

 Pesquisar

Geraldo Barbosa do Amarante

1 emblemas, 900 pontos

[Hoje](#) [Trilhas](#) [Módulos](#) [Trailmixes](#)

Tutorial Opencv

Este tutorial apresenta conceitos importantes de processamento de imagens (filtros) e de visão computacional (segmentação, classificação, reconhecimento de padrão e rastreamento). Estes conceitos serão introduzidos utilizando a biblioteca OpenCV, que é distribuída gratuitamente e possui documentação farta na internet, com exemplos e aplicações práticas. Para utilizar essa biblioteca em python acesse o terminal com ambiente de python ativado e digite `pip install opencv-python`.

2. Tutorial Opencv

```
# 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)
```

[Copy](#)

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')
```

Copy

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);

1. Visão Computacional

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.

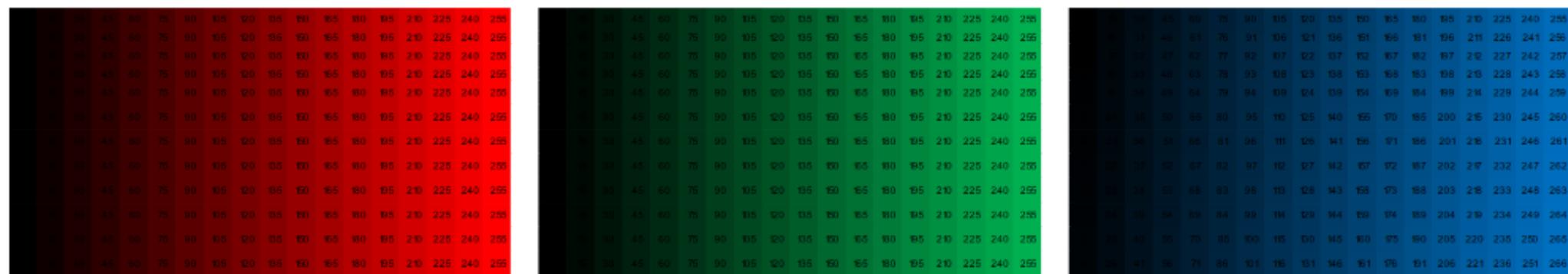


Figura 2- Na imagem temos um exemplo das 3 matrizes que compõe o sistema RGB. Cada pixel da imagem, portanto, é composto por 3 componentes de 8 bits cada, sem sinal, o que gera 256 combinações por cor. Portanto, a representação é de 256 vezes 256 vezes 256 ou 256^3 que é igual a 16,7 milhões de cores.

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
```

Copy

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

Copy

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.



Figura 4 - imagem a ser processada (ponte.jpg)

```
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)
```

Copy

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



Figura 5 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).

2. [Tutorial Opencv](#)

Com uma 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)
```

Copy

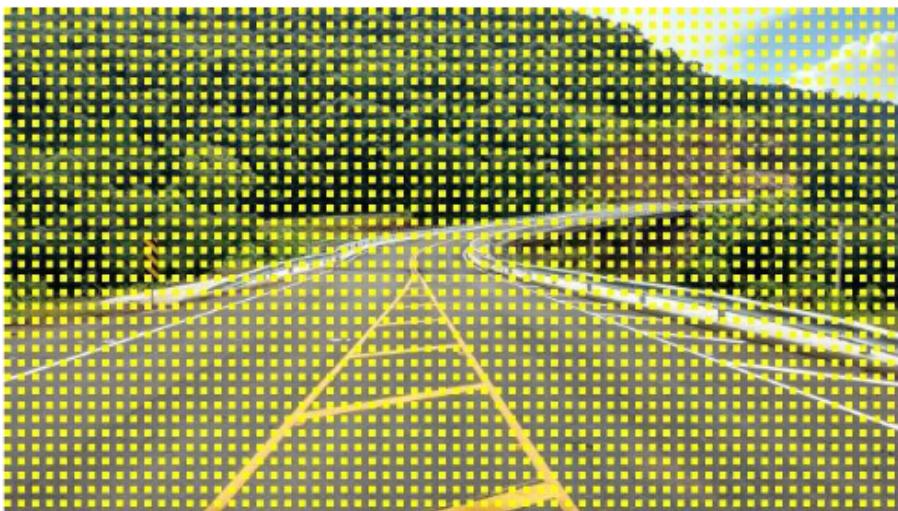


Figura 6 Código gerou quadrados amarelos de 5x5 pixels sobre a toda a imagem.
[Tutorial Opencv](#)

Cortando uma imagem / Crop

Veja o código abaixo onde criamos uma nova imagem a partir de um pedaço da imagem original (ROI) 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
```

Copy

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 7 Imagem recortada da imagem original

2. [Tutorial Opencv](#)

