

<i>Prefácio</i>	2
<i>Visão Computacional</i>	2
<i>Sistema de coordenadas e manipulação de pixels</i>	5
<i>Recorte e desenho sobre a imagem</i>	10
<i>Transformações e máscaras</i>	14
<i>Cortando uma imagem / ‘Crop’</i>	14
<i>Redimensionamento / Resize</i>	14
<i>Espelhando uma imagem / Flip</i>	17
<i>Rodando uma imagem / Rotate</i>	18
<i>Máscaras</i>	19
<i>Sistemas de cores</i>	20
<i>Canais da imagem colorida</i>	22
<i>Histogramas e equalização de imagem</i>	24
<i>Equalização de Histograma</i>	28
<i>Suavização de imagens</i>	32
<i>Suavização por cálculo da média</i>	32
<i>Suavização pela Gaussiana</i>	33
<i>Suavização pela mediana</i>	34
<i>Suavização com filtro bilateral</i>	36
<i>Binarização com limiar</i>	37
<i>Threshold adaptive</i>	38
<i>Segmentação e métodos de detecção de bordas</i>	39
<i>Sobel</i>	39
<i>Filtro Laplaciano</i>	41
<i>Detector de bordas Canny</i>	42
<i>Identificando e contando objetos</i>	43
<i>Detecção de faces em imagens</i>	47
<i>Detecção de faces em vídeos</i>	51
<i>Rastreamento de objetos em vídeos</i>	52

## Prefácio

Os sentidos são os meios através dos quais os seres vivos percebem e reconhecem outros objetos e as características do meio ambiente em que se encontram, noutras palavras, são as traduções do mundo físico para a mente.

São cinco, os mais conhecidos: a visão, a audição, o tato, o paladar e o olfato, mas é consenso na comunidade científica que os seres humanos possuem muitos outros, sendo a visão um dos mais importantes para o ser humana.

Por este motivo, implementar o sentido da visão numa máquina, gera um resultado impressionante. Imagens estão em todos os lugares e a capacidade de reconhecer objetos, paisagens, rostos, sinais e gestos torna as máquinas muito mais úteis. A finalidade destes apontamentos é de servir de tópico para o estudo de visão computacional.

## Visão Computacional

**Visão computacional** é a ciência e tecnologia das máquinas que enxergam. Ela desenvolve teoria e tecnologia para a construção de sistemas artificiais que obtém informação de imagens ou quaisquer dados multidimensional.

São exemplo de aplicações, as que incluem o controle de processos, como robôs industriais ou veículos autônomos, detecção de eventos, organização de informação, modelagem de objetos ou ambientes e interação (associados a interação homem-computador).

A visão computacional também pode ser descrita como um complemento da visão biológica. Na visão biológica, a percepção visual dos humanos e outros animais é estudada, resultando modelos de sistemas que operam em termos de processos fisiológicos.

Por outro lado, a visão computacional estuda e descreve sistemas de visão artificial implementados por hardware ou software.

A definição é da Wikipédia e deve já ter sido alterada, afinal o campo de estudos sobre visão computacional está em constante evolução. Neste ponto é importante frisar 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 muitas aplicações e outros sistemas que usamos. O Facebook já reconhece objetos automaticamente para classificar suas fotos, além disso, já aponta onde estão as pessoas na imagem para você “marcar”. O mesmo ocorre com smartphones que já disparam a foto quando as pessoas estiverem sorrindo, pois conseguem reconhecer tais expressões.

Além disso, outros sistemas de reconhecimento como o chamado popularmente “OCR” das placas dos veículos já estão espalhados por todo o lado, onde as imagens dos veículos são capturadas, a placa reconhecida e convertida para textos e números

que podem ser comparados diretamente com bases de dados de veículos furtados ou com taxas de impostos em atraso. Enfim, a visão computacional já está aí!

A ideia é focar-nos no histograma da imagem relacionada com processamento de imagem, uma vez que o reconhecimento dum objeto existente numa fotografia está relacionado com a visão computacional.

Vamos utilizar a linguagem Python neste apontamento, especificamente a versão 2.7 juntamente com a biblioteca OpenCV versão 3.2 que poderá baixar e configurar através de vários tutoriais na internet. No final dos aportamentos teremos um apêndice com as instruções detalhadas. Algumas bibliotecas Python também são necessárias como Numpy (Numeric Python), Scipy (Scientific Python) e Matplotlib que é uma biblioteca para ‘plotagem’ de gráficos com sintaxe similar ao Matlab. Utilizamos nas imagens dos aportamentos o sistema operacional ‘macOS High Sierra Versão 10.13.3’.

Vamos executar o nosso primeiro programa. Um “Alô mundo!” onde iremos abrir um arquivo de imagem do disco e exibi-lo no ecrã e salvaguardar no disco com outro nome.

```
# Importação das bibliotecas
import cv2

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

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

cv2.imshow("Imagen original", imagem) # atribui nome à janela

# salvaguardar a imagem no disco com função imwrite()
cv2.imwrite('ronaldo_out.jpg', imagem)

# espera por pressionar qualquer tecla
cv2.waitKey(0)
cv2.destroyAllWindows()
# força o destroyAllWindows (osx).
cv2.waitKey(1)
```

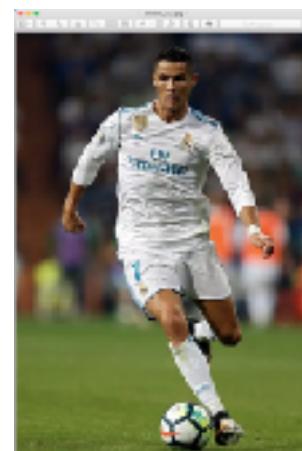
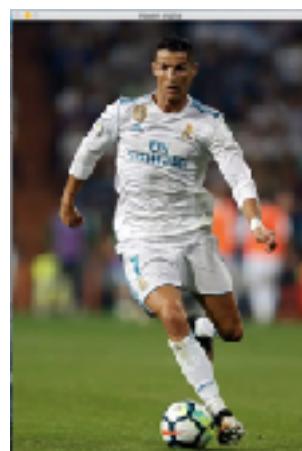


Figura 0. Leitura e salvaguarda da imagem ‘ronaldo.jpg’ (prog. 00\_hello\_openVC\_World.py)

