

# Introdução a Visão Computacional com Python e OpenCV

Versão 0.8 – Não corrigida

Ricardo Antonello

[www.antonello.com.br](http://www.antonello.com.br)

# O autor

Ricardo Antonello é mestre em Ciência da Computação pela Universidade Federal de Santa Catarina – UFSC e bacharel em Ciência da Computação pelo Centro Universitário de Brasília – UniCEUB. Possui as certificações Java Sun Certified Java Programmer - SCJP e Sun Certified Web Component Developer - SCWCD. Em 2000 iniciou sua carreira em instituições de grande porte do mercado financeiro no Brasil e desde 2006 é professor universitário. Na Universidade do Oeste de Santa Catarina - Unoesc foi coordenador do Núcleo de Inovação Tecnológica – NIT e da Pré-Incubadora de Empresas. Também atuou como coordenador das atividades do Polo de Inovação Vale do Rio do Peixe – Inovale. Atualmente é professor de Linguagens de Programação em regime de dedicação exclusiva no Instituto Federal Catarinense – IFC, câmpus Luzerna. Trabalha em projetos de pesquisa e extensão nas áreas de inteligência artificial, processamento de imagens e robôs autônomos.

Contato: ricardo@antonello.com.br ou ricardo.antonello@luzerna.ifc.edu.br

Mais informações no blog: [www.antonello.com.br](http://www.antonello.com.br)

# Prefácio

Os cinco sentidos dos seres humanos são: olfato, tato, audição, paladar e visão. De todos eles, temos que concordar que a visão é o mais, ou pelo menos, um dos mais importantes.

Por este motivo, “dar” o sentido da visão para uma máquina gera um resultado impressionante. Imagens estão em todo o lugar e a capacidade de reconhecer objetos, paisagens, rostos, sinais e gestos torna as máquinas muito mais úteis.

É nesse sentido que explicamos a importância deste livro, que foi criado para ser utilizado em uma disciplina optativa chamada “Tópicos especiais em visão computacional” lecionada por mim no curso de Engenharia de Controle e Automação do Instituto Federal Catarinense – IFC, *campus Luzerna*.

Existe muita literatura a respeito do tema em inglês mas em português temos pouco material, então a primeira sugestão ao leitor é que APRENDA inglês o quanto antes. Porém, dada a imcapacidade de muitos alunos em lerem fluentemente na língua inglesa, surgiu a necessidade de criar esta obra onde aproveitei para incluir minha experiência prática com algoritmos e projetos de pesquisa sobre visão computacional.

Aproveito para agradecer ao Dr. Roberto de Alencar Lotufo da Unicamp onde fiz meu primeiro curso de extensão em Visão Computacional e ao Adrian Rosebrock que além de PhD em Ciência da Computação pela Universidade de Marylan, mantém o site [www.pyimagesearch.com](http://www.pyimagesearch.com) que me ajudou muito a ver exemplos sobre o tema.

**Ricardo Antonello**

# Sumário

1 Bem vindo ao mundo da visão computacional .....	6
2 Sistema de coordenadas e manipulação de pixels .....	9
3 Fatiamento e desenho sobre a imagem .....	12
4 Transformações e máscaras .....	16
4.1 Cortando uma imagem / Crop .....	16
4.2 Redimensionamento / Resize.....	16
4.3 Espelhando uma imagem / Flip .....	18
4.4 Rotacionando uma imagem / Rotate.....	19
4.5 Máscaras .....	20
5 Sistemas de cores.....	23
5.1 Canais da imagem colorida.....	24
6 Histogramas e equalização de imagem.....	26
6.1 Equalização de Histograma .....	29
7 Suavização de imagens .....	33
7.1 Suavização por cálculo da média.....	33
7.2 Suavização pela Gaussiana.....	34
7.3 Suavização pela mediana.....	35
7.4 Suavização com filtro bilateral .....	36
8 Binarização com limiar.....	38
8.1 Threshold adaptativo .....	39
8.2 Threshold com Otsu e Riddler-Calvard.....	40
9 Segmentação e métodos de detecção de bordas .....	41
9.1 Sobel .....	41
9.2 Filtro Laplaciano.....	42
9.3 Detector de bordas Canny.....	43
10 Identificando e contando objetos.....	45
11 Detecção de faces em imagens .....	49
12 Detecção de faces em vídeos .....	53
13 Rastreamento de objetos em vídeos.....	55
14 Reconhecimento de caracteres.....	56
14.1 Biblioteca PyTesseract .....	56
14.2 Padronização de placas .....	56
14.3 Filtros do OpenCV.....	57

14.4 Criação do dicionário.....	58
14.5 jTessBoxEditor .....	58
14.6 Serak Tesseract Trainer .....	58
14.7 Resultados.....	60
15 Treinamento para identificação de objetos por Haar Cascades .....	63
15.1 Coletando o bando de dados de imagens .....	63
15.2 Organizando as imagens negativas.....	63
15.3 Recortar e marcar imagens “positivas” .....	64
15.4 Criando um vetor de imagens “positivas” .....	64
15.5 Haar-Training .....	65
15.6 Criando o arquivo XML .....	67
16 Criação de um identificador de objetos com Haar Cascades.....	68

# 1 Bem vindo ao mundo da visão computacional

No prefácio, que recomendo que você leia, já iniciamos as explicações do que é visão computacional. Mas vale trazer uma definição clássica: “Visão computacional é a ciência e tecnologia das máquinas que enxergam. Ela desenvolve teoria e tecnologia para a construção de sistemas artificiais que obtém informação de imagens ou quaisquer dados multi-dimensionais”.

Sim a definição é da wikipédia e deve já ter sido alterada, afinal o campo de estudos sobre visão computacional esta em constante evolução. Neste ponto é importante frizar que, além de fazer máquinas “encherarem”, reconhecerem objetos, paisagens, gestos, faces e padrões, os mesmos algoritmos podem ser utilizados para reconhecimento de padrões em grandes bases de dados, não necessariamente feitos de imagens.

Porém quando se trata de imagens, temos avanços significativos já embutidos em muitos apps e outros sistemas que usamos. O Facebook já reconhece objetos automaticamente para classificar suas fotos, além disso, já aponta onde estão as pessoas na imagem para você “marcar”. O mesmo ocorre com smartphones que já disparam a foto quando as pessoas estiverem sorrindo, pois conseguem reconhecer tais expressões.

Além disso, outros sistemas de reconhecimento como o chamado popularmente “OCR” de placas de veículos já estão espalhados pelo Brasil, onde as imagens dos veículos são capturadas, a placa reconhecida e convertida para textos e números que podem ser comparados diretamente com banco de dados de veículos roubados ou com taxas em atraço. Enfim, a visão computacional já está aí!

Não vou entrar no mérito de discutir o que é visão computacional e sua diferente com outro termo muito utilizado que é processamento de imagens. Leia sobre isso na web e tire suas próprias conclusões. Mas minha opinião é que passar um filtro para equalizar o histograma da imagem esta relacionado com processamento de imagem. Já reconhecer um objeto existente em uma foto esta relacionado a visão computacional.

Vamos utilizar a linguagem Python neste livro, especificamente a versão 3.4 juntamente com a biblioteca OpenCV versão 3.0 que você pode baixar e configurar através de vários tutoriais na internet. Ao final do livro temos um apêndice com as instruções detalhadas.

Algumas bibliotecas Python também são necessárias como Numpy (Numeric Python), Scipy (Cientific Python) e Matplotlib que é uma biblioteca para plotagem de gráficos com sintaxe similar ao Matlab.

Utilizamos nas imagens do livro o sistema operacional Linux/Ubuntu rodando em uma máquina virtual com Oracle Virtual Box o que recomendo fortemente para facilitar a configuração uma única vez e a utilização para testes em várias outras máquinas já que a configuração do ambiente é trabalhosa.

Feita a configuração descrita no Apêndice A vamos rodar nosso primeiro programa. Um “Alô mundo!” da visão computacional onde iremos apenas abrir uma arquivo de imagem do disco e exibi-lo na tela. Feito isso o código espero o pressionamento de uma tecla para fechar a janela e encerrar o programa.

```

# Importação das bibliotecas
import cv2

# Leitura da imagem com a função imread()
imagem = cv2.imread('entrada.jpg')

print('Largura em pixels: ', end='')
print(imagem.shape[1]) #largura da imagem
print('Altura em pixels: ', end='')
print(imagem.shape[0]) #altura da imagem
print('Qtde de canais: ', end='')
print(imagem.shape[2])

# Mostra a imagem com a função imshow
cv2.imshow("Nome da janela", imagem)
cv2.waitKey(0) #espera pressionar qualquer tecla

# Salvar a imagem no disco com função imwrite()
cv2.imwrite("saída.jpg", imagem)

```

Este programa abre uma imagem, mostra suas propriedades de largura e altura em pixels, mostra a quantidade de canais utilizados, mostra a imagem na tela, espera o pressionar de alguma tecla para fechar a imagem e salva em disco a mesma imagem com o nome ‘saída.jpg’. Vamos explicar o código em detalhes abaixo:

```

# Importação das bibliotecas
import cv2

# Leitura da imagem com a função imread()
imagem = cv2.imread('entrada.jpg')

```

A importação da biblioteca padrão da OpenCV é obrigatória para utilizar suas funções. A primeira função usada é para abrir a imagem através de cv2.imread() que leva como argumento o nome do arquivo em disco.

A imagem é lida e armazenada em ‘imagem’ que é uma variável que dará acesso ao objeto da imagem que nada mais é que uma matriz de 3 dimensões (3 canais) contendo em cada dimensão uma das 3 cores do padrão RGB (red=vermelho, green-verde, blue=azul). No caso de uma imagem preto e branca temos apenas um canal, ou seja, apenas uma matriz de 2 dimensões.

Para facilitar o entendimento podemos pensar em uma planilha eletrônica, com linhas e colunas, portanto, uma matriz de 2 dimensões. Cada célula dessa matriz é um pixel, que no caso de imagens preto e brancas possuem um valor de 0 a 255, sendo 0 para preto e 255 para branco. Portanto, cada célula contém um inteiro de 8 bits (sem sinal) que em Python é definido por “uint8” que é um *unsigned integer* de 8 bits.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R		
1				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
2				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
3				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
4				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
5				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
6				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
7				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
8				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
9				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255
10				15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255

**Figura 1** Imagem preto e branca representada em uma matriz de inteiros onde cada célula é um inteiro sem sinal de 8 bits que pode conter de 0 (preto) até 255 (branco). Perceba os vários tons de cinza nos valores intermediários como 30 (cinza escuro) e 210 (cinza claro).

No caso de imagens preto e branca é composta de apenas uma matriz de duas dimensões como na imagem acima. Já para imagens coloridas temos três dessas matrizes de duas dimensões cada uma representando uma das cores do sistema RGB. Portanto, cada pixel é formado de uma tupla de 3 inteiros de 8 bits sem sinal no sistema (R,G,B) sendo que (0,0,0) representa o preto, (255,255,255) o branco. Nesse sentido, as cores mais comuns são:

- Branco - RGB (255,255,255);
  - Azul - RGB (0,0,255);
  - Vermelho - RGB (255,0,0);
  - Verde - RGB (0,255,0);
  - Amarelo - RGB (255,255,0);
  - Magenta - RGB (255,0,255);
  - Ciano - RGB (0,255,255);
  - Preto - RGB (0,0,0).

As imagens coloridas, portanto, são compostas normalmente de 3 matrizes de inteiros sem sinal de 8 bits, a junção das 3 matrizes produz a imagem colorida com capacidade de reprodução de 16,7 milhões de cores, sendo que os 8 bits tem capacidade para 256 valores e elevando a 3 temos  $256^3 = 16,7$  milhões.

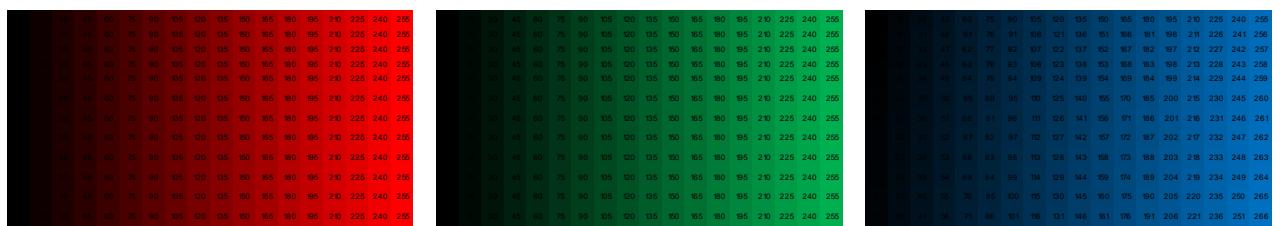


Figura 2 Na imagem temos um exemplo das 3 matrizes que compõe o sistema RGB. Cada pixel da imagem, portanto, é composto por 3 componentes de 8 bits cada, sem sinal, o que gera 256 combinações por cor. Portanto, a representação é de 256 vezes 256 vezes 256 ou  $256^3$  que é igual a 16,7 milhões de cores.

## 2 Sistema de coordenadas e manipulação de pixels

Conforme vimos no capítulo anterior, temos uma representação de 3 cores no sistema RGB para cada pixel da imagem colorida. Podemos alterar a cor individualmente para cada pixel, ou seja, podemos manipular individualmente cada pixel da imagem.

Para isso é importante entender o sistema de coordenadas (linha, coluna) onde o pixel mais a esquerda e acima da imagem está na posição (0,0) está na linha zero e coluna zero. Já em uma imagem com 300 pixels de largura, ou seja, 300 colunas e tendo 200 pixels de altura, ou seja, 200 linhas, terá o pixel (199,299) como sendo o pixel mais a direita e abaixo da imagem.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
L0	0	0	0	0	0	0	0	0	0	0
L1	0	50	50	50	50	50	50	50	50	0
L2	0	50	100	100	100	100	100	100	50	0
L3	0	50	100	150	150	150	150	100	50	0
L4	0	50	100	150	200	200	150	100	50	0
L5	0	50	100	150	200	200	150	100	50	0
L6	0	50	100	150	150	150	150	100	50	0
L7	0	50	100	100	100	100	100	100	50	0
L8	0	50	50	50	50	50	50	50	50	0
L9	0	0	0	0	0	0	0	0	0	0

**Figura 3** O sistema de coordenadas envolve uma linha e coluna. Os índices iniciam em zero. O pixel (2,8), ou seja, na linha índice 2 que é a terceira linha e na coluna índice 8 que é a nona linha possui a cor 50.

A partir do entendimento do sistema de coordenadas é possível alterar individualmente cada pixel ou ler a informação individual do pixel conforme abaixo:

```
import cv2
imagem = cv2.imread('ponte.jpg')
(b, g, r) = imagem[0, 0] #veja que a ordem BGR e não RGB
```

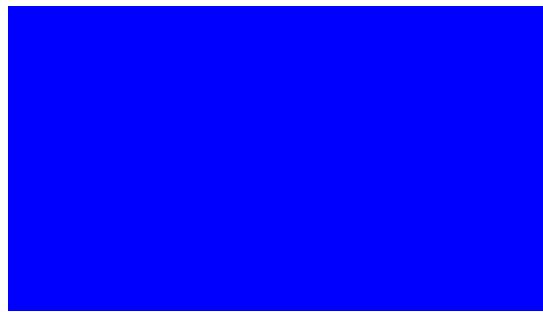
Imagens são matrizes Numpy neste caso retornadas pelo método “imread” e armazenada em memória através da variável “imagem” conforme acima. Lembre-se que o pixel superior mais a esquerda é o (0,0). No código é retornado na tupla (b, g, r) os respectivos valores das cores do píxel superior mais a esquerda. Veja que o método retorna a sequência BGR e não RGB como poderíamos esperar. Tendo os valores inteiros de cada cor é possível exibi-los na tela com o código abaixo:

```
print('O pixel (0, 0) tem as seguintes cores:')
print('Vermelho:', r, 'Verde:', g, 'Azul:', b)
```

Outra possibilidade é utilizar dois laços de repetição para “varrer” todos os pixels da imagem, linha por linha como é o caso do código abaixo. Importante notar que esta estratégia pode não ser muito performática já que é um processo lento varrer toda a imagem pixel a pixel.

```
import cv2
imagem = cv2.imread('ponte.jpg')
for y in range(0, imagem.shape[0]):
    for x in range(0, imagem.shape[1]):
        imagem[y, x] = (255,0,0)
cv2.imshow("Imagen modificada", imagem)
```

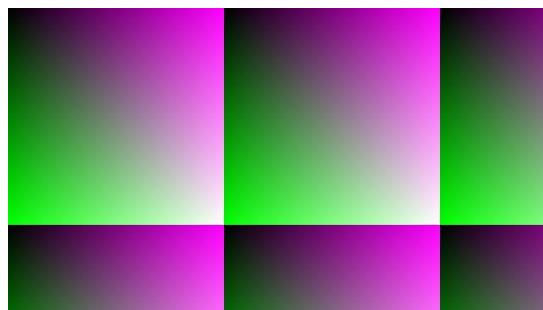
O resultado é uma imagem com todos os pixels substituídos pela cor azul (255,0,0):



**Figura 4 Imagem completamente azul pela alteração de todos os pixels para (255,0,0). Lembrando que o padrão RGB é na verdade BRG pela tupla (B, R, G).**

Outro código interessante segue abaixo onde incluímos as variáveis de linha e coluna para serem as componentes de cor, lembrando que as variáveis componentes da cor devem assumir o valor entre 0 e 255 então utilizamos a operação “resta da divisão por 256” para manter o resultado entre 0 e 255.

```
import cv2
imagem = cv2.imread('ponte.jpg')
for y in range(0, imagem.shape[0]): #percorre linhas
    for x in range(0, imagem.shape[1]): #percorre colunas
        imagem[y, x] = (x%256, y%256, x%256)
cv2.imshow("Imagen modificada", imagem)
cv2.waitKey(0)
```

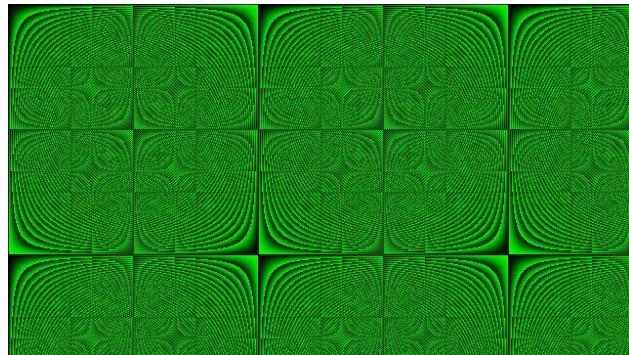


**Figura 5 A alteração nas componentes das cores da imagem conforme as coordenadas de linha e coluna geraram a imagem acima.**

Alterando minimamente o código, especificamente no componente “green”, temos a

imagem abaixo. Veja que utilizamos os valores de linha multiplicado pela coluna ( $x*y$ ) no componente “G” da tupla que forma a cor de cada pixel e deixamos o componente azul e vermelho zerados. A dinâmica da mudança de linhas e colunas gera esta imagem.

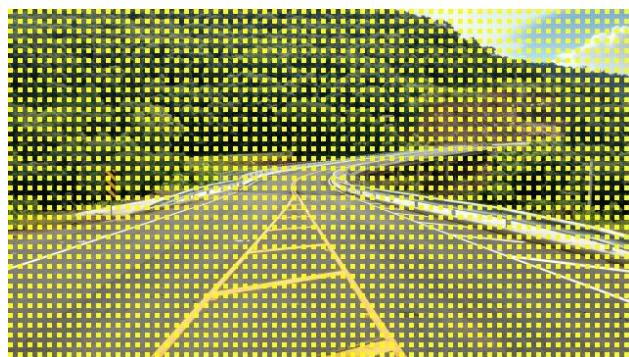
```
import cv2
imagem = cv2.imread('ponte.jpg')
for y in range(0, imagem.shape[0], 1): #percorre as linhas
    for x in range(0, imagem.shape[1], 1): #percorre as colunas
        imagem[y, x] = (0, (x*y)%256, 0)
cv2.imshow("Imagen modificada", imagem)
cv2.waitKey(0)
```



**Figura 6** A alteração dinâmica da cor de cada pixel gera esta bela imagem.  
A imagem original se perdeu pois todos os pixels foram alterados.

