

# Introdução à visão computacional e ao processamento de imagens com OPENCV

## Módulo I - processamento de imagens

Thiago T. Santos  
[thiago.santos@embrapa.br](mailto:thiago.santos@embrapa.br)

Embrapa Informática Agropecuária

19, 20 e 21 de fevereiro de 2013



# O que é Visão Computacional?

*Visão computacional é a transformação de dados provenientes de imagem ou vídeo em uma **decisão** ou em uma **nova interpretação**.*

*Bradski & Kaehler, Learning OpenCV*

*(...) **técnicas matemáticas** para recuperar a **forma tridimensional** e a aparência de objetos em imagens.*

*Szeliski, Computer Vision: Algorithms and Applications*

# O que é OpenCV?

*Open Source Computer Vision Library*

- ▶ Uma biblioteca para VC escrita em C/C++
  - ▶ Diversas otimizações em C/C++
  - ▶ Diversos algoritmos têm implementação em GPU
- ▶ APIs para C++, Java e Python
  - ▶ Integração com NUMPY
- ▶ Utilizada extensivamente pela comunidade de VC
  - ▶ IBM, Microsoft, Intel, Sony, Siemens, Google,...
  - ▶ Stanford, MIT, CMU, Cambridge, INRIA, USP,...
- ▶ Licença BSD permite o desenvolvimento de
  - ▶ **software livre**
  - ▶ **comercial**

# O que um computador “vê”?

Um *summer project* que já dura quase 50 anos

Em 1966, Marvin Minsky (MIT) pediu a seu aluno Gerald J. Sussman:

*Gaste o verão ligando uma câmera a um computador e faça-o descrever o que ele vê.*

Passamos os últimos 47 anos resolvendo o que Sussman não conseguiu!

# O que um computador “vê”?

Uma imagem é uma matriz



158	161	163	131	50	53	59	50	63
159	159	163	162	124	42	53	80	65
159	154	163	165	166	122	26	40	65
158	158	162	163	165	166	118	25	48
151	154	156	166	163	161	158	101	18
153	158	159	163	159	163	159	148	72
155	156	153	158	159	161	159	156	119
148	153	159	158	154	154	154	150	135
144	150	153	153	152	153	148	145	145
151	153	151	151	149	149	148	147	137

# O que um computador “vê”?

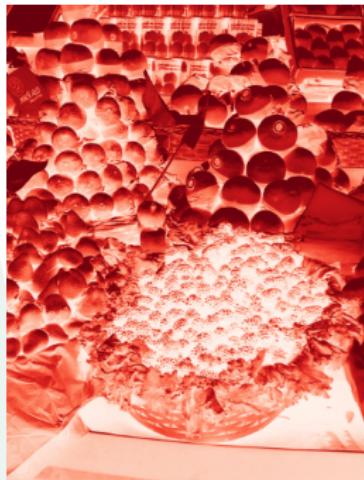
Uma imagem colorida é formada por 3 matrizes



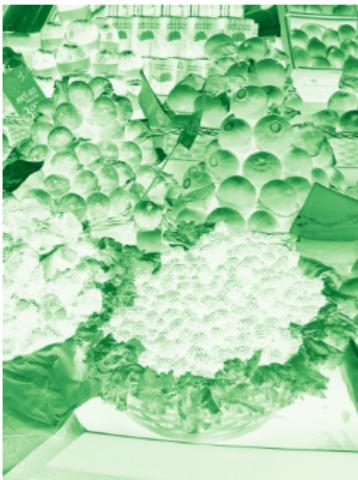
# O que um computador “vê”?

Uma imagem colorida é formada por 3 matrizes

R



G



B

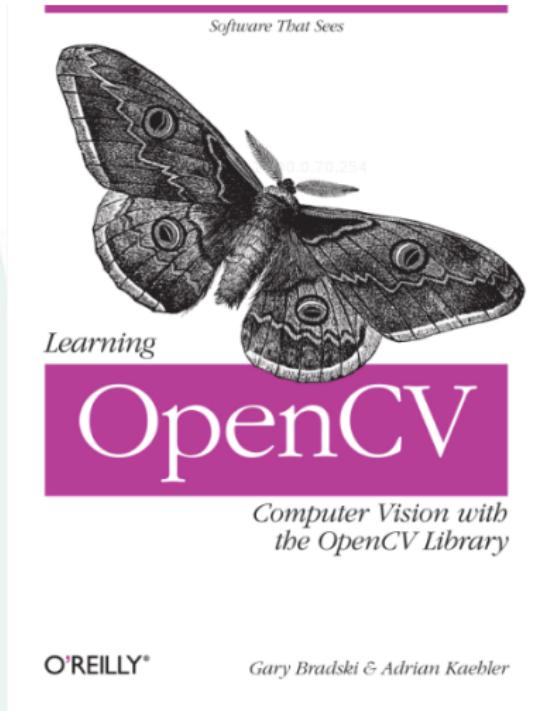


- ▶ **1999** OPENCV nasce na **Intel**
  - ▶ Iniciativa de **Gary Bradski**
  - ▶ Promover uso intensivo de CPU
  - ▶ Promover *Intel's Integrated Performance Primitives* (IPP)
  - ▶ Desenvolvida na Intel Rússia sob **Vadim Pisarevsky**
- ▶ **2000** Primeira versão Beta
- ▶ **2006** Lançamento da versão 1.0
- ▶ **2008** **Willow Garage** assume o desenvolvimento
- ▶ **2009** Lançamento da versão 2.0
- ▶ **2011** Versão 2.3.1
- ▶ **2012** Versão 2.4
- ▶ **2013** Versão 2.4.6, API para Java

- ▶ **Portal principal**  
<http://opencv.org>
- ▶ **Documentação das APIs, produzida via Sphinx, em**  
<http://docs.opencv.org>
- ▶ **Q&A**  
<http://answers.opencv.org>
- ▶ **Tutoriais**  
<http://docs.opencv.org/doc/tutorials/tutorials.html>
- ▶ **Livros**

# Documentação

## Livros



### Learning OpenCV

Autores Gary Bradski  
Adrian Kaeblner

Editora O'Reilly Media, Inc.

Data setembro de 2008

ISBN-13 978-0-596-51613-0

<http://goo.gl/PEcaV>

**Aguarde a 2<sup>a</sup> edição - out. de 2013**

<http://goo.gl/L26hf>

# Documentação

## Livros



## OpenCV 2 Computer Vision Application Programming Cookbook

Autor Robert Laganière

Editora PACKT Publishing

Data maio de 2011

ISBN-13 978-1-84951-324-1

<http://goo.gl/6B1xU>

# Documentação

## Livros



## Mastering OpenCV with Practical Computer Vision Projects

Autor Daniel Baggio *et al.*

Editora PACKT Publishing

Data dezembro de 2012

ISBN-13 9781849517829

<http://goo.gl/MbhTJ>

- ▶ **GNU/Linux**  
<http://goo.gl/8ds4i>
- ▶ **GNU/Linux (ARM)**  
<http://goo.gl/LygrF>
- ▶ **Windows**  
<http://goo.gl/nLlIi>
- ▶ **iOS**  
<http://goo.gl/N9qqi>
- ▶ **Desenvolvimento para Android**  
<http://goo.gl/uTcId>

# Ambiente de desenvolvimento utilizado neste curso

## IPython

- ▶ IPython
  - ▶ Ambiente interativo para computação científica
  - ▶ Interface com NumPy e Matplotlib
- ▶ Alternativa: Eclipse
  - ▶ CDT para desenvolvimento C/C++
  - ▶ PyDev para desenvolvimento Python

# Ambiente de desenvolvimento utilizado neste curso

## IPython

```
IPython
File Edit View Kernel Magic Window Help
Python 2.7.4 (default, Apr 19 2013, 18:28:01)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.2 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%gui      -> A brief reference about the graphical user interface.

In [1]: %pylab inline

Welcome to pylab, a matplotlib-based Python environment [backend: module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.

In [2]: import cv2

In [3]: I = cv2.imread('/home/aluno/Imagens/lenna.tiff')

In [4]: I = cv2.cvtColor(I, cv2.COLOR_BGR2RGB)

In [5]: imshow(I)
Out[5]: <matplotlib.image.AxesImage at 0x2830950>


```

# Outline

Introdução à OpenCV

Operações com matrizes

Cor

Operações básicas em processamento de imagens

Transformações de imagem

Histogramas

# Hello, world, estilo OpenCV

## Utilização da API C++

```
1 #include <opencv2/core/core.hpp>
2 #include <opencv2/imgproc/imgproc.hpp>
3 #include <opencv2/highgui/highgui.hpp>
4 #include <iostream>
5
6 using namespace cv;
7
8 int main(int argc, char **argv) {
9
10    /* Carregar uma imagem a partir de um arquivo */
11    char * file_path = argv[1];
12    Mat image = imread(file_path, CV_LOAD_IMAGE_COLOR);
13
14    /* Exibir a imagem */
15    imshow("Display", image);
16
17    /* Aguardar um sinal do teclado antes de fechar a janela e
18     * terminar o programa */
19    waitKey(0);
```

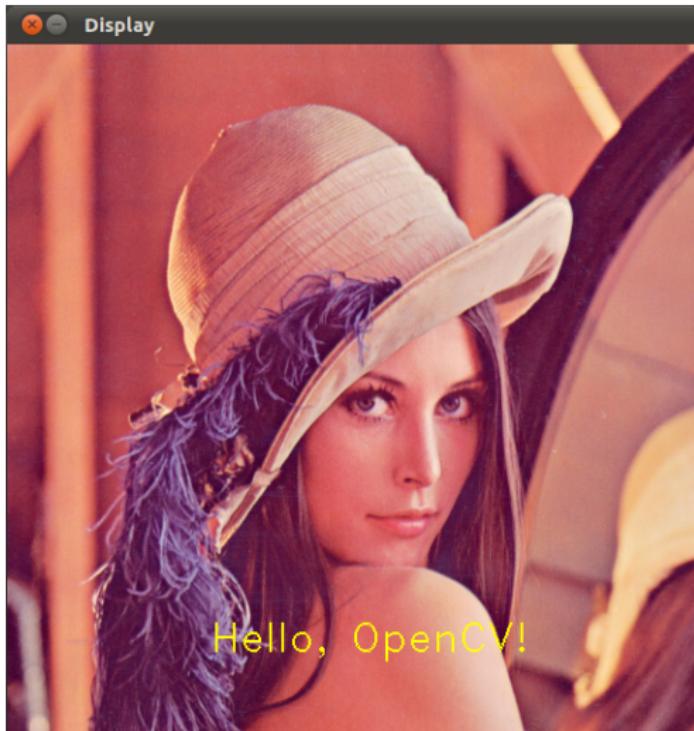
# Hello, world, estilo OpenCV

## Utilização da API C++

```
20
21 /* Definindo uma cor em BGR (blue, green, red - 8 bits cada) */
22 Scalar yellow = Scalar(0, 255, 255);
23
24 /* Escrever uma mensagem na imagem, alterando seus pixels */
25.putText(image, "Hello, OpenCV!", Point(150, 450),
26         CV_FONT_HERSHEY_DUPLEX, 1.0, yellow);
27
28 /* Exibir a imagem */
29 imshow("Display", image);
30 waitKey(0);
31
32 /* Desalocar memoria (nao eh realmente necessario ja que
33 * estamos no fim do programa mas... */
34 image.deallocate();
35
36 return 0;
}
```

# Hello, world, estilo OpenCV

Utilização da API C++



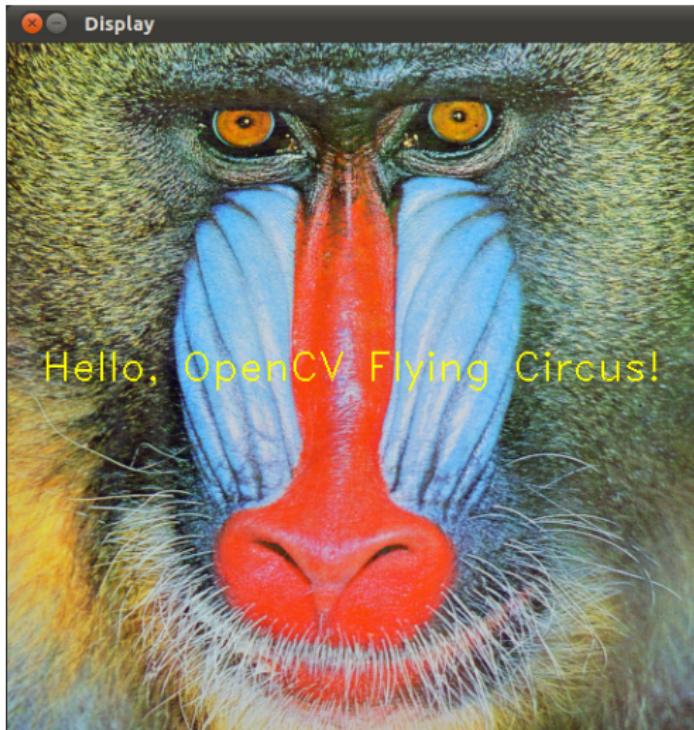
# Hello, world, estilo OpenCV

## Utilização da API Python

```
1 import cv2
2 import sys
3
4 if __name__ == '__main__':
5
6     # Carregar uma imagem a partir de um arquivo
7     image = cv2.imread(sys.argv[1])
8
9     # Exibir a imagem e aguardar uma entrada do teclado
10    cv2.imshow('Display', image)
11    cv2.waitKey(0)
12
13    # Definindo uma cor em BGR (blue, green, red - 8 bits cada)
14    yellow = (0, 255, 255)
15    cv2.putText(image, "Hello, OpenCV Flying Circus!", (25, 250),
16                , cv2.FONT_HERSHEY_DUPLEX, 1.0, yellow)
17    cv2.imshow('Display', image)
18    cv2.waitKey(0)
```

# Hello, world, estilo OpenCV

Utilização da API Python



# Sobre pylab.imshow

## O que é pylab?

- ▶ **pylab** é um ambiente de computação numérica para Python, proposto como uma alternativa ao Matlab
- ▶ Composto por dois módulos
  - `numpy` implementação de arrays N-dimensionais
  - `matplotlib` biblioteca para 2D *plotting*
- ▶ **pylab.imshow** pertence à biblioteca **matplotlib.pyplot**

# Sobre pylab.imshow

Por que utilizar **pylab.imshow** ao invés de **cv2.imshow**?

1. Porque **pylab.imshow** utiliza um **mapeamento de cores** que permite visualizar regiões de baixo contraste
2. Porque **pylab.imshow** permite visualização de imagens em ponto flutuante fora do intervalo [0,255] ou [0,1]
3. Porque **pylab.imshow** permite **zoom** em regiões da imagem
4. Porque **pylab.imshow** redimensiona automaticamente imagens muito pequenas ou muito grandes

# Sobre pylab.imshow

## Como utilizar pylab.imshow?

### pylabdemo.py

```
1 import cv2
2 import pylab
3 from sys import argv
4
5 if __name__ == '__main__':
6
7     image = cv2.imread(argv[1], cv2.CV_LOAD_IMAGE_COLOR)
8     pylab.imshow(image)
9     pylab.title('Imagen BGR')
10    pylab.show()
11
12    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
13    pylab.imshow(image)
14    pylab.title('Imagen RGB')
15    pylab.show()
```

# Sobre pylab.imshow

## Como utilizar pylab.imshow?

### pylabdemo.py

```
17 [r, g, b] = cv2.split(image)
18 pylab.imshow(r, cmap=pylab.cm.Reds)
19 pylab.colorbar()
20 pylab.title('Canal R visto com um colormap vermelho')
21 pylab.show()
22
23 image = cv2.imread(argv[1], cv2.CV_LOAD_IMAGE_GRAYSCALE)
24 pylab.imshow(image)
25 pylab.colorbar()
26 pylab.title('Imagen em tons de cinza vista com colormap
27         default')
28 pylab.show()
```

# Sobre pylab.imshow

## Como utilizar pylab.imshow?

### pylabdemo.py

```
29     pylab.imshow(image, cmap=pylab.cm.gray)
30     pylab.colorbar()
31     pylab.title('Imagen em tons de cinza vista com colormap
32                 cinza')
33     pylab.show()
34
35     pylab.imshow(image, cmap=pylab.cm.gray, interpolation='
36                 nearest')
37     pylab.colorbar()
38     pylab.title('Imagen vista com intepolation=\\'nearest\\\'')
39     pylab.show()
```

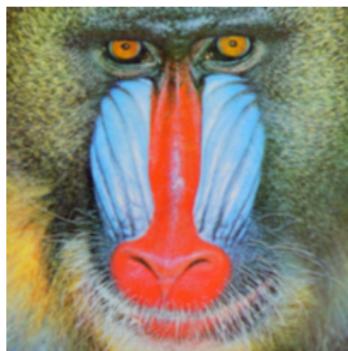
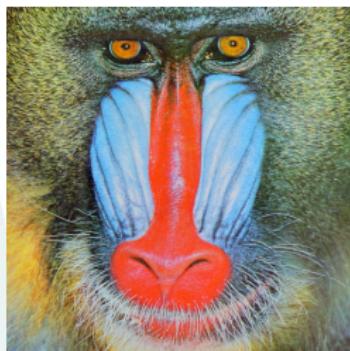
# Uma transformação simples

## Borramento Gaussiano (*Gaussian Blur*)

```
1 import cv2, sys, numpy as np
2
3 if __name__ == '__main__':
4
5     source = cv2.imread(sys.argv[1], cv2.CV_LOAD_IMAGE_COLOR)
6     cv2.imshow("Source", source)
7
8     smooth = np.zeros(source.shape, dtype=source.dtype)
9     cv2.GaussianBlur(source, (5,5), 3.0, smooth)
10    cv2.imshow("Smooth", smooth)
11    key = cv2.waitKey(0)
12
13    while (key & 255) != ord('Q') and (key & 255) != ord('q'):
14        cv2.GaussianBlur(smooth, (5,5), 3.0, smooth)
15        cv2.imshow("Smooth", smooth)
16        key = cv2.waitKey(0)
```

# Uma transformação simples

Borramento Gaussiano (*Gaussian Blur*)



# Imagens são matrizes N-dimensionais

Exemplo com C++

profile.cpp

```
13 Mat image = imread(file_path, CV_LOAD_IMAGE_COLOR);  
14  
15 cout << file_path << " eh uma imagem com " << image.cols << " x "  
16     << image.rows << " pixels\n";  
17 cout << file_path << " possui " << image.channels() << "  
18     canais\n";  
19  
20 // Certifique-se que se tratam de 3 canais de 8 bits  
21 CV_Assert(image.type() == CV_8UC3);  
22  
23 // A tupla BGR eh representada por um vetor com 3 dimensoes  
24 Vec3b pix_val = image.at<Vec3b>(11,7);  
25 cout << "O pixel na posicao (11,7) apresenta os valores "  
26     << (int) pix_val[0]  
     << ", " << (int) pix_val[1]  
     << ", " << (int) pix_val[2];
```

# Imagens são matrizes N-dimensionais

Exemplo com C++

```
./data/mandrill.tiff eh uma imagem com 512 x 512 pixels  
./data/mandrill.tiff possui 3 canais  
O pixel na posicao (11,7) apresenta os valores 56, 135,  
132
```

# Imagens são matrizes N-dimensionais

Exemplo com Python e NumPy

profile.py

```
6     image = cv2.imread(file_path, cv2.CV_LOAD_IMAGE_COLOR)
7
8     rows, cols, channels = image.shape
9     print '%s eh uma imagem com %d x %d pixels' % (file_path,
10           rows, cols)
11    print '%s possui %d canais' % (file_path, channels)
12
13    pix_val = image[11][7]
14    print 'O pixel na posicao (11,7) apresenta os valores %s' %
15          str(pix_val)
```

# Imagens são matrizes N-dimensionais

Exemplo com Python e NumPy

```
./data/mandrill.tif eh uma imagem com 512 x 512 pixels
./data/mandrill.tif possui 3 canais
O pixel na posicao (11,7) apresenta os valores [ 56 135
132]
```

# Outline

Introdução à OpenCV

Operações com matrizes

Cor

Operações básicas em processamento de imagens

Transformações de imagem

Histogramas

# Diferença absoluta entre duas matrizes

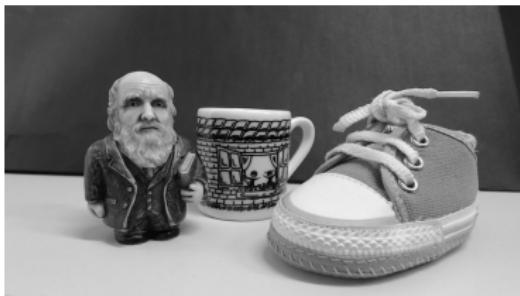
C++

```
cv::absdiff(image1, image2, diff)
```

Python

```
diff = cv2.absdiff(image1, image2)
```

$I_1$



$I_2$



$I_d$



$$I_d(i, j) = |I_1(i, j) - I_2(i, j)|$$

# Média e desvio padrão

C++      `cv::meanStdDev(image, mean, stddev)`  
Python    `mean, stddev = cv2.meanStdDev(image)`



$\mu_r$	88.0964
$\mu_g$	81.7431
$\mu_b$	26.6025
$\sigma_r$	87.1673
$\sigma_g$	74.0328
$\sigma_b$	27.7083

- ▶ Computa a média e o desvio padrão para **cada canal**

# Comparação entre imagens

C++      `cv::compare(image1, image2, result, cmpop)`

Python    `result = cv2.compare(image1, image2, cmpop)`

excompare.py

```
5     plant = cv2.imread(sys.argv[1])
6     blue, green, red = cv2.split(plant)
7     disease = cv2.compare(green, red, cv2.CMP_LT)
```

- ▶ A operação `split` separa os canais da imagem
- ▶ A Linha 7 compara os canais verde e vermelho:

$$\text{disease}(i,j) = \begin{cases} 255 & \text{se } \text{green}(i,j) < \text{red}(i,j) \\ 0 & \text{caso contrário} \end{cases}$$

CMP_EQ	=	CMP_GT	>
CMP_GE	≥	CMP_LT	<
CMP_LE	≤	CMP_NE	≠

# Comparação entre imagens

C++      cv::compare(image1, image2, result, cmpop)

Python    result = cv2.compare(image1, image2, cmpop)

plant

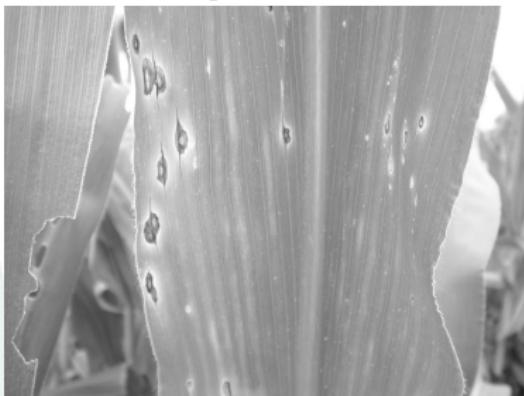


# Comparação entre imagens

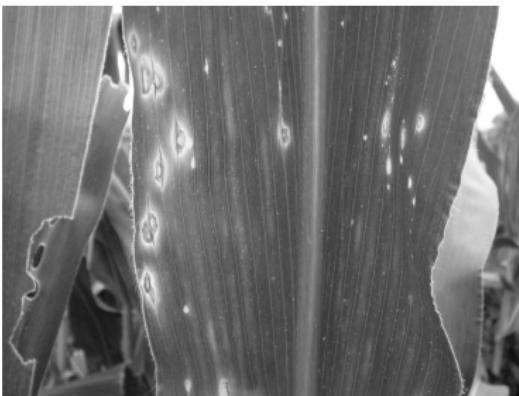
C++      `cv::compare(image1, image2, result, cmpop)`

Python    `result = cv2.compare(image1, image2, cmpop)`

green



red



# Comparação entre imagens

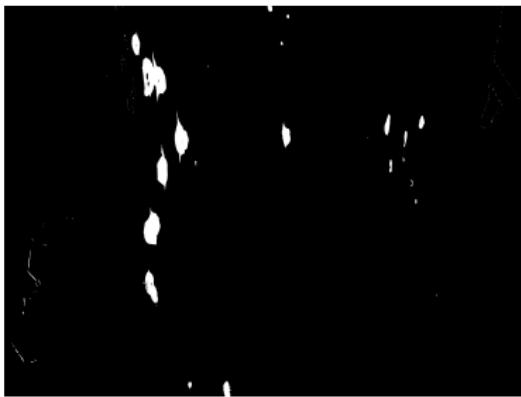
C++      `cv::compare(image1, image2, result, cmpop)`

Python    `result = cv2.compare(image1, image2, cmpop)`

plant



disease



# Conversão e transformação linear

C++      `M.convertTo(N, CV_8UC1, 2, 32)`

Python    Usar NumPy

exconvert.cpp

```
13 Mat M = imread(file_path, CV_LOAD_IMAGE_GRAYSCALE);  
14 Mat N = Mat(M.rows, M.cols, CV_8UC1);  
15  
16 // N(i,j) = 2 * M(i,j) + 32 com saturacao para uint8  
17 M.convertTo(N, CV_8UC1, 2, 32);
```

- ▶ Realiza uma **transformação linear** no valor do pixel:

$$N(i,j) = \text{satura}_{\text{tipo}}(a \cdot M(i,j) + b)$$

- ▶ Também **satura** e converte para o tipo de destino:

$$\text{satura}_{\text{uint8}}(x) = \langle \text{uint8} \rangle \min(\max(\text{round}(x), 0), 255)$$

# Conversão e transformação linear

C++

M.convertTo(N, CV\_8UC1, 2, 32)

Python

Usar NumPy

*M*



*N*



# Matrizes e álgebra linear

Explorando espaços vetoriais com OpenCV

	calcCovarMatriz (M)	matriz de covariância
	determinant (M)	determinante de uma matriz quadrada
C++	M.cross (N)	produto vetorial, $M \times N$
Python	cross (M, N)	
C++	M.dot	produto escalar, $M \cdot N$
Python	dot (M, N)	
	eigen	auto-vetores e auto-valores
C++	M.inv ()	inversa
Python	inv (M)	
C++	M.t ()	transposta
Python	M.transpose	
	solve	soluciona um sistema linear

# Matrizes e álgebra linear

Explorando espaços vetoriais com OpenCV

C++	SVD::compute	decomposição em valores singulares
Python	SVDecomp	

# Outline

Introdução à OpenCV

Operações com matrizes

Cor

Operações básicas em processamento de imagens

Transformações de imagem

Histogramas

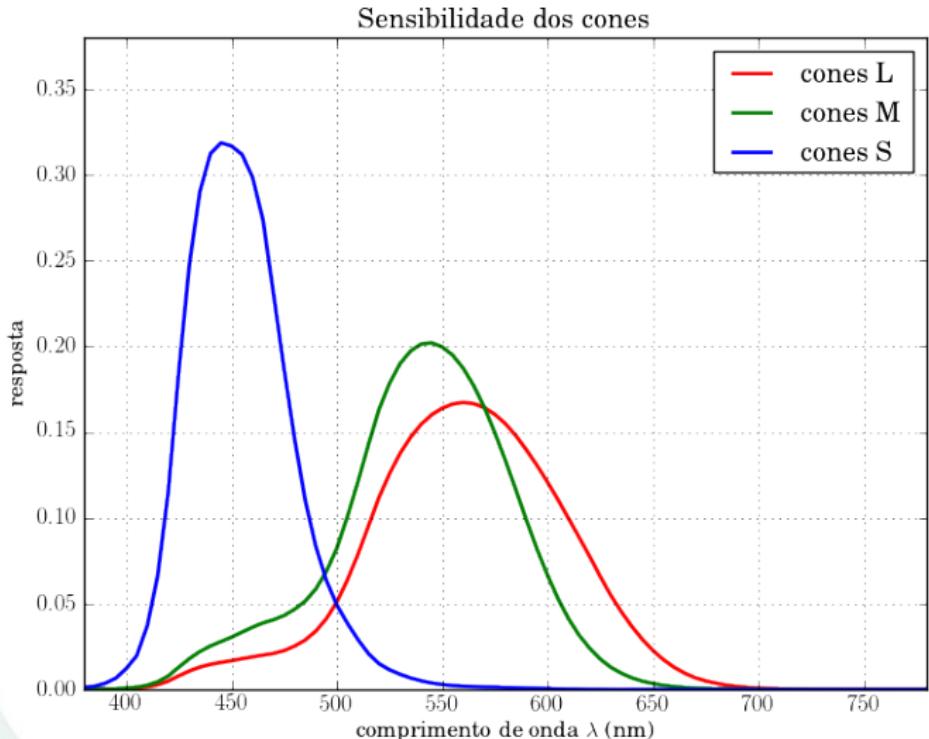
# Cor

Por que podemos representar cores com RGB?

- ▶ Cor é uma resposta sensorial a um *tri-stimulus*
- ▶ **Cones** são as células da retina que respondem a vários comprimentos de onda  $\lambda$
- ▶ Há **três tipos** de cone: S, M e L

# Cor

Por que podemos representar cores com RGB?

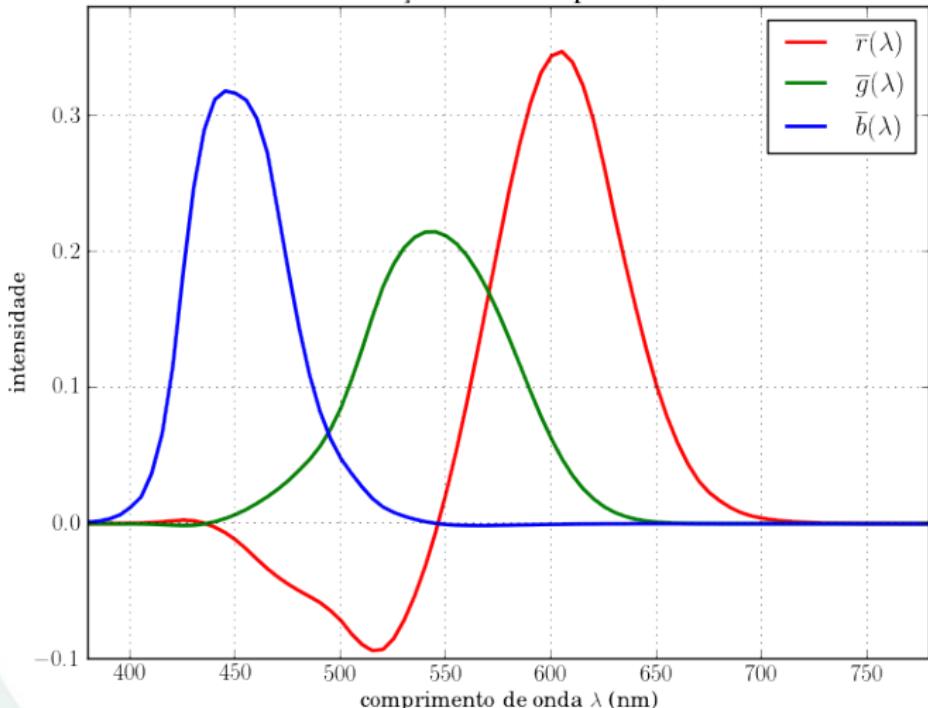


# CIE 1931 RGB

Commission Internationale d'Eclairage

- ▶ Em 1930 a CIE conduziu um experimento para estabelecer a correspondência cor/*tri-stimulus*
- ▶ Três **cores primárias** foram utilizadas:
  - ▶ vermelho (700.0 nm)
  - ▶ verde (546.1 nm)
  - ▶ azul (435.8 nm)
- ▶ Dada um cor de referência, o observador deveria ajustar a intensidade das três cores primárias até produzir a mesma resposta
- ▶ O resultado foram as **funções de correspondência de cor**
- ▶ As funções foram utilizadas para definir o espaço de cor RGB

CIE 1931 - Funções de correspondência de cores



## Espaço perceptualmente uniforme

É um espaço de cor no qual alterações de valor de mesma magnitude produzem percepções similares de mudança no sistema visual humano.

- ▶ CIE L\*a\*b\* é um espaço de cor **perceptualmente uniforme**
- ▶ Separa luminosidade e cromaticidade (L vs. a\*b\*)
- ▶ Obtido a partir do espaço RGB por uma transformação não-linear.

Imagen colorida



L



Imagen colorida



a\*

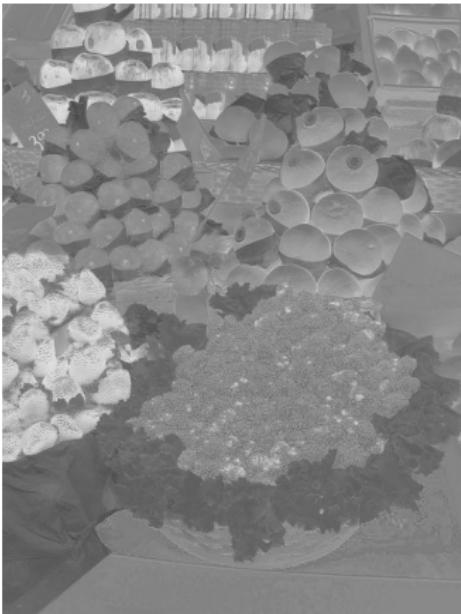
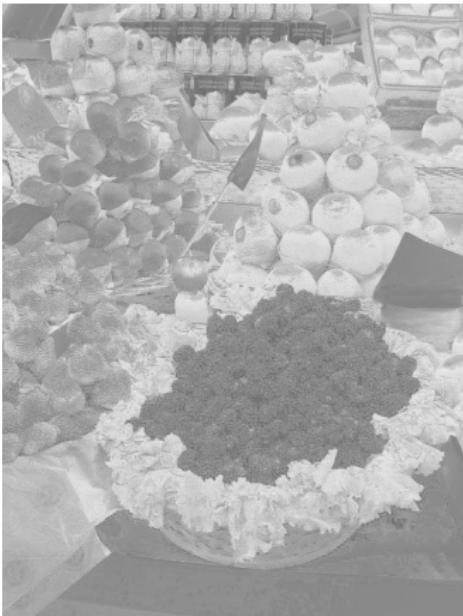


Imagen colorida



b\*



- ▶ O espaço RGB pode ser visto como um **cubo**
- ▶ HSV é um espaço de cor obtido por uma **transformação não-linear** do espaço RGB
- ▶ O cubo RGB é transformado em um **cilindro HSV**
  - ▶ H - matiz (*hue*) é um **ângulo**
  - ▶ S - saturação é o **raio**
  - ▶ V - valor é a **altura**

## O cilindro HSV como forma de seleção de cores

### Matiz

#### Primárias

vermelho	$H = 0^\circ$
verde	$H = 120^\circ$
azul	$H = 240^\circ$

#### Secundárias

amarelo	$H = 60^\circ$
ciano	$H = 180^\circ$
magenta	$H = 300^\circ$

- ▶ Saturação  $S = 0$ 
  - ▶ eixo cinza, mistura equitativa das componentes R, G e B
- ▶ Saturação  $S = 100$ 
  - ▶ máxima pureza da cor

### Valor

preto	$V = 0$
branco	$V = 100$
	$S = 0$
cor intensa	$V = 100$
	$S = 100$

# HSV e seleção de cores

Exemplo com o aplicativo GIMP

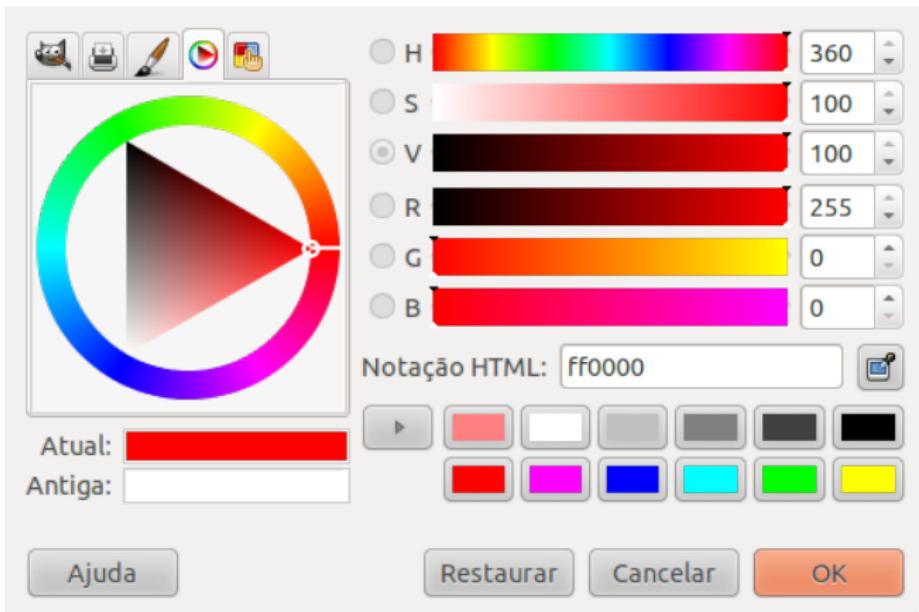


Imagen colorida



H (matiz)



Imagen colorida



H (S e V constantes)

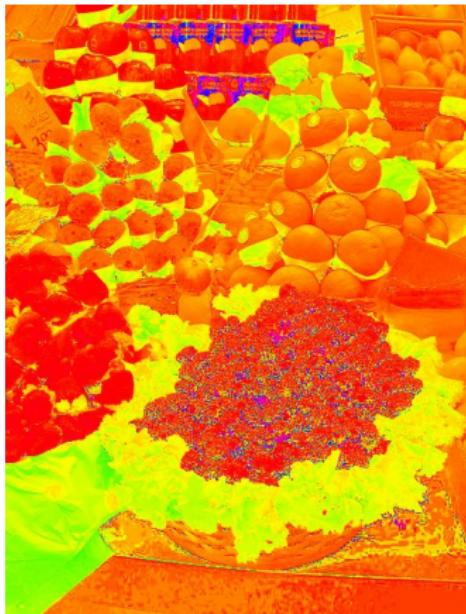


Imagen colorida



S (saturação)



Imagen colorida



V (valor)



# Conversão entre espaços de cor com OpenCV

C++      `cv::cvtColor(image1, image2, code)`

Python    `image2 = cv2.cvtColor(image1, code)`

- ▶ OpenCV usa como padrão RGB, ordenado em BGR
- ▶ Código COLOR\_<de>2<para> especifica a conversão
- ▶ Exemplos:
  - ▶ COLOR\_BGR2HSV para converter de RGB para HSV
  - ▶ COLOR\_Lab2LBGR para converter de CIE La\*b\* para RGB
- ▶ Diversos espaços de cor disponíveis: CIE XYZ, YCrCb, Bayer, HLS, Lu\*v\*, tons de cinza
- ▶ Documentação da OpenCV detalha
  - ▶ as **fórmulas de conversão**
  - ▶ as representações utilizadas em 8, 16 e 32 bits
- ▶ Visualização deve ser feita em RGB ou tons de cinza (displays operam em RGB)

# Exemplo

Encontrar pixels apresentando verde intenso

excolorcvt.py

```
5 img_bgr = cv2.imread(sys.argv[1], cv2.CV_LOAD_IMAGE_COLOR)
6 # Devido aos 8 bits da image, OpenCV aplica H <- H/2
7 img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
8 [h, s, v] = cv2.split(img_hsv)
```

- ▶ A Linha 6 realiza a conversão de RGB para HSV
- ▶ A Linha 7 separa a imagem em seus 3 canais
- ▶ OpenCV aplica

$$h'(i,j) \leftarrow h(i,j)/2$$

obtendo um intervalo  $[0^\circ, 180^\circ]$  para imagens de 8 bits

# Exemplo

Encontrar pixels apresentando verde intenso

excolorcvt.py

```
9     diff = cv2.convertScaleAbs(h, h, 2, -120)
10    val, greenish = cv2.threshold(diff, 60, 255, cv2.
THRESH_BINARY_INV)
```

- ▶ A Linha 9 converte ao intervalo  $[0^\circ, 360^\circ]$  e subtrai o valor do verde primário ( $H = 120^\circ$ )
- ▶ A Linha 10 aplica um **limiar**:

$$\text{greenish}(i, j) = \begin{cases} 255 & \text{se } \text{diff}(i, j) < 60 \\ 0 & \text{caso contrário} \end{cases}$$

# Exemplo

Encontrar pixels apresentando verde intenso

excolorcvt.py

```
11     val, saturated = cv2.threshold(s, 192, 255, cv2.  
12                                     THRESH_BINARY)  
greens = greenish & saturated
```

- ▶ A Linha 11 aplica um limiar à saturação:

$$\text{saturated}(i, j) = \begin{cases} 255 & \text{se } s(i, j) > 192 \\ 0 & \text{caso contrário} \end{cases}$$

- ▶ Linha 12 aplica um **operador lógico** para selecionar pixels verdes e saturados

# Exemplo

Encontrar pixels apresentando verde intenso

Imagen colorida



Verdes intensos



# Outline

Introdução à OpenCV

Operações com matrizes

Cor

Operações básicas em processamento de imagens

Transformações de imagem

Histogramas

# Suavização

*Smoothing ou blurring*

- ▶ Intuição: ato de **borrar** uma imagem
- ▶ Diversas aplicações:
  - ▶ remoção de ruído,
  - ▶ pré-processamento em análise multi-resolução (pirâmides),
  - ▶ pré-processamento em segmentação de imagens.



# Filtros lineares

- ▶ Combinações ponderadas de pixels em uma vizinhança

$$g(i, j) = \sum_{k,l} f(i - k, j - l)h(k, l)$$

- ▶ Operação também chamada de **convolução**:

$$g = f * h$$

- ▶ Pesos são determinados pelo **núcleo de convolução**  $h$
- ▶ Filtros lineares podem realizar diversas operações:
  - ▶ suavização,
  - ▶ detecção de borda,
  - ▶ auto-reforço.

# Filtros lineares

203	15	124	202	32
33	199	247	183	127
116	120	<b>232</b>	81	112
82	147	108	179	92
236	85	88	212	117

\*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

# Filtros lineares

203	15	124	202	32
33	199	247	183	127
116	120	<b>232</b>	81	112
82	147	108	179	92
236	85	88	212	117

\*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

# Filtros lineares

129	131	172	167	145
105	148	165	152	133
127	140	<b>172</b>	144	132
121	136	136	140	135
139	116	134	138	155

# Filtros lineares

- ▶ Combinações ponderadas de pixels em uma vizinhança

$$g(i, j) = \sum_{k,l} f(i - k, j - l)h(k, l)$$

- ▶ Operação também chamada de **convolução**:

$$g = f * h$$

- ▶ Pesos são determinados pelo **núcleo de convolução**  $h$
- ▶ Filtros lineares podem realizar diversas operações:
  - ▶ suavização,
  - ▶ detecção de borda,
  - ▶ auto-reforço.

- ▶ Combinações ponderadas de pixels em uma vizinhança

$$g(i, j) = \sum_{k,l} f(i - k, j - l)h(k, l)$$

- ▶ Operação também chamada de **convolução**:

$$g = f * h$$

- ▶ Pesos são determinados pelo **núcleo de convolução**  $h$
- ▶ Filtros lineares podem realizar diversas operações:
  - ▶ suavização,
  - ▶ detecção de borda,
  - ▶ auto-reforço.

- ▶ Combinações ponderadas de pixels em uma vizinhança

$$g(i, j) = \sum_{k,l} f(i - k, j - l)h(k, l)$$

- ▶ Operação também chamada de **convolução**:

$$g = f * h$$

- ▶ Pesos são determinados pelo **núcleo de convolução**  $h$
- ▶ Filtros lineares podem realizar diversas operações:
  - ▶ suavização,
  - ▶ detecção de borda,
  - ▶ auto-reforço.

# Suavização linear: filtro caixa (*box filter*)

C++      cv::blur(src, dst, ksize, anchor=Point(-1,-1), borderType=BORDER\_DEFAULT )  
Python    cv2.blur(src, ksize[, dst[, anchor[, borderType]]])

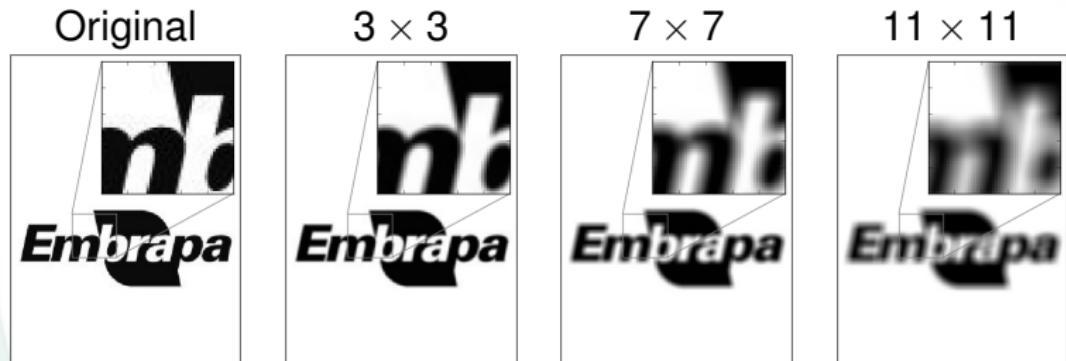
- ▶ Kernel produz a **média** dos pixels na vizinhança:

$$h_{3 \times 3} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$h_{M \times N} = \frac{1}{M \cdot N} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 \\ \vdots & & & & & \\ 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 \end{bmatrix}$$

# Suavização linear: filtro caixa (*box filter*)

C++      cv2::blur(src, dst, ksize, anchor=Point(-1,-1), borderType=BORDER\_DEFAULT )  
Python    cv2.blur(src, ksize[, dst[, anchor[, borderType]]])



# Suavização linear: filtro caixa (*box filter*)

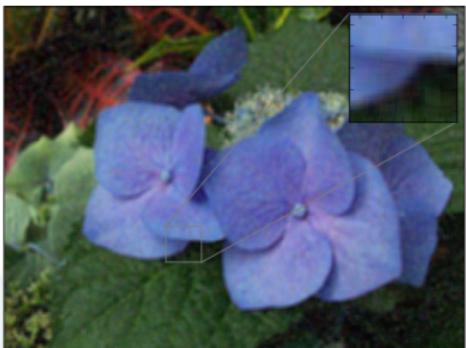
Original



$3 \times 3$



$7 \times 7$



$11 \times 11$

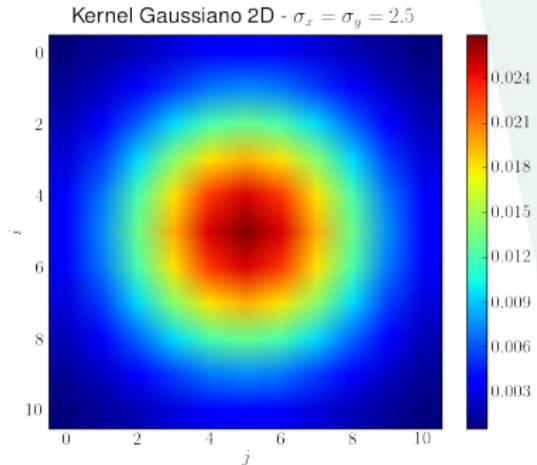
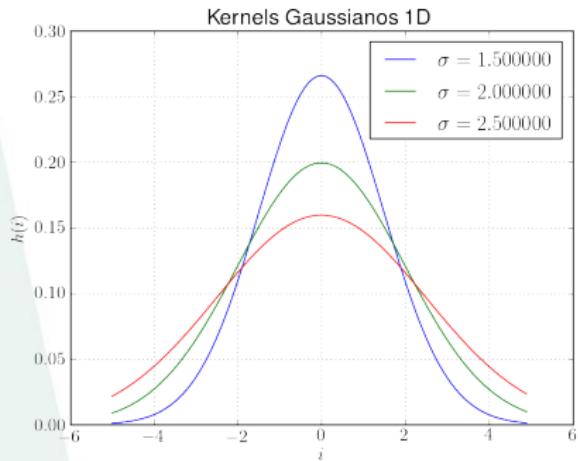


# Suavização linear: filtro Gaussiano

C++      `cv2::GaussianBlur(src, dst, ksize,  $\sigma_x$ ,  $\sigma_y$ , borderType=BORDER_DEFAULT )`

Python    `dst = cv2.GaussianBlur(src, ksize,  $\sigma_x$ [, dst[,  $\sigma_y$ [, borderType]]])`

## ► Convolução com um kernel Gaussiano

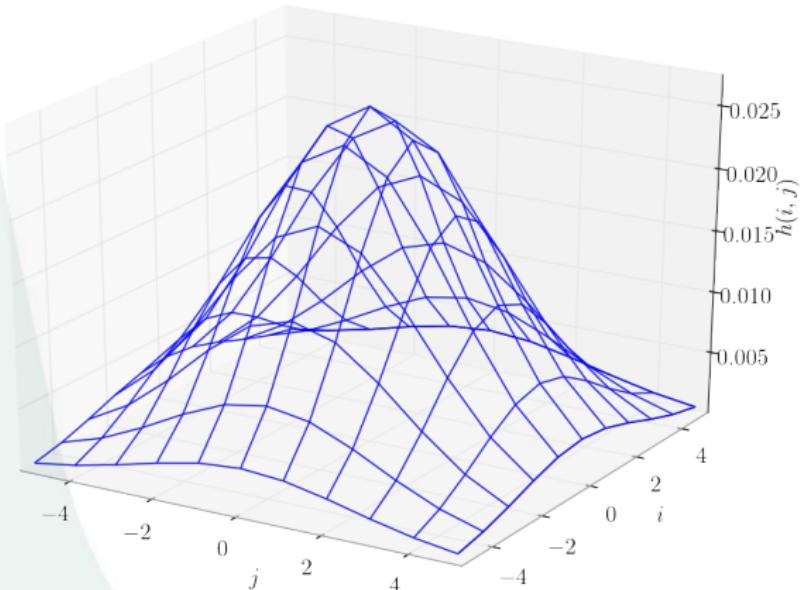


# Suavização linear: filtro Gaussiano

C++      `cv2::GaussianBlur(src, dst, ksize,  $\sigma_x$ ,  $\sigma_y$ , borderType=BORDER_DEFAULT )`

Python    `dst = cv2.GaussianBlur(src, ksize,  $\sigma_x$ [, dst[,  $\sigma_y$ [, borderType]]])`

- ▶ Convolução com um **kernel Gaussiano**



# Suavização linear: filtro Gaussiano

C++      cv2::GaussianBlur(src, dst, ksize,  $\sigma_x$ ,  $\sigma_y$ , borderType=BORDER\_DEFAULT )  
Python    dst = cv2.GaussianBlur(src, ksize,  $\sigma_x$ [, dst[,  $\sigma_y$ [, borderType]]])

- ▶ Convolução com um **kernel Gaussiano**

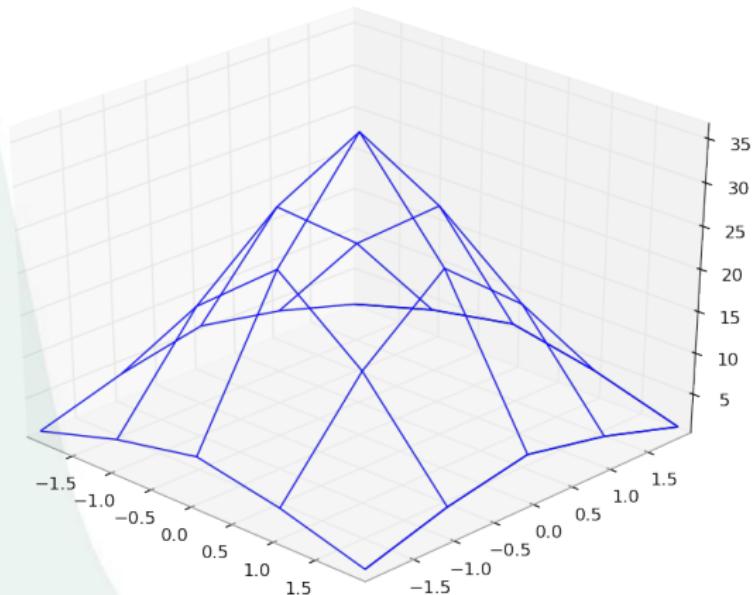
$$h = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

# Suavização linear: filtro Gaussiano

C++      `cv2::GaussianBlur(src, dst, ksize,  $\sigma_x$ ,  $\sigma_y$ , borderType=BORDER_DEFAULT )`

Python    `dst = cv2.GaussianBlur(src, ksize,  $\sigma_x$ [, dst[,  $\sigma_y$ [, borderType]]])`

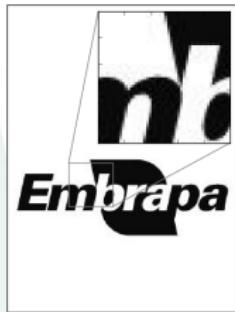
- ▶ Convolução com um **kernel Gaussiano**



# Suavização linear: filtro Gaussiano

C++      cv2::GaussianBlur(src, dst, ksize,  $\sigma_x$ ,  $\sigma_y$ , borderType=BORDER\_DEFAULT )  
Python    dst = cv2.GaussianBlur(src, ksize,  $\sigma_x$ [, dst[,  $\sigma_y$ [, borderType]]])

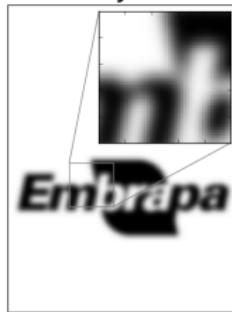
Original



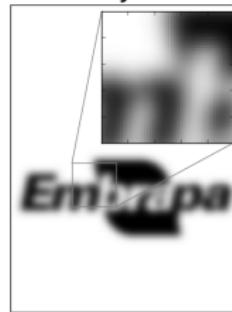
$\sigma_x = \sigma_y = 1.5$



$\sigma_x = \sigma_y = 3.0$



$\sigma_x = \sigma_y = 4.5$

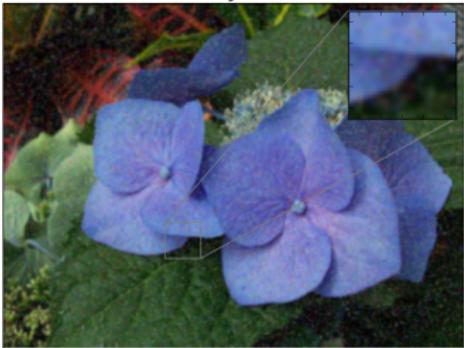


# Suavização linear: filtro Gaussiano

Original



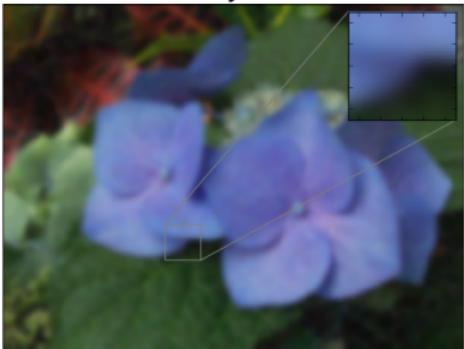
$\sigma_x = \sigma_y = 1.5$



$\sigma_x = \sigma_y = 3.0$



$\sigma_x = \sigma_y = 4.5$



# Suavização não-linear: filtro mediana

C++      `cv2::medianBlur(src, dst, ksize)`

Python    `dst = cv2.medianBlur(src, ksize)`

## Mediana

Mediana é o valor que ocupa a posição central da série de observações, quando estas estão ordenadas. É o valor que separa a metade superior da metade inferior no conjunto de valores.

- ▶ O filtro toma a **mediana** na vizinhança do pixel
- ▶ Não é um filtro linear

# Suavização não-linear: filtro mediana

C++      cv2::medianBlur(src, dst, ksize)

Python    dst = cv2.medianBlur(src, ksize)

203	15	124	202	32
33	199	247	183	127
116	120	<b>232</b>	81	112
82	147	108	179	92
236	85	88	212	117

# Suavização não-linear: filtro mediana

C++      cv2::medianBlur(src, dst, ksize)

Python    dst = cv2.medianBlur(src, ksize)

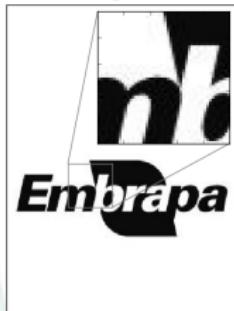
15	32	33	81	82
85	88	92	108	112
116	117	<b>120</b>	124	127
147	179	183	199	202
203	212	232	236	247

# Suavização não-linear: filtro mediana

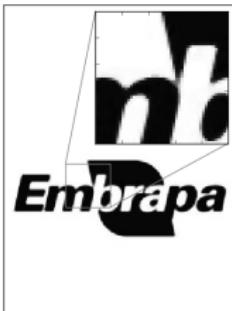
C++      `cv2::medianBlur(src, dst, ksize)`

Python    `dst = cv2.medianBlur(src, ksize)`

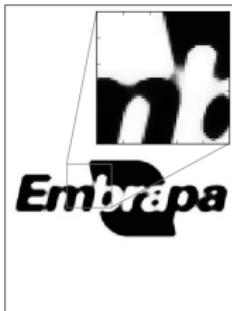
Original



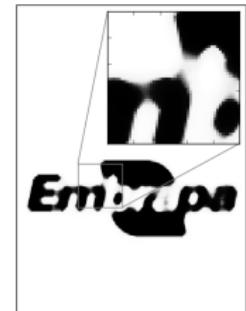
$3 \times 3$



$7 \times 7$



$11 \times 11$



# Suavização não-linear: filtro mediana

Original



$3 \times 3$



$7 \times 7$



$11 \times 11$



# Suavização não-linear: filtro bilateral

C++      `cv2::bilateralFilter(src, dst, d, σcor, σespaço, borderType=BORDER_DEFAULT )`

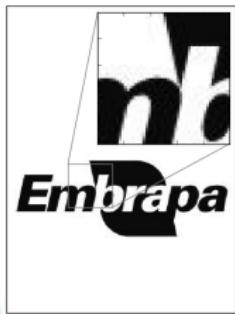
Python    `cv2.bilateralFilter(src, d, σcor, σespaço, borderType)`

- ▶ Alguns filtros borram as **bordas** da imagem
- ▶ O filtro bilateral é um filtro que preserva bordas
- ▶ O peso de cada vizinho apresenta dois componentes:
  - ▶ distância - quanto maior a distância ao vizinho, menor seu peso (similar ao filtro Gaussiano)
  - ▶ cor - quanto maior a diferença de cor do vizinho, menor seu peso
- ▶ O filtro bilateral não é linear

# Suavização não-linear: filtro bilateral

$\sigma_{cor} = 32$

Original



$\sigma_{espaço} = 3$



$\sigma_{espaço} = 7$



$\sigma_{espaço} = 11$



# Suavização não-linear: filtro bilateral

$\sigma_{cor} = 256$

Original



$\sigma_{espaço} = 3$



$\sigma_{espaço} = 7$



$\sigma_{espaço} = 11$



# Suavização não-linear: filtro bilateral

Fotografia com ruído (ISO 3200)

Original



$\sigma_{\text{cor}} = 16, \sigma_{\text{espaço}} = 3$



# Exercício

## Exercício 1

Carregue uma imagem com texturas (*Mandrill* ou *Armadura* por exemplo). Suavize a imagem de diversas formas utilizando GaussianBlur:

1. Use *kernels* de tamanhos variados como  $3 \times 3$ ,  $7 \times 7$  e  $11 \times 11$ .
2. Aplicar suavização **duas vezes** na mesma imagem com um *kernel*  $5 \times 5$  ou **uma vez** com um *kernel*  $11 \times 11$  produz **quase** idênticos? Por quê?

## Exercício 2

Crie uma imagem “zerada” de  $31 \times 31$  pixels. Faça com que o pixel central  $(15, 15)$  assuma o valor 255.

1. Suavize a imagem com um filtro Gaussiano com *kernel*  $5 \times 5$ . O que ocorre?
2. Repita, agora com um *kernel*  $11 \times 11$ .

# Operadores morfológicos

- ▶ **Morfologia matemática** é uma técnica para análise de estruturas espaciais:
  - ▶ teoria de conjuntos,
  - ▶ álgebra de reticulados.
- ▶ Técnica poderosa para análise de imagens:
  - ▶ filtragem,
  - ▶ segmentação,
  - ▶ medição.
- ▶ Emprega operadores não-lineares que trabalham com **máximos e mínimos** de uma vizinhança.
- ▶ Esta vizinhança é determinada por um kernel chamado **elemento estruturante**.

# Operador de máximo local: dilatação

$\delta_E(I)$

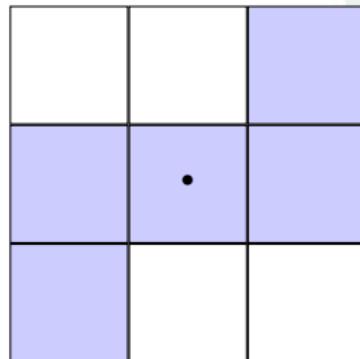
1. Mova o elemento estruturante  $E$  ao longo da imagem  $I$ 
  - ▶ o elemento tem um ponto âncora
2. Encontre o valor **máximo** dentro da estrutura
3. Atribua ao pixel sob o ponto âncora este valor máximo

# Operador de máximo local: dilatação

C++      cv2::dilate(src, dst, element[, anchor[, iterations[, borderType[, borderValue]]]]])

Python    dst = cv2.dilate(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])

203	15	124	202	32
33	199	247	183	127
116	120	<b>132</b>	81	112
82	147	108	179	92
236	85	88	212	117

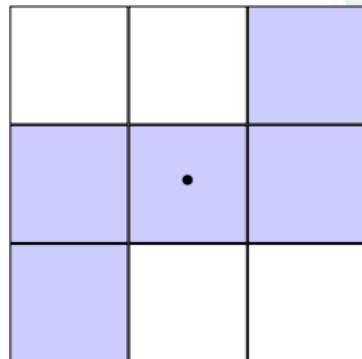


# Operador de máximo local: dilatação

C++      cv2::dilate(src, dst, element[, anchor[, iterations[, borderType[, borderValue]]]]])

Python    dst = cv2.dilate(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])

203	15	124	202	32
33	199	247	183	127
116	120	<b>132</b>	81	112
82	147	108	179	92
236	85	88	212	117



# Operador de máximo local: dilatação

C++      cv2::dilate(src, dst, element[, anchor[, iterations[, borderType[, borderValue]]]]))

Python    dst = cv2.dilate(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]))

203	203	202	247	202
199	247	247	247	183
199	247	<b>183</b>	132	179
236	236	179	179	212
236	236	212	212	212

# Operador de máximo local: dilatação

Original  $I$

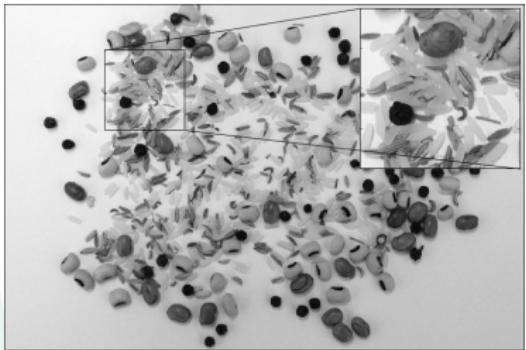


Dilatada  $\delta_E(I)$

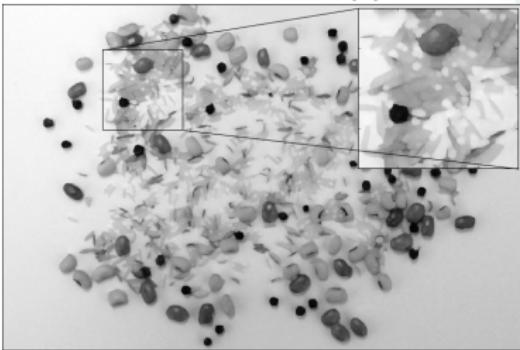


# Operador de máximo local: dilatação

Original  $I$



Dilatada  $\delta_E(I)$



# Operador de mínimo local: erosão

$\varepsilon_E(I)$

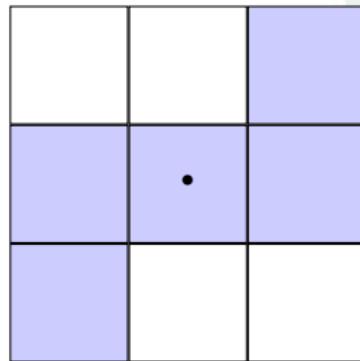
1. Mova o elemento estruturante  $E$  ao longo da imagem  $I$ 
  - ▶ o elemento tem um ponto âncora
2. Encontre o valor **mínimo** dentro da estrutura
3. Atribua ao pixel sob o ponto âncora este valor mínimo

# Operador de mínimo local: erosão

C++      cv2::erode(src, dst, element[, anchor[, iterations[, borderType[, borderValue]]]]])

Python    dst = cv2.erode(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]))

203	15	124	202	32
33	199	247	183	127
116	120	<b>132</b>	81	112
82	147	108	179	92
236	85	88	212	117

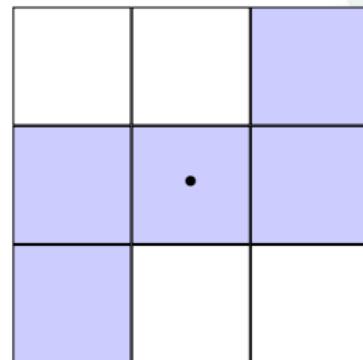


# Operador de mínimo local: erosão

C++      cv2::erode(src, dst, element[, anchor[, iterations[, borderType[, borderValue]]]]))

Python    dst = cv2.erode(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]))

203	15	124	202	32
33	199	247	183	127
116	120	<b>132</b>	81	112
82	147	108	179	92
236	85	88	212	117



# Operador de mínimo local: erosão

C++      cv2::erode(src, dst, element[, anchor[, iterations[, borderType[, borderValue]]]]))

Python    dst = cv2.erode(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]]))

15	15	15	32	32
15	33	120	32	32
82	82	<b>81</b>	81	81
82	82	81	88	92
85	85	85	88	92

# Operador de mínimo local: erosão

Original  $I$

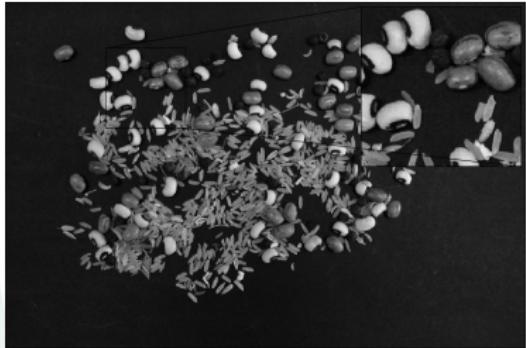


Erodida  $\varepsilon_E(I)$

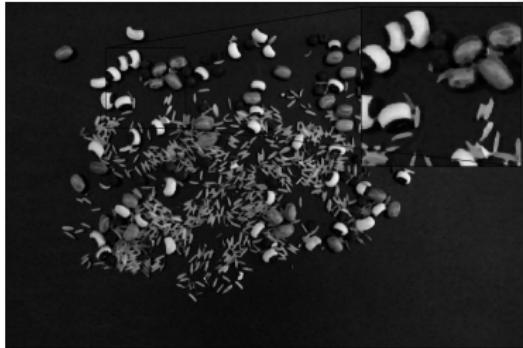


# Operador de mínimo local: erosão

Original  $I$



Erodida  $\varepsilon_E(I)$



# Abertura (erosão seguida por dilatação)

C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_OPEN, element[, anchor[, (...)]])
```

Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_OPEN, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



$\gamma_E(I) = \delta_E(\varepsilon_E(I))$



# Abertura (erosão seguida por dilatação)

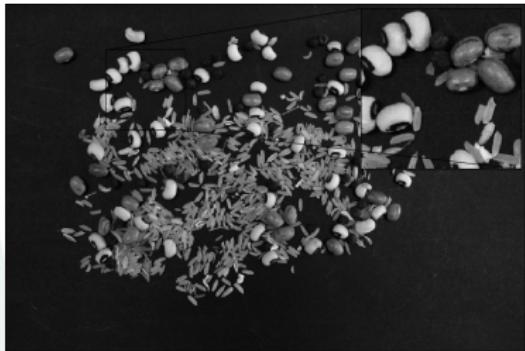
C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_OPEN, element[, anchor[, (...)]])
```

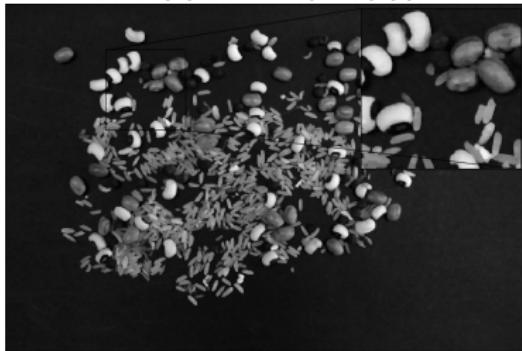
Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_OPEN, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



$$\gamma_E(I) = \delta_E(\varepsilon_E(I))$$



# Fechamento (dilatação seguida por erosão)

C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_CLOSE, element[, anchor[, (...)]])
```

Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_CLOSE, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



$\phi_E(I) = \varepsilon_E(\delta_E(I))$



# Fechamento (dilatação seguida por erosão)

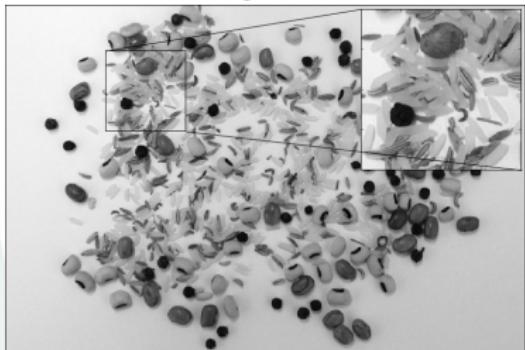
C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_CLOSE, element[, anchor[, (...)]])
```

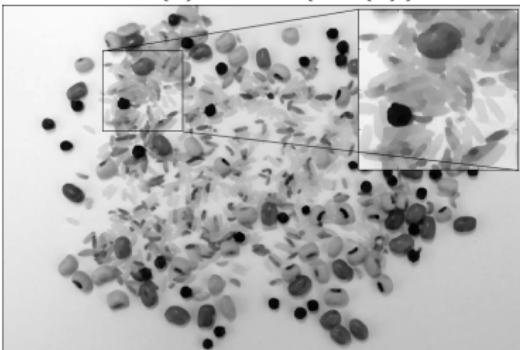
Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_CLOSE, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



$\phi_E(I) = \varepsilon_E(\delta_E(I))$



# Gradiente morfológico (dilatação - erosão)

C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_GRADIENT, element[, anchor[, (...)]])
```

Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_GRADIENT, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



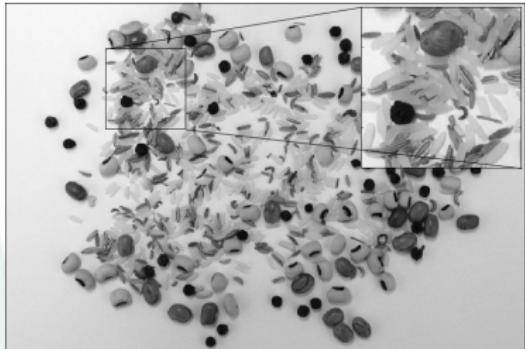
$$\rho_E(I) = \delta_E(I) - \varepsilon_E(I)$$



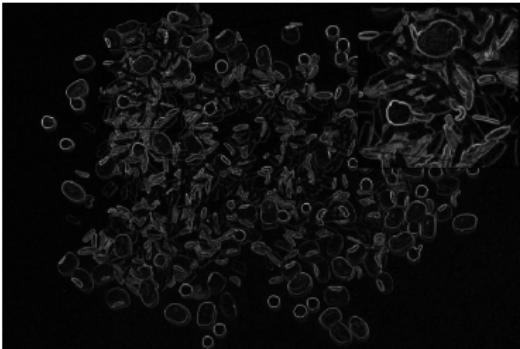
# Gradiente morfológico (dilatação - erosão)

C++      `cv2::morphologyEx(src, dst, cv2.MORPH_GRADIENT, element[, anchor[, (...)]])`  
Python    `dst = cv2.morphologyEx(src, cv2.MORPH_GRADIENT, kernel[, dst[, anchor[, (...)]]])`

Original  $I$



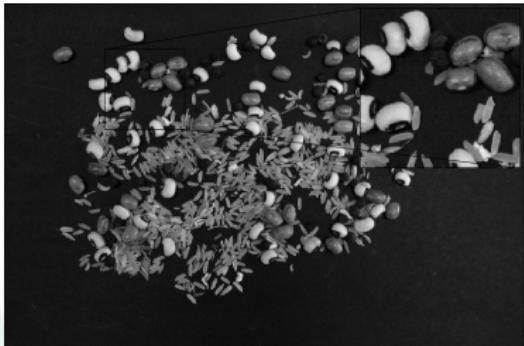
$$\rho_E(I) = \delta_E(I) - \varepsilon_E(I)$$



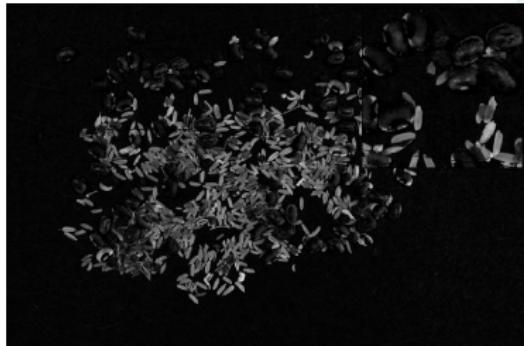
# Top Hat

C++      cv2::morphologyEx(src, dst, cv2.MORPH\_TOPHAT, element[, anchor[, (...)]])  
Python    dst = cv2.morphologyEx(src, cv2.MORPH\_TOPHAT, kernel[, dst[, anchor[, (...)]]])

Original  $I$



$I - \gamma_E(I)$

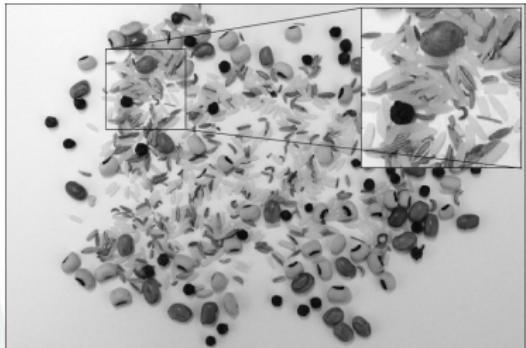


- ▶ Seleciona elementos
  1. menores que o elemento estruturante;
  2. mais claros que a área ao redor

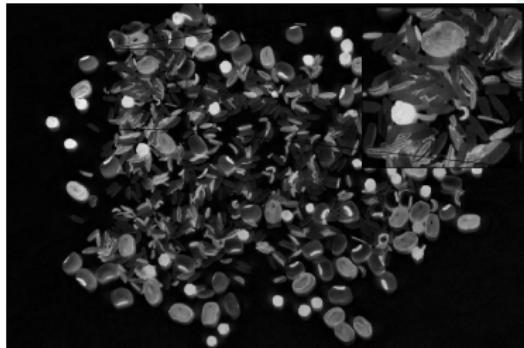
# Black Hat

C++      `cv2::morphologyEx(src, dst, cv2.MORPH_BLACKHAT, element[, anchor[, (...)]])`  
Python    `dst = cv2.morphologyEx(src, cv2.MORPH_BLACKHAT, kernel[, dst[, anchor[, (...)]]])`

Original  $I$



$\phi_E(I) - I(61 \times 61)$



- ▶ Seleciona elementos
  1. menores que o elemento estruturante;
  2. mais escuros que a área ao redor

# Black Hat

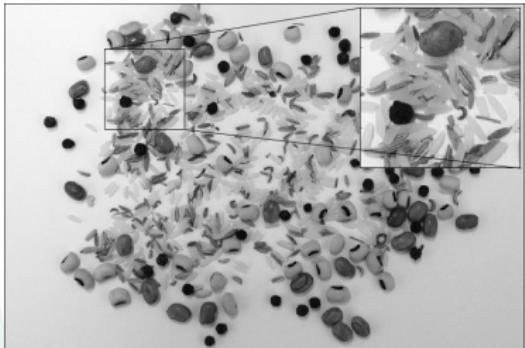
C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_BLACKHAT, element[, anchor[, (...)]])
```

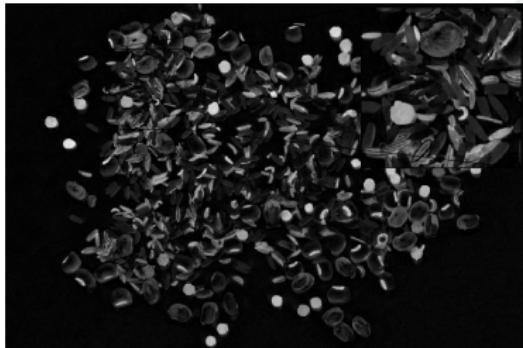
Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_BLACKHAT, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



$\phi_E(I) - I$  ( $31 \times 31$ )



- ▶ Seleciona elementos
  1. menores que o elemento estruturante;
  2. mais escuros que a área ao redor

# Black Hat

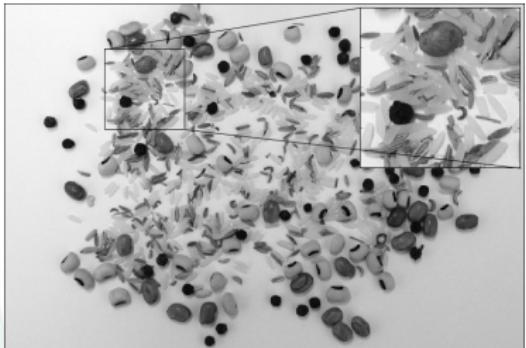
C++

```
cv2::morphologyEx(src, dst, cv2.MORPH_BLACKHAT, element[, anchor[, (...)]])
```

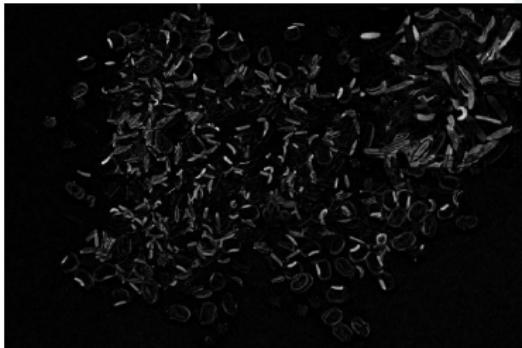
Python

```
dst = cv2.morphologyEx(src, cv2.MORPH_BLACKHAT, kernel[, dst[, anchor[, (...)]]])
```

Original  $I$



$\phi_E(I) - I(11 \times 11)$

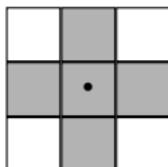
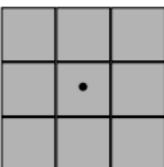


- ▶ Seleciona elementos
  1. menores que o elemento estruturante;
  2. mais escuros que a área ao redor

# Flood Fill

Segmentação de imagens por crescimento de regiões

- ▶ Vizinhos de um pixel

 $N_4$  $N_8$ 

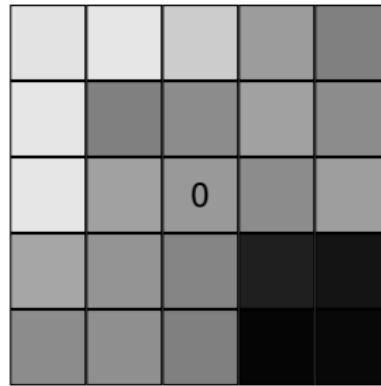
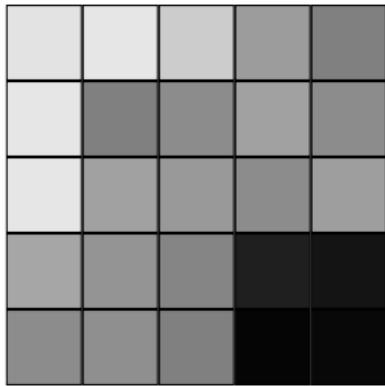
- ▶ O algoritmo inicia com um pixel **semente**  $p_0$
- ▶ Considere uma **máscara**  $M$  e uma **fila**  $F = \langle p_0 \rangle$
- ▶ A cada iteração:
  1. Remova o primeiro pixel da fila,  $p_i$
  2. Se  $p_i$  apresentar cor similar ao pixels de  $M$ :
    - 2.1 Adicione  $p_i$  a  $M$
    - 2.2 Adicione os vizinhos de  $p_i$  à fila  $F$

# Flood Fill

Com vizinhança  $N_4$

$$M = \{\}$$

$$F = \langle 0 \rangle$$

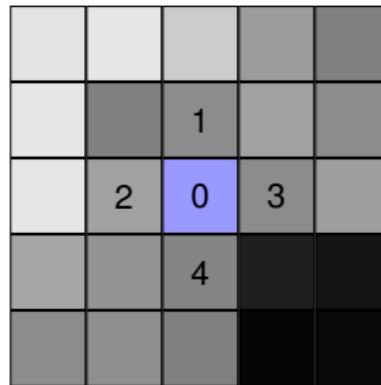
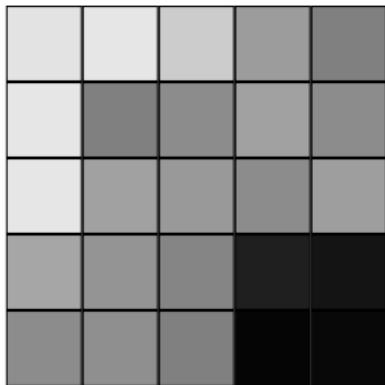


# Flood Fill

Com vizinhança  $N_4$

$$M = \{0\}$$

$$F = \langle 1, 2, 3, 4 \rangle$$

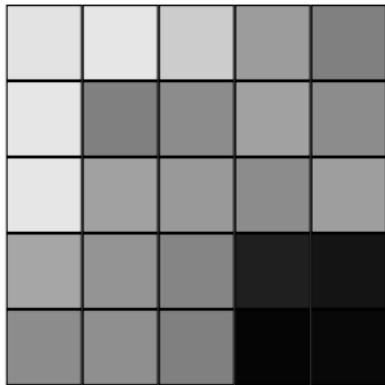


# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1\}$$

$$F = \langle 2, 3, 4, 5, 6, 7 \rangle$$



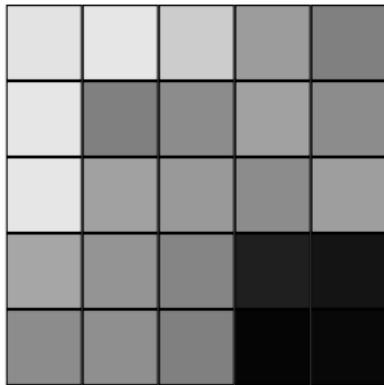
		5		
	6	1	7	
	2	0	3	
		4		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2\}$$

$$F = \langle 3, 4, 5, 6, 7, 8, 9 \rangle$$



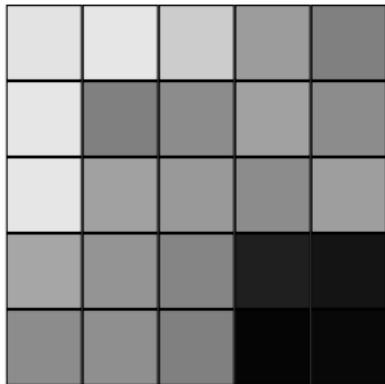
		5		
	6	1	7	
8	2	0	3	
	9	4		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3\}$$

$$F = \langle 4, 5, 6, 7, 8, 9, 10, 11 \rangle$$



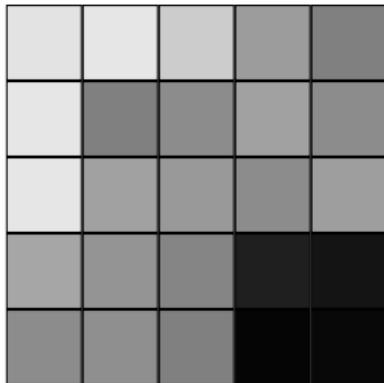
		5		
	6	1	7	
8	2	0	3	10
	9	4	11	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4\}$$

$$F = \langle 5, 6, 7, 8, 9, 10, 11, 12 \rangle$$



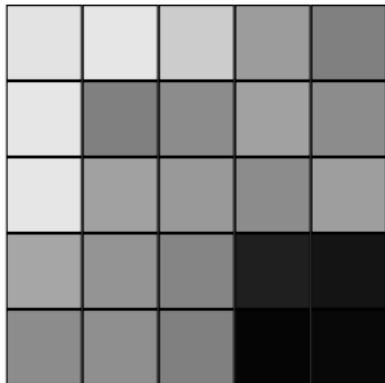
		5		
	6	1	7	
8	2	0	3	10
	9	4	11	
		12		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4\}$$

$$F = \langle 6, 7, 8, 9, 10, 11, 12 \rangle$$



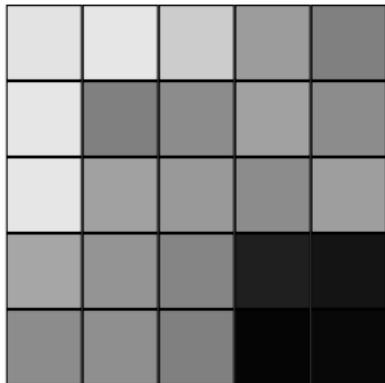
		5		
	6	1	7	
8	2	0	3	10
	9	4	11	
		12		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6\}$$

$$F = \langle 7, 8, 9, 10, 11, 12, 13, 14 \rangle$$



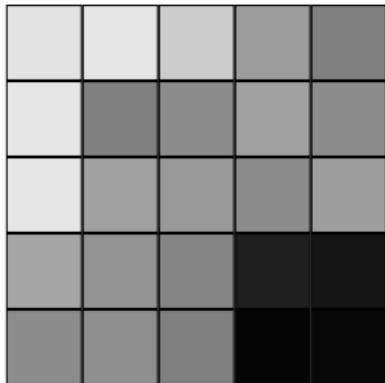
	13	5		
14	6	1	7	
8	2	0	3	10
	9	4	11	
		12		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7\}$$

$$F = \langle 8, 9, 10, 11, 12, 13, 14, 15, 16 \rangle$$



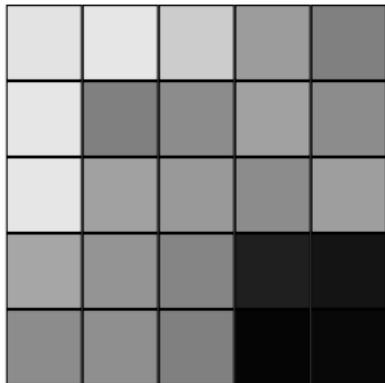
	13	5	15	
14	6	1	7	16
8	2	0	3	10
	9	4	11	
		12		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7\}$$

$$F = \langle 9, 10, 11, 12, 13, 14, 15, 16 \rangle$$



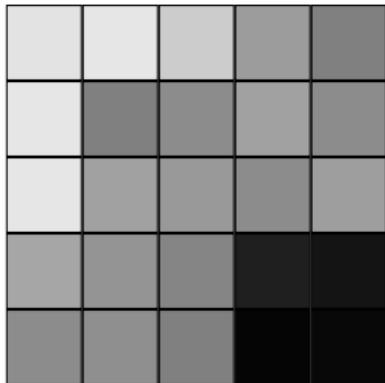
	13	5	15	
14	6	1	7	16
8	2	0	3	10
	9	4	11	
	12			

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9\}$$

$$F = \langle 10, 11, 12, 13, 14, 15, 16, 17, 18 \rangle$$



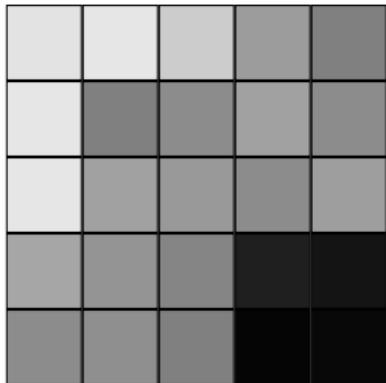
	13	5	15	
14	6	1	7	16
8	2	0	3	10
17	9	4	11	
	18	12		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10\}$$

$$F = \langle 11, 12, 13, 14, 15, 16, 17, 18, 19 \rangle$$



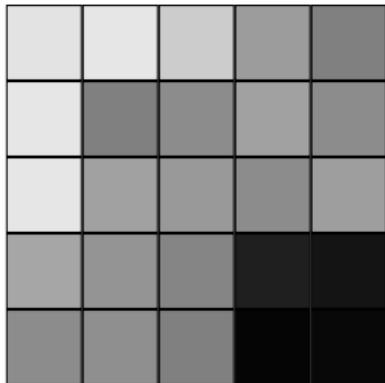
	13	5	15	
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
18	12			

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10\}$$

$$F = \langle 12, 13, 14, 15, 16, 17, 18, 19 \rangle$$



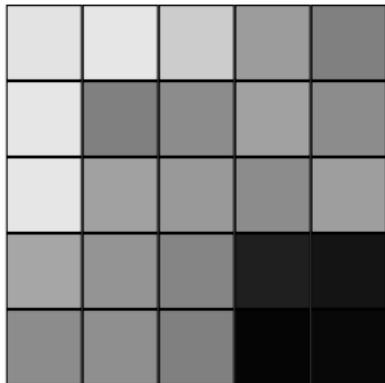
	13	5	15	
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
18	12			

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12\}$$

$$F = \langle 13, 14, 15, 16, 17, 18, 19, 20 \rangle$$



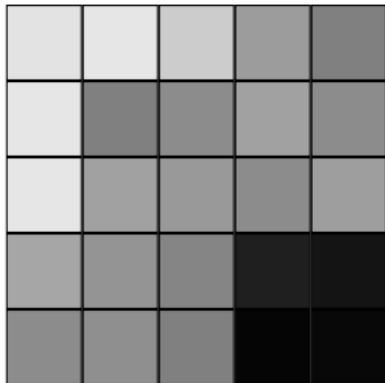
	13	5	15	
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
18	12	20		

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12\}$$

$$F = \langle 14, 15, 16, 17, 18, 19, 20 \rangle$$



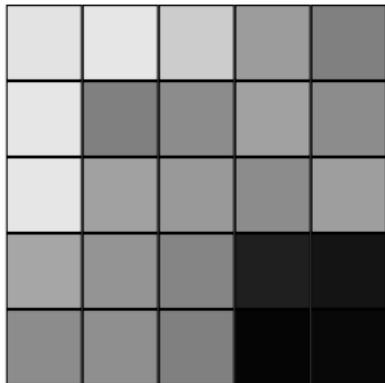
	13	5	15	
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12\}$$

$$F = \langle 15, 16, 17, 18, 19, 20 \rangle$$



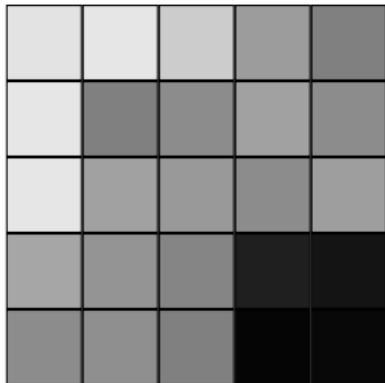
	13	5	15	
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15\}$$

$$F = \langle 16, 17, 18, 19, 20, 21 \rangle$$



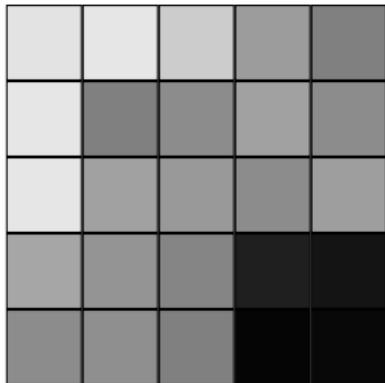
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16\}$$

$$F = \langle 17, 18, 19, 20, 21 \rangle$$



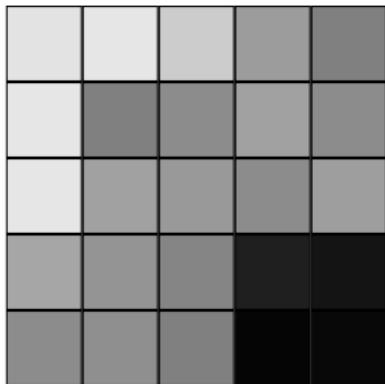
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16, 17\}$$

$$F = \langle 18, 19, 20, 21, 22 \rangle$$



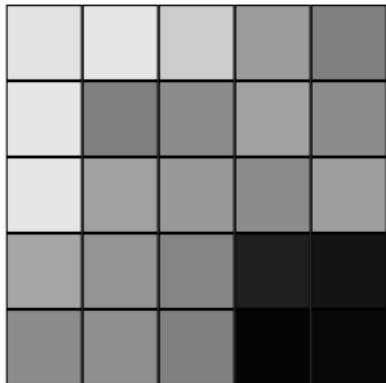
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
22	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16, 17, 18\}$$

$$F = \langle 19, 20, 21, 22 \rangle$$



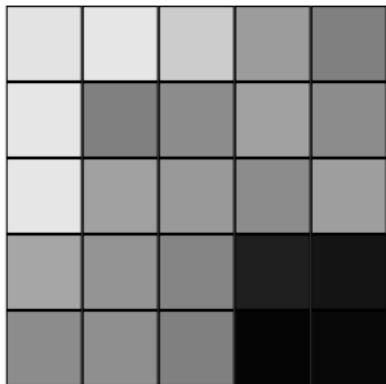
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
22	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16, 17, 18\}$$

$$F = \langle 20, 21, 22 \rangle$$



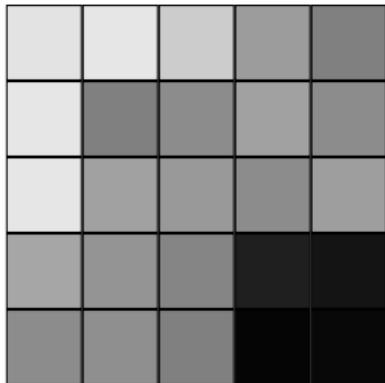
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
22	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16, 17, 18\}$$

$$F = \langle 21, 22 \rangle$$



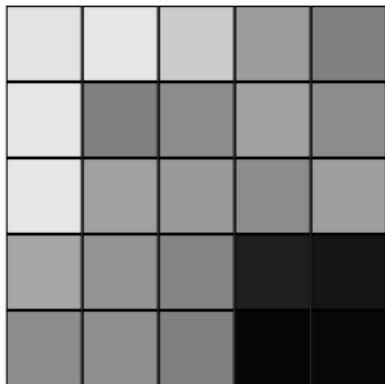
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
22	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16, 17, 18, 21\}$$

$$F = \langle 22 \rangle$$



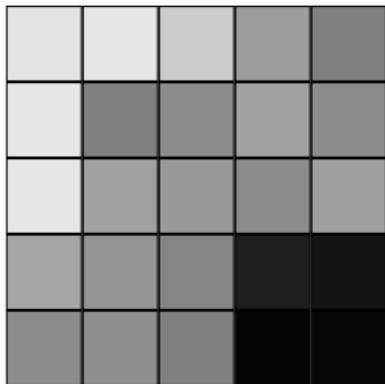
	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
22	18	12	20	

# Flood Fill

Com vizinhança  $N_4$

$$M = \{0, 1, 2, 3, 4, 6, 7, 9, 10, 12, 15, 16, 17, 18, 21, 22\}$$

$$F = \langle \rangle$$



	13	5	15	21
14	6	1	7	16
8	2	0	3	10
17	9	4	11	19
22	18	12	20	

# Flood Fill

C++      cv2::floodFill(image, mask, seed, newVal[, rect[, loDiff[, upDiff[, flags]]]])  
Python    cv2.floodFill(image, mask, seedPoint, newVal[, loDiff[, upDiff[, flags]]])

- ▶ O que significa “cor similar” em `cv2.floodFill`?
- ▶ Intervalo fixo (`cv2.FLOODFILL_FIXED_RANGE`):

$$I(p_0) - \text{loDiff} \leq I(p_i) \leq I(p_0) + \text{upDiff}$$

- ▶ Intervalo variável:

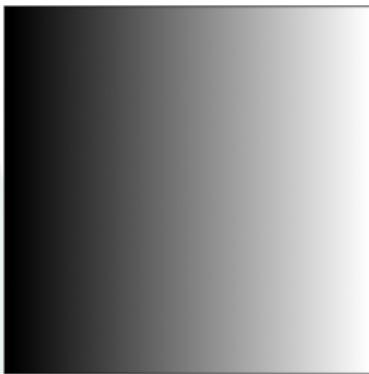
$$I(p_j) - \text{loDiff} \leq I(p_i) \leq I(p_j) + \text{upDiff}$$

para algum  $p_j$  tal que  $p_j \in M$  e  $p_j \in N$

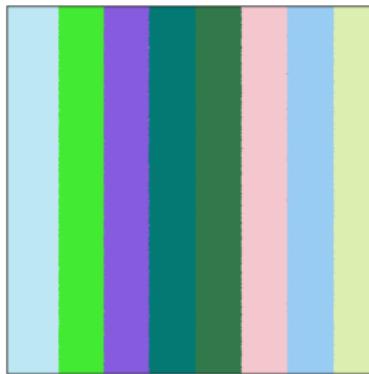
# Flood Fill

C++      cv2::floodFill(image, mask, seed, newVal[, rect[, loDiff[, upDiff[, flags]]]])  
Python    cv2.floodFill(image, mask, seedPoint, newVal[, loDiff[, upDiff[, flags]]])

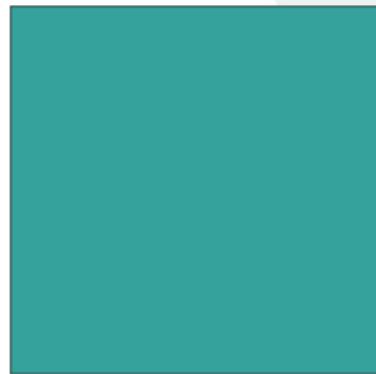
Original



Diferença à semente



Diferença ao vizinho



# Flood Fill

C++      cv2::floodFill(image, mask, seed, newVal[, rect[, loDiff[, upDiff[, flags]]]])  
Python    cv2.floodFill(image, mask, seedPoint, newVal[, loDiff[, upDiff[, flags]]])

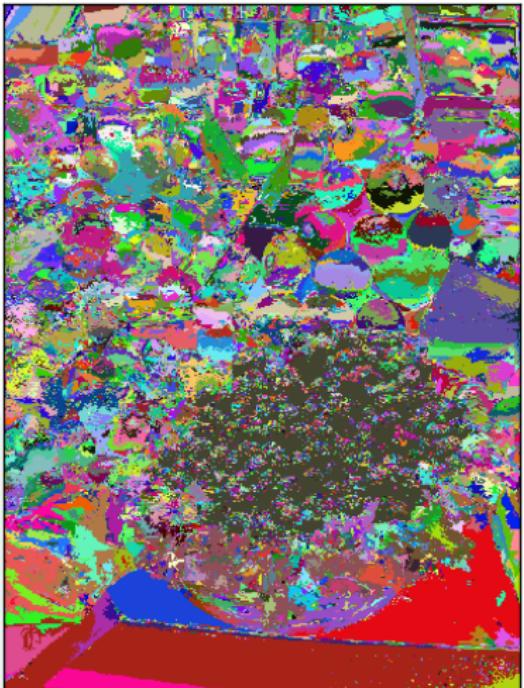
floodfill.py

```
16 def segment(image, M, N, flags):  
17  
18     mask = zeros(shape=(M+2, N+2), dtype=uint8)  
19     c = 0  
20  
21     for i in range(M):  
22         for j in range(N):  
23             # Como a mascara eh maior que a imagem, um pixel (i,  
             # j) na imagem  
24             # corresponde ao pixel i+1, j+1 na mascara  
25             if mask[i+1, j+1] == 0:  
26                 c += 1  
27                 new_val = random_color_code()  
28                 val, rect = cv2.floodFill(image, mask, (j,i),  
29                                         new_val, low_diff, high_diff, flags)  
30  
31     return c
```

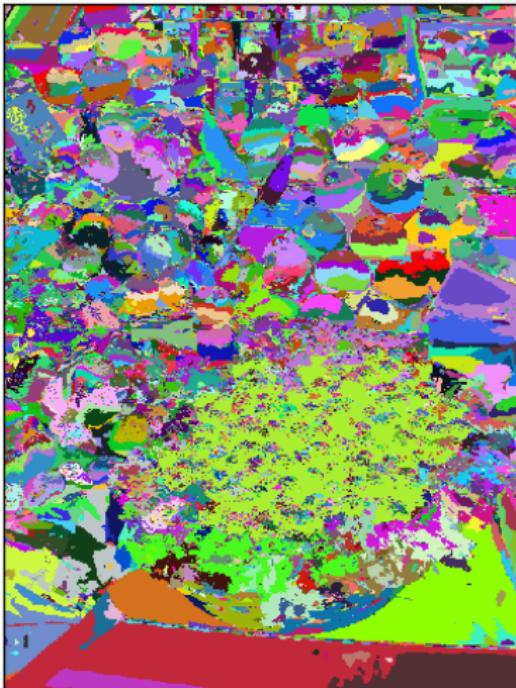
# Flood Fill

Distância à semente no espaço RGB

RGB



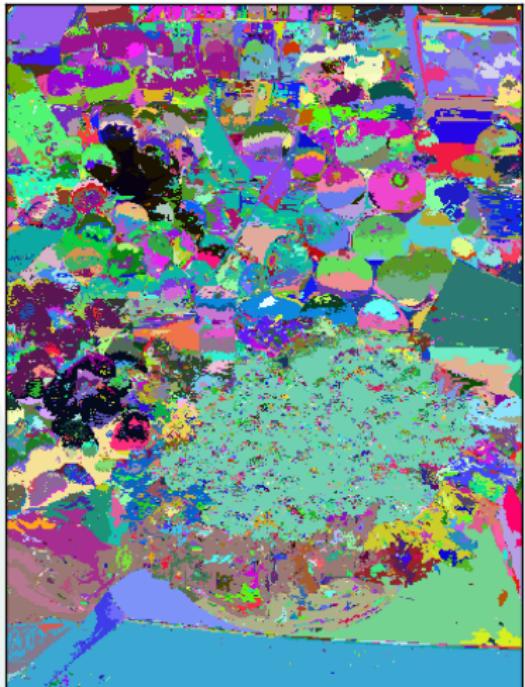
RGB e bilinear



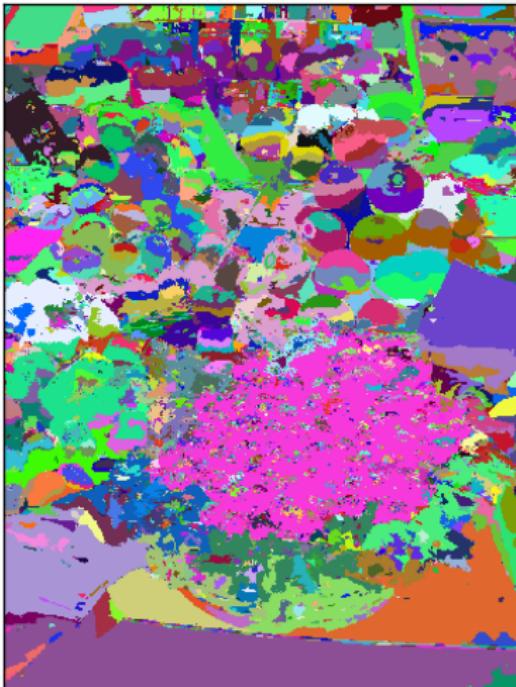
# Flood Fill

Distância à semente no espaço La\*b\*

La\*b\*



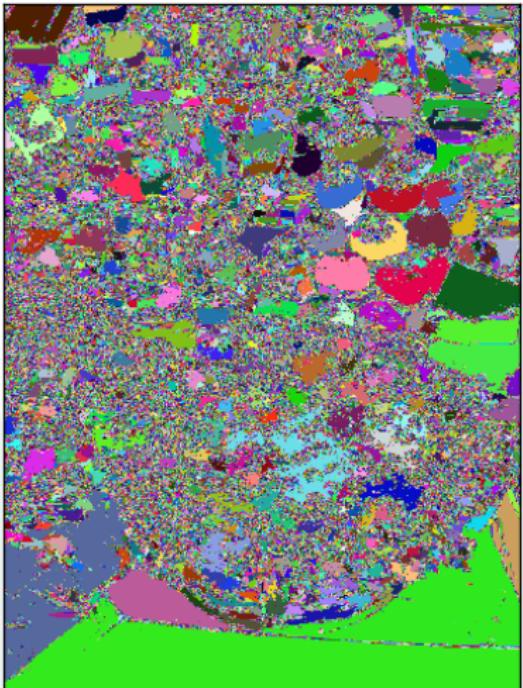
La\*b\* e bilinear



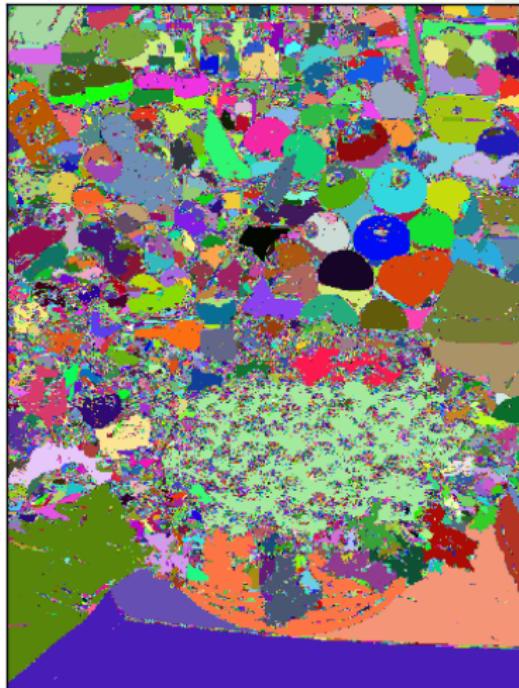
# Flood Fill

Distância ao vizinho no espaço RGB

RGB



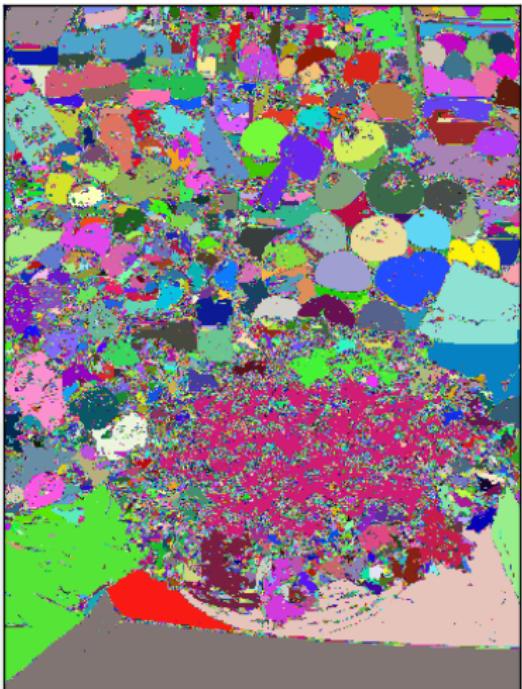
RGB e bilinear



# Flood Fill

Distância ao vizinho no espaço  $La^*b^*$

$La^*b^*$



$La^*b^*$  e bilinear

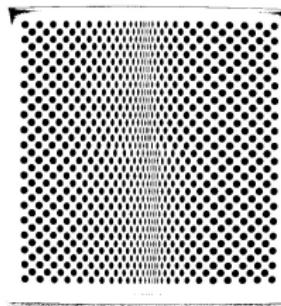
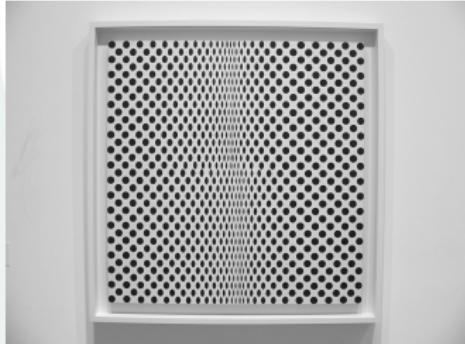
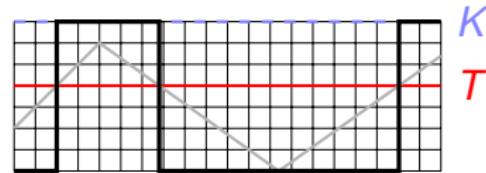


# Limiarização binária - cv2.THRESH\_BINARY

C++ `cv2::threshold(src, dst, T, K, cv2::THRESH_BINARY)`

Python `v, dst = cv2.threshold(src, T, K, cv2.THRESH_BINARY[, dst])`

$$I'(i,j) = \begin{cases} K & \text{se } I(i,j) > T \\ 0 & \text{caso contrário} \end{cases}$$

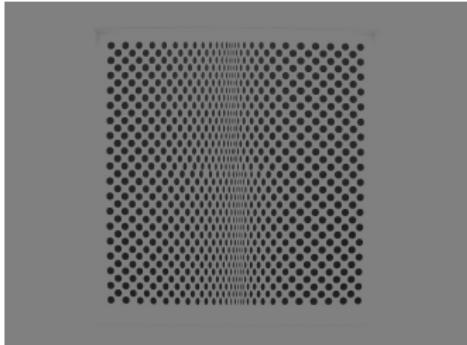
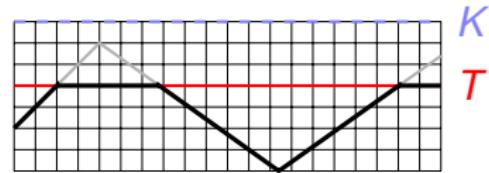


# Limiarização truncada - cv2.THRESH\_TRUNC

C++ `cv2::threshold(src, dst, T, K, cv2::THRESH_TRUNC)`

Python `v, dst = cv2.threshold(src, T, K, cv2.THRESH_TRUNC[, dst])`

$$I'(i,j) = \begin{cases} T & \text{se } I(i,j) > T \\ I(i,j) & \text{caso contrário} \end{cases}$$

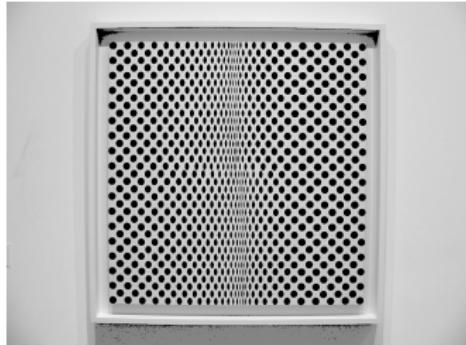
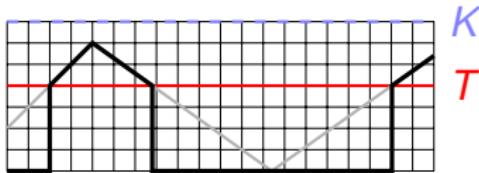


# Limiarização a zero - cv2.THRESH\_TOZERO

C++ `cv2::threshold(src, dst, T, K, cv2::THRESH_TOZERO)`

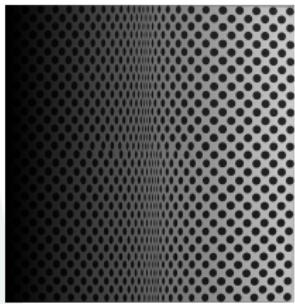
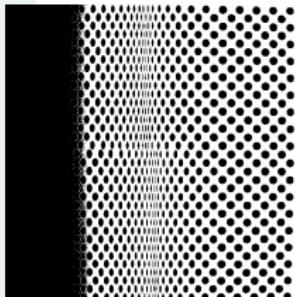
Python `v, dst = cv2.threshold(src, T, K, cv2.THRESH_TOZERO[, dst])`

$$I'(i,j) = \begin{cases} I(i,j) & \text{se } I(i,j) > T \\ 0 & \text{caso contrário} \end{cases}$$



# Limiarização adaptativa

Original

Limiarização,  $T = 55$ 

- ▶ Problemas com um **limiar global**
- ▶ Assume iluminação homogênea
- ▶ Alternativa: **limiar adaptativo**
  - ▶ Limiar  $T(i, j)$  computado para cada pixel
- ▶ cv2.adaptiveThreshold
  - ▶ Toma uma janela  $b \times b$  centrada em  $(i, j)$
  - ▶ Computa a média (ponderada)  $m(i, j)$
  - ▶ Assume  $T(i, j) = m(i, j) - c$
  - ▶  $c$  é uma constante
  - ▶ A média  $m(i, j)$  pode ser:
    - ▶ média simples
  - ▶ ADAPTIVE THRESH MEAN C
  - ▶ pesos Gaussianos
  - ▶ ADAPTIVE THRESH GAUSSIAN C

# Limiarização adaptativa

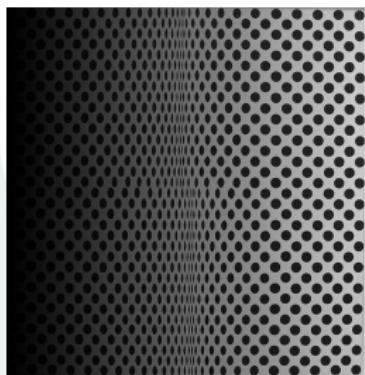
C++

```
cv2::adaptiveThreshold(src, dst, maxValue, adaptMethod, threshType, blockSize, C)
```

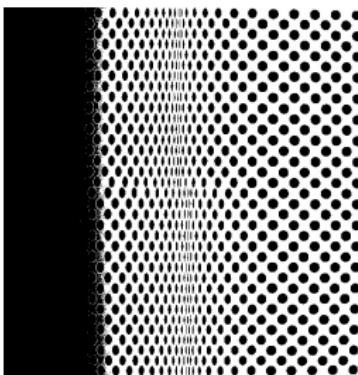
Python

```
dst = cv2.adaptiveThreshold(src, maxValue, adaptMethod, threshType, blockSize, C)
```

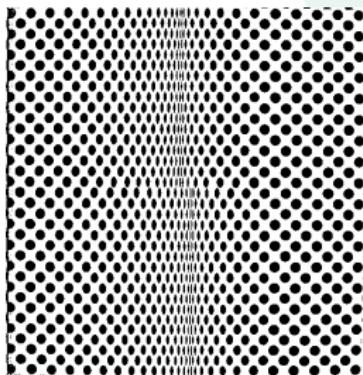
Original



Limiar T = 55



Limiar adaptativo  $b = 25, