

MO443 - Introdução ao Processamento de Imagem Digital

Relatório 1

Aluno: Julio Vinicius Amaral Oliveira

RA: 230537

Instituto de Computação
Universidade Estadual de Campinas

Campinas, 07 de Abril de 2025.

Sumário

1	Introdução	2
2	Materiais e métodos	2
3	Códigos auxiliares	3
4	Exercícios e discussão	4
4.1	Exercício 1: Esboço a lápis	4
4.2	Exercício 2: Ajuste de brilho	5
4.3	Exercício 3: Mosaico	6
4.4	Exercício 4: Alteração de cores	8
4.5	Exercício 5: Transformação de imagens coloridas . .	9
4.6	Exercício 6: Planos de bits	10
4.7	Exercício 7: Combinação de imagens	11
4.8	Exercício 8: Transformação de intensidade	12
4.9	Exercício 9: Quantização de imagens	14
4.10	Exercício 10: Filtragem de imagens	16

1 Introdução

O seguinte trabalho visa mostrar a aplicação de técnicas iniciais de processamento de imagem digital. Tais como conversão de cores, filtragem utilizando máscaras, manipulação de imagens como espelhamento. Essas técnicas são importantes para formar o alicerce teórico e prático que será utilizado ao longo de todas as técnicas futuras de processamento de imagem. Ao longo do trabalho, mostraremos como os desafios propostos foram resolvidos, além de justificar as escolhas tomadas e problemas enfrentados durante o desenvolvimento.

2 Materiais e métodos

Para resolução dos exercícios propostos, utilizamos as bibliotecas:

- **OpenCV**: Utilizada para ler, processar e exibir imagens
- **Numpy**: Responsável pelo tratamento dos dados das imagens como arrays. É usada para realizar operações matemáticas e manipulação dos pixels de forma vetorizada, fundamental para o processamento de imagens de forma performática
- **scipy**: Foi utilizada apenas a função `convolve2d` que aplica operações de convolução em imagens de forma a implementar os filtros de extração de bordas e aprimoramento de detalhes
- **matplotlib**: Utilizado para a visualização das imagens no ambiente do Jupyter Notebook

3 Códigos auxiliares

Para a resolução dos desafios propostos, criamos uma série de códigos auxiliares que serviram para encapsular a lógica de uma sequência de comandos. Os códigos auxiliares criados foram:

```
1 def show_image_in_window(image):
2     cv2.imshow('image', image)
3     cv2.waitKey(0)
4     cv2.destroyAllWindows()
5
6 def print_mono_image(image):
7     plt.imshow(image, cmap='gray')
8     plt.axis('off')
9     plt.show()
10
11 def print_image(image):
12     plt.imshow(image)
13     plt.axis('off')
14     plt.show()
```

Listing 1: Funções auxiliares para visualização de imagens

A função `show_image_in_window` serve para o OpenCV abrir uma imagem em uma tela separada, assim podemos verificar qual foi a imagem gerada. Além disso, temos a função `print_mono_image` que possibilita vermos a imagem sendo gerada no próprio Jupyter Notebook para imagens monocromáticas, e a função `print_image` para imagens coloridas, facilitando também o trabalho de correção.

Durante o desenvolvimento, a função `show_image_in_window` foi mais utilizada pois não havia necessidade das imagens ficarem como output de código o tempo todo, porém para a versão final do código, modificamos

todas as suas chamadas para funções de visualização no notebook, como a `print_mono_image` e `print_image`, conforme apropriado para cada tipo de imagem.

4 Exercícios e discussão

4.1 Exercício 1: Esboço a lápis

Objetivo: Transformar uma imagem colorida em um esboço a lápis por meio da conversão para tons de cinza, aplicação de blur gaussiano e divisão pixel a pixel.

Estratégia: Carregamos a imagem e convertemos para tons de cinza utilizando `cv2.cvtColor`. Em seguida, aplicamos um blur gaussiano com kernel de tamanho `(21, 21)` e `sigmaX = sigmaY = 0`, o que deixa o OpenCV calcular o sigma automaticamente. Para realçar os contornos, usamos a função `cv2.divide` dividindo a imagem original em tons de cinza por sua versão desfocada.

Discussão: A divisão sugerida no enunciado da questão tinha como objetivo tornar mais claras regiões em que o `pixel_blur < pixel_mono` resultam já que a divisão resultariam em um número maior que 1 e ao reescalarmos a divisão ficaria com um valor próximo do limite superior. Analogamente, o oposto aconteceria quando `pixel_blur > pixel_mono`, e essa última opção acontece principalmente em regiões de borda, já que o blur aumenta o valor dos pixels ao redor de uma borda ao aplicar uma média dos pixels ao redor.

Resultado: A imagem resultante se assemelha a um desenho a lápis, com destaque para as bordas e contornos principais.

