

Introdução a Visão Computacional com Python e OpenCV

PARTE 1 de 2

Versão 0.1a – Não corrigida

Ricardo Antonello

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

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 incapacidade de muitos alunos em lêrem 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 que me ajudou muito a ver exemplos sobre o tema.

Ricardo Antonello

Sumário

1 Bem vindo ao mundo da visão computacional.....	5
2 Sistema de coordenadas e manipulação de pixels	8
3 Fatiamento e desenho sobre a imagem	11
4 Transformações e máscaras	15
4.1 Cortando uma imagem / Crop	15
4.2 Redimensionamento / Resize.....	15
4.3 Espelhando uma imagem / Flip	17
4.4 Rotacionando uma imagem / Rotate.....	18
4.5 Máscaras	19
5 Sistemas de cores.....	20
5.1 Canais da imagem colorida.....	21
6 Histogramas e equalização de imagem.....	23
6.1 Equalização de Histograma	26
7 Suavização de imagens	30
7.1 Suavização por cálculo da média.....	30
7.2 Suavização pela Gaussiana.....	31
7.3 Suavização pela mediana.....	32
7.4 Suavização com filtro bilateral	33
8 Binarização com limiar.....	35
8.1 Threshold adaptativo	36
8.2 Threshold com Otsu e Riddler-Calvard.....	37
9 Segmentação e métodos de detecção de bordas	38
9.1 Sobel	38
9.2 Filtro Laplaciano.....	39
9.3 Detector de bordas Canny.....	40
10 Identificando e contando objetos	42
11 Em breve.....	46
11.1 Detecção de faces em imagens	46
11.2 Detecção de faces em vídeos	46
11.3 Rastreamento de objetos em vídeos.....	46
11.4 Reconhecimento de caracteres manuscritos	46
11.5 Treinamento para identificação de objetos por Haar Cascades	46
11.6 Criação de um identificador de objetos com Haar Cascades.....	46

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 obtêm 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 “enxergarem”, 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 sorindo, 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 atraso. Enfim, a visão computacional já esta 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 imagem 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 Apência 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("saida.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 'saida.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.

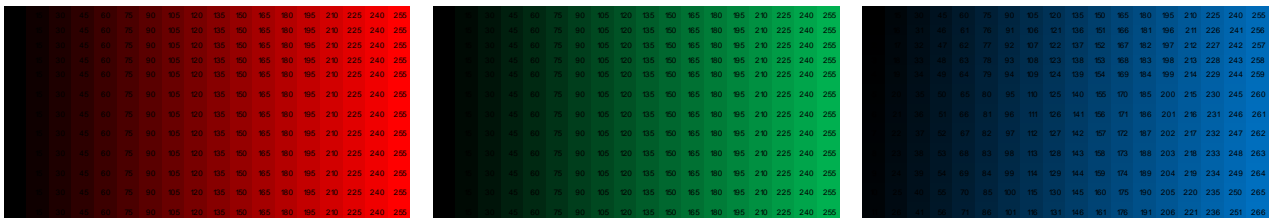


Figura 2 Na imagem temos um exemplo das 3 matrizes que compõe o sistema RGB. Cada pixel da imagem, portanto, é composto por 3 componentes de 8 bits cada, sem sinal, o que gera 256 combinações por cor. Portanto, a representação é de 256 vezes 256 vezes 256 ou 256^3 que é igual a 16,7 milhões de cores.

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 esta na posição (0,0) esta 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 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 exibí-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("Imagem modificada", imagem)
```

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

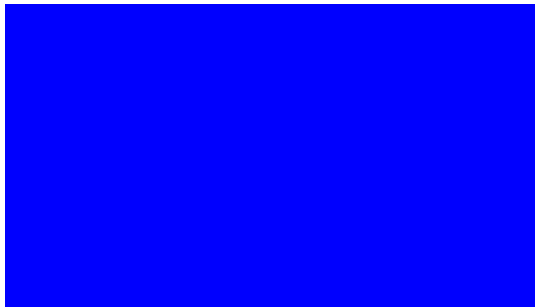


Figura 4 Imagem completamente azul pela alteração de todos os pixels para (255,0,0). Lembrando que o padrão RGB é na verdade BRG pela tupla (B, R, G).

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("Imagem modificada", imagem)
cv2.waitKey(0)
```

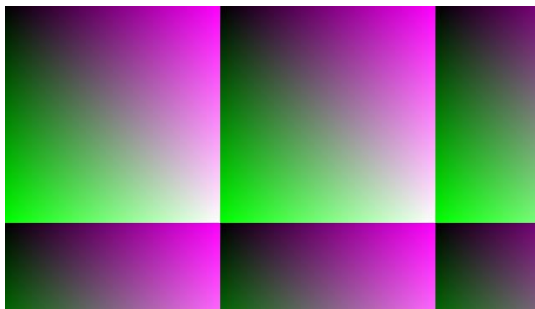


Figura 5 A alteração nas componentes das cores da imagem conforme as coordenadas de linha e coluna geram a imagem acima.

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("Imagem modificada", imagem)
cv2.waitKey(0)
```

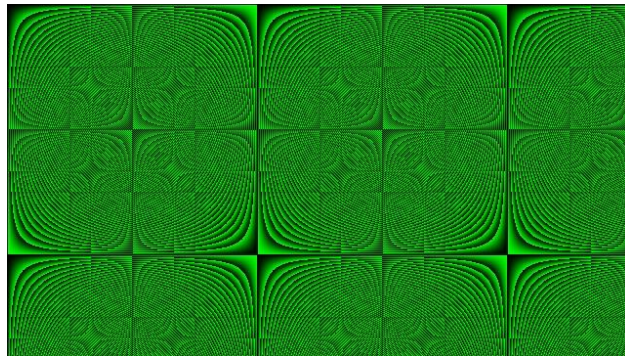


Figura 6 A alteração dinâmica da cor de cada pixel gera esta bela imagem.
A imagem original se perdeu pois todos os pixels foram alterados.

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("Imagem modificada", imagem)
cv2.waitKey(0)
```

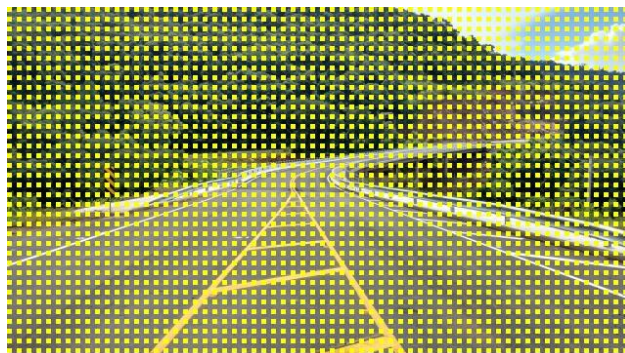


Figura 7 Código gerou quadrados amarelos de 5x5 pixels sobre a toda a imagem.

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 retangulo 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 retangulo amarelo por toda a altura da imagem
image[:, 200:220] = (0, 255, 255)

