

Relatório Trabalho 1

MC920 - Introdução ao Processamento de Imagem Digital

Victor Costa Dominguite - RA: 245003

Abril 2023

1 Introdução

Neste trabalho foram feitas diversas operações básicas de processamento de imagens digitais. Ao longo deste relatório serão discutidos os métodos empregados e os resultados obtidos em cada etapa do processo.

2 Execução dos programas

Foram escritos oito programas (arquivos *.py), cada um correspondendo a um item do trabalho. Para executar cada um desses, basta rodar seu *script* em python (isto é, em um terminal, executar o comando `python3 nome_do_arquivo.py`). Além disso, vale notar que as imagens utilizadas (“baboon.png”, “butterfly.png”, “city.png” e “araras_RGB.png”) devem estar em um diretório chamado “images”, o qual, por sua vez, deve estar no mesmo diretório do programa que está sendo executado.

É necessário também que os pacotes `imageio`, `numpy` e `matplotlib` estejam instalados, pois os programas os utilizam para realizar, respectivamente, a leitura, manipulação e apresentação das imagens.

3 Análise das soluções e resultados obtidos

3.1 Mosaico

Para essa primeira tarefa do trabalho, a leitura da imagem foi feita com o uso da função `imread` do `imageio`, a qual devolve uma matriz bidimensional do tipo `numpy.array` contendo os valores de cada pixel na imagem.

A montagem do mosaico foi feita praticamente com o uso de uma função do `numpy`: `np.block()`, a qual concatena uma sequência de *arrays* fornecida. Assim, bastou fornecer a essa função os blocos da imagem original na ordem desejada, mais especificamente, na ordem em que foi especificada no enunciado. Os recortes foram feitos manualmente, realizando seções nos índices das linhas e colunas do *array* que representa a imagem original.

O resultado e a imagem original estão apresentados abaixo:

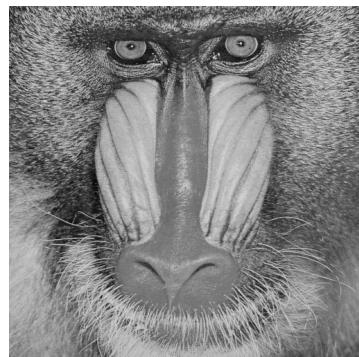


Figura 1: imagem original *baboon.png*

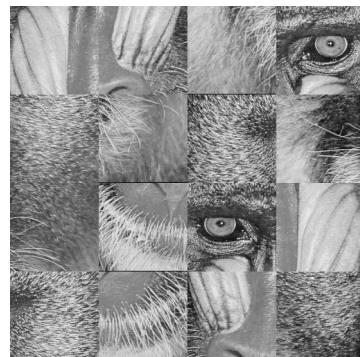


Figura 2: mosaico

Com o método adotado para a resolução dessa tarefa, só é possível rearranjar imagens em 16 blocos, formando um mosaico de 4 blocos por 4 blocos, sempre na mesma ordem de rearranjo apresentado acima. Porém, como vantagem, a imagem pode ter quaisquer dimensões, uma vez que as medidas dos blocos foram obtidas dividindo por 4 as dimensões devolvidas pelo método *shape* do *array*, em vez de simplesmente assumir um tamanho constante para a imagem (por exemplo de 512 x 512). Porém há um risco de se perder fileiras de pixels caso o tamanho da imagem original não for divisível por 4, uma vez que as dimensões de cada um dos 16 blocos são obtidas pela divisão inteira da altura e do comprimento da imagem original por 4.

3.2 Combinação de Imagens

Assim como na etapa anterior, a leitura da imagem foi feita novamente com o uso da função `imageio.v3.imread`.

As operações para combinação das duas imagens foi feita de maneira consideravelmente direta e sem grandes dificuldades, apenas multiplicando os *arrays* que representam as imagens A e B por seus respectivos fatores, em seguida somando as matrizes termo a termo e guardando a matriz resultante em uma variável `result`, também do tipo `np.array`.

Vale ressaltar que ao multiplicar as matrizes pelos fatores que eram números decimais (de ponto flutuante), os resultados poderiam não ser inteiros. Para lidar com isso, foi utilizada a função `np.rint` que recebe um *array* e arredonda os valores ali contidos para o inteiro mais próximo. Houve a preferência pelo arredondamento em vez de outros métodos, como o truncamento, pois com o arredondamento o resultado inteiro final se aproxima mais do real valor (o qual seria decimal) que aquele pixel deveria ter, mantendo, com isso, uma maior fidelidade com o resultado esperado.

Enfim, os resultados obtidos para as diferentes combinações das imagens, assim como as imagens originais utilizadas nas combinações estão apresentadas abaixo.

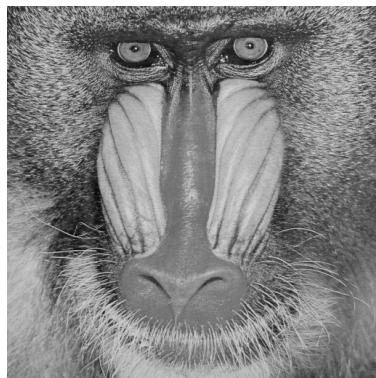


Figura 3: imagem A (*baboon.png*) Figura 4: imagem B (*butterfly.png*)



Figura 5: $0.2 \cdot A + 0.8 \cdot B$

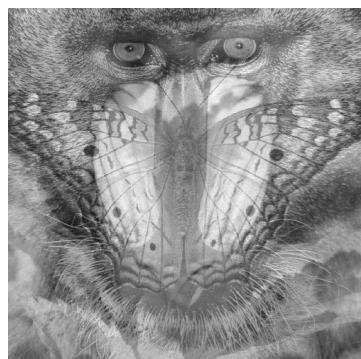


Figura 6: $0.5 \cdot A + 0.5 \cdot B$

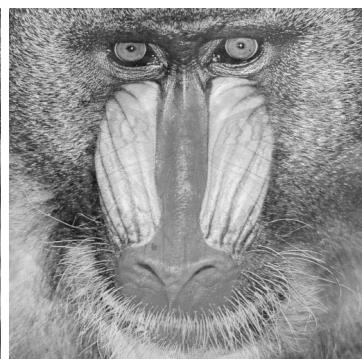


Figura 7: $0.8 \cdot A + 0.2 \cdot B$

3.3 Transformação de Intensidade

Neste item, foi utilizada a imagem *city.png* para realizar as manipulações desejadas. A imagem original está apresentada a seguir:



Figura 8: imagem original *city.png*

A primeira transformação aplicada foi a obtenção do negativo da imagem. Para isso, foi feita uma simples operação de subtração, onde cada pixel da imagem resultante negativa teria o valor $255 - \text{valor original}$. O resultado pode ser observado abaixo.

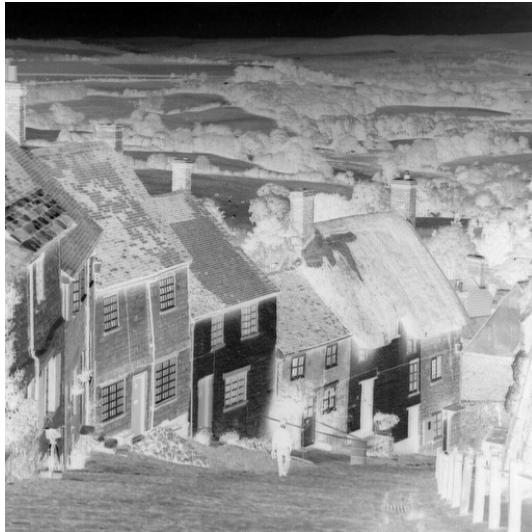


Figura 9: negativo da imagem

A segunda transformação consistia em converter o intervalo de intensidades da imagem de $[0, 255]$ para $[100, 200]$. Para isso, foi utilizada uma conversão considerando a proporcionalidade entre os pixels resultantes e os originais. Ou seja, foi aplicado o procedimento mais conhecido como “regra de três”, convertendo primeiramente os pixels de $[0, 255]$ para $[0, 100]$, usando a equação

$$\text{pixel resultante} = \frac{100}{255} \times \text{pixel original}$$

e, em seguida, somando 100 a cada pixel, levando-os para o intervalo desejado: $[100, 200]$. Para que a imagem foi mostrada corretamente pela função `imshow` do `matplotlib`, foi necessário adicionar dois

parâmetros à função: `vmin=0` e `vmax=255`, pois, caso contrário, a biblioteca faz um ajuste automático de contraste da imagem e ela é mostrada como a original. Assim, embora em memória os pixels da imagem resultante possuíssem valores entre 100 e 200, a imagem era ajustada para ser mostrada com maior contraste e, dessa forma, aparentava não ter havido qualquer transformação. Mas, ao especificar os valores mínimos e máximos dos pixels com esses parâmetros adicionais, a imagem é mostrada corretamente, conforme desejado. O resultado encontra-se abaixo:



Figura 10: imagem com níveis de cinza em [100, 200]

A terceira transformação pedia que fossem invertidas (espelhadas) as linhas pares da imagem. Esse procedimento foi feito primeiramente copiando-se a imagem original para um *array* auxiliar, no qual de fato a inversão foi feita com a seguinte instrução:

```
par_invertido[0::2, : ] = par_invertido[0::2, ::-1]
```

Esse comando faz com que as linhas sejam percorridas de duas em duas, partindo da linha 0 e que a ordem dos pixels seja a inversa da original. O resultado pode ser observado a seguir:



Figura 11: imagem com linhas pares invertidas

Para a quarta transformação, as linhas da metade superior da imagem deveriam ser refletidas para a metade inferior. Isso pode ser efetuado com a seguinte linha de código:

```
refletida[refletida.shape[0]//2:, :] = refletida[refletida.shape[0]//2:0:-1, :]
```

Nessa linha, `refletida` é um *array* que contém uma cópia da imagem original. Assim, observa-se que a imagem está sendo alterada da metade das linhas em diante, copiando a porção superior, a qual é percorrida em ordem inversa do meio até a linha 0. O resultado é apresentado na Figura 12.

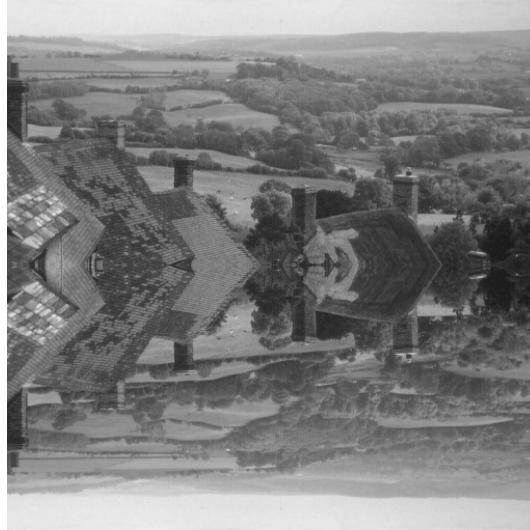


Figura 12: imagem com linhas refletidas

Por fim, a quinta e última transformação consistia em espelhar verticalmente a imagem. Para isso, foi suficiente percorrer as linhas em ordem contrária e copiá-las uma a uma para a imagem resultante. Novamente, o procedimento pôde ser realizado com uma linha de código:

```
espelhada[ :, :] = espelhada[ ::-1, : ]
```

A variável `espelhada` foi inicializada como uma cópia da imagem original. O resultado dessa operação pode ser observado na Figura 13.



Figura 13: imagem espelhada verticalmente

3.4 Imagens Coloridas

Para a realização das operações propostas nesta etapa, foi utilizada a imagem *araras_RGB.png*, apresentada a seguir na Figura 14.



Figura 14: imagem original *araras-RGB.png*

A leitura da imagem foi feita pela função `imread` do pacote `imageio`. Para imagens RGB, essa função retorna um *array* tridimensional em que a terceira dimensão representa cada uma das 3 cores, sendo a sendo que para uma imagem genérica, `imagem[i, j, 0]` representa as intensidades da cor vermelha para um pixel na posição (i, j) , `imagem[i, j, 1]` representa a cor verde e, por fim, `imagem[i, j, 2]` representa a cor azul.

No item (a), foi feita uma alteração em cada camada de cor para obter novos valores R' , G' e B' com base numa composição dos valores originais R , G e B de cada pixel. Analisando essa nova composição, observa-se que, para todo pixel R' assumirá o maior valor, seguido de G' e por fim B' . Com isso, foi obtida uma imagem com tom mais amarelado, o que era esperado, considerando que os tons vermelho e verde se sobressaem na composição dos novos pixels.

Para a construção do resultado em si, foi criado um *array* auxiliar nomeado `result` com as dimensões da imagem original, inicialmente vazio, com o uso da função `np.empty`, que recebe o *shape* desejado e devolve um *array* vazio com as dimensões especificadas. Em seguida, cada uma das camadas de cor foi alterada combinando os valores de R , G , B de cada pixel para obter R' , G' e B' . Para isso, foi suficiente multiplicar as camadas que representavam R , G e B na imagem original e somá-las.

Alguns cuidados também foram necessários para garantir que os pixels na imagem resultante as sumissem valores válidos. Primeiramente, foi utilizada a função `np.rint`, descrita na seção 3.2, a qual arredonda os valores do *array* para o inteiro mais próximo. Em seguida, foi utilizado o método `result.clip(0, 255)`, o qual recebe um valor mínimo e um máximo (nesse caso 0 e 255), garantindo que todos os valores no *array* estejam entre esses valores e, caso houver algum menor que o limite inferior, ele é reajustado para o mínimo e, analogamente, se houver algum valor acima do máximo, então ele será reajustado para o máximo. Por fim, há o uso do método `result.astype(np.uint8)`, o qual converte os valores para o tipo especificado (no caso `unsigned int` de 8 bits), evitando quaisquer erros que possam vir a ocorrer na apresentação da imagem final.

O resultado pode ser observado na Figura 15, apresentada a seguir.



Figura 15: imagem com cores transformadas

No item (b), a imagem foi convertida para escala de cinza, em que cada pixel foi obtido por uma média ponderada das intensidades de R, G e B da imagem original. O processo de montagem da imagem resultante foi muito semelhante ao item anterior, porém, em vez de calcular 3 camadas (uma para cada cor) foi necessário calcular apenas os valores de uma matriz bidimensional que representa a imagem em tons de cinza, a qual está apresentada na Figura 16.



Figura 16: imagem em tons de cinza

3.5 Ajuste de Brilho

Nesta etapa, o objetivo era aplicar a correção *gamma* para ajustar o brilho da imagem. Para isso, primeiramente, os valores de cada pixel foram divididos por 255, para que todos fossem levados para valores entre 0 e 1. Em seguida, aplicou-se a transformação, elevando cada pixel a $(1/\gamma)$. Por fim, a imagem foi multiplicada por 255 para levar os valores dos pixels novamente para o intervalo original [0,255]. Além disso, foram aplicadas as funções `rint` e `clip` do `numpy`, já mencionadas anteriormente, para arredondar os valores dos pixels para inteiros e garantir que nenhum esteja acima de 255 ou abaixo de 0. Isso foi feito com a seguinte linha de código, onde `img` representa a imagem original e `result` a alterada:

```
result = (np.rint(((img/255)**(1/gamma))*255)).clip(0,255)
```

Esse procedimento foi repetido 3 vezes, para os seguintes valores de *gamma*: $\gamma = 1.5$, $\gamma = 2.5$ e $\gamma = 3.5$. Os resultados, assim como a imagem original utilizada (*baboon.png*), podem ser observados a seguir.

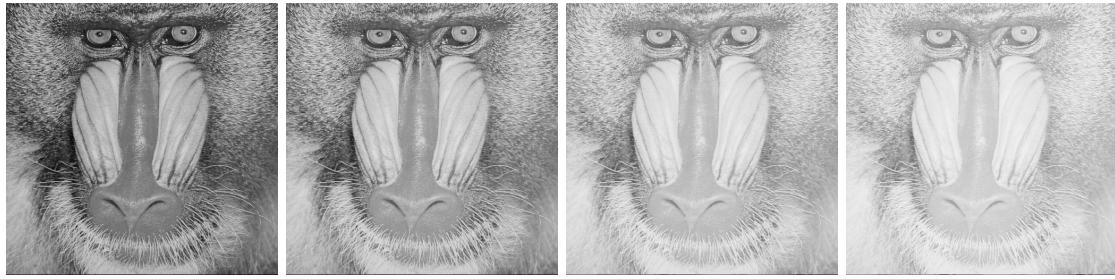


Figura 17: *baboon.png*

Figura 18: $\gamma = 1.5$

Figura 19: $\gamma = 2.5$

Figura 20: $\gamma = 3.5$

3.6 Quantização de Imagens

Neste item, pediu-se para que o número de níveis de cinza utilizados para representar a imagem fosse alterado para valores específicos. No caso, a imagem *baboon.png* foi alterada para que fosse representada com as seguintes quantidades de níveis de cinza: 256, 64, 32, 16, 8, 4 e 2.

Essa conversão dos tons de cinza foi feito com a seguinte formulação em código:

```
(np.rint((nvl-1)*(img/max_pixel))*(255/(nvl-1))).clip(0,255).astype(np.uint8)
```

Nessa fórmula, nvl representa a quantidade desejada de níveis de cinza, img é a imagem original e max_pixel é o maior valor dentre todos os pixels na imagem original. A ideia por trás dessa linha de código é de basicamente realizar uma transformação dos pixels para uma escala que contenha apenas a quantidade de níveis de cinza desejada, ou seja, no intervalo $[0, nvl - 1]$. Por exemplo, para $nvl = 4$, os pixels seriam todos convertidos para algum valor no conjunto de números inteiros $\{0, 1, 2, 3\}$, contando, assim, com 4 níveis de cinza. Essa conversão de valores dos pixels é feita seguindo a seguinte regra de proporcionalidade:

$$novo\ pixel = \frac{pixel\ original}{max\ pixel} (nvl - 1) \quad (1)$$

Após ser realizada essa transformação, é utilizada a função `np.rint` para garantir que os valores resultantes sejam aproximados para o inteiro mais próximo.

