

Universidade Estadual de Campinas
MC920 - Introdução ao Processamento de Imagem Digital
Prof. Dr. Hélio Pedrini

Trabalho 1

Mateus de Lima Almeida
242827

7 de abril de 2025

1 Introdução

O presente trabalho aborda a implementação, realização e exploração de diversos aspectos e técnicas de manipulação de imagens sugeridas em sala na matéria de Introdução ao Processamento de Imagem Digital (MC920). Com o objetivo de introduzir e solidificar o conhecimento aos conceitos básicos de processamento de imagens, diversas operações e transformações diferentes foram feitas em imagens, e seus resultados foram analisados.

Este relatório detalha a metodologia adotada na execução dos exercícios sugeridos e também realiza a discussão de seus resultados, bem como a exploração de hipóteses surgidas no processo, junto também com a discussão de seus testes e resultados.

2 Materiais e Métodos

Para todas as tarefas a seguir, os materiais utilizados foram a linguagem Python 3.9 junto com as bibliotecas OpenCV2, Scikit-Image, Numpy e Pillow. Para utilização das bibliotecas e execução dos códigos, também foi utilizado um ambiente virtual (venv), onde foram instaladas as bibliotecas citadas.

2.1 Organização de Arquivos

A pasta do projeto foi organizada de modo a facilitar a visualização dos resultados de cada exercício de maneira independente. Dentro da pasta estão:

- Arquivos Python de cada exercício (de 1 a 10)
- Arquivo *requirements.txt* com as dependências necessárias do projeto.
- Pasta *images* que contém:
 - Imagens PNG que serão os Inputs dos exercícios
 - Pasta *outputs* que contém pastas (enumeradas de 1 a 10) contendo as imagens-saída de cada exercício

2.2 Versão Python e Bibliotecas

A versão de Python utilizada, junto com as bibliotecas e suas versões foram as seguintes:

- Python 3.9.1
- Pillow 11.1.0
- Opencv-python 4.11.0.86
- Numpy 2.0.2

2.3 Execução dos códigos

Os códigos feitos para realizar as operações e transformações sugeridas neste trabalho foram todos executados dentro de um ambiente virtual *venv*, então é recomendável que para areprodução seja feito o mesmo. É necessário que as dependências de *requirements.txt* sejam instaladas.

Para executar um determinado programa, basta inserir o comando básico para rodar um arquivo *.py* no terminal (certifique-se de estar na pasta correta).

```
python ex1.py
```

O arquivo exemplo utilizado foi o *ex1.py*, mas a pasta contém arquivos enumerados de 1 à 10. Em seguida, basta digitar o nome da imagem que se deseja usar, ou seja, se desejamos executar *ex2.py* e usar a imagem *city.png*, deve-se fazer:

```
python ex2.py
Digite o nome da imagem (sem '.png'): city
```

A imagem selecionada deve estar presente na pasta *images*. A seguir, uma lista das imagens presentes na pasta, que podem ser usadas como input:

- Imagens Monocromáticas:

- baboon_monocromatica
- buckethead
- butterfly
- house
- city
- iwakura_lain
- seagull
- tung_sahur_greyscale

- Imagens RGB:

- watch
- windmill
- tung_sahur

Caso seja desejado, mais imagens podem ser adicionadas manualmente na pasta *images* para serem usadas como input.

3 Resultados, testes e discussão

3.1 Esboço à Lápis

O objetivo do primeiro item proposto foi realizar a implementação de um efeito de esboço à lápis em uma imagem. A imagem escolhida para aplicar o efeito foi a imagem *watch.png* (Figura 1), que pode ser vista abaixo.



Figura 1: watch.png

O primeiro passo para aplicar o efeito desejado foi converter a imagem de RGB para escala de cinzas (Figura 2) com ajuda da biblioteca Pillow. Em seguida, um filtro de desfoco gaussiano foi aplicado na imagem em tons de cinza. Para aplicação do filtro, foi utilizada a seguinte função da biblioteca OpenCV2:

```
grey_img_blur_np = cv2.GaussianBlur(grey_img, (21, 21), 0)
```

Onde o parâmetro *grey_img* é a imagem em tons de cinza (greyscale), *(21, 21)* é o tamanho da máscara (21x21), que deve ser sempre uma matriz de dimensões ímpares, para que haja um centro e *0* é o desvio padrão. Para tratamento de bordas, a função *GaussianBlur()* possui um parâmetro *'borderType'*, onde o valor padrão (que é o nosso caso) faz com que os valores de filtro que fiquem para fora da imagem sejam "refletidos", sendo utilizados dentro da imagem. O resultado da aplicação do filtro de desfoco gaussiano é uma imagem desfocada (Figura 3).



Figura 2: watch.png em tons de cinza



Figura 3: watch.png em tons de cinza com desfoco gaussiano

Após isso, basta dividir a imagem em tons de cinza pela imagem desfocada para obter a imagem desejada com o efeito de esboço. Esse processo, porém, possui detalhes acerca dos valores dos pixels. Ao dividir uma imagem pela outra, obteremos uma queda significativa no valor dos pixels da nossa imagem resultado, gerando uma imagem quase completamente escura (Figura 4), por isso, a imagem resultante da divisão deve ser normalizada

novamente para um intervalo $[0, 255]$. Como a imagem desfocada é suficientemente parecida com a versão sem o desfoque em greyscale, a divisão de uma pela outra resultará em uma imagem onde a grande maioria dos pixels se encontrará no intervalo $[0, 1]$, isso significa que para obter uma imagem legível, basta multiplicar todos os pixels da imagem escura por 255. A divisão de uma imagem por outra e a multiplicação por 255 vão gerar valores que não se encontram no domínio dos inteiros, mas ao converter o array numpy para uma imagem usando a biblioteca Pillow, a função se encarrega desses casos. O resultado final é uma imagem que se assemelha a um esboço à lapis (Figura 5).



Figura 4: watch.png após divisão, sem normalização

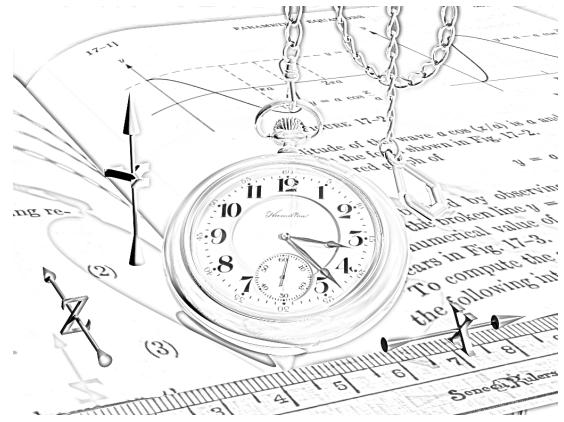


Figura 5: watch.png final após normalização

Ao analisar o processo e o resultado, é possível inferir que o funcionamento desse processo se dá por causa das bordas bem definidas e boa saturação da imagem original escolhida. O desfoque gaussiano torna cada pixel uma espécie de "combinação" / "média" dos pixels em volta, fazendo com que as bordas, antes bem definidas por uma alta diferença de valores de cinza, agora se tornem menos marcantes pois essas diferenças foram atenuadas. Essa análise levanta diversos questionamentos: e se a imagem escolhida para aplicar o processo for uma imagem com baixa saturação? Qual diferença o tamanho da máscara faz no processo? A seguir, esses questionamentos serão explorados.