Redimensionamento / Resize

Para reduzir ou aumentar o tamanho da imagem, existe uma função já pronta da 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)
```

Copy

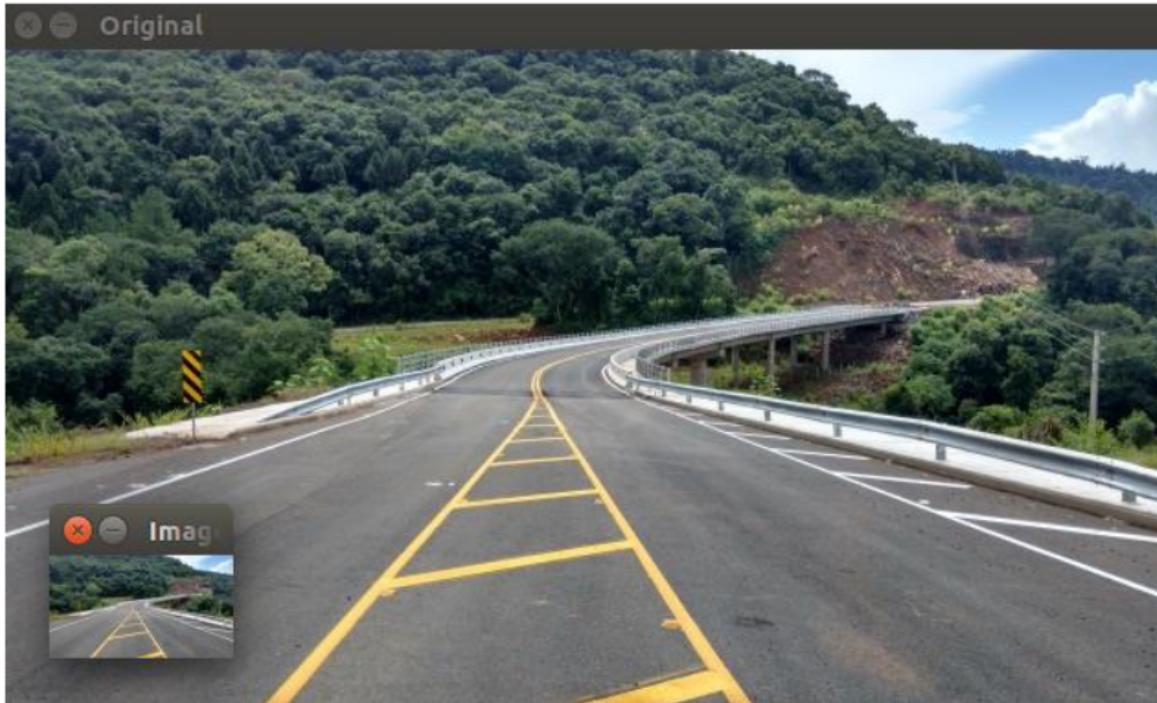


Figura 8 - No canto inferior esquerdo da imagem é possível notar a imagem redimensionada.

Veja que a função ‘rezise’ utiliza uma propriedade aqui definida com 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.

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)
```

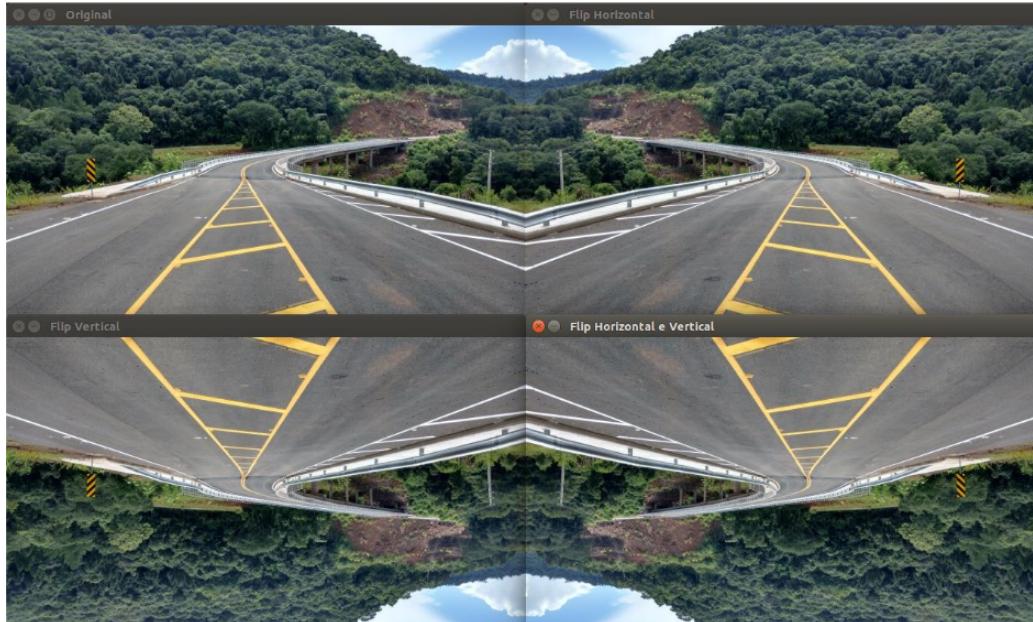
Copy

Figura 9 Resultado do flip horizontal, vertical e horizontal e vertical na mesma imagem.

Rotacionando uma imagem / Rotate

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 30 graus", img_rotacionada)  
cv2.waitKey(0)
```

Copy



Figura 10 Imagem rotacionada em 30 graus.

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.

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 :

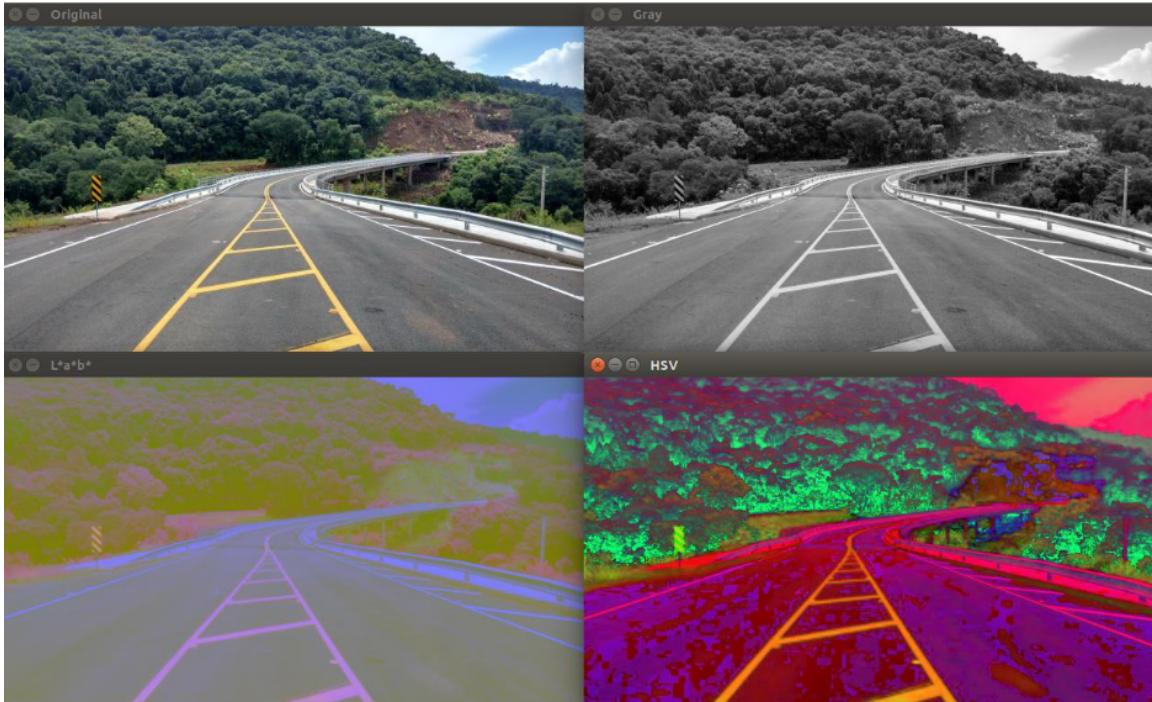


Figura 11 Outros espaços de cores com a mesma imagem.

Segue código para exibir imagens em outros formatos, alem do RGB:

```
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)
```

Copy

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)
```

Copy

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

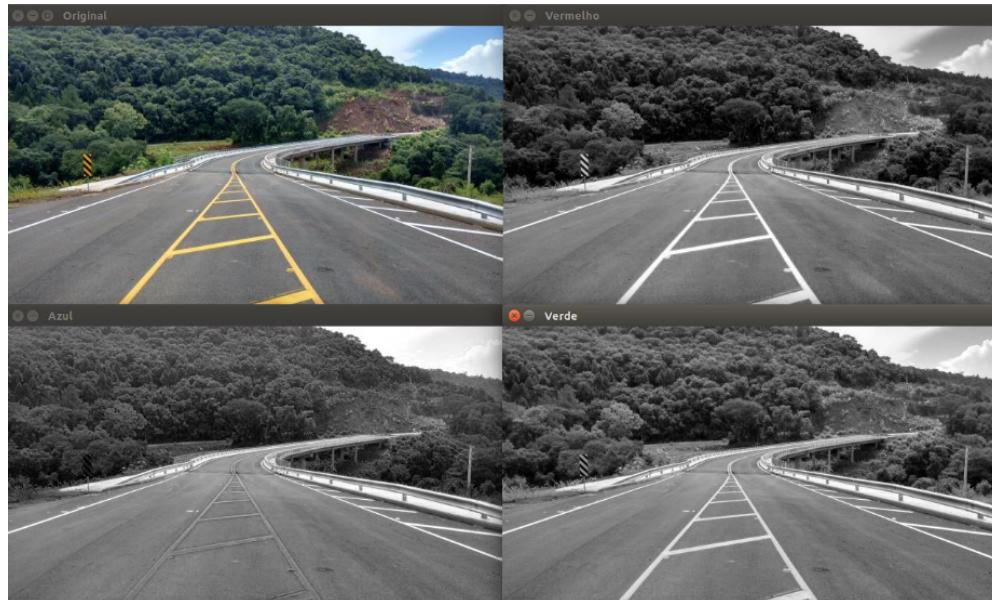


Figura 12 Perceba como a linha amarela (que é formada por verde e vermelho) fica quase imperceptível no canal azul.

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])
```

Copy

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)
```

Copy

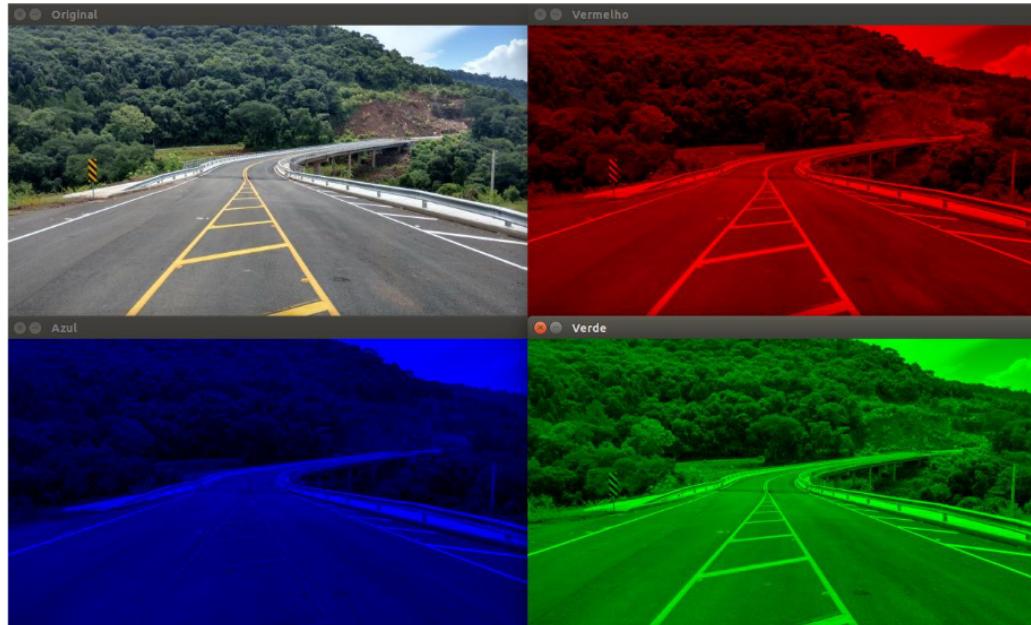


Figura 13 Exibindo os canais separadamente.