Em seguida, os pixels são convertidos de novo para a escala original $[0, 255]$, ao serem multiplicados por $255/(nvl - 1)$. Porém, como na escala anterior só existiam nvl valores de cinza, ao multiplicar cada pixel por uma constante, não é alterada a quantidade de níveis de cinza distintos presentes na imagem. Por exemplo, para o caso $nvl = 2$, temos que a primeira conversão fará com que todos os pixels sejam designados para os valores $\{0, 1\}$. A multiplicação por $255/(2 - 1)$ fará com que os dois valores de pixel presentes na imagem agora sejam $\{0, 255\}$.

Por fim, é utilizado o método `clip`, assegurando que os valores dos pixels não ultrapassem 255, e também o método `astype`, com o argumento `np.uint8`, comando esse que converte o tipo dos valores do *array* para inteiros sem sinal de 8 bits.

Os resultados estão apresentados abaixo.

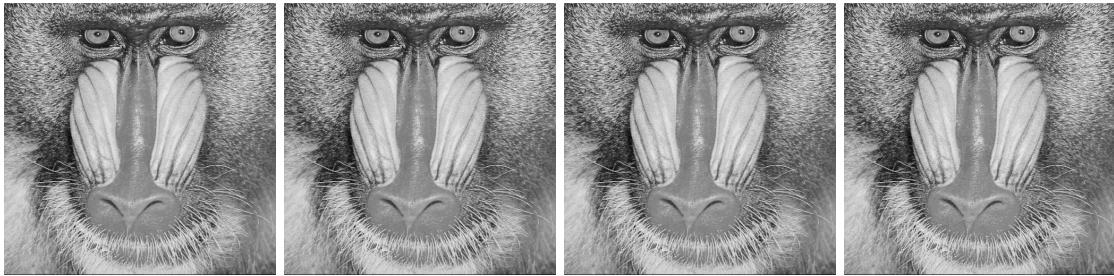


Figura 21: 256 níveis

Figura 22: 64 níveis

Figura 23: 32 níveis

Figura 24: 16 níveis

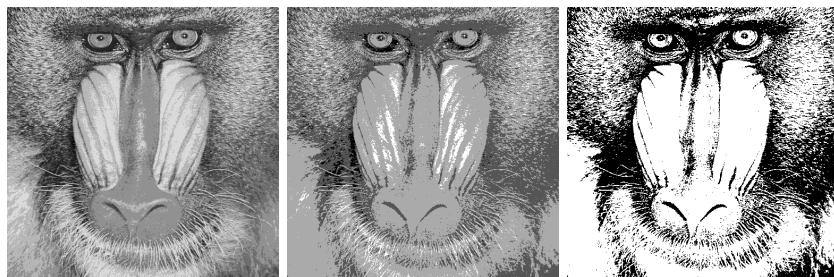


Figura 25: 8 níveis

Figura 26: 4 níveis

Figura 27: 2 níveis

Analisando os resultados, podemos observar que as 3, ou até mesmo as 4 primeiras figuras não apresentam diferenças muito notáveis para os olhos humanos, demonstrando assim que quantidades de níveis de cinza acima 32 ou 16 podem não ter um impacto significativo na qualidade da imagem para um observador humano comum.. Porém, para 8, 4 ou 2 níveis de cinza, já é possível observar, cada vez mais, uma perda considerável de qualidade de imagem, sendo que o discernimento de detalhes se torna cada vez mais difícil devido à perda de informações que uma quantidade limitada de níveis de cinza permite representar. Assim, dependendo do propósito da imagem, pode ser vantajoso representá-la com uma menor quantidade de níveis de cinza, o que diminuiria o número de bits necessários para representar cada pixel e, portanto, haveria uma economia de memória para armazenar a imagem.

3.7 Plano de Bits

Para essa etapa do trabalho, foram isolados os planos de bits de uma imagem em tons de cinza. Para realizar esse isolamento, foram realizadas operações bit a bit nos pixels da imagem com o uso de máscaras que isolavam o bit que representava o plano desejado-se observar. Mais especificamente, essa operação foi feita com a seguinte linha de código:

```
result[ : , : ] = img[ : , : ] & plano
```

As variáveis `result` e `img` representam, respectivamente, a imagem alterada com o plano de bits desejado e a imagem original, enquanto `plano` é um inteiro representando uma máscara para isolar o plano de bits desejado. Por exemplo, a máscara `0b00000001` isola o plano 0, enquanto `0b10000000` isola o plano 7.