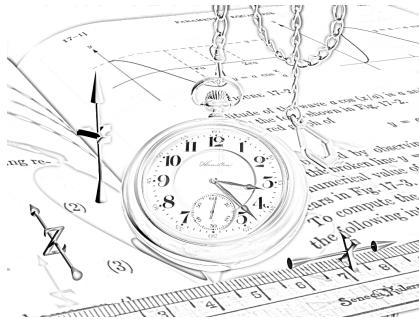


Figura 1: Resultado do Exercício 1

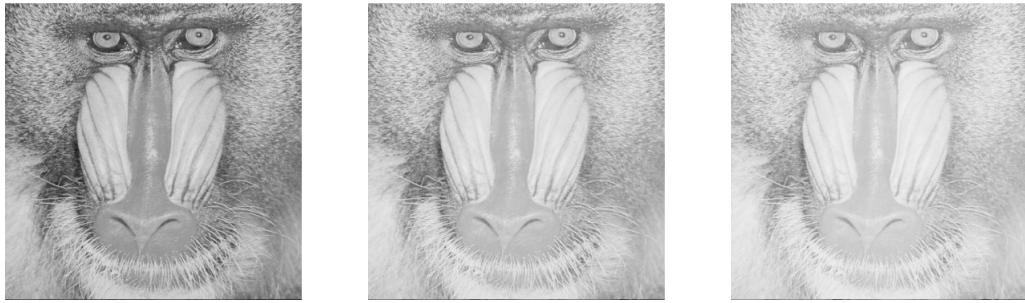
4.2 Exercício 2: Ajuste de brilho

Objetivo: Ajustar o brilho de uma imagem monocromática utilizando correção gama com diferentes valores de γ .

Estratégia: Primeiramente, carregamos a imagem monocromática e normalizamos os valores de pixel para o intervalo $[0, 1]$. Depois, aplicamos a transformação gama com `np.power` usando o valor `gamma = 3.5`. Após isso, reescalamos os valores de volta para o intervalo $[0, 255]$.

Discussão: O uso do `np.power` para aplicar a transformação $A^{1/\gamma}$ permite ajustar a percepção de brilho de forma não linear de forma vetorizada, contribuindo para performance da operação. Valores de gamma maiores que 1 clareiam a imagem pois sua operação semelhante a aplicar uma raiz de γ em um número entre 0 e 1, que causa o aumento desse número, e a normalização garante que os pixels estejam corretamente escalados antes e depois da transformação.

Resultado: Imagens mais claras ou mais escuras, dependendo do valor de γ , evidenciam diferentes aspectos da cena.



(a) Para $\gamma = 1.5$

(b) Para $\gamma = 2.5$

(c) Para $\gamma = 3.5$

Figura 2: Resultado do exercício 2

4.3 Exercício 3: Mosaico

Objetivo: Reorganizar blocos de uma imagem monocromática segundo uma nova ordem específica, formando um mosaico.

Estratégia: Carregamos a imagem e dividimos em blocos 4×4 . Em seguida, usamos um dicionário (`image_blocks_mapper`) para mapear a posição original de cada bloco para uma nova posição para que então eu utilizasse um novo dicionário que mapeasse as mesmas posições mas agora como índices de uma matriz (Ex: posição 2 da matriz original viraria uma tupla $(0, 1)$), o algoritmo usado para realizar o mapeamento está descrito abaixo, trata-se de um algoritmo simples que acha o valor da coluna ao obter o valor do resto do índice pelo tamanho do bloco e o valor da linha pelo valor da sua divisão inteira. Por fim, utilizei slicing com NumPy para copiar os blocos da imagem original para suas novas posições na imagem final.

```
1 def get_pos_from_coords(pos):
2     row = (pos - 1) // block_size
3     column = (pos - 1) % block_size
4     return row, column
```

Listing 2: Algoritmo para encontrar index de linhas e colunas

Discussão: O uso de slicing com mapeamento via dicionário permitiu reorganizar os blocos de forma clara. Durante o desenvolvimento, também precisei criar uma nova matriz para colocar os pixels embaralhados, já que percebi que não era possível usar a mesma matriz através do algoritmo escolhido por mim, já que da forma que foi pensada ao mapearmos, por exemplo, a posição 1 da matriz da esquerda na posição 1 da matriz da direita, perdíamos a informação do número anterior que fora substituído. Segue abaixo o algoritmo que fez a reordenação, sendo h a altura e w a largura da imagem:

```
1 for left_coord, right_coord in coords_mapper.items():
2     left_row, left_column = left_coord[0], left_coord[1]
3     right_row, right_column = right_coord[0], right_coord[1]
4     result_img[left_row * h: (left_row + 1) * h, left_column * w: (left_column + 1) *
      w] = image[right_row * h: (right_row + 1) * h, right_column * w: (right_column
      + 1) * w]
```

Listing 3: Algoritmo de slicing utilizado

Resultado: A imagem final apresenta um rearranjo dos blocos.

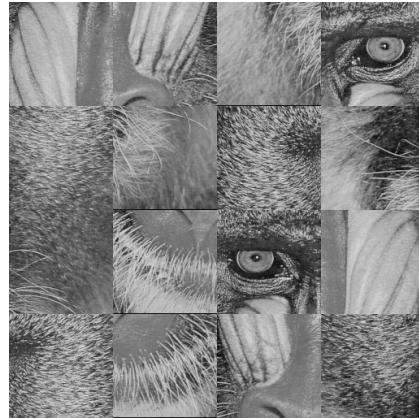


Figura 3: Resultado do rearranjo dos blocos

4.4 Exercício 4: Alteração de cores

Objetivo: Aplicar uma transformação linear nos canais RGB para simular o efeito de fotografias antigas.

Estratégia: Convertemos a imagem de BGR para RGB, e aplicamos uma multiplicação matricial entre cada pixel e a matriz de transformação sugerida. A matriz foi transposta (`M.transpose()`) para alinhar corretamente com os vetores de cores. O resultado foi limitado ao intervalo [0, 255] e convertido para `uint8`.

Discussão: A conversão foi necessária para que durante a multiplicação de matrizes os canais fossem multiplicados de forma correta, foi usado o operador `np.matmul()` para que a operação fosse vetorizada e também a imagem foi clipada para que os valores não ultrapassem 255.

Resultado: A imagem adquire tonalidades amareladas, características de fotos envelhecidas.