Com mais uma pequena modificação temos o código abaixo. O objetivo agora é saltar a cada 10 pixels ao percorrer as linhas e mais 10 pixels ao percorrer as colunas. A cada salto é criado um quadrado amarelo de 5x5 pixels. Desta vez parte da imagem original é preservada e podemos ainda observar a ponte por baixo da grade de quadrados amarelos.

```
import cv2
imagem = cv2.imread('ponte.jpg')
for y in range(0, imagem.shape[0], 10): #percorre linhas
    for x in range(0, imagem.shape[1], 10): #percorre colunas
        imagem[y:y+5, x: x+5] = (0, 255, 255)
cv2.imshow("Imagen modificada", imagem)
cv2.waitKey(0)
```



**Figura 7** Código gerou quadrados amarelos de 5x5 pixels sobre a toda a imagem.

Aproveite e crie suas próprias fórmulas e gere novas imagens.

### 3 Fatiamento e desenho sobre a imagem

No capítulo anterior vimos que é possível alterar um único pixel da imagem com o código abaixo:

```
imagem[0, 0] = (0, 255, 0) #altera o pixel (0,0) para verde
```

No caso acima o primeiro pixel da imagem terá a cor verde. Também podemos utilizar a técnica de “slicing” para alterar vários pixels da imagem de uma única vez como abaixo:

```
image[30:50, :] = (255, 0, 0)
```

Este código acima cria um retângulo azul a partir da linha 31 até a linha 50 da imagem e ocupa toda a largura disponível, ou seja, todas as colunas.

O código abaixo abre uma imagem e cria vários retângulos coloridos sobre ela.

```
import cv2
image = cv2.imread('ponte.jpg')

#Cria um retângulo azul por toda a largura da imagem
image[30:50, :] = (255, 0, 0)

#Cria um quadrado vermelho
image[100:150, 50:100] = (0, 0, 255)

#Cria um retângulo amarelo por toda a altura da imagem
image[:, 200:220] = (0, 255, 255)

#Cria um retângulo verde da linha 150 a 300 nas colunas 250 a 350
image[150:300, 250:350] = (0, 255, 0)

#Cria um quadrado ciano da linha 150 a 300 nas colunas 250 a 350
image[300:400, 50:150] = (255, 255, 0)

#Cria um quadrado branco
image[250:350, 300:400] = (255, 255, 255)

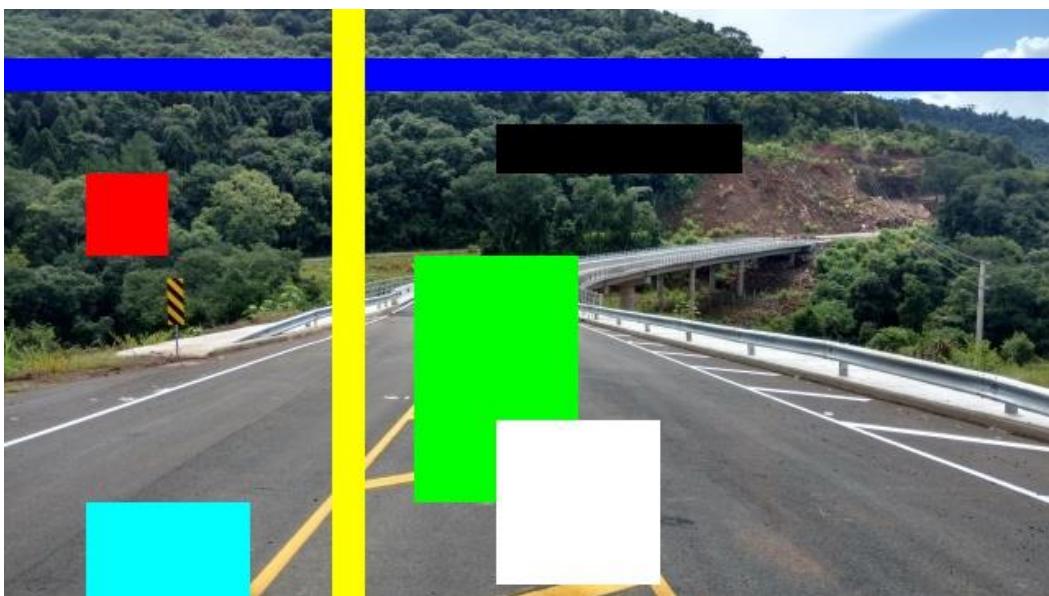
#Cria um quadrado preto
image[70:100, 300:450] = (0, 0, 0)

cv2.imshow("Imagen alterada", image)
cv2.imwrite("alterada.jpg", image)
cv2.waitKey(0)
```

Vários retângulos coloridos são criados sobre a imagem com o código acima. Veja a seguir a imagem original ponte.jpg e a imagem alterada.



**Figura 8 Imagem original.**



**Figura 9 Imagem após a execução do código acima com os retângulos coloridos incluídos pela técnica de “slicing”.**

Com a técnica de slicing é possível criar quadrados e retângulos, contudo, para outras formas geométricas é possível utilizar as funções da OpenCV, isso é útil principalmente no caso de desenho de círculos e textos sobre a imagem, veja:

```
import numpy as np
import cv2

imagem = cv2.imread('ponte.jpg')

vermelho = (0, 0, 255)
verde = (0, 255, 0)
azul = (255, 0, 0)
```

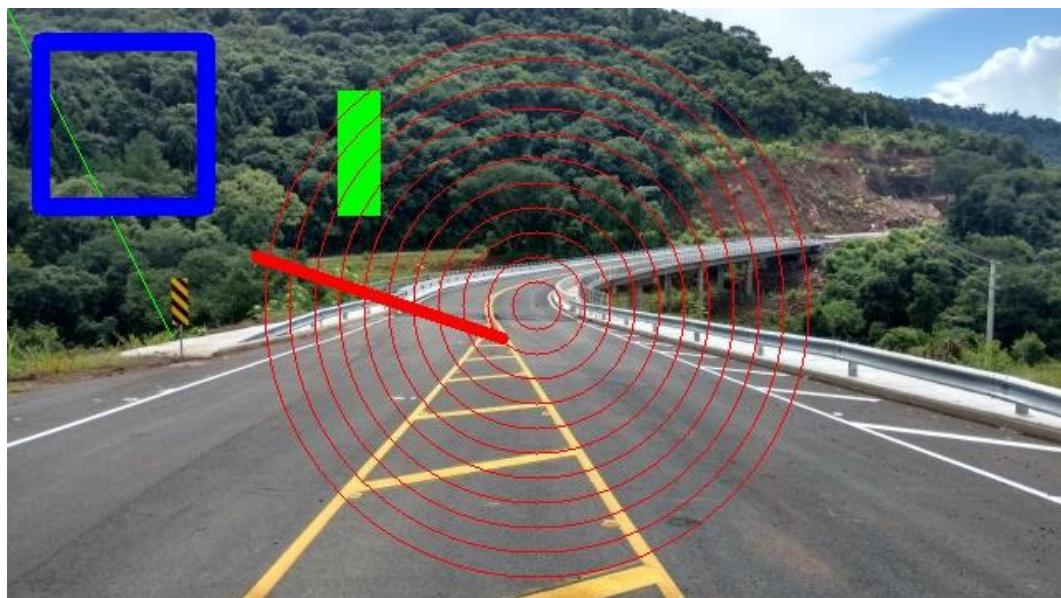
```

cv2.line(imagem, (0, 0), (100, 200), verde)
cv2.line(imagem, (300, 200), (150, 150), vermelho, 5)
cv2.rectangle(imagem, (20, 20), (120, 120), azul, 10)
cv2.rectangle(imagem, (200, 50), (225, 125), verde, -1)

(X, Y) = (imagem.shape[1] // 2, imagem.shape[0] // 2)
for raio in range(0, 175, 15):
    cv2.circle(imagem, (X, Y), raio, vermelho)

cv2.imshow("Desenhando sobre a imagem", imagem)
cv2.waitKey(0)

```



**Figura 10 Utilizando funções do OpenCV para desenhar sobre a imagem.**

Outra função muito útil é a de escrever textos sobre a imagem. Para isso lembre-se que a coordenada do texto se refere a base onde os caracteres começaram a serem escritos. Então para um calculo preciso estude a função ‘getTextSize’. O exemplo abaixo tem o resultado a seguir:

```

import numpy as np
import cv2

imagem = cv2.imread('ponte.jpg')

fonte = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(imagem,'OpenCV',(15,65), fonte,
2,(255,255,255),2,cv2.LINE_AA)

cv2.imshow("Ponte", imagem)
cv2.waitKey(0)

```



**Figura 11 exemplo de escrita sobre a iamgem. é possível escolher fonte, tamanho e posição.**

## 4 Transformações e máscaras

Em muitas ocasiões é necessário realizar transformações sobre a imagem. Ações como redimensionar, cortar ou rotacionar uma imagem são necessariamente frequentes. Esse processamento pode ser feito de várias formas como vereamos nos exemplos a seguir.

### 4.1 Cortando uma imagem / Crop

A mesma técnica já usada para fazer ‘slicing’ pode ser usada para criar uma nova imagem ‘recortada’ da imagem original, o termo em inglês é ‘crop’. Veja o código abaixo onde criamos uma nova imagem a partir de um pedaço da imagem original e a salvamos no disco.

```
import cv2
imagem = cv2.imread('ponte.jpg')
recorte = imagem[100:200, 100:200]
cv2.imshow("Recorte da imagem", recorte)
cv2.imwrite("recorte.jpg", recorte) #salva no disco
```

Usando a mesma imagem ponte.jpg dos exemplos anteriores, temos o resultado abaixo que é da linha 101 até a linha 200 na coluna 101 até a coluna 200:



**Figura 12 Imagem recortada da imagem original e salvada em um arquivo em disco.**

### 4.2 Redimensionamento / Resize

Para reduzir ou aumentar o tamanho da imagem, existe uma função já pronta kda OpenCV, trata-se da função ‘resize’ mostrada abaixo. Importante notar que é preciso calcular a proporção da altura em relação a largura da nova imagem, caso contrário ela poderá ficar distorcida.

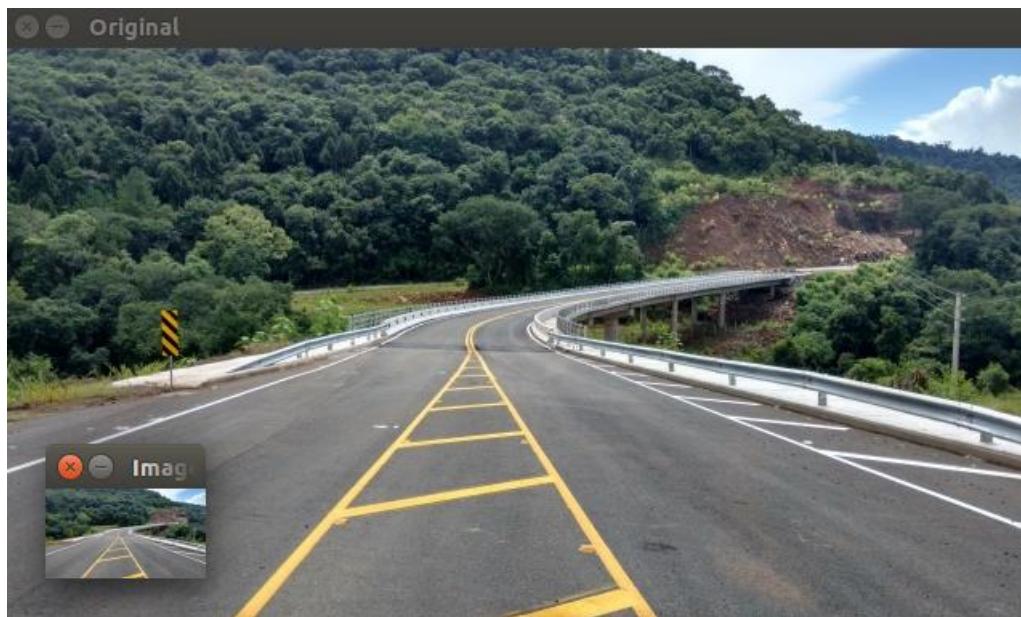
```
import numpy as np
import cv2

img = cv2.imread('ponte.jpg')
cv2.imshow("Original", img)
largura = img.shape[1]
altura = img.shape[0]
proporcao = float(altura/largura)
largura_nova = 320 #em pixels
altura_nova = int(largura_nova*proporcao)
```

```
tamanho_novo = (largura_nova, altura_nova)

img_redimensionada = cv2.resize(img,
    tamanho_novo, interpolation = cv2.INTER_AREA)

cv2.imshow('Resultado', img_redimensionada)
cv2.waitKey(0)
```



**Figura 13** No canto inferior esquerdo da imagem é possível notar a imagem redimensionada.

Veja que a função ‘rezise’ utiliza uma propriedade aqui definida como cv2.INTER\_AREA que é uma especificação do cálculo matemático para redimensionar a imagem. Apesar disso, caso a imagem seja redimensionada para um tamanho maior é preciso ponderar que ocorrerá perda de qualidade.

Outra maneira de redimensionar a imagem para tamanhos menores ou maiores é utilizando a técnica de ‘slicing’. Neste caso é fácil cortar pela metade o tamanho da imagem com o código abaixo:

```
import numpy as np
import imutils
import cv2

img = cv2.imread('ponte.jpg')
cv2.imshow("Original", img)
img_redimensionada = img[::2, ::2]

cv2.imshow("Imagen redimensionada", img_redimensionada)
cv2.waitKey(0)
```

O código basicamente refaz a imagem interpolando linhas e colunas, ou seja, pega a primeira linha, ignora a segunda, depois pega a terceira linha, ignora a quarta, e assim por diante. O mesmo é feito com as colunas. Então temos uma imagem que é exatamente  $\frac{1}{4}$  (um

quarto) da original, tendo a metade da altura e a metade da largura da orginal. Veja o resultado abaixo:



**Figura 14 Imagem gerada a partir da técnica de slicing.**

### 4.3 Espelhando uma imagem / Flip

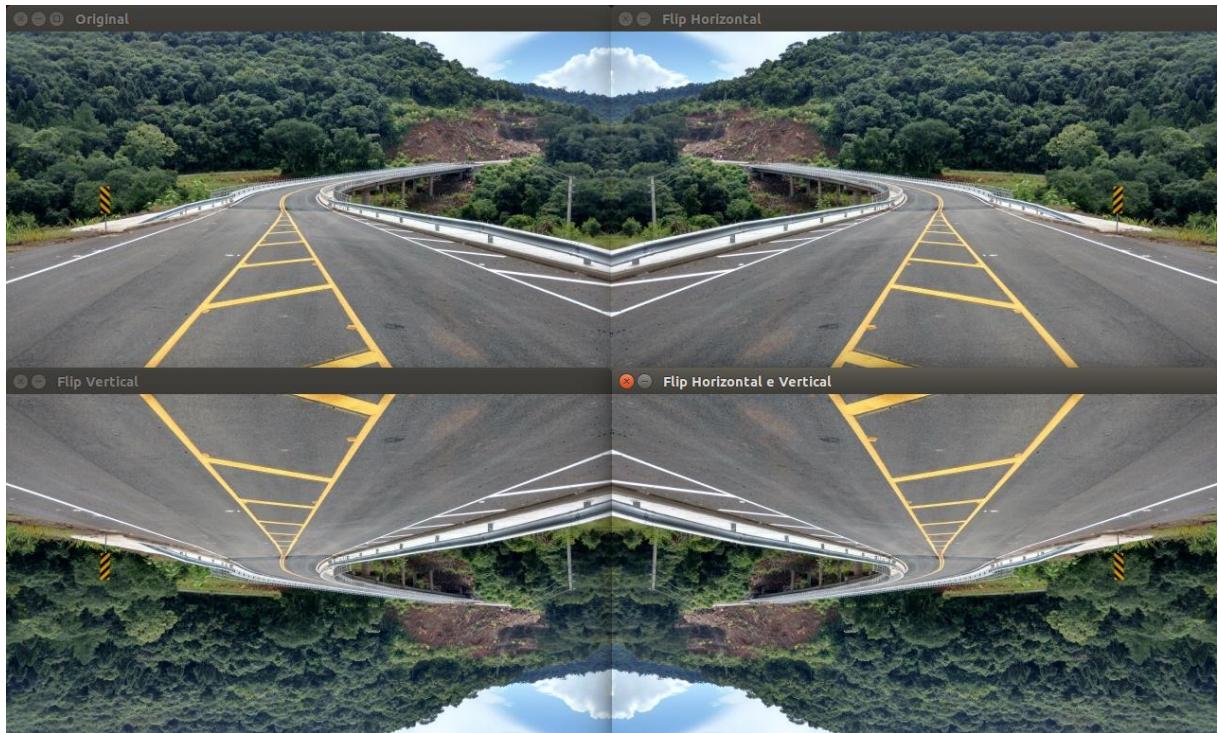
Para espelhar uma imagem, basta inverter suas linhas, suas colunas ou ambas. Invertendo as linhas temos o flip horizontal e invertendo as colunas temos o flip vertical.

Podemos fazer o espelhamento/flip tanto com uma função oferecida pela OpenCV (função `flip`) como através da manipulação direta das matrizes que compõe a imagem. Abaixo temos os dois códigos equivalentes em cada caso.

```
import cv2
img = cv2.imread('ponte.jpg')
cv2.imshow("Original", img)
flip_horizontal = img[::-1,:] #comando equivalente abaixo
#flip_horizontal = cv2.flip(img, 1)

cv2.imshow("Flip Horizontal", flip_horizontal)
flip_vertical = img[:,::-1] #comando equivalente abaixo
#flip_vertical = cv2.flip(img, 0)

cv2.imshow("Flip Vertical", flip_vertical)
flip_hv = img[::-1,::-1] #comando equivalente abaixo
#flip_hv = cv2.flip(img, -1)
cv2.imshow("Flip Horizontal e Vertical", flip_hv)
cv2.waitKey(0)
```



**Figura 15 Resultado do flip horizontal, vertical e horizontal e vertical na mesma imagem.**

#### 4.4 Rotacionando uma imagem / Rotate

Em latin *affinis* significa “conectado com” ou que possui conexão. É por isso que uma das mais famosas transformações na geometria que também é utilizada em processamento de imagem se chama “*affine*”.

A transformação *affine* ou mapa *affine*, é uma função entre espaços *affine* que preservam os pontos, grossura de linhas e planos. Além disso, linhas paralelas permanecem paralelas após uma transformação *affine*. Essa transformação não necessariamente preserva a distância entre pontos mas ela preserva a proporção das distâncias entre os pontos de uma linha reta. Uma rotação é um tipo de transformação *affine*.

```
img = cv2.imread('ponte.jpg')
(alt, lar) = img.shape[:2] #captura altura e largura
centro = (lar // 2, alt // 2) #acha o centro

M = cv2.getRotationMatrix2D(centro, 30, 1.0) #30 graus
img_rotacionada = cv2.warpAffine(img, M, (lar, alt))

cv2.imshow("Imagen rotacionada em 45 graus", img_rotacionada)
cv2.waitKey(0)
```



**Figura 16 Imagem rotacionada em 30 graus.**

#### 4.5 Máscaras

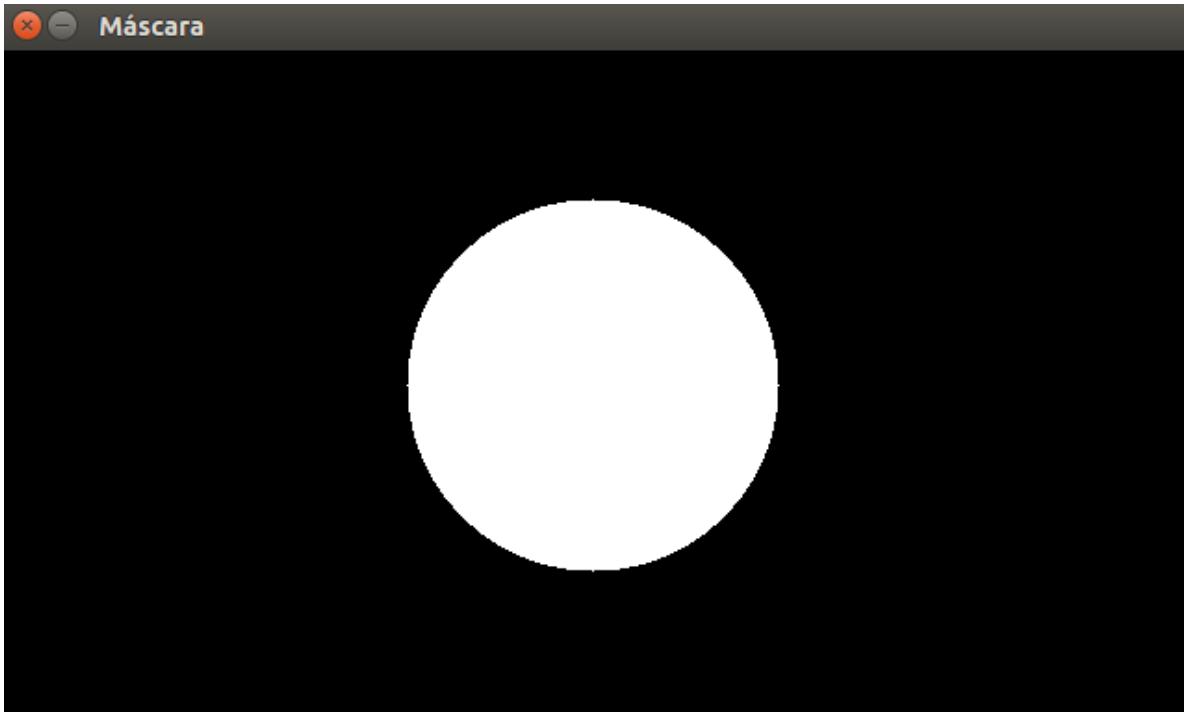
Agora que já vimos alguns tipos de processamento vamos avançar para o assunto de máscaras. Primeiro é importante definir que uma máscara nada mais é que uma imagem onde cada pixel pode estar “ligado” ou “desligado”, ou seja, a máscara possui pixels pretos e brancos apenas. Veja um exemplo:



**Figura 17 Imagem original à esquerda e à direita com a aplicação da máscara.**

O código necessário esta abaixo:

```
img = cv2.imread('ponte.jpg')
cv2.imshow("Original", img)
mascara = np.zeros(img.shape[:2], dtype = "uint8")
(cX, cY) = (img.shape[1] // 2, img.shape[0] // 2)
cv2.circle(mascara, (cX, cY), 100, 255, -1)
img_com_mascara = cv2.bitwise_and(img, img, mask = mascara)
cv2.imshow("Máscara aplicada à imagem", img_com_mascara)
cv2.waitKey(0)
```



---

```
import cv2
import numpy as np

img = cv2.imread('ponte.jpg')

cv2.imshow("Original", img)

mascara = np.zeros(img.shape[:2], dtype = "uint8")

(cX, cY) = (img.shape[1] // 2, img.shape[0] // 2)

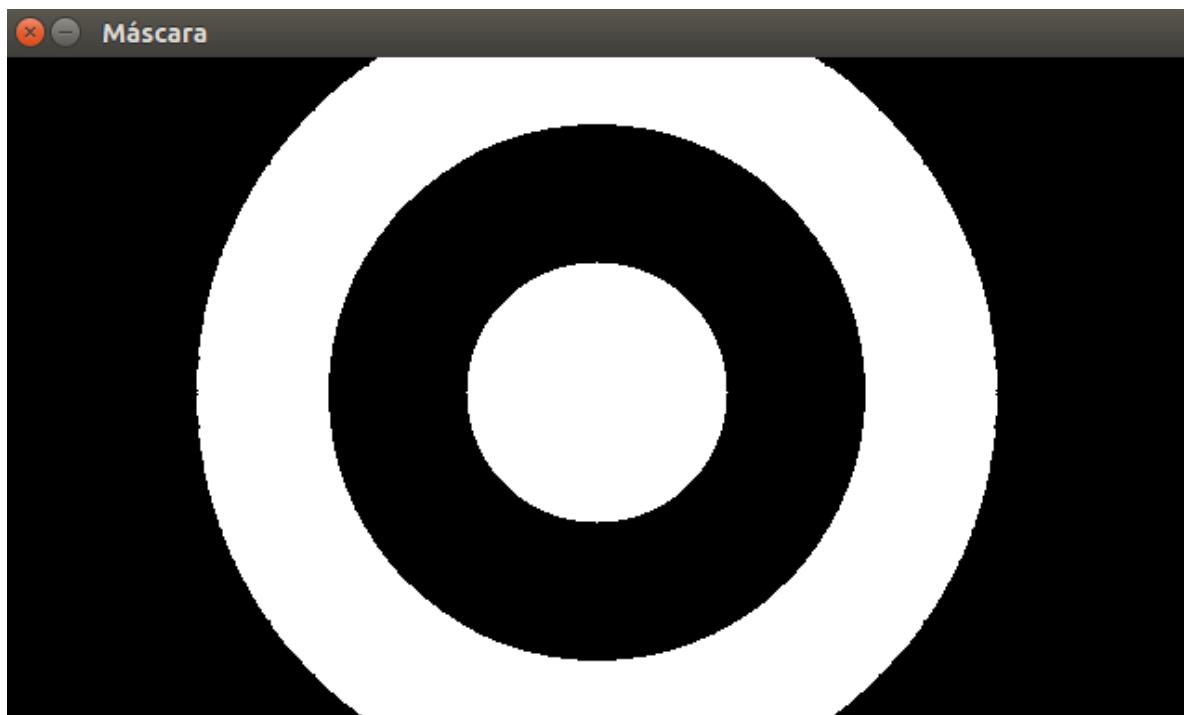
cv2.circle(mascara, (cX, cY), 180, 255, 70)
cv2.circle(mascara, (cX, cY), 70, 255, -1)

cv2.imshow("Máscara", mascara)

img_com_mascara = cv2.bitwise_and(img, img, mask = mascara)
```

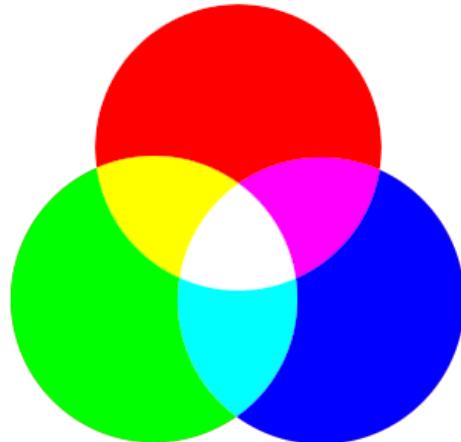
```
cv2.imshow("Máscara aplicada à imagem", img_com_mascara)
```

```
cv2.waitKey()
```



## 5 Sistemas de cores

Já conhecemos o tradicional espaço de cores RGB (Red, Green, Blue) que sabemos que em OpenCV é na verdade BGR dada a necessidade de colocar o azul como primeiro elemento e o vermelho como terceiro elemento de uma tupla que compõe as cores de pixel.



**Figura 18 Sistema de cores RGB.**

Contudo, existem outros espaços de cores como o próprio “Preto e Branco” ou “tons de cinza”, além de outros coloridos como o L\*a\*b\* e o HSV. Abaixo temos um exemplo de como ficaria nossa imagem da ponte nos outros espaços de cores.



**Figura 19 Outros espaços de cores com a mesma imagem.**

O código para gerar o resultado visto acima é o seguinte:

```
img = cv2.imread('ponte.jpg')
```

```

cv2.imshow("Original", img)

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imshow("Gray", gray)
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
cv2.imshow("HSV", hsv)

lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
cv2.imshow("L*a*b*", lab)
cv2.waitKey(0)

```

## 5.1 Canais da imagem colorida

Como já sabemos uma imagem colorida no formato RGB possui 3 canais, um para cada cor. Existem funções do OpenCV que permitem separar e visualizar esses canais individualmente. Veja:

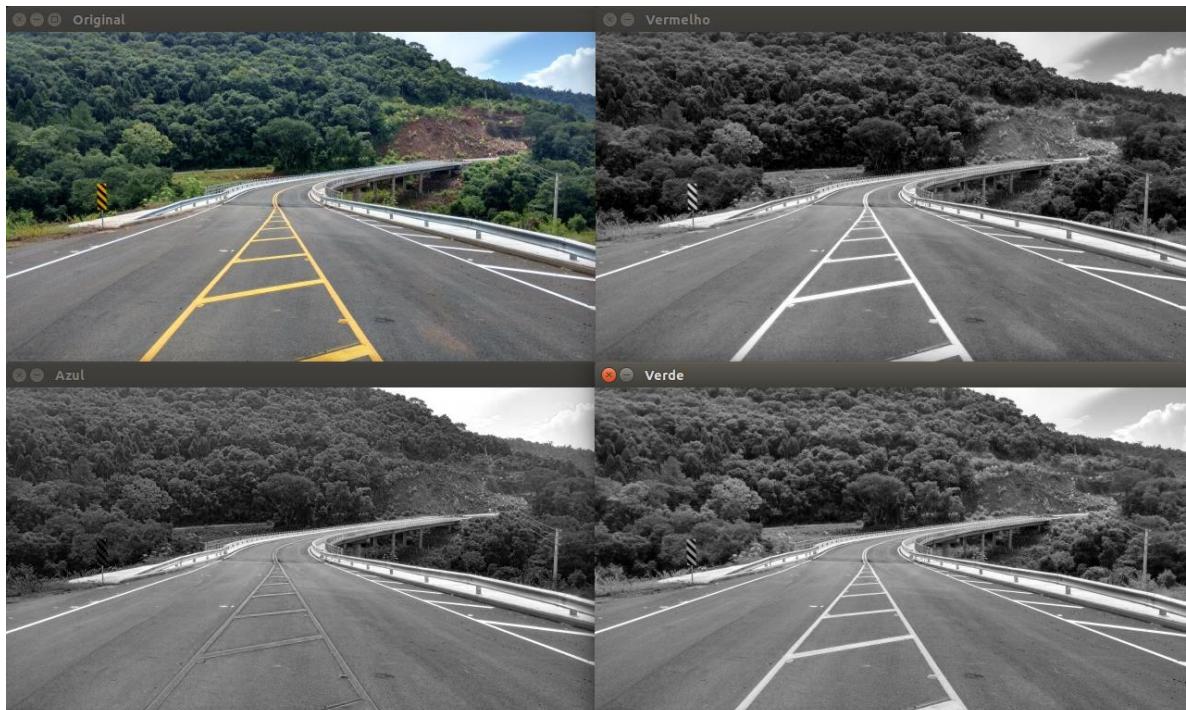
```

img = cv2.imread('ponte.jpg')
(canalAzul, canalVerde, canalVermelho) = cv2.split(img)

cv2.imshow("Vermelho", canalVermelho)
cv2.imshow("Verde", canalVerde)
cv2.imshow("Azul", canalAzul)
cv2.waitKey(0)

```

A função ‘split’ faz o trabalho duro separando os canais. Assim podemos exibi-los em tons de cinza conforme mostra a imagem abaixo:



**Figura 20** Perceba como a linha amarela (que é formada por verde e vermelho) fica quase imperceptível no canal azul.

Também é possível alterar individualmente as Numpy Arrays que formam cada canal e depois juntá-las para criar novamente a imagem. Para isso use o comando:

```
resultado = cv2.merge([canalAzul, canalVerde, canalVermelho])
```

Também é possível exibir os canais nas cores originais conforme abaixo:

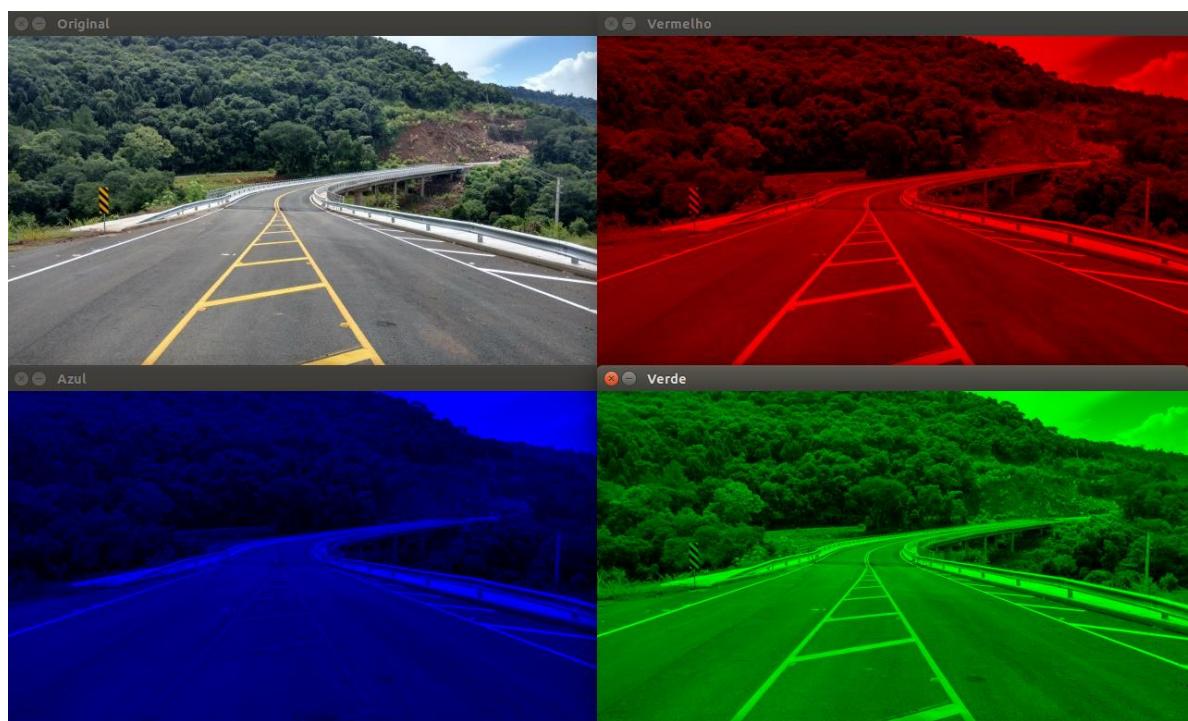
```
import numpy as np
import cv2
img = cv2.imread('ponte.jpg')

(canalAzul, canalVerde, canalVermelho) = cv2.split(img)

zeros = np.zeros(img.shape[:2], dtype = "uint8")

cv2.imshow("Vermelho", cv2.merge([zeros, zeros, canalVermelho]))

cv2.imshow("Verde", cv2.merge([zeros, canalVerde, zeros]))
cv2.imshow("Azul", cv2.merge([canalAzul, zeros, zeros]))
cv2.imshow("Original", img)
cv2.waitKey(0)
```



**Figura 21 Exibindo os canais separadamente.**

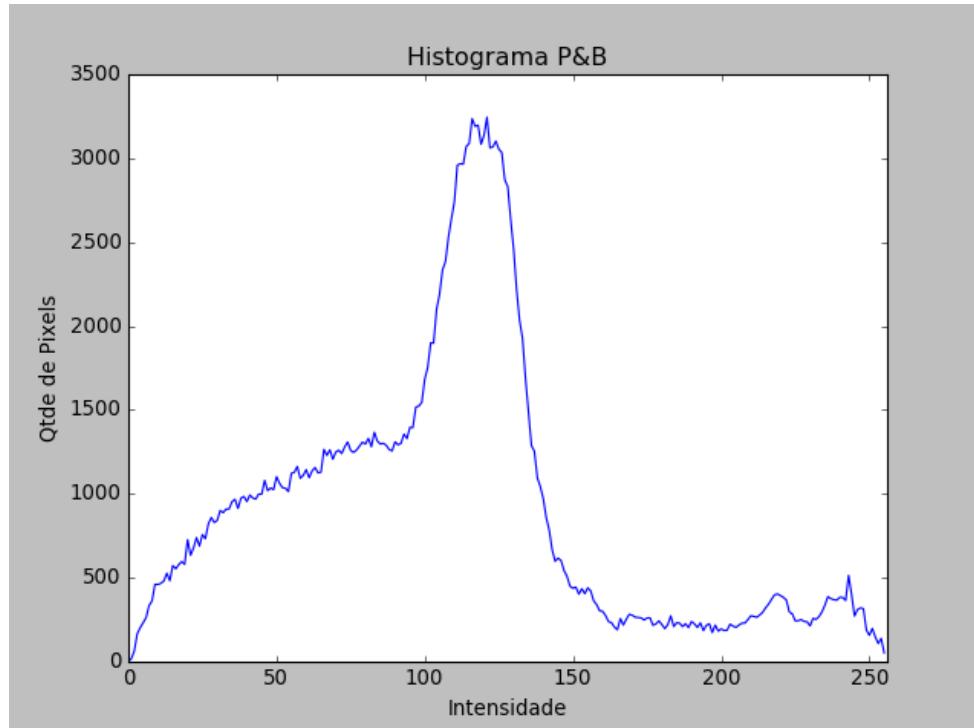
## 6 Histogramas e equalização de imagem

Um histograma é um gráfico de colunas ou de linhas que representa a distribuição dos valores dos pixels de uma imagem, ou seja, a quantidade de pixeis mais claros (próximos de 255) e a quantidade de pixels mais escuros (próximos de 0).

O eixo X do gráfico normalmente possui uma distribuição de 0 a 255 que demonstra o valor (intensidade) do pixel e no eixo Y é plotada a quantidade de pixels daquela intensidade.



**Figura 22 Imagem original já convertida para tons de cinza.**



**Figura 23 Histograma da imagem em tons de cinza.**

Perceba que no histograma existe um pico ao centro do gráfico, entre 100 e 150,

demonstrando a grande quantidade de pixels nessa faixa devido a estrada que ocupa grande parte da imagem possui pixels nessa faixa.

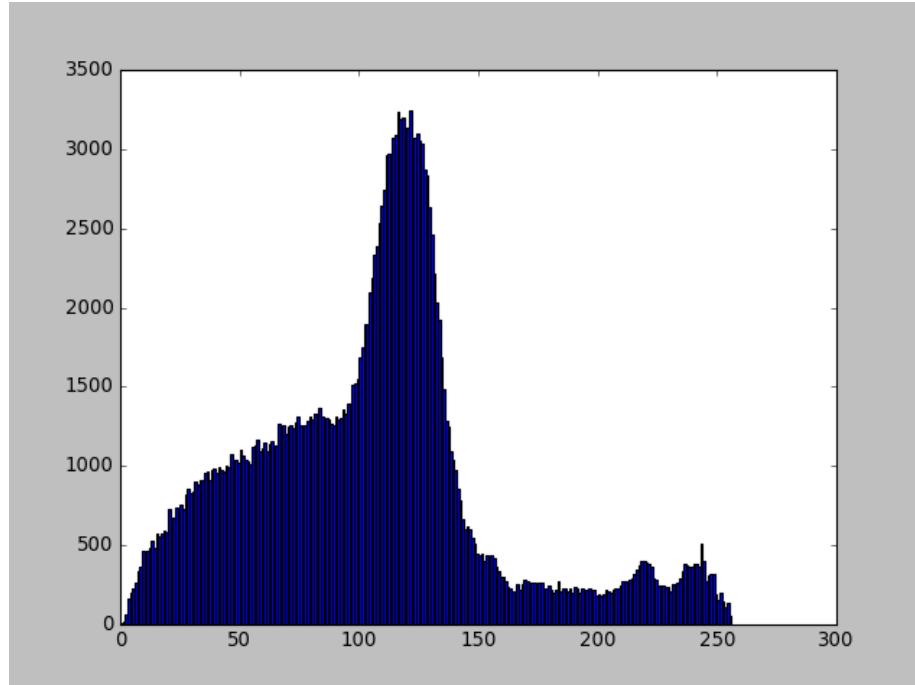
O código para gerar o histograma segue abaixo:

```
from matplotlib import pyplot as plt
import cv2

img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #converte P&B
cv2.imshow("Imagen P&B", img)
#Função calcHist para calcular o hisograma da imagem
h = cv2.calcHist([img], [0], None, [256], [0, 256])
plt.figure()
plt.title("Histograma P&B")
plt.xlabel("Intensidade")
plt.ylabel("Qtde de Pixels")
plt.plot(h)
plt.xlim([0, 256])
plt.show()
cv2.waitKey(0)
```

Também é possível plotar o histograma de outra forma, com a ajuda da função ‘ravel()’. Neste caso o eixo X avança o valor 255 indo até 300, espaço que não existem pixels.

```
plt.hist(img.ravel(), 256, [0, 256])
plt.show()
```