Os resultados obtidos por essas operações podem ser observados a seguir.

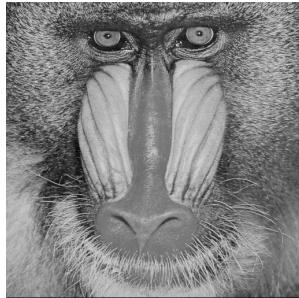


Figura 28: imagem original `baboon.png`

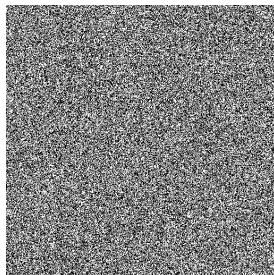


Figura 29: plano 0

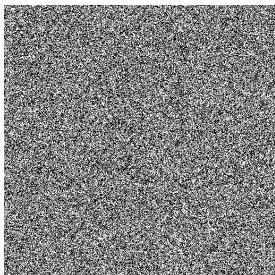


Figura 30: plano 1

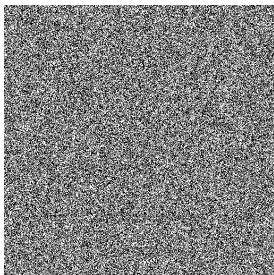


Figura 31: plano 2

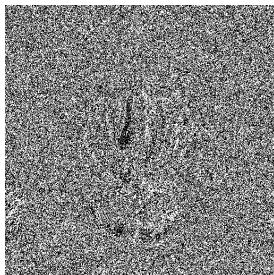


Figura 32: plano 3

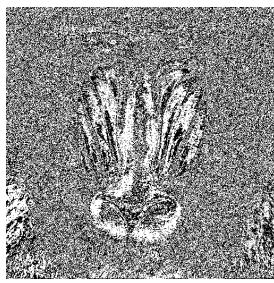


Figura 33: plano 4

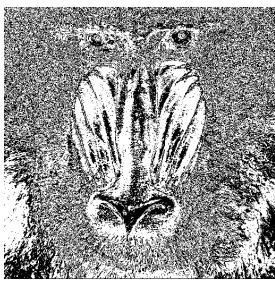


Figura 34: plano 5

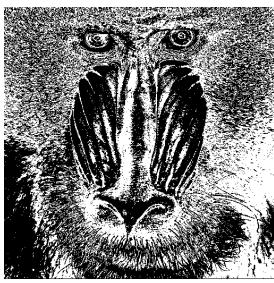


Figura 35: plano 6

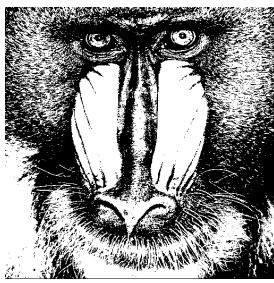


Figura 36: plano 7

Analisando as imagens geradas, pode-se concluir que os planos de bits menos significativos, em especial os planos 0 a 2, não guardam informações muito relevantes para o conteúdo da imagem, uma vez que apresentam conter majoritariamente ruído. Isso permite chegar a uma conclusão semelhante à do item 3.6 (quantização de planos), de que, dependendo da finalidade da imagem, uma menor quantidade de bits pode ser suficiente para formar uma imagem sem perdas notáveis de qualidade. O fato desses primeiros planos não apresentarem um conteúdo observável na imagem final permite

também um uso alternativo deles, por exemplo, contendo mensagens ou informações ocultas à primeira vista.

3.8 Filtragem de Imagens

Na oitava e última etapa desse trabalho, foi feita a filtragem de imagens aplicando o método da correlação. Isto é, o novo valor do pixel é calculado posicionando-se o centro da máscara em cima do pixel original e, em seguida, realizando a soma dos produtos dos valores da máscara com os pixels da imagem sobre o qual estão posicionados. Esse procedimento é repetido para todos os pixels.

Para contemplar os casos de borda, nos quais a máscara estaria cobrindo parcialmente uma área fora da imagem original, foi criada uma borda na imagem com a função `np.pad`, a qual recebe um *array*, uma largura para a borda e, opcionalmente, o modo de borda que se deseja (como um valor constante, o valor médio do *array*, reflexão das bordas originais, entre outros). No caso, foi utilizada a função com os seguintes parâmetros:

```
np.pad(img, filters[i].shape[0]//2, 'symmetric')
```

Assim, temos `img` como a imagem original, `filters[i]`, como uma máscara, sendo que ao se obter seu `shape[0]//2`, tem-se o valor desejado para a borda. O argumento `'symmetric'` indica que a borda deve ser feita refletindo os últimos pixels da imagem, como se esses estivessem sendo espelhados para a borda. Essa escolha foi feita visando manter uma maior consistência na aplicação do filtro nas bordas, obtendo-se valores semelhantes aos que estavam nessa região na imagem original.