Figura 4: Imagem envelhecida

4.5 Exercício 5: Transformação de imagens coloridas

Objetivo: Aplicar duas transformações: uma linear nos canais RGB e outra para gerar uma imagem com uma única banda de cor baseada em média ponderada.

Estratégia: Para a primeira parte, reproveitamos a transformação linear feita no exercício anterior. Na segunda parte, utilizamos uma matriz 1×3 com os pesos de luminância (0.2989, 0.5870, 0.1140) e aplicamos sobre a imagem RGB usando multiplicação matricial, gerando uma imagem com apenas uma banda de cor.

Discussão: A primeira transformação altera a imagem de forma estética (efeito sépia). Já a segunda transformação, baseada na média ponderada de canais RGB, é uma conversão para tons de cinza. Um desafio técnico encontrado foi a incompatibilidade entre as ordens dos canais esperadas pelo OpenCV (BGR) e pelo Matplotlib (RGB). Para mostrar corretamente a imagem processada, foi preciso reconverter o resultado utilizando `cv2.cvtColor()` com o parâmetro `cv2.COLOR_RGB2BGR` antes da visualização, conforme evidenciado pela comparação na Figura 5c, onde a

ausência dessa reconversão resultava em canais trocados.

Resultado: Obtivemos imagens estilizadas ou com intensidade baseada na luminância.

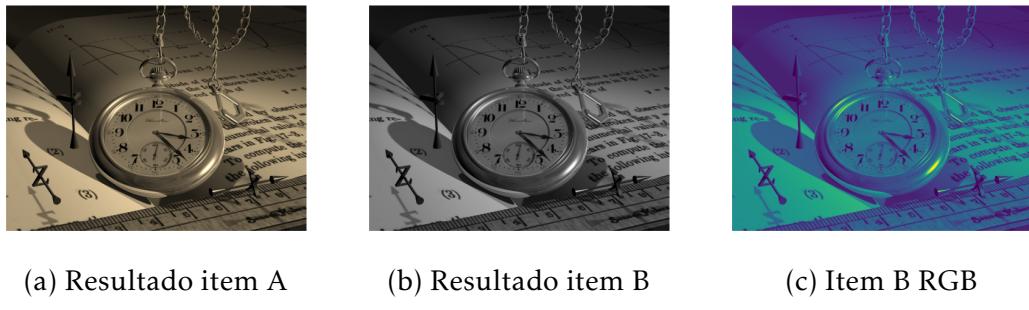


Figura 5: Resultado do exercício 5

4.6 Exercício 6: Planos de bits

Objetivo: Extrair os planos de bits de uma imagem monocromática.

Estratégia: Utilizamos operações bit a bit («, &, ») para extraír planos de bits específicos da imagem. Para cada plano, criamos uma máscara com `1 « bit_index`, aplicamos `bitwise_and`, depois deslocamos os bits para a direita e multiplicamos por 255 para exibir o plano como imagem.

Discussão: Esse método permite extraír de forma performática os bits que compõem os valores de intensidade da imagem. A visualização de cada plano ajuda a entender como diferentes bits afetam a representação visual da imagem, bits mais significativos, tendem a se parecer mais com a imagem original, já bits menos significativos só nos mostram um ruído.

```
1 def extract_bit(bit_index):  
2     bit_mask = 1 << bit_index
```

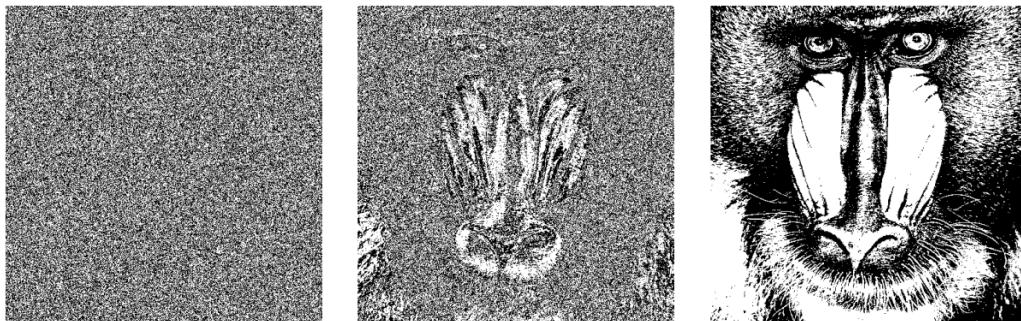
```

3     extracted_bits = np.bitwise_and(image, bit_mask) #Gera um valor como
4         00100000
5     normalized_bit_image = extracted_bits >> bit_index # Converte os valores como
6         00000000 ou 00100000 para 0 ou 1
7     return normalized_bit_image * 255

```

Listing 4: Função utilizada para extração de bits

Resultado: Planos inferiores contêm mais ruído, enquanto os superiores preservam a estrutura principal da imagem.



(a) Plano de bits ordem 0 (b) Plano de bits ordem 4 (c) Plano de bits ordem 7

Figura 6: Resultado do exercício 6

4.7 Exercício 7: Combinação de imagens

Objetivo: Combinar duas imagens monocromáticas por meio de média ponderada.

Estratégia: Carregamos duas imagens monocromáticas (baboon e butterfly) e convertemos para float32. Utilizamos multiplicação escalar com os pesos α e β , em seguida somamos os resultados com `np.add`. A

imagem final foi normalizada para o intervalo $[0, 255]$ com `np.clip` e convertida para `uint8`.

Discussão: A utilização dessa estratégia foi bem direta, só precisamos ter que tomar cuidado para não realizar operações de forma escalar, por isso usamos o `np.multiply` e o `np.add` para garantir a vetorização das operações.