**Figura 24 Histograma em barras.**

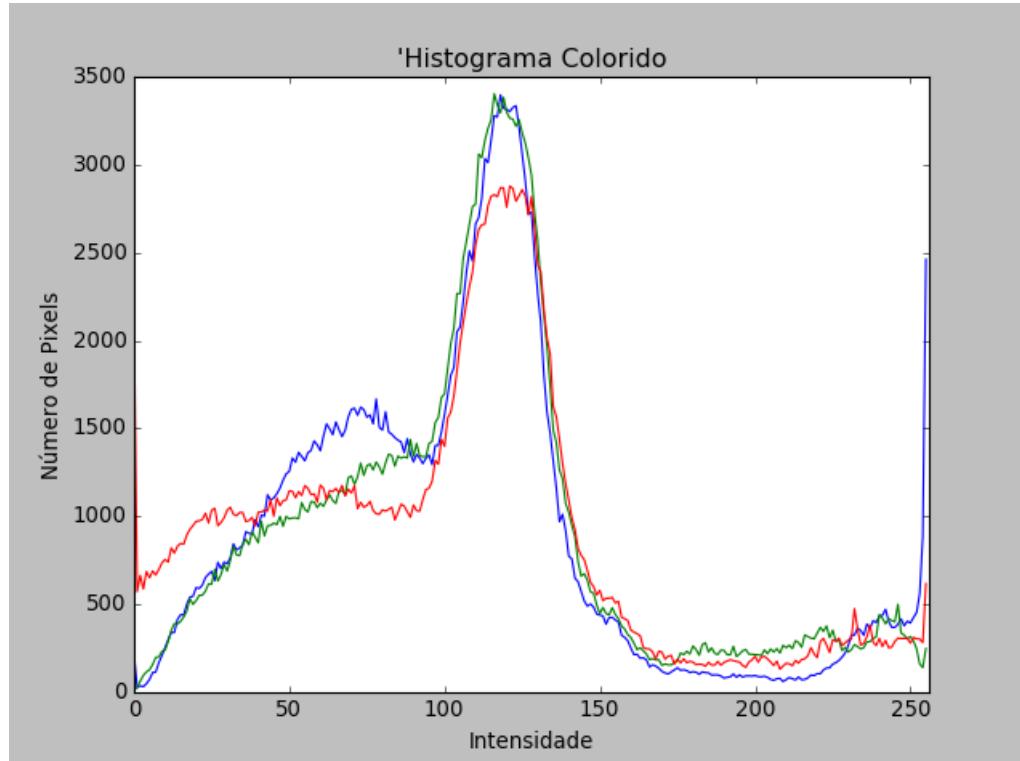
Além do histograma da imagem em tons de cinza é possível plotar um histograma da imagem colorida. Neste caso teremos três linhas, uma para cada canal. Veja abaixo o código

necessário. Importante notar que a função ‘zip’ cria uma lista de tuplas formada pelas união das listas passadas e não tem nada a ver com um processo de compactação como poderia se esperar.

```
from matplotlib import pyplot as plt
import numpy as np
import cv2

img = cv2.imread('ponte.jpg')
cv2.imshow("Imagen Colorida", img)

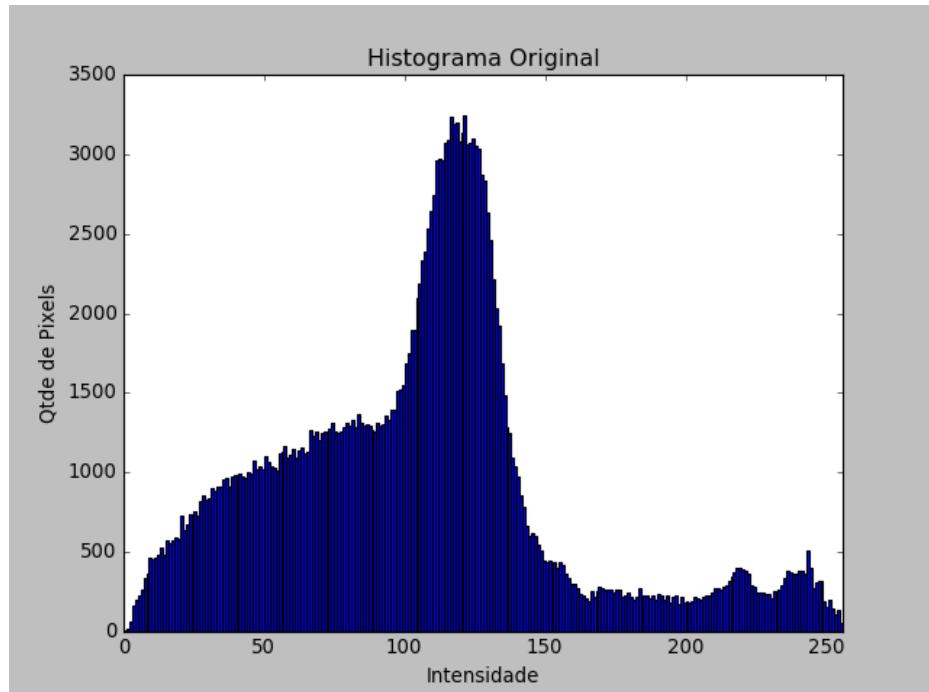
#Separa os canais
canais = cv2.split(img)
cores = ("b", "g", "r")
plt.figure()
plt.title("'Histograma Colorido")
plt.xlabel("Intensidade")
plt.ylabel("Número de Pixels")
for (canal, cor) in zip(canais, cores):
    #Este loop executa 3 vezes, uma para cada canal
    hist = cv2.calcHist([canal], [0], None, [256], [0, 256])
    plt.plot(hist, cor = cor)
    plt.xlim([0, 256])
plt.show()
```



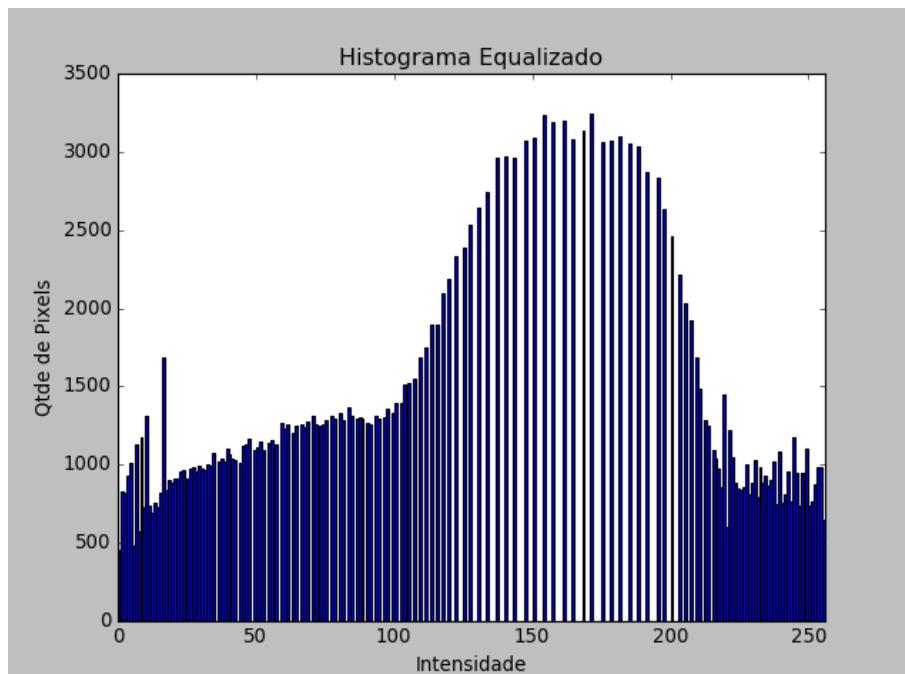
**Figura 25 Histograma colorido da imagem. Neste caso são protados os 3 canais RGB.**

## 6.1 Equalização de Histograma

É possível realizar um cálculo matemático sobre a distribuição de pixels para aumentar o contraste da imagem. A intenção neste caso é distribuir de forma mais uniforme as intensidades dos pixels sobre a imagem. No histograma é possível identificar a diferença pois o acumulo de pixels próximo a alguns valores é suavizado. Veja a diferença entre o histograma original e o equalizado abaixo:



**Figura 26** Histograma da imagem original.



**Figura 27** Histograma da imagem cujo histograma foi equalizado.

O código utilizado para gerar os dois histogramas segue abaixo:

```

from matplotlib import pyplot as plt
import numpy as np
import cv2
img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
h_eq = cv2.equalizeHist(img)

plt.figure()
plt.title("Histograma Equalizado")
plt.xlabel("Intensidade")
plt.ylabel("Qtde de Pixels")
plt.hist(h_eq.ravel(), 256, [0,256])
plt.xlim([0, 256])
plt.show()

plt.figure()
plt.title("Histograma Original")
plt.xlabel("Intensidade")
plt.ylabel("Qtde de Pixels")
plt.hist(img.ravel(), 256, [0,256])
plt.xlim([0, 256])
plt.show()
cv2.waitKey(0)

```

Na imagem a diferença também é perceptível, veja:



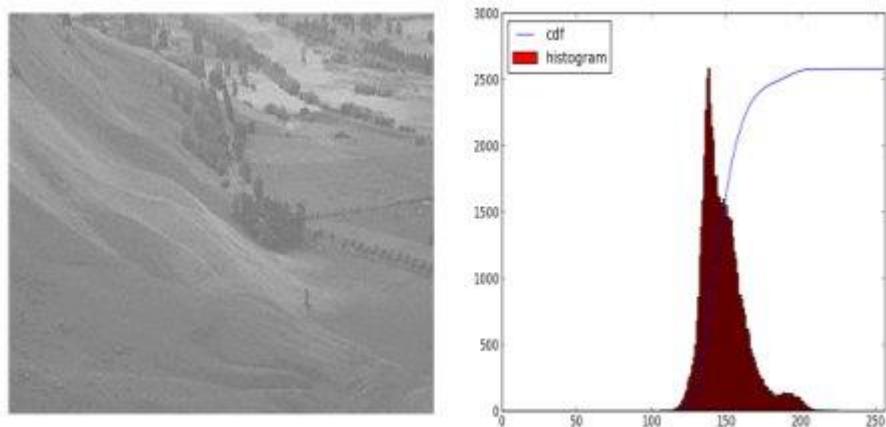
**Figura 28 Imagem original (acima) e imagem cujo histograma foi equalizado (abaixo). Na imagem cujo histograma foi equalizado percebemos maior contraste.**

Contudo, conforme vemos na imagem é possível que ocorram distorções e alterações nas cores da imagem equalizada, portanto, nem sempre a imagem mantém suas características

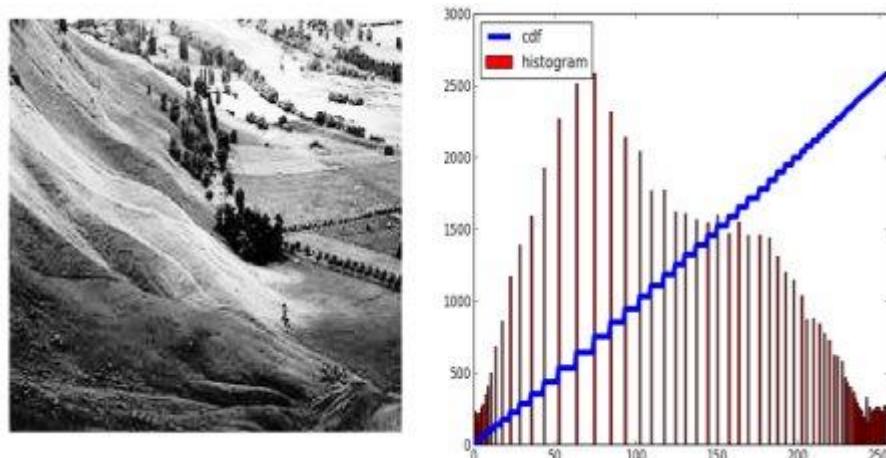
originais. Porém caso exista a necessidade de destacar detalhes na imagem a equalização pode ser uma grande aliada, isso normalmente é feito em imagens para identificação de objetos, imagens de estudos de áreas por satélite e para identificação de padrões em imagens médicas por exemplo.

O código para equalização do histograma da imagem segue abaixo. O cálculo matemático é feito pela função ‘equalizeHist’ disponibilizada pela OpenCV.

A explicação do algoritmo utilizado pela função é extremamente bem feita própria documentação da OpenCV<sup>1</sup> que mostra o seguinte exemplo:



**Figura 29 Exemplo extraído da documentação da OpenCV mostrando o histograma de uma imagem com baixo contraste.**



**Figura 30 Exemplo extraído da documentação da OpenCV mostrando o histograma da mesma imagem mas dessa vez com o histograma equalizado pela função ‘equalizeHist’.**

Perceba que a equalização faz com que a distribuição das intensidades dos pixels de 0 a 255 seja uniforme. Portanto teremos a mesma quantidade de pixels com valores na faixa de 0 a 10 (pixels muito escuros) e na faixa de 245 a 255 (pixels muito claros).

A função usa o seguinte algoritmo:

---

<sup>1</sup> Disponível em: <[http://docs.opencv.org/3.1.0/d5/daf/tutorial\\_py\\_histogram\\_equalization.html](http://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html)>. Acesso em: 11 mar. 2017.

Passo 1: Calcula o histograma 'H' da imagem.

Passo 2: Normaliza o histograma para garantir que os valores das intensidades dos pixels estejam entre 0 e 255.

Passo 3: Calcula o histograma acumulado  $H'_i = \sum_{0 \leq j \leq i} H(j)$

Passo 4: Transforma a imagem:  $dst(x,y) = H'(img(x,y))$

Dessa forma temos uma distribuição mais uniforme das intensidades dos pixels na imagem. Lembre-se que detalhes podem inclusive serem perdidos com este processamento de imagem. Contudo, o que é garantido é o aumento de contraste.

# 7 Suavização de imagens

A suavização da imagem (do inglês *Smoothing*), também chamada de ‘blur’ ou ‘blurring’ que podemos traduzir para “borrão”, é um efeito que podemos notar nas fotografias fora de foco ou desfocadas onde tudo fica embasado.

Na verdade esse efeito pode ser criado digitalmente, basta alterar a cor de cada pixel misturando a cor com os pixels ao seu redor. Esse efeito é muito útil quando utilizamos algoritmos de identificação de objetos em imagens pois os processos de detecção de bordas por exemplo, funcionam melhor depois de aplicar uma suavização na imagem.

## 7.1 Suavização por cálculo da média

Neste caso é criada uma “caixa de pixels” para envolver o pixel em questão e calcular seu novo valor. O novo valor do pixel será a média simples dos valores dos pixels dentro da caixa, ou seja, dos pixels da vizinhança. Alguns autores chamam esta caixa de janela de cálculo ou *kernel* (do inglês núcleo).

30	100	130
130	Pixel	160
50	100	210

**Figura 31 Caixa 3x3 pixels. O número de linhas e colunas da caixa deve ser ímpar para que existe sempre o pixel central que será alvo do cálculo.**

Portanto o novo valor do pixel será a média da sua vizinhança o que gera a suavização na imagem como um todo.

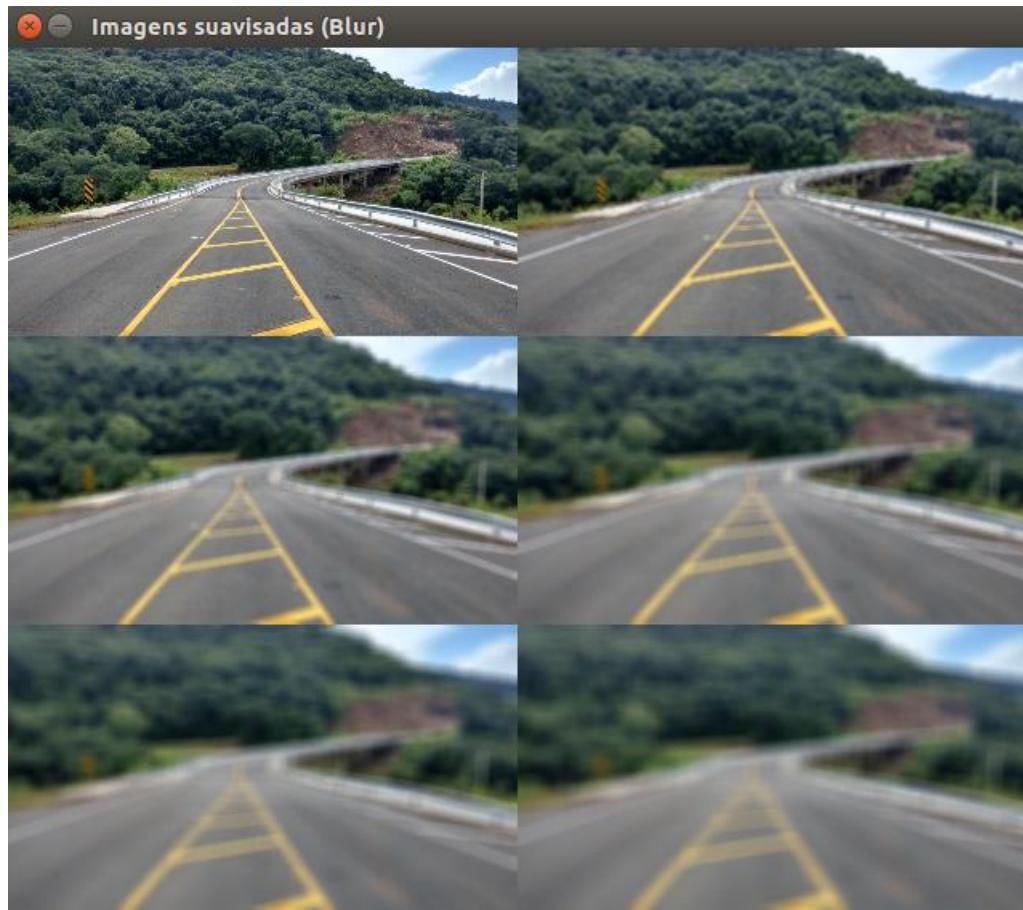
No código abaixo percebemos que o método utilizado para a suavização pela média é o método ‘blur’ da OpenCV. Os parâmetros são a imagem a ser suavizada e a janela de suavização. Colocarmos números impares para gerar as caixas de cálculo pois dessa forma não existe dúvida sobre onde estará o pixel central que terá seu valor atualizado.

Perceba que usamos as funções vstack (pilha vertical) e hstack (pilha horizontal) para juntar as imagens em uma única imagem final mostrando desde a imagem original e seguinte com caixas de calculo de 3x3, 5x5, 7x7, 9x9 e 11x11. Perceba que conforme aumenta a caixa maior é o efeito de borrão (*blur*) na imagem.

```
img = cv2.imread('ponte.jpg')
img = img[:, :, ::2] # Diminui a imagem

suave = np.vstack([
    np.hstack([img, cv2.blur(img, (3, 3))]),
    np.hstack([cv2.blur(img, (5, 5)), cv2.blur(img, (7, 7))]),
    np.hstack([cv2.blur(img, (9, 9)), cv2.blur(img, (11, 11))])
])

cv2.imshow("Imagens suavisadas (Blur)", suave)
cv2.waitKey(0)
```



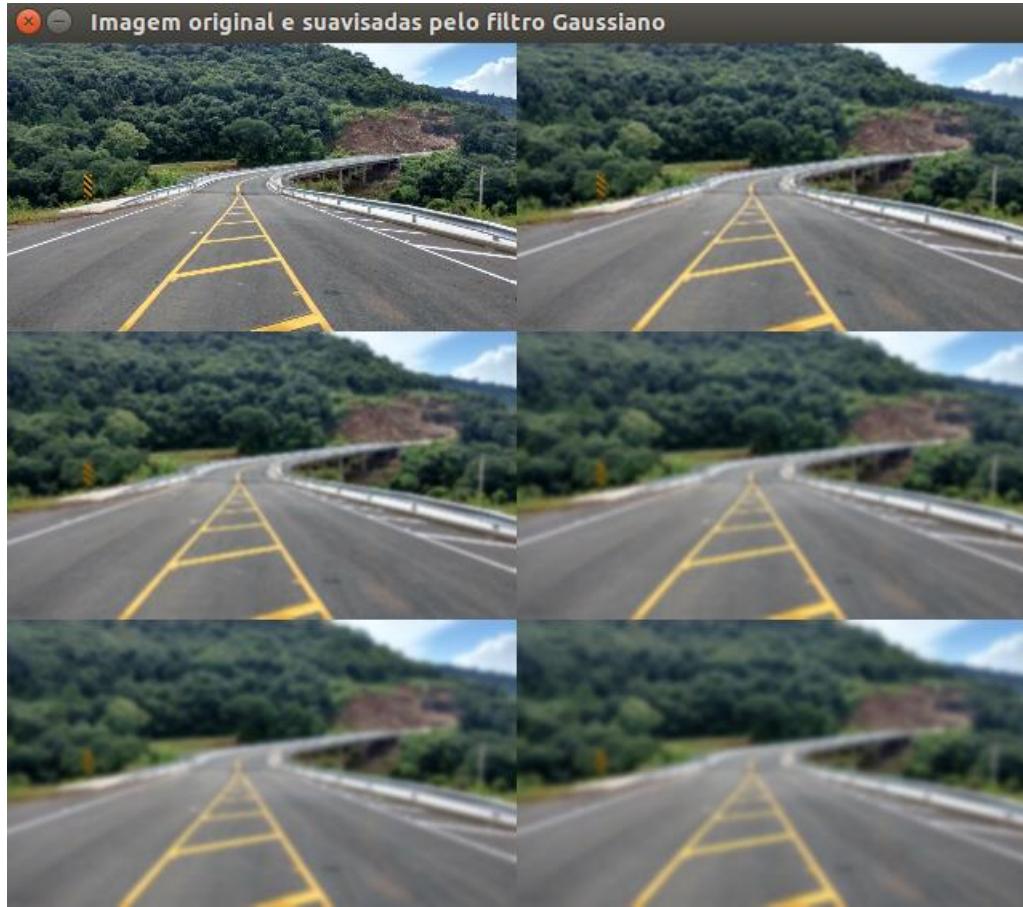
**Figura 32** Imagem original seguida da esquerda para a direita e de cima para baixo com imagens tendo caixas de cálculo de 3x3, 5x5, 7x7, 9x9 e 11x11. Perceba que conforme aumenta a caixa maior é o efeito de borrão (blur) na imagem.

## 7.2 Suavização pela Gaussiana

Ao invés do filtro de caixa é utilizado um kernel gaussiano. Isso é calculado através da função cv2.GaussianBlur(). A função exige a especificação de uma largura e altura com números ímpares e também, opcionalmente, é possível especificar a quantidade de desvios padrão no eixo X e Y (horizontal e vertical).

```
img = cv2.imread('ponte.jpg')
img = img[::2,::2] # Diminui a imagem
suave = np.vstack([
    np.hstack([img,
               cv2.GaussianBlur(img, ( 3, 3), 0)]),
    np.hstack([cv2.GaussianBlur(img, ( 5, 5), 0),
               cv2.GaussianBlur(img, ( 7, 7), 0)]),
    np.hstack([cv2.GaussianBlur(img, ( 9, 9), 0),
               cv2.GaussianBlur(img, (11, 11), 0)]),
])
cv2.imshow("Imagen original e suavisadas pelo filtro Gaussiano", suave)
```

```
cv2.waitKey(0)
```



**Figura 33 Imagem original seguida da esquerda para a direita e de cima para baixo com imagens tendo caixas de cálculo de 3x3, 5x5, 7x7, 9x9 e 11x11 utilizando o Gaussian Blur.**

Veja nas imagens como o filtro de kernel gaussiano gera menos borrão na imagem mas também gera um efeito mais natural e reduz o ruído na imagem.

### 7.3 Suavização pela mediana

Da mesma forma que os cálculos anteriores, aqui temos o cálculo de uma caixa ou janela quadrada sobre um pixel central onde matematicamente se utiliza a mediana para calcular o valor final do pixel. A mediana é semelhante à média, mas ela despreza os valores muito altos ou muito baixos que podem distorcer o resultado. A mediana é o número que fica exatamente no meio do intervalo.

A função utilizada é a `cv2.medianBlur(img, 3)` e o único argumento é o tamaho da caixa ou janela usada.

É importante notar que este método não cria novas cores, como pode acontecer com os anteriores, pois ele sempre altera a cor do pixel atual com um dos valores da vizinhança.

Veja o código usado:

```
import numpy as np
import cv2
```

```

img = cv2.imread('ponte.jpg')
img = img[::2,::2] # Diminui a imagem

suave = np.vstack([
    np.hstack([img,
               cv2.medianBlur(img, 3)]),
    np.hstack([cv2.medianBlur(img, 5),
               cv2.medianBlur(img, 7)]),
    np.hstack([cv2.medianBlur(img, 9),
               cv2.medianBlur(img, 11)]),
])
cv2.imshow("Imagen original e suavisadas pela mediana", suave)
cv2.waitKey(0)

```



**Figura 34** Da mesma forma temos a imagem original seguida pelas imagens alteradas pelo filtro de mediana com o tamanho de 3, 5, 7, 9, e 11 nas caixas de cálculo.

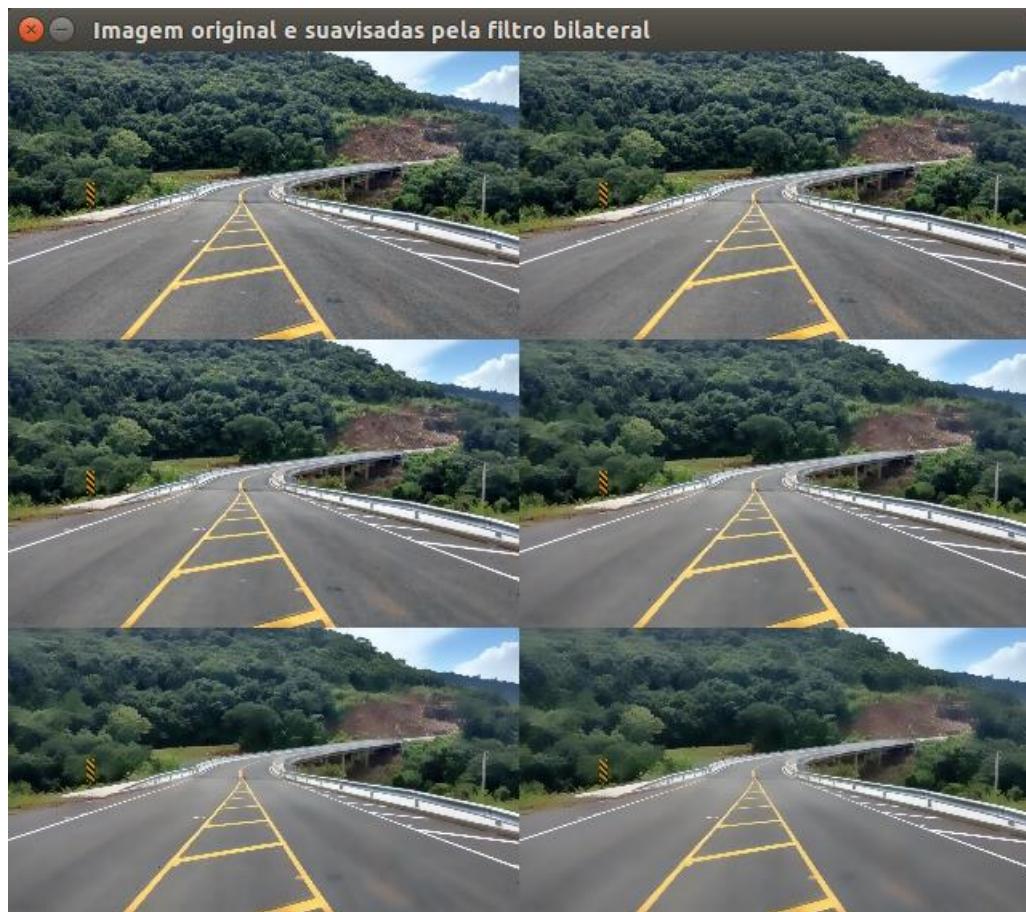
## 7.4 Suavização com filtro bilateral

Este método é mais lento para calcular que os anteriores mas como vantagem apresenta a preservação de bordas e garante que o ruído seja removido.

Para realizar essa tarefa, além de um filtro gaussiano do espaço ao redor do pixel

também é utilizado outro cálculo com outro filtro gaussiano que leva em conta a diferença de intensidade entre os pixels, dessa forma, como resultado temos uma maior manutenção das bordas das imagens. A função usada é cv2.bilateralFilter() e o código usado segue abaixo:

```
img = cv2.imread('ponte.jpg')
img = img[::2, ::2] # Diminui a imagem
suave = np.vstack([
    np.hstack([img,
               cv2.bilateralFilter(img, 3, 21, 21)]),
    np.hstack([cv2.bilateralFilter(img, 5, 35, 35),
               cv2.bilateralFilter(img, 7, 49, 49)]),
    np.hstack([cv2.bilateralFilter(img, 9, 63, 63),
               cv2.bilateralFilter(img, 11, 77, 77)])
])
```



**Figura 35 Imagem original e imagens alteradas pelo filtro bilateral. Veja como mesmo com a grande interferência na imagem no caso da imagem mais à baixo e à direita as bordas são preservadas.**

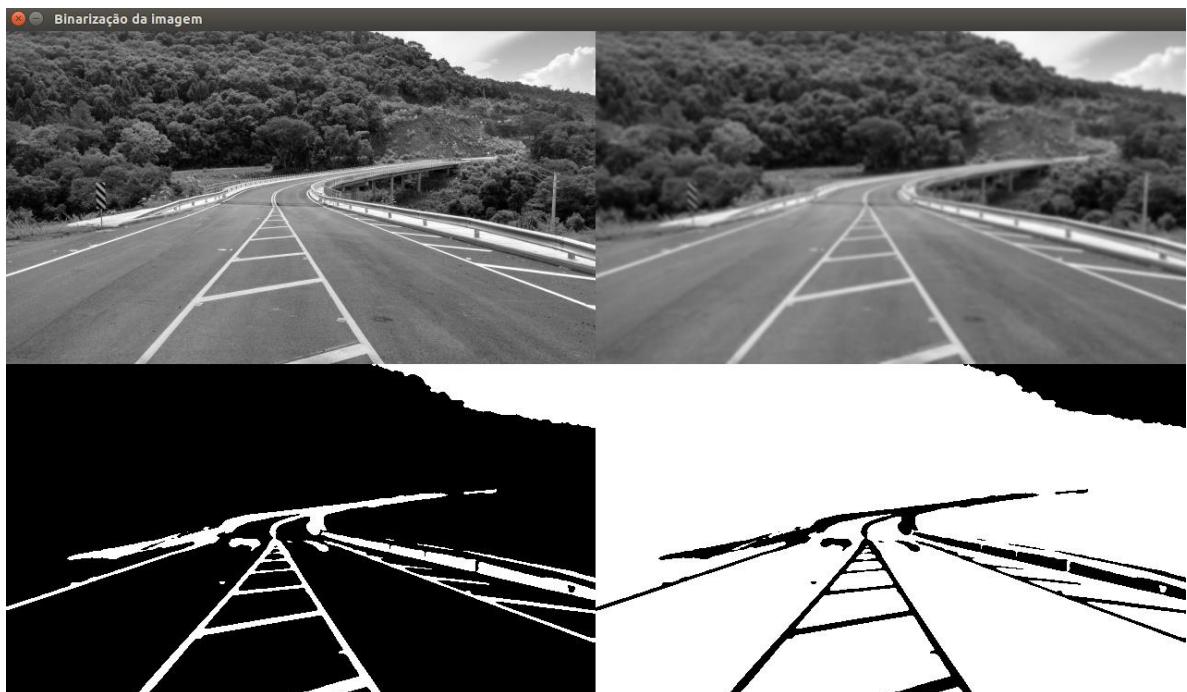
## 8 Binarização com limiar

Thresholding pode ser traduzido por limiarização e no caso de processamento de imagens na maior parte das vezes utilizamos para binarização da imagem. Normalmente convertemos imagens em tons de cinza para imagens preto e branco onde todos os pixels possuem 0 ou 255 como valores de intensidade.

```
img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
suave = cv2.GaussianBlur(img, (7, 7), 0) # aplica blur
(T, bin) = cv2.threshold(suave, 160, 255, cv2.THRESH_BINARY)
(T, binI) = cv2.threshold(suave, 160, 255,
cv2.THRESH_BINARY_INV)
resultado = np.vstack([
    np.hstack([suave, bin]),
    np.hstack([binI, cv2.bitwise_and(img, img, mask = binI)])
])

cv2.imshow("Binarização da imagem", resultado)
cv2.waitKey(0)
```

No código realizamos a suavização da imagem, o processo de binarização com threshold de 160 e a inversão da imagem binarizada.



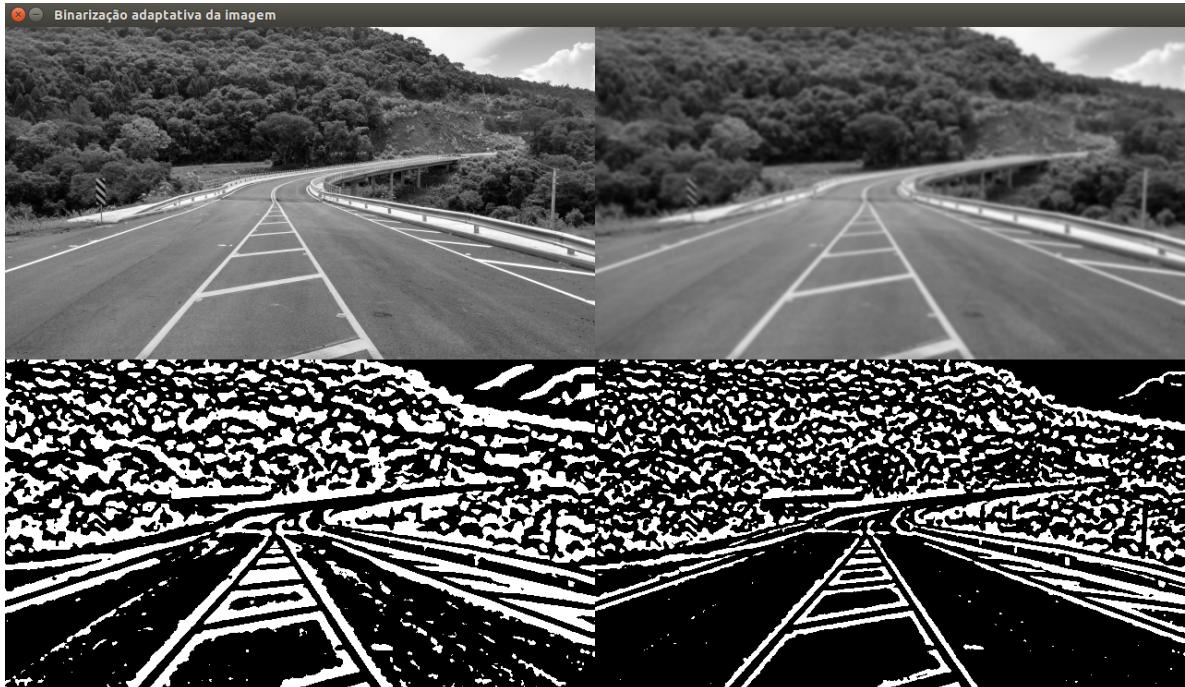
**Figura 36** Da esquerda para a direta e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada e a imagem binarizada invertida.

No caso das estradas, esta é uma das técnicas utilizadas por carros autônomos para identificar a pista. A mesma técnica também é utilizada para identificação de objetos.

## 8.1 Threshold adaptativo

O valor de intensidade 160 utilizada para a binarização acima foi arbitrado, contudo, é possível otimizar esse valor matematicamente. Esta é a proposta do threshold adaptativo.

Para isso precisamos dar um valor da janela ou caixa de cálculo para que o limiar seja calculado nos pixels próximos das imagens. Outro parâmetro é um inteiro que é subtraído da média calculada dentro da caixa para gerar o threshold final.



**Figura 37 Threshold adaptativo.** Da esquerda para a direta e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada pela média e a imagem binarizada com Gauss.

```
img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # converte
suave = cv2.GaussianBlur(img, (7, 7), 0) # aplica blur

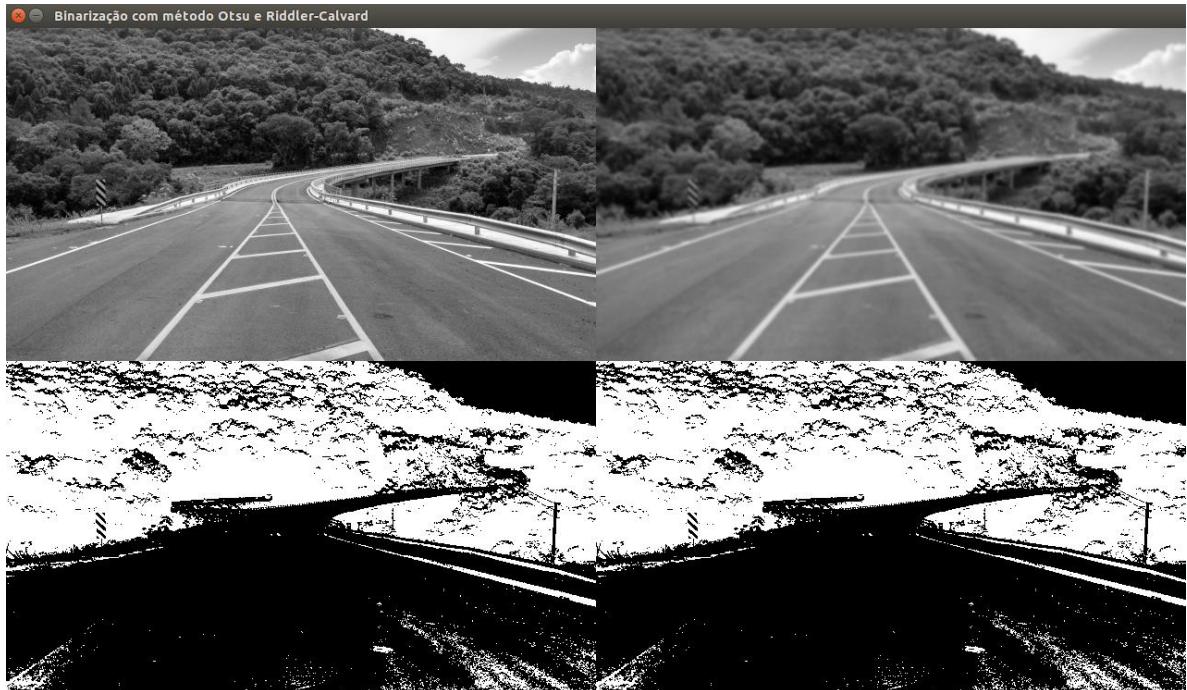
bin1 = cv2.adaptiveThreshold(suave, 255,
    cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 21, 5)
bin2 = cv2.adaptiveThreshold(suave, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV,
    21, 5)

resultado = np.vstack([
    np.hstack([img, suave]),
    np.hstack([bin1, bin2])
])

cv2.imshow("Binarização adaptativa da imagem", resultado)
cv2.waitKey(0)
```

## 8.2 Threshold com Otsu e Riddler-Calvard

Outro método que automaticamente encontra um threshold para a imagem é o método de Otsu. Neste caso ele analiza o histograma da imagem para encontrar os dois maiores picos de intensidades, então ele calcula um valor para separar da melhor forma esses dois picos.



**Figura 38 1.1 Threshold com Otsu e Riddler-Calvard.**

```

import mahotas
import numpy as np
import cv2

img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # converte
suave = cv2.GaussianBlur(img, (7, 7), 0) # aplica blur
T = mahotas.thresholding.otsu(suave)
temp = img.copy()
temp[temp > T] = 255
temp[temp < 255] = 0
temp = cv2.bitwise_not(temp)
T = mahotas.thresholding.rc(suave)
temp2 = img.copy()
temp2[temp2 > T] = 255
temp2[temp2 < 255] = 0
temp2 = cv2.bitwise_not(temp2)
resultado = np.vstack([
    np.hstack([img, suave]),
    np.hstack([temp, temp2])
])
cv2.imshow("Binarização com método Otsu e Riddler-Calvard",
resultado)
cv2.waitKey(0)

```

# 9 Segmentação e métodos de detecção de bordas

Uma das tarefas mais importantes para a visão computacional é identificar objetos. Para essa identificação uma das principais técnicas é a utilização de detectores de bordas a fim de identificar os formatos dos objetos presentes na imagem.

Quando falamos em segmentação e detecção de bordas, os algoritmos mais comuns são o Canny, Sobel e variações destes. Basicamente nestes e em outros métodos a detecção de bordas se faz através de identificação do gradiente, ou, neste caso, de variações abruptas na intensidade dos pixels de uma região da imagem.

A OpenCV disponibiliza a implementação de 3 filtros de gradiente (*High-pass filters*): Sobel, Scharr e Laplacian. As respectivas funções são: cv2.Sobel(), cv2.Scharr(), cv2.Laplacian().