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 pixels 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 14 Imagem original já convertida para tons de cinza

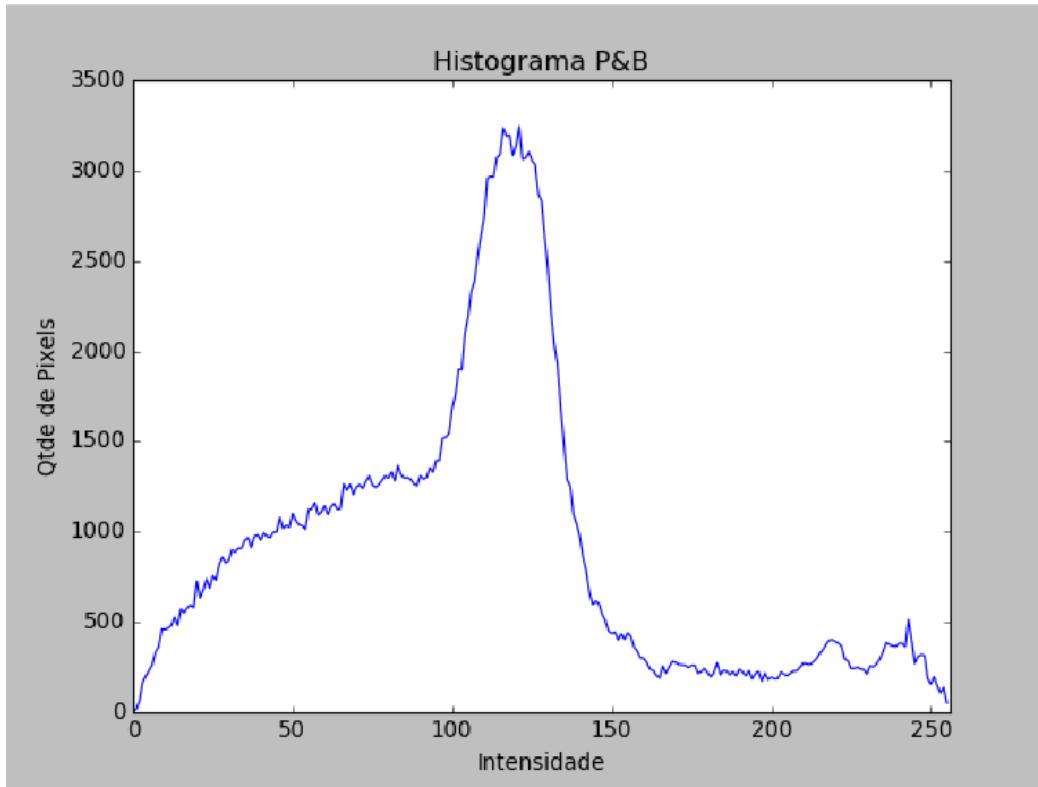


Figura 15 Histograma da imagem em tons de cinza.

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("Imagem P&B", img)
#Função calcHist para calcular o histograma 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() Computacional  
cv2.waitKey(0)
```

Copy

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()
```