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 “ronaldo\_out.jpg”. Vamos explicar o código em detalhes abaixo:

```
# Importação das bibliotecas
import cv2

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

A importação da biblioteca padrão da OpenCV é obrigatória para utilizar as suas funções. A primeira função usada é para abrir a imagem através de cv2.imread() que tem 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 branco temos apenas um canal, ou seja, apenas uma matriz de 2 dimensões.

Para facilitar o entendimento podemos pensar num ecrã, 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 branco representada numa matriz de inteiros onde cada célula é um inteiro sem sinal de 8 bits que pode conter de 0 (preto) até 255 (branco). Entenda os vários tons de cinza nos valores intermediários como 30 (cinza escuro) e 210 (cinza claro).

No caso de imagens preto e branco é composto de apenas uma matriz de duas dimensões, como na imagem acima. 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:

*Nota(\*)*: -Em matemática, uma tupla é uma sequência finita (também chamada de “lista ordenada”) de objetos.

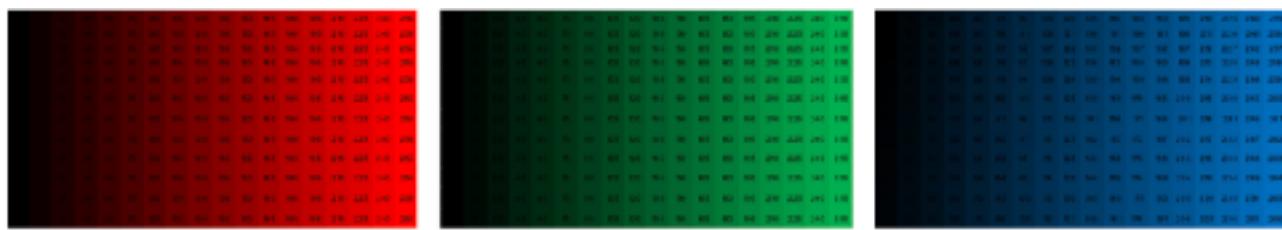


Figura 2. A imagem é composta por 3 componentes de 8 bits cada, sem sinal, o que gera 256 combinações por cor. Portanto, a representação é de 256 vezes 256 vezes 256 ou  $256^3$  que é igual a 16,7 milhões de cores.

## Sistema de coordenadas e manipulação de pixels

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

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

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

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

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

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

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

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

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

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

for y in range(0, imagem.shape[0]):
    for x in range(0, imagem.shape[1]):
        imagem[y, x] = (255,0,0)
cv2.imshow("Imagen modificada", imagem)

# espera por pressionar qualquer tecla
cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

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). (Prog. ex\_openVC\_2.py)

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

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

for y in range(0, imagem.shape[0]):      #percorre linhas
    for x in range(0, imagem.shape[1]):  #percorre colunas
        imagem[y, x] = (x%256,y%256,x%256)

cv2.imshow('Imagen modificada', imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

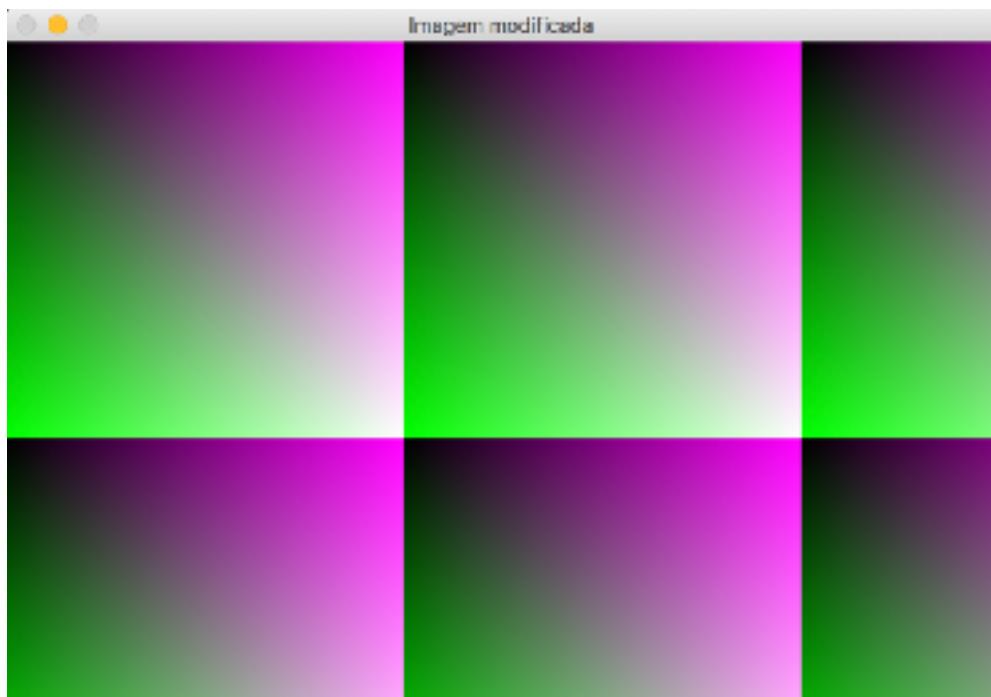


Figura 5. A alteração nas componentes das cores da imagem conforme as coordenadas de linha e coluna geram a imagem acima. (Prog. ex\_openVC\_3.py)

Alterando minimamente o código, especificamente no componente ‘green’, teremos 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 a zeros. A dinâmica da mudança de linhas e colunas gera esta imagem.

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

for y in range(0, imagem.shape[0], 1):      #percorre as linhas
    for x in range(0, imagem.shape[1], 1): #percorre as colunas
        imagem[y, x] = (0,(x*y)%256,0)

cv2.imshow("Imagen modificada", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

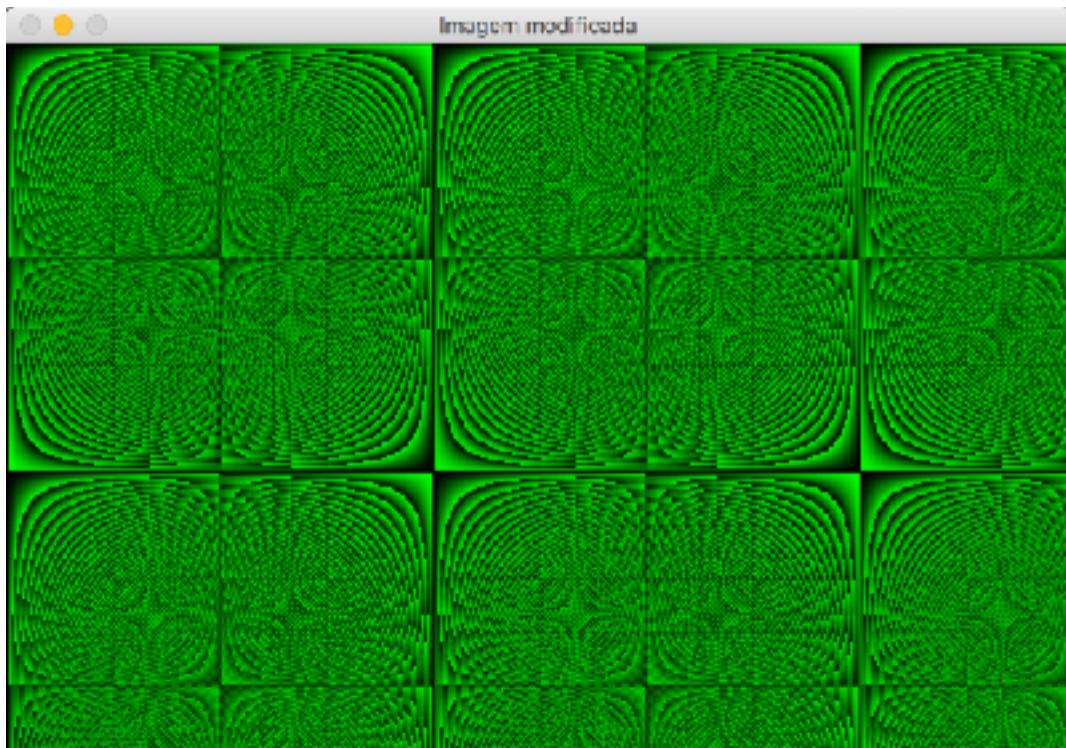


Figura 6. A alteração dinâmica da cor de cada pixel gera esta bela imagem. A imagem original se perdeu pois todos os pixels foram alterados. (Prog. ex\_openVC\_4.py)

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

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

for y in range(0, imagem.shape[0], 10):      #percorre linhas
    for x in range(0, imagem.shape[1], 10): #percorre colunas
        imagem[y:y+5, x: x+5] = (0,255,255)

cv2.imshow("Imagen modificada", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

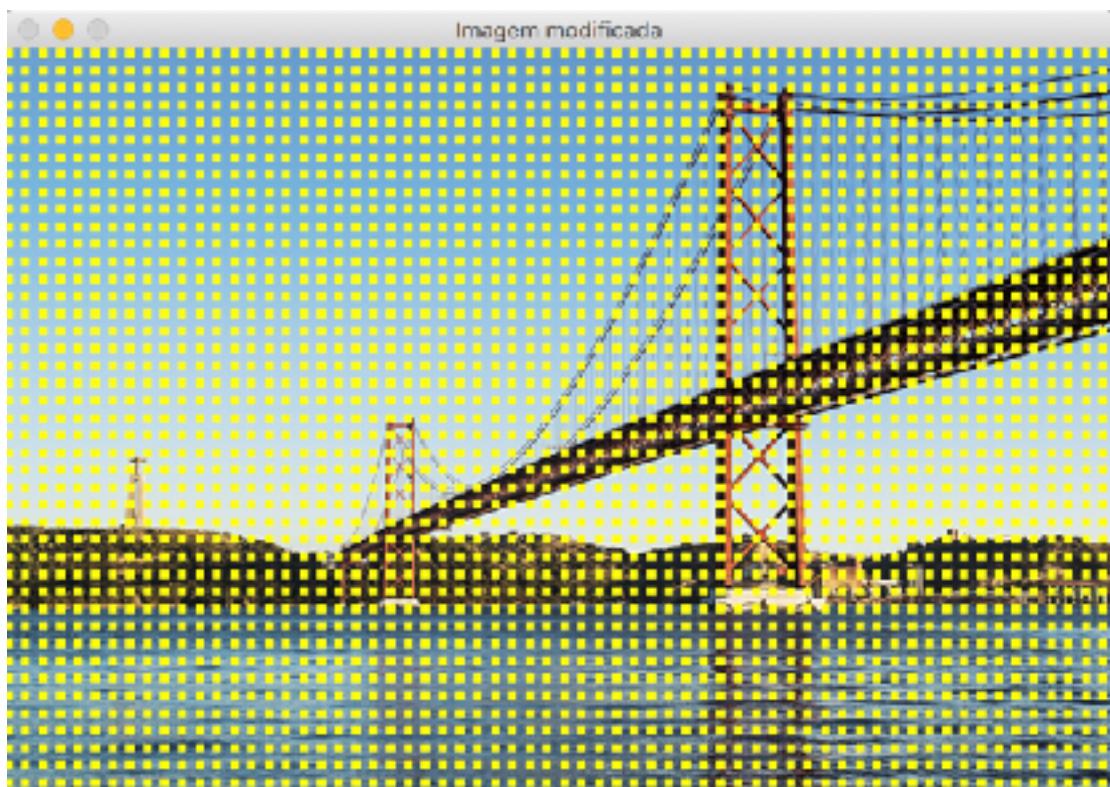


Figura 7. O código gerou quadrados amarelos de 5x5 pixels sobre a toda a imagem.  
(Prog. ex\_openVC\_4.py)

## Recorte 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:

```
imagem[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.

```
#Cria um retangulo azul por toda a largura da imagem
import cv2
imagem = cv2.imread('ponte.jpg')
cv2.imshow('ponte', imagem)

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

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

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

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

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

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

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

cv2.imshow('Imagen alterada', imagem)
cv2.imwrite('alterada.jpg', imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

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

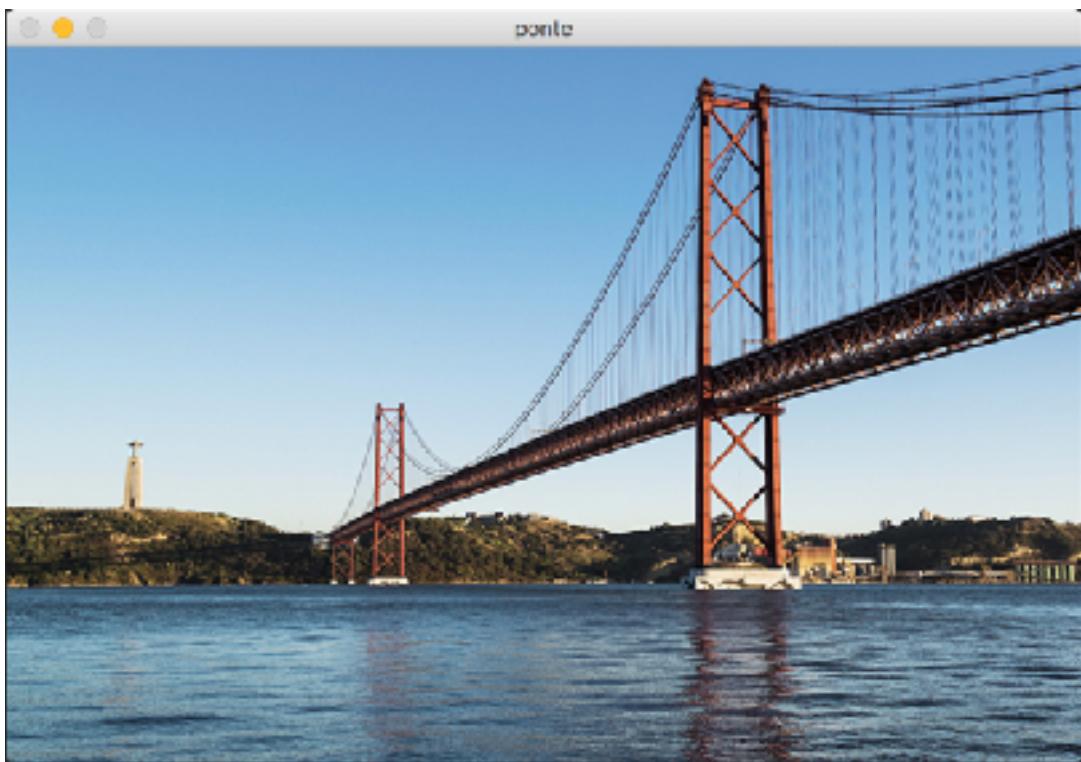


Figura 8. Imagem original ‘ponte.jpg’

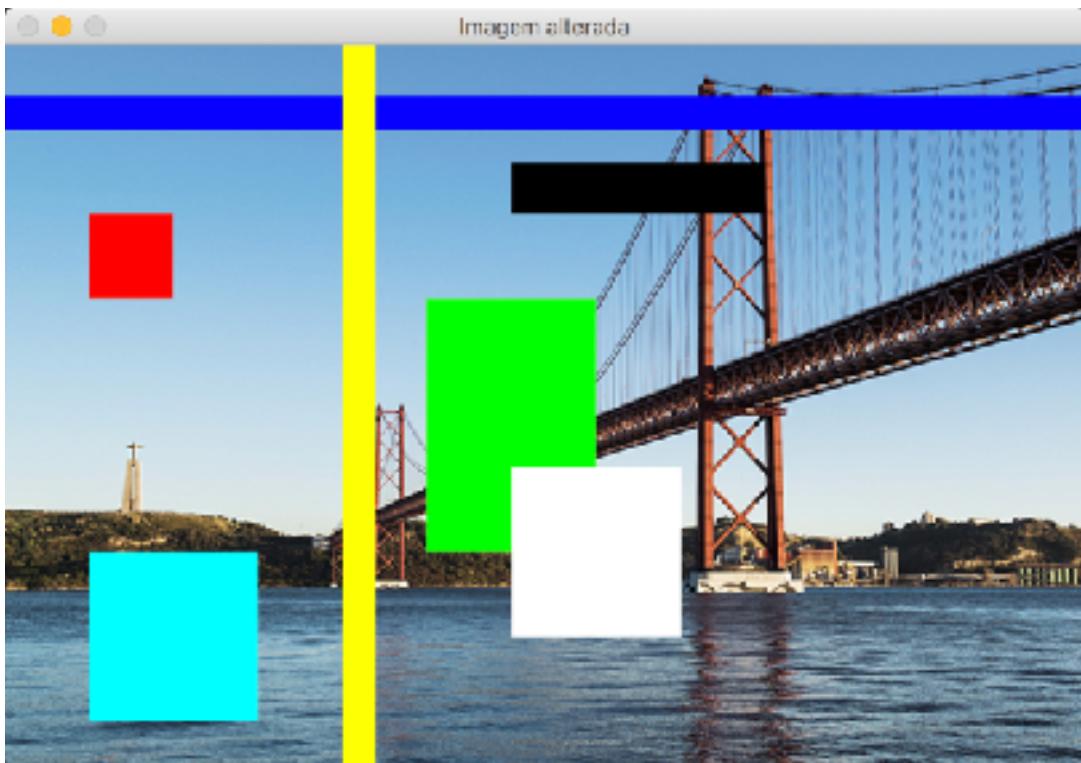


Figura 9 Imagem após a execução do código acima com os retângulos coloridos incluídos pela técnica de ‘recorte’ (slicing). (Prog. ex\_openVC\_6.py)

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, observe:

Rev. 09