#Cria um retangulo 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("Imagem 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)

```

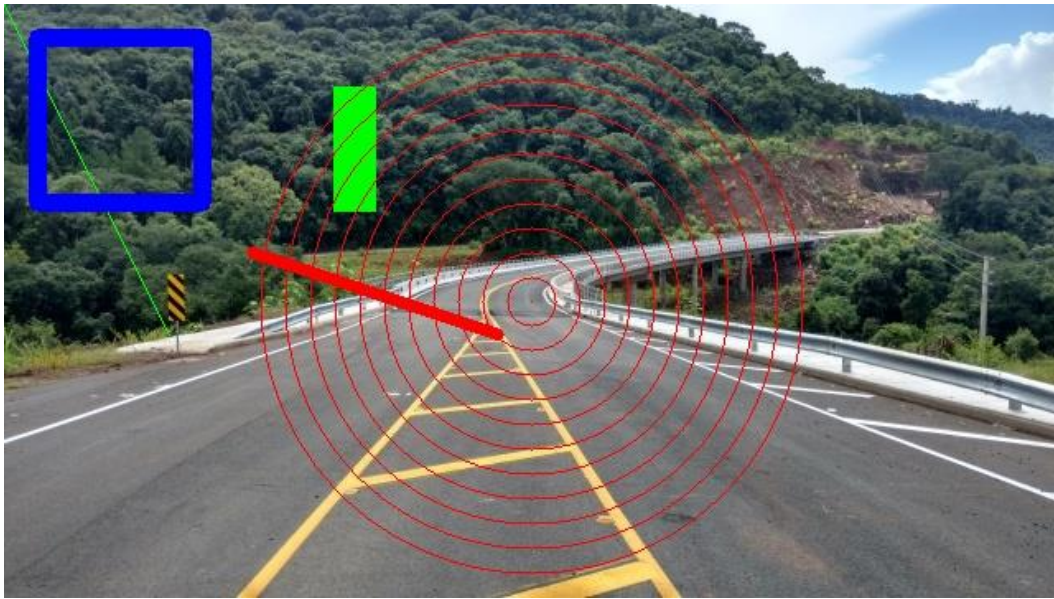


Figura 10 Utilizando funções do OpenCV para desenhar sobre a imagem.

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 cálculo 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 imagem. 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 veremos 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 salva em um arquivo em disco.

4.2 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 imutils
import cv2

img = cv2.imread('ponte.jpg')
cv2.imshow("Original", img)
proporcao = 100.0 / img.shape[1]
tamanho_novo = (100, int(img.shape[0] * proporcao))
img_redimensionada = cv2.resize(img, tamanho_novo,
interpolation = cv2.INTER_AREA)
```

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

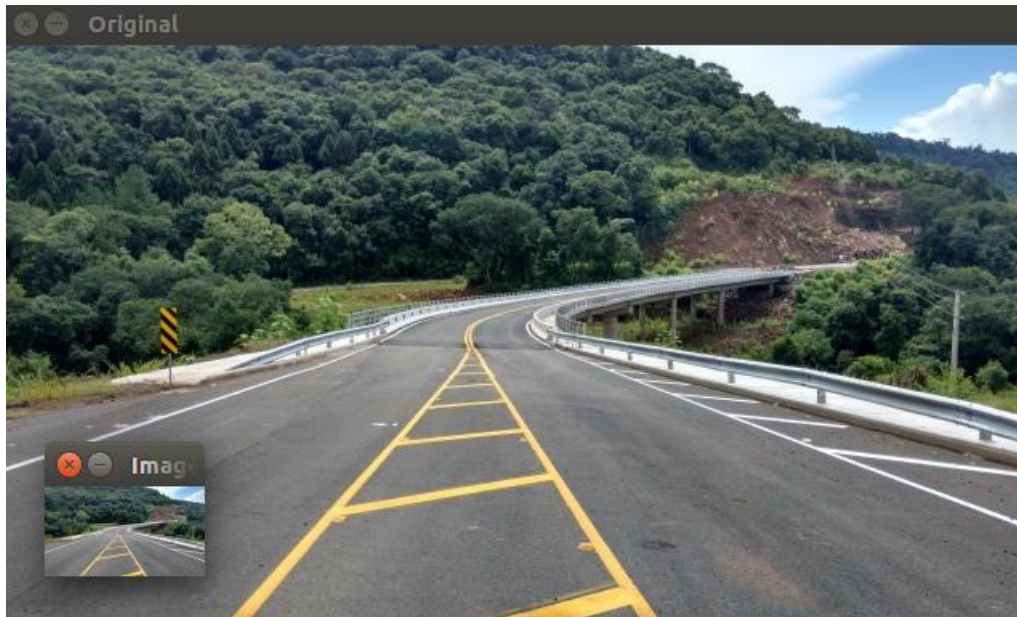


Figura 13 No canto inferior esquerdo da imagem é possível notar a imagem redimensionada.

Veja que a função ‘resize’ 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("Imagem 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 original. 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)
```

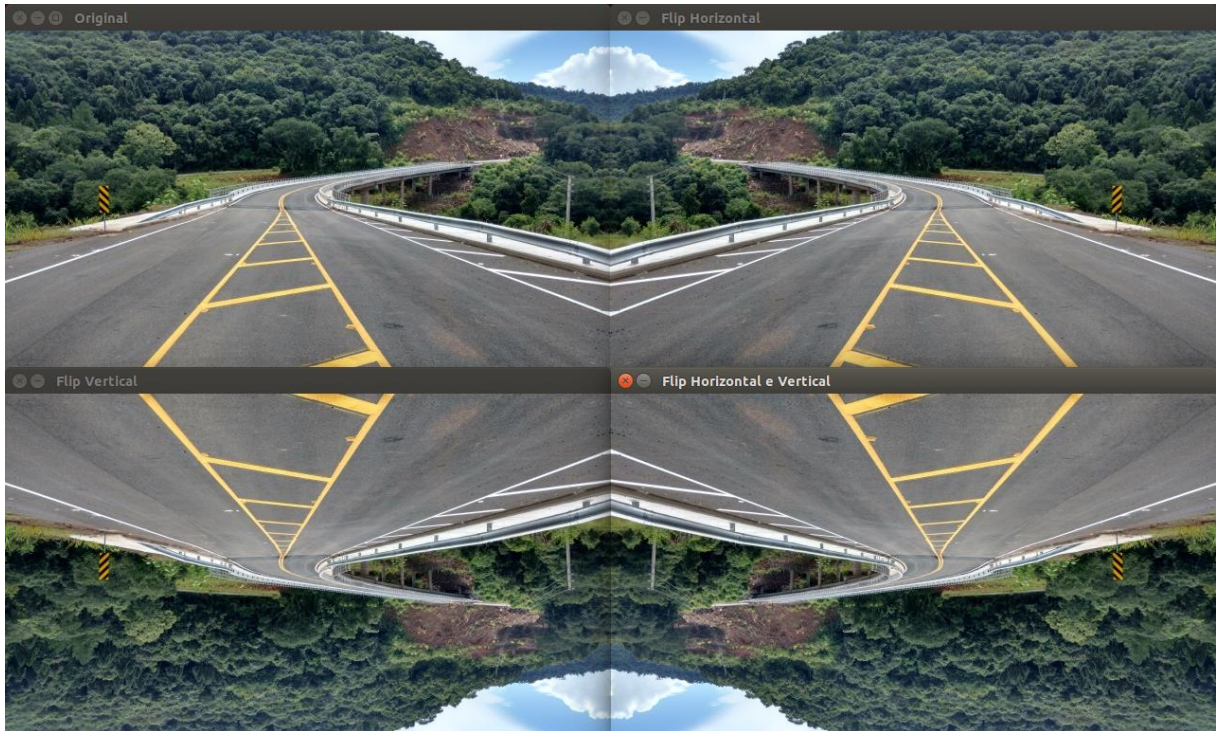


Figura 15 Resultado do flip horizontal, vertical e horizontal e vertical na mesma imagem.

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("Imagem 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 está 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)
```

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.

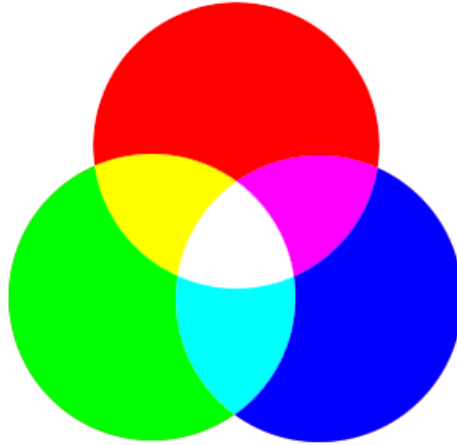


Figura 18 Sistema de cores RGB.

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 mesm aimagem.

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 exibí-los em tons de cinza conforme mostra a imagem abaixo:

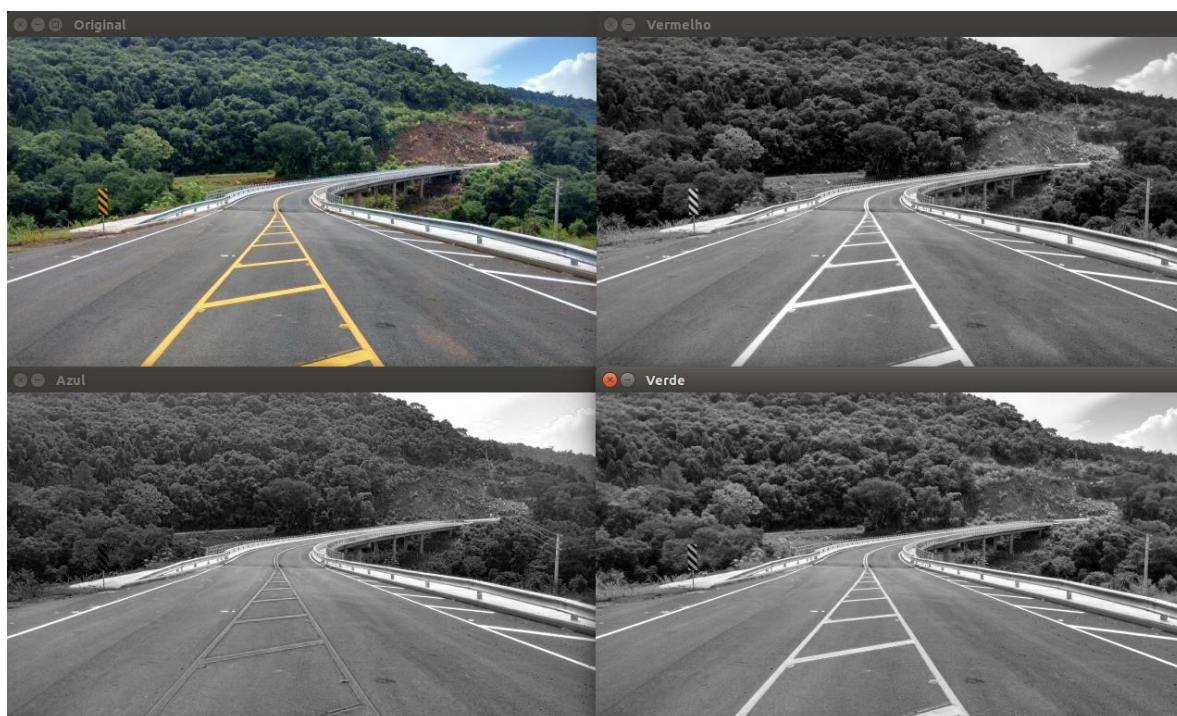


Figura 20 Perceba como a linha amarela (que é formada por verde e vermelho) fica quase imperceptível no canal azul.

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)
```

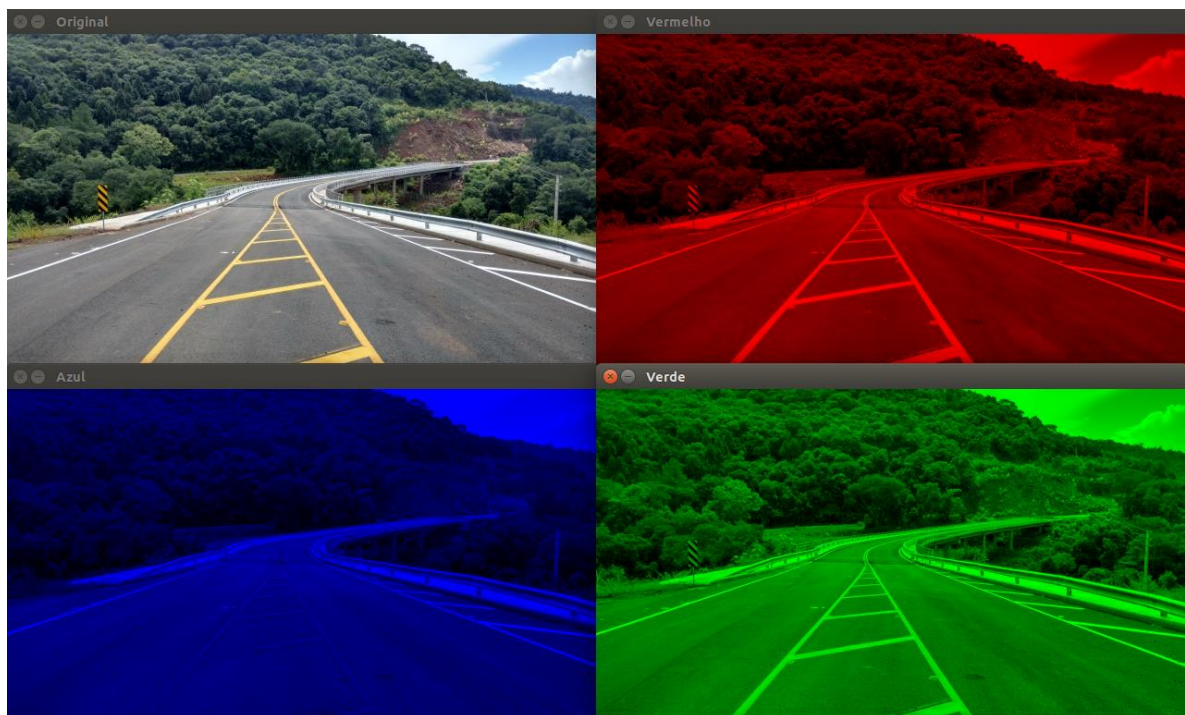


Figura 21 Exibindo os canais separadamente.

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 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 22 Imagem original já convertida para tons de cinza.

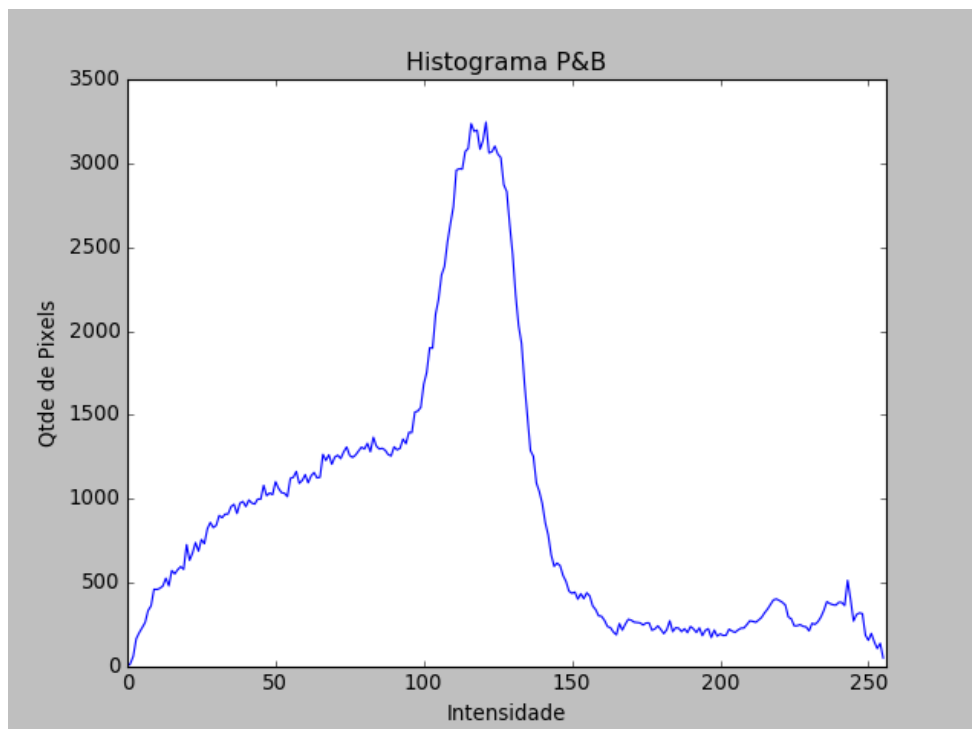


Figura 23 Histograma da imagem em tons de cinza.

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("Imagem 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()
```

