[PSI3471 aula 1, início]

Alunos e alunas da PSI3471, 2024:

Na primeira parte do curso, com prof. Magno, vocês estudaram regressão linear, perceptron, rede neural, até chegar à rede neural convolucional.

Na segunda parte do curso também vamos chegar à rede neural convolucional. Porém, vamos seguir um caminho diferente. Vamos estudar processamento de imagens, algoritmos clássicos de aprendizado de máquina (existem muitos algoritmos de aprendizado de máquina diferentes de redes neurais!), extração de atributos, rede neural, até chegar à rede neural convolucional. Acredito que as abordagens das duas partes do curso fornecerão visões complementares sobre o aprendizado profundo.

A minha parte irá esclareder como as pesquisas em processamento de imagens e aprendizado de máquina acabaram resultando no atual aprendizado profundo.

Nesta disciplina, veremos somente os tópicos básicos. Os assuntos avançados serão estudados na disciplina do 2º semestre: "PSI-3472 Concepção e Implemementação de Sistemas Eletrônicos Inteligentes", que é a continuação desta disciplina.

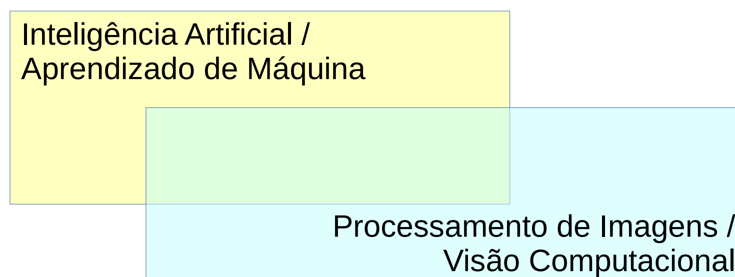


Figura F: Relações entre diferentes áreas envolvidas neste curso.

Na segunda parte do curso, vamos estudar (figura F):

1. Processamento de imagens e visão computacional sem usar aprendizado de máquina.
2. Aprendizado de máquina clássico.
3. Aprendizado de máquina usando rede neural convolucional, com aplicações em processamento de imagens e visão computacional.

Técnicas básicas para Processamento de Imagens e Visão Computacional

[Nota para mim: Em PSI5790, pode correr no início da disciplina (que usa C++) que não dá tempo de dar todo o conteúdo em 12 aulas.]

Utilizaremos dois ambientes de programação neste curso.

1) Na primeira parte do curso, utilizaremos a linguagem C++ com biblioteca OpenCV.

1.1) Para usar os computadores da sala GD-06:
a) User: aluno   password: aluno@gd06
b) Nos computadores da sala GD-06, bibliotecas OpenCV v2 e v3 estão instalados dentro de Cekeikon. Comando para ativar Cekeikon no terminal:

```
diretorio$ source ~/cekeikon5/bin/ativa_cekcpu
```

c) No final da aula, copiem os seus programas/documentos em seu pendrive ou na nuvem. Depois apaguem os seus arquivos, pois outros alunos irão usar o mesmo computador.

1.2) Para rodar programas C++ no seu computador, há duas formas:

a) Instalar e usar Cekeikon. Cekeikon é um pacote com biblioteca OpenCV e está disponível para Windows e Linux em:
   http://www.lps.usp.br/hae/software/cekeikon56.html
Não deve ter bug grave, pois foi testado durante muitos anos por centenas de alunos.
Instalando Cekeikon, você estará instalando todo o ambiente necessário para compilar e executar programas C++: compilador de C++ (só no caso Windows, pois em Linux usará o compilador nativo do sistema) e biblioteca OpenCV versões 2 e 3.

b) Usar OpenCV versão 4 nativo das versões atuais do Linux, sem Cekeikon. Para usar este ambiente, siga as instruções do "manual-nocek" em:
   http://www.lps.usp.br/hae/software/cekeikon56.html
Se você usa Windows, você deve instalar primeiro WSL (Windows Subsystem for Linux) e seguir o mesmo procedimento.

2) Na segunda parte, utilizaremos a linguagem Python com várias bibliotecas, entre elas Tensorflow e Keras, no computador local ou em Google Colab.

Nota: Testes independentes mostram que C++ é algo como 10 a 100 vezes mais rápido e normalmente usa menos memória do que Python:
   https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gpp-python3.html
   https://www.freecodecamp.org/news/python-vs-c-plus-plus-time-complexity-analysis/
Isto é especialmente verdade em Processamento de Imagens, onde normalmente os programas precisam varrer imagens.

1. Programa mostra

Para testar o primeiro programa, abra um terminal (prompt de comando), vá para algum diretório de trabalho (cd diretorio_trabalho), e descompacte neste diretório as imagens do arquivo http://www.lps.usp.br/hae/apostila/basico.zip. Em Linux, isto pode ser feito pelos seguintes comandos:

```
$ wget -nc -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/basico.zip
$ unzip basico
```

Dê copy-paste do programa abaixo em algum editor de texto, salve o programa no diretório com o nome "mostra_cv.cpp".

```
1 //mostra_cv.cpp
2 #include <opencv2/opencv.hpp>
3 using namespace std;
4 using namespace cv;
5 int main() {
6   Mat_<Vec3b> a=imread("lenna.jpg",1);
7   imshow("janela",a);
8   waitKey(0);
9 }
```

Para compilar mostra_cv.cpp em ambiente Cekeikon, execute:

```
  cekeikon$ compila mostra_cv -ocv     (OpenCV2)
  cekeikon$ compila mostra_cv -ocv -v3 (OpenCV3)
```

Em Cekeikon, é possível especificar se deseja linkar com OpenCV2 ou OpenCV3. Se você não especificar nada, será linkada a versão 2.

Para compilar esse programa em Linux sem Cekeikon:

```
   diretorio$ g++ -std=gnu++14 mostra_cv.cpp -o mostra_cv -fmax-errors=2
`pkg-config opencv4 --libs --cflags` -O3 -s
```

Se você criou compila.sh conforme manual-nocek.pdf:

```
  nocek$ compila.sh mostra_cv
```

O programa deve abrir uma janela e mostrar a imagem "lenna.jpg". Para sair do programa, aperte ESC.

A linha 2 pede para incluir os cabeçalhos das funções da biblioteca OpenCV. As linhas 3 e 4 dizem que vamos usar as funções dos espaços de nomes *std* (biblioteca padrão de C++) e *cv* (OpenCV).

A linha 6 declara uma matriz onde cada elemento é capaz de armazenar uma Vec3b, isto é 3 bytes formado pelas bandas B, G e R. Depois, lê a imagem "lenna.jpg" como uma imagem colorida (parâmetro 1) e armazena na matriz *a*.

A linha 7 mostra a matriz *a* numa janela chamada "janela" usando comando "imshow". Após *imshow*, é necessário pedir para mostrar a imagem até que tecle algo com *waitKey(0)*. Argumento "zero" indica para esperar indefinidamente até que o usuário aperte alguma tecla. Sem o comando *waitKey*, nada será mostrado na tela.

O mesmo programa em Python:

<table>
<tr>
<td><pre>1
2
3
4
5
6
7
8
9</pre></td>
<td>

```
#leimg1.py
import cv2
nome="lenna.jpg"
a=cv2.imread(nome,1)
cv2.imshow("janela",a)
cv2.waitKey()
```

</td>
<td>

```
#leimg2.py
import cv2;
from matplotlib import pyplot as plt;
nome="lenna.jpg"
a=cv2.imread(nome,1)
a=cv2.cvtColor(a,cv2.COLOR_BGR2RGB)
plt.imshow(a)
plt.show()
```

</td>
</tr>
<tr>
<td></td>
<td>Este programa pode não funcionar em Google Colab.
Linux>python3 leimg1.py
Windows>python leimg1.py</td>
<td>Este programa funciona no computador local e em Google Colab.
Linux>python3 leimg2.py
Windows>python leimg2.py</td>
</tr>
</table>

2. Programa inverte

Agora, vamos ler uma imagem binária, calcular a sua imagem negativa (onde preto e branco são trocados) e imprimi-la.

É possível programar em C++/OpenCV com ou sem programação por template. Usando template, o tipo da matriz fica definida durante a compilação. Sem usar template, o tipo da matriz é definida durante a execução. Neste curso, vamos usar a programação com template, pois acredito que simplifica a programação.

Usando template, o tipo da matriz *a* (uchar) fica explícito:

```
 1 //inv_ocv.cpp - usando template
 2 #include <opencv2/opencv.hpp>
 3 using namespace std;
 4 using namespace cv;
 5 int main()
 6 { Mat_<uchar> a=imread("mickey_reduz.bmp",0);
 7   for (int l=0; l<a.rows; l++)
 8     for (int c=0; c<a.cols; c++)
 9       if (a(l,c)==0) a(l,c)=255;
10       else a(l,c)=0;
11   imwrite("inv_ocv.pgm",a);
12 }
```

Compile com o comando:

```
cekeikon> compila inv_ocv -ocv
nocek> compila.sh inv_ocv
```

Na linha 6, Mat_<uchar> é uma matriz capaz de armazenar uma imagem em níveis de cinza (ou binária). Em C++, *uchar* indica uma variável 8 bits sem sinal, que outras bibliotecas chamam de *uint8_t* (C++), *unsigned char* (C) ou *BYTE* (Microsoft). OpenCV não possui estrutura própria para armazenar imagens binárias. Assim, estamos usando imagem em níveis de cinzas para armazenar imagem binária, onde preto é 0 e branco é 255. Isto é um certo desperdício de memória, pois utiliza 8 bits por pixel quando poderia usar 1 bit... Por outro lado, o processamento fica mais rápido.

A matriz *a* tem *a.rows* linhas e *a.cols* colunas. A numeração das linhas da imagem *a* vai de 0 até *a.rows-1*, de cima para baixo. A numeração das colunas vai de 0 até *a.cols-1*, da esquerda para direita.

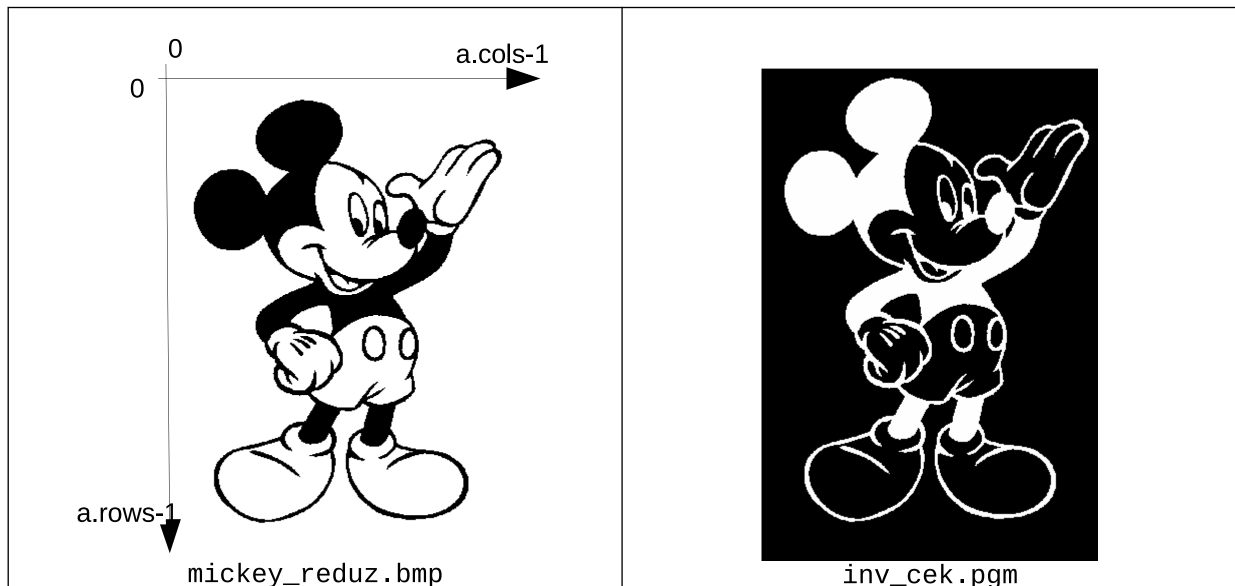


Figura 1: Calcular imagem negativa de uma imagem binária.

O mesmo programa sem usar "template":

```
1  //inv_ocv2.cpp
2  #include <opencv2/opencv.hpp>
3  using namespace std;
4  using namespace cv;
5  int main()
6  { Mat a=imread("mickey_reduz.bmp",0);
7    for (int l=0; l<a.rows; l++)
8      for (int c=0; c<a.cols; c++)
9        if (a.at<uchar>(l,c)==0) a.at<uchar>(l,c)=255;
10       else a.at<uchar>(l,c)=0;
11   imwrite("inv_ocv.pgm",a);
12 }
```

Note que a matriz *a* está declarada sem tipo. Como o compilador não conhece o tipo de matriz *a*, é necessário informar o tipo de matriz *a* toda vez que a acessa (em amarelo).

O programa abaixo é a solução em Python. Repare como a sintaxe é praticamente idêntica.

```
 1 # inverte.py
 2 import cv2
 3 a = cv2.imread('mickey_reduz.bmp',0)
 4 
 5 for l in range(a.shape[0]):
 6   for c in range(a.shape[1]):
 7     if a[l,c]==0:
 8       a[l,c]=255
 9     else:
10       a[l,c]=0
11 
12 cv2.imwrite('inverte_py.png',a)
```

Para rodar este programa:

```
  windows> python inverte.py
  linux$ python3 inverte.py
```

Uma outra solução para este problema em C++ é acessar a matriz *a* usando um único índice (em vez de linha e coluna). Ao incrementar o índice unidimensional, a imagem é percorrida em ordem “raster”, isto é, da esquerda para direita e de cima para baixo.

```
 1 //inv_1d2.cpp - usando template
 2 #include <opencv2/opencv.hpp>
 3 using namespace std;
 4 using namespace cv;
 5 int main()
 6 { Mat_<uchar> a=imread("mickey_reduz.bmp",0);
 7   for (int i=0; i<a.total(); i++)
 8     if (a(i)==0) a(i)=255;
 9     else a(i)=0;
10   imwrite("inv_1d2.bmp",a);
11 }
```

Na linha 7, a.total() indica o número total de elementos da matriz.

Para ilustrar o acesso aos pixels, escrevemos explicitamente o cálculo da imagem inversa dentro de um ou dois loops. Porém, na verdade, todos os comandos dentro dos loops podem ser substituídos por um único comando que subtrai de 255 cada um dos elementos da matriz. Isto funciona tanto em C++ como em Python:

```
  a = 255 - a;
```

3. Programa borda

Agora, vamos escrever um programa que detecta as bordas da imagem mickey_reduz.bmp (imagem obtida eliminando ruído e diminuindo resolução de “mickey.bmp”).

Para detectar as bordas, podemos usar a lógica mostrada na figura 3b: Pixel A é borda se A≠B ou A≠C.

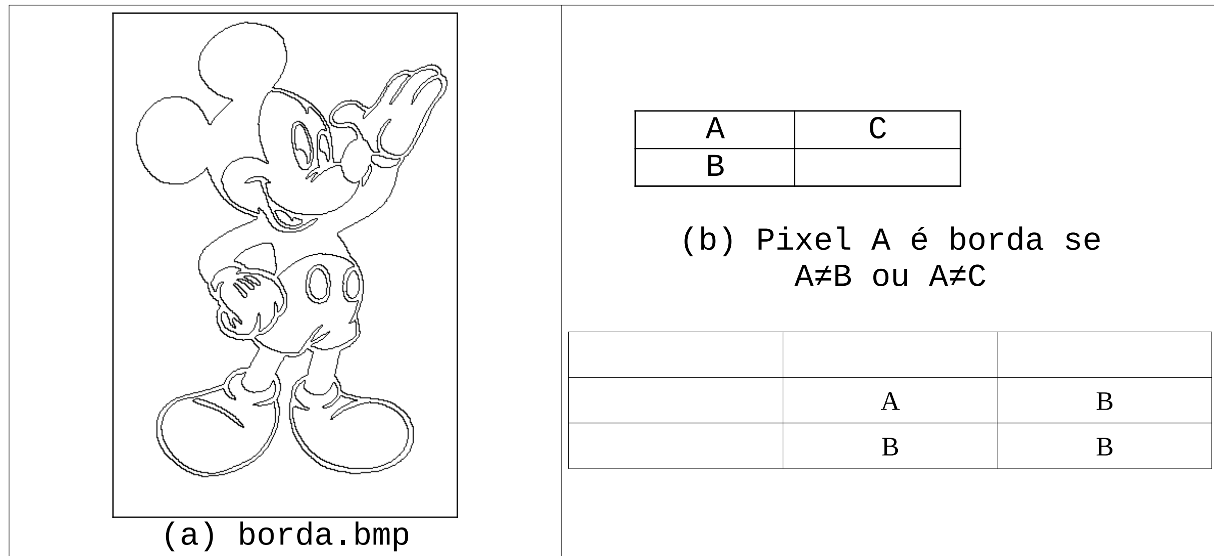


Figura 3: Detecta bordas.

A solução está no programa abaixo, usando a biblioteca Cekeikon:

```
 1 //borda.cpp
 2 #include <opencv2/opencv.hpp>
 3 using namespace std;
 4 using namespace cv;
 5 int main() {
 6   Mat_<uchar> a=imread("mickey_reduz.bmp",0);
 7   Mat_<uchar> b(a.rows,a.cols);
 8   for (int l=0; l<a.rows-1; l++)
 9     for (int c=0; c<a.cols-1; c++)
10       if (a(l,c)!=a(l,c+1) || a(l,c)!=a(l+1,c))
11         b(l,c)=0;
12       else
13         b(l,c)=255;
14   imwrite("borda.bmp",b);
15 }
```

Este programa tem um pequeno erro, que faz com que a última linha e a última coluna da imagem de saída sejam (normalmente) pretas. Explique como consertar este erro.

Aqui, precisamos trabalhar com duas imagens: uma imagem de entrada *a* e uma imagem de saída *b*. A linha 7 cria a imagem *b* com as mesmas dimensões da imagem *a*. Note que se escrevêssemos:

```
Mat_<uchar> b; //cria imagem b nula (0 linhas e 0 colunas)
```

estaríamos criando uma matriz *b* vazia, com 0 linhas e 0 colunas. É possível alocar espaço para matriz vazia *b* usando depois o comando:

```
b.create(rows,cols);
```

Isto alocaria rows×cols elementos para a matriz *b*. Também é possível criar a matriz e alocar espaço ao mesmo tempo (que é o que fizemos acima):

```
Mat_<uchar> b(rows,cols); //cria imagem b com rows*cols pixels
```

Uma terceira possibilidade é criar a matriz, alocar espaço e preencher todos os elementos com um determinado valor:

```
Mat_<uchar> b(rows,cols,255); //cria imagem b com rows*cols pixels
                             //e preenche com 255.
```

Isto funciona sempre, exceto quando quer preencer a matriz com zeros, pois "0" tem um significado especial (apontador nulo) em C++:

```
Mat_<uchar> b(rows,cols,0); //Dá erro. Não sabe se 0 é número ou endere-
ço.
```

Para contornar isto, podemos escrever:

```
Mat_<uchar> b(rows,cols,uchar(0)); //cria imagem b com rows*cols pixels
                                    //e preenche com 0.
```

[PSI3471 aula 1, lição de casa #1] Escreva um programa que elimina o ruído branco da imagem "mickeyruibr.bmp".
Nota: Você deve tomar cuidado para não acessar pixels fora do domínio da imagem (pixel antes da primeira coluna (linha) e pixel depois da última coluna (linha)), pois os valores nesses pixels estão indefinidos em C++ e pode gerar erro de índice inválido em Python.

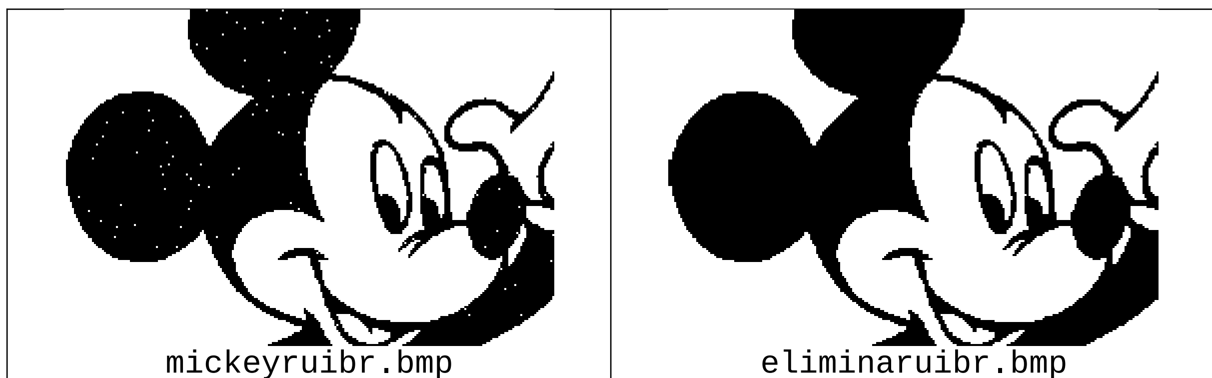
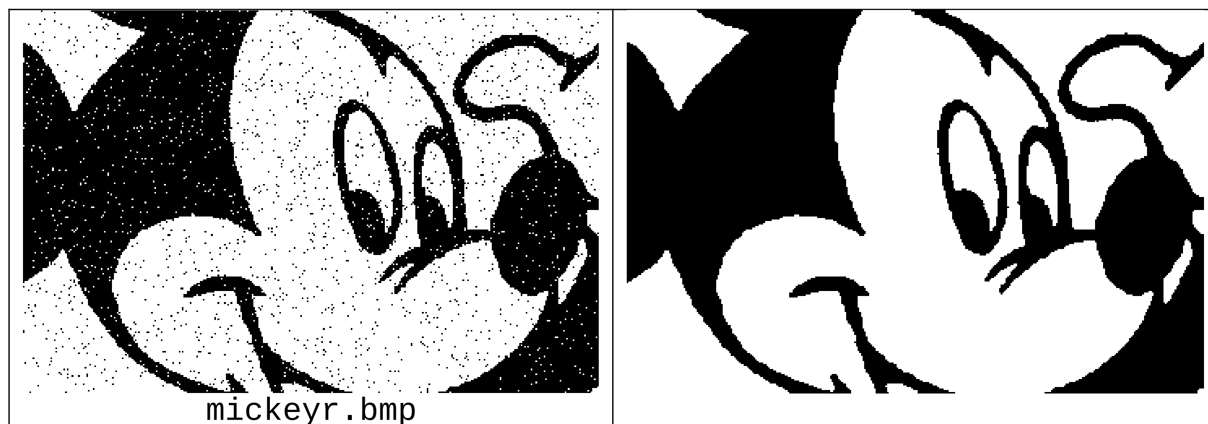


Figura 2: Elimina ruído.

[PSI3471 aula 1, lição de casa extra. +2 pontos] Escreva um programa que elimina o ruídos brancos e pretos da imagem “mickeyr.bmp”.



mickeyr.bmp

[PSI3471 aula 1, fim]

4. Inverter uma imagem colorida

Agora, vamos escrever um programa que inverte as cores de uma imagem colorida, para mostrar como acessar os pixels de uma imagem colorida.

```
 1 //invertec_ocv.cpp - pos2024
 2 #include <opencv2/opencv.hpp>
 3 using namespace std;
 4 using namespace cv;
 5 int main() {
 6   Mat_<Vec3b> a;
 7   a=imread("lenna.jpg",1);
 8   for (int l=0; l<a.rows; l++)
 9     for (int c=0; c<a.cols; c++) { //BGR
10       a(l,c)[0] = 255-a(l,c)[0]; // blue
11       a(l,c)[1] = 255-a(l,c)[1]; // green
12       a(l,c)[2] = 255-a(l,c)[2]; // red
13     }
14   imwrite("invertec.jpg",a);
15 }
```

As linhas 8-9 mostram como acessar cada uma das bandas de um pixel (l,c) da matriz *a*. OpenCV armazena as cores na ordem BGR, contrária à grande maioria das bibliotecas que trabalham com ordem RGB.

Figura 4: Entrada e saída do programa invertec.cpp.

```python
# invertec.py
import cv2
a = cv2.imread('lenna.jpg',1)
for l in range(a.shape[0]):
  for c in range(a.shape[1]):
    a[l,c,0]=255-a[l,c,0]
    a[l,c,1]=255-a[l,c,1]
    a[l,c,2]=255-a[l,c,2]
cv2.imwrite('invertec_py.jpg',a)
```

Como antes, este programa acessa explicitamente os pixels da imagem só para servir de exemplo. Os dois loops poderiam ser substituídos por um único comando:

```
Python: a = 255 – a;
C++:    a = Scalar(255,255,255) - a;
```

5. Detectar pixels com coloração amarelada

Agora, vamos escrever um programa que detecta os pixels com coloração amarelada e pinta-os de vermelho (figura 5). Aqui, não podemos comparar o valor do pixel com um único valor. Em vez disso, precisamos delimitar um região no espaço das cores RGB que representa os pixels com coloração amarela. O programa abaixo define (20, 200, 200) como a cor amarela “típica” e delimita uma esfera de raio 100 em torno desse ponto. Pinta de branco todos os pixels dentro dessa esfera.

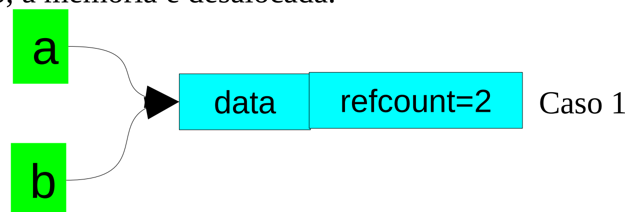
```
 1 //elefante_am.cpp 2024
 2 #include <opencv2/opencv.hpp>
 3 #include <cmath>
 4 using namespace std;
 5 using namespace cv;
 6
 7 int distancia(Vec3b a, Vec3b b) {
 8   return sqrt( pow(a[0]-b[0],2) + pow(a[1]-b[1],2) + pow(a[2]-b[2],2) );
 9 }
10
11 int main() {
12   Mat_<Vec3b> a=imread("elefante.jpg",1);
13   Mat_<Vec3b> b;
14   Vec3b amarelo(20,200,200);
15   //b=a; // Esta errado. b e a sao dois nomes para mesma imagem.
16   b=a.clone(); // OU a.copyTo(b); Aqui temos duas imagens diferentes.
17   for (int l=0; l<a.rows; l++)
18     for (int c=0; c<a.cols; c++)
19       if (distancia(amarelo,a(l,c))<100)
20         b(l,c)=Vec3b(0,0,255);
21   imwrite("elefante_am.png",b);
22 }
```

A função *distancia* na linha 7 calcula a distância euclidiana entre duas cores.

Preste atenção nas linhas 16-17. O comando:

```
b=a;
```

não copia a matriz *a* para *b*, mas cria dois cabeçalhos para o mesmo dado. Em OpenCV/C++, o campo “refcount” controla quantos cabeçalhos estão apontando para um certo dado. Quando refcount tornar-se zero, a memória é desalocada.



Se quiser criar duas cópias independentes de matriz, deve-se dar o comando:

```
b=a.clone();  OU
a.copyTo(b);
```

Caso 2

Isto é assim tanto em OpenCV/C++ como em OpenCV/Python (onde escreveria `b=a` no caso 1 e `b=a.copy()` no caso 2).

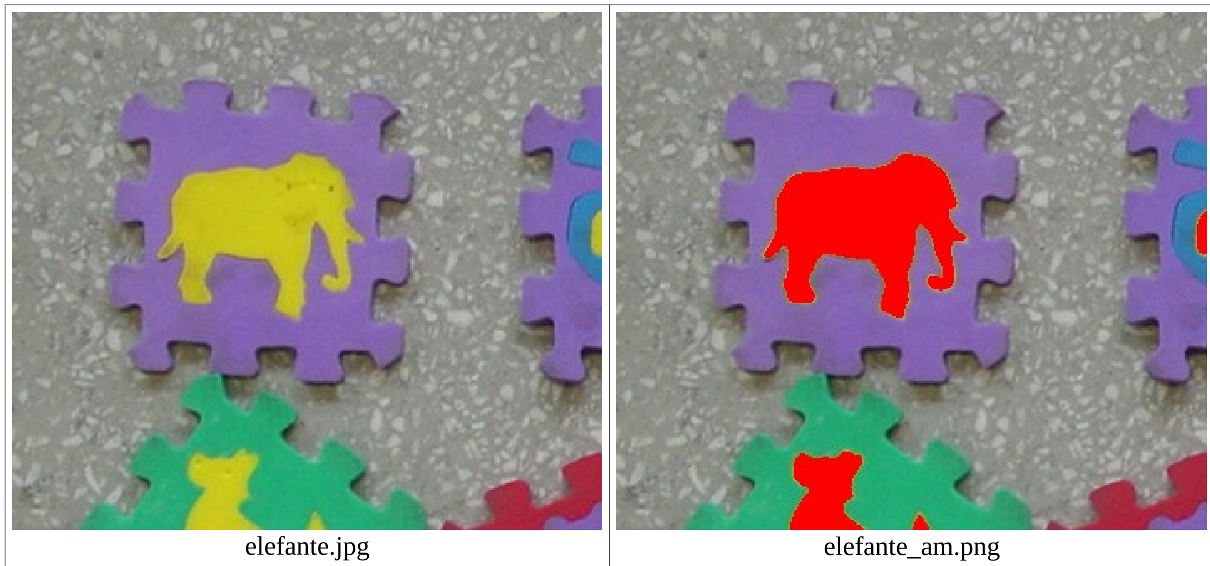
O mesmo programa em Python:

```
# elefante_am.py 2024
import numpy as np
import cv2

def distancia(a, b):
  dist = a-b
  return np.sqrt(np.dot(dist.T,dist))

a=cv2.imread("elefante.jpg",1)
amarelo=(20,200,200);
#b=a; # Esta errado. b e a sao dois nomes para mesma imagem.
b=a.copy(); # Esta certo. Temos duas matrizes b e a distintas.
for l in range(a.shape[0]):
  for c in range(a.shape[1]):
    if distancia(amarelo,a[l,c])<70:
      b[l,c]=(0,0,255)
cv2.imwrite("elefante_am.png",b)
```

A saída do programa está na figura 5.



elefante.jpg	elefante_am.png

Figura 5: Detecção de pixels de coloração amarela.

Exercício: Modifique o programa acima para pintar de preto os pixels com coloração roxa.

O programa “kcek projcor” de Cekeikon projeta as cores de uma imagem nos planos RG, RB e GB.

```
>kcek projcor elefante.jpg rg.png rb.png gb.png
```

Fazendo isso, as imagens da figura 6 são obtidas. Tanto na imagem RB como na imagem GB, fica claro que pixels com coloração amarela podem ser identificadas facilmente, pois estão separadas das demais cores.

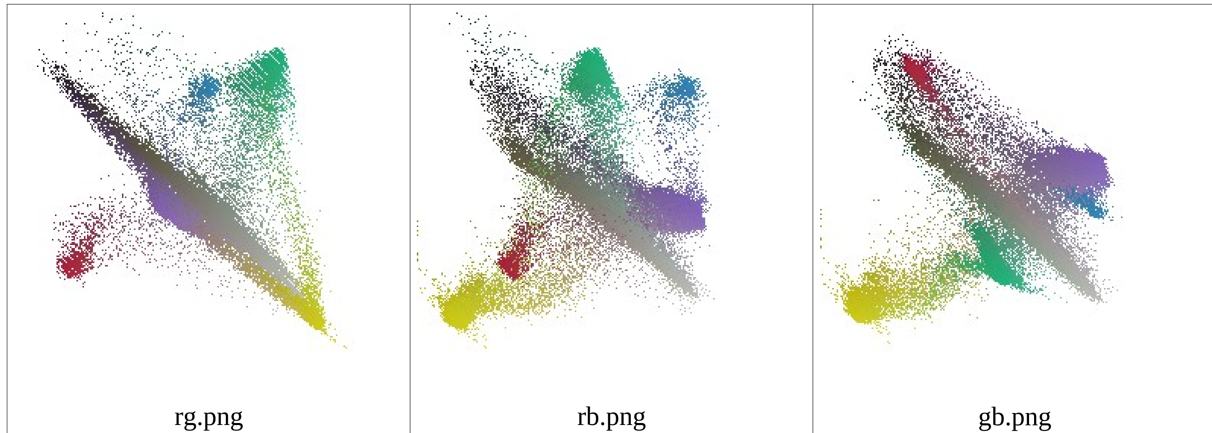


Figura 6: Projeção das cores de elefante.jpg nos planos RG, RB e GB.

Exercício: Quebre uma imagem colorida em componentes RGB, como mostra a figura 7.

Figura 7: Quebra da imagem “flor.jpg” nas componentes R, G e B.

6. Conversão de imagem colorida para níveis de cinzas

Como converter uma imagem colorida para níveis de cinza? A primeira ideia é tirar média das bandas RGB:

```
1) Y = (R+G+B)/3;
```

Só que o olho humano é mais sensível à banda verde e menos sensível à banda azul. Ajustando os pesos para corresponder à percepção humana, temos:

```
2) Y = 0.299*R + 0.587*G + 0.114*B;
```

O programa abaixo testa se há alguma diferença entre usar a fórmula (1) e (2).

```
 1 // rgb2y.cpp - 2016
 2 #include <cekeikon.h>
 3 int main() {
 4   Mat_<COR> a;
 5   le(a,"mandrill.jpg");
 6   Mat_<GRY> gcorreto(a.size());
 7   Mat_<GRY> gmedia(a.size());
 8   Mat_<GRY> gopencv;
 9   for (unsigned i=0; i<a.total(); i++) {
10     gcorreto(i) = round( 0.299*a(i)[2] + 0.587*a(i)[1] + 0.114*a(i)[0] );
11     gmedia(i)   = round( (a(i)[0]+a(i)[1]+a(i)[2])/3.0 );
12   }
13   cvtColor(a,gopencv,CV_BGR2GRAY);
14   imp(gcorreto,"gcorreto.pgm");
15   imp(gmedia,"gmedia.pgm");
16   imp(gopencv,"gopencv.pgm");
17 }
```

A imagem colorida mandrill.jpg é convertida para níveis de cinza usando 3 métodos:

1) gmedia usa a fórmula (1).
2) gcorreto usa a fórmula (2).
3) gopencv usa a função cvtColor com opção `CV_BGR2GRAY` para fazer a conversão.

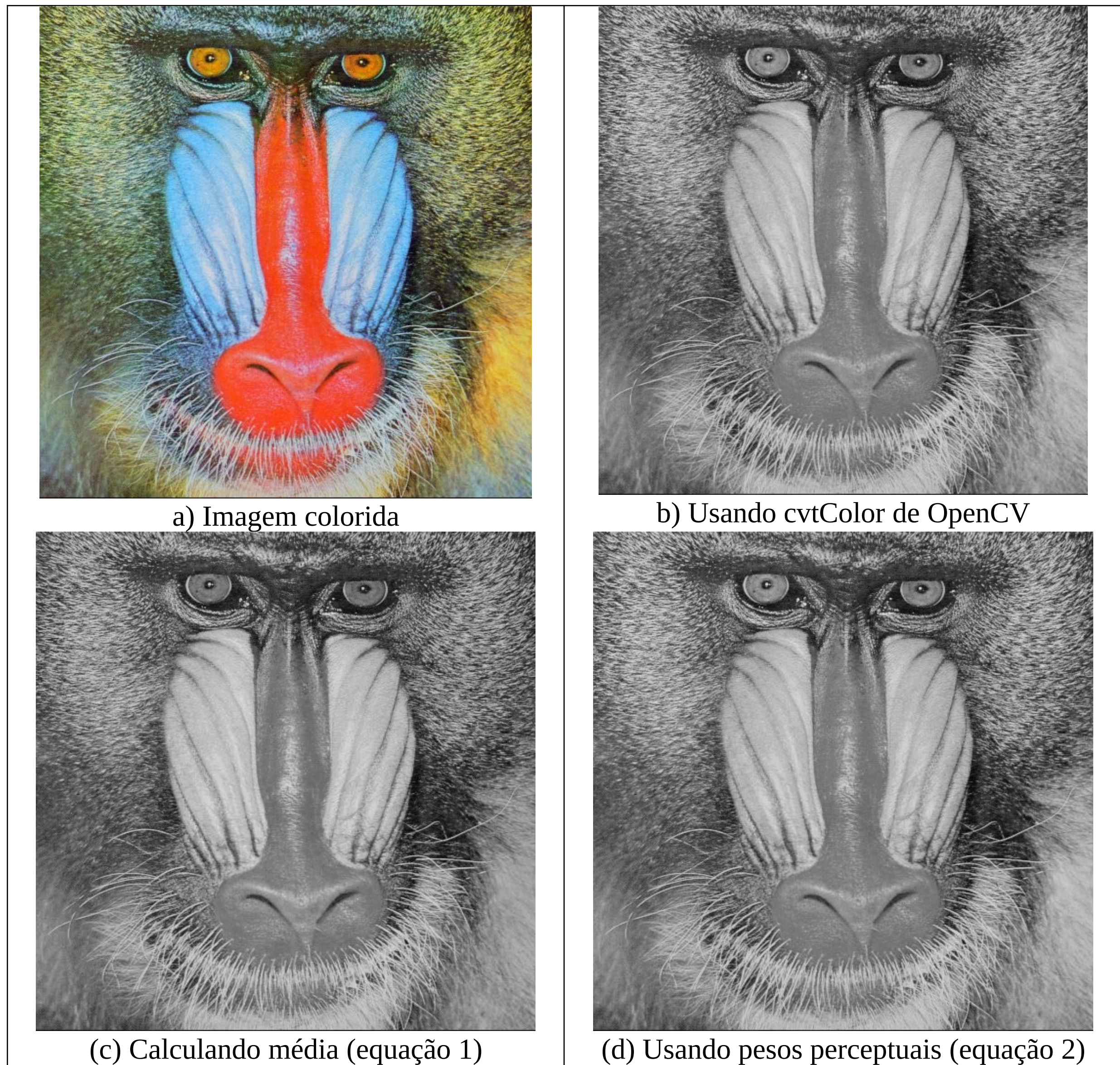
As imagens obtidas pelas fórmulas (1) e (2) estão na figura 8. É difícil enxergar alguma diferença entre elas.

Agora, vamos medir a diferença entre essas imagens e a imagem obtida usando a função cvtColor do OpenCV (linha 13 do código). O programa "kcek distg" do Cekeikon calcula a diferença entre duas imagens. MAE significa "mean absolute error". A maior média da diferença absoluta entre duas imagens em níveis de cinza é 255 (diferença entre uma imagem completamente branca e outra completamente preta). Esta diferença foi dividida por 255 e expressa como porcentagem. Os resultados foram:

```
>kcek distg gopencv.pgm gcorreto.pgm
MAE (max=100%)...................:                 0.00%

>kcek distg gopencv.pgm gmedia.pgm
MAE (max=100%)...................:                 2.39%
```

Isto é, a função cvtColor do OpenCV usa exatamente a fórmula (2) para converter imagem colorida para níveis de cinzas.



a) Imagem colorida

b) Usando cvtColor de OpenCV

(c) Calculando média (equação 1)

(d) Usando pesos perceptuais (equação 2)

Figura 8: Conversão de imagem colorida para níveis de cinzas usando diferentes métodos.

7. Sistemas de cores

7.1 CMY

As cores podem ser representadas em diferentes sistemas de cores. Já vimos o sistema RGB, que é um sistema aditivo (representa soma de luzes de cores básicas). As luzes de cores R, G e B vão se somando em diferentes proporções para formar diferentes cores. Se somar todas as luzes RGB resulta cor branca e se não colocar nenhuma luz resulta cor preta (figura 9).

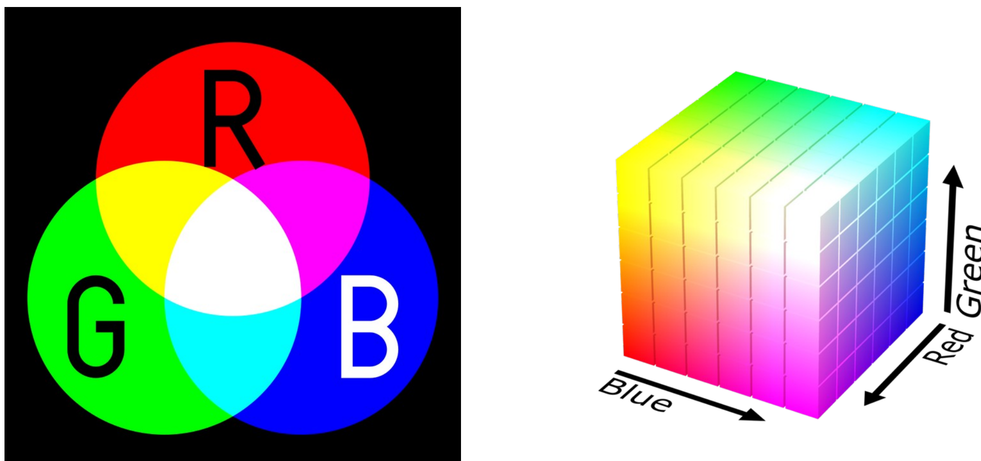


Figura 9: Sistema RGB é aditivo (fonte: Wikipedia).

Já os sistemas CMY (cyan, magenta, yellow) e CMYK (CMY + black) são subtrativos, usado para tintas. Se misturarmos todas as tintas CMY obtemos preto enquanto que se não jogarmos nenhuma tinta no papel temos branco (figura 10a). O sistema CMYK acrescenta a tinta preta, pois misturando tintas CMY normalmente não obtém preto perfeito e porque a tinta preta permite economizar tintas coloridas.

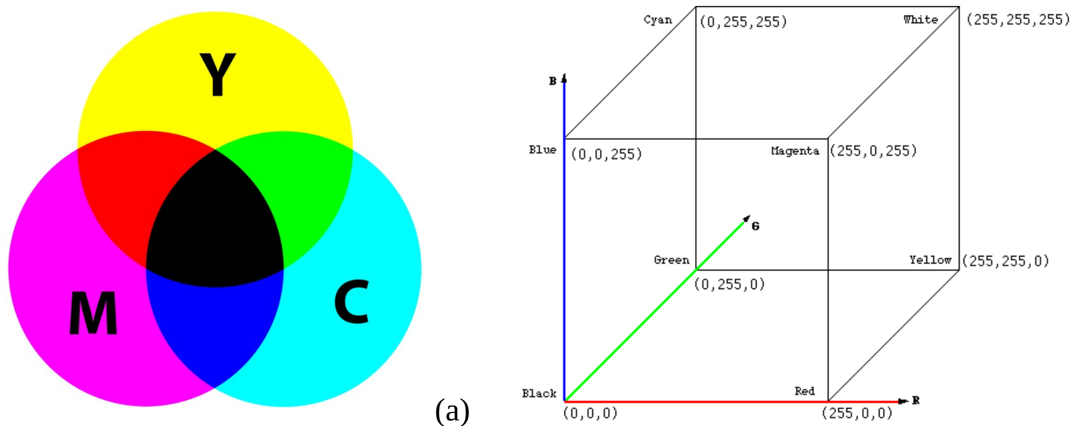


Figura 10: (a) Sistema CMY e CMYK são subtrativos. (b) Cubo de cores RGB e CMY (fonte: Wikipedia).

Veja na figura 10b que qualquer cor dentro do cubo de cores pode ser especificada fornecendo as coordenadas RGB a partir do centro do sistema de coordenadas preto [0,0,0] ou fornecendo as coordenadas CMY a partir do centro do sistema de coordenadas branco [255,255,255]. Por exemplo, a cor RGB=[100, 0, 0] seria CMY=[155, 255, 255].

Exercício: Converta a cor RGB=[50,100,150] para sistema CMY.

7.2 HSI ou HSL

Existem muitos outros sistemas de cores. Sistema de cores HSL, HLS ou HSI representa uma cor através dos atributos hue, saturation e lightness. Hue (coloração) é um ângulo, onde 0° representa vermelho, 120° representa verde e 240° representa azul. Saturação indica se a cor é pura (saturada, na borda do bicone da figura 11) ou tende para tonalidades de cinza (insaturada, perto do bicone da figura 11). No eixo central do bicone da figura 11 temos as cores completamente insaturadas, que correspondem às tonalidades de cinza. Por fim, lightness indica aproximadamente a cor convertida para níveis de cinza.

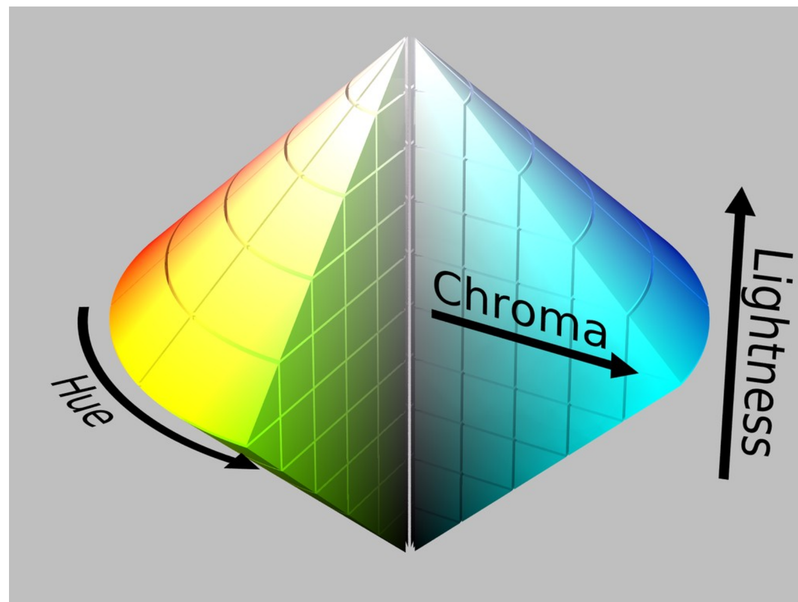


Figura 11: Sistema de cores HSL.

O programa abaixo a imagem colorida mandrill.jpg para sistema HSL e imprime as 3 bandas obtidas. A rotina cvtColor faz esta conversão, com a opção `CV_BGR2HLS`. Em OpenCV, a banda H vai de 0 a 180 (quando armazenada numa imagem 8 bits sem sinal), representando 0 a 360 graus. Os componentes L e S vão de 0 a 255.

```
// separa pos-2012
#include <cekeikon.h>
int main()
{ Mat_<COR> a;
  le(a,"mandrill.jpg");
  cvtColor(a, a, CV_BGR2HLS);
  // a(l,c)[0] = hue
  // a(l,c)[1] = lightness
  // a(l,c)[2] = saturation
  Mat_<GRY> h(a.rows,a.cols); //entre 0 e 180
  Mat_<GRY> s(a.rows,a.cols);
  Mat_<GRY> i(a.rows,a.cols);
  for (int l=0; l<a.rows; l++)
    for (int c=0; c<a.cols; c++) {
      h(l,c)=a(l,c)[0];
      s(l,c)=a(l,c)[2];
      i(l,c)=a(l,c)[1];
    }
  imp(h,"hue.tga"); imp(s,"sat.tga"); imp(i,"int.tga");
}
```

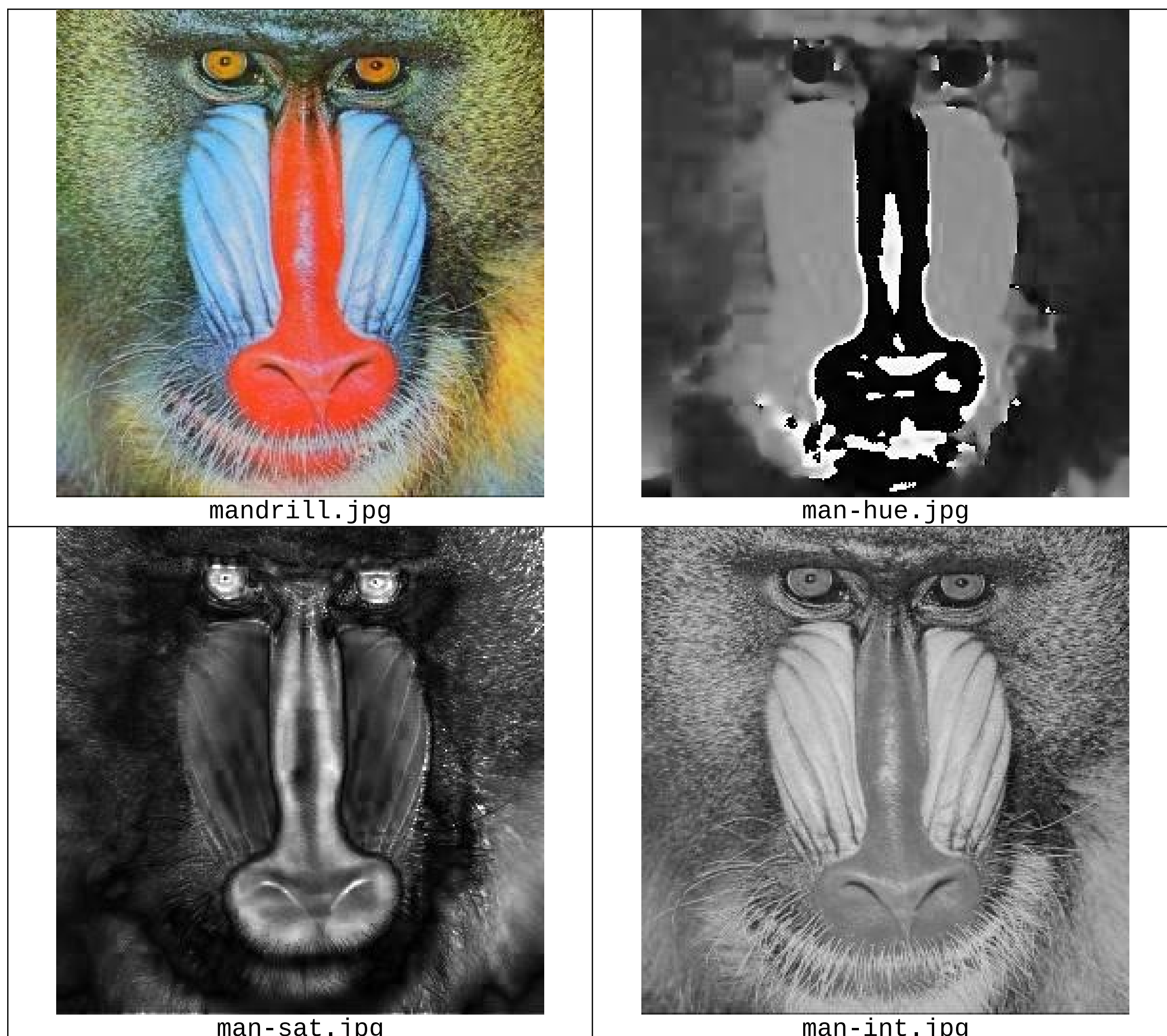


Figura 12: Imagem mandrill convertido para sistema HSI (ou HSL).

Veja que na imagem coloração (hue), dentro do nariz do macaco, há transição súbita de preto para cinza claro. Isto acontece pois hue é ângulo e vermelho é representado por 0° (valor 0) ou 360° (valor 180). Uma pequena mudança de tonalidade de vermelho pode fazer hue mudar de valor próximo de zero para valor próximo de 180.

Na banda saturação, as cores puras (amarelo do olho, vermelho do nariz) estão representadas em cinza clara. As cores tendendo para branco, cinza ou preto (bochecha azul-celeste, pelo verde-escuro e barba branca) são representadas em cinza escuro.

Por fim, intensidade é praticamente a imagem colorida convertida para níveis de cinzas.

Exercício: Considere sistema HSI onde H pode ir de 0 a 180 (respectivamente representando 0 graus e 360 graus como no OpenCV). Que cores representam as seguintes coordenadas HSI:

H=60 S=0 I=255 representa a cor _____________
H=0 S=0 I=128 representa a cor _____________
H=0 S=255 I=128 representa a cor _____________
H=120 S=255 I=128 representa a cor _____________