```
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('Desenho sobre a imagem', imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

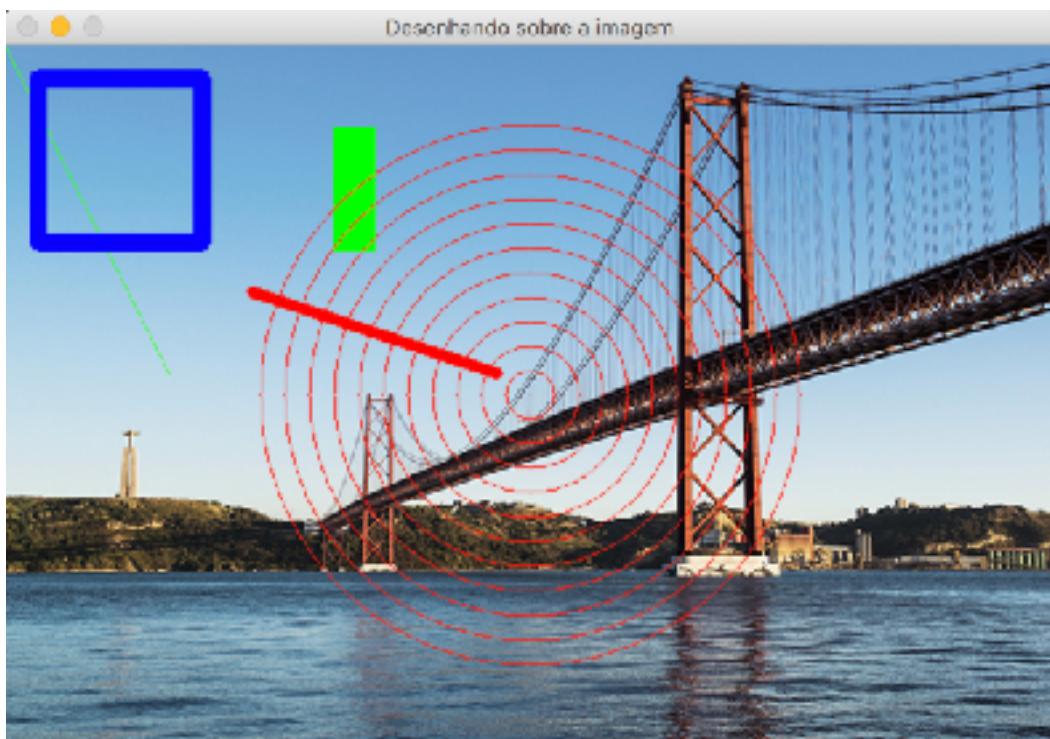


Figura 10. Utilizando funções do OpenCV para desenhar sobre a imagem.  
(Prog. ex\_openVC\_7.py)

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

```
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)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

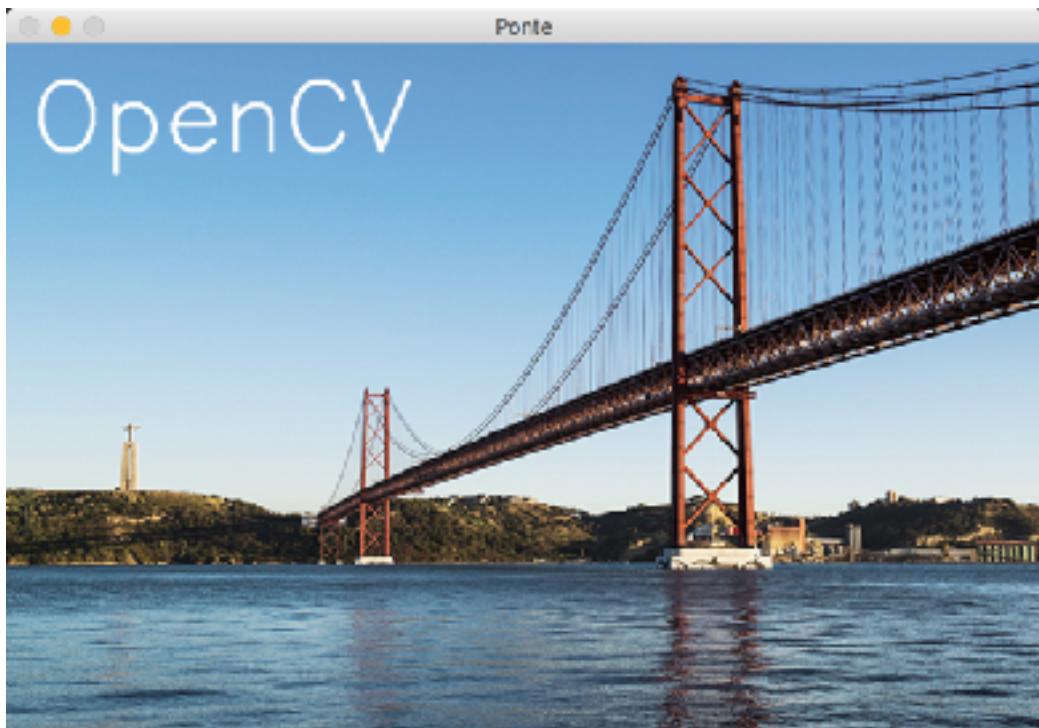


Figura 11. exemplo de escrita sobre a imagem. é possível escolher fonte, tamanho e posição. (Prog. ex\_openVC\_8.py)

## Transformações e máscaras

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

---

### Cortando uma imagem / ‘Crop’

A mesma técnica já usada para fazer o corte (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[100:300, 100:300]
cv2.imshow("Recorte da imagem", recorte)
cv2.imwrite('recorte.jpg', recorte)      #salvaguarda no disco

cv2.waitKey(0)
cv2.destroyAllWindows()
# no Mac força o destroyAllWindows.
cv2.waitKey(1)
```

Usando a mesma imagem ponte.jpg dos exemplos anteriores, temos o resultado abaixo que é da linha 101 até a linha 300 na coluna 101 até a coluna 300:

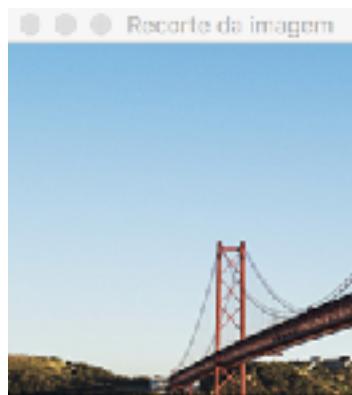


Figura 12 Imagem recortada da imagem original e salvada em um arquivo em disco. (Prog. ex\_openVC\_9.py)

---

### Redimensionamento / Resize

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

```
import cv2
import numpy as np
imagem = cv2.imread('ponte.jpg')
cv2.imshow('Original', imagem)

altura  = imagem.shape[0]
largura = imagem.shape[1]
porpocao = float(altura)/float(largura)

largura_nova = 220 #pixeis
altura_nova = int(largura_nova * porpocao)
tamanho_novo = (largura_nova, altura_nova)

imagem_redimensionada = cv2.resize(imagem, tamanho_novo , interpolation =
cv2.INTER_AREA)

cv2.imshow('Imagen redimensionada', imagem_redimensionada)

cv2.waitKey(0)
cv2.destroyAllWindows()

# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



Figura 13. No canto inferior esquerdo da imagem é possível notar a imagem redimensionada. ([Prog. ex\\_openVC\\_10.py](#))

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

```
import numpy as np
import imutils
import cv2

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

cv2.imshow("Imagen redimensionada", img_redimensionada)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

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 1/4 (um quarto) da original, tendo metade da altura e metade da largura da imagem original. Veja abaixo o resultado.

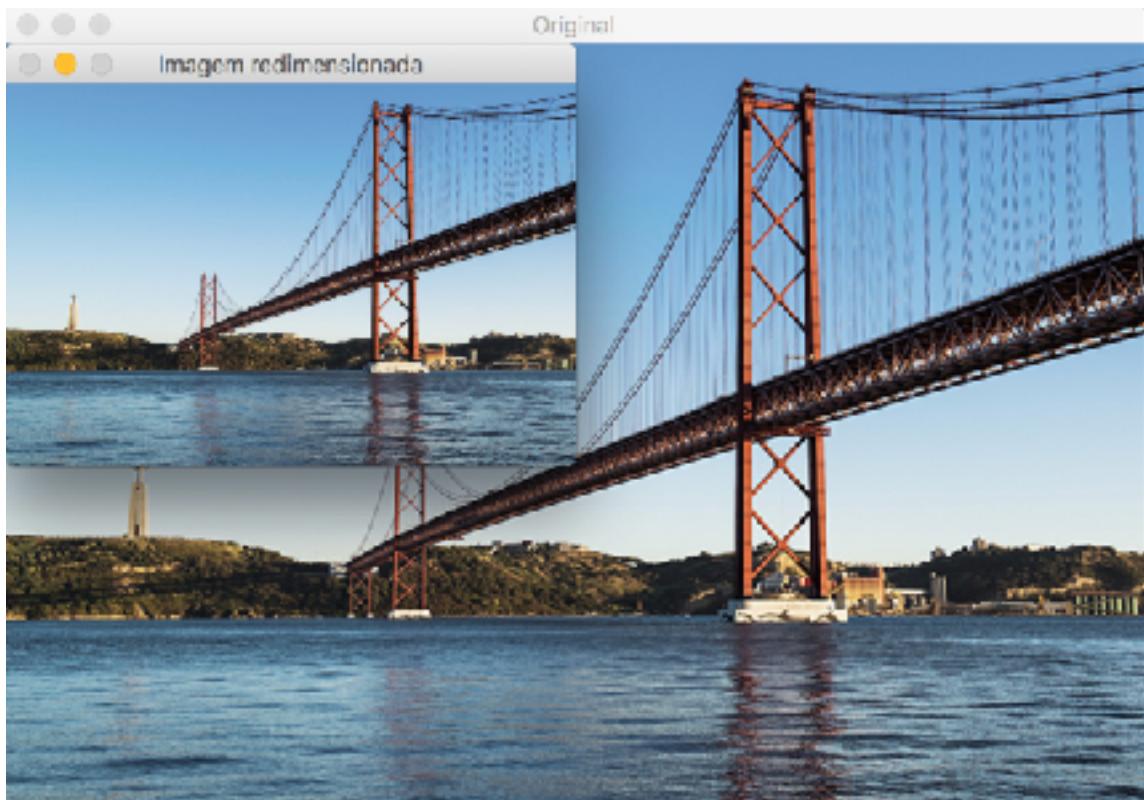


Figura 14. Imagem gerada a partir da técnica de slicing (Prog. ex\_openVC\_11.py)

---

## 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
imagem = cv2.imread('ponte.jpg')
cv2.imshow("Original", imagem)

#flip_horizontal = imagem[:,::-1]      #comando equivalente abaixo
flip_horizontal = cv2.flip(imagem, 1)
cv2.imshow("Flip horizontal", flip_horizontal)

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

#flip_hv = imagem[::-1,::-1]           #comando equivalente abaixo
flip_hv = cv2.flip(imagem, -1)
cv2.imshow("Flip Horizontal e Vertical", flip_hv)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

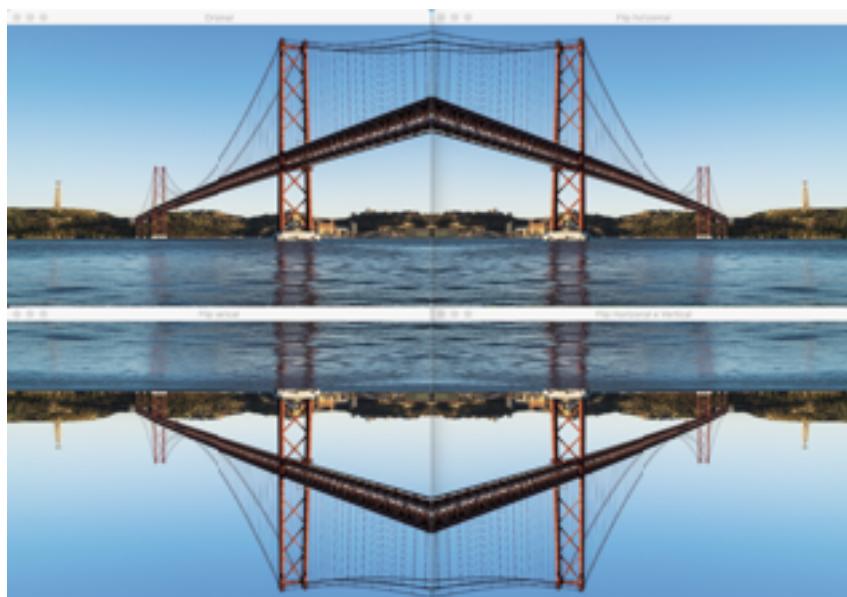


Figura 15. Resultado do flip horizontal, vertical e horizontal e vertical na mesma imagem  
(Prog. ex\_openVC\_12.py)

---

## Rodando uma imagem / Rotate

Em latin *affinis* significa “ligado com” ou que possui ligação. 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.

```
import cv2
imagem = cv2.imread('ponte.jpg')
cv2.imshow("Original", imagem)

(alt, lar) = imagem.shape[:2] #captura altura e largura
centro = (lar // 2, alt // 2) #acha o centro
M = cv2.getRotationMatrix2D(centro, 45, 1.0) #45 graus

img_rotacionada = cv2.warpAffine(imagem, M, (lar, alt))
cv2.imshow("Imagen rotacionada em 45 graus", img_rotacionada)

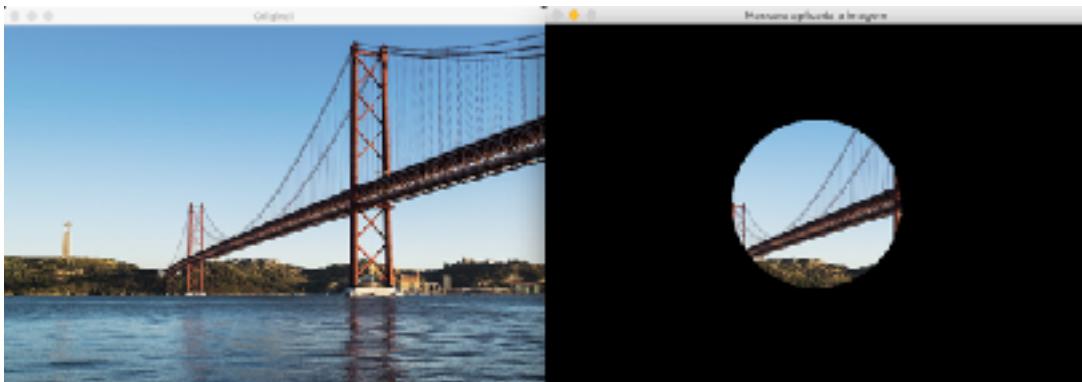
cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



Figura 16. Imagem rodada em 30 graus. (Prog. ex\_openVC\_13.py)

## 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 .

Imagen original à esquerda e à direita com a aplicação da máscara.  
(Prog.ex\_openVC\_14.py)

```
import cv2
import numpy as np

imagem = cv2.imread('ponte.jpg')
cv2.imshow("Original", imagem)
mascara = np.zeros(imagem.shape[:2], dtype = "uint8")

(cX, cY) = (imagem.shape[1] // 2, imagem.shape[0] // 2)
cv2.circle(mascara, (cX, cY), 100, 255, -1)
img_com_mascara = cv2.bitwise_and(imagem, imagem, mask = mascara)
cv2.imshow("Máscara aplicada a imagem", img_com_mascara)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



Figura 18. Aplicação da máscara com imagem original mascarada à direita  
(Prog. ex\_openVC\_15.py)

```

import cv2
import numpy as np
img = cv2.imread('ponte.jpg')
cv2.imshow('original', img)

mascara = np.zeros(img.shape[:2], dtype = 'uint8')
(cX, cY) = (img.shape[1] // 2, img.shape[0] // 2)
cv2.circle(mascara, (cX, cY), 180, 255, -1)
cv2.circle(mascara, (cX, cY), 70, 255, -1)
cv2.imshow("Mascara", mascara)
img_com_mascara = cv2.bitwise_and(img, img, mask = mascara)
cv2.imshow('Mascara aplicada a imagem', img_com_mascara)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)

```

## Sistemas de cores

Conhecemos o tradicional espaço de cores RGB (Red, Green, Blue) 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

Façamos uma síntese do modelo RGB (24 bit):

- RGB. Representa a cor natural como uma combinação de 3 canais de cor: **RED**, **GREEN** e **BLUE**.
- É um modelo aditivo. As cores são criadas por adição e mistura das cores primárias: **RED**, **GREEN** e **BLUE**
- Funciona muito à semelhança do olho humano.
- Os monitores e os scanners seguem o modelo **RGB**.
- Formatos RGB, também conhecidos por true-color, usam 8-bits por canal. A paleta de pixéis é, pois, de 24-bits, ou seja, 16.7 milhões de cores ( $2^{24}=16777216$  cores).
- Imagens JPEG —de 16, 24 e 32 bits— são imagens RGB.

Contudo, existem outros sistemas de cores como o ‘Preto e Branco’ - **BW** ou ‘tons de cinzento’, 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 sistemas de cores.

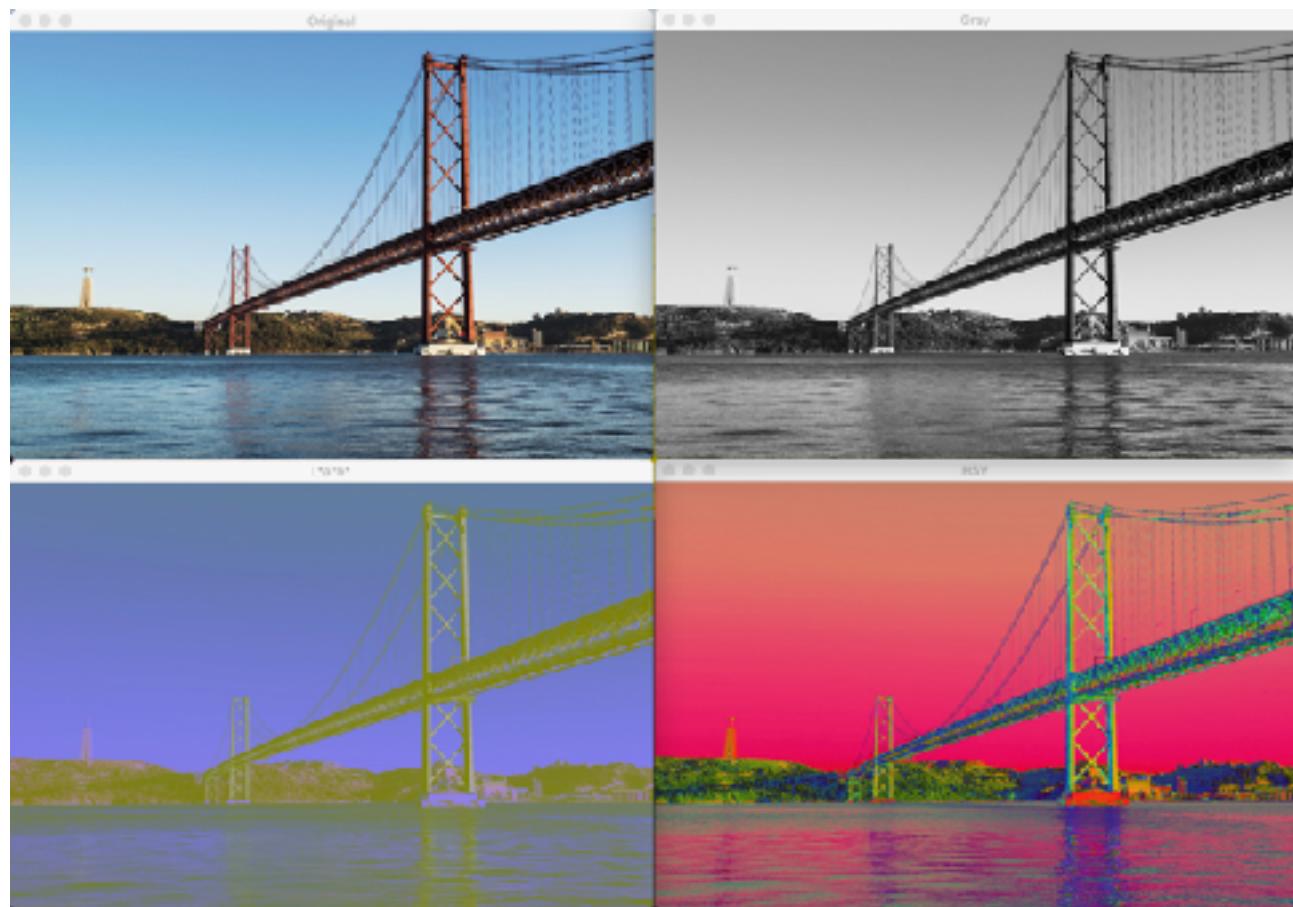


Figura 19 Outros sistemas de cores com a mesma imagem  
(Prog. ex\_openVC\_16.py)

O código para gerar o resultado acima é o seguinte :

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

cv2.imshow('Original', imagem)
gray = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
cv2.imshow('Gray', gray)
hsv = cv2.cvtColor(imagem, cv2.COLOR_BGR2HSV)
cv2.imshow('HSV', hsv)
lab = cv2.cvtColor(imagem, cv2.COLOR_BGR2LAB)
cv2.imshow('L*a*b*', lab)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(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:

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

cv2.imshow("Original", imagem)
(canalAzul, canalVerde, canalVermelho) = cv2.split(imagem)
cv2.imshow("Vermelho", canalVermelho)
cv2.imshow("Verde", canalVerde)
cv2.imshow("Azul", canalAzul)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

A função ‘split’ faz separação dos canais. Assim podemos exibi-los em tons de cinza conforme mostra a imagem abaixo:



Figura 20. Exibição dos 3 canais em tons de cinzento  
(Prog. ex\_openVC\_17.py)

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

imagem = cv2.imread('ponte.jpg')
(canalAzul, canalVerde, canalVermelho) = cv2.split(imagem)
zeros = np.zeros(imagem.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", imagem)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