A realização da correlação foi feita com o uso de *loops*, o que tornou a aplicação dos filtros um pouco mais lenta, em relação às operações com uso de vetorização. Porém, nesse caso, não apresentava haver uma alternativa muito mais eficiente.

Os resultados para a aplicação de cada filtro podem ser observados nas figuras a seguir.

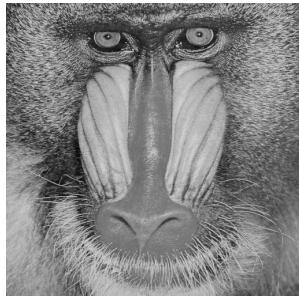


Figura 37: imagem original *baboon.png*

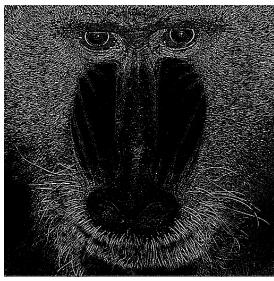


Figura 38: filtro h_1

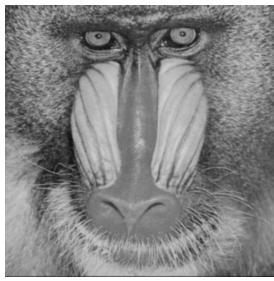


Figura 39: filtro h_2

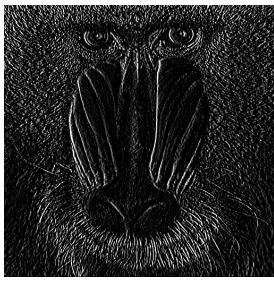


Figura 40: filtro h_3

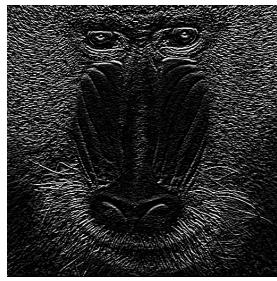
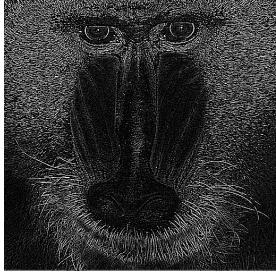
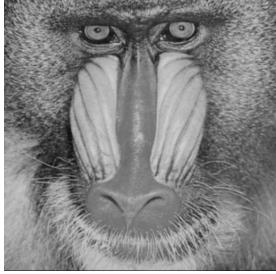
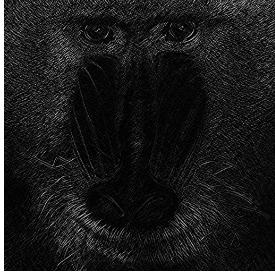
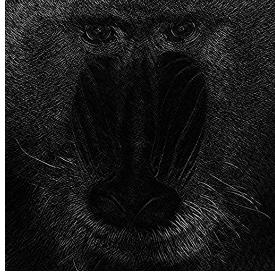
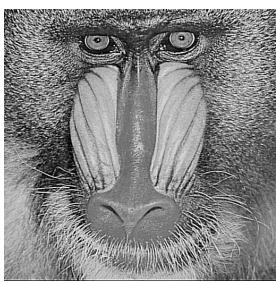
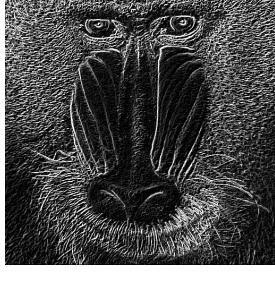


Figura 41: filtro h_4

Figura 42: filtro h_5 Figura 43: filtro h_6 Figura 44: filtro h_7 Figura 45: filtro h_8 Figura 46: filtro h_9 Figura 47: filtro h_{10} Figura 48: filtro h_{11} Figura 49: combinação de h_3 e h_4

Analisando as máscaras utilizadas para cada filtragem, assim como os resultados obtidos pela sua aplicação, pode-se dizer aproximadamente qual seria o efeito desejado para cada filtro.