## 9.1 Sobel

Não entraremos na explicação matemática de cada método mas é importante notar que o Sobel é direcional, então temos que juntar o filtro horizontal e o vertical para ter uma transformação completa, veja:

```
import numpy as np
import cv2

img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

sobelX = cv2.Sobel(img, cv2.CV_64F, 1, 0)
sobelY = cv2.Sobel(img, cv2.CV_64F, 0, 1)
sobelX = np.uint8(np.absolute(sobelX))
sobelY = np.uint8(np.absolute(sobelY))
sobel = cv2.bitwise_or(sobelX, sobelY)

resultado = np.vstack([
    np.hstack([img, sobelX]),
    np.hstack([sobelY, sobel])
])

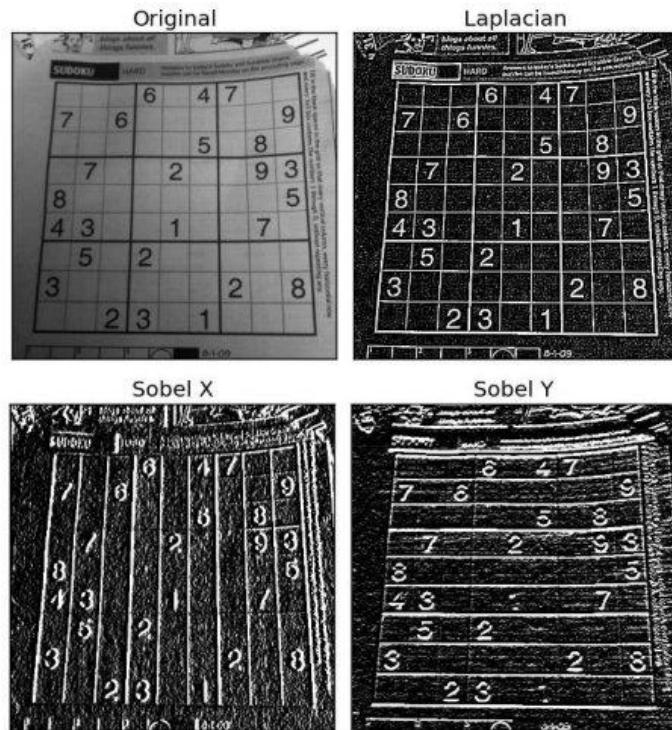
cv2.imshow("Sobel", resultado)
cv2.waitKey(0)
```

Note que devido ao processamento do Sobel é preciso trabalhar com a imagem com ponto flutuante de 64 bits (que suporta valores positivos e negativos) para depois converter para uint8 novamente.



**Figura 39** Da esquerda para a direita e de cima para baixo temos: a imagem original, Sobel Horizontal (sobelX), Sobel Vertical (sobelY) e a imagem com o Sobel combinado que é o resultado final.

Um belo exemplo de resultado Sobel esta na documentação da OpenCV, veja:



**Figura 40** Exemplo de processamento Sobel Vertical e Horizontal disponível na documentação da OpenCV.

## 9.2 Filtro Laplaciano

O filtro Laplaciano não exige processamento individual horizontal e vertical como o

Sobel. Um único passo é necessário para gerar a imagem abaixo. Contudo, também é necessário trabalhar com a representação do pixel em ponto flutuante de 64 bits com sinal para depois converter novamente para inteiro sem sinal de 8 bits.



**Figura 41 Filtro Laplaciano.**

O código segue abaixo:

```
import numpy as np
import cv2
img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
lap = cv2.Laplacian(img, cv2.CV_64F)
lap = np.uint8(np.absolute(lap))
resultado = np.vstack([img, lap])
cv2.imshow("Filtro Laplaciano", resultado)
cv2.waitKey(0)
```

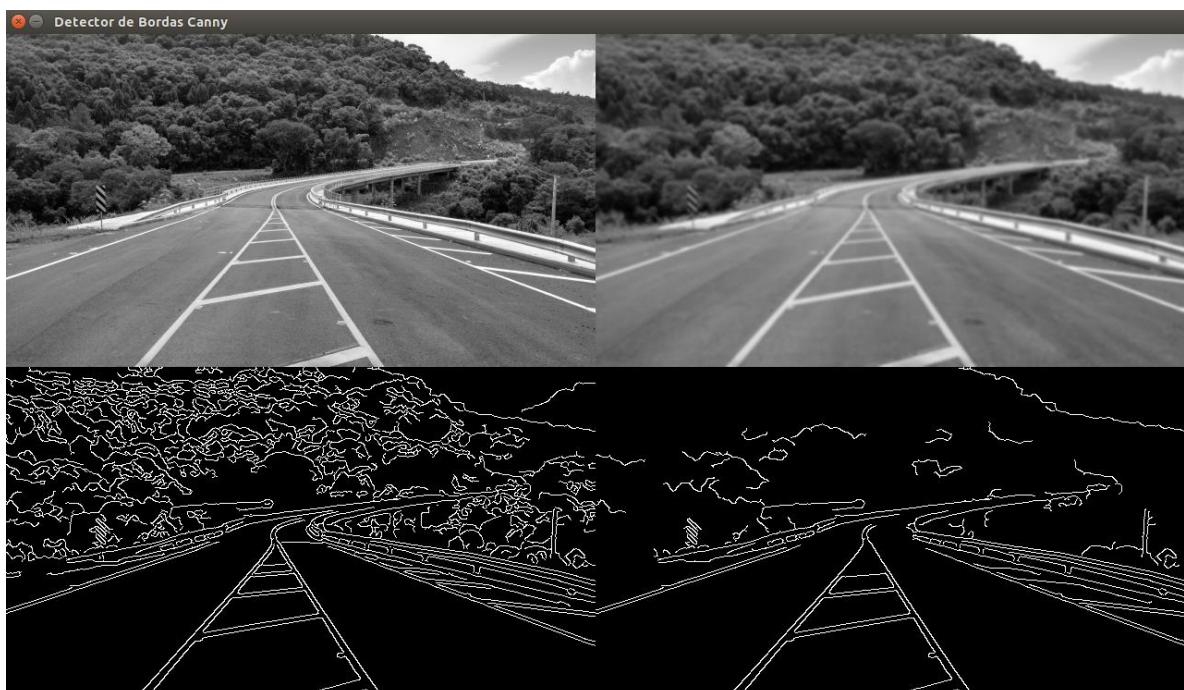
### 9.3 Detector de bordas Canny

Em inglês *canny* pode ser traduzido para esperto, esta no dicionário. E o *Carry Hedge Detector* ou detector de bordas Caany realmente é mais inteligente que os outros. Na verdade ele se utiliza de outras técnicas como o Sobel e realiza múltiplos passos para chegar ao resultado final.

Basicamente o Canny envolve:

1. Aplicar um filtro gaussiano para suavizar a imagem e remover o ruído.
2. Encontrar os gradientes de intensidade da imagem.
3. Aplicar Sobel duplo para determinar bordas potenciais.
4. Aplicar o processo de “hysteresis” para verificar se o pixel faz parte de uma borda “forte” suprimindo todas as outras bordas que são fracas e não conectadas a bordas fortes.

É preciso fornecer dois parâmetros para a função cv2.Canny(). Esses dois valores são o limiar 1 e limiar 2 e são utilizados no processo de “hysteresis” final. Qualquer gradiente com valor maior que o limiar 2 é considerado como borda. Qualquer valor inferior ao limiar 1 não é considerado borda. Valores entre o limiar 1 e limiar 2 são classificados como bordas ou não bordas com base em como eles estão conectados.



**Figura 42 Canny com parâmetros diferentes. A esquerda deixamos um limiar mais baixo (20,120) e à direita a imagem foi gerada com limiares maiores (70,200).**

```
import numpy as np
import cv2
img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
suave = cv2.GaussianBlur(img, (7, 7), 0)

canny1 = cv2.Canny(suave, 20, 120)
canny2 = cv2.Canny(suave, 70, 200)
resultado = np.vstack([
    np.hstack([img, suave]),
    np.hstack([canny1, canny2])
])
cv2.imshow("Detector de Bordas Canny", resultado)
cv2.waitKey(0)
```

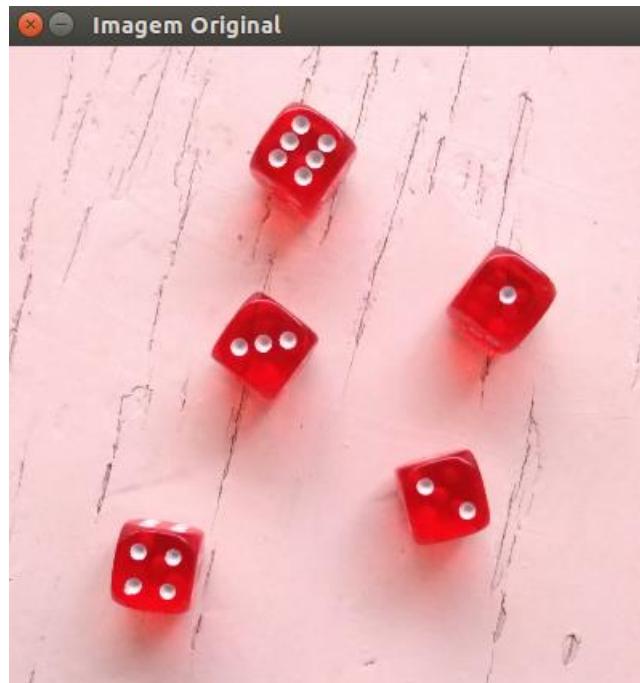
## 10 Identificando e contando objetos

Como todos sabem, a atividade de jogar dados é muito útil. Muito útil para jogar RPG, General e outros jogos. Mas depois do sistema apresentado abaixo, não será mais necessário clicar no mouse ou pressionar uma tecla do teclado para jogar com o computador. Você poderá jogar os dados de verdade e o computador irá “ver” sua pontuação.

Para isso precisamos identificar:

1. Onde estão os dados na imagem.
2. Quantos dados foram jogados.
3. Qual é o lado que está para cima.

Inicialmente vamos identificar os dados e contar quantos dados existem na imagem, em um segundo momento iremos identificar quais são esses dados. A imagem que temos esta abaixo. Não é uma imagem fácil pois além dos dados serem vermelhos e terem um contraste menor que dados brancos sobre uma mesa preta, por exemplo, eles ainda estão sobre uma superfície branca com ranhuras, ou seja, não é uma superfície uniforme. Isso irá dificultar nosso trabalho.



**Figura 43 Imagem original. A superfície branca com ranhuras dificultará o processo.**

Os passos mostrados na sequência de imagens abaixo são:

1. Convertemos a imagem para tons de cinza.
2. Aplicamos *blur* para retirar o ruído e facilitar a identificação das bordas.
3. Aplicamos uma binarização na imagem resultando em pixels só brancos e pretos.
4. Aplicamos um detector de bordas para identificar os objetos.
5. Com as bordas identificadas, vamos contar os contornos externos para achar a quantidade de dados presentes na imagem.

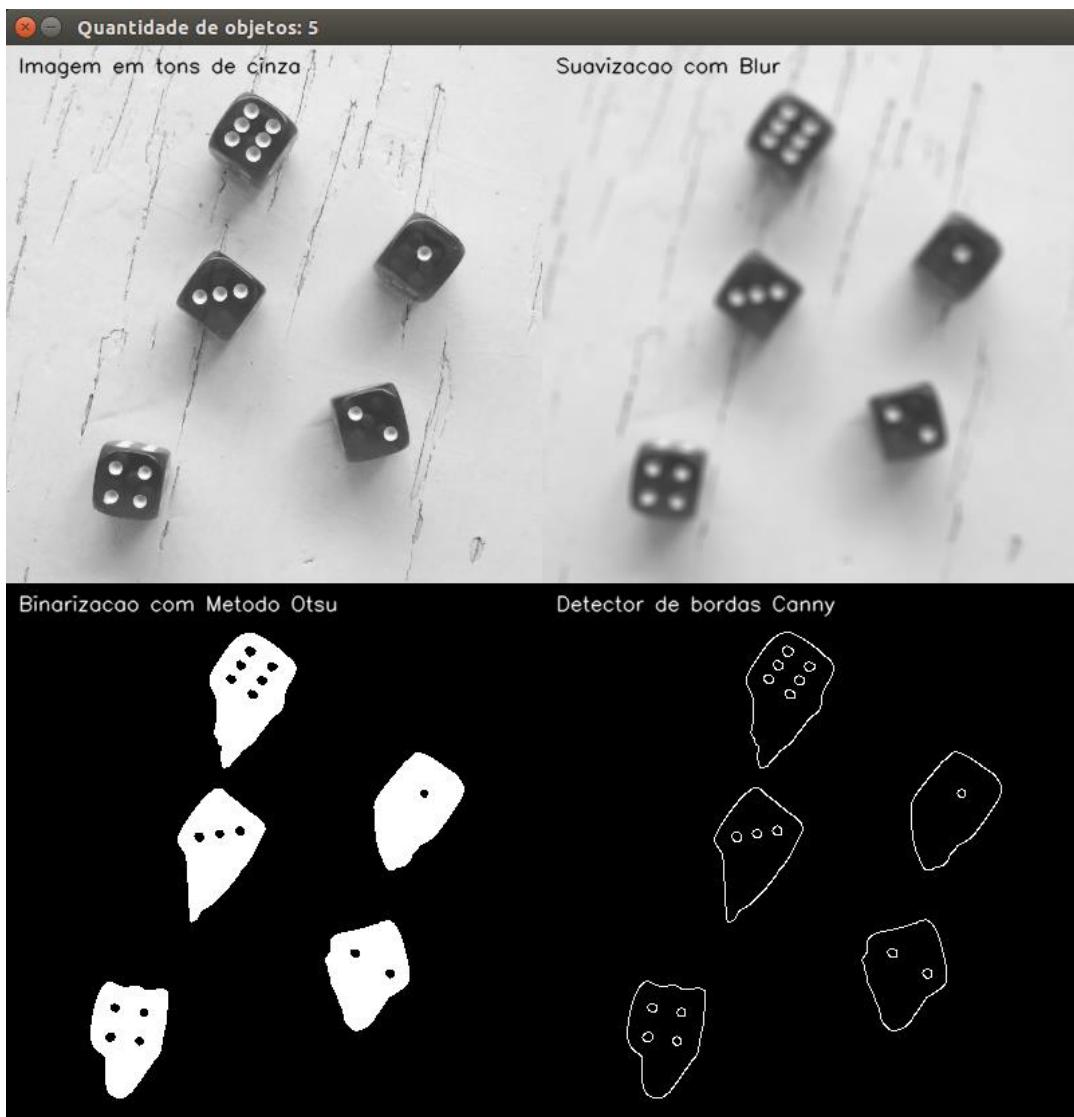


Figura 44 Passos para identificar e contar os dados na imagem.

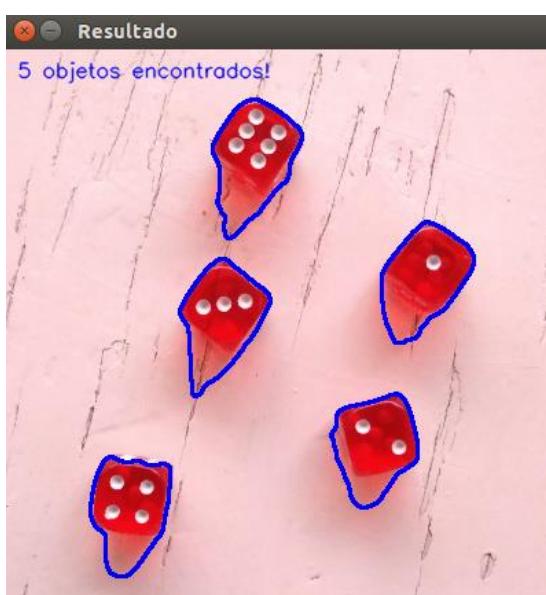


Figura 45 Resultado sobre a imagem original.

O código para gerar as saídas acima segue abaixo comentado:

```

import numpy as np
import cv2
import mahotas

#Função para facilitar a escrita nas imagem
def escreve(img, texto, cor=(255,0,0)):
    fonte = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(img, texto, (10,20), fonte, 0.5, cor, 0,
               cv2.LINE_AA)

imgColorida = cv2.imread('dados.jpg') #Carregamento da imagem

#Se necessário o redimensionamento da imagem pode vir aqui.

#Passo 1: Conversão para tons de cinza
img = cv2.cvtColor(imgColorida, cv2.COLOR_BGR2GRAY)

#Passo 2: Blur/Suavização da imagem
suave = cv2.blur(img, (7, 7))

#Passo 3: Binarização resultando em pixels brancos e pretos
T = mahotas.thresholding.otsu(suave)
bin = suave.copy()
bin[bin > T] = 255
bin[bin < 255] = 0
bin = cv2.bitwise_not(bin)

#Passo 4: Detecção de bordas com Canny
bordas = cv2.Canny(bin, 70, 150)

#Passo 5: Identificação e contagem dos contornos da imagem
#cv2.RETR_EXTERNAL = conta apenas os contornos externos
(lx, objetos, lx) = cv2.findContours(bordas.copy(),
                                      cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
#A variável lx (lixo) recebe dados que não são utilizados

escreve(img, "Imagen em tons de cinza", 0)
escreve(suave, "Suavizacao com Blur", 0)
escreve(bin, "Binarizacao com Metodo Otsu", 255)
escreve(bordas, "Detector de bordas Canny", 255)
temp = np.vstack([
    np.hstack([img, suave]),
    np.hstack([bin, bordas])
])

cv2.imshow("Quantidade de objetos: "+str(len(objetos)), temp)
cv2.waitKey(0)
imgC2 = imgColorida.copy()
cv2.imshow("Imagen Original", imgColorida)

```

```
cv2.drawContours(imgC2, objetos, -1, (255, 0, 0), 2)
escreve(imgC2, str(len(objetos))+" objetos encontrados!")
cv2.imshow("Resultado", imgC2)
cv2.waitKey(0)
```

A função *cv2.findContours()* não foi mostrada anteriormente neste livro. Encorajamos o leitor a buscar compreender melhor a função na documentação da OpenCV. Resumidamente ela busca na imagem contornos fechados e retorna um mapa que é um vetor contendo os objetos encontrados. Este mapa neste caso foi armazenado na variável ‘objetos’.

É por isso que usamos a função *len(objetos)* para contar quantos objetos foram encontrados. O terceiro argumento definido como -1 define que todos os contornos de ‘objetos’ serão desenhados. Mas podemos identificar um contorno específico sendo ‘0’ para o primeiro objeto, ‘1’ para o segundo e assim por diante.

Agora é preciso identificar qual é o lado do dado que está virado para cima. Para isso precisaremos contar quantos pontos brancos existe na superfície do dado. É possível utilizar várias técnicas para encontrar a solução. Deixaremos a implementação e testes dessa atividade a cargo do amigo leitor ;)

# 11 Detecção de faces em imagens

Uma das grandes habilidades dos seres humanos é a capacidade de rapidamente identificar padrões em imagens. Isso sem dúvida foi crucial para a sobrevivência da humanidade até os dias de hoje. Busca-se desenvolver a mesma habilidade para os computadores através da visão computacional e várias técnicas foram criadas nos últimos anos visando este objetivo.

O que há de mais moderno (estado da arte) atualmente são as técnicas de “deep learning” ou em uma tradução livre “aprendizado profundo” que envolvem algoritmos de inteligência artificial e redes neurais para treinar identificadores.

Outra técnica bastante utilizada e muito importante são os “haar-like cascades features” que traduzindo seria algo como “características em cascata do tipo haar” já que a palavra “haar” não possui tradução já que o nome deriva dos “wavelets Haar” que foram usados no primeiro detector de rosto em tempo real. Essa técnica foi criada por Paul Viola e Michael J. Jones no artigo Rapid Object Detection using a Boosted Cascade of Simple Features de 2001. O trabalho foi melhorado por Rainer Lienhart e Jochen Maydt em 2002 no trabalho An Extended Set of Haar-like Features for Rapid Object Detection.

As duas referências seguem abaixo:

Paul Viola and Michael J. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. IEEE CVPR, 2001. The paper is available online at  
[http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones\\_CVPR2001.pdf](http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones_CVPR2001.pdf)

Rainer Lienhart and Jochen Maydt. An Extended Set of Haar-like Features for Rapid Object Detection. IEEE ICIP 2002, Vol. 1, pp. 900-903, Sep. 2002. This paper, as well as the extended technical report, can be retrieved at  
<http://www.multimedia-computing.de/mediawiki//images/5/52/MRL-TR-May02-revised-Dec02.pdf>

A principal vantagem da técnica é a baixa necessidade de processamento para realizar a identificação dos objetos, o que se traduz em alta velocidade de detecção.

Historicamente os algoritmos sempre trabalharam apenas com a intensidades dos pixels da imagem. Contudo, uma publicação de Oren Papageorgio "A general framework for object detection" publicada em 1998 mostrou um recurso alternativo baseado em Haar wavelets em vez das intensidades de imagem. Viola e Jones então adaptaram a idéia de usar ondas Haar e desenvolveram as chamadas Haar-like features ou características Haar. Uma característica Haar considera as regiões retangulares adjacentes num local específico (janela de detecção) da imagem, então se processa a intensidades dos pixel em cada região e se calcula a diferença entre estas somas. Esta diferença é então usada para categorizar subseções de uma imagem.

Por exemplo, digamos que temos imagens com faces humanas. É uma característica

comum que entre todas as faces a região dos olhos é mais escura do que a região das bochechas. Portanto, uma característica Haar comum para a detecção de face é um conjunto de dois retângulos adjacentes que ficam na região dos olhos e acima da região das bochechas. A posição desses retângulos é definida em relação a uma janela de detecção que age como uma caixa delimitadora para o objeto alvo (a face, neste caso).

Na fase de detecção da estrutura de detecção de objetos Viola-Jones, uma janela do tamanho do alvo é movida sobre a imagem de entrada, e para cada subseção da imagem é calculada a característica do tipo Haar. Essa diferença é então comparada a um limiar aprendido que separa não-objetos de objetos. Como essa característica Haar é apenas um classificador fraco (sua qualidade de detecção é ligeiramente melhor que a suposição aleatória), um grande número de características semelhantes a Haar são necessárias para descrever um objeto com suficiente precisão. Na estrutura de detecção de objetos Viola-Jones, as características de tipo Haar são, portanto, organizadas em algo chamado cascata de classificadores para formar classificador forte. A principal vantagem de um recurso semelhante ao Haar sobre a maioria dos outros recursos é a velocidade de cálculo. Devido ao uso de imagens integrais, um recurso semelhante a Haar de qualquer tamanho pode ser calculado em tempo constante (aproximadamente 60 instruções de microprocessador para um recurso de 2 retângulos).

Uma característica Haar-like pode ser definida como a diferença da soma de pixels de áreas dentro do retângulo, que pode ser em qualquer posição e escala dentro da imagem original. Esse conjunto de características modificadas é chamado de características de 2 retângulos. Viola e Jones também definiram características de 3 retângulos e características de 4 retângulos. Cada tipo de recurso pode indicar a existência (ou ausência) de certos padrões na imagem, como bordas ou alterações na textura. Por exemplo, um recurso de 2 retângulos pode indicar onde a borda está entre uma região escura e uma região clara.

A OpenCV já possui o algoritmo pronto para detecção de Haar-like features, contudo, precisamos dos arquivo XML que é a fonte dos padrões para identificação dos objetos. A OpenCV já oferece arquivos prontos que identificam padrões como faces e olhos. Em [github.com/opencv/opencv/tree/master/data/haarcascades](https://github.com/opencv/opencv/tree/master/data/haarcascades) é possível encontrar outros arquivos para identificar outros objetos.

Abaixo veja um exemplo de código que utiliza a OpenCV com Python para identificar faces:

```
#Carrega arquivo e converte para tons de cinza
i = cv2.imread('iamgem.jpg')
iPB = cv2.cvtColor(i, cv2.COLOR_BGR2GRAY)

#Criação do detector de faces
df = cv2.CascadeClassifier('xml/frontalface.xml')

#Executa a detecção
faces = df.detectMultiScale(iPB,
    scaleFactor = 1.05, minNeighbors = 7,
    minSize = (30,30), flags = cv2.CASCADE_SCALE_IMAGE)

#Desenha retangulos amarelos na iamgem original (colorida)
for (x, y, w, h) in faces:
```

```
cv2.rectangle(i, (x, y), (x + w, y + h), (0, 255, 255), 7)

#Exibe imagem. Título da janela exibe número de faces
cv2.imshow(str(len(faces))+' face(s) encontrada(s)', imgC)
cv2.waitKey(0)
```

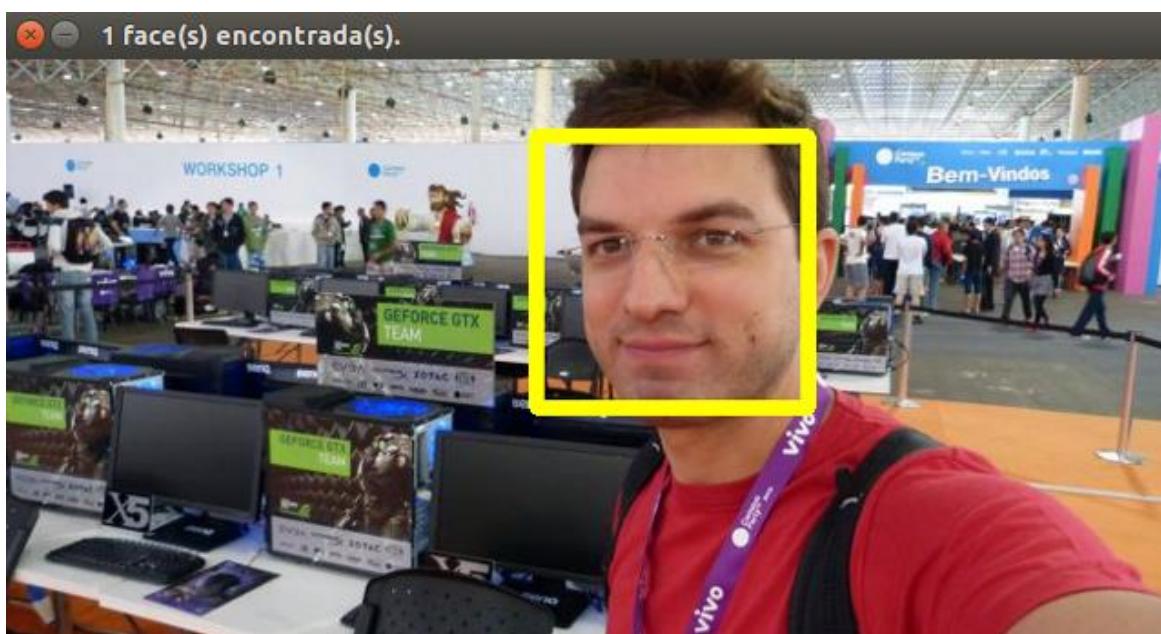


Figura 46 Identificação de face frontal.

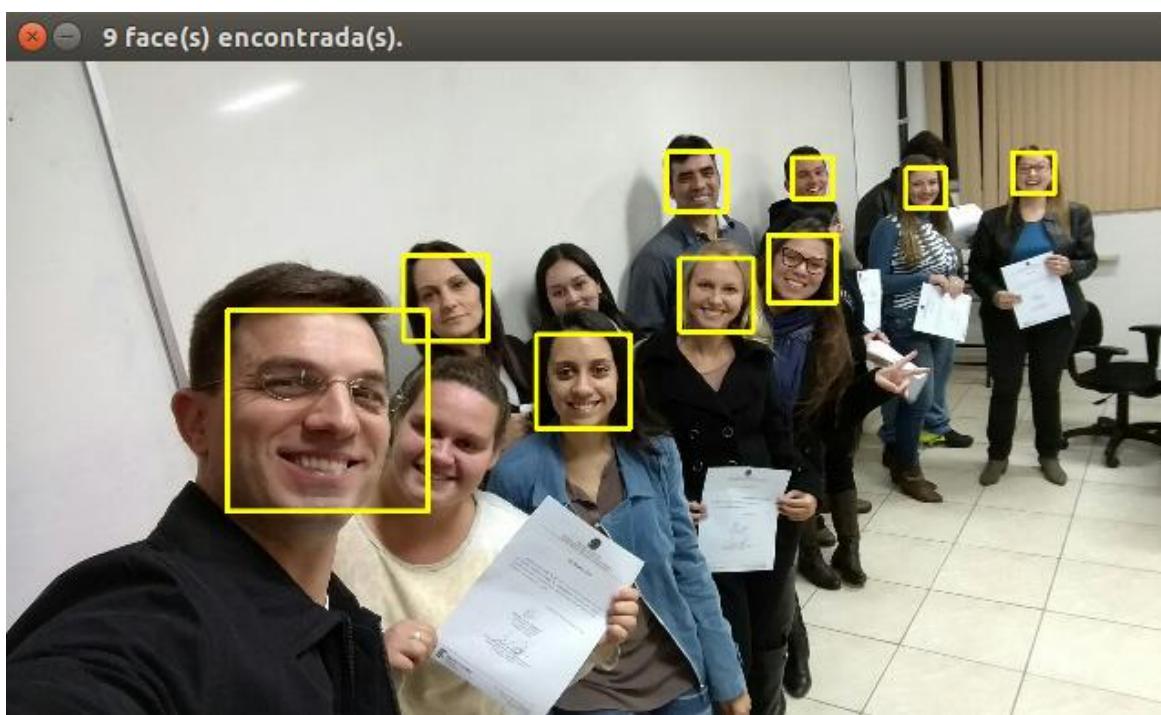


Figura 47 Identificação de face frontal.

Os argumentos que precisam ser informados para o método de detecção além do arquivo XML que contém a descrição do objeto seguem abaixo:

- **ScaleFactor:** Quanto o tamanho da imagem é reduzido em cada busca da imagem. Isso é necessário porque podem ter objetos grandes (próximos) ou menores (mais distantes) na imagem. Um valor de 1,05 indica que a imagem será reduzida em 5% de cada vez.
- **minNeighbors:** Quantos vizinhos cada janela deve ter para a área na janela ser considerada um rosto. O classificador em cascata detectará várias janelas ao redor da face e este parâmetro controla quantas janelas positivas são necessárias para se considerar como um rosto válido.
- **minSize:** Uma tupla de largura e altura (em pixels) indicando o tamanho mínimo da janela para que caixas menores do que este tamanho serão ignoradas.

Abaixo temos um código completo com varredura de diretório, ou seja, é possível repassar um diretório para busca e o algoritmo irá procurar por todas as imagens do diretório, identificar as faces e criar um retângulo sobre as faces encontradas:

```
import os, cv2

#Faz a varredura do diretório imagens buscando arquivos JPG, JPEG e
#PNG.
diretorio = 'imagens'
arquivos = os.listdir(diretorio)
for a in arquivos:
    if a.lower().endswith('.jpg') or a.lower().endswith('.png') or
    a.lower().endswith('.jpeg'):
        imgC = cv2.imread(diretorio+'/'+a)
        imgPB = cv2.cvtColor(imgC, cv2.COLOR_BGR2GRAY)

        df =
cv2.CascadeClassifier('xml/haarcascade_frontalface_default.xml')
faces = df.detectMultiScale(imgPB,
                           scaleFactor = 1.2, minNeighbors = 7,
                           minSize = (30,30), flags = cv2.CASCADE_SCALE_IMAGE)

for (x, y, w, h) in faces:
    cv2.rectangle(imgC, (x, y), (x + w, y + h), (0, 255, 255), 7)
alt = int(imgC.shape[0]/imgC.shape[1]*640)
imgC = cv2.resize(imgC, (640, alt), interpolation =
cv2.INTER_CUBIC)

cv2.imshow(str(len(faces))+' face(s) encontrada(s).', imgC)
cv2.waitKey(0)
```

## 12 Detecção de faces em vídeos

O processo de detecção de objetos em vídeos é muito similar a detecção em imagens. Na verdade um vídeo nada mais é do que um fluxo contínuo de imagens que são enviadas a partir da fonte como uma webcam ou ainda um arquivo de vídeo mp4.

Um looping é necessário para processar o fluxo contínuo de imagens do vídeo. Para facilitar a compreensão veja o exemplo abaixo:

```
import cv2

def redim(img, largura): #função para redimensionar uma imagem
    alt = int(img.shape[0]/img.shape[1]*largura)
    img = cv2.resize(img, (largura, alt), interpolation =
cv2.INTER_AREA)
    return img

#Cria o detector de faces baseado no XML
df = cv2.CascadeClassifier('xml/haarcascade_frontalface_default.xml')

#Abre um vídeo gravado em disco
camera = cv2.VideoCapture('video.mp4')

#Também é possível abrir a próprio webcam
#do sistema para isso segue código abaixo
#camera = cv2.VideoCapture(0)

while True:
    #read() retorna 1-Se houve sucesso e 2-O próprio frame
    (sucesso, frame) = camera.read()
    if not sucesso: #final do vídeo
        break
    #reduz tamanho do frame para acelerar processamento
    frame = redim(frame, 320)
    #converte para tons de cinza
    frame_pb = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    #detecta as faces no frame
    faces = df.detectMultiScale(frame_pb, scaleFactor = 1.1,
minNeighbors=3, minSize=(20,20), flags=cv2.CASCADE_SCALE_IMAGE)
    frame_temp = frame.copy()
    for (x, y, lar, alt) in faces:
        cv2.rectangle(frame_temp, (x, y), (x + lar, y + alt), (0,
255, 255), 2)
    #Exibe um frame redimensionado (com perca de qualidade)
    cv2.imshow("Encontrando faces...", redim(frame_temp, 640))
    #Espera que a tecla 's' seja pressionada para sair
    if cv2.waitKey(1) & 0xFF == ord("s"):
        break

#fecha streaming
camera.release()
cv2.destroyAllWindows()
```



**Figura 48 Detecção em vídeo.**

## 13 Rastreamento de objetos em vídeos

O rastreamento de objetos é muito útil em aplicações reais. Realizar o rastreamento envolve identificar um objeto e após isso acompanhar sua trajetória. Uma das maneiras para identificar um objeto é utilizar a técnica das características Haar-like. Outra maneira, ainda mais simples é simplesmente definir uma cor específica para rastrear um objeto.

O código abaixo realiza essa tarefa. O objetivo é identificar e acompanhar um objeto azul na tela. Perceba que a cor azul pode ter várias tonalidades e é por isso que a função cv2.inRange() é tão importante. Essa função recebe uma cor azul-claro e outra azul-escuro e tudo que estiver entre essas duas tonalidades será identificado como sendo parte de nosso objeto. A função retorna uma imagem binarizada. Veja o exemplo abaixo:

```
import numpy as np
import cv2

azulEscuro = np.array([100, 67, 0], dtype = "uint8")
azulClaro = np.array([255, 128, 50], dtype = "uint8")
#camera = cv2.VideoCapture(args["video"])
camera = cv2.VideoCapture('video.mp4')
while True:
    (sucesso, frame) = camera.read()
    if not sucesso:
        break
    obj = cv2.inRange(frame, azulEscuro, azulClaro)
    obj = cv2.GaussianBlur(obj, (3, 3), 0)
    (_, cnts, _) = cv2.findContours(obj.copy(),
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(cnts) > 0:
        cnt = sorted(cnts, key = cv2.contourArea, reverse =
True)[0]
        rect = np.int32(cv2.boxPoints(cv2.minAreaRect(cnt)))
        cv2.drawContours(frame, [rect], -1, (0, 255, 255),
2)
        cv2.imshow("Tracking", frame)
        cv2.imshow("Binary", obj)
        if cv2.waitKey(1) & 0xFF == ord("q"):
            break

camera.release()
cv2.destroyAllWindows()
```

A linha “sorted(cnts, key = cv2.contourArea, reverse = True)[0]” garante que apenas o maior contorno seja rastreado.

Outra possibilidade é utilizar um identificador Haar-like para rastrear um objeto. O procedimento é análogo ao algoritmo exposto acima. Mas ao invés da função inRange() utilizaremos a função *detectMultiScale()*.

# 14 Reconhecimento de caracteres

Neste capítulo contamos com a colaboração de Leonardo Leite através do artigo do qual fui coautor “Identificação automática de placa de veículos através de processamento de imagem e visão computacional”. O artigo foi gerado baseado em estudos de um projeto de pesquisa do qual fui orientador e esta em fase de publicação. Nas próximas versões deste livro iremos publicar o endereço eletrônico da publicação. Parte do artigo é transcrita abaixo, notadamente o que trata da biblioteca PyTesseract e da identificação e reconhecimento de caracteres de placas de veículos.

## 14.1 Biblioteca PyTesseract

O Tesseract-OCR é uma biblioteca de código aberto desenvolvida pela Google, originalmente para a linguagem C++. O seu objetivo é a leitura de textos e caracteres de uma imagem. Ela é capaz de transformar a imagem de um texto em um arquivo .txt do mesmo.

Graças a comunidade de programadores, essa biblioteca foi modificada, permitindo o funcionamento da mesma em programação Python. Por tal motivo, o nome desta biblioteca para a linguagem Python tornou-se “PyTesseract”.

O funcionamento o PyTesseract se deve a diversos arquivos externos, chamados de dicionários. Nestes arquivos há diferentes tipos de fontes de letras e diferentes combinações de palavras. Assim é possível que o programa faça a leitura de qualquer frase ou caractere que se encaixe nesses arquivos.

## 14.2 Padronização de placas

O Brasil passou por diversos padrões de placas desde o surgimento dos carros no país, no ano de 1901. Naquela época as placas continham apenas números e uma letra, que especificava se o veículo era particular, de aluguel, dentre outras especificações. Uma característica desse momento é que as placas eram emitidas pelas prefeituras, permitindo assim, placas iguais em diversos locais do Brasil.



**Figura 49 Carro Antigo (2017) com uma placa originária do primeiro sistema de padronização**

A partir da década de 90, o Brasil mudou o padrão para o que temos até os dias atuais. O motivo para tal mudança é a quantidade de carros em território nacional. Este novo modelo

de placa, é formado por 3 letras e 4 números inteiros. É possível que haja 150 milhões combinações possíveis, permitindo em teoria, essa mesma quantidade em carros.



**Figura 50 Padrão atual brasileiro de placas de veículos (ITARO, 2016).**

As placas brasileiras possuem em seu topo a cidade e a unidade federativa de origem, e abaixo 3 letras e 4 números. Vale ressaltar que dessas 3 letras, caracteres com acento e cedilha não são considerados. Todos esses caracteres são escritos com a fonte “*Mandatory*”, a mesma utilizada pelas nações da União Europeia.

### 14.3 Filtros do OpenCV

As imagens de placas apresentam ruídos intensos, que para um programa de identificação são extremamente prejudiciais à leitura. Para uma correção utiliza-se a função “*Threshold*”, que serve nesse caso como uma espécie de filtro, intensificando o que é importante na imagem.

O código de filtragem pode ser visto na tabela abaixo.

```
import cv2
import pytesseract
from PIL import Image
import string
import re

img2 = cv2.imread("/home/pyimagesearch/Desktop/placa-carro.jpg")
img = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

(T,Thresh1) = cv2.threshold(img, 44, 54, cv2.THRESH_TRUNC)
(T,Thresh3) = cv2.threshold(Thresh1, 43, 44, cv2.THRESH_BINARY)
(T,Thresh2) = cv2.threshold(Thresh3, 0 ,255,
                           cv2.ADAPTIVE_THRESH_GAUSSIAN_C)
(T,Thresh4) = cv2.threshold(Thresh2, 30, 255, cv2.CALIB_CB_ADAPTIVE_THRESH)

cv2.imshow("Imagen 01", Thresh4)
cv2.waitKey(0)
```



**Figura 51 Placa sem filtro (O autor).**



**Figura 52 Placa com filtro (O autor).**

## 14.4 Criação do dicionário

As placas nacionais são escritas pela fonte Mandatory, como já mencionado anteriormente. Infelizmente não há nenhum dicionário do *PyTesseract* destinado à leitura de tais caracteres. Para isso utilizou-se programas externos à biblioteca, criados pela comunidade. Os programas utilizados foram:

- **jTessBoxEditor:** este é responsável por salvar os algoritmos de cada caractere em um arquivo “.TIFF”.
- **Serak Tesseract Trainer:** este é responsável por unir todos os arquivos “.TIFF” necessários em um arquivo “.traineddata”, formato padrão do *PyTesseract*.