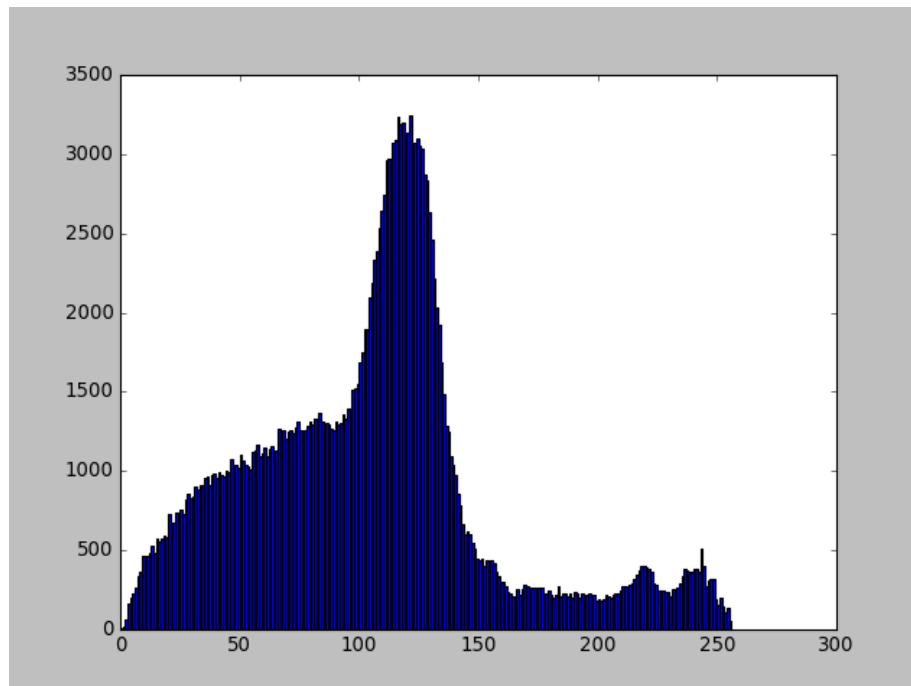


Figura 24 Histograma em barras.

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 pela 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("Imagem 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()
```

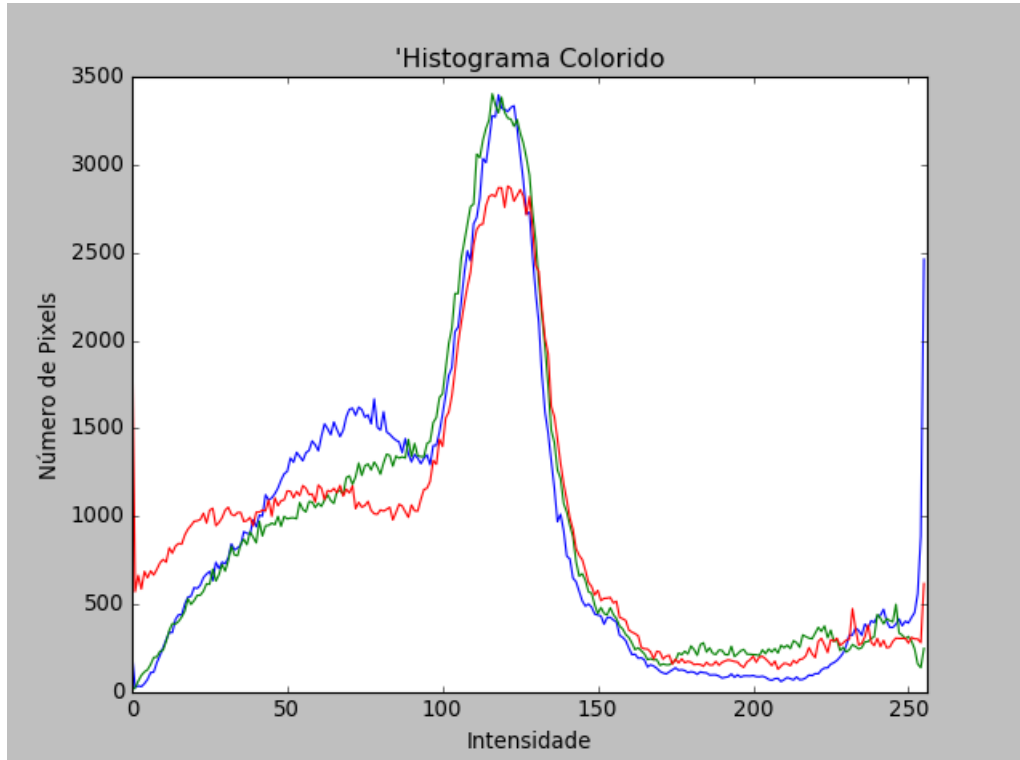


Figura 25 Histograma colorido da imagem. Neste caso são plotados os 3 canais RGB.

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:

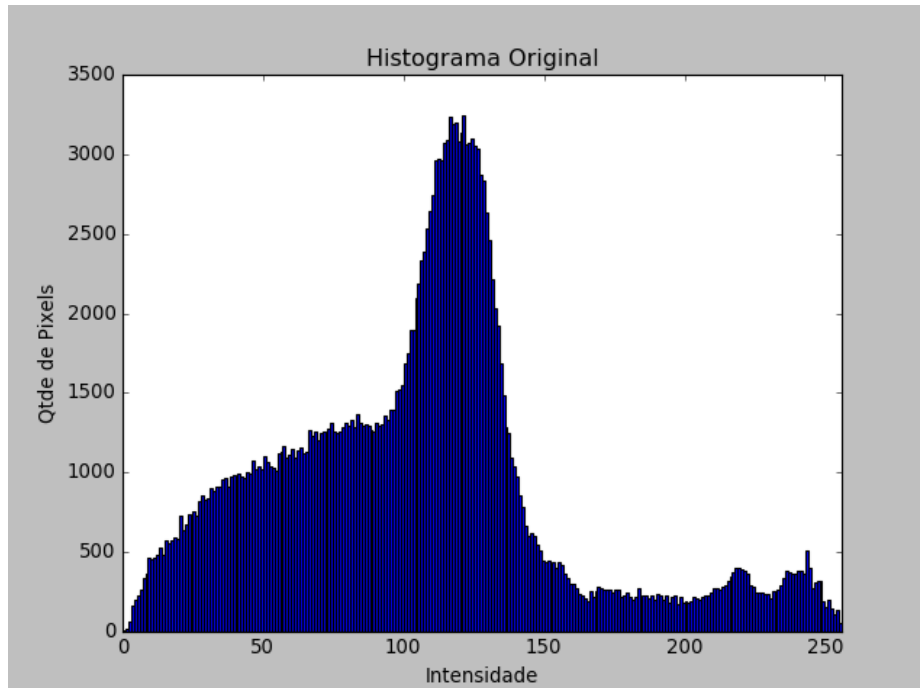


Figura 26 Histograma da imagem original.

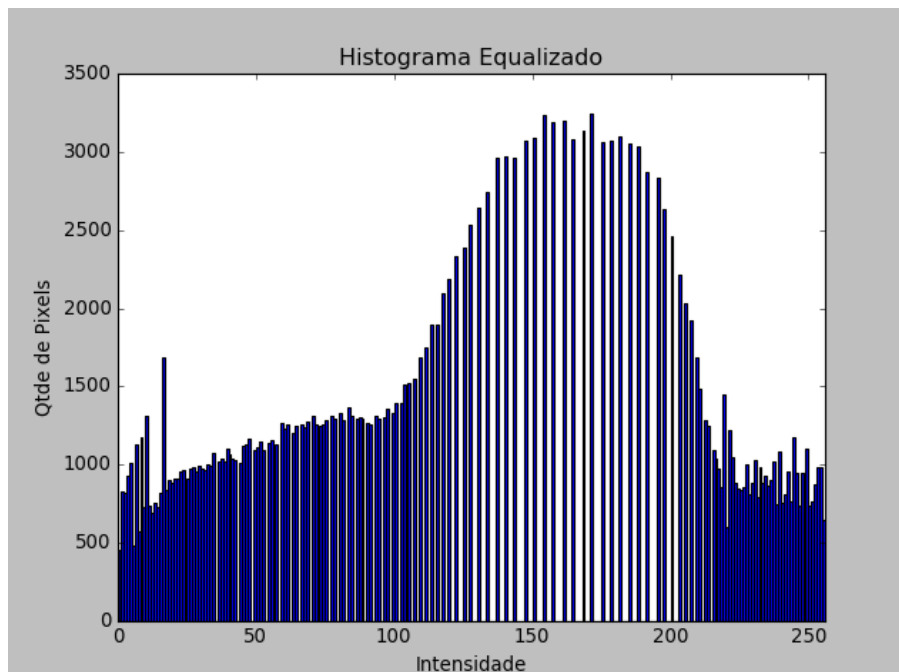


Figura 27 Histograma da imagem cujo histograma foi equalizado.

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 sem 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 OpenCV¹ que mostra o seguinte exemplo:

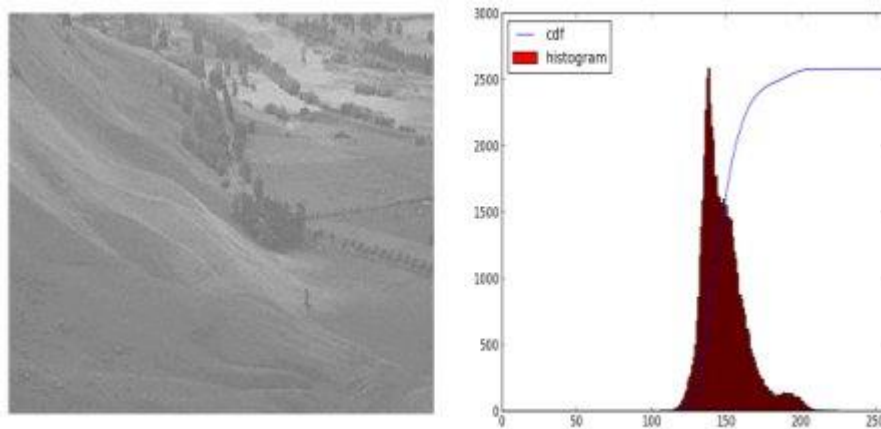


Figura 29 Exemplo extraído da documentação da OpenCV mostrando o histograma de uma imagem com baixo contraste.

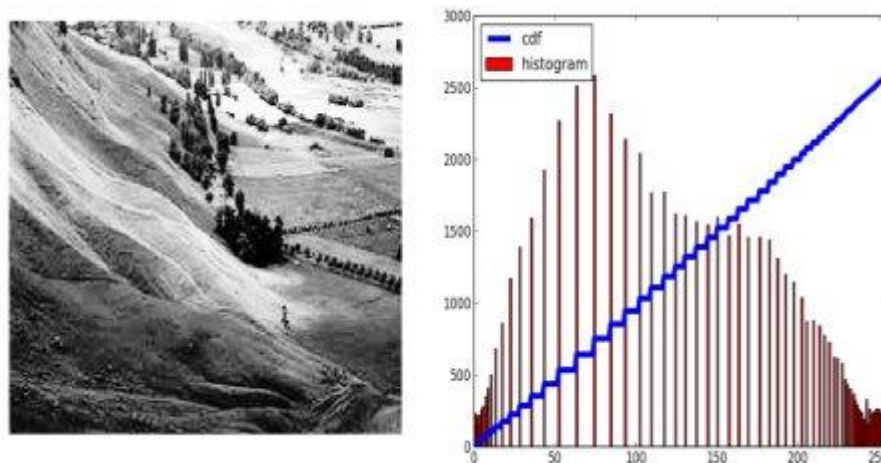


Figura 30 Exemplo extraído da documentação da OpenCV mostrando o histograma da mesma imagem mas desta vez com o histograma equalizado pela função ‘equalizeHist’.

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:

¹ Disponível em: <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 embaçado.

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 ímpares 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 cálculo 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 suavizadas (Blur)", suave)
cv2.waitKey(0)
```

