

MC920 - Trabalho 1

Tobias Conran Zorzetto, RA: 166214

April 2023

1 Introdução

Esse trabalho tem o objetivo de realizar uma série de 8 tarefas de conteúdo básico a respeito de processamentos em imagens digitais. Nesse contexto, esse relatório tem como objetivo apresentar essas tarefas e suas soluções, trazendo com si análises a respeito dos algoritmos utilizados e seus resultados.

Assim, o texto está dividindo em 3 seções: **Introdução**, **Execução**, onde será explicado de maneira geral como foram feitos os programas, além das bibliotecas e ferramentas utilizadas, e **Problemas**, onde serão apresentadas as soluções e resultados de cada tarefa dada.

2 Execução

Para a solução das tarefas dadas no trabalho 1, alguns padrões foram adotados. Primeiramente, todos os códigos que implementam os algoritmos soluções foram feitos em python. Além disso, foram utilizadas 3 bibliotecas em geral para resolução.

A biblioteca *numpy* foi utilizada para manipular e fazer operações nas imagens em forma de vetorização. A biblioteca *imageio* foi utilizada para ler as imagens originais, salvando-as em vetores do *numpy*. Por fim, *matplotlib.pyplot* foi utilizada para salvar as imagens transformadas.

Além disso, a pasta onde está enviada como resolução final do arquivo está organizada da seguinte maneira: Os códigos para resolução de cada tarefa estão nomeados como **exercicioX.py**, sendo X o índice da tarefa dentro do item "Especificação do Problema" no enunciado do trabalho. Já os resultados de cada tarefa podem ser encontrados dentro da pasta **resultados**, dentro de seu respectivo exercício. Na pasta também estão presentes as imagens **baboon.png**, **butterfly.png**, **city.png** e **parrot.png** (figuras 1, 2, 3 e 4), todas no formato PNG, que serão utilizadas como imagens originais para modificação em cada tarefa.

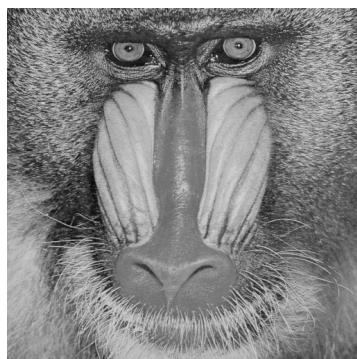


Figura 1: baboon



Figura 2: butterfly



Figura 3: city



Figura 4: parrot

3 Resolução e Análise de Resultados dos Problemas

3.1 Mosaico

Essa tarefa não foi realizada.

3.2 Combinação de Imagens

Para a resolução dessa questão, foram utilizadas as imagens 1 como imagem A e 2 como imagemB. Como o objetivo é a combinação de duas imagens, pode-se ser usada a função `add()` do numpy, em que dados dois vetores, ou matriz de vetores, no caso as 2 imagens citadas, consegue-se somar cada uma de suas respectivas posições, ou até mesmo somar 2 operações sobre cada uma das respectivas posições. Dessa forma, a função pode ser utilizada da seguinte maneira:

```
np.add(k1*img1,k2*img2)
```

que retornará uma imagem em que cada pixel é a soma de cada pixel das imagens multiplicados pelas suas respectivas contantes `k1` e `k2`.

Assim, os resultados para **(a)** $0.2 * A + 0.8 * B$, **(b)** $0.5 * A + 0.5 * B$ e **(c)** $0.8 * A + 0.2 * B$ podem ser vistas nas figuras 3.2, 3.2 e 7. Pode-se ver que quanto maior a constante que multiplica cada pixel da imagem, mais fácil a distinção dessa imagem na combinação resultante. Dessa forma, na imagem da figura 3.2 é mais fácil distinguir o babuíno, enquanto na imagem da figura 7 é mais fácil distinguir a borboleta, enquanto que na figura 3.2, as imagens parecem estar igualmente presentes.

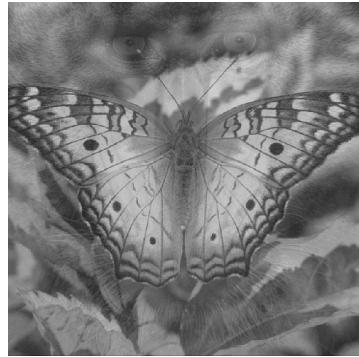


Figura 5: $0.2 * A + 0.8 * B$

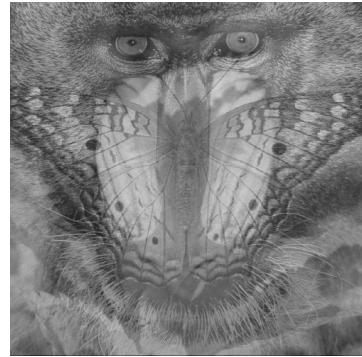


Figura 6: $0.5 * A + 0.5 * B$

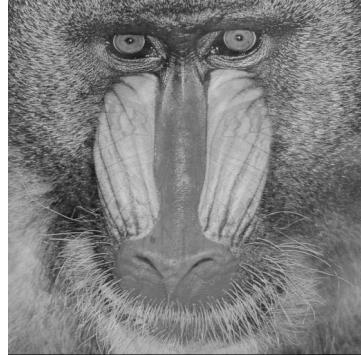


Figura 7: $0.8 * A + 0.2 * B$

3.3 Transformação de Intensidade

Para esse exercício de transformação de intensidade dos níveis de cinza da imagem, a imagem original escolhida foi a figura 3. Assim, para cada item foi feito uma função no código que realiza o que é pedido.

Para o **item (b)** foi feita a função `negative` que recebe a imagem original como parâmetro e retorna a imagem negativada. Isso é feito a partir da função `add` do *numpy* introduzida no tópico anterior, mas com a seguinte configuração:

```
np.add(255,-img)
```

que subtrai cada um dos valores dos pixels de 255, o que inverte seus valores na escala de cinza, assim como pode ser visto no resultado mostrado na figura 8, em que os pontos mais escuros da imagens se tornaram os mais claros e vice-versa.



Figura 8: imagem negativada

Para o **item (c)** foi feita a função `transform` que recebe a imagem original como parâmetro e retorna a imagem com níveis de cinza transformados. Isso é feito a partir das funções `add` `multiply`

do *numpy* que operam de forma similar sobre cada ponto de uma imagem. Dessa forma, pode-se utilizar da formula $p_{novo} = \frac{100}{255}p + 100$ para converter cada valor da imagem original de um intervalo de intensidades de 0-255 para um intervalo entre 100-200, por final a imagem é retornada com a função `np.int16()` que transforma todos os valores em inteiros. O resultado disso pode ser visto na imagem da figura 9, que mostra uma imagem com menos contraste que a anterior, com tons mais próximos, dentro de uma menor escala de cinzas de intensidade mais mediana.



Figura 9: imagem transformada

Para o **item (d)** foi feita a função `invertOdds` que recebe a imagem original como parametro e inverte as suas linhas pares. Isso é feito a partir da propriedade de vetorização trazida pelo *numpy* que permite que essa ação seja feita com apenas a seguinte linha:

```
img[1::2,:] = img[1::2,-1]
```

que indica que todas as linhas pares (1::2 indica que a cada 2 linhas, tal linha receberá a seguinte operação), receberam as mesmas linhas pares invertidas (-1 indica a alteração de posição das colunas na linha para a posição invertida). O resultado dessa função pode ser visto na figura 10.



Figura 10: imagem com as linhas pares invertidas

Para o **item (e)** foi feita a função `reflectRows` que recebe a imagem original como parametro e reflete a segunda metade das suas linhas baseadas na primeira metade. Sabendo que com `rows = img.shape[0]` tem-se o número de linhas na imagem, isso também é feito a partir da vetorização da seguinte forma:

```
img[int(rows/2)-1:] = img[int(rows/2):: -1]
```

que indica que a segunda metade das linhas (`int(rows/2)-1::` indica a segunda metade do conjunto de linhas sem nenhuma mudança na posição delas) recebe as linhas da primeira metade na ordem invertida. O resultado disso pode ser visto na figura 11 que apresenta uma modificação da imagem original com simetria no eixo horizontal.

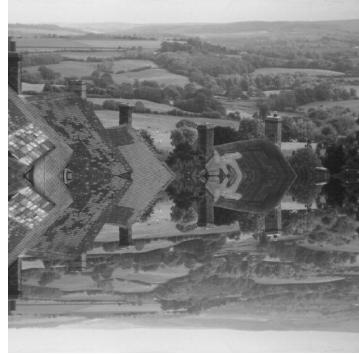


Figura 11: imagem com as linhas refletidas no eixo horizontal

Para o **item (f)** foi feita a função `VerticalMirroring` que recebe a imagem original como parâmetro e inverte ela no sentido vertical. Também com vetorização isso é feito assim:

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

em que toda a imagem original recebe a mesma porém com a ordem das linhas invertida (`[::-1]` garante a inversão da dimensão indicada). Isso pode ser visto na figura 12, que apresenta a imagem original de ponta cabeça, porém sem inversão das colunas.



Figura 12: imagem invertida verticalmente

3.4 Imagens Coloridas

Essa tarefa foi feita utilizando a imagem da figura 4. Para o **item (a)**, foi possível utilizar da vetorização em `numpy` para mudar os valores de cada cor em cada pixel. A seguinte linha de código explica:

```
newImgA[:, :, C] = img[:, :, 0]*k0 + img[:, :, 1]*k1 + img[:, :, 2]*k2
```

O que essa linha indica é que para cada posição C do ponto (C variando de 0 a 2, indicando os valores de vermelho, verde e azul), esse ponto receberá a combinação de $k_0 * valor_{azul} + k_1 * valor_{verde} + k_2 * valor_{azul}$. Sendo os valores dos k_s definidos pelo problema, o resultado dessa transformação é dado pela figura 13.



Figura 13: imagem com seus valores de cor transformados

Para o **item (b)**, algo parecido foi feito:

```
newImgB = img[:, :, 2]*k0 + img[:, :, 1]*k1 + img[:, :, 0]*k2
```

Tal linha mostra que cada ponto receberá a soma das combinações das cores RGb multiplicadas pelas constantes indicadas no trabalho. Assim, o resultado desse item pode ser visto na figura 14



Figura 14: imagem colorida convertida para tons de cinza

3.5 Ajuste de Brilho

Para a resolução desse tópico foi utilizada como imagem fonte a figura 1. Com isso foi feita a função gama que recebe como parametros a imagem fonte (`img`) e o valor de gama utilizado (`value`). Assim, aplica-se diretamente a fórmula $B = A^{(1/\gamma)}$ da seguinte forma:

```
img**(1/value)
```

que permite que a operação seja feita em cada ponto da imagem devido ao funcionamento da vetorização em *numpy*. Vale observar que *matplotlib.pyplot* já converte os valores para o intervalo 0-255 ao salvar a imagem em um arquivo PNG.

Assim, os resultados para valores de $\gamma = 1.5$, $\gamma = 2.5$ e $\gamma = 3.5$ podem ser vistos nas figuras 15, 16 e 17. Pode-se ver que a variação do gamma muda a percepção do brilho na imagem. Visualmente o aumento do gamma parece aumentar linearmente a luminosidade da imagem.

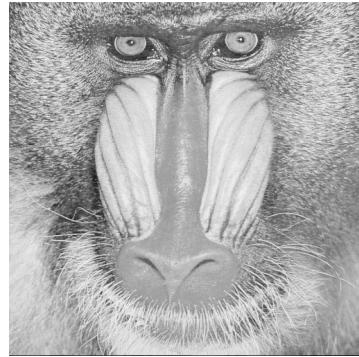


Figura 15: $\gamma = 1.5$

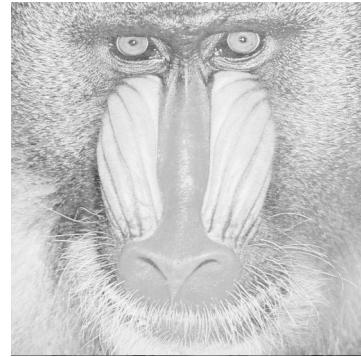


Figura 16: $\gamma = 2.5$

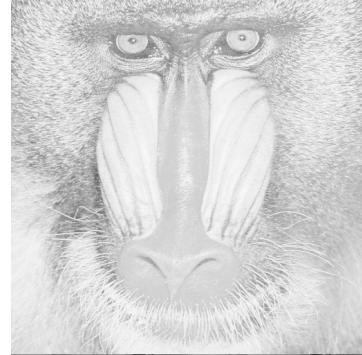


Figura 17: $\gamma = 3.5$

3.6 Quantização de Imagens

Para a tarefa de quantização de imagens, foi utilizada a imagem 1 como imagem fonte. Ela pode ser pensada de forma binária. Na qual se a imagem original possui 256 níveis de cinza, um *bitshift* a direita de cada pixel da imagem cortará a possibilidade de níveis de cinza pela metade. Isso pode ser feito a partir de uma função `level` que recebe como parametros a imagem fonte (`img`) e a quantidade de shift a direita desejada (`value`), que retorna a imagem com menos níveis de cinza a partir de:

```
img >> value
```

O resultado disso para 64, 32, 16, 8, 4 e 2 níveis podem ser vistos nas imagens das figuras 18, 19, 20, 21, 22, 23. Pode se observar que quanto menor o nível de cinza, maior o contraste da imagem, porém além disso, menor o nível de detalhamento, já que a cada *bitshift* a direita feito, os bits menos significativos são ignorados, ignorando as variações de intensidade decorrentes desses valores.

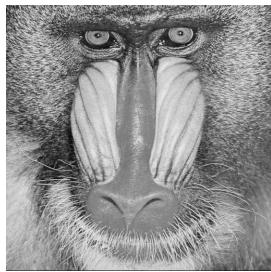


Figura 18: imagem com 64 níveis de cinza

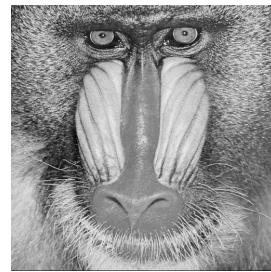


Figura 19: imagem com 32 níveis de cinza

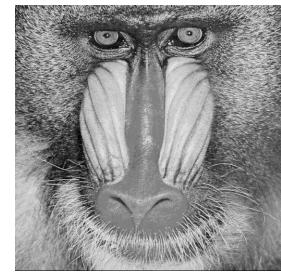


Figura 20: imagem com 16 níveis de cinza

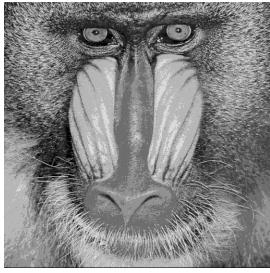


Figura 21: imagem com 8 níveis de cinza

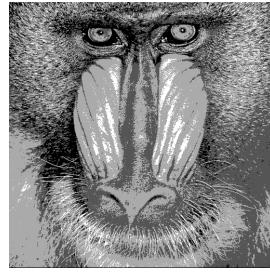


Figura 22: imagem com 4 níveis de cinza

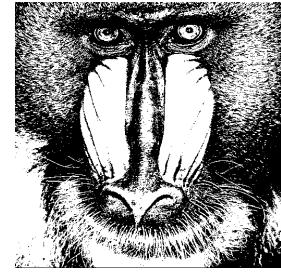


Figura 23: imagem com 2 níveis de cinza

3.7 Planos de Bits

Para extraír os planos de bits de uma imagem em escala de cinza, pode-se aplicar uma operação binária sobre os seus valores, assim como na questão anterior. O que pode ser feito é um *bitwise and* com cada valor da imagem e 2^b sendo b o bit em que se interessa o plano. Dessa forma os valores desse bit podem ser isolados na imagem modificada. Isso pode ser feito a partir de uma função `bitPlane` que recebe o bit de interesse (`bit`) e a imagem fonte (`img`) e realiza

```
np.bitwise_and(img,2**bit)
```

que realiza tal operação descrita pra cada ponto da imagem. Os resultados dos planos de bit 0,4,7 para a imagem 1 como imagem fonte estão nas figuras 24, 25 e 26. Vê-se com eles de que quanto maior o bit representado pelo plano, mais similar à imagem original parece ser a imagem transformada da original, enquanto quanto mais baio, mais ruídososa se torna a imagem. Isso pode ser explicado pelo fato de que quanto maior a posição do bit de um valor, maior será a variação que ele carrega, assim, maior o seu valor para formação da imagem como um todo.

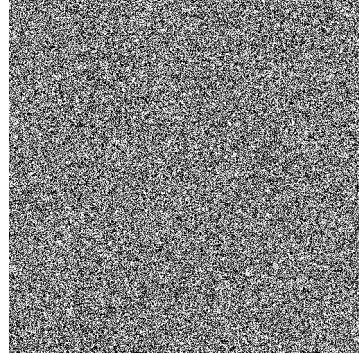


Figura 24: plano de bit 0



Figura 25: plano de bit 4

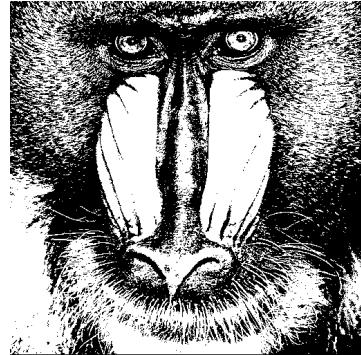


Figura 26: plano de bit 7

3.8 Filtragem de Imagens

Para a resolução dessa tarefa foi utilizada como imagem fonte a imagem 1.

Para a filtragem de imagens com diferentes filtros, de tamanhos e funções diferentes, foi criada a função `filter` que recebe como parametro a imagem alvo (`img`), a máscara de filtragem (`mask`) com os valores inteiros do filtro, e o divisor (`divider`), presente em algumas mascaras.

A função inicia criando uma `newImg` vazia de mesmo formato que a imagem fonte, depois é calculada a borda necessária para a imagem original para que a filtragem possa ocorrer nas bordas também. Assim, as novas bordas são aplicadas a imagem original. O método escolhido para a determinação da nova borda foi o modelo "*symmetric*" próprio do `numpy`, que reflete o vetor a qual a borda está sendo adicionada. Essa nova imagem com borda adicionada é guardada em uma variável `imgPadded`.

Após isso, cada item de `imgPadded` é percorrido, ignorando os valores de borda. O resto dos valores, que não são borda então são passados cada um como parâmetro para a função `setValue`, que definirá o valor correspondente para `newImg`, a nova imagem filtrada.

A função `setValue` citada anteriormente recebe como parametros a matriz de filtragem (`matrix`), o divisor (`divider`), a imagem com as bordas a mais (`imgPadded`) e as posições $pixel_i$ e $pixel_j$ em `imgPadded` do ponto a que será transformado. Assim, devolve o novo valor para tal ponto, fazendo a conrrelação dos pontos a sua volta com os valores da máscara, e então dividindo esse valor pelo `divider`.

Assim, após aplicar a função `setValue` para todos os pontos da nova imagem, os valores de `newImg` são arredondados e em seguida são limitados a valores entre 0-255.

Esse processo é feito para cada um dos 11 filtros, de h1-11. Assim, pode ser feita uma análise dos resultados de cada filtro.

Para o filtro h1, a imagem obtida pode ser vista na figura 27. Ele parece destacar as bordas da figura, porém sem diminuição de ruídos. A imagem ainda contém um certo nível de detalhamento.

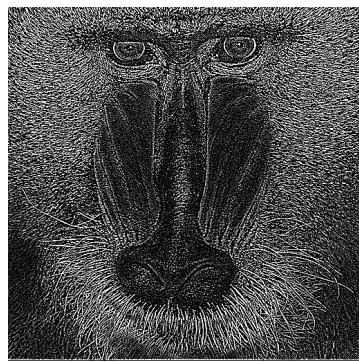


Figura 27: resultado da aplicação do filtro h1 em imagem

Para o filtro h2 (figura 28), nota-se uma atenuação dos ruídos. Apesar de a imagem ainda estar muito bem definida, existe um borrão, que parece ser proveniente do espalhamento dos níveis de cinza pelo filtro.

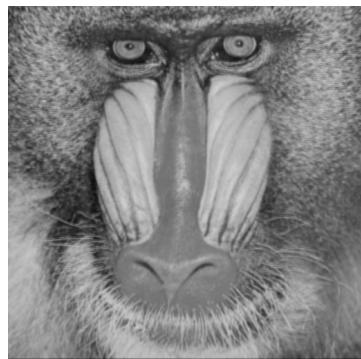


Figura 28: resultado da aplicação do filtro h2 em imagem

Com a aplicação do filtro h3 (figura 29), pode ser visto que prevalecem as linhas de continuidade mais verticais, com menor prevalência de linhas horizontais.

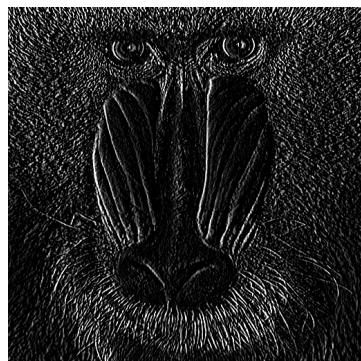


Figura 29: resultado da aplicação do filtro h3 em imagem

Já para o filtro h4 (figura 30), o contrário pode ser visto em relação ao h3. As linhas horizontais estão destacadas, enquanto as linhas verticais estão menos presentes.



Figura 30: resultado da aplicação do filtro h4 em imagem

A aplicação do filtro h5 (figura 31) é similar em comportamento ao filtro h1, porém nota-se uma menor presença de ruídos e um maior destaque das bordas em relação aos pontos ao seu entorno.