O filtro h_1 é um filtro passa-alta que aparenta ressaltar as partes em que a variação de tons de cinza é maior. Dessa forma, a parte mais próxima ao nariz do animal, onde as cores são mais homogêneas, encontra-se mais escura e pouco ressaltada após a aplicação do filtro (Figura 38), enquanto os seus pelos, chamuscados, com muitos tons de cinza variando em uma pequena região estão consideravelmente realçados na imagem filtrada.

Já o filtro h_2 corresponde a um filtro gaussiano, com objetivo de suavizar a imagem. Assim, é possível observar uma certa homogeneização nas cores de certas áreas, especialmente nos pelos aos arredores dos olhos (Figura 39), onde as cores eram muito variadas na imagem original, além das linhas de contorno se tornarem mais suaves. Nesse caso, o desvio padrão σ do filtro não é muito alto, consequentemente, o grau de suavização pode não ter sido tão perceptível.

O filtro h_3 aparenta realçar a borda direita de linhas verticais, uma vez que possui valores positivos na coluna mais à direita e valores negativos à esquerda. Esse efeito pode ser mais claramente observado na imagem filtrada (Figura 40), especialmente na região próxima ao nariz do animal, onde as linhas verticais, especialmente aquelas nas bordas direitas, estão bem demarcadas.

O filtro h_4 realiza um papel semelhante ao h_3 , porém para as bordas inferiores das linhas horizontais. Na Figura 41, é possível reparar que as linhas horizontais estão mais salientes, estando esse efeito mais notável na região próxima aos olhos e também nos pelos.

O filtro h_5 é do tipo passa-alta e tem por característica realçar os detalhes da imagem, as bordas e os contornos. Por exemplo, na Figura 42, vê-se que o contorno dos olhos e outras regiões está destacado.

O filtro h_6 é passa-baixa e realiza a filtragem pela média dos valores de 9 pixels cobertos pela máscara. Essa operação faz com que haja uma suavização da imagem (Figura 43), semelhante ao filtro h_2 , porém com essa técnica ligeiramente diferente.

Os filtros h_7 e h_8 atuam de maneira muito semelhante, o primeiro tendendo a ressaltar linhas diagonais de baixo para cima, da esquerda para a direita, enquanto o segundo tende a ressaltar as linhas diagonais de cima para baixo, da esquerda para a direita, uma vez que nas máscaras há um peso positivo para essas diagonais e negativo para os demais pixels. Esse efeito pode ser visto muito mais claramente ao se ampliar as imagens filtradas (Figuras 44 e 45), sendo que tais imagens aparentam estar sendo compostas por diversos pequenos “riscos” nessas direções diagonais.

O filtro h_9 aparenta realizar um borramento da imagem, uma vez que faz uma média entre 9 pixels na diagonal secundária da máscara. Na Figura 46 é possível observar justamente esse efeito de

borramento da imagem original no sentido da diagonal secundária.

O filtro h_{10} aparenta ter ressaltado os detalhes da imagem, acentuando as diferenças de tons de cinza de regiões próximas onde essa diferença não era tão contrastante, conforme pode ser observado na Figura 47.

O filtro h_{11} tem pesos positivos no canto inferior direito, zerado na diagonal secundária e negativos no canto superior esquerdo. Assim, espera-se que sejam ressaltadas as linhas inferiores e direitas, conforme observado na Figura 48.

Por fim, na Figura 49, observa-se a combinação das imagens filtradas resultantes pela aplicação dos filtros h_3 e h_4 pela fórmula $\sqrt{(h_3)^2 + (h_4)^2}$. Conforme esperado, como h_3 ressaltava linhas verticais e h_4 linhas horizontais, a combinação dos dois filtros faz com que a imagem resultante apresente ambas linhas (bordas) verticais e horizontais realçadas.

4 Conclusão

As operações simples de transformações e manipulação de imagens, geralmente, podem ser feitas de maneira consideravelmente mais prática e eficiente com o uso de vetorização, quando possível. Porém, em alguns casos, como ocorrido no item 3.8, na filtragem de imagens por correlação, foi necessário o uso de laços encadeados para percorrer a imagem, o que tornou o processo significativamente mais lento.

Também é importante se atentar à forma como as bibliotecas de leitura e exibição de imagens trabalham com os dados. Isto é, deve-se saber como será devolvida a imagem após a leitura, tanto para níveis de cinza como para RGB. Da mesma forma, deve-se ter consciência de como a exibição é feita e dos parâmetros que devem ser utilizados para que a imagem seja exibida da maneira desejada.

Enfim, essa série de exercícios foi uma importante prática para que pudesse ser compreendidos conceitos fundamentais do processamento de imagens digitais.