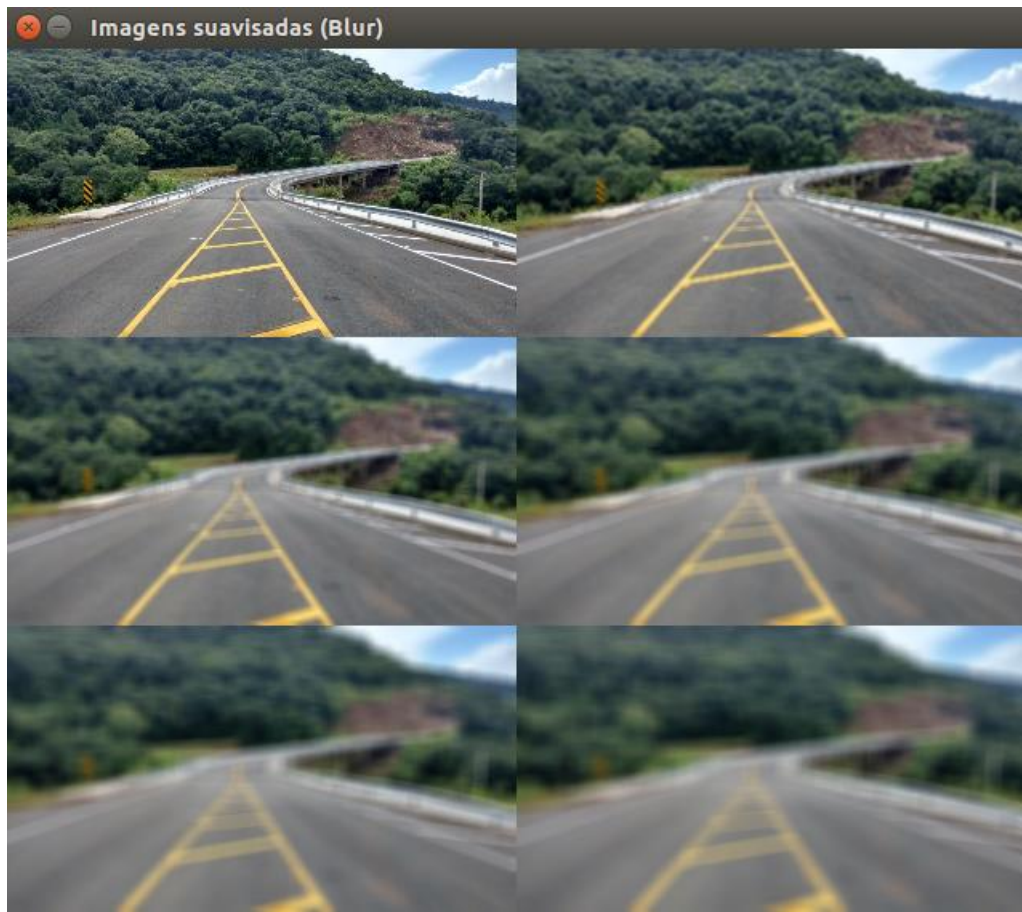


Figura 32 Imagem original seguida da esquerda para a direita e de cima para baixo com imagens tendo caixas de cálculo de 3x3, 5x5, 7x7, 9x9 e 11x11. Perceba que conforme aumenta a caixa maior é o efeito de borrão (blur) na imagem.

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("Imagem original e suavizadas pelo filtro
           Gaussiano", suave)
```

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

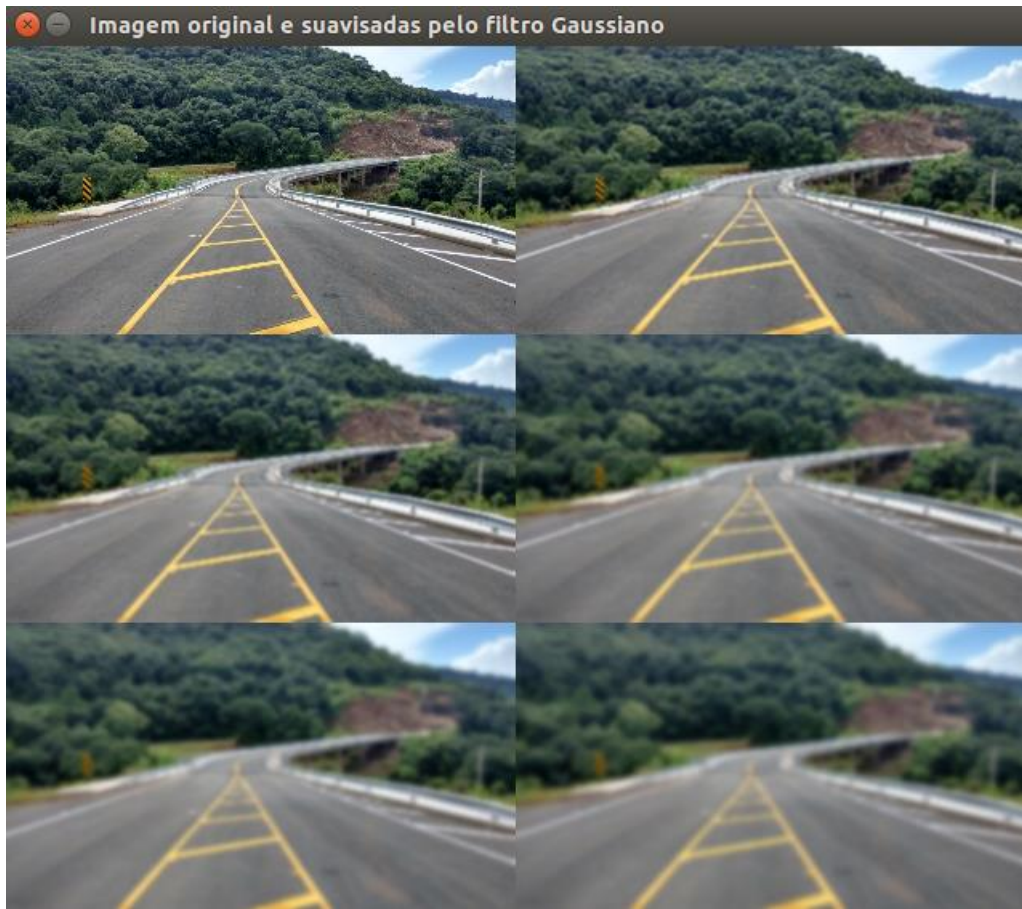


Figura 33 Imagem original seguida da esquerda para a direita e de cima para baixo com imagens tendo caixas de cálculo de 3x3, 5x5, 7x7, 9x9 e 11x11 utilizando o *Gaussian Blur*.

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 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("Imagem original e suavizadas pela mediana", suave)
cv2.waitKey(0)

```

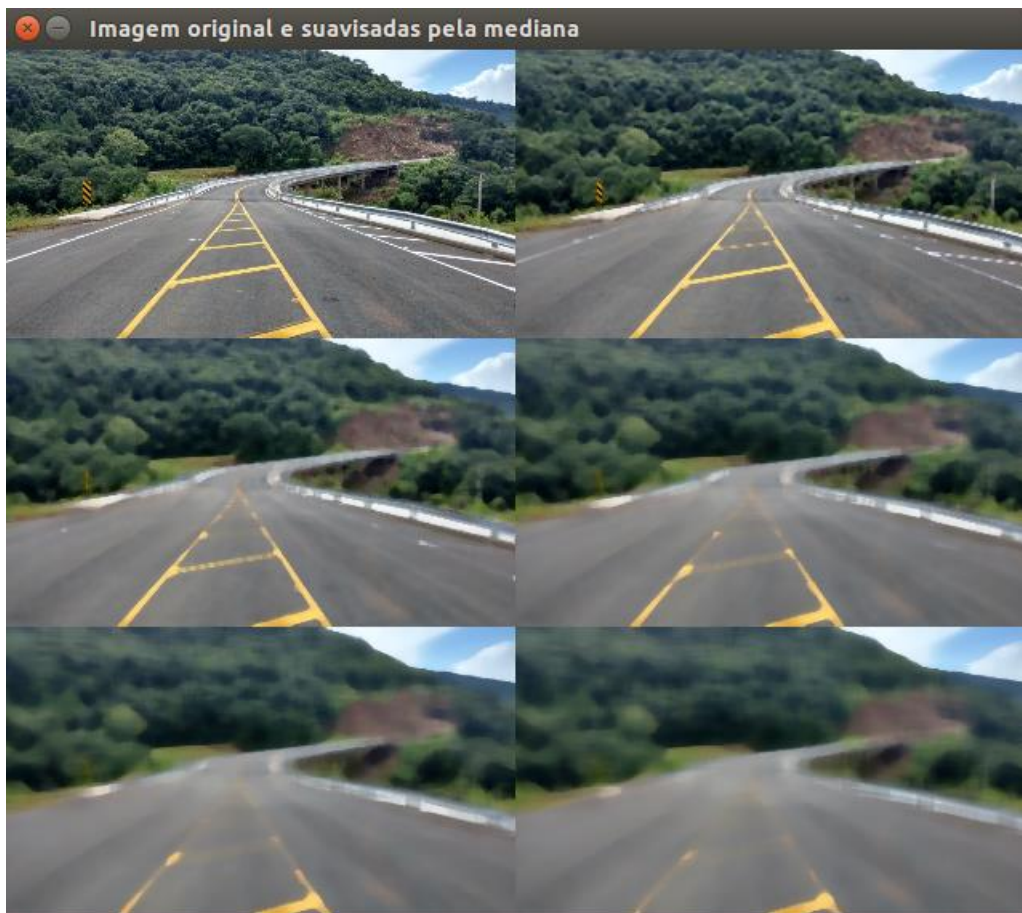


Figura 34 Da mesma forma temos a imagem original seguida pelas imagens alteradas pelo filtro de mediana com o tamanho de 3, 5, 7, 9, e 11 nas caixas de cálculo.

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 imagem. A função usada é `cv2.bilateralFilter()` e o código usado segue abaixo:

```
img = cv2.imread('ponte.jpg')
img = img[:, :, 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)])
])
```

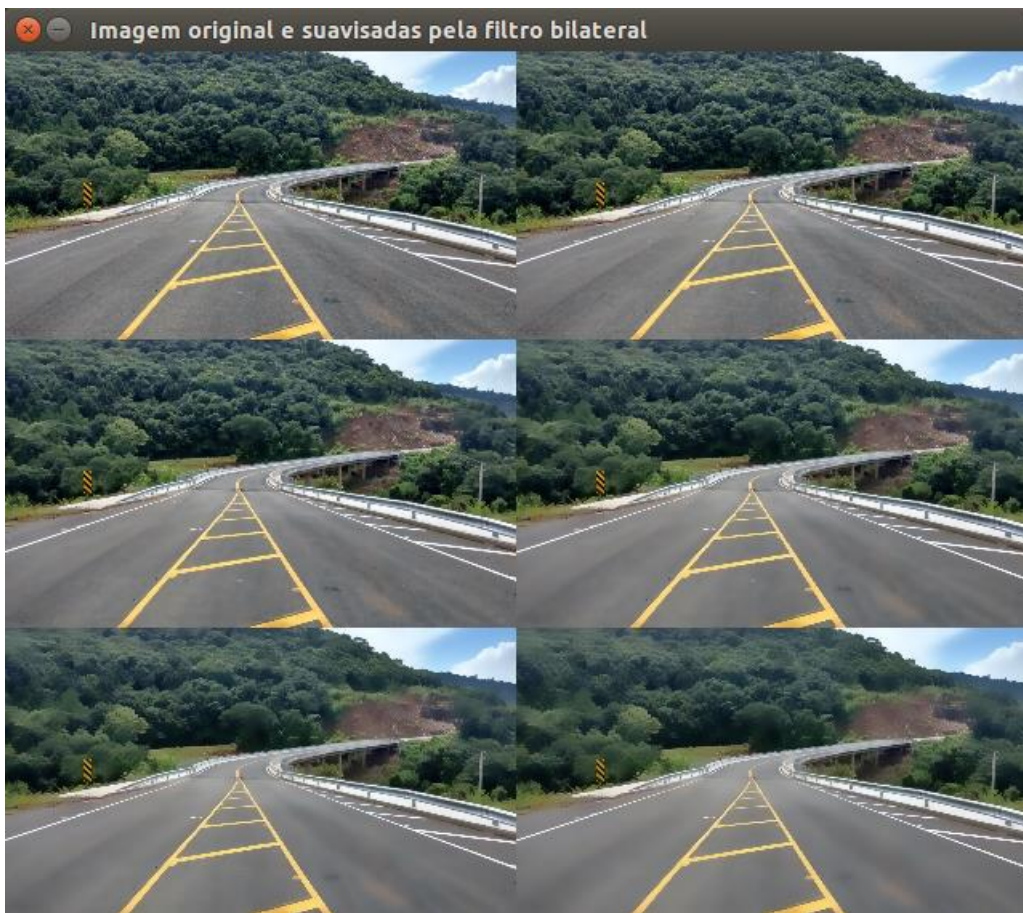


Figura 35 Imagem original e imagens alteradas pelo filtro bilateral. Veja como mesmo com a grande interferência na imagem no caso da imagem mais à baixo e à direita as bordas são preservadas.