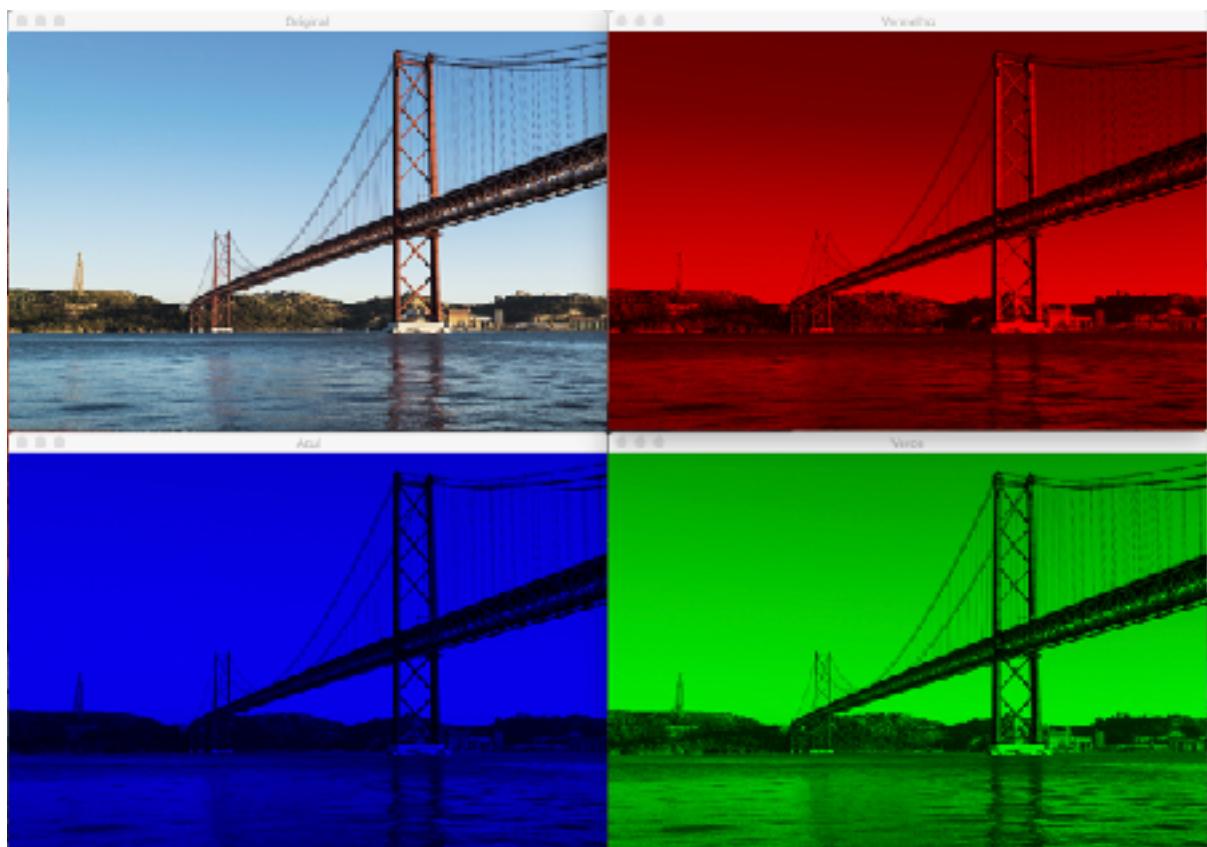


Figura 21 Exibição dos 3 canais separadamente (Prog. ex\_openVC\_18.py)

## 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 cinzento  
(Prog. ex\_openVC\_19.py)

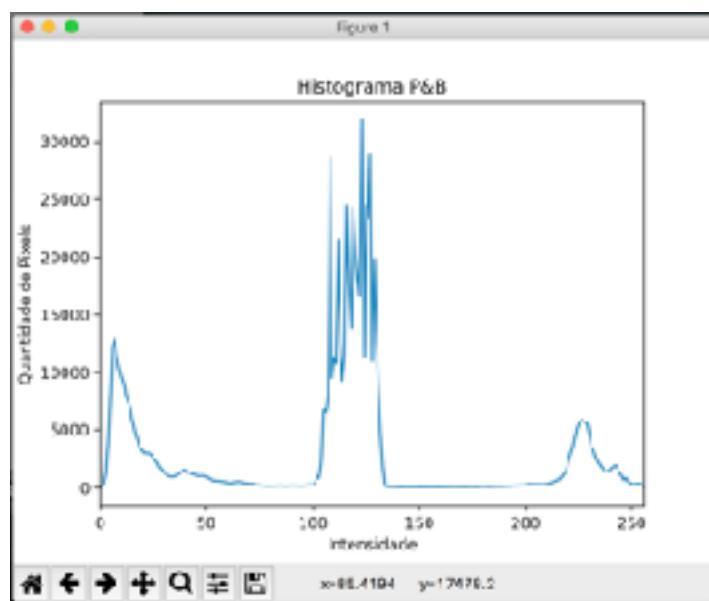


Figura 23 Histograma da imagem em tons de cinza  
(Prog. ex\_openVC\_19.py)

Verifique que no histograma existe um pico ao centro do gráfico, entre 100 e 150, demonstrando a grande quantidade de pixels nessa faixa devido ao edificação do

memorial que ocupa grande parte da imagem. O código para gerar o histograma segue abaixo:

```
from matplotlib import pyplot as plt
import cv2

imagem = cv2.imread('memo.jpg')
imagem = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
cv2.imshow('Imagen Preto Branco', imagem)

h = cv2.calcHist([imagem], [0], None, [256], [0, 256])
plt.figure()
plt.title('Histograma P&B')
plt.xlabel('Intensidade')
plt.ylabel('Quantidade de Pixels')
plt.plot(h)
plt.xlim([0, 256])
plt.show()

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

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.

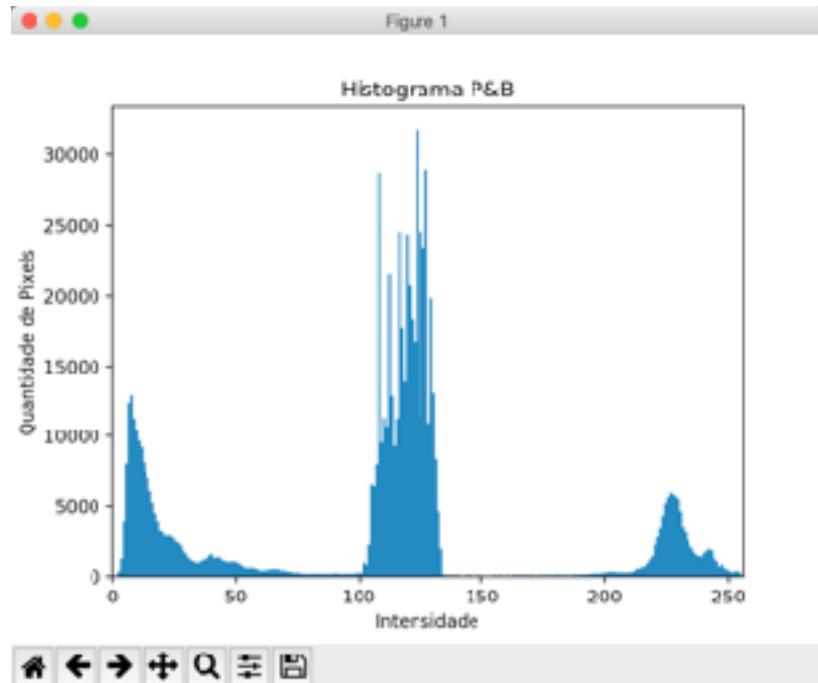


Figura 24. Histograma em barras  
(Prog. ex\_openVC\_19.py)

```
from matplotlib import pyplot as plt
import cv2

imagem = cv2.imread('memo.jpg')
imagem = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
cv2.imshow('Imagen Preto Branco', imagem)

h = cv2.calcHist([imagem], [0], None, [256], [0, 256])
plt.figure()
plt.title('Histograma P&B')
plt.xlabel('Intensidade')
plt.ylabel('Quantidade de Pixels')
plt.xlim([0, 256])
plt.hist(imagem.ravel(), 256, [0, 256])
plt.show()

cv2.waitKey(0)
cv2.destroyAllWindows()

# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

Além do histograma da imagem em tons de cinza é possível ‘plotar’ um histograma da imagem colorida. Neste caso teremos três linhas, uma por cada canal.

Veja o código necessário, abaixo. É importante notar que a função ‘zip’ cria uma lista de triplas formada pela união das linhas passadas e não tem a ver com o processo de compactarão como poderia esperar.

```
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('memo.jpg', -1)
cv2.imshow('Ponte', img)

color = ('b', 'g', 'r')

plt.figure()
plt.title('Histograma Colorido')
plt.xlabel('Intensidade')
plt.ylabel('Número de Pixels')

for channel, col in enumerate(color):
    historico = cv2.calcHist([img], [channel], None, [256], [0, 256])
    plt.plot(historico, color=col)
    plt.xlim([0, 256])
plt.show()

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



Figura 25a. Imagem importada ‘memo.jpg’

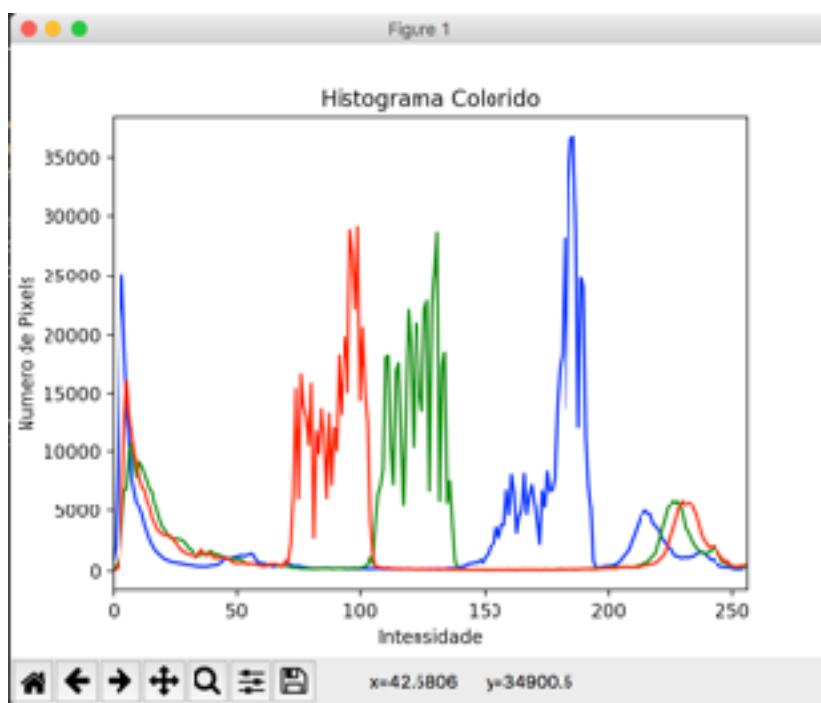


Figura 25b. Histograma colorido da imagem. Neste caso são ‘plotados’ os 3 canais RGB  
(Prog. ex\_openVC\_20.py)

## 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 a intensidade 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:

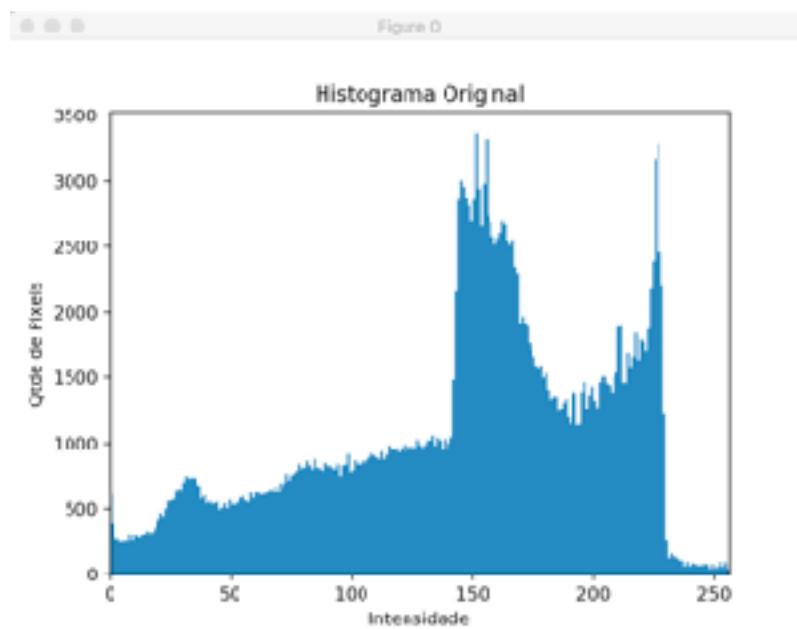


Figura 26. Histograma da imagem original  
Prog. ex\_openVC\_21.py)

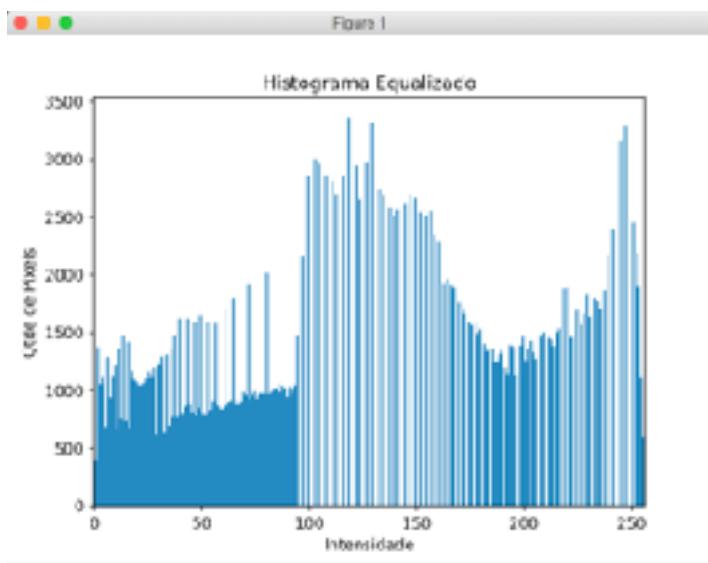


Figura 27. Histograma da imagem equalizado  
Prog. ex\_openVC\_21.py)

O código utilizado para gerar os dois histogramas segue abaixo:

```
from matplotlib import pyplot as plt
import cv2

img = cv2.imread('ponte.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imshow('Original BW', img)
h_eq = cv2.equalizeHist(img)
cv2.imshow('Imagen equalizada', h_eq)

plt.figure(0)
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(0)

plt.figure(1)
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(1)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

Na imagem a diferença também é perceptível, veja:



Figura 28. Imagem original (esquerda) e imagem cujo histograma foi equalizado (direita). Na imagem cujo histograma foi equalizado percebemos maior contraste

Conforme vemos na imagem é possível que ocorram distorções e alterações nas cores da imagem equalizada. 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 na própria documentação da OpenCV(\*) que mostra o seguinte exemplo:

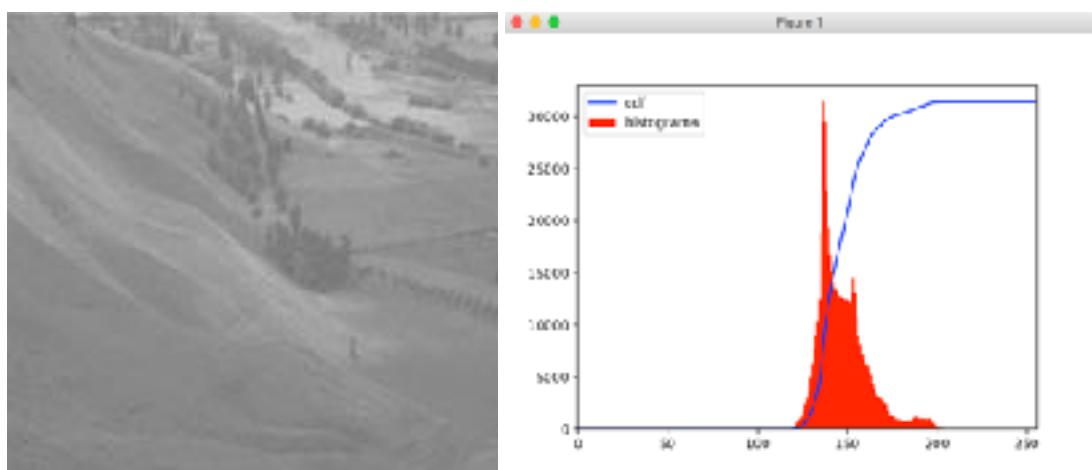


Figura 29. Exemplo extraído da documentação da OpenCV mostrando o histograma de uma imagem com baixo contraste. (Prog. ex\_openVC\_22.py)

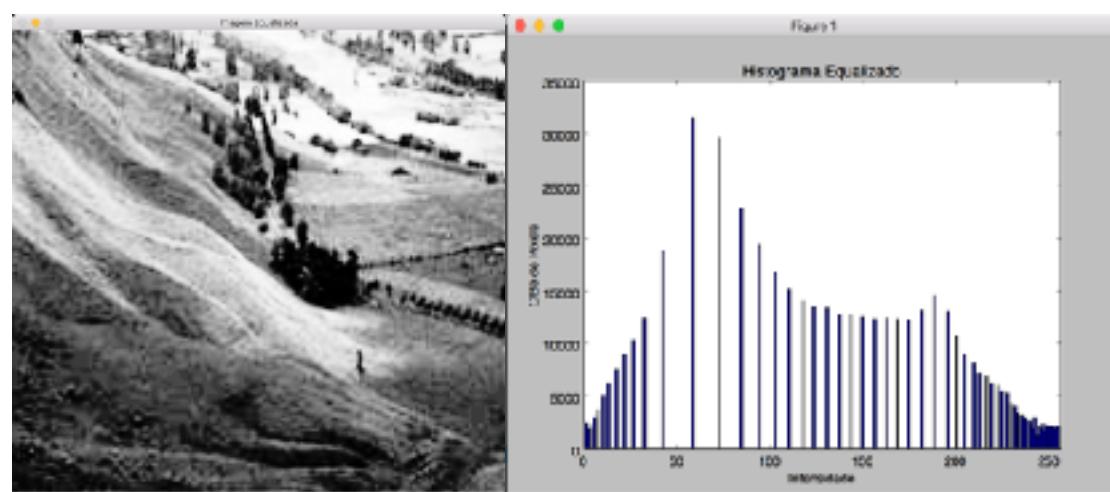


Figura 30. Exemplo extraído da documentação da OpenCV mostrando o histograma da mesma imagem mas desta vez com o histograma equalizado. (Prog. ex\_openVC\_23.py)

nota(\*) - [https://docs.opencv.org/3.1.0/d5/daf/tutorial\\_py\\_histogram\\_equalization.html](https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html)

Abaixo, o código utilizado para gerar o histogramas:

```

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

img = cv2.imread('wiki.jpg', 0)

hist, bins = np.histogram(img.flatten(), 256, [0, 256])

cdf = hist.cumsum()
cdf_m = np.ma.masked_equal(cdf, 0)
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf = np.ma.filled(cdf_m, 0).astype('uint8')
img2 = cdf[img]
cv2.imshow('Imagen Equalizada', img2)

plt.figure()
plt.title("Histograma Equalizado")
plt.xlabel("Intensidade")
plt.ylabel("Qtde de Pixels")
plt.hist(img2.flatten(), 256, [0,256], color= 'b')
plt.xlim([0, 256])
plt.show()

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)

```

Note que a equalização faz com que a distribuição da intensidade 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:

A função usa o seguinte algoritmo:

Passo 1: Calcula o histograma ‘H’ da imagem.

Passo 2: Normaliza o histograma para garantir que os valores da intensidade 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 intensidade 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.

## 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.

### 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.

Anote que usamos as funções vstack (pilha vertical) e hstack (pilha horizontal) para juntar as imagens numa 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.

```
import cv2
import numpy as np

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)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

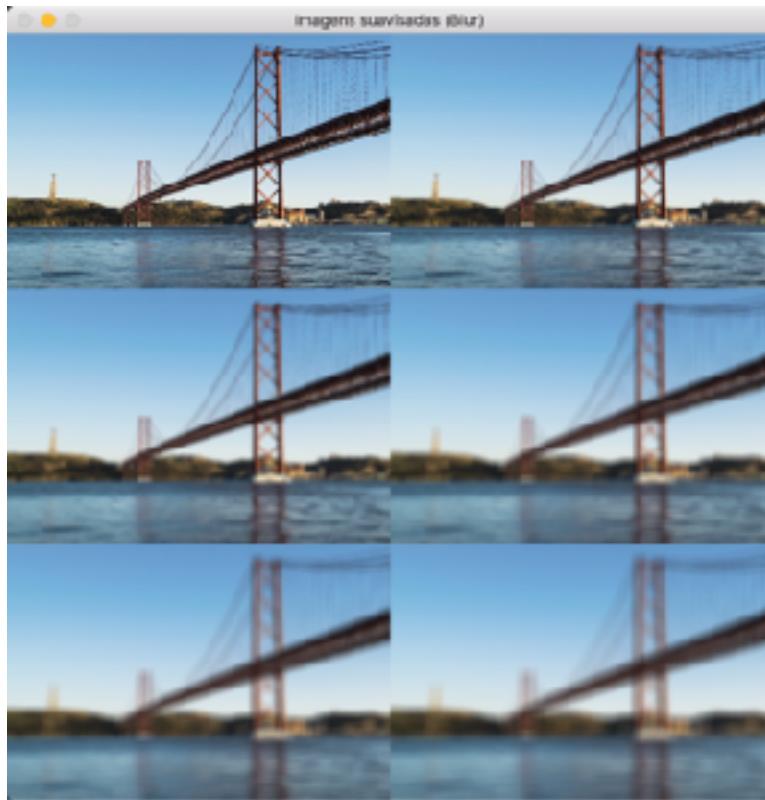


Figura 32 Imagem original seguida da esquerda para a direita e de cima para baixo com imagens tendo caixas de cálculo de 3x3, 5x5, 7x7, 9x9 e 11x11. Perceba que conforme aumenta a caixa maior é o efeito de borrão (blur) na imagem.  
(Prog. ex\_openVC\_24.py).

---

## 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).

```
import cv2
import numpy as np

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

img = img[::2,::2] # Diminui a imagem

suave = np.vstack([
    np.hstack([img,
              cv2.GaussianBlur(img, ( 3, 3), 0)]),
    np.hstack([cv2.GaussianBlur(img, ( 5, 5), 0),
              cv2.GaussianBlur(img, ( 7, 7), 0)]),
    np.hstack([cv2.GaussianBlur(img, ( 9, 9), 0),
              cv2.GaussianBlur(img, (11, 11), 0)])])
cv2.imshow("Imagen original e suavisadas pelo filtro "
           "Gaussiano", suave)
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



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. (Prog. ex\_openVC\_25.py).

---

### 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 cv2
import numpy as np

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

img = img[::2, ::2] # Diminui a imagem
suave = np.vstack([
    np.hstack([img,
               cv2.medianBlur(img, 3)]),
    np.hstack([cv2.medianBlur(img, 5),
               cv2.medianBlur(img, 7)]),
    np.hstack([cv2.medianBlur(img, 9),
               cv2.medianBlur(img, 11)]), ])
cv2.imshow("Imagen original e suavisadas pela mediana", suave)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



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. (Prog. ex\_openVC\_26.py).

## 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 dum filtro gaussiano do espaço em 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:

```
import cv2
import numpy as np

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)]), ])
cv2.imshow("Imagen original e suavisadas pela mediana", suave)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



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. (Prog. ex\_openVC\_27.py).

## 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.