Primeiramente, exploraremos mudanças no tamanho da máscara do desfoque gaussiano realizando o mesmo procedimento descrito anteriormente, mas mudando o argumento da função *GaussianBlur()* de $(21, 21)$ para $(5, 5)$, $(51, 51)$ e $(1023, 1023)$, ou seja, faremos testes com máscaras maiores e outro teste com uma máscara menor. Os resultados podem ser vistos a seguir:



Figura 6: watch.png com desfoco (máscara 51x51)

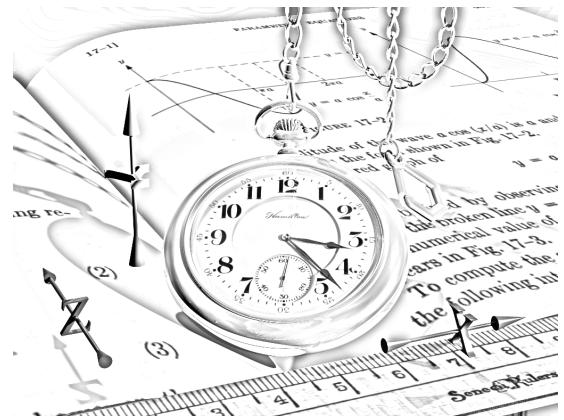


Figura 7: watch.png final usando máscara 51x51

Observando os resultados utilizando a máscara 51x51, é possível notar na Figura 6 que, assim como esperado, o desfoco gaussiano aplicado com uma máscara maior fez com que a imagem ficasse mais desfocada. Também é possível notar na Figura 7 que o resultado final possui linhas mais escuras, e outras partes escuras da imagem original (como a sombra sob o relógio) também foram realçadas. Com isso, podemos inferir que, com uma máscara suficientemente grande, a imagem desfocada seria apenas uma imagem cinza irreconhecível, e nosso resultado, em vez de um esboço, seria a imagem original (em greyscale) com o brilho aumentado, mas ainda com degradês e tons de cinza mais claros presentes, pois a imagem desfocada seria mais distante (visualmente) da imagem sem o desfoco, fazendo com a divisão entre elas não seja capaz de "filtrar" detalhes mais específicos da imagem. Para comprovar isso, foi gerada também uma imagem final obtida usando uma máscara 1023x1023 (Figura 9) e a imagem desfocada com o mesmo filtro (Figura 8).

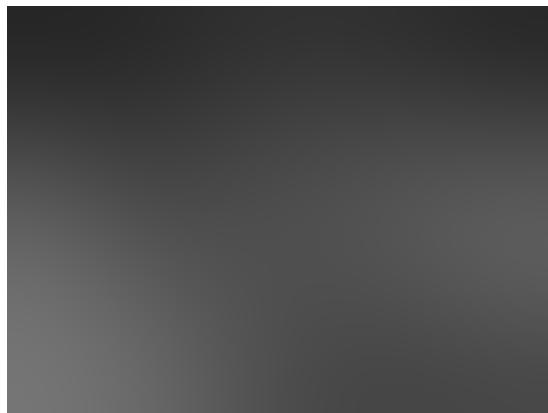


Figura 8: watch.png com desfoco (máscara 1023x1023)



Figura 9: watch.png final usando máscara 1023x1023

Agora, faremos o mesmo procedimento com uma máscara (5,5):



Figura 10: *watch.png* com desfoque (máscara 5x5)

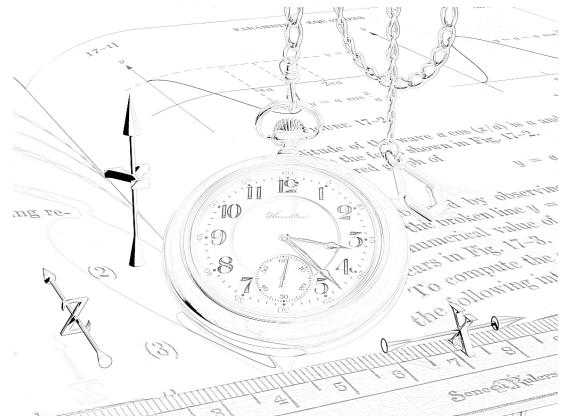


Figura 11: *watch.png* final usando máscara 5x5

Como podemos ver na Figura 10, a utilização de uma máscara menor na aplicação do desfoque gaussiano nos proporcionou uma imagem com menos desfoque, e na Figura 11 podemos ver que o resultado obtido na imagem final é o contrário do resultado obtido com um filtro grande, já que as linhas se tornaram mais fracas.

Agora, faremos o procedimento com uma versão de *watch.png* normalizada para o intervalo de [100, 150], dessa forma as diferenças de valores nas bordas dos objetos da imagem serão bem mais baixas. O resultado esperado é que, dessa vez, a imagem resultante seja bem fraca, ou até mesmo completamente branca ou cinza. A imagem normalizada pode ser vista na Figura 12 e o resultado do processo pode ser visto na Figura 13. Assim como esperado, o resultado final é uma imagem onde as linhas do efeito são quase irreconhecíveis devido à claridade. Com isso, conclui-se que quanto mais variado for o intervalo da imagem utilizada no processo, melhor será o resultado final.



Figura 12: *watch.png* normalizada em [100, 150]



Figura 13: *watch.png* final

3.2 Ajuste de Brilho

O objetivo do segundo item proposto foi aplicar uma correção gama para ajustar o brilho de uma imagem monocromática. A imagem escolhida para aplicar o efeito foi a imagem *baboon_monocromatica.png* (Figura 14).

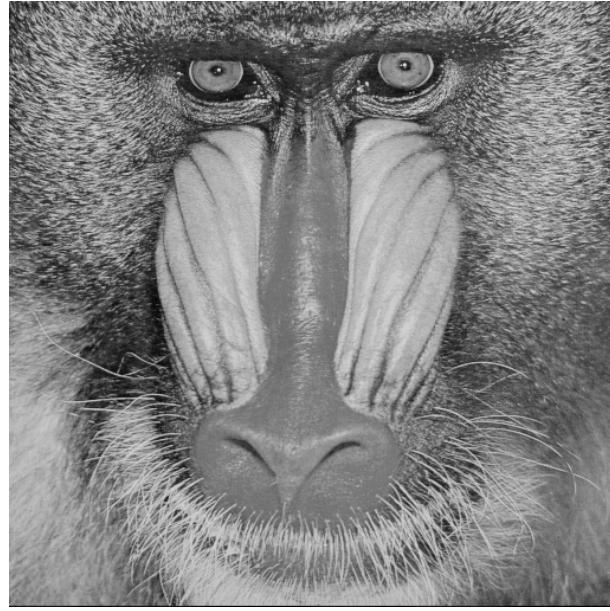


Figura 14: baboon_monocromatica.png

A equação que relaciona o valor de gama com a imagem se dá por $A = B^{\frac{1}{\gamma}}$, ou seja, para todo pixel B em uma imagem, a imagem resultado terá um pixel A que obedece a igualdade. Porém, para utilizar a equação na imagem, é necessário transformar o intervalo $[0, 255]$ dos pixels em um intervalo $[0, 1]$. Para realizar isso, basta dividir todos os pixels por 255. Os valores de γ escolhidos para serem aplicados foram $\gamma = 1.5$, $\gamma = 2.5$ e $\gamma = 3.5$.