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.

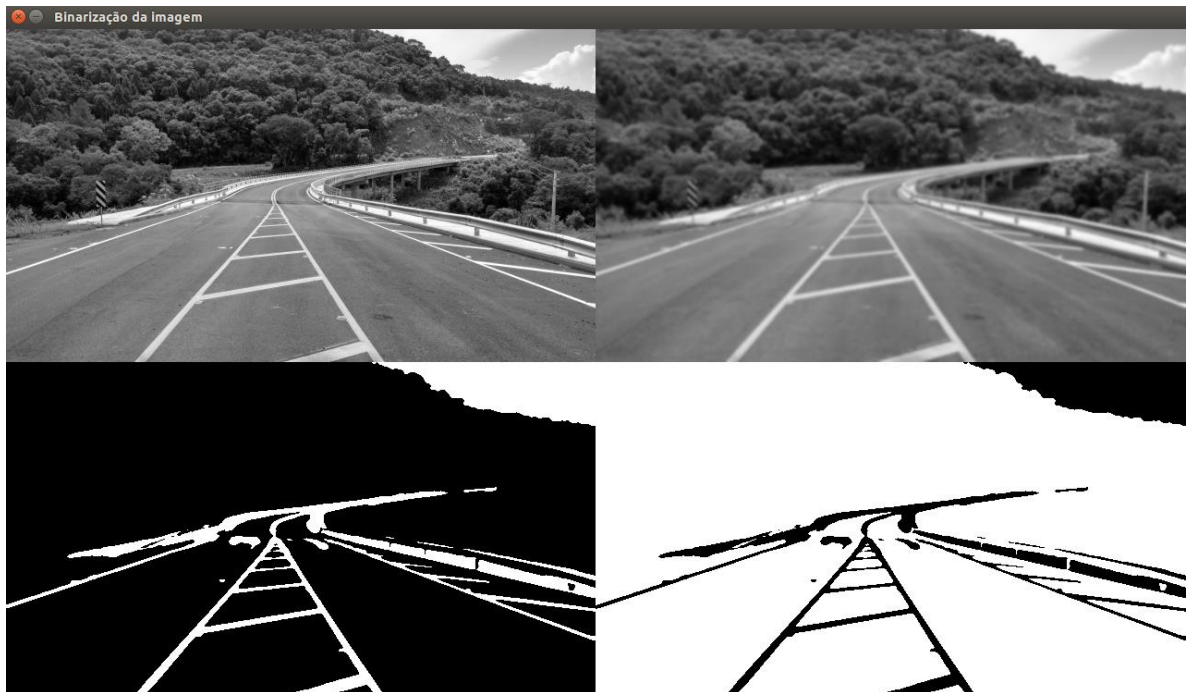


Figura 36 Da esquerda para a direita e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada e a imagem binarizada invertida.

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 imagem. Outro parâmetro é um inteiro que é subtraído da média calculada dentro da caixa para gerar o threshold final.

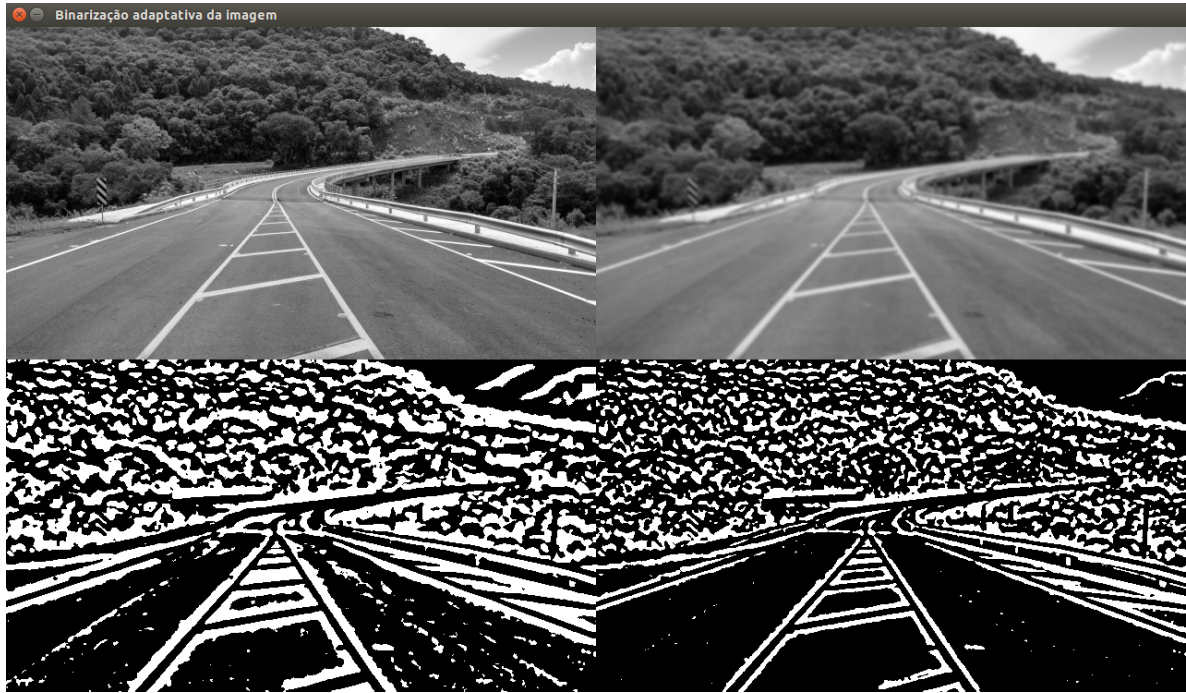


Figura 37 Threshold adaptativo. Da esquerda para a direita e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada pela média e a imagem binarizada com Gauss.

```
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)
```

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 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 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:

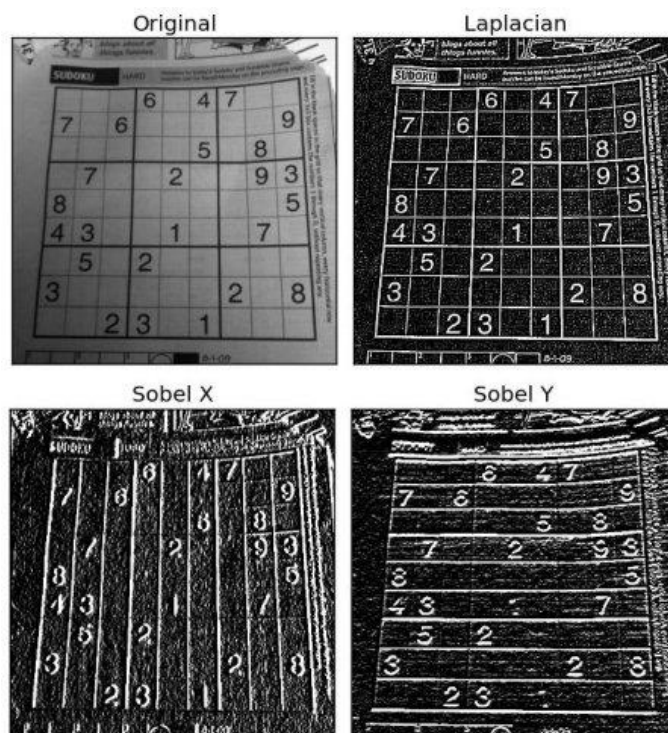


Figura 40 Exemplo de processamento Sobel Vertical e Horizontal disponível na documentação da OpenCV.

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.

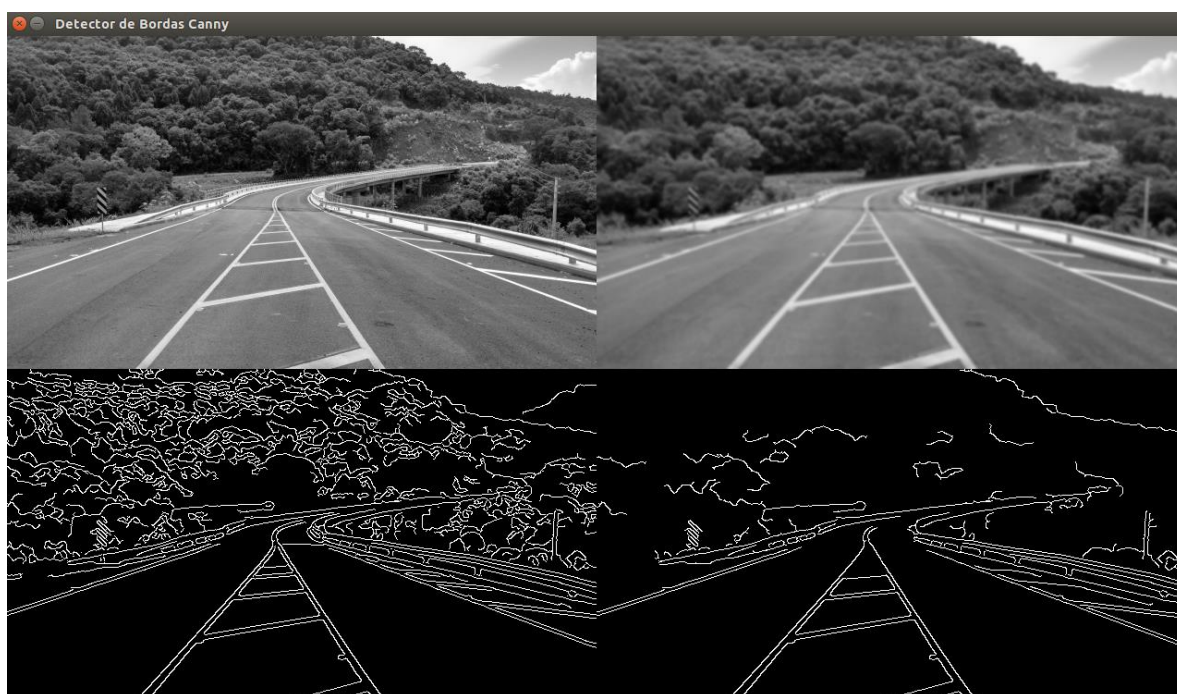


Figura 42 Canny com parâmetros diferentes. A esquerda deixamos um limiar mais baixo (20,120) e à direita a imagem foi gerada com limiares maiores (70,200).

```
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 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 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.

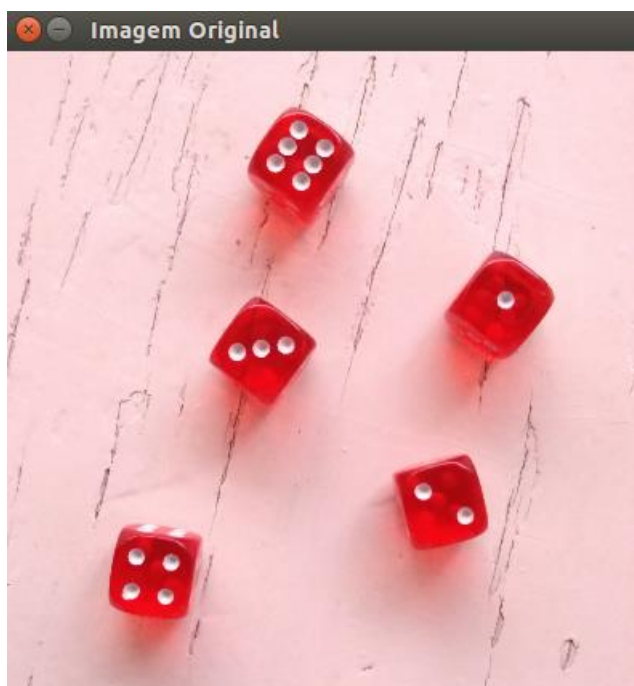


Figura 43 Imagem original. A superfície branca com ranhuras dificultará o processo.

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.

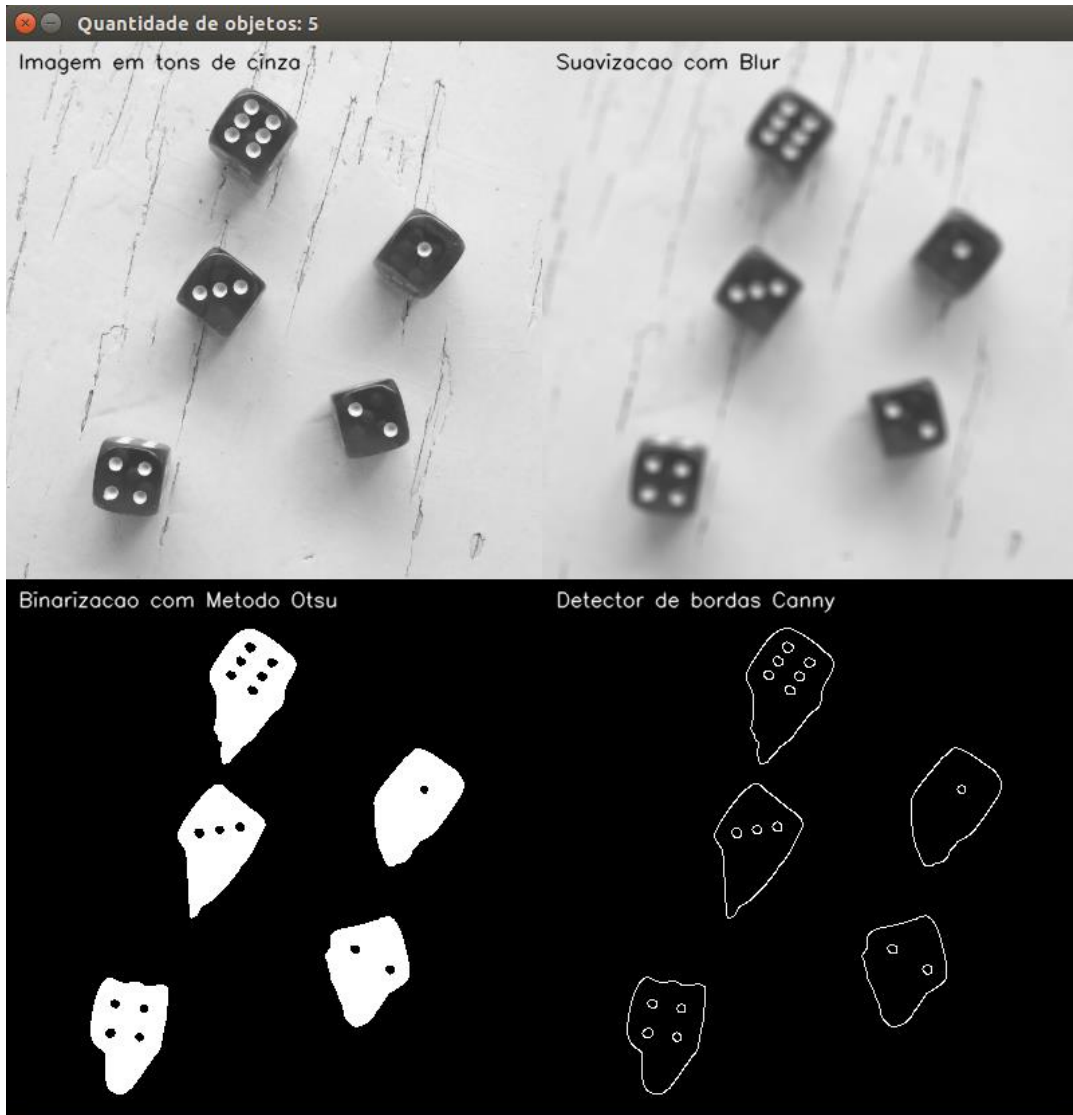


Figura 44 Passos para identificar e contar os dados na imagem.



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, "Imagem 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("Imagem 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 existem 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 Em breve...

11.1 Detecção de faces em imagens

11.2 Detecção de faces em vídeos

11.3 Rastreamento de objetos em vídeos

11.4 Reconhecimento de caracteres manuscritos

11.5 Treinamento para identificação de objetos por Haar Cascades

11.6 Criação de um identificador de objetos com Haar Cascades