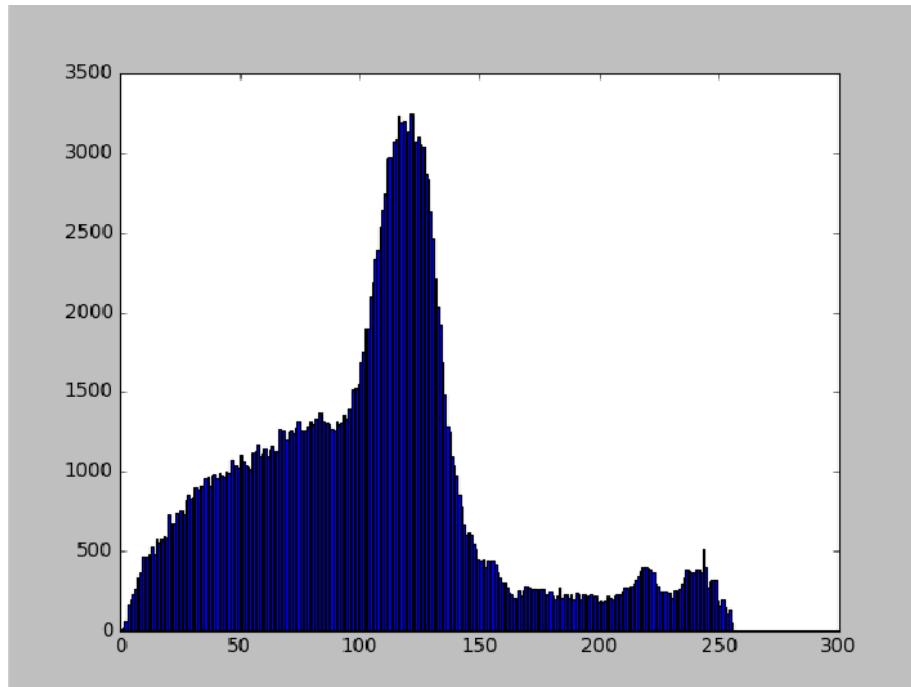
Copy

Figura 16 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() Final Opencv
```

[Copy](#)

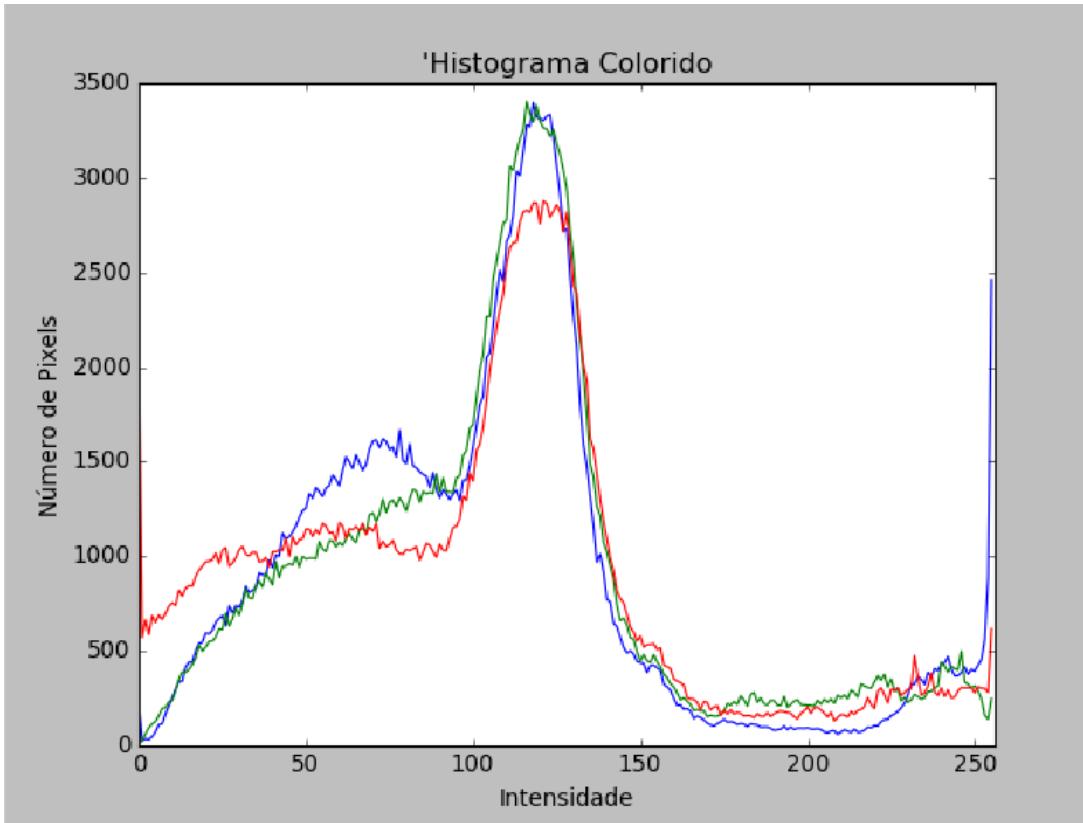


Figura 17 Histograma colorido da imagem. Neste caso são plotados os 3 canais RGB.

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:

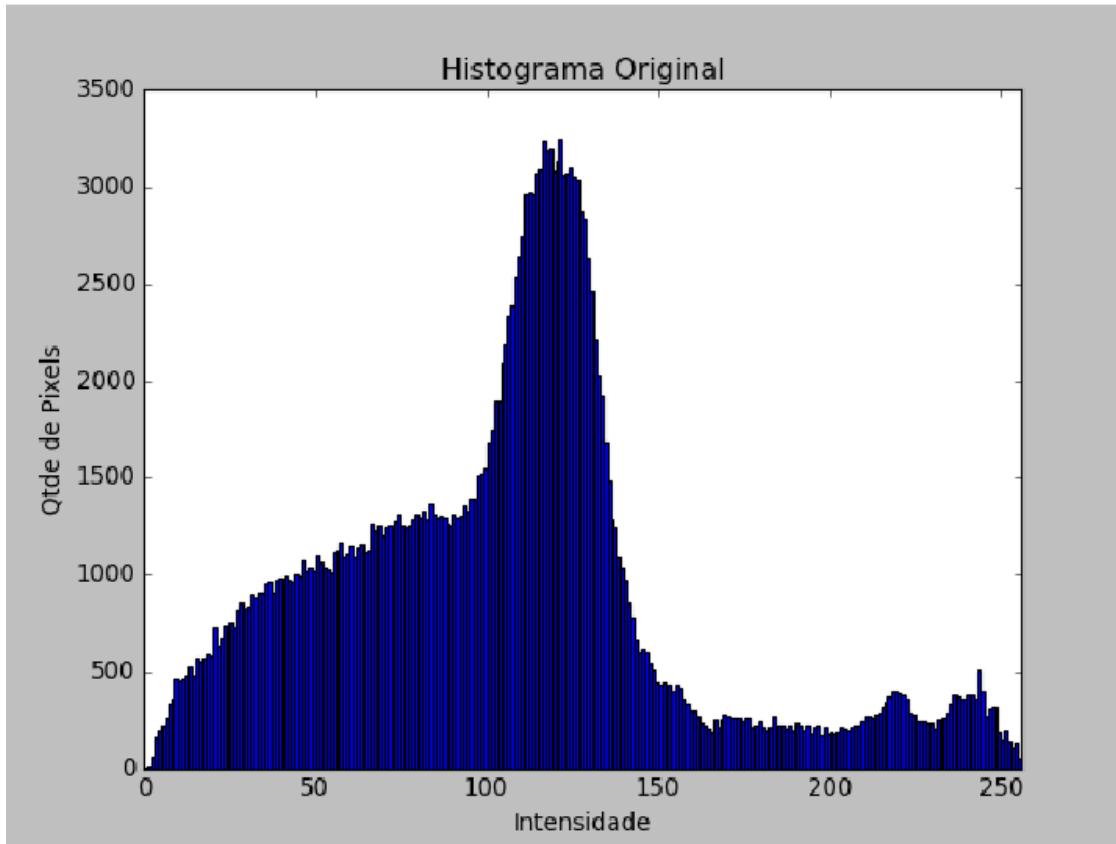


Figura 18 Histograma da imagem original.

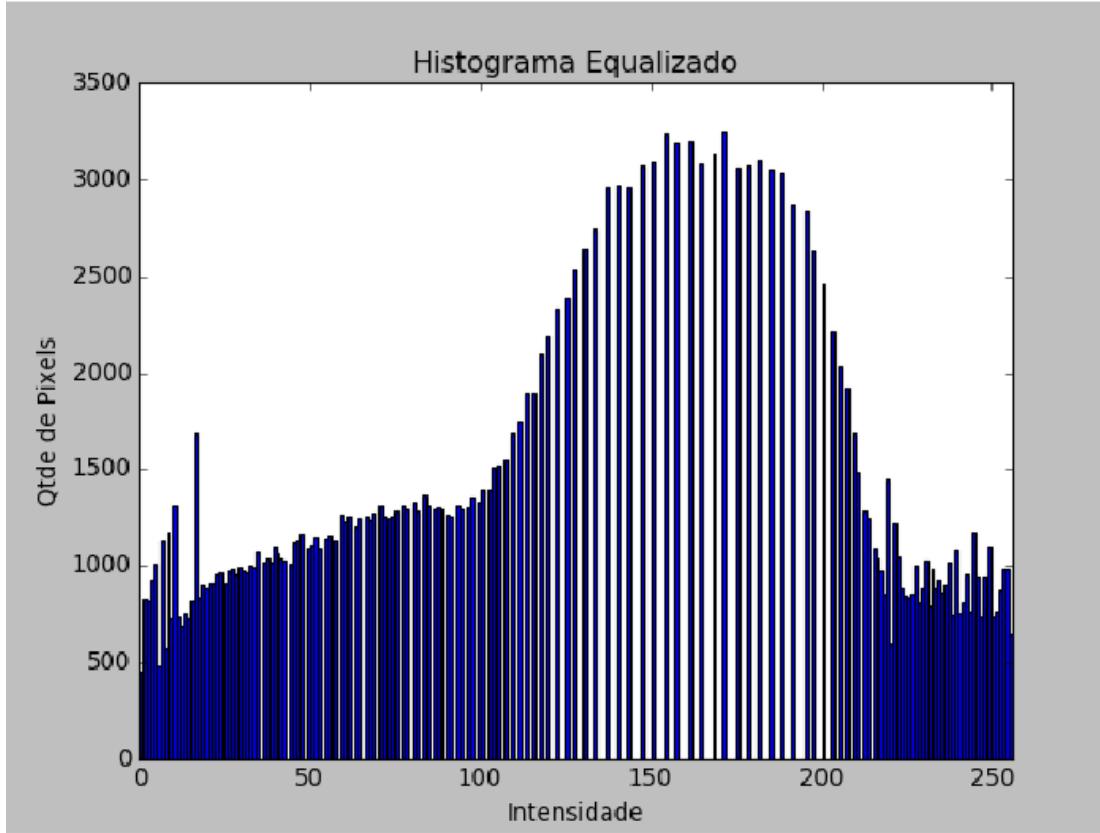


Figura 19 Histograma equalizado.

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")onal
plt.ylabel("Qtde de Pixels")
plt.hist(img.ravel(), 256, [0,256])
plt.xlim([0, 256])
plt.show()
cv2.waitKey(0)
```

Copy

Na imagem a diferença também é perceptível, veja:



Figura 20 Imagem original (acima) e imagem cujo histograma foi equalizado (abaixo). Na imagem cujo histograma foi equalizado percebemos maior contraste.

Suavização de imagens

1. Visão Computacional

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.

Suavização por cálculo da média

Neste caso é criada uma “máscara 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 máscara, ou seja, dos pixels da vizinhança. Alguns autores chamam esta máscara de janela de cálculo ou kernel (do inglês núcleo).

30	100	130
130	Pixel	160
50	100	210

Figura 21 Máscara 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,::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)
```

Copy



Figura 22 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.

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 tamanho 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 suavizadas pela mediana", suave)
cv2.waitKey(0)
```

Copy



Figura 23 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.

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.

1. Visão Computacional

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)])
])
```

]) 2. Tutorial Opencv

Copy



Figura 24 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.

Binarização com limiar

[1. Visão Computacional](#)

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)
```

[Copy](#)

[2. Tutorial Opencv](#)

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

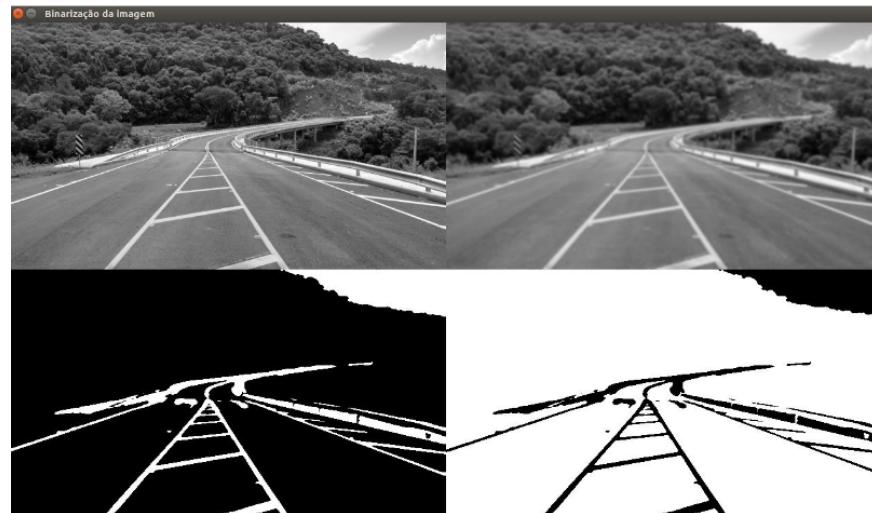


Figura 25 Da esquerda para a direta e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada e a imagem binarizada invertida.

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.

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 [1. Visão Computacional 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.

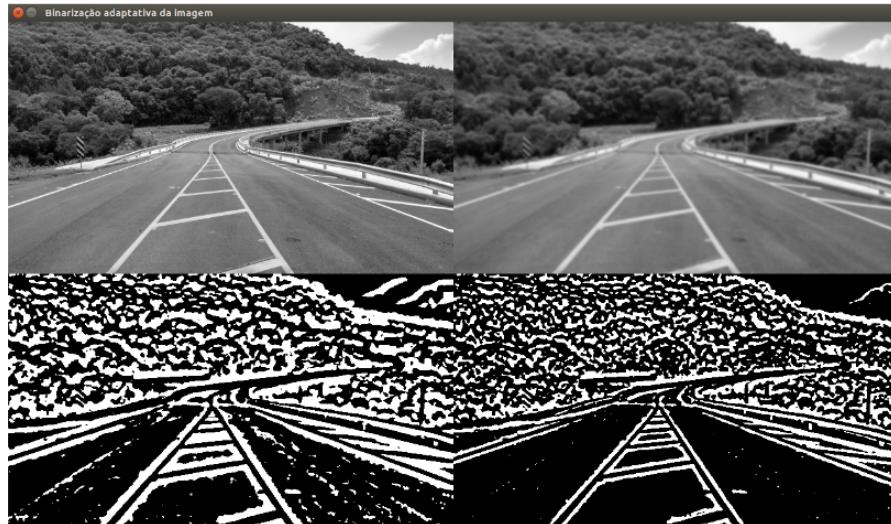


Figura 26 Threshold adaptativo. Da esquerda para a direita e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada pela média e a imagem binarizada com Gauss.

```
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)
```

[Copy](#)

Threshold com Otsu e Riddler-Calvard

Outro método que automaticamente encontra um threshold para a imagem é o método de Otsu. Neste caso ele analisa 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.

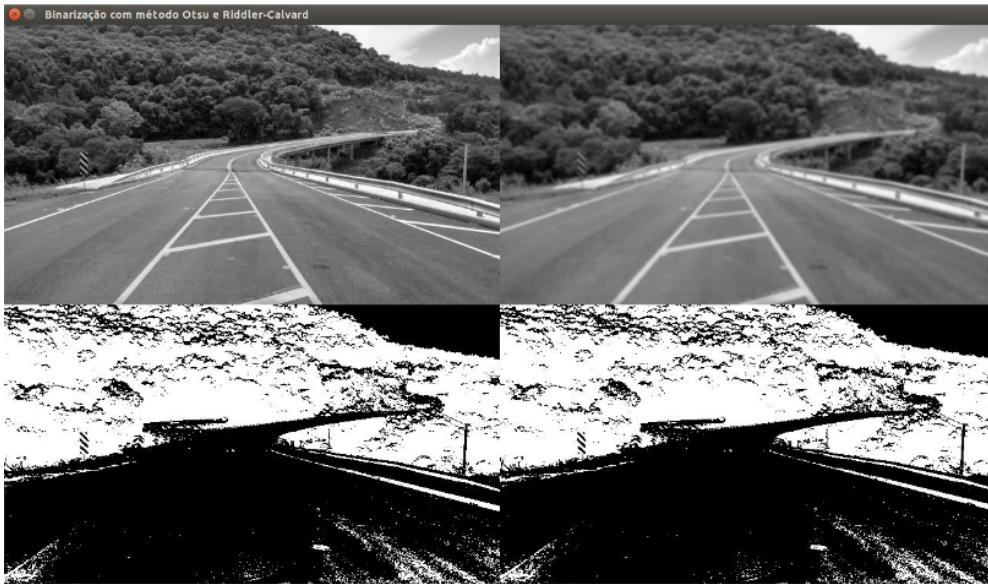


Figura 27 Threshold com Otsu e Riddler-Calvard.

```
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)
```

Copy

Segmentação e métodos de detecção de bordas

1. Visão Computacional

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().

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:

2. Tutorial Opencv

```
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)
```

Copy

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 28 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 .

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 flutuant de 64 bits com sinal para depois converter novamente para inteiro sem sinal de 8 bits.



Figura 29 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)
```

Copy

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 multiplos passos para chegar ao resultado final.

Basicamente o Canny envolve:

1. Aplicar um filtro gaussiano para suavizar a imagem e remover o ruído.
2. [Tutorial Opencv](#)
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.

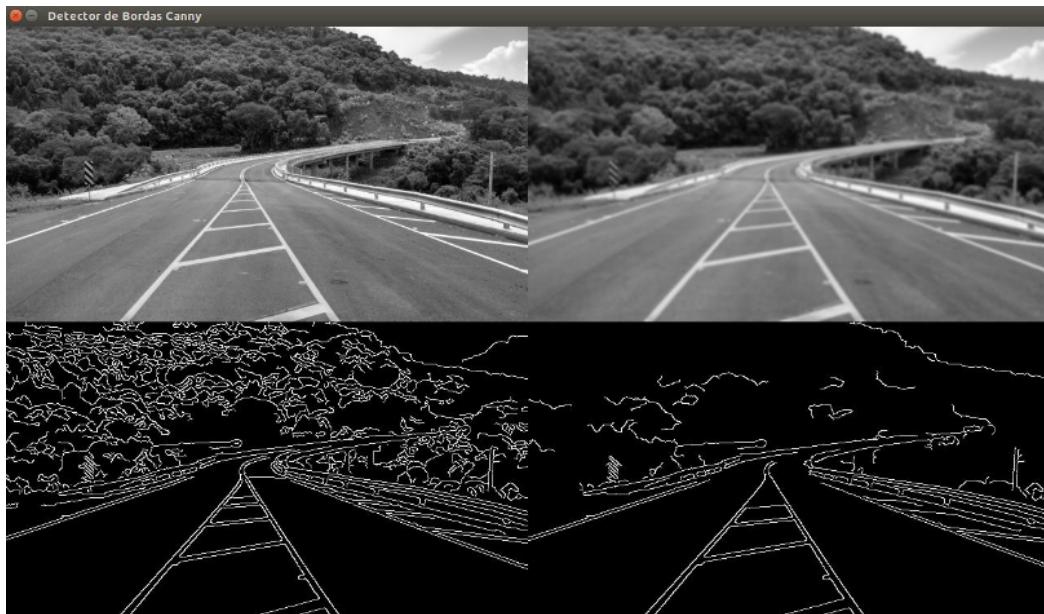


Figura 30 Filtro canny com parâmetros diferentes. A esquerda deixamos um limiar mais baixo (20,120) e à direita a imagem foi gerada com limiares maiores (70,200).

```
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)
```

[Copy](#)

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. Visão Computacional. Onde estão os dados na imagem.
2. Quantos dados foram jogados.
3. Qual é o lado que esta 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 está 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.

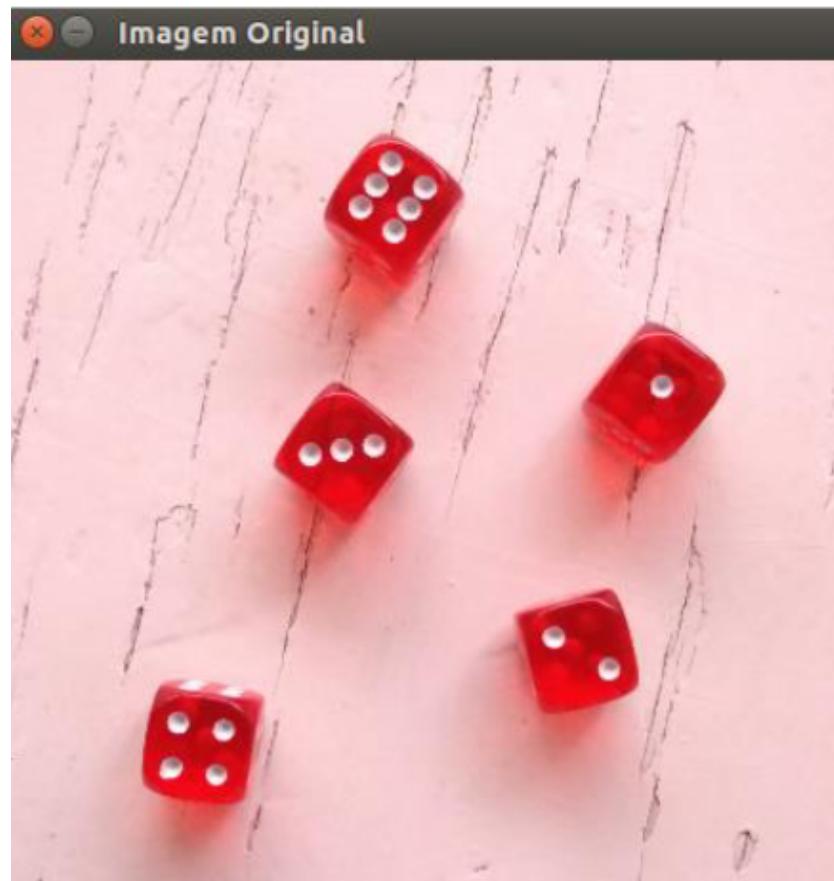


Figura 31 Imagem original. A superfície branca com ranhuras dificultará o processo.

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.

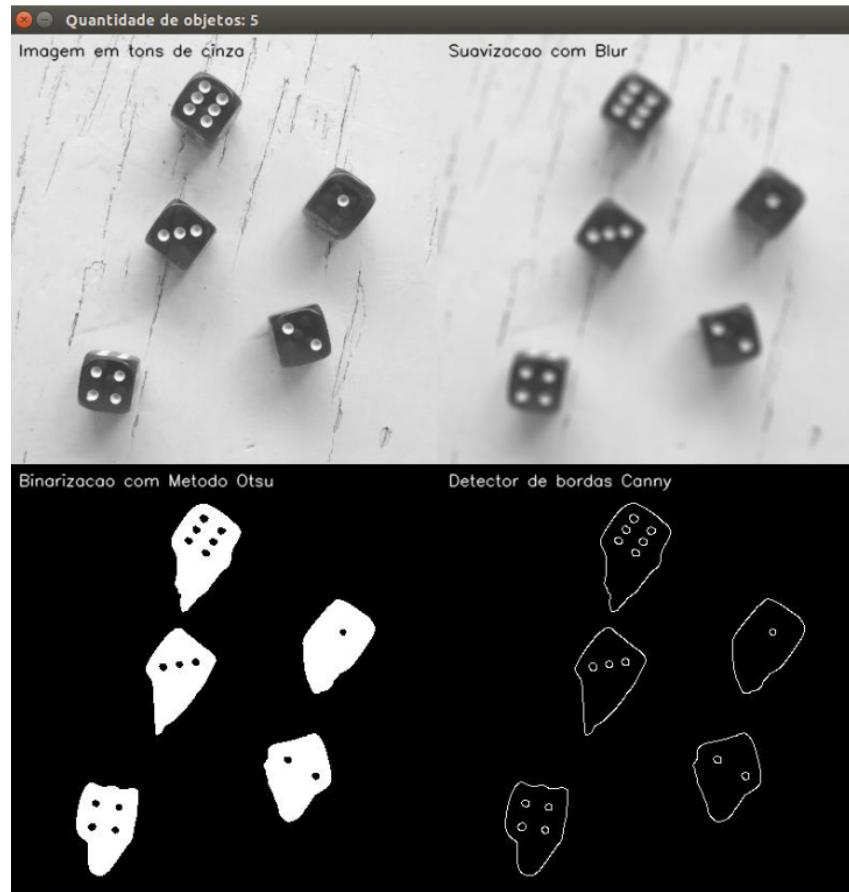


Figura 32 Passos para identificar e contar os dados na imagem.



Figura 33 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 imagens
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) OpenCV
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)

```

Copy

A função cv2.findContours() não foi mostrada anteriormente neste tutorial. 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 esta virado para cima. Para isso precisaremos contar quantos pontos brancos existe na superfície do dado. É possível utilizar várias técnicas para encontrar a solução.

1A aplicação das operações, filtro da mediana, canny e limiar de Otsu permitem respectivamente:

- A.Detectação de bordas, remoção de ruído e segmentação, respectivamente.
- B.Remoção de ruído, detecção de bordas e binarização, respectivamente
- C.Binarização, remoção de ruído e detecção de bordas, respectivamente
- D.Remoção de ruido, suavização da imagem e detecção de bordas

Confira o teste para ganhar 100 pontos

1. Visão Computacional
A segunda tentativa vale 50 pontos. A terceira tentativa e as seguintes valem 25 pontos.



Instituto Atlântico Whatsapp Academy MCTI Futuro

Português (Brasil) ▾

2. Tutorial Opencv