Resultado: A imagem resultante apresenta uma mistura entre as duas imagens originais.



(a) $\alpha=0.8$ e $\beta = 0.2$

(b) $\alpha=0.5$ e $\beta = 0.5$

(c) $\alpha=0.2$ e $\beta = 0.8$

Figura 7: Resultado do exercício 7

4.8 Exercício 8: Transformação de intensidade

Objetivo: Aplicar diferentes transformações nos níveis de cinza da imagem, como negativo, mapeamento de intervalo e espelhamento.

Estratégia: Implementamos diversas transformações sobre uma imagem monocromática:

- (a) Inversão dos tons com `np.negative`.

- (b) Aplicação de transformação linear com a equação $g = \frac{20 \cdot f + 5100}{51}$.
- (c) Inversão horizontal de linhas pares com slicing `img[::2] = img[::2, ::-1]`.
- (d) Espelhamento vertical da metade inferior da imagem com slicing reverso.
- (e) Espelhamento total da imagem, invertendo todas as linhas.

Discussão:

- (a) Para o item A, não houve nenhum desafio muito grande, a aplicação direta do `np.negative` já retornou a imagem esperada
- (b) Para o item B, foi feito uma regra de proporcionalidade simples para chegarmos na equação $g = \frac{20 \cdot f + 5100}{51}$, em que f é a imagem original e g sua imagem transformada. Entretanto, não tivemos sucesso em obter a imagem proposta pelo exercício, a imagem ficou com aspecto de lavada e não com o aspecto escuro assim como o exercício sugeriu.
- (c) Para o item C, fizemos apenas uma manipulação simples usando slicing `img[::2] = img[::2, ::-1]`.
- (d) Para o item D, realizamos uma atribuição onde a segunda metade do array `img` (`img[w//2:w]`) recebe os valores da primeira metade (`img[w//2:0:-1]`) em ordem reversa, criando um espelhamento horizontal
- (e) Para o item E, fizemos com que a imagem recebesse sua versão revertida com o comando `img = img[w:0:-1]`

Resultado: Obtivemos imagens modificadas com efeitos visuais distintos, como inversão de direção e simetria.

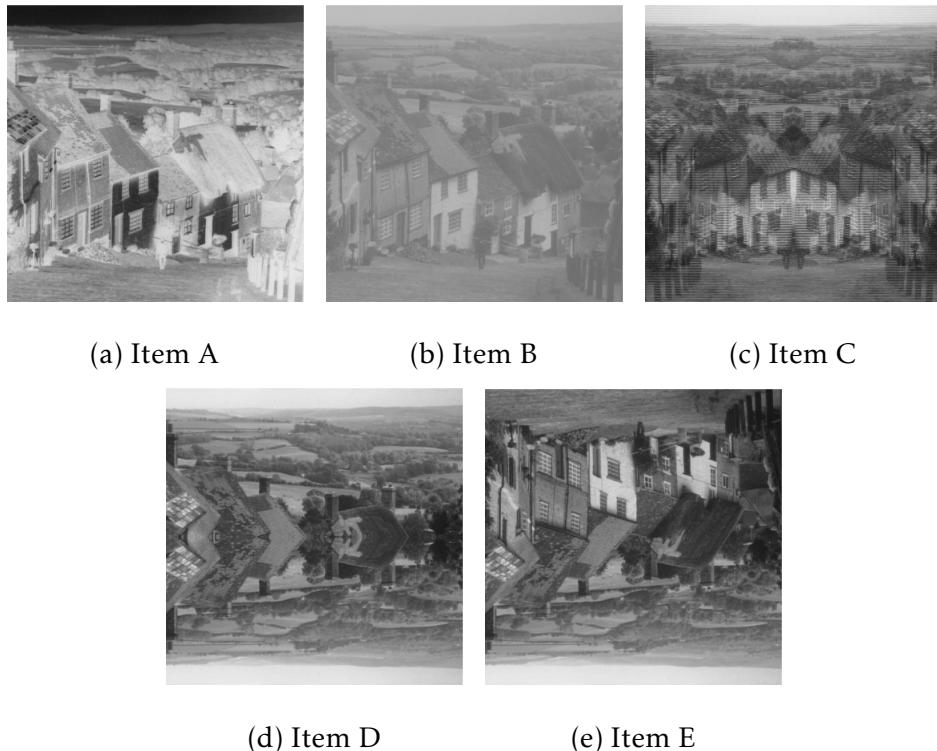


Figura 8: Resultados Exercício 8

4.9 Exercício 9: Quantização de imagens

Objetivo: Reduzir o número de níveis de cinza usados para representar uma imagem.

Estratégia: Aplicamos a quantização reduzindo os níveis de cinza para múltiplos de k , onde $k = \frac{256}{n}$, com n sendo os níveis desejados, depois reescalamos a imagem multiplicando por k . A expressão ficou então:

$$\left\lfloor \frac{\text{img}}{k} \right\rfloor \cdot k$$

.

Discussão: Neste exercício, embora a solução adotada pareça simples, envolvendo apenas uma multiplicação e uma divisão, conseguimos resultados visuais muito próximos das imagens fornecidas no PDF.

No entanto, ao reduzir para o **nível 2**, enfrentamos um comportamento inesperado na geração da imagem. Esperávamos que o algoritmo aplicasse uma **binarização clássica**, como:

Se $x < 128 \Rightarrow 0$ (preto)

Se $x \geq 128 \Rightarrow 255$ (branco)

Mas o que ocorreu foi:

Se $x < 128 \Rightarrow 0$ (preto)

Se $x \geq 128 \Rightarrow 128$ (cinza escuro)

Isso gerou uma imagem com tons mais escuros, ao invés do contraste marcado entre preto e branco.

```
1 def quantize(img, n):
2     k = 256 // n
3     return (img // k) * k
```

Listing 5: Algoritmo utilizado para quantização das imagens

Resultado: A imagem perde gradativamente detalhes e suavidade à medida que os níveis diminuem.