Ao dividir a imagem por 255, como no item anterior, obteremos uma imagem escura onde os pixels se encontram no intervalo $[0, 1]$. Em seguida, basta aplicar a equação com o gama escolhido na imagem. Após isso, as imagens devem ser normalizadas novamente para um intervalo $[0, 255]$. Os resultados para cada valor de γ podem ser vistos nas imagens abaixo (Figura 15, Figura 16 e Figura 17).

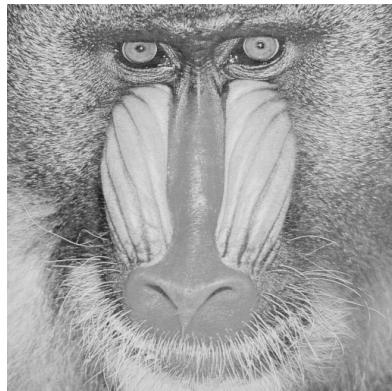


Figura 15: $\gamma = 1.5$

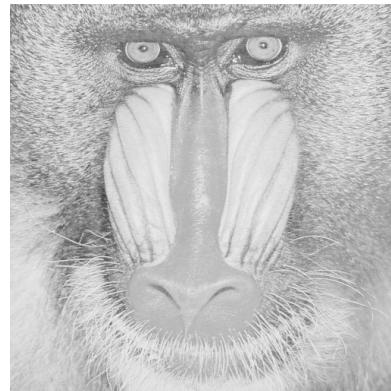


Figura 16: $\gamma = 2.5$

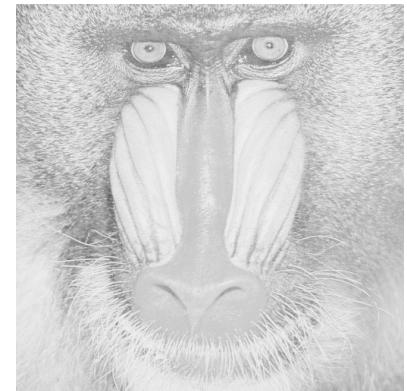


Figura 17: $\gamma = 3.5$

3.3 Mosaico

O objetivo do terceiro item proposto foi rearranjar uma imagem em um mosaico de 4 x 4 blocos seguindo o padrão visto na (Figura 20). Nessa tarefa, a imagem *baboon_monocromatica.png* (Figura 54) foi selecionada intencionalmente, visto que suas di-

mensões de 512 x 512 permitem a divisão da imagem em 4 blocos tanto verticalmente quanto horizontalmente, uma vez que 512 é divisível por 4.

Para montar o mosaico, o numpy array representando a imagem é recortado 4 vezes verticalmente e horizontalmente utilizando slicing, gerando 16 matrizes menores (submatrizes 128 x 128 da original) que representam os 16 blocos mostrados na Figura 19.

Após obter as 16 submatrizes 128 x 128, elas são concatenadas utilizando:

```
linha_1 = np.concatenate((pedaco_6, pedaco_11, pedaco_13, pedaco_3), 1)
```

Onde *linha_1* representa uma submatriz 128 x 512 formada pelos blocos 6, 11, 13 e 3 (no código chamados de *pedacos*) da Figura 20 e o argumento 1 é o eixo pelo qual os arrays serão unidos, nesse caso o eixo escolhido é 1, pois deseja-se concatenar as matrizes horizontalmente. Esse processo é realizado para cada uma das 4 linhas de submatrizes, e por fim, as linhas são concatenadas da mesma forma, mas usando o argumento 0 para o parâmetro de eixo, para uní-las verticalmente. O resultado é a imagem mostrada na Figura 21.

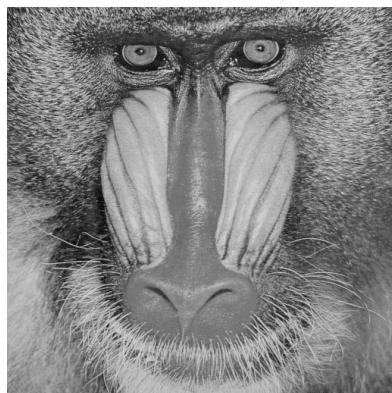


Figura 18: (a) imagem

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figura 19: (b) ordem dos blocos

6	11	13	3
8	16	1	9
12	14	2	7
4	15	10	5

Figura 20: (c) nova ordem dos blocos

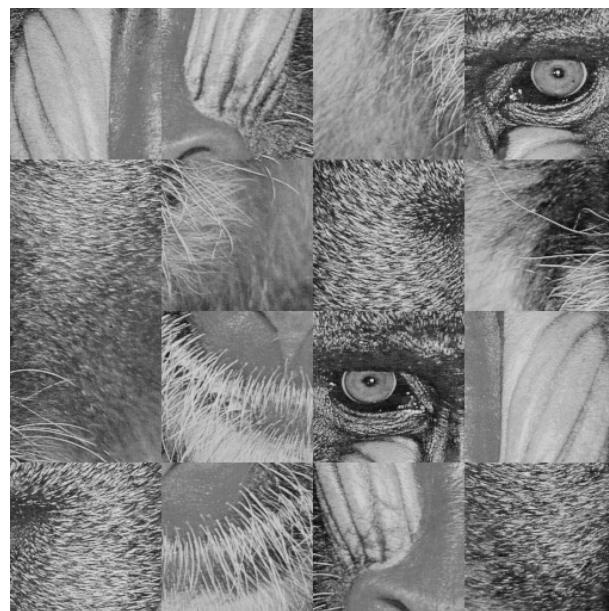


Figura 21: baboon_mosaico.png

3.4 Alteração de Cores

O objetivo do quarto item proposto foi aplicar uma transformação linear nas cores de uma imagem através de uma matriz de transformação:

$$A = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix}$$

O filtro descrito aplica na imagem um efeito de fotografia antiga por conta da mudança nos valores RGB da imagem. A imagem escolhida para a aplicar a transformação foi a *windmill.png* (Figura 22).



Figura 22: *windmill.png*

Para realizar a transformação, foi realizado um produto escalar entre a transposta da matriz A e o array numpy que representa a imagem (uma lista de matrizes). Para realizar o produto foi utilizada a função *dot()* do numpy, como é possível ver a seguir:

```
transposta = filtro.T
final_img = img.dot(transposta)
```

Para que a transformação seja feita corretamente, é necessário ter em mente que a primeira coluna da matriz A diz respeito aos valores que multiplicarão o plano R da imagem, já a segunda coluna possui os valores que multiplicarão o plano G e a terceira coluna possui os valores que multiplicarão o plano B. Para que essas condições sejam verdadeiras, é necessário transpor a matriz A. O resultado poderá conter valores fora do intervalo [0, 255], então, após realizar o produto escalar, é necessário normalizar/restringir os valores no intervalo desejado. Nesse caso, a função escolhida para a tarefa foi a função *clip()*.

```
final_img = np.clip(final_img, 0, 255)
```

Essa função faz com que todo pixel $B > 255$ fique igual a 255, e quando $B < 0$, B se torna igual a 0 (os valores de 0 e 255 foram colocados como argumentos na função). Após isso, temos o resultado da aplicação do filtro, que pode ser visto na Figura 23.



Figura 23: *windmill.png* com filtro de envelhecimento aplicado

Caso façamos o produto escalar sem transpor a matriz A , teremos um resultado bem diferente do desejado, com a imagem ficando muito esverdeada, como é possível ver na Figura 24.