## 14.5 jTessBoxEditor

A primeira etapa deste processo é colocar os caracteres em um arquivo “.txt” para depois importá-lo para o programa. O programa abre esse arquivo, lê cada caractere e salva suas coordenadas para, mais tarde, realizar a localização dos mesmos em uma imagem. Esse processo pode ser visto na imagem 06:

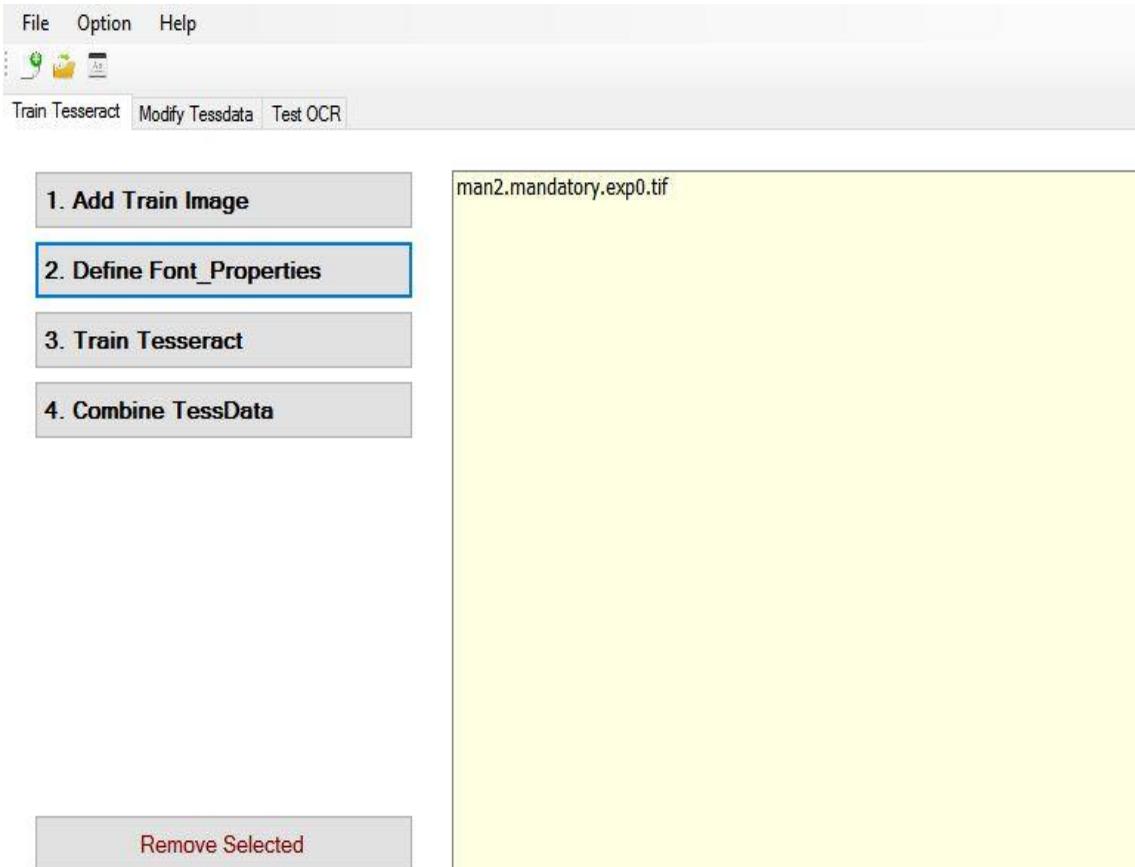
	Box Coordinates			Box View	
	Char	X	Y	Width	Height
1	A	105	127	48	76
2	B	196	127	47	76
3	C	285	127	48	76
4	D	377	127	47	76
5	E	466	127	48	76
6	F	556	127	48	76
7	G	646	127	48	76
8	H	737	127	48	76
9	I	827	127	13	76
10	J	883	127	48	76
11	K	973	127	48	76
12	L	1063	127	48	76
13	M	1153	127	48	76
14	N	1243	127	48	76
15	O	1334	127	48	76
16	P	1424	127	48	76
17	Q	1514	127	48	76
18	R	1604	127	48	76
19	S	1695	127	48	76
20	T	1786	127	47	76
21	U	1875	127	48	76
22	V	1965	127	48	76
23	W	2056	127	48	76
24	X	2146	127	48	76
25	Y	2236	127	47	76
26	Z	2326	127	48	76
27	-	2417	159	27	13
28	0	105	279	48	76
29	1	195	279	13	76
30	2	251	279	48	76
31	3	341	279	48	76
32	4	431	279	48	76
33	5	522	279	48	76
34	6	612	279	48	76
35	7	702	279	48	76
36	8	792	279	48	76
37	9	883	279	48	76

**Figura 53 Programa JTessBoxEditor em funcionamento (O autor).**

Para a realização deste projeto utilizou-se apenas os caracteres do alfabeto, os números de 0 e 9 e o hífen, pois são apenas estes caracteres que estão presentes em uma placa de carro.

## 14.6 Serak Tesseract Trainer

Este programa também foi criado pela comunidade, possuindo seu código aberto a todos. Seu nome deve-se ao seu criador Serak Shiferaw. Sua função é transformar o arquivo criado pelo *jTessBoxEditor* em um arquivo “.traineddata” para a biblioteca do *PyTesseract*. O funcionamento do programa pode ser visto na imagem 07.



**Figura 54 Programa “Serak Tesseract Trainer” em funcionamento (O autor).**

Este programa também permite colocar “padrões” de resultados, permitindo que o programa sempre tente esses padrões nas imagens. As placas brasileiras utilizam um padrão de 3 letras, um hífen e 4 números: **ABC-XXXX**.

Para isso criou-se um código básico em *Python* responsável pela criação randômica de milhares placas, para que o programa sempre tente procurar resultados parecidos com alguma dessas placas. Tal código pode ser visto abaixo e o resultado de compilação vem na sequência.

```
from random import *
letras =
["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q",
 "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
num = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
a = 0
while(a>=0):
    x = randint (0,25)
    y = randint (0,25)
    z = randint (0,25)
    a = randint (0, 9)
    b = randint (0, 9)
    c = randint (0, 9)
    d = randint (0, 9)
    res = letras[x]+letras[y]+letras[z]+“-“+num[a]+num[b]+num[c]+num[d]
    print(res)
    a+=1
```

```

XWZ-1321
YZK-8580
KSO-3087
GAU-7012
IKW-2286
HUP-7904
PGT-8193
HLA-6058
ZDL-4732
NHT-7693
RGA-7661
EEM-9290
EYT-7239
MJB-7989
DCR-6381
CAX-6205

```

**Figura 55** Algumas placas randômicas criadas pelo programa (O autor).

Por fim, o programa cria o arquivo “.traineddata” com todas essas informações já citadas, permitindo assim a leitura das placas.

## 14.7 Resultados

Tendo em mãos tudo que é necessário, criou-se um código base para leitura de placas, que contém, uma parte de filtros e outra para o reconhecimento de caracteres.

Nos primeiros testes notou-se um problema de identificação, pois as letras “I” e “O” são iguais aos números “1” e “0”, respectivamente. Por tal motivo o programa ao invés de identificar a placa como “PLA-0000”, identificava como “PLA-OOOO”. Para solucionar tal problema foi preciso dividir a *string* em duas partes: parte com letras e a parte com números. A partir daí é só colocar no código algumas linhas de substituição de caracteres. Caso a parte destinada a números tenha alguma letra, como por exemplo o “O”, substitua pelo caractere “0”. O resultado de tal compilação pode ser visto na imagem 10.



**Figura 56** Placa para identificação.

```

A placa é (sem modificações): PLA-0000
A placa é (com modificações): PLA-0000

```

```
Process finished with exit code 0
```

**Figura 57** Resultado.

Vale ressaltar que o programa ainda não foi direcionado para a leitura dos caracteres das cidades, sendo por enquanto seu objetivo apenas a leitura dos caracteres de identificação.

O código final criado pode ser visto abaixo:

```

import cv2
import pytesseract
from PIL import Image
import string
import re

img2 = cv2.imread("/home/pyimagesearch/Desktop/placa-carro.jpg")
img = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

(T,Thresh1) = cv2.threshold(img, 40, 50, cv2.THRESH_TRUNC)
(T,Thresh3) = cv2.threshold(Thresh1, 26, 44, cv2.THRESH_BINARY)
(T,Thresh2)      =           cv2.threshold(Thresh3,          0         ,255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C)
(T,Thresh4) = cv2.threshold(Thresh2, 30, 255, cv2.CALIB_CB_ADAPTIVE_THRESH)

pronta = cv2.resize(Thresh4, (300, 200))
prontal = cv2.GaussianBlur(pronta, (9,9), 1000)

cv2.imwrite("placa.jpg", prontal)
caracs = pytesseract.image_to_string(Image.open("placa.jpg"), lang="mdt")

letras = caracs[:3]
num = caracs[4:8]
num = num.replace('O', "0")
num = num.replace('I', "1")
letras = letras.replace('0', "O")
letras = letras.replace('1', "I")
num = num.replace('G', "6")
letras = letras.replace('6', "G")
num = num.replace('B', "3")
letras = letras.replace('3', "B")
num = num.replace('T', "1")
letras = letras.replace('1', "T")
print("A placa é (sem modificações): " + caracs)
print("A placa é (com modificações): " + letras + '-' + num)

```

Referências utilizadas neste capítulo:

CARRO ANTIGO (Comp.). **Década de 10.** Disponível em:  
<[http://www.carroantigo.com/portugues/conteudo/fotos\\_10.htm](http://www.carroantigo.com/portugues/conteudo/fotos_10.htm)>. Acesso em: 20 abr. 2017.

ITARO (Comp.). **Conheça os significados das placas de carro.** 2016. Disponível em: <<https://www.itaro.com.br/blog/2016/02/conheca-o-significado-das-placas-de-carro/>>. Acesso em: 20 abr. 2017.

# 15 Treinamento para identificação de objetos por Haar Cascades

No capítulo 11 utilizamos “haar-like cascades features” para identificar objetos citada. Contudo, naquela oportunidade foram utilizados classificadores (arquivos XML) que contínuam as características já definidas para determinados tipos de objetos como faces. Na web é possível encontrar outros classificadores (arquivos XML) criados para identificar outros objetos. Contudo, é fundamental para criarmos uma aplicação personalizada aprendermos a criar nossos próprios arquivos contendo as características “haar-like”, ou seja, precisamos aprender a criar nossos próprios classificadores.

Para isso, neste capítulo utilizamos como principal fonte o excelente tutorial produzido em formato de artigo intitulado: “Creating a Cascade of Haar-Like Classifiers: Step by Step” por Mahdi Rezaei do Departamento de Ciência da Computação da Universidade de Auckland na Nova Zelândia. No artigo o autor informa seu e-mail [m.rezaei@auckland.ac.nz](mailto:m.rezaei@auckland.ac.nz) e seu website [www.MahdiRezaei.com](http://www.MahdiRezaei.com) para consulta.

A sequência de passos que precisamos desenvolver para criar nosso arquivos classificador Haar-like (arquivo XML) é:

1. Criar nossa coleção de arquivos “positivos” e “negativos”.
2. Marcar os arquivos “positivos” com as coordenadas de onde exatamente estão os objetos que o classificador deve encontrar. Faremos isso com o utilitário *objectmarker.exe* ou a ferramenta *ImageClipper*.
3. Criar um arquivo de vetores (.vec) baseados nas marcações das imagens positivas usando o utilitário *createsamples.exe*.
4. Treinar o classificador usando o utilitário *haartraining.exe*.
5. Executar o classificador utilizando o *cvHaarDetectObjects()*.

## 15.1 Coletando o bando de dados de imagens

As imagens “positivas” são imagens que contém o objeto e as “negativas” são imagens que não contém o objeto. Quanto maior o número de imagens positivas e negativas melhor será a performance do seu classificador.

## 15.2 Organizando as imagens negativas

As imagens negativas são imagens que contém uma cena normal ao fundo mas que não contém o objeto que buscamos encontrar. Coloque essas imagens na pasta “..*training/negative*” e rode o arquivo utilitário *create\_list.bat* para processar as imagens. Este arquivo contém apenas um comando “dir /b \*.jpg >bg.txt” que irá gerar um arquivo texto contendo uma lista com os nomes das imagens na pasta. Este arquivo será necessário para treinar o classificador.

Abaixo temos exemplos de imagens “negativas”, lembrando que elas necessariamente precisam estar em tons de cinza.



**Figura 58 Exemplos de imagens negativas, ou seja, que não possuem o objeto pesquisado.**

### 15.3 Recortar e marcar imagens “positivas”

Agora é preciso criar um arquivo de dados (arquivo com vetores) que contenham os nomes das imagens “positivas” e a localização do objeto alvo do treinamento dentro da imagem positiva. A localização é dada por coordenadas em um arquivo texto gerado contendo em cada linha o nome do arquivo e ao lado as coordenadas dos objetos alvo.

Para automatizar esse processo é possível utilizar o *Objectmarker* ou o *Image Clipper* que seguem junto com o link disponibilizado no início deste capítulo. O primeiro é mais rápido e simples, já o segundo é mais versátil mas consome um pouco mais de tempo para configurar e executar. Vamos utilizar o *Objectmarker* neste exemplo.

Antes de proceguir confira se os arquivos estão em formato “.bmp” pois este é o formato suportado.

Coloque suas imagens na pasta “`../training/positive/rawdata`”. Na pasta “`../training/positive`” existe o arquivo `objectmaker.exe` que é preciso executar para marcar as imagens. Importante notar que as bibliotecas `cv.dll` e `highgui.dll` precisam estar na mesma pasta.

Ao iniciar o processamento a imagem aparece na tela e é preciso clicar no canto esquerdo superior e arrastar até o canto direito inferior para marcar o objeto. Pressione [Espaço] para registrar a marcação. É possível repetir a marcação se a imagem contém mais de um objeto. Após o término pressione [Enter]. Se deseja sair antes de passar por todos os arquivos pressione [Esc]. O arquivo “`info.txt`” é gerado. Note que cada vez que executar o “`objectmarker.exe`” o arquivo “`info.txt`” é sobreescrito, portanto, mantenha cópia de segurança.

### 15.4 Criando um vetor de imagens “positivas”

Na pasta “`../training`” existe um arquivo “.bat” com o nome “`samples_creation.bat`”. Este arquivo contém o seguinte comando: “`createsamples.exe -info positive/info.txt -vec vector/facevector.vec -num 200 -w 24 -h 24`” onde temos os seguintes parâmetros:

- Parâmetro 1 “`-info positive/info.txt`”: caminho para o arquivo índice de imagens positivas.
- Parâmetro 2 “`-vec vector/facevector.vec`”: caminho para a saída do vetor que será gerado.
- Parâmetro 3 “`-num 200`”: Número de objetos positivos para ser empacotado no vetor.
- Parâmetro 4 “`-w 24`”: largura dos objetos.
- Parâmetro 5 “`-h 24`”: altura dos objetos.

Importante notar que no parâmetro 3 é preciso definir o tamanho total de objetos disponíveis nas imagens positivas. Por exemplo, se você tem 100 arquivos mas cada arquivo possui 3 objetos (marcados no passo 3) então é preciso definir neste parâmetro 300.

Lembre-se que é preciso ter os arquivos cv097.dll, cxcore097.dll, highgui097.dll, e libguide40.dll nas pasta “..\\training” antes de rodar o “createsamples.exe”.

## 15.5 Haar-Training

Na pasta “..\\training” é possível modificar o arquivo “haartraining.bat” para alterar os parâmetros conforme abaixo:

Comando: “haartraining.exe -data cascades -vec vector/facevector.vec -bg negative/bg.txt -npos 204 -nneg 200 -nstages 15 -mem 1024 -mode ALL -w 24 -h 24 – nonsym”

Parâmetros:

- data cascades: caminho para guardar o “cascade”
- vec data/vector.vec: caminho do vetor
- bg negative/bg.txt: caminho da lista de arquivos negativos
- npos 200: Número de exemplos positivos  $\leq$  número de arquivos BMP.
- nneg 200: Número de exemplos negativos  $\geq$  npos
- nstages 15: Número estágios identados de treinamento
- mem 1024: Quantidade de memória associada em megabytes.
- mode ALL: Olhe na literatura para maiores informações
- w 24 -h 24: Tamanho dos exemplos devem ser os mesmos de “samples\_creation.bat”
- nonsym: Usar apenas se os objetos não são simétricos horizontalmente

O comando “haartraining.exe” coleta um novo conjunto de exemplos negativos para cada estágio e “-nneg” define o limite para o tamanho do conjunto. O programa usa a informação dos estágios anteriores para determinar qual dos “exemplos candidatos” esta mal classificado. O treinamento termina quando a proporção dos exemplos mal classificados em relação aos exemplos candidatos é menor que FR (número do estágio), esta é a condição de parada.

Independentemente do número de estágios “-nstages” que você definiu, o programa pode terminar antes se a condição de parada for alcançada. Alcançar essa condição é normalmente um bom sinal a não ser que você tenha poucos arquivos positivos para treinar (normalmente um número menor que 500 é considerado baixo).

Lembre-se que para executar o “haartraining.exe” você precisa dos arquivos cv097.dll, cxcore097.dll, e highgui097.dll na pasta “..\\training”.

As informações disponibilizadas durante o treinamento conforme imagem a seguir são listadas a abaixo:

- Parent node: define o estágio atual do processo de treinamento
- N: número de características usadas neste estágio
- %SMP: percentual de exemplos usados para esta característica
- F: “+” quando a simetria é aplicada e “-“ se a simetria não é aplicada
- ST.THR: “gatilho” do estágio (threshold)
- HR: Taxa de acerto baseada no threshold

- FA: Alarme falso baseado no threshold
- EXP. ERR: erro exponencial do classificador forte

Este parâmetros podem ser consultados na imagem a seguir que exemplifica um processo de treinamento.

```
Command Prompt
Parent node: 9

*** 1 cluster ***
POS: 200 200 1.000000
NEG: 200 0.000113522
BACKGROUND PROCESSING TIME: 7.99
Precalculation time: 6.31
+-----+-----+-----+-----+
| N |%SMP|F| ST.THR | HR | FA | EXP. ERR |
+-----+-----+-----+-----+
| 1 |100%|-|-0.837398| 1.000000| 1.000000| 0.255000|
+-----+-----+-----+-----+
| 2 |100%|+|-1.542245| 1.000000| 1.000000| 0.192500|
+-----+-----+-----+-----+
| 3 | 85%|-|-1.262312| 1.000000| 0.880000| 0.162500|
+-----+-----+-----+-----+
| 4 | 86%|+|-1.501497| 1.000000| 0.805000| 0.185000|
+-----+-----+-----+-----+
| 5 | 87%|-|-1.170155| 1.000000| 0.630000| 0.137500|
+-----+-----+-----+-----+
| 6 | 86%|+|-1.609438| 1.000000| 0.670000| 0.125000|
+-----+-----+-----+-----+
| 7 | 88%|-|-1.307158| 1.000000| 0.480000| 0.097500|
+-----+-----+-----+-----+
Stage training time: 6.67
Number of used features: 7

Parent node: 9
Chosen number of splits: 0

Total number of splits: 0

Tree Classifier
Stage
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0| 1| 2| 3| 4| 5| 6| 7| 8| 9| 10|
+-----+-----+-----+-----+-----+-----+-----+-----+
0---1---2---3---4---5---6---7---8---9---10
```

Figura 59 Processo de treinamento.

## 15.6 Criando o arquivo XML

Depois de terminar o treinamento é preciso gerar o arquivo XML. Na pasta “..../training/cascades” voce irá encontrar um conjunto de pastas. Se tudo ocorreu bem haverá uma pasta para cada estágio processado. Lembre-se que é possível que o treinamento termine antes de alcançar o número de estágios definido no início do treinamento. Por exemplo, na imagem anterior o treinamento realizado foi iniciado definindo 15 estágios mas apenas 12 estágios foram necessários para ser alcançado o erro mínimo e o treinamento parou. Então apenas 12 pastas foram geradas.

Cada pasta terá um arquivos chamado “AdaBoostCARTHaarClassifier.txt”. Agora é preciso copiar todas as pastas para “..../cascade2xml/data” para entrar gerar o XML.

O próximo passo é executar o arquivo “convert.bat” em “..../cascade2xml” que possui o comando: “haarconv.exe data detector\_de\_objeto.xml 24 24” sendo que o nome do arquivo XML pode ser alterado livremente. Lembre-se de manter a largura e altura (24 e 24) iguais as utilizadas no treinamento. Na mesma pasta será criado o arquivo XML.

Agora é necessário testar o arquivo XML realizando uma verificação da detecção.

## 16 Criação de um identificador de objetos com Haar Cascades

Em construção...