```
import cv2
import numpy as np

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

img = img[::2, ::2] # Diminui a imagem
suave = 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("Binarizacao da imagem", resultado)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

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

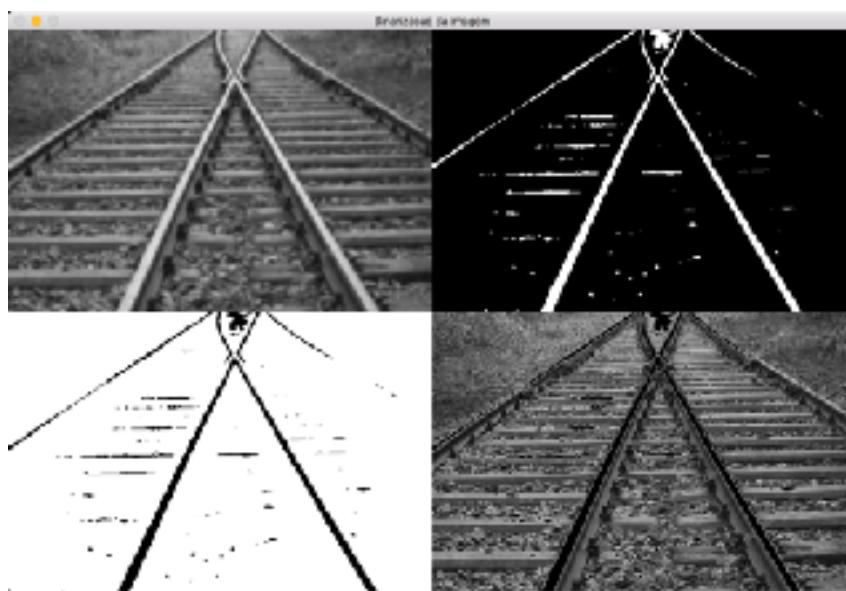


Figura 36 Da esquerda para a direta e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada e a imagem binarizada invertida.  
(Prog. ex\_openVC\_28.py).

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

---

## Threshold adaptive

O valor de intensidade 160 utilizada para a binarização acima foi arbitrado, contudo, é possível otimizar esse valor matematicamente. Esta é a proposta do threshold adaptativo. Para isso precisamos dar um valor da janela ou caixa de cálculo para que o limiar seja calculado nos pixels próximos das imagens. Outro parâmetro é um inteiro que é subtraído da média calculada dentro da caixa para gerar o threshold final.

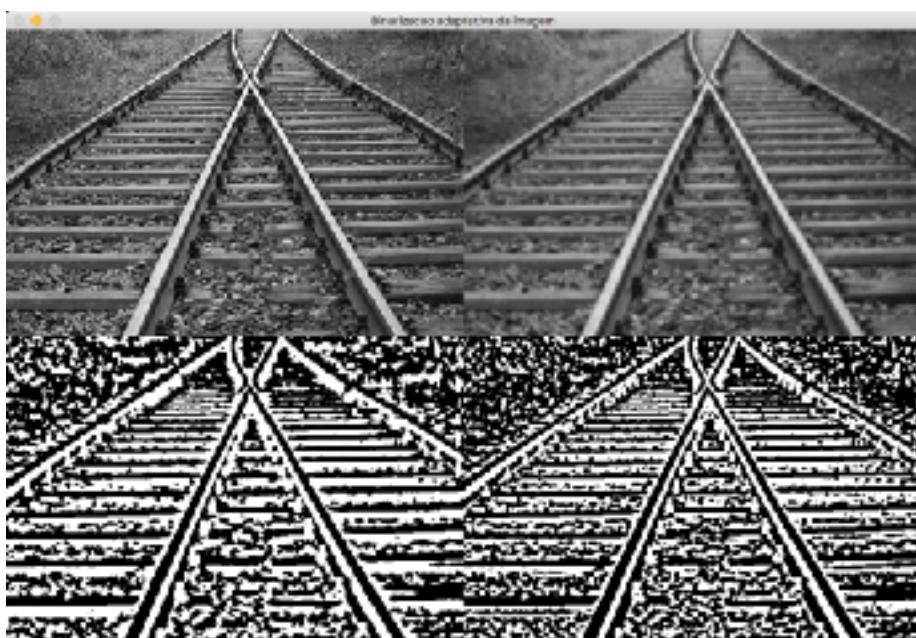


Figura 37. Threshold adaptativo. Da esquerda para a direta e de cima para baixo temos: a imagem, a imagem suavizada, a imagem binarizada pela média e a imagem binarizada com Gauss  
(Prog. ex\_openVC\_29.py).

```
import cv2
import numpy as np

img = cv2.imread('carris.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("Binarizacao adaptativa da imagem", resultado)
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

---

## 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

---

### 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:

```
img = cv2.imread('carris.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # converte

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)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

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.

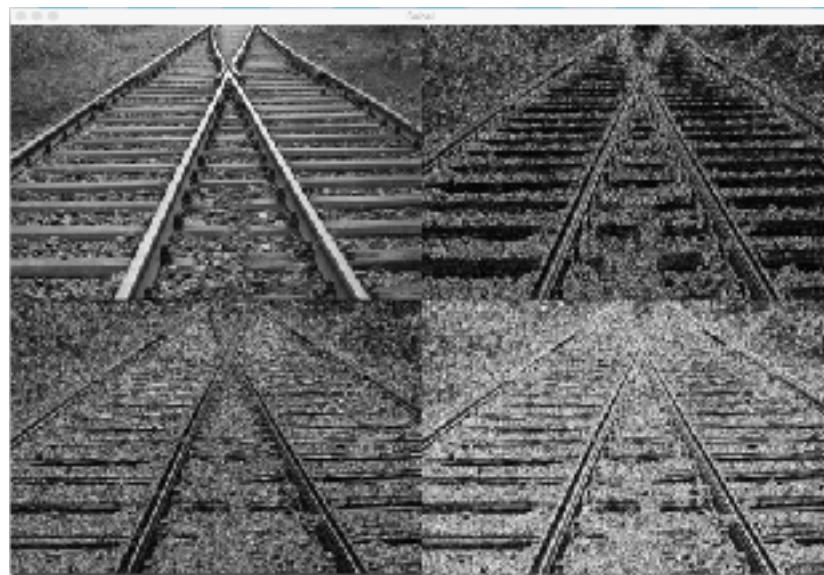


Figura 39. Da esquerda para a direita e de cima para baixo temos: a imagem original, Sobel Horizontal (sobelX), Sobel Vertical (sobelY) e a imagem com o Sobel combinado que é o resultado final. ([Prog. ex\\_openVC\\_30.py](#)).

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

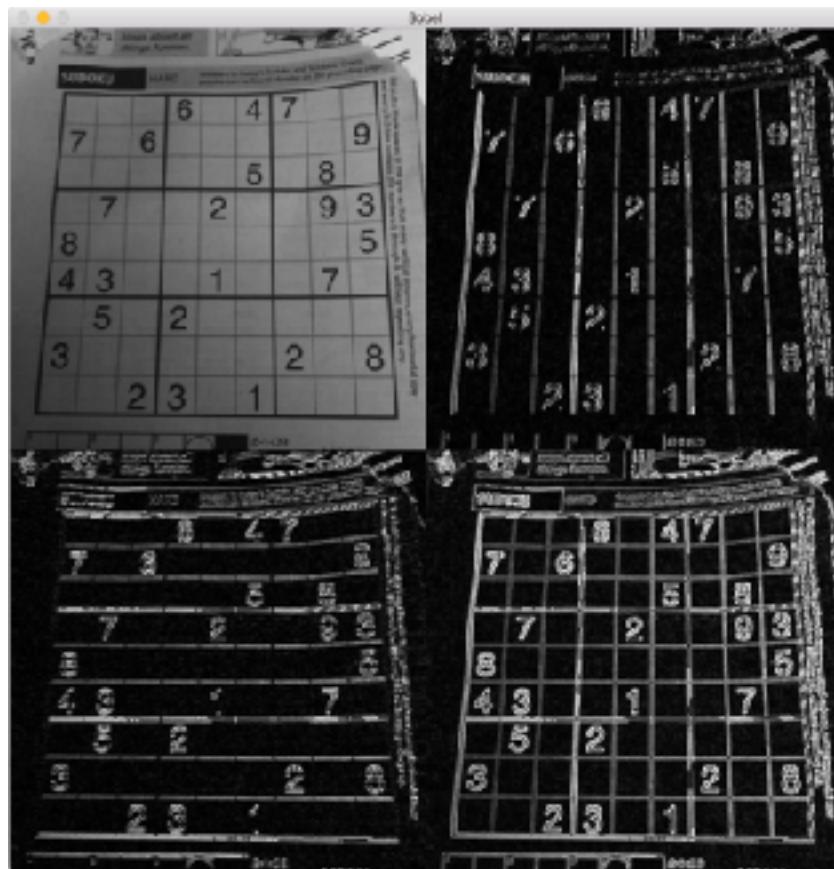


Figura 40. Exemplo de processamento Sobel Vertical e Horizontal disponível na documentação da OpenCV. ([Prog. ex\\_openCV\\_30.py](#)).

## 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. (*Prog. ex\_openCV\_31.py*)

A baixo é descrito o código

```
import cv2
import numpy as np

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)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

---

## 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 ligadas 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 ligados.

A baixo é descrito o código:

```
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)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



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).  
(Prog. ex\_openCV\_32.py)

---

## Identificando e contando objetos

Como sabemos, 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, num 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 quantidade de dados presentes na imagem.

O código para gerar as saídas acima segue abaixo comentado:

```
import numpy as np
import cv2
import mahotas

#Funcao 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 necessario o redimensioamento da imagem pode vir aqui.
# Passo 1: Conversao para tons de cinza
img = cv2.cvtColor(imgColorida, cv2.COLOR_BGR2GRAY)

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

# Passo 3: Binarizacao 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: Deteccao de bordas com Canny
bordas = cv2.Canny(bin, 70, 150)

# Passo 5: Identificacao 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 variavel lx (lixo) recebe dados que nao sao utilizados
escreve(img, "Imagen em tons de cinza", 0)
escreve(suave, "Suavizacao com Blur", 0)
escreve(bin, "Binarizacao com Metodo Otsu", 255)
escreve(bordas, "Detector de bordas Canny", 255)
temp = np.vstack([
    np.hstack([img, suave]),
    np.hstack([bin, bordas])
])
cv2.imshow("Quantidade de objetos: "+str(len(objetos)), temp)
cv2.waitKey(0)
imgC2 = imgColorida.copy()
cv2.imshow("Imagen Original", imgColorida)
cv2.drawContours(imgC2, objetos, -1, (255, 0, 0), 2)
escreve(imgC2, str(len(objetos))+" objetos encontrados!")
cv2.imshow("Resultado", imgC2)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```

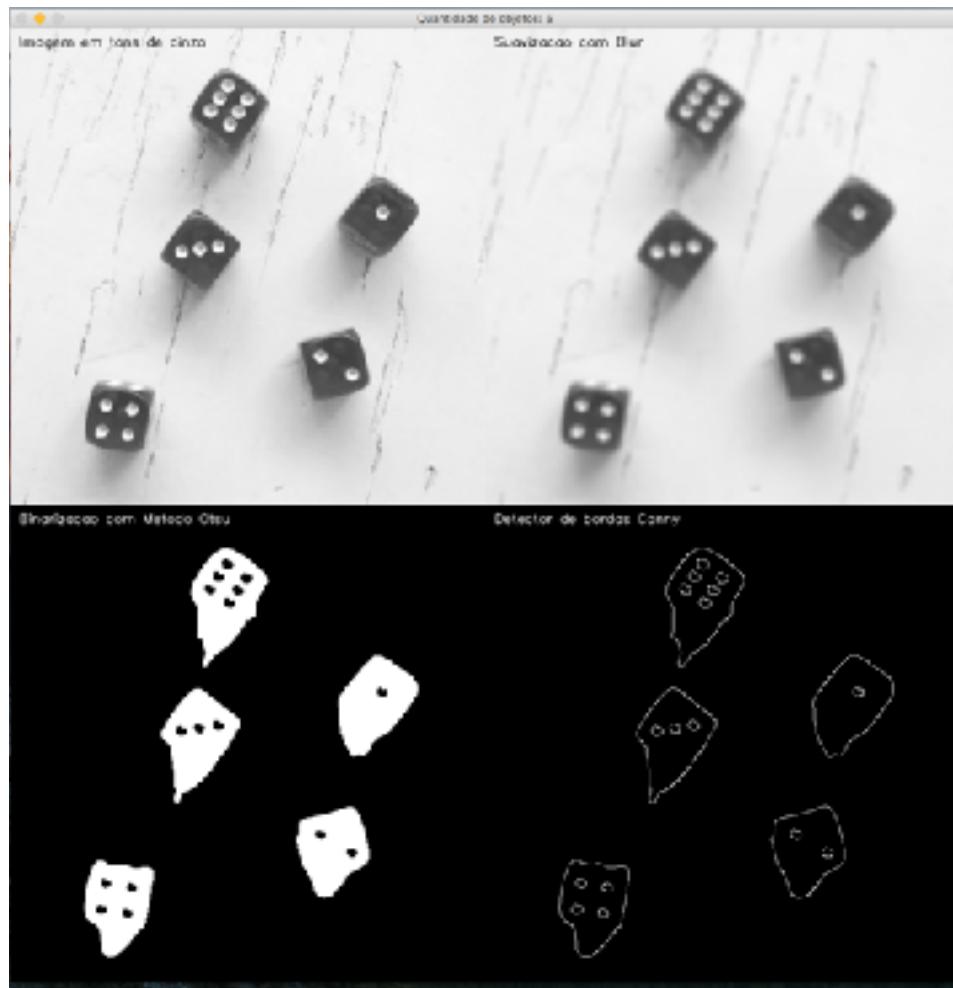


Figura 44 Passos para identificar e contar os dados na imagem.  
(Prog. ex\_openCV\_33.py)



Figura 45. Resultado sobre a imagem original  
(Prog. ex\_openCV\_33.py)

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

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

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

## Detectção de faces em imagens

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

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

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

Detection. As duas referências seguem abaixo:

*Paul Viola and Michael J. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. IEEE CVPR, 2001. The paper is available online at:  
<https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf>*

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

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

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

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

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

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

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

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