Figura 24: *windmill.png* com filtro aplicado incorretamente

3.5 Transformação de Imagens Coloridas

O objetivo do quinto item proposto consistem em alterar uma imagem colorida de acordo com dois padrões de operações dados. A imagem na qual realizaremos tais transformações será a *windmill.png* (22), a mesma utilizada no item anterior, o motivo para isso logo ficará mais claro.

3.5.1 Transformação RGB

A primeira alteração será feita de acordo com as operações:

$$R' = 0.393R + 0.769G + 0.189B$$

$$G' = 0.349R + 0.686G + 0.168B$$

$$B' = 0.272R + 0.534G + 0.131B$$

Isso significa que, a imagem desejada terá cada um de seus planos RGB, denotados por R', G' e B', compostos por diferentes ponderações dos três planos R, G e B da imagem original. Na prática, isso é exatamente o que foi feito para obter a imagem final. A camada R' é composta pela soma da camada R da imagem original multiplicada por 0.393, da camada G da imagem original multiplicada por 0.769 e da camada B da imagem original multiplicada por 0.189. As camadas G' e B' seguem a mesma lógica, porém com diferentes números multiplicando R, G e B. A versão em python dessa operação para R' se encontra a seguir:

```
filtro = np.array([[0.393, 0.769, 0.189], ..., [0.272, 0.534, 0.131]])
img_R1 = img[:, :, 2] * filtro[0, 0]
img_G1 = img[:, :, 1] * filtro[0, 1]
img_B1 = img[:, :, 0] * filtro[0, 2]
img_R = img_R1 + img_G1 + img_B1
```

Após a soma de R', G' e B', cada matriz é normalizada usando *clip()* para garantir que os pixels estão no intervalo [0, 255]. E em seguida, R', G' e B' são concatenadas usando:

```
final_img = np.stack([img_R, img_G, img_B], axis=-1)
```

O resultado da operação é uma imagem (Figura 25) idêntica à imagem obtida no item anterior, uma versão da imagem original, mas com um filtro de envelhecimento.



Figura 25: windmill.png com filtro de envelhecimento

Com isso, é possível concluir que, na verdade, as operações aqui feitas também não se diferem das operações realizadas pela função *dot()* utilizada anteriormente.

3.5.2 Transformação 1 Banda de Cor

A segunda operação a ser explorada será:

$$I = 0.2989R + 0.5870G + 0.1140B$$

Analizando o formato da operação, é possível notar que, desta vez, o resultado será apenas uma banda de cor, ou seja, nossa imagem resultante será representada por uma única matriz (I) em vez de três matrizes (R , G e B), indicando que nosso resultado esperado muito provavelmente se trata de uma imagem em tons de cinza. O processo de realização da operação indicada é exatamente o mesmo realizado na operação anterior, porém com menos etapas, visto que dessa vez o resultado será apenas 1 banda de cor. A imagem resultante do processo (Figura 26) é justamente o que esperávamos analisando a operação, uma imagem em tons de cinza.



Figura 26: windmill.png em tons de cinza

3.6 Planos de Bits

O objetivo do sexto item proposto foi extrair os planos de bit de uma imagem monocromática. A imagem escolhida para extrair os planos de bits é a imagem *city.png*, mostrada na Figura 27.



Figura 27: city.png

Para realizar a extração dos planos de bit, a técnica utilizada foi construir uma máscara de bits que coletasse apenas o bit menos significativo do valor de cada pixel e executar uma quantidade de operações *right shift* necessárias para colocar os bits desejados como bits menos significativos. Cada pixel da imagem é representado por um valor que pertence ao intervalo $[0, 255]$, ou seja, sua representação em binário possui 8 bits, tendo a forma:

bbbbbbbb (b = 0 ou 1)

Desta forma, a máscara utilizada, uma operação *AND* executada sobre o valor 1, é capaz de extrair o bit menos significativo da seguinte forma:

```
01011001      <- valor arbitrário exemplo  
00000001      <- máscara (valor 1 binário)  
-----  
00000001      <- resultado do AND
```

Após isso, como a imagem resultado é formada apenas por 0s ou 1s, basta multiplicar o array por 255, fazendo com que os valores que antes eram 1 se tornem 255, obtendo uma imagem em preto e branco como resultado. Na prática, essa operação se dá da seguinte forma:

```
plano_k = ((img >> k) & 1) * 255
```

Com img sendo o array numpy da imagem original da qual se quer extrair os planos e k sendo o número do plano desejado. Então, se desejamos obter o plano de bits de ordem 0, nenhum *right shift* será executado, pois $k = 0$, e a máscara, extraírá o bit menos significativo. Se queremos obter o plano de bits de ordem 7, então $k = 7$ e *right shift* deslocará 7 bits para a direita, fazendo com que o bit mais significativo original, agora seja o menos significativo, e este será extraído pela máscara. Como img se trata se um array numpy, as operações serão executadas em todos os elementos, garantindo que $plano_k$ seja nossa imagem desejada.

Ao executar a lógica proposta acima para valores de k no intervalo $[0, 7]$, obteremos como resultado as 8 imagens a seguir (Figuras 28 à 35), cada uma representando um plano de bit de 0 à 7.



Figura 28: Plano de Figura 29: Plano de Figura 30: Plano de Figura 31: Plano de
bit 7 bit 6 bit 5 bit 4

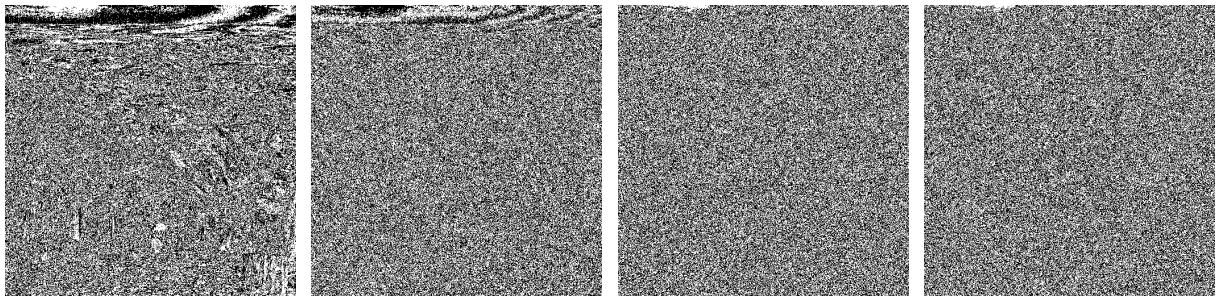


Figura 32: Plano de Figura 33: Plano de Figura 34: Plano de Figura 35: Plano de
bit 3 bit 2 bit 1 bit 0

Ao observar os resultados, é fácil notar que conforme os planos de bit vão chegando perto de 0, ou seja, conforme vamos extraiendo os bits menos significativos dos valores, as imagens resultantes contém cada vez menos conteúdo identificável. Com o plano de bit 7 contendo uma silhueta/blocagem bem nítida da imagem, e o plano de bit 0 sendo, basicamente, ruído.

A observação desse padrão pode levantar muitos questionamentos, entre eles, a dúvida de se esse padrão se repete da exata mesma forma em outros tipos de imagem. A imagem *city.png* usada se trata de uma foto, a seguir, testaremos essa extração com uma imagem feita completamente no computador (um desenho digital, visto na Figura 36) e com uma imagem feita por IA generativa (Figura 37).



Figura 36: *iwakura_lain.png*



Figura 37: *tung_sahur_greyscale.png*

Realizando o mesmo processo nas imagens acima, temos os resultados seguintes, sendo as Figuras 38 à 45 referentes à *iwakura_lain.png* e as Figuras 46 à 53 referentes à *tung_sahur.png*



Figura 38: Plano de bit 7



Figura 39: Plano de bit 6



Figura 40: Plano de bit 5



Figura 41: Plano de bit 4



Figura 42: Plano de bit 3



Figura 43: Plano de bit 2

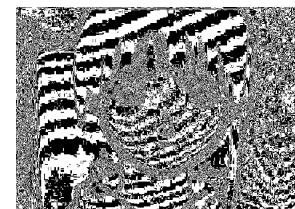


Figura 44: Plano de bit 1

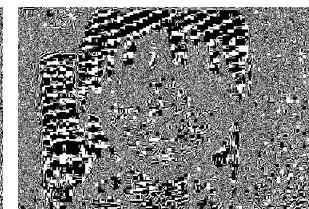


Figura 45: Plano de bit 0

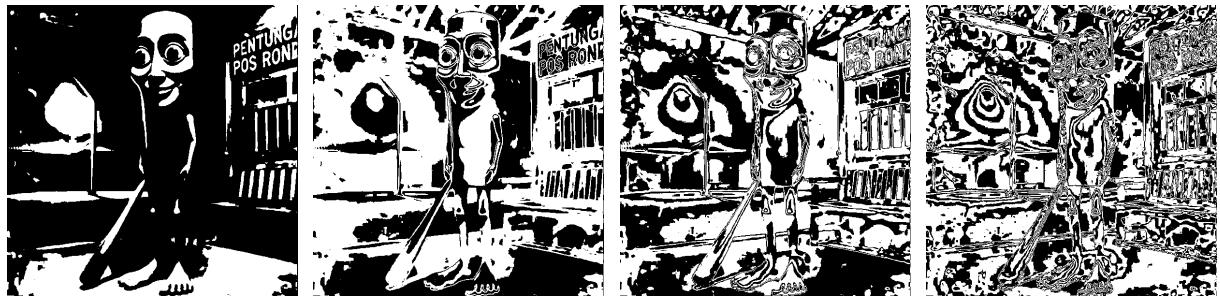


Figura 46: Plano de bit 7 Figura 47: Plano de bit 6 Figura 48: Plano de bit 5 Figura 49: Plano de bit 4

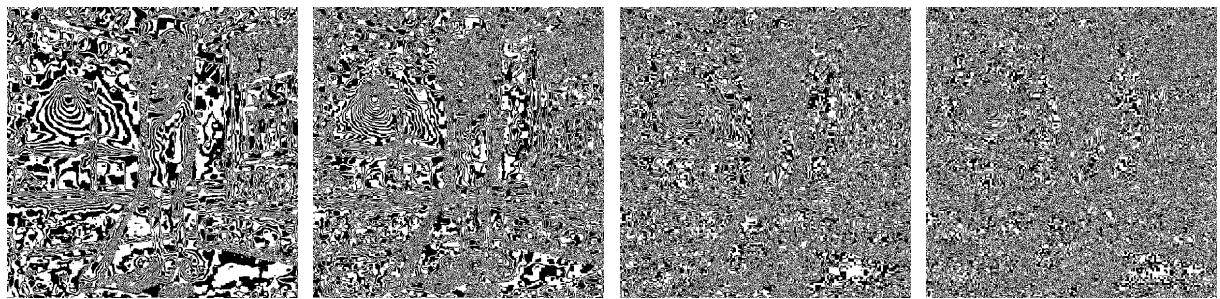


Figura 50: Plano de bit 3 Figura 51: Plano de bit 2 Figura 52: Plano de bit 1 Figura 53: Plano de bit 0

Como pode ser observado, a tendência de perder conteúdo reconhecível conforme exploramos planos de bits menos significativos se mantém para vários tipos de imagem. No caso da imagem *iwakura_lain.png*, também é possível notar que a perda de reconhecimento foi menor em relação às outras imagens, visto que é possível reconhecer partes da imagem original ainda na extração do plano de bit 0. Um dos motivos que pode causar esse efeito é o fato de a imagem ser mais simples em suas cores e formas.

3.7 Combinação de Imagens

O objetivo do sétimo item proposto foi combinar duas imagens monocromáticas de mesmo tamanho. As duas imagens escolhidas para serem combinadas foram *baboon_monocromatica.png* (Figura 54) e *house.png* (Figura 55)

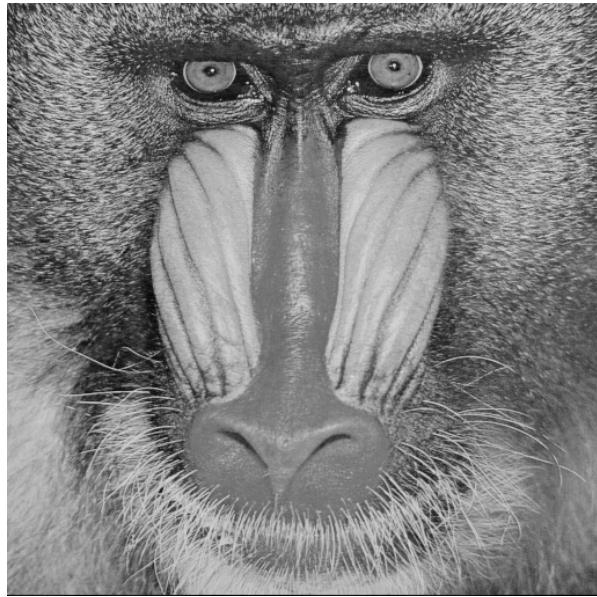


Figura 54: baboon_monocromatica.png



Figura 55: house.png

A combinação foi realizada calculando o nível dos pixels da imagem-resultado através de uma média ponderada dos níveis de cinza das imagens-entrada. Supondo que queremos combinar duas imagens A e B em uma única imagem C, a relação do valor de cada pixel de A, B e C se dará por $C = A * x + B * (1 - x)$, onde o valor de x será variado, fazendo com que obtenhamos múltiplas imagens diferentes como resultado, a depender do valor de x escolhido.

Os valores de x escolhidos foram 0.2, 0.5 e 0.8. Para melhor compreensão dos valores presentes em cada imagem, trataremos *baboon_monocromatica.png* como a imagem A e *house.png* como a imagem B. Os três resultados obtidos usando os valores de x escolhidos podem ser vistos nas Figuras 56, 57 e 58 abaixo.

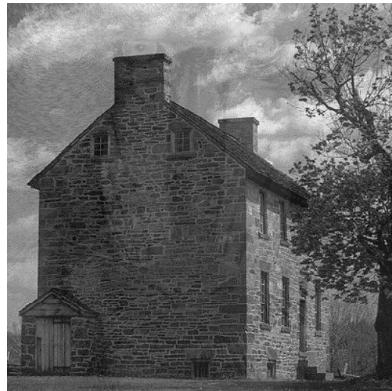


Figura 56: 0.2*A + 0.8*B



Figura 57: 0.5*A + 0.5*B

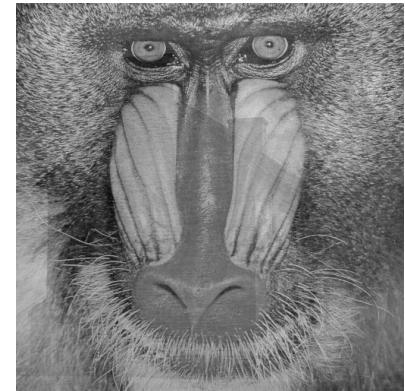


Figura 58: 0.8*A + 0.2*B

3.8 Transformação de Intensidade

O objetivo do oitavo item proposto foi, dada uma imagem monocromática (Figura 120930123), realizar uma série de transformações vistas a seguir. A imagem escolhida para realizar as transformações foi *buckethead.png* (Figura 59)



Figura 59: buckethead.png

3.8.1 Negativo da Imagem

Para obter o negativo, é necessário transformar os valores dos pixels da imagem de uma forma que um pixel 0 se transforme em 255, um pixel 1 se transforme em um 254, e assim por diante. Para realizar essa transformação foi utilizada a operação:

$$I = 255 - \text{img}$$

Onde I é a imagem resultante e img é a imagem original. O resultado dessa operação pode ser visto na Figura 60. É possível notar que, assim como o nome sugere, a negativa é o oposto, em termos de intensidade, da imagem original. Valores antes escuros, se tornam claros na mesma medida e vice-versa.



Figura 60: negativo da imagem

3.8.2 Transformação de intervalo

O intuito da transformação de intervalo é converter o intervalo de intensidades da imagem para [100, 200]. Para realizar essa operação, primeiramente foi realizada a restrição os valores ao intervalo [100, 200] com a função `clip()`, ou seja, os valores acima de 200, se tornaram igual a 200, e os valores abaixo de 100 se tornaram igual a 100. Após isso, a imagem foi normalizada, e o resultado pode ser visto na Figura 61. É possível notar um escurecimento na imagem, causado pela restrição (e depois normalização) dos valores em um intervalo de intensidade mais restrito.

Caso a imagem não seja normalizada após a restrição, o resultado é a imagem mostrada na Figura 62, uma versão da imagem original que sofreu uma espécie de desaturação, um resultado que faz sentido, uma vez que, sem a normalização, estamos apenas restringindo o intervalo para [100, 200], efetivamente garantindo que nenhum pixel tenha valor fora do intervalo, ou seja, os pixels nas faixas de intensidade mais claras e mais escuras são alterados para um intervalo mais acinzentado, tornando a imagem mais cinza.



Figura 61: imagem transformada



Figura 62: imagem transformada sem normalização

3.8.3 Inversão de Linhas

O intuito da operação de inversão de linhas é inverter os valores dos pixels das linhas pares da imagem. Para realizar essa operação, basta utilizar slices nos arrays numpy da imagem para substituir apenas as linhas pares da imagem original por suas versões invertidas. Na prática, isso foi feito da seguinte forma:

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

Onde `img_inv` é uma cópia da imagem original (isso é feito para evitar que as mudanças feitas aqui afetem a imagem original diretamente, já que usaremos ela mais tarde para outras operações e slices afetam a imagem diretamente, sem criar cópias). A segunda linha do código mostrado faz exatamente o proposto para a transformação, substitui apenas as linhas pares da imagem (`img_inv[::2]`) por linhas pares invertidas (`img_inv[::2, ::-1]`).

O resultado, que pode ser visto na Figura 63, é uma mistura/combinação da imagem original com uma versão horizontalmente invertida dela mesma. Observando a imagem resultante normalmente, existe a impressão de que a imagem foi duplicada, mas ao aplicarmos um zoom na imagem (Figura 99), é possível ver de forma mais nítida a separação das linhas, isto é, as linhas que foram invertidas e as que não foram, isso significa que a impressão inicial de que há uma simetria com eixo vertical na imagem é falsa, assim como a impressão de duplicidade, pois as linhas invertidas continuam sendo diferentes de suas vizinhas superiores e inferiores.



Figura 63: linhas pares invertidas



Figura 64: zoom na imagem resultante

3.8.4 Reflexão de Linhas

O intuito da reflexão de linhas é espelhar as linhas da metade superior da imagem na parte inferior da imagem. Essa operação foi realizada utilizando a mesma lógica e técnica de slicing do item anterior, porém substituindo a segunda metade (metade da metade inferior) pelas linhas da metade superior invertidas. Na prática, isso foi feito da seguinte forma:

```
img_refl[height:] = img_refl[:height] [::-1]
```

Onde *height* na verdade é o valor da metade da altura da imagem. O resultado pode ser visto na Figura 65.



Figura 65: reflexão de linhas

3.8.5 Espelhamento Vertical

O intuito do espelhamento vertical é, como o nome sugere, aplicar um espelhamento onde a imagem é verticalmente espelhada. Essa operação foi realizada simplesmente invertendo a ordem das linhas da matriz que representa a imagem, ou seja, a primeira linha passou a se tornar a última (e vice-versa), a segunda se torna a penúltima, e assim por diante. Na prática, essa transformação foi realizada com a seguinte operação de slice:

```
img_esp = img_esp[::-1]
```

O resultado é uma imagem verticalmente invertida (Figura 66).



Figura 66: espelhamento vertical

3.9 Quantização de Imagens

O objetivo do nono item proposto consiste em quantizar uma imagem, ou seja, alterar o número de bits necessários para armazenar a imagem, efetivamente limitando também o números de cores que podem estar presentes na imagem. A imagem escolhida para realizar as quantizações é a *seagull.png* (Figura 67), ela será apresentada em quantizações de 2, 4, 8, 16, 32, 64 e 256 níveis.



Figura 67: *seagull.png*

Para realizar as quantizações da imagem, primeiramente, a imagem foi normalizada para um intervalo que compreendesse o número de cores desejadas, por exemplo, para uma quantização em 2 níveis, a imagem foi normalizada para o intervalo [0, 1], pois só haverão 2 cores diferentes (nesse caso, branco e preto). Após isso, a função *np.uint8()* foi utilizada para transformar todos os números do array da imagem em inteiros, isso significa que, para o caso da quantização de 2 níveis, só haveriam 0s e 1s. Depois disso, o intervalo é normalizado novamente para [0, 255]. Na prática, isso é feito da seguinte forma:

```
niveis_2 = cv2.normalize(img, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX)
niveis_2 = cv2.normalize(np.uint8(niveis_2), None, alpha=0, beta=255,
                      norm_type=cv2.NORM_MINMAX)
```

Dessa forma, é possível discretizar os valores dos pixels nas quantidades desejadas. Os resultados das quantizações podem ser vistos a seguir nas Figuras 68 à 74. Como é possível ver nos resultados, nas quantizações de 2 e de 4 níveis, a diferença é notável, sendo possível inclusive contar as 2 cores distintas na Figura 68 e 4 cores distintas na Figura 69, porém, da quantização de 8 níveis em diante, as diferenças ficam muito mais sutis. A quantização de 256 níveis é exatamente igual à imagem original, pois possuem 256 cores.



Figura 68: 2 níveis



Figura 69: 4 níveis

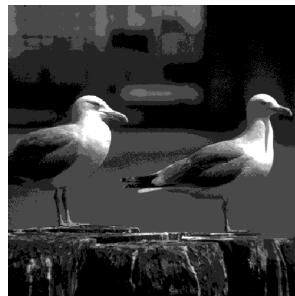


Figura 70: 8 níveis

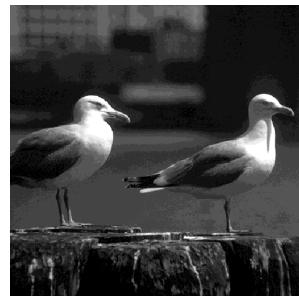


Figura 71: 16 níveis



Figura 72: 32 níveis

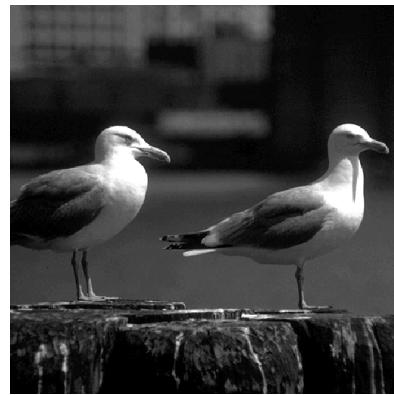


Figura 73: 64 níveis



Figura 74: 256 níveis

3.10 Filtragem de Imagens

O objetivo do décimo item proposto é aplicar diferentes filtragens a uma imagem digital. Para isso, os seguintes filtros foram fornecidos:

$$\begin{array}{ll}
h1 = \times \frac{1}{1} & \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix} \\
h3 = \times \frac{1}{1} & \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\
h5 = \times \frac{1}{1} & \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix} \\
h7 = \times \frac{1}{1} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
h9 = \times \frac{1}{9} & \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} \\
h11 = \times \frac{1}{1} & \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \\
h2 = \times \frac{1}{256} & \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 16 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \\
h4 = \times \frac{1}{1} & \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \\
h6 = \times \frac{1}{9} & \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \\
h8 = \times \frac{1}{1} & \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \\
h10 = \times \frac{1}{8} & \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}
\end{array}$$

Para aplicar cada filtro à imagem, foi realizada uma operação de convolução utilizando a função *filter2D()* da biblioteca *OpenCV*. Na prática, todas as aplicações se deram da seguinte forma:

```
img_filtro = cv2.filter2D(img, -1, filtro)
```

Onde *img_filtro* é a imagem resultante após aplicação do filtro, *img* é a imagem original, *filtro* é o filtro sendo aplicado, e, por fim, *-1* é um parâmetro da função referente à profundidade da imagem, na prática, o parâmetro passado faz com que a imagem resultante possua a mesma profundidade da imagem de entrada. Para tratamento de bordas, a função *filter2D()* possui um parâmetro '*borderType*', onde o valor padrão (que é o nosso caso) faz com que os valores de filtro que fiquem para fora da imagem sejam "refletidos", sendo utilizados dentro da imagem. A imagem selecionada para aplicar as filtragens foi *city.png* (Figura 76).



Figura 75: city.png

3.10.1 Filtro h1

O resultado da aplicação do filtro h1 da imagem pode ser visto abaixo na Figura 77, e, ao lado dela, a imagem original (Figura 76).



Figura 76: imagem original



Figura 77: imagem com filtro h1 aplicado

É possível observar que o filtro aplicado realçou as partes da imagem onde houve algum tipo de mudança expressiva nos valores de cinza, isto é, realçou linhas onde havia contraste entre duas partes. Isso pode ser observado principalmente nos telhados e janelas das casas na imagem. Como esse filtro realça as bordas e suaviza as partes mais homogêneas da imagem, pode-se dizer que se trata de um filtro passa-alta, justamente porque ele filtra as frequências mais baixas.

3.10.2 Filtro h2

O resultado da aplicação do filtro h2 da imagem pode ser visto abaixo na Figura 79, e, ao lado dela, a imagem original.



Figura 78: imagem original



Figura 79: imagem com filtro h2 aplicado

É possível observar que o filtro aplicado desfocou levemente a imagem original, mas ainda manteve sua forma geral, isto é, as formas e níveis de cinza em cada região não foram descaracterizadas. Com isso, é possível concluir que, assim como o filtro Gaussiano, esse filtro é um filtro passa-baixa, ou seja, ele filtra as frequências altas. Isso pode ser inferido pois, enquanto o filtro h1 realçava as bordas/transições, esse filtro faz o oposto, tornando as transições de cores menos definidas.

3.10.3 Filtro h3

O resultado da aplicação do filtro h3 da imagem pode ser visto abaixo na Figura 81, e, ao lado dela, a imagem original.



Figura 80: imagem original

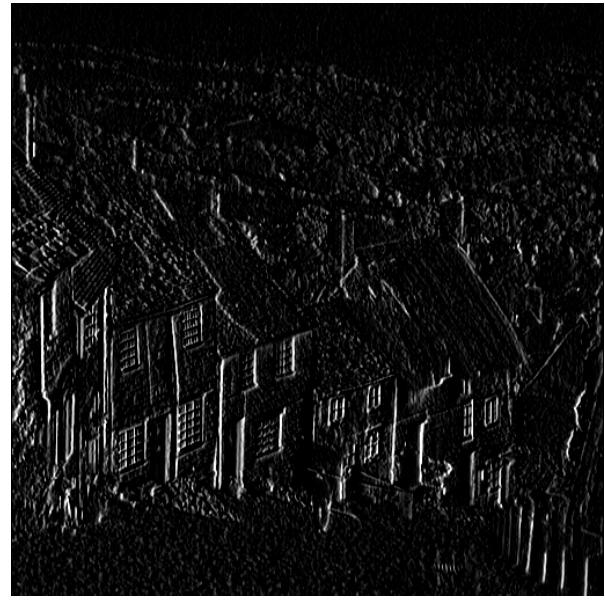


Figura 81: imagem com filtro h3 aplicado

É possível observar que o filtro aplicado, assim como o filtro h1, realçou linhas onde havia contraste de intensidade, porém, diferentemente do primeiro filtro, este realçou quase exclusivamente linhas verticais da imagem. Ao observar a matriz deste filtro, o efeito faz sentido, já que a matriz multiplica por zero os pixels adjacentes superiores e inferiores (considerando adjacência em vizinhança-4), enquanto os laterais tem multiplicações não-nulas.

3.10.4 Filtro h4

O resultado da aplicação do filtro h4 da imagem pode ser visto abaixo na Figura 83, e, ao lado dela, a imagem original.



Figura 82: imagem original

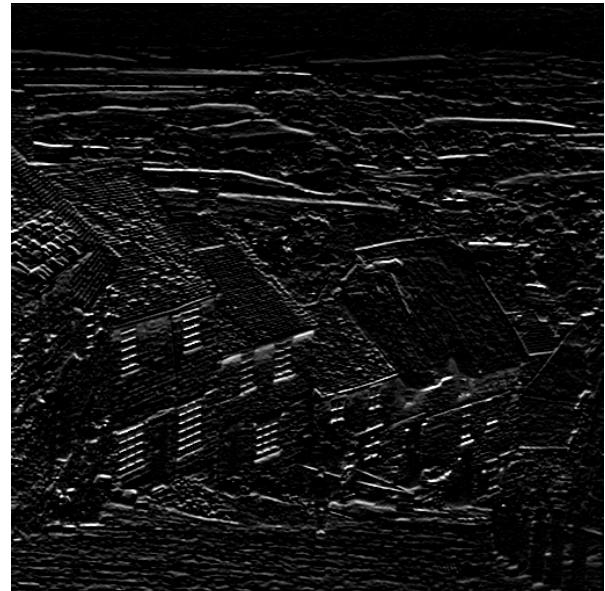


Figura 83: imagem com filtro h4 aplicado

É possível observar que o filtro aplicado possui o mesmo efeito do filtro anterior, mas em vez de realçar linhas verticais, ele ralçou linhas horizontais da imagem. Mais uma vez, o efeito pode ser inferido diretamente da matriz, já que esta multiplica por 0 os vizinhos adjacentes (considerando adjacência em vizinhança-4) da direita e esquerda.

3.10.5 Filtro h5

O resultado da aplicação do filtro h5 da imagem pode ser visto abaixo na Figura 85, e, ao lado dela, a imagem original.



Figura 84: imagem original



Figura 85: imagem com filtro h5 aplicado

É possível observar que o filtro aplicado possui um efeito muito semelhante ao filtro h1, sendo também um filtro passa-alta.

3.10.6 Filtro h6

O resultado da aplicação do filtro h6 da imagem pode ser visto abaixo na Figura 87, e, ao lado dela, a imagem original.



Figura 86: imagem original



Figura 87: imagem com filtro h6 aplicado

É possível observar que o filtro aplicado possui um efeito muito semelhante ao filtro h2, sendo também um filtro passa-baixa. Este filtro difere também ao h1 em seu tamanho, visto que é uma matrix 3×3 , enquanto o filtro h1 era uma matriz 5×5 , fazendo com que operações com este filtro sejam mais baratas computacionalmente.

3.10.7 Filtro h7

O resultado da aplicação do filtro h7 da imagem pode ser visto abaixo na Figura 89, e, ao lado dela, a imagem original.



Figura 88: imagem original

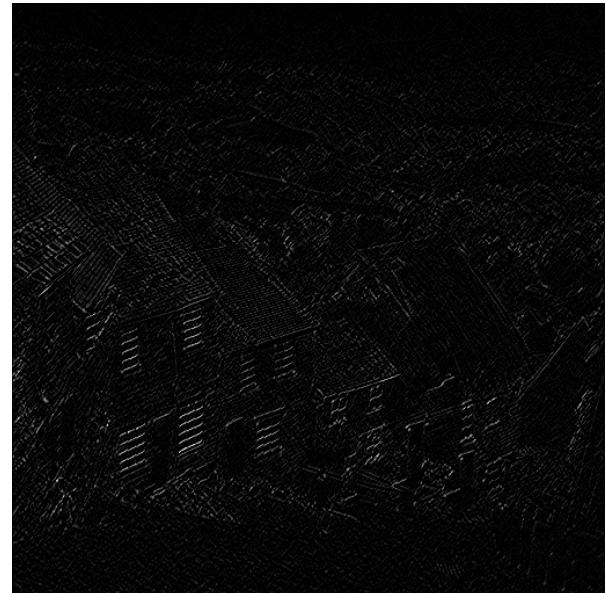


Figura 89: imagem com filtro h7 aplicado

É possível observar que o filtro aplicado possui um efeito muito semelhante ao filtro h1, sendo também um filtro passa-alta que realça bordas, entretanto, ao analisar a matriz

desse filtro, podemos inferir que sua função, em vez de realçar todas as bordas (como $h1$) ou apenas verticais/horizontais (como $h3$ e $h4$), é realçar diagonais

3.10.8 Filtro $h8$

O resultado da aplicação do filtro $h8$ da imagem pode ser visto abaixo na Figura 91, e, ao lado dela, a imagem original.



Figura 90: imagem original



Figura 91: imagem com filtro $h8$ aplicado

É possível observar que o filtro aplicado possui um efeito muito semelhante ao filtro anterior, sendo também um filtro passa-alta que realça bordas diagonais. Este porém, realça diagonais de direção oposta ao filtro anterior.

3.10.9 Filtro $h9$

O resultado da aplicação do filtro $h9$ da imagem pode ser visto abaixo na Figura 93, e, ao lado dela, a imagem original.



Figura 92: imagem original



Figura 93: imagem com filtro h9 aplicado

É possível observar que o filtro aplicado desfocou a imagem assim como o filtro h2, porém, o efeito de desfoque que este filtro aplicou também adicionou uma sensação de movimento à imagem, como se a foto tivesse sido tirada em movimento.

3.10.10 Filtro h10

O resultado da aplicação do filtro h10 da imagem pode ser visto abaixo na Figura 95, e, ao lado dela, a imagem original.



Figura 94: imagem original



Figura 95: imagem com filtro h10 aplicado

É possível observar que o filtro aplicado, assim como o filtro h1, realçou as bordas da imagem, porém, assim como o filtro h2, ele também não descaracterizou a imagem, realçando as bordas de deixando a imagem mais nítida e legível, ou seja, o filtro h10 possui características tanto do filtro passa-alta h1, quanto do passa-baixa h2.

3.10.11 Filtro h11

O resultado da aplicação do filtro h11 da imagem pode ser visto abaixo na Figura 97, e, ao lado dela, a imagem original.



Figura 96: imagem original

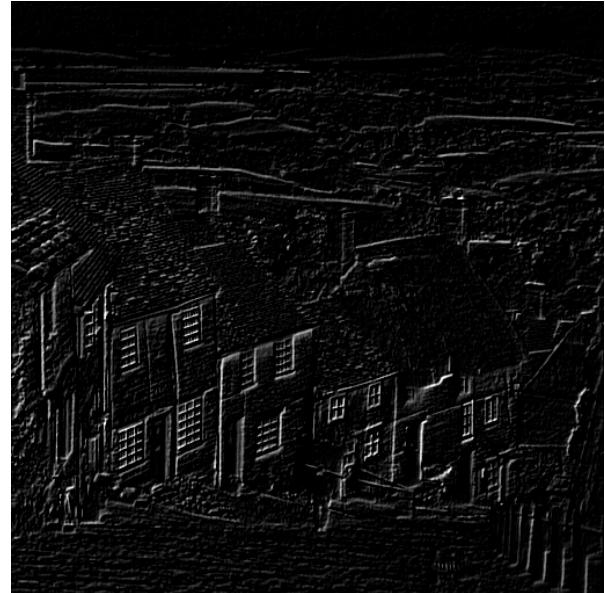


Figura 97: imagem com filtro h11 aplicado

É possível observar que o filtro aplicado, assim como os filtros h7 e h8, realça bordas diagonais.

3.10.12 Combinação h3 e h4

A combinação de h3 e h4 foi realizada elevando as imagens transformadas pelos filtros h3 e h4 ao quadrado, somando-as e calculando a raiz quadrada da soma. Também é interessante ressaltar que, antes de realizar a operação de exponenciação, foi necessário converter as imagens resultado de h3 e h4 para *uint16* a fim de evitar problemas com *overflow*, que faziam com que o resultado fosse uma imagem inteiramente preta. Na prática, a operação foi feita da seguinte forma:

```
img_comb = np.sqrt(img_h3_c.astype(np.uint16)**2 + img_h4_c.astype(np.uint16)**2)
```

O resultado da combinação de h3 e h4 pode ser visto abaixo na Figura 97, e, ao lado dela, a imagem original.



Figura 98: imagem original



Figura 99: combinação h3 e h4

É possível observar que o efeito obtido na imagem é a combinação dos resultados da aplicação do filtro h3 e do h4. O filtro h3 realçava bordas verticais e o h4 horizontais, então, nosso resultado é uma imagem com bordas verticais e horizontais realçadas.

4 Conclusão

Após realizar diversas operações básicas essenciais para o processamento de imagens, foi possível analisar e compreender diversos conceitos e aspectos de extrema importância para a área de processamento de imagens digitais, não só conceitos teóricos como também conceitos práticos relacionados à implementação de operações e transformações utilizando *Python* e bibliotecas essenciais como *OpenCV*, *Pillow* e *NumPy*.