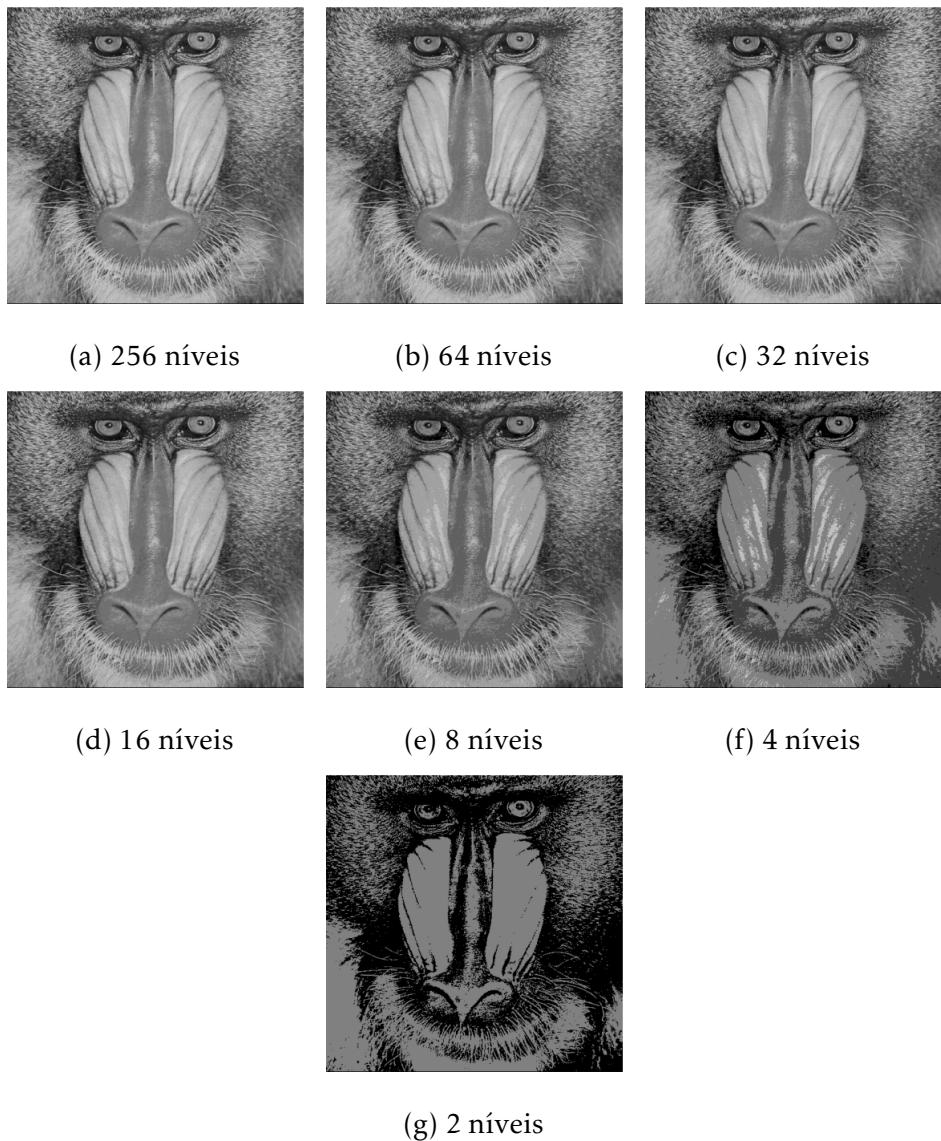


Figura 9: Resultados Exercício 9

4.10 Exercício 10: Filtragem de imagens

Objetivo: Aplicar filtros espaciais sobre uma imagem monocromática, utilizando operações de convolução com máscaras definidas previamente. O objetivo é observar os diferentes efeitos produzidos pelos filtros sobre a

imagem original, como realce de bordas, suavização e detecção de padrões.

Estratégia: Foi utilizada convolução 2D entre a imagem de entrada e as máscaras fornecidas. Cada filtro foi aplicado individualmente, exceto os filtros h_3 e h_4 , que também foram combinados de acordo com a fórmula:

$$\text{resultado} = \sqrt{(h_3)^2 + (h_4)^2}$$

A função ‘convolve2d’ do pacote `scipy.signal` foi utilizada com o modo de borda “same” para preservar o tamanho da imagem original. Os resultados foram normalizados para o intervalo [0, 255]. Dado a natureza repetitiva da solução, optamos por criar uma função que recebia o kernel e o caminho da imagem original, retornando depois a imagem transformada, sendo ela:

```
1 def apply_filter(kernel, image):
2     convolved_image = convolve2d(image, kernel, mode='same', boundary='fill',
3                                   fillvalue=0)
4     clipped_image = np.clip(convolved_image, 0, 255)
5     print_mono_image(clipped_image.astype(np.uint8))
6     return clipped_image
```

Listing 6: Algoritmo utilizado para quantização das imagens

Discussão: A seguir descrevemos os efeitos observados com cada filtro:

1. **Filtro h_1 :** Destaca fortemente os contornos da imagem, fazendo com que nós percebemos nitidamente, a imagem escolhida para mostrar isso é a da cidade em que é possível ver o desenho de todas as casas

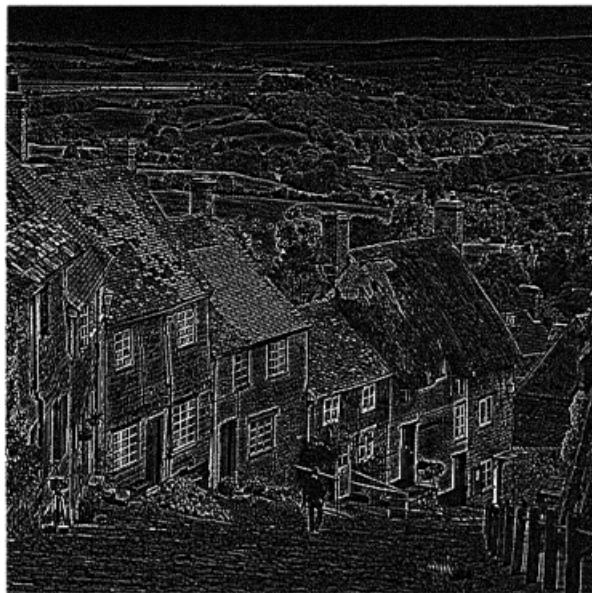


Figura 10: Filtro h_1

2. **Filtro h_2 :** Este filtro aplica um blur gaussiano 5x5, suavizando a imagem. Comparando a imagem original na Figura 11a com o resultado após a aplicação do filtro na Figura 11b, podemos observar claramente a perda de nitidez.

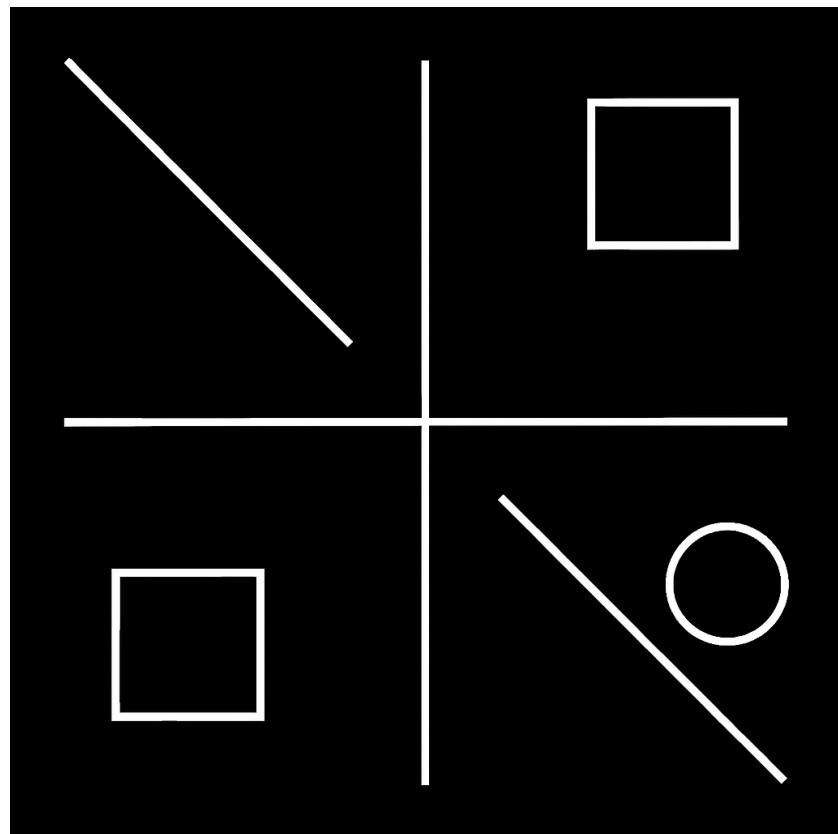


(a) Imagem original de um trem



(b) Imagem após filtro h_2

3. **Filtro h_3 :** Este filtro realça bordas verticais. Para verificar isso, utilizamos a imagem Figura 12a, gerada através de inteligência artificial, que tem elementos bem simples mas que tornam a visualização da funcionalidade do filtro clara. A Figura 12b mostra o filtro marcando apenas as bordas verticais



(a) Imagem original gerada artificialmente



(b) Imagem após filtro h_3

4. **Filtro h_4 :** Filtro que realça bordas horizontais, utilizando a mesma imagem que o exemplo anterior, podemos ver que agora o filtro está destacando os elementos horizontais da imagem

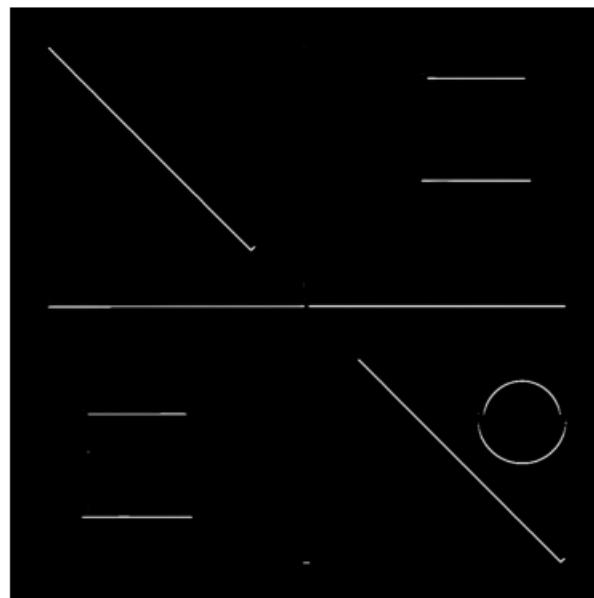


Figura 13: Filtro h_4

5. **Combinação h_3 e h_4 :** Utilizando a equação $\sqrt{(h_3)^2 + (h_4)^2}$ obtemos um resultado mais completo, esse filtro destaca os contornos em todas as direções. Podemos ver através da Figura 14 que todos os contornos estão muito bem definidos

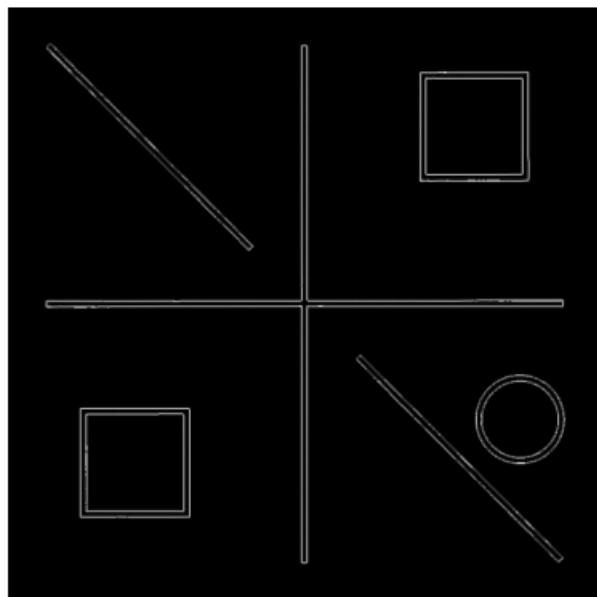
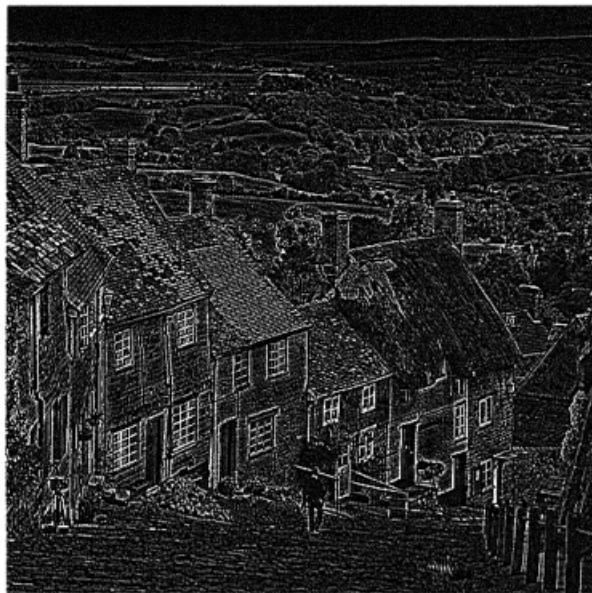


Figura 14: Combinação h_3 e h_4

6. **Filtro h_5 :** Destaca o contorno da imagem, mas comparando com o filtro h_1 , podemos notar que ele destaca mais fracamente esses contornos. É notório, por exemplo, que existe um contraste grande nos tijolos e telhados das casas ao utilizar o filtro h_1 e esse contraste exacerbado não é percebido usando o filtro h_5



(a) Imagem da cidade com filtro h_1



(b) Imagem da cidade com filtro h_5

7. **Filtro h_6 :** Gera um blur suave e reduz pequenos detalhes da imagem, podemos ver por exemplo, que na imagem com o filtro existem

menos detalhes nas folhagens a esquerda, que são bem mais perceptíveis ao olhar a imagem original.



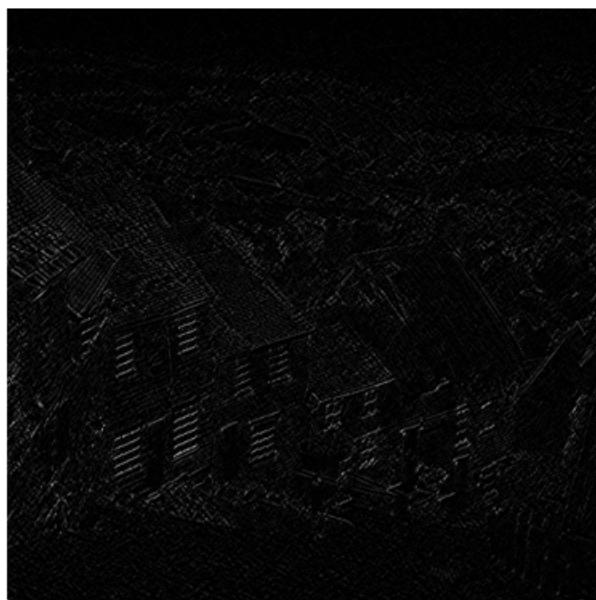
(a) Imagem original de um trem



(b) Imagem após filtro h_6

8. **Filtros h_7 e h_8 :** Estes filtros detectam bordas diagonais, mas em direções opostas. O filtro h_7 destaca contornos na direção diagonal descendente (\swarrow), enquanto o filtro h_8 realça elementos na direção diagonal ascendente (\nearrow). Esta diferença fica evidente ao analisar os resultados: nas imagens processadas com h_7 (Figura 17a), os contor-

nos diagonais descendentes aparecem mais nítidos, enquanto com h_8 (Figura 17b) os telhados e outras estruturas com inclinação ascendente ganham destaque.



(a) Imagem após filtro h_7



(b) Imagem após filtro h_8

9. **Filtro h_9** : Aplica um blur direcionado ao longo da diagonal descendente (\nwarrow). Analisando os resultados na imagem da Mona Lisa, observamos que os detalhes finos e texturas originais se transformam em listras diagonais, criando um efeito que ressalta essa direção específica.

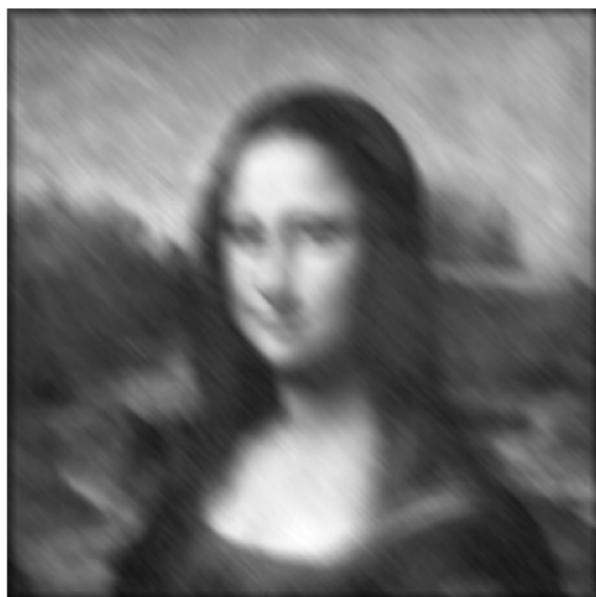
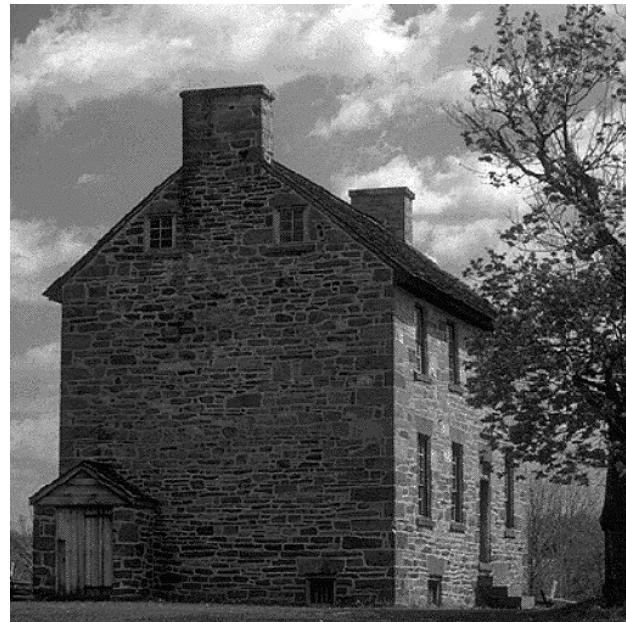
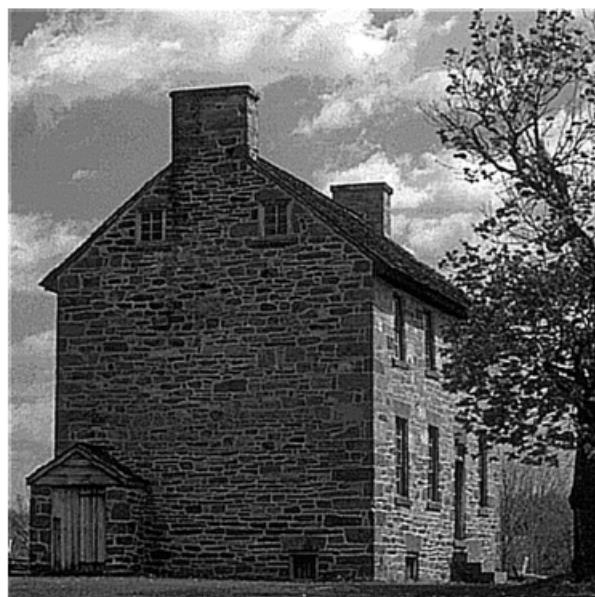


Figura 18: Imagem após filtro h_9

10. **Filtro h_{10}** : Produz um efeito de realce de bordas, pode ser percebido, por exemplo, utilizando a imagem da casa, é notório que os tijolos da frente tem seus contornos mais marcados ao serem comparados com a imagem original.



(a) Imagem original da casa



(b) Imagem após filtro h_{10}

11. **Filtro h_{11} :** Realça bordas diagonais na direção ascendente (\nearrow). Na imagem do baboon monocromática, observamos que os pelos orien-

tados nessa direção são significativamente destacados em comparação com pelos em outras orientações.

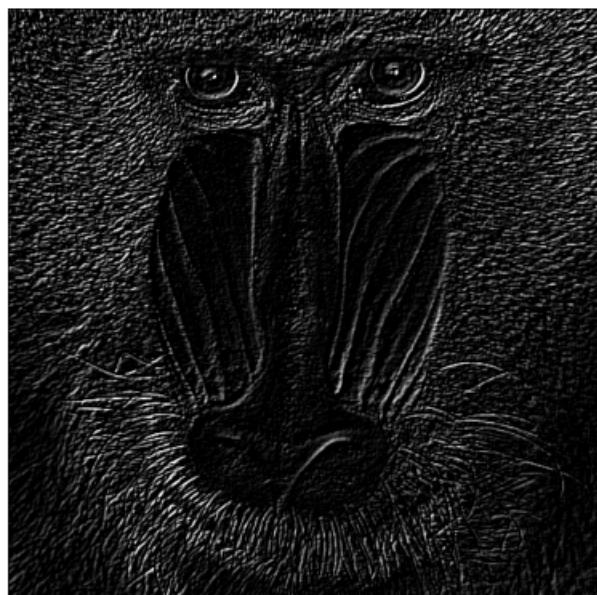


Figura 20: Imagem após filtro h_{11}