamma}$$

Onde:

- A é o valor de intensidade do pixel normalizado (entre 0 e 1)
- γ é o parâmetro de correção

- B é o valor de intensidade corrigido

Como a imagem está normalizada, temos que:

- Se $\gamma < 1$ então $1/\gamma > 1$ e isso faz com que os valores de cada elemento da imagem diminua o seu valor. Dessa forma, a imagem ficará mais escura (perto de zero).
- Se $\gamma > 1$ então $1/\gamma < 1$ e isso faz com que cada elemento aumente o seu valor e, portanto, a imagem fique mais clara (perto de 1).
- Se $\gamma = 1$ então a imagem permanecerá igual a original.

2.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-2.py](#)

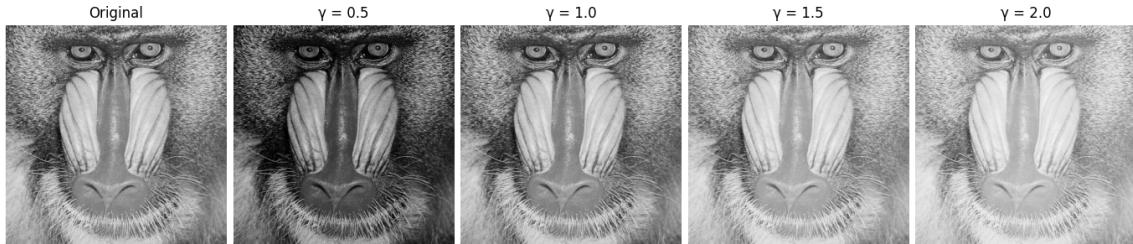


Figure 3: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

3 Questão 3 - Mosaico

3.1 Código em Python

```

1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem em escala de cinza e converte para um array NumPy
6 img = np.array(Image.open('baboon_monocromatica.png').convert('L'))
7
8 # Obtém a altura e largura da imagem
9 altura, largura = img.shape
10
11 # Calcula a altura e largura de cada bloco (dividindo a imagem em 4x4 blocos)
12 altura_bloco = altura // 4
13 largura_bloco = largura // 4
14
15 # Divide a imagem em blocos 4x4 e reorganiza as dimensões para facilitar o acesso
16 blocos = img.reshape(4, altura_bloco, 4, largura_bloco).transpose(0, 2, 1, 3)
17
18 # Define a nova ordem dos blocos (matriz 4x4 com os índices dos blocos)
19 nova_ordem = np.array([
20     [6, 11, 13, 3],
21     [8, 16, 1, 9],
22     [12, 14, 2, 7],
23     [4, 15, 10, 5]
24 ])
25 # Ajusta os índices para começar de 0 (subtraindo 1)
26 nova_ordem = nova_ordem - 1
27
28 # Reorganiza os blocos de acordo com a nova ordem e reconstrói a imagem
29 mosaico = blocos[nova_ordem//4, nova_ordem%4].transpose(0, 2, 1, 3).reshape(altura,
30     largura)
31
32 # Plota a imagem original e a imagem em mosaico lado a lado
33 fig, axs = plt.subplots(1, 2, figsize=(10, 5))
34
35 # Exibe a imagem original
36 axs[0].imshow(img, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
37 axs[0].set_title('Original')
38 axs[0].axis('off')
39
40 # Exibe a imagem em mosaico
41 axs[1].imshow(mosaico, cmap=plt.get_cmap('gray'), vmin=0, vmax=255)
42 axs[1].set_title('Mosaico')
43 axs[1].axis('off')
44
45 # Ajusta o layout e exibe o gráfico
46 plt.tight_layout()
47 plt.show()

```

3.2 Explicação do código

Para entender o código é crucial entender como funciona o reshape e o transpose aplicado. Para isso, vamos começar com o reshape:

3.2.1 Entendendo o reshape

```
img.reshape(4, altura_bloco, 4, largura_bloco)
```

Inicialmente, temos uma imagem img com as dimensões (altura, largura), mas agora precisamos separar nos blocos necessários. Para isso, precisamos dividir as linhas e as

colunas em "conjuntos" de 4 elementos, ou seja, algo do tipo (4, altura//4, 4, largura//4). Que, basicamente, é o reshape que foi aplicado. Note que dentro de cada bloco os pixels continuam organizados, porque o reshape mantém a ordem original.

3.2.2 Entendendo o transpose

Após isso, é realizado a seguinte operação:

```
img.reshape(4, altura_bloco, 4, largura_bloco).transpose(0, 2, 1, 3)
```

Com a imagem img já seccionada em blocos, são modificadas as dimensões de maneira que os blocos 4x4 fiquem nas primeiras duas dimensões e torne mais fácil a manipulação da imagem em blocos. Dessa maneira, é possível acessar os blocos como blocos[i, j].

3.2.3 Entendendo a formação do mosaico

```
mosaico = blocos[nova_ordem//4, nova_ordem%4].transpose(0, 2, 1, 3).reshape(altura, largura)
```

O mosaico é formado utilizando a matriz nova_ordem, que define a nova disposição dos blocos da imagem. Para isso, acessamos a matriz blocos e reorganizamos seus elementos conforme os índices especificados em nova_ordem. Em seguida, aplicamos transpose e reshape para remontar a imagem, garantindo que ela retorne ao formato original, mas agora com os blocos rearranjados na nova configuração.

3.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-3.py](#)

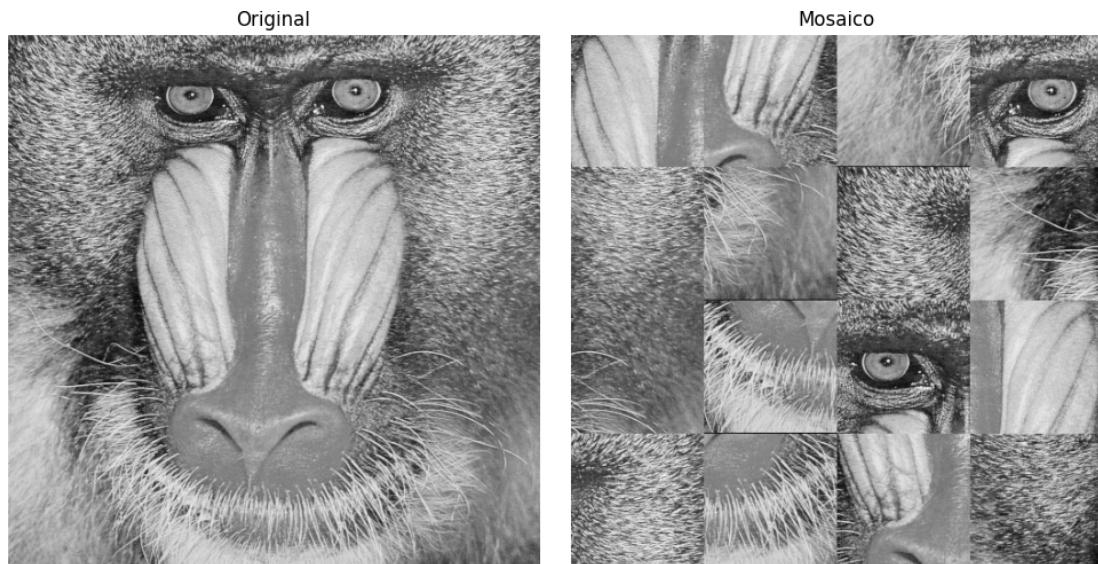


Figure 4: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

4 Questão 4 - Alteração de Cores

4.1 Código em Python

```
1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem 'watch.png', converte para RGB e a transforma em um array NumPy
6 img = np.array(Image.open('watch.png').convert('RGB'))
7
8 # Define a matriz de transformação para o efeito sépia
9 sepia_matrix = np.array([
10     [0.393, 0.769, 0.189],
11     [0.349, 0.686, 0.168],
12     [0.272, 0.534, 0.131]
13 ])
14
15 # Aplica a matriz de transformação sépia à imagem original
16 sepia_img = np.dot(img, sepia_matrix.T)
17
18 # Normaliza os valores da imagem sépia para o intervalo de 0 a 255
19 sepia_img = np.interp(sepia_img, (sepia_img.min(), sepia_img.max()), (0, 255))
20 sepia_img = sepia_img.astype(np.uint8) # Converte os valores para inteiros de 8 bits
21
22 # Exibe a imagem original e a imagem com efeito sépia lado a lado
23 plt.figure(figsize=(10, 5))
24
25 # Subplot para a imagem original
26 plt.subplot(1, 2, 1)
27 plt.imshow(img, vmin=0, vmax=255) # Exibe a imagem original
28 plt.title('Original') # Título do subplot
29 plt.axis('off') # Remove os eixos
30
31 # Subplot para a imagem com efeito sépia
32 plt.subplot(1, 2, 2)
33 plt.imshow(sepia_img, vmax=255, vmin=0) # Exibe a imagem sépia
34 plt.title('Efeito Sépiado') # Título do subplot
35 plt.axis('off') # Remove os eixos
36
37 # Ajusta o layout para evitar sobreposição
38 plt.tight_layout()
39
40 # Mostra as imagens na tela
41 plt.show()
```

4.2 Explicação do código

4.2.1 Operação dot da biblioteca Numpy

Operação dot da biblioteca Numpy

Documentação: `numpy.dot(a, b, out = None)`

- Dot product of two arrays.
 - a: array_like (First Argument)
 - b: array_like (Second Argument)
 - output: ndarray (Returns the dot product of a and b. If a and b are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If out is given, then it is returned.)

Dessa forma, ao realizar a operação dot entre a imagem img e a transposta da matriz sépia estariamos realizando uma combinação linear dos canais RGB originais e gerando uma imagem dessa combinação. É interessante notar que o tom sépia é conhecida por possuir tons quentes mais empoderados e tons frios reduzidos deixando a imagem com um tom amarronzado e aparentemente envelhecido. Isso é claramente observado na matriz sépia transposta, visto que:

$$\begin{bmatrix} R_{\text{sepia}} \\ G_{\text{sepia}} \\ B_{\text{sepia}} \end{bmatrix} = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R_{\text{sepia}} \\ G_{\text{sepia}} \\ B_{\text{sepia}} \end{bmatrix} = \begin{bmatrix} 0.393R + 0.769G + 0.189B \\ 0.349R + 0.686G + 0.168B \\ 0.272R + 0.534G + 0.131B \end{bmatrix}$$

onde:

- R, G, B são os valores do pixel original (0 a 255),
- $R_{\text{sepia}}, G_{\text{sepia}}, B_{\text{sepia}}$ são os valores do pixel após a transformação.

Note que os maiores pesos estão nos canais R e G, que contribuem para tons alaranjados e marrons. O canal B tem os menores pesos, reduzindo sua influência e evitando tons azulados.

- Objetos claros: Adquirem um tom amarelado/marrom.
- Áreas escuras: Tornam-se marrom-escuro em vez de preto puro.
- Cores vivas (como azul ou verde): São suavizadas e convertidas para tons terrosos.

4.2.2 Por que o peso do campo G é maior?

O verde é o canal mais brilhante no modelo RGB (o olho humano é mais sensível ao verde). Se o verde não tivesse um peso alto, a imagem ficaria escura e com tons vermelhos saturados (não naturais). Dar mais peso ao G ajuda a manter o brilho da

imagem enquanto adiciona um tom marrom. Se usássemos $R > G$, a imagem ficaria muito avermelhada (como um filtro vermelho, não sépia). O azul (B) tem os menores pesos em todas as equações porque:

- O sépia é um tom quente, e o azul é uma cor fria.
- Incluir muito azul neutralizaria o efeito marrom e deixaria a imagem com um tom acinzentado.
- O olho humano é menos sensível ao azul, então sua redução não torna a imagem escura demais.

Portanto, a lógica por trás dos pesos é a seguinte:

Table 1: Pesos dos canais de cor na transformação sépia para R_{sepia}

Canal	Peso	Efeito
R	0.393	Adiciona avermelhado, mas não domina.
G	0.769	Mantém brilho e contribui para o marrom.
B	0.189	Reduz influência do azul para evitar neutralização.

4.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-4.py](#)



Figure 5: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

5 Questão 5 - Transformação de Imagens Coloridas

5.1 Código em Python

```
1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem 'watch.png', converte para RGB e a transforma em um array NumPy
6 img = np.array(Image.open('watch.png').convert('RGB'))
7
8 # Matriz de transformação para aplicar o efeito sépia
9 sepia_matrix = np.array([
10     [0.393, 0.769, 0.189],
11     [0.349, 0.686, 0.168],
12     [0.272, 0.534, 0.131]
13 ])
14
15 # Aplica a matriz de transformação sépia à imagem original
16 sepia_img = np.dot(img, sepia_matrix.T)
17
18 # Normaliza os valores da imagem sépia para o intervalo [0, 255]
19 sepia_img = np.interp(sepia_img, (sepia_img.min(), sepia_img.max()), (0, 255))
20 sepia_img = sepia_img.astype(np.uint8) # Converte os valores para inteiros de 8 bits
21
22 # Configura o tamanho da figura para exibição
23 plt.figure(figsize=(10, 5))
24
25 # Exibe a imagem original
26 plt.subplot(1, 3, 1)
27 plt.imshow(img, vmin=0, vmax=255)
28 plt.title('Original') # Título da imagem original
29 plt.axis('off') # Remove os eixos
30
31 # Exibe a imagem com o efeito sépia aplicado
32 plt.subplot(1, 3, 2)
33 plt.imshow(sepia_img, vmax=255, vmin=0)
34 plt.title('Efeito aplicado') # Título da imagem com efeito sépia
35 plt.axis('off') # Remove os eixos
36
37 # Converte a imagem original para escala de cinza
38 gray = np.dot(img, [0.299, 0.587, 0.114]) # Fórmula para conversão em escala de cinza
39 gray = np.interp(gray, (gray.min(), gray.max()), (0, 255)) # Normaliza os valores
40     usando interp
41 gray = gray.astype(np.uint8) # Converte os valores para inteiros de 8 bits
42
43 # Exibe a imagem em escala de cinza
44 plt.subplot(1, 3, 3)
45 plt.imshow(gray, cmap=plt.get_cmap('gray')) # Define o mapa de cores como escala de
46     cinza
47 plt.title('Escala de cinza') # Título da imagem em escala de cinza
48 plt.axis('off') # Remove os eixos
49
50 # Ajusta o layout para evitar sobreposição
51 plt.tight_layout()
52
53 # Mostra as imagens na tela
54 plt.show()
```

5.2 Explicação do código

5.2.1 Transformação do efeito

O efeito é exatamente o mesmo do aplicado na imagem sépia do exercício anterior. Dessa forma, os passos aplicados no código são os mesmos e o resultado obtido também é idêntico.

5.2.2 Transformação em tons de cinza

A transformação de uma imagem RGB para uma imagem em tons de cinza foi aplicada da mesma forma que no exercício 1 no seguinte trecho:

```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])
```

Entretanto, no código do exercício 5 utilizei diretamente o produto para simplificar.

5.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-5.py](#)



Figure 6: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

6 Questão 6 - Plano de Bits

6.1 Código em Python

```

1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem em escala de cinza e converte para um array NumPy de 8 bits
6 img = np.array(Image.open('baboon_monocromatica.png').convert('L')).astype(np.uint8)
7
8 # Define máscaras binárias para cada bit da imagem
9 bin0 = 0b00000001 # Máscara para o bit menos significativo (LSB)
10 bin1 = 0b00000010
11 bin2 = 0b00000100
12 bin3 = 0b00001000
13 bin4 = 0b00010000
14 bin5 = 0b00100000
15 bin6 = 0b01000000
16 bin7 = 0b10000000 # Máscara para o bit mais significativo (MSB)
17
18 # Função para normalizar os valores para o intervalo [0, 255]
19 def normalize(img):
20     return np.interp(img, (img.min(), img.max()), (0, 255))
21
22 # Extrai e normaliza o bit 0 da imagem
23 img0 = normalize(np.bitwise_and(img, bin0))
24
25 # Extrai e normaliza o bit 1 da imagem
26 img1 = normalize(np.bitwise_and(img, bin1))
27
28 # Extrai e normaliza o bit 2 da imagem
29 img2 = normalize(np.bitwise_and(img, bin2))
30
31 # Extrai e normaliza o bit 3 da imagem
32 img3 = normalize(np.bitwise_and(img, bin3))
33
34 # Extrai e normaliza o bit 4 da imagem
35 img4 = normalize(np.bitwise_and(img, bin4))
36
37 # Extrai e normaliza o bit 5 da imagem
38 img5 = normalize(np.bitwise_and(img, bin5))
39
40 # Extrai e normaliza o bit 6 da imagem
41 img6 = normalize(np.bitwise_and(img, bin6))
42
43 # Extrai e normaliza o bit 7 da imagem
44 img7 = normalize(np.bitwise_and(img, bin7))
45
46 # Cria uma grade de subplots para exibir as imagens
47 fig, axs = plt.subplots(3, 3, figsize=(10, 10))
48
49 # Exibe a imagem correspondente ao bit 0
50 axs[0, 0].imshow(img0, cmap='gray', vmin=0, vmax=255)
51 axs[0, 0].set_title('0 bit') # Título do subplot
52 axs[0, 0].axis('off') # Remove os eixos
53
54 # Exibe a imagem correspondente ao bit 1
55 axs[0, 1].imshow(img1, cmap='gray', vmin=0, vmax=255)
56 axs[0, 1].set_title('1 bits')
57 axs[0, 1].axis('off')
58
59 # Exibe a imagem correspondente ao bit 2
60 axs[0, 2].imshow(img2, cmap='gray', vmin=0, vmax=255)
61 axs[0, 2].set_title('2 bits')
62 axs[0, 2].axis('off')
63
64 # Exibe a imagem correspondente ao bit 3
65 axs[1, 0].imshow(img3, cmap='gray', vmin=0, vmax=255)
66 axs[1, 0].set_title('3 bits')
67 axs[1, 0].axis('off')

```

```

68 # Exibe a imagem correspondente ao bit 4
69 axs[1, 1].imshow(img4, cmap='gray', vmin=0, vmax=255)
70 axs[1, 1].set_title('4 bits')
71 axs[1, 1].axis('off')
72
73 # Exibe a imagem correspondente ao bit 5
74 axs[1, 2].imshow(img5, cmap='gray', vmin=0, vmax=255)
75 axs[1, 2].set_title('5 bits')
76 axs[1, 2].axis('off')
77
78 # Exibe a imagem correspondente ao bit 6
79 axs[2, 0].imshow(img6, cmap='gray', vmin=0, vmax=255)
80 axs[2, 0].set_title('6 bits')
81 axs[2, 0].axis('off')
82
83 # Exibe a imagem correspondente ao bit 7
84 axs[2, 1].imshow(img7, cmap='gray', vmin=0, vmax=255)
85 axs[2, 1].set_title('7 bits')
86 axs[2, 1].axis('off')
87
88 # Exibe a imagem original
89 axs[2, 2].imshow(img, cmap='gray', vmin=0, vmax=255)
90 axs[2, 2].set_title('Original')
91 axs[2, 2].axis('off')
92
93 # Ajusta o layout para evitar sobreposição
94 plt.tight_layout()
95
96 # Mostra o gráfico com todas as imagens
97 plt.show()

```

6.2 Explicação do código

6.2.1 Operação bitwise_and da biblioteca Numpy

Operação bitwise_and da biblioteca Numpy

Documentação: `numpy.bitwise_and(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature]) = ufunc 'bitwise_and'`

- Compute the bit-wise AND of two arrays element-wise.
- x1, x2: array_like (Only integer and boolean types are handled. If x1.shape != x2.shape, they must be broadcastable to a common shape (which becomes the shape of the output)

Portanto, utilizei a função para fazer uma operação de AND entre as máscaras binárias e o valor de cada pixel de maneira que sobre apenas o binário do plano respectivo. Dessa forma:

$$a_{m-1} \cdot 2^{m-1} + a_{m-2} \cdot 2^{m-2} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0 \text{ AND } Máscara_k$$

Onde, $Máscara_k = 2^k$ para $k \in \{0, 1, 2, 3, 4, 5, 6, 7\}$. Dessa forma, ao realizar a operação teremos algo do tipo:

$$\text{Resultado} = a_k \cdot 2^k$$

Observe que retiramos dessa operação apenas o canal de interesse e isso resulta o resultado esperado bastando mostrar a imagem após a operação. Foi realizada uma operação de

normalização tradicional. Essa transformação torna a imagem em preto e branco, visto que os pixels só possuirão dois tipos de valor 0 e 2^k então a normalização manterá 0 como zero e tornará o pixel não nulo em preto (deixando a imagem mais visível e perceptível).

6.2.2 Mas e o quê é obtido no final?

O código retorna uma visualização gráfica composta por nove imagens em tons de cinza. As oito primeiras representam os planos de bits da imagem original, do bit menos significativo (bit 0) até o bit mais significativo (bit 7). A nona imagem mostra a imagem original em escala de cinza.

Cada uma das oito primeiras imagens mostra visualmente quais pixels possuem o respectivo bit ativado (valor 1) ou desativado (valor 0). Por exemplo, a imagem do bit 0 destaca os pixels onde o bit menos significativo está ativado. Como esse bit representa a menor contribuição possível para o valor total de intensidade de um pixel ($2^0 = 1$), sua influência visual é muito pequena e tende a formar um padrão semelhante a "ruído" aleatório. Por outro lado, a imagem do bit 7, o mais significativo ($2^7 = 128$), destaca os pixels cuja intensidade total é muito alta, já que esse bit contribui com mais da metade do valor máximo possível (255). Dessa forma, conforme o número do bit aumenta, sua influência na composição da imagem final também aumenta.

O resultado final é uma decomposição da imagem original em seus componentes binários. Essa decomposição permite entender como cada bit contribui para a formação dos tons de cinza.

6.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-6.py](#)

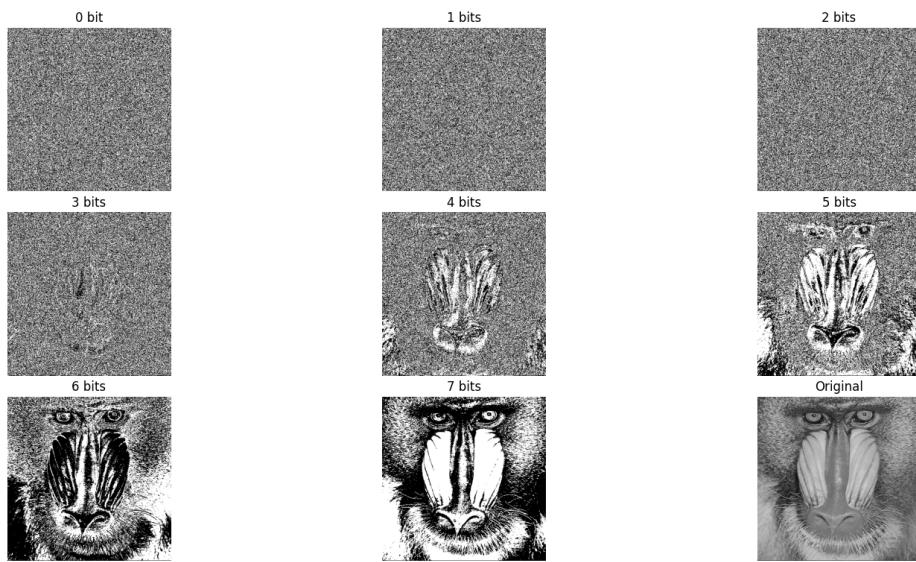


Figure 7: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

7 Questão 7 - Combinação de Imagens

7.1 Código em Python

```

1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem 'baboon_monocromatica.png', converte para escala de cinza e
6 # transforma em um array NumPy
7 img1 = np.array(Image.open('baboon_monocromatica.png').convert('L'))
8
9 # Carrega a imagem 'butterfly.png', converte para escala de cinza e transforma em um
10 # array NumPy
11 img2 = np.array(Image.open('butterfly.png').convert('L'))
12
13 # Cria uma lista de pesos igualmente espaçados entre 0.1 e 0.9 (9 valores)
14 weights = np.linspace(0.1, 0.9, 9)
15
16 # Combina as duas imagens usando os pesos
17 # Para cada peso, calcula uma combinação linear das imagens (img1 * (1 - peso) + img2 * peso)
18 # Adiciona uma nova dimensão para os pesos e realiza a operação em broadcast
19 img_combinadas = np.add(np.multiply(img1[:, :, None], (1 - weights)), np.multiply(img2[:, :, None], weights))
20
21 # Move o eixo dos pesos para o início, criando um array onde cada índice representa uma
22 # imagem combinada
23 img_combinadas = np.moveaxis(img_combinadas, -1, 0)
24
25 # Cria uma grade de subplots (3x3) para exibir as imagens combinadas
26 fig, axs = plt.subplots(3, 3, figsize=(10, 10))
27
28 # Itera sobre as imagens combinadas e exibe cada uma em um subplot
29 for i, img in enumerate(img_combinadas):
30     axs[i // 3, i % 3].imshow(img, cmap='gray', vmin=0, vmax=255) # Exibe a imagem em
31     # escala de cinza
32     axs[i // 3, i % 3].set_title(f'Baboon: {1 - weights[i]:.1f}, Butterfly: {weights[i]:.1f}') # Define o título com os pesos
33     axs[i // 3, i % 3].axis('off') # Remove os eixos para uma visualização limpa
34
35 # Ajusta o layout para evitar sobreposição
36 plt.tight_layout()
37
38 # Exibe o gráfico com as imagens combinadas
39 plt.show()

```

7.2 Explicação do código

7.2.1 Como foi feita a combinação

A combinação das imagens foi realizada com base na seguinte equação:

$$Resultado = img1 \cdot (1 - peso) + img2 \cdot peso$$

A técnica de média ponderada consiste em combinar duas imagens atribuindo pesos diferentes a cada uma delas. No código apresentado, o parâmetro chamado peso varia de 0.1 a 0.9 em passos de 0.1, permitindo visualizar diferentes combinações.

O objetivo da média ponderada é controlar a influência visual de cada imagem no resultado final. Dessa forma:

- Se o peso atribuído à imagem `img1` (Baboon) for maior, ela será visualmente mais dominante na imagem combinada.

- Se o peso da imagem `img2` (Butterfly) for maior, a imagem da borboleta se sobresairá.

Considerando que as imagens estão em escala de cinza, cada pixel possui um valor no intervalo $[0, 255]$. Como a média ponderada mantém a soma dos pesos igual a 1, os valores dos pixels resultantes também estarão contidos nesse mesmo intervalo. Isso garante que a imagem final seja válida e possa ser exibida corretamente sem necessidade de normalização adicional.

Essa abordagem permite visualizar como diferentes ponderações afetam a fusão entre duas imagens.

7.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-7.py](#)

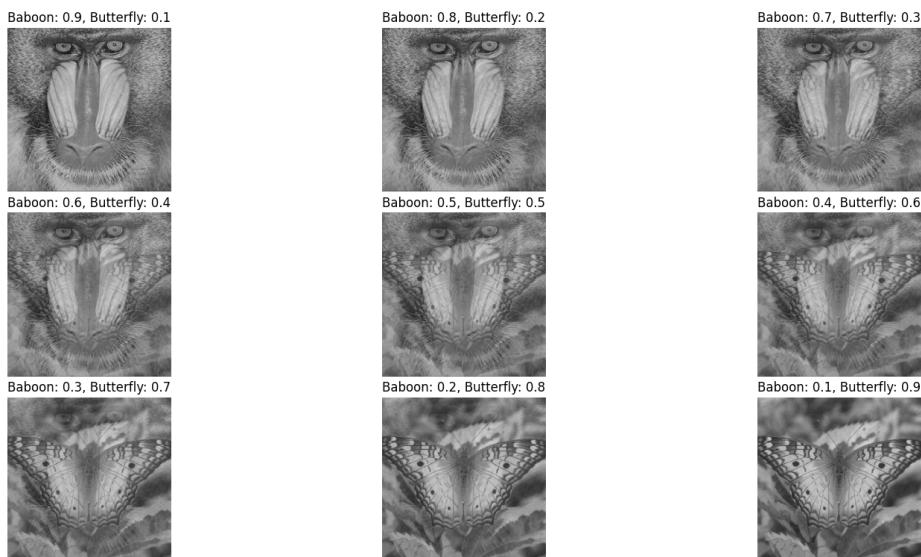


Figure 8: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

8 Questão 8 - Transformação de Intensidade

subsection Código em Python

```

1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem 'city.png', converte para escala de cinza e transforma em um array
6 # NumPy
7 img = np.array(Image.open('city.png').convert('L'))
8
9 # Cria uma figura com uma grade de subplots (2 linhas e 3 colunas)
10 fig, axs = plt.subplots(2, 3, figsize=(10, 6))
11
12 # Exibe a imagem original no primeiro subplot
13 axs[0, 0].imshow(img, cmap='gray', vmin=0, vmax=255)
14 axs[0, 0].set_title('Original') # Define o título do subplot
15 axs[0, 0].axis('off') # Remove os eixos
16
17 # Gera a imagem negativa (inversão de cores) e exibe no segundo subplot
18 img_negativo = np.subtract(255, img)
19 axs[0, 1].imshow(img_negativo, cmap='gray', vmin=0, vmax=255)
20 axs[0, 1].set_title('Negativo')
21 axs[0, 1].axis('off')
22
23 # Realiza uma transformação linear nos valores de pixel no intervalo [100, 200]
24 imagem_ajustada = np.interp(np.clip(img, 100, 200), (100, 200), (0, 255))
25 axs[0, 2].imshow(imagem_ajustada, cmap='gray', vmin=0, vmax=255)
26 axs[0, 2].set_title('Transformada [100, 200]')
27 axs[0, 2].axis('off')
28
29 # Inverte horizontalmente as linhas pares da imagem e exibe no quarto subplot
30 img_invertida = img.copy()
31 img_invertida[::-2, :] = img_invertida[::2, ::-1]
32 axs[1, 0].imshow(img_invertida, cmap='gray')
33 axs[1, 0].set_title('Invertida')
34 axs[1, 0].axis('off')
35
36 # Espelha a metade inferior da imagem em relação à linha central horizontal
37 img_espelhada = img.copy()
38 img_espelhada[img.shape[0]//2:, :] = img_espelhada[img.shape[0]//2 - 1 :: -1, :]
39 axs[1, 1].imshow(img_espelhada, cmap='gray', vmin=0, vmax=255)
40 axs[1, 1].set_title('Espelhada')
41 axs[1, 1].axis('off')
42
43 # Espelha a imagem verticalmente (de cabeça para baixo) e exibe no último subplot
44 img_espelhada_vertical = img.copy()
45 img_espelhada_vertical = img_espelhada_vertical[::-1, :]
46 axs[1, 2].imshow(img_espelhada_vertical, cmap='gray', vmin=0, vmax=255)
47 axs[1, 2].set_title('Espelhada Vertical')
48 axs[1, 2].axis('off')
49
50 # Ajusta o layout para evitar sobreposição entre os subplots
51 plt.tight_layout()
52
53 # Exibe a figura com os subplots
54 plt.show()

```

8.1 Explicação do código

8.1.1 Imagem Negativa

A imagem negativa é definida da seguinte forma: Seja $I(x, y)$ o valor da intensidade de um pixel na posição (x, Y) , com dentro do intervalo $[0, 255]$ podemos definir a imagem negativa $I_{neg}(x, y)$ assim:

$$I_{neg} = 255 - I(x, y)$$

Para obter esse resultado utilizei a função subtract da biblioteca Numpy que realiza a subtração de dois np.array. Note que se os arrays tiverem tamanhos diferentes eles precisam ser possíveis de realizar broadcast (que é o caso, pois um dos argumentos é um inteiro).

8.1.2 Transformação de Intensidade para o Intervalo [100, 200]

Para obter o resultado desejado utilizamos a seguinte operação com funções da biblioteca Numpy:

```
imagem_ajustada = np.interp(np.clip(img, 100, 200), (100, 200), (0, 255))
```

- **np.clip(img, 100, 200)**: Limita os valores da imagem original ao intervalo [100, 200]. Todos os pixels com intensidade menor que 100 são convertidos para 100, e os maiores que 200 são convertidos para 200. Isso garante que a transformação ocorra apenas dentro da faixa desejada, sem influência de valores extremos.
- **np.interp(..., (100, 200), (0, 255))**: Realiza a interpolação linear entre os valores ajustados, mapeando a faixa [100, 200] para o intervalo [0, 255]. Assim:
 - O valor 100 será mapeado para 0 (preto),
 - O valor 200 será mapeado para 255 (branco),
 - Os valores intermediários serão proporcionalmente distribuídos.

Dessa forma, temos que todos os pixels que são mais escuros que 100 se tornarão completamente pretos no resultado final, e todos os pixels que são mais claros que 200 se tornarão completamente brancos. Isso é interessante, pois a imagem final destacará os tons intermediários (aqueles originalmente entre 100 e 200) e "eliminará" visualmente os tons mais claros e mais escuros.

Como consequência, o contraste da imagem transformada será maior dentro da faixa de interesse, e a variação de intensidades fora dessa faixa será minimizada. Para imagens cujo histograma apresenta maior concentração nos extremos (tons escuros e claros), essa técnica tende a reduzir a frequência de transições abruptas entre regiões, promovendo uma aparência mais suave e realçando os detalhes nos tons médios.

Essa abordagem é útil quando se deseja enfatizar regiões específicas da imagem e ignorar ruídos ou variações que estejam fora da faixa de interesse definida.

8.1.3 Inversão das Linhas Pares da Imagem

O objetivo é inverter horizontalmente apenas as linhas pares da imagem original. Para isso, foi utilizado o seguinte trecho de código:

```
img_invertida = img.copy()
img_invertida[::-2, :] = img_invertida[::-2, ::-1]
```

- **img.copy()**: Cria uma cópia da imagem original para que a operação não afete a imagem de origem.
- **img_invertida[::-2, :]**: Seleciona todas as **linhas pares** da imagem, ou seja, linhas com índice 0, 2, 4, etc.

- **img.invertida[::-2, ::-1]**: Percorre as linhas pares de trás para frente de maneira que elas se tornem invertidas na variável img.invertida.

Note que como criamos uma cópia da imagem, já temos as linhas ímpares definidas corretamente e basta que alteremos as linhas pares. Ao inverter apenas as linhas pares, o padrão resultante na imagem final cria uma sensação de "serrilhado" ou de distorção visual intercalada. Esse tipo de transformação é simples de aplicar utilizando indexação avançada do NumPy, o que torna o código eficiente e de fácil entendimento.

8.1.4 Espelhamento da Metade Superior da Imagem na Parte Inferior

Nesta transformação, temos como objetivo espelhar verticalmente a metade superior da imagem na metade inferior, ou seja, as linhas da parte de cima da imagem são copiadas para a parte de baixo, criando um efeito de simetria em relação ao eixo horizontal central. Para isso, realizamos as seguintes operações:

```
img_espelhada = img.copy()
img_espelhada[img.shape[0]//2:, :] = img_espelhada[img.shape[0]//2 - 1::-1, :]
```

- **img.copy()**: Cria uma cópia da imagem original para preservar a imagem de entrada.
- **img.shape[0]//2**: Calcula a linha central da imagem (metade da altura) para termos o valor que precisamos para construir a imagem espelhada (não é preciso alterar a metade superior).
- **img_espelhada[img.shape[0]//2:, :]**: Seleciona todas as linhas da metade inferior da imagem (do meio até o final).
- **img_espelhada[img.shape[0]//2 - 1::-1, :]**: Acessa as linhas da metade superior em ordem invertida (de baixo para cima) e as atribui na metade inferior, realizando o espelhamento vertical da parte superior.

Dessa forma, enquanto na direita estaremos "subindo" na imagem original na esquerda estaremos "descendo" de maneira que gere o espelhamento da imagem. É criada uma simetria visual na imagem interessante. É uma técnica interessante para criar padrões ou simular reflexos verticais.

8.1.5 Espelhamento Vertical da Imagem

Diferente da operação anterior, é realizado um espelhamento completo da imagem em relação ao eixo horizontal. Assim todas as linhas da imagem serão invertidas verticalmente, como se a imagem estivesse sendo observada em um espelho colocado abaixo dela ou vista de cabeça para baixo. Segue a implementação:

```
img_espelhada_vertical = img.copy()
img_espelhada_vertical = img_espelhada_vertical[::-1, :]
```

- **img.copy()**: Garante que a imagem original não seja modificada.

- `img[::-1, :]`: Inverte a ordem de todas as linhas da imagem, ou seja, a primeira linha troca de lugar com a última, a segunda com a penúltima, e assim por diante.

A imagem final obtida, portanto, terá a inversão no eixo horizontal. A operação é simples e eficiente graças ao uso da indexação avançada do NumPy.

8.2 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-8.py](#)

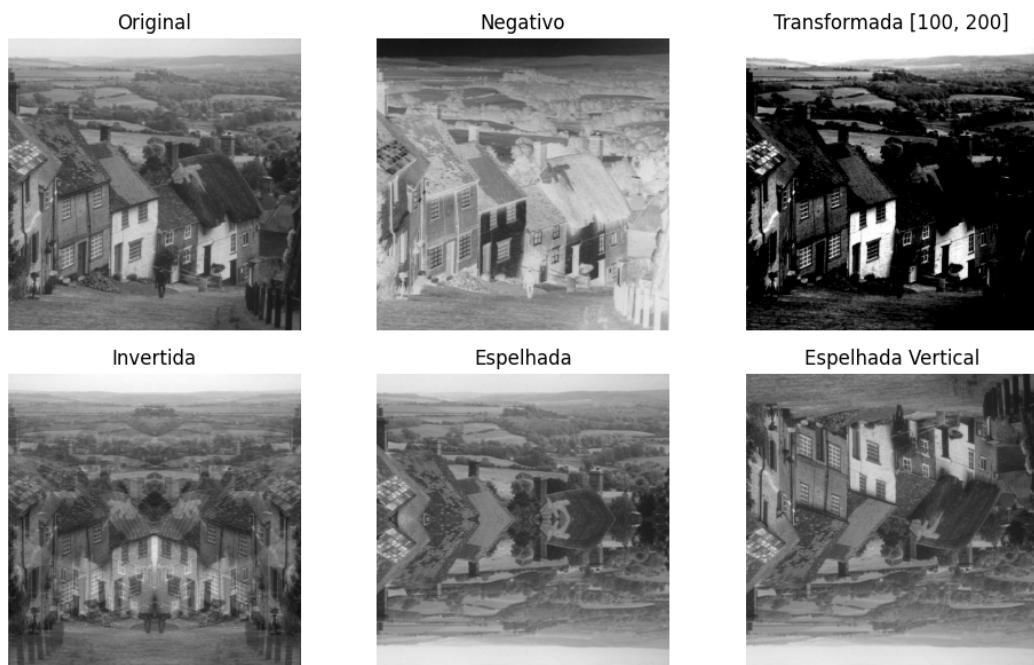


Figure 9: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

9 Questão 9 - Quantização de Imagens

9.1 Código em Python

```
1 import numpy as np
2 from PIL import Image
3 import matplotlib.pyplot as plt
4
5 # Carrega a imagem 'baboon_monocromatica.png', converte para escala de cinza ('L') e
6 # transforma em um array NumPy
7 img = np.array(Image.open('baboon_monocromatica.png').convert('L'))
8
9 # Define os níveis de quantização desejados
10 niveis = [64, 32, 16, 8, 4, 2]
11
12 # Define as máscaras binárias correspondentes para cada nível de quantização
13 mascaras = [0b11111100, 0b11111000, 0b11110000, 0b11100000, 0b11000000, 0b10000000]
14
15 # Cria uma grade de subplots (3x3) para exibir as imagens
16 fig, axs = plt.subplots(3, 3, figsize=(10, 6))
17
18 # Desativa os eixos dos subplots que não serão usados
19 axs[0,0].axis('off')
20 axs[0,2].axis('off')
21
22 # Exibe a imagem original no subplot central da primeira linha
23 axs[0,1].imshow(img, cmap='gray', vmin=0, vmax=255)
24 axs[0,1].set_title('Original: 256 níveis') # Define o título para a imagem original
25 axs[0,1].axis('off') # Desativa os eixos
26
27 # Loop para aplicar a quantização e exibir as imagens resultantes
28 for i, nivel in enumerate(niveis):
29     # Aplica a máscara binária para reduzir os níveis de cinza
30     img_quantizada = np.bitwise_and(img, mascaras[i])
31
32     # Reescala os valores da imagem quantizada para o intervalo [0, 255]
33     img_quantizada = np.interp(img_quantizada, (img_quantizada.min(), img_quantizada.max()), (0, 255))
34
35     # Exibe a imagem quantizada no subplot correspondente
36     axs[i//3 + 1, i%3].imshow(img_quantizada, cmap='gray', vmin=0, vmax=255)
37     axs[i//3 + 1, i%3].set_title(f'{nivel} níveis') # Define o título com o número de níveis
38     axs[i//3 + 1, i%3].axis('off') # Desativa os eixos
39
40 # Ajusta o layout para evitar sobreposição de elementos
41 plt.tight_layout()
42
43 # Exibe o gráfico com todas as imagens
44 plt.show()
```

9.2 Explicação do código

9.2.1 Quantização de Imagens Monocromáticas

Quantização representa a redução da quantidade de níveis de cinza usados para representar uma imagem em escala de cinza. Isso está diretamente relacionado à profundidade de bits da imagem.

O código a seguir mostra como foi realizada a quantização da imagem `baboon_monocromatica.png` para diferentes níveis de quantização: 64, 32, 16, 8, 4 e 2 níveis. Em cada nível foi desenvolvida uma máscara binária correspondente aplicada à imagem original por meio da operação `bitwise_and`, a fim de preservar apenas os bits mais significativos (MSBs).

```
img_quantizada = np.bitwise_and(img, mascara)
```

A implementação mostrada faz com que apenas os bits mais significativos para a quantização escolhida sejam mantidos, descartando os menos significativos. Tal como, para 2 níveis (1 bit), apenas o bit mais significativo é preservado. Para 4 níveis (2 bits), os dois bits mais significativos são mantidos, e assim por diante. Isso cria versões cada vez mais simplificadas da imagem original.

Após a aplicação da máscara, é feita uma reescala dos valores para o intervalo [0, 255] para garantir uma visualização adequada da imagem em escala de cinza utilizando o imshow:

```
np.interp(img_quantizada, (img_quantizada.min(), img_quantizada.max()), (0, 255))
```

- **np.interp:** A função realiza interpolação linear para ajustar os valores resultantes da quantização para a faixa completa de intensidade de 8 bits ([0, 255]), facilitando a comparação visual entre as diferentes versões quantizadas.

O resultado final deixa claro como a imagem perde detalhes conforme diminuímos a quantidade de níveis de cinza disponíveis. Com 64 níveis, a imagem ainda se mantém bastante semelhante à original. Já com 2 níveis, vemos uma simplificação extrema onde só existem regiões completamente pretas ou completamente brancas, perdendo quase toda a informação de textura e profundidade visual. Interessante notar que, como em 64 níveis a perda de detalhamento visualmente é pequeno, pode ser uma forma útil de reduzir o tamanho de uma imagem e compactar o espaço de memória necessário para armazená-la.

9.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-9.py](#)

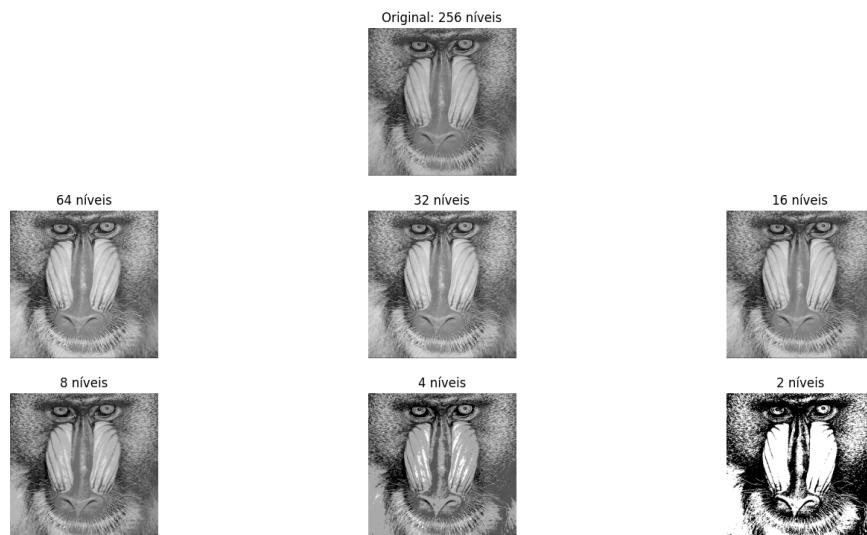


Figure 10: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)

10 Questão 10 - Filtragem das Imagens

10.1 Código em Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.ndimage import convolve
4 from PIL import Image
5
6 # Função para aplicar um filtro (máscara de convolução) em uma imagem monocromática
7 def aplicar_filtro(imagem, filtro):
8     """Aplica um filtro (máscara de convolução) em uma imagem monocromática."""
9     # Aplica a convolução da imagem com o filtro fornecido
10    imagem_filtrada = convolve(imagem, filtro, mode='constant', cval=0.0)
11    return imagem_filtrada
12
13 # Definição de vários filtros (máscaras de convolução) para diferentes propósitos
14 h1 = np.array([[0, 0, -1, 0, 0],
15                 [0, -1, -2, -1, 0],
16                 [-1, -2, 16, -2, -1],
17                 [0, -1, -2, -1, 0],
18                 [0, 0, -1, 0, 0]])
19
20 h2 = np.array([[1/256, 4/256, 6/256, 4/256, 1/256],
21                 [4/256, 16/256, 24/256, 16/256, 4/256],
22                 [6/256, 24/256, 36/256, 24/256, 6/256],
23                 [4/256, 16/256, 24/256, 16/256, 4/256],
24                 [1/256, 4/256, 6/256, 4/256, 1/256]])
25
26 h3 = np.array([[[-1, 0, 1],
27                 [-2, 0, 2],
28                 [-1, 0, 1]]])
29
30 h4 = np.array([[[-1, -2, -1],
31                 [0, 0, 0],
32                 [1, 2, 1]]])
33
34 h5 = np.array([[[-1, -1, -1],
35                 [-1, 8, -1],
36                 [-1, -1, -1]]])
37
38 h6 = np.array([[1/9, 1/9, 1/9],
39                 [1/9, 1/9, 1/9],
40                 [1/9, 1/9, 1/9]])
41
42 h7 = np.array([[[-1, -1, 2],
43                 [-1, 2, -1],
44                 [2, -1, -1]]])
45
46 h8 = np.array([[2, -1, -1],
47                 [-1, 2, -1],
48                 [-1, -1, 2]])
49
50 h9 = np.array([[1/9, 0, 0, 0, 0, 0, 0, 0, 0],
51                 [0, 1/9, 0, 0, 0, 0, 0, 0, 0],
52                 [0, 0, 1/9, 0, 0, 0, 0, 0, 0],
53                 [0, 0, 0, 1/9, 0, 0, 0, 0, 0],
54                 [0, 0, 0, 0, 1/9, 0, 0, 0, 0],
55                 [0, 0, 0, 0, 0, 1/9, 0, 0, 0],
56                 [0, 0, 0, 0, 0, 0, 1/9, 0, 0],
57                 [0, 0, 0, 0, 0, 0, 0, 1/9, 0],
58                 [0, 0, 0, 0, 0, 0, 0, 0, 1/9]])
59
60 h10 = np.array([[1/8, -1/8, -1/8, -1/8, -1/8],
61                 [-1/8, 2/8, 2/8, 2/8, -1/8],
62                 [-1/8, 2/8, 8/8, 2/8, -1/8],
63                 [-1/8, 2/8, 2/8, 2/8, -1/8],
64                 [-1/8, -1/8, -1/8, -1/8, -1/8]])
65
66 h11 = np.array([[[-1, -1, 0],
67                 [-1, 0, 1],
```

```
68             [0, 1, 1])
69
70 # Carrega a imagem em escala de cinza
71 imagem = np.array(Image.open('watch.png').convert('L'))
72
73 # Dicionário para armazenar os resultados de cada filtro
74 resultados = {
75     'Filtro h1': aplicar_filtro(imagem, h1),
76     'Filtro h2': aplicar_filtro(imagem, h2),
77     'Filtro h3': aplicar_filtro(imagem, h3),
78     'Filtro h4': aplicar_filtro(imagem, h4),
79     'Filtro h5': aplicar_filtro(imagem, h5),
80     'Filtro h6': aplicar_filtro(imagem, h6),
81     'Filtro h7': aplicar_filtro(imagem, h7),
82     'Filtro h8': aplicar_filtro(imagem, h8),
83     'Filtro h9': aplicar_filtro(imagem, h9),
84     'Filtro h10': aplicar_filtro(imagem, h10),
85     'Filtro h11': aplicar_filtro(imagem, h11),
86     'Combinação sqrt(h3^2+h4^2)': np.zeros_like(imagem) # Inicializa com zeros
87 }
88
89 # Calcula a combinação dos filtros h3 e h4
90 h3_result = aplicar_filtro(imagem, h3)
91 h4_result = aplicar_filtro(imagem, h4)
92 combinacao = np.sqrt(h3_result**2 + h4_result**2) # Combinação pela raiz quadrada da
93 combinacao = np.interp(combinacao, (combinacao.min(), combinacao.max()), (0, 255)) # 
94 combinacao = combinacao / 255
95 resultados['Combinação sqrt(h3^2+h4^2)'] = combinacao
96
97 # Cria uma figura com subplots para exibir os resultados
98 fig, axs = plt.subplots(3, 4, figsize=(12, 12))
99
100 # Plota cada imagem filtrada
101 for idx, (titulo, img) in enumerate(resultados.items()):
102     row = idx // 4 # Calcula a linha do subplot
103     col = idx % 4 # Calcula a coluna do subplot
104     axs[row, col].imshow(img, cmap='gray', vmin=0, vmax=255) # Exibe a imagem em escala
105     # de cinza
106     axs[row, col].set_title(titulo) # Define o título do subplot
107     axs[row, col].axis('off') # Remove os eixos
108
109 # Ajusta o espaçamento entre os subplots
110 plt.subplots_adjust(wspace=0.5, hspace=0.5)
111 plt.show() # Exibe a figura
```

10.2 Explicação do código

10.2.1 Função convolve da biblioteca `scipy.ndimage`

Função `convolve` da biblioteca `scipy.ndimage`

Documentação: `scipy.ndimage.convolve(input, weights, output=None, mode='reflect', cval=0.0, origin=0, *, axes=None)`

- Realiza uma convolução multidimensional entre o array de entrada e um kernel de pesos.
- `input`: `array_like` — Imagem ou array de entrada a ser processado.
- `weights`: `array_like` — Máscara ou filtro a ser aplicado.
- `mode='constant'`: Define como os valores fora dos limites da imagem serão tratados. O modo '`constant`' preenche com um valor constante.
- `cval=0.0`: Valor constante usado nas bordas quando `mode='constant'`.

10.2.2 Função `aplicar_filtro`

A função tem como objetivo aplicar um filtro sobre a imagem monocromática fornecida na entrada. Para isso, utiliza-se a função `convolve` da biblioteca `scipy.ndimage`, que realiza a operação de convolução multidimensional entre a imagem e a matriz do filtro. O filtro é uma matriz contendo os pesos que serão aplicados sobre os pixels vizinhos, de acordo com a operação de convolução. A chamada `convolve(imagem, filtro, mode='constant', cval=0.0)` aplica esse filtro utilizando um modo específico de tratamento de bordas.

O parâmetro '`mode='reflect'`' indica que, ao aplicar o filtro nas regiões próximas às bordas da imagem, os valores fora dos limites do array original serão obtidos por reflexão dos pixels da própria imagem. Isso significa que, quando o filtro (ou kernel) ultrapassa os limites da imagem ao ser centralizado sobre pixels próximos às bordas, os valores necessários para completar a operação de convolução são espelhados a partir dos valores reais da imagem, como se houvesse uma continuação simétrica do conteúdo. Essa abordagem evita a introdução de valores artificiais (como zeros, no caso do '`mode='constant'`') e, por isso, tende a preservar melhor a continuidade das estruturas visuais da imagem, especialmente nas bordas. Isso resulta em uma filtragem mais suave e natural, reduzindo artefatos indesejados.

Apesar do filtro "deslizar" pela imagem e envolver regiões além das bordas, a função `convolve` mantém as dimensões da imagem resultante iguais às da imagem original. Isso ocorre porque a operação de convolução é centralizada em cada pixel da imagem original, e os valores externos (necessários para o cálculo nas bordas) são artificialmente preenchidos conforme o modo escolhido.

Como resultado, `imagem_filtrada` contém os valores processados da convolução que é o objetivo para aplicar o filtro.

10.2.3 Filtro h_1

O filtro h_1 é uma máscara de convolução bidimensional utilizada para realçar regiões de contraste elevado em uma imagem, funcionando como um realçador de bordas com ênfase no centro da máscara. A matriz do filtro é dada por:

$$h_1 = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

Esse filtro possui um valor central elevado (16), cercado por valores negativos que somam -16, o que mantém o balanço da matriz em zero. Isso caracteriza um filtro do tipo **realce de alta frequência**, pois ele enfatiza variações bruscas de intensidade na imagem (bordas), ao mesmo tempo que minimiza regiões de intensidade constante.

Efeito esperado: ao aplicar h_1 a uma imagem, os pixels cujos vizinhos apresentarem grande variação de intensidade (ou seja, bordas) terão sua intensidade aumentada, enquanto áreas homogêneas serão atenuadas. O resultado final é uma imagem com as bordas acentuadas, destacando contornos e detalhes estruturais.

10.2.4 Filtro h_2

O filtro h_2 é uma máscara de convolução utilizada para suavizar imagens e reduzir ruídos. Ele se baseia na distribuição gaussiana e atribui pesos maiores aos pixels centrais e pesos decrescentes à medida que se afastam do centro. A matriz do filtro é a seguinte:

$$h_2 = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Ela é simétrica e possui o maior peso concentrado no centro, promovendo uma média ponderada que suaviza transições bruscas na imagem. O fator $\frac{1}{256}$ garante que a soma dos elementos da máscara seja igual a 1, preservando a intensidade média da imagem após a filtragem.

Efeito esperado: ao aplicar h_2 em uma imagem, ocorre uma atenuação de detalhes finos e ruídos. Bordas e texturas pequenas tendem a desaparecer, sendo útil como pré-processamento em tarefas como segmentação ou detecção de contornos (quando aplicado antes de um filtro de borda). A tendência é que a frequência da imagem diminua, pois o resultado final é mais suave em transições.

10.2.5 Filtro h_3

O filtro h_3 é conhecido como **filtro de Sobel horizontal** e é utilizado com objetivo de **detecção de bordas** na direção horizontal (ou seja, mudanças de intensidade ao longo do eixo horizontal). Sua matriz é a seguinte:

$$h_3 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Esse filtro calcula uma aproximação da derivada da intensidade em relação ao eixo x . Os valores negativos à esquerda e positivos à direita da matriz contribuem para detectar transições bruscas de intensidade da esquerda para a direita e vice-versa.

Efeito esperado: ao aplicar o filtro h_3 em uma imagem, as regiões onde há variações horizontais de intensidade (como bordas verticais) são destacadas, enquanto regiões com pouca ou nenhuma variação horizontal permanecem escuras. O resultado é uma imagem que evidencia contornos verticais.

10.2.6 Filtro h_4

Muito parecido com o h_3 , o filtro h_4 é conhecido como **filtro de Sobel vertical** e é utilizado para a **detecção de bordas na direção vertical**, ou seja, mudanças de intensidade ao longo do eixo vertical da imagem. Sua matriz é a seguinte:

$$h_4 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Esse filtro realça as variações verticais de intensidade na imagem, respondendo fortemente a bordas horizontais. A estrutura da máscara aplica pesos maiores aos pixels centrais das regiões superior e inferior da vizinhança, o que aumenta a sensibilidade a mudanças verticais.

Resultado esperado: uma imagem que evidencia as bordas horizontais, destacando transições de tons claros para escuros (ou vice-versa) no sentido vertical.

10.2.7 Filtro h_5

O filtro h_5 é um exemplo clássico de **filtro de realce de bordas**. Ele é usado para detectar mudanças bruscas na intensidade dos pixels, ou seja, bordas em todas as direções (horizontal, vertical e diagonal). A matriz referente ao filtro é:

$$h_5 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Esse kernel calcula a segunda derivada da imagem, identificando regiões onde ocorrem transições rápidas de intensidade. O valor central é positivo e maior que a soma dos negativos ao redor, o que faz com que regiões homogêneas resultem em valores próximos de zero e bordas (onde há contraste) sejam destacadas com valores altos.

Resultado esperado: uma imagem que realça as bordas em todas as direções, deixando áreas uniformes praticamente pretas e destacando contornos claros onde há variação brusca de intensidade.

10.2.8 Filtro h_6

O filtro h_6 é conhecido como **filtro de média** é usado para realizar uma **suavização** ou **remoção de ruído** na imagem. Ele atua atenuando variações bruscas de intensidade ao calcular a média dos pixels vizinhos. A matriz do filtro é:

$$h_6 = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Ele possui todos os valores iguais a $\frac{1}{9}$, o que significa que, ao ser aplicado em uma imagem, cada pixel passa a ser a média aritmética dos 9 pixels (ele mesmo e seus 8 vizinhos imediatos).

Resultado esperado: uma imagem suavizada, com redução de ruídos e detalhes finos. Bordas e texturas podem ser levemente borradadas, o que é útil quando se deseja reduzir o efeito de ruídos aleatórios sem perder completamente a estrutura da imagem.

10.2.9 Filtro h_7

O filtro h_7 é um exemplo de **filtro de detecção de padrões diagonais**, e sua estrutura é assimétrica, o que o torna sensível a variações específicas de intensidade em determinadas direções. A matriz do filtro é:

$$h_7 = \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

Esse filtro destaca regiões da imagem onde há transições de intensidade em diagonais específicas, realçando áreas onde há contrastes fortes nessas direções.

Resultado esperado: realce de bordas diagonais com orientação principal da parte inferior esquerda para a superior direita (e vice-versa). Pode ser útil em tarefas onde padrões diagonais são relevantes.

10.2.10 Filtro h_8

O filtro h_8 também é um exemplo de **filtro de detecção de padrões diagonais**, semelhante ao filtro h_7 , mas com uma distribuição diferente dos pesos. Sua matriz é:

$$h_8 = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

Esse filtro possui pesos positivos ao longo da diagonal principal (de canto superior esquerdo a canto inferior direito), e pesos negativos fora dela. Isso faz com que ele reforce estruturas diagonais que seguem essa direção principal.

Resultado esperado: o filtro h_8 realça transições de intensidade que ocorrem ao longo da diagonal principal da imagem, sendo útil para detectar bordas diagonais específicas ou padrões que seguem essa orientação. Ele também pode ajudar a destacar certas texturas ou formas que não seriam tão visíveis com filtros mais simétricos ou voltados para direções horizontais/verticais.

10.2.11 Filtro h_9

O filtro h_9 é uma matriz de convolução 9×9 bastante esparsa, com elementos não nulos apenas na diagonal principal. Sua estrutura é dada por:

$$h_9 = \frac{1}{9} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Esse filtro possui valor $\frac{1}{9}$ apenas nos elementos da diagonal principal e zero em todas as outras posições. Isso significa que ele realiza uma **média ao longo da diagonal principal da imagem**.

Resultado esperado: esse tipo de filtro suaviza apenas os pixels que estão alinhados na diagonal principal da imagem (de canto superior esquerdo para canto inferior direito), ignorando os demais.

10.2.12 Filtro h_{10}

O filtro h_{10} é uma máscara 5×5 projetada para realçar o contraste da imagem, destacando regiões centrais e atenuando as vizinhanças. Sua estrutura é a seguinte:

$$h_{10} = \frac{1}{8} \begin{bmatrix} 1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Esse filtro tem um valor central elevado (1) cercado por valores positivos intermediários ($2/8$) e negativos ($-1/8$), com um único valor positivo ($1/8$) no canto superior esquerdo.

Comportamento: o filtro atua como um realçador de contraste, promovendo a intensificação do pixel central em relação à sua vizinhança. Os valores negativos nas bordas reduzem a contribuição dos pixels ao redor, enquanto os valores positivos próximos ao centro reforçam as diferenças locais. Esse tipo de filtro é útil para **destacar formas e padrões centrais**, funcionando como uma versão suavizada de filtros de nitidez.

Resultado esperado: a imagem resultante tende a enfatizar os detalhes centrais de regiões homogêneas, enquanto as bordas ao redor dessas regiões são suavemente contrastadas, podendo produzir um efeito de realce sem introduzir ruído intenso.

10.2.13 Filtro h_{11}

O filtro h_{11} é uma máscara de convolução 3×3 com a seguinte estrutura:

$$h_{11} = \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Este filtro é utilizado para **deteção de bordas diagonais**, especialmente aquelas orientadas no sentido da diagonal que vai do canto inferior esquerdo ao canto superior

direito. A combinação de valores negativos no canto superior esquerdo e positivos no canto inferior direito cria um padrão que realça as variações de intensidade nessa direção.

Comportamento: ele destaca mudanças diagonais na intensidade dos pixels, funcionando como um detector de bordas em ângulo de 45 graus. O valor central é zero, o que significa que o pixel central não influencia diretamente no resultado, pois o foco está nas diferenças entre vizinhos diagonais opostos.

Resultado esperado: ao aplicar este filtro em uma imagem, as bordas diagonais de objetos inclinados na direção mencionada se tornam mais visíveis, enquanto outras orientações podem ser suavizadas ou ignoradas.

10.2.14 Combinação dos filtros h_3 e h_4

Os filtros h_3 e h_4 são:

$$h_3 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad h_4 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Para detectar bordas em todas as direções, combinamos as respostas de ambos os filtros utilizando a seguinte expressão:

$$\text{Combinação} = \sqrt{(h_3 * I)^2 + (h_4 * I)^2}$$

Onde $*$ representa a operação de convolução e I é a imagem original. Essa fórmula é baseada no cálculo da magnitude do gradiente, considerando as componentes horizontal (G_x) e vertical (G_y):

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

Após calcular essa magnitude, é realizada uma normalização para ajustar os valores obtidos para o intervalo de intensidade da imagem (0 a 255):

$$G_{\text{normalizado}} = \text{interp}(G, [G_{\min}, G_{\max}], [0, 255])$$

Resultado esperado: a imagem resultante apresenta as bordas mais evidentes, independentemente da direção. As áreas de maior variação de intensidade (bordas) aparecem com valores mais altos, enquanto as regiões homogêneas permanecem com valores baixos.

10.3 Resultado obtido no código

O código pode ser acesso através do seguinte link: [questao1-10.py](#)

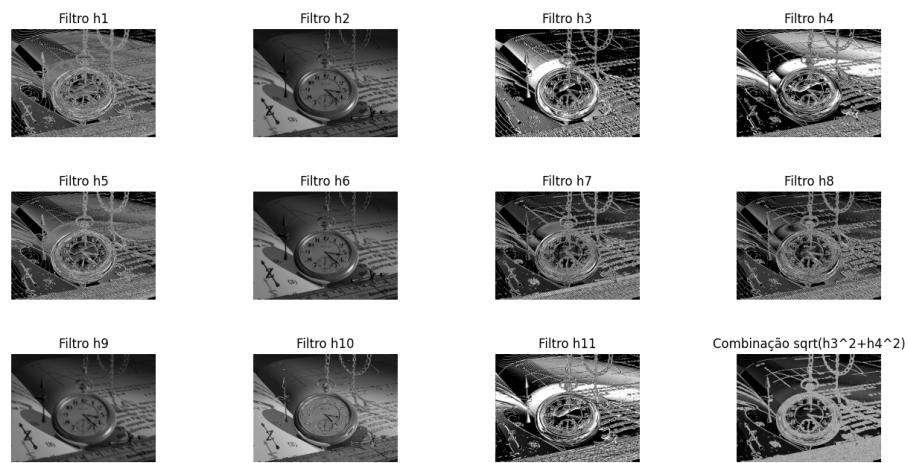


Figure 11: Imagem retornada pela função `plt.show` ao executar o código. Disponível em: [link da imagem](#)