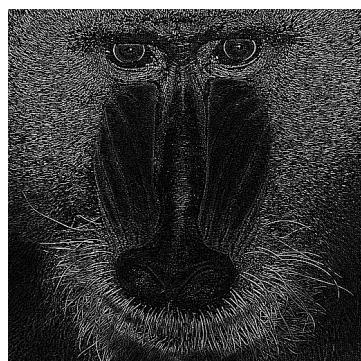


Figura 31: resultado da aplicação do filtro h5 em imagem

O filtro h6 (figura 32), também traz resultado similar ao filtro h2. Apesar disso, seu efeito de borramento é menor, já que o tamanho da máscara menor faz um "espalhamento" em menor região em cada ponto da figura.

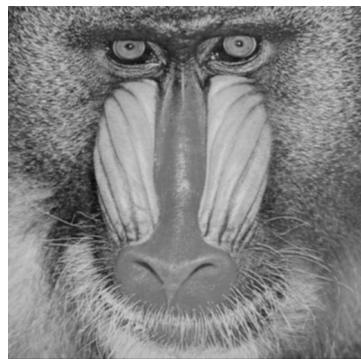


Figura 32: resultado da aplicação do filtro h6 em imagem

Para o filtro h7 (figura 33), o efeito causado também parece ressaltar as linhas diagonais da imagem original, atenuando as outras.

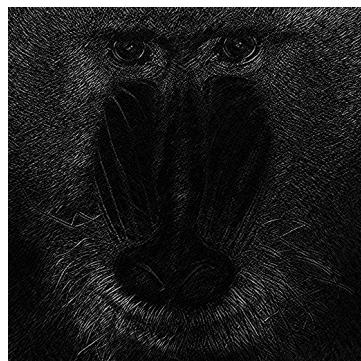


Figura 33: resultado da aplicação do filtro h7 em imagem

O filtro h8 (figura 34), por outro lado, ressalta as linhas diagonais também, porém na direção contrária as linhas ressaltadas pelo filtro h7.

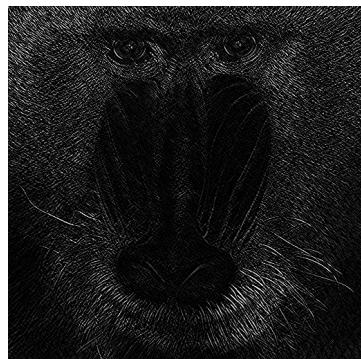


Figura 34: resultado da aplicação do filtro h8 em imagem

Para a aplicação do filtro h9 (figura 35), o resultado é uma imagem ainda muito detalhada, porém borrada de forma a passar a sensação visual de ”movimento” em diagonal para a imagem a que foi aplicada.

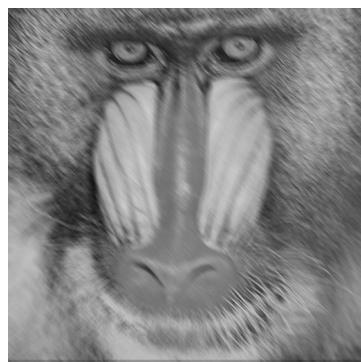


Figura 35: resultado da aplicação do filtro h9 em imagem

Já a imagem resultante da aplicação do filtro h10 (figura 36), parece não haver grande borramento da figura, porém se mostra com menos ruídos que a original e, além disso um pouco mais clara.

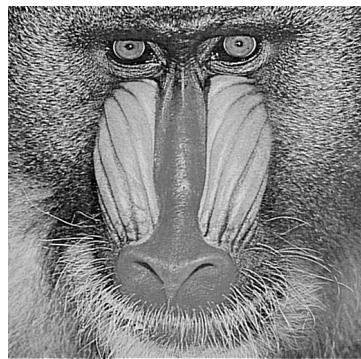


Figura 36: resultado da aplicação do filtro h10 em imagem

Por final, o filtro h11 (figura 37) mostra um realce das bordas inferiores direitas da imagem original.

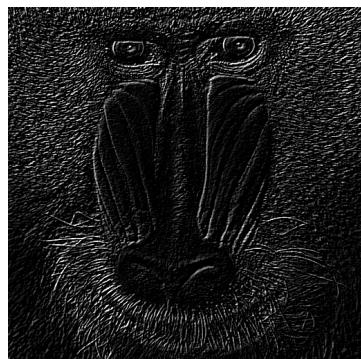


Figura 37: resultado da aplicação do filtro h11 em imagem