```
import cv2

# Carrega arquivo e converte para tons de cinzento
img = cv2.imread('caras.jpg')
iPB = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Criacao do detector de faces
df = cv2.CascadeClassifier('haarcascade_eye.xml')

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

# Desenha retangulos amarelos na iamgem original (colorida)
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 255), 3)

# Exibe imagem. Titulo da janela exibe numero de faces
cv2.imshow(str(len(faces))+' face(s) encontrada(s)', img)

cv2.waitKey(0)
cv2.destroyAllWindows()
# On Mac need to force to destroyAllWindows.
cv2.waitKey(1)
```



Figura 46. Identificação de face frontal. (prog. ex\_openCV\_34)

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

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

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

```
# (Prog. ex_openCV_36.py)
import os
import cv2

# Faz a varredura do diretório imagens buscando arquivos JPG, JPEG e PNG.
diretório = 'imgs'
arquivos = os.listdir(diretório)
for a in arquivos:
    if a.lower().endswith('.jpg') or a.lower().endswith('.png') \
       or a.lower().endswith('.jpeg'):

        imgC = cv2.imread(diretório+'/'+a)
        imgPB = cv2.cvtColor(imgC, cv2.COLOR_BGR2GRAY)
        df = cv2.CascadeClassifier('data/haarcascade_frontalface_default.xml')
        faces = df.detectMultiScale(
            imgPB,
            scaleFactor=1.2,
            minNeighbors=2,
            minSize=(30, 30),
            flags=cv2.CASCADE_SCALE_IMAGE)

        for (x, y, w, h) in faces:
            cv2.rectangle(imgC, (x, y), (x + w, y + h), (0, 255, 255), 2)
        alt = int(float(imgC.shape[0])/float(imgC.shape[1])*640))
        imgC = cv2.resize(imgC, (640, alt), interpolation=cv2.INTER_CUBIC)
        cv2.imshow(str(len(faces))+' face(s) encontrada(s).', imgC)

        cv2.waitKey(0)
        cv2.destroyAllWindows ()
        # On Mac need to force to destroyAllWindows.
        cv2.waitKey(1)
```

## Deteção de faces em vídeos

O processo de detecção de objetos em vídeos é muito similar a detecção em imagens. Na verdade um vídeo nada mais é do que um fluxo contínuo de imagens que são enviadas a partir da fonte como uma webcam ou ainda um arquivo de vídeo mp4. Um looping é necessário para processar o fluxo contínuo de imagens do vídeo. Para facilitar a compreensão veja o exemplo abaixo:

```
import cv2

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

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

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

# Tambem e possivel abrir a proprio webcam
# do sistema para isso segue codigo abaixo
#camera = cv2.VideoCapture(0)

while True:
    #read() retorna 1-Se houve sucesso e 2-0 proprio frame
    (sucesso, frame) = camera.read()
    if not sucesso: #final do video
        break

    #reduz tamanho do frame para acelerar processamento
    frame = redim(frame, 320)

    #converte para tons de cinza
    frame_pb = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    #detecta as faces no frame
    faces = df.detectMultiScale(frame_pb,
                                scaleFactor = 1.1,
                                minNeighbors=3,
                                minSize=(20,20),
                                flags=cv2.CASCADE_SCALE_IMAGE)

    frame_temp = frame.copy()
    for (x, y, lar, alt) in faces:
        cv2.rectangle(frame_temp,
                      (x, y),
                      (x + lar, y + alt),
                      (0, 255, 255), 2)

    #Exibe um frame redimensionado (com perca de qualidade)
    cv2.imshow("Encontrando faces...", redim(frame_temp, 640))
```

```
#Espera que a tecla 's' seja pressionada para sair
if cv2.waitKey(1) & 0xFF == ord("s"):
    break

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

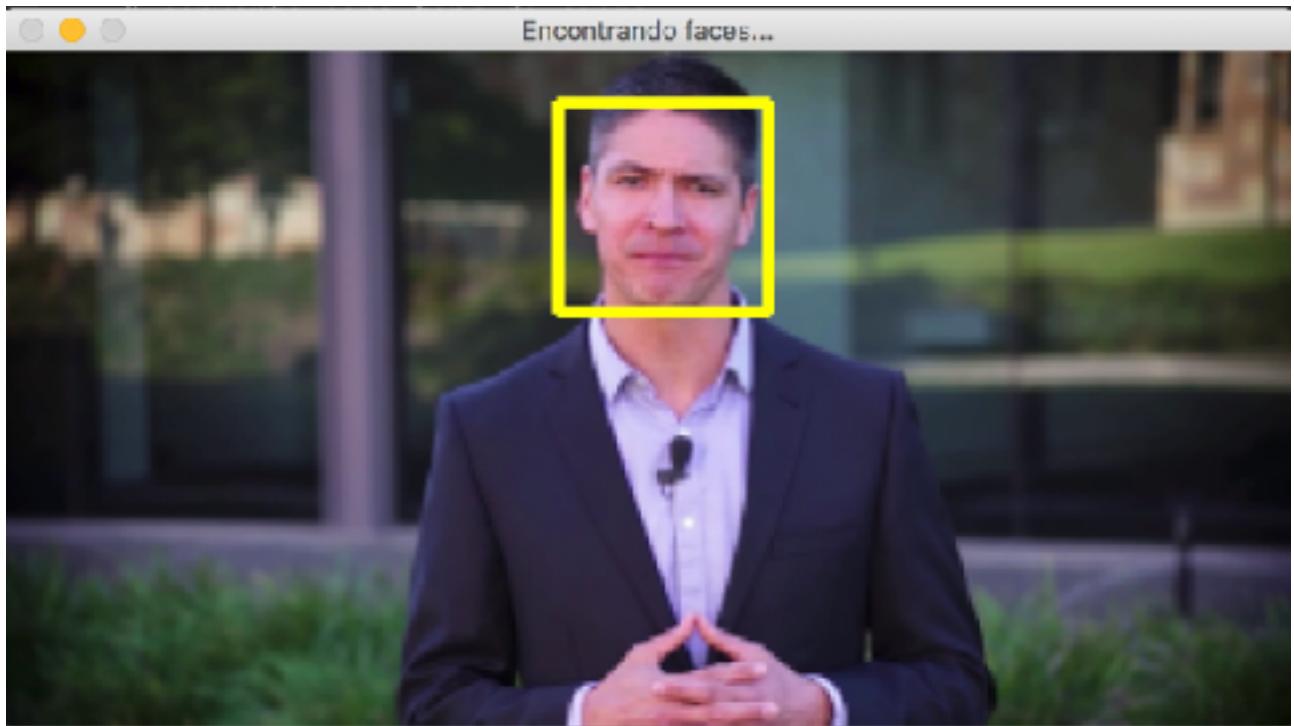


Figura 48. Detecção em vídeo. (Prog. ex\_openCV\_36.py)

---

## Rastreamento de objetos em vídeos

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

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

```
# (Prog. ex_openCV_37.py)
import numpy as np
import cv2

azulEscuro = np.array([100, 67, 0], dtype = "uint8")
azulClaro = np.array([255, 128, 50], dtype = "uint8")

# camera = cv2.VideoCapture(args["video"])
camera = cv2.VideoCapture('video.mp4')

while True:
    (sucesso, frame) = camera.read()
    if not sucesso:
        break

    obj = cv2.inRange(frame, azulEscuro, azulClaro)
    obj = cv2.GaussianBlur(obj, (3, 3), 0)
    (_, cnts, _) = cv2.findContours(obj.copy(),
                                    cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    if len(cnts) > 0:
        cnt = sorted(cnts, key=cv2.contourArea, reverse=True)[0]
        rect = np.int32(cv2.boxPoints(cv2.minAreaRect(cnt)))
        cv2.drawContours(frame, [rect], -1, (0, 255, 255), 2)
        cv2.imshow("Tracking", frame)
        cv2.imshow("Binary", obj)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

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

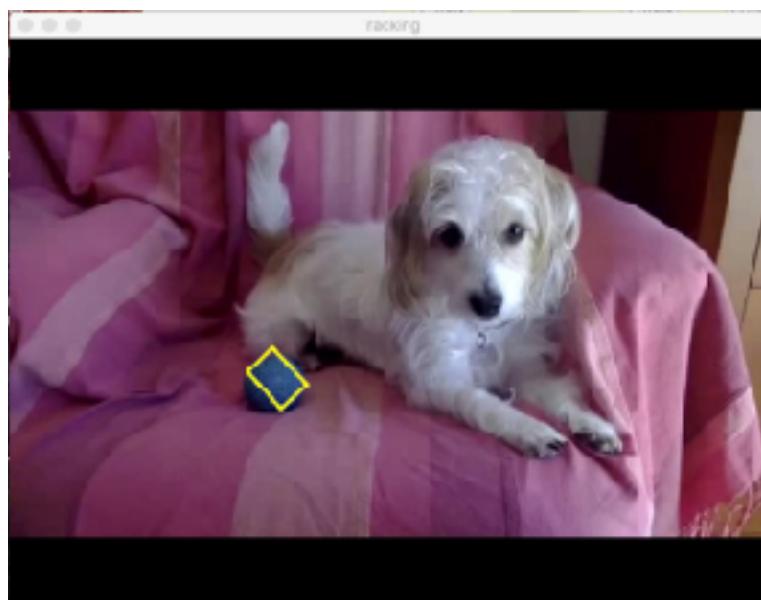


Figura 49. Detecção em vídeo - 'blue mask' (Prog. ex\_openCV\_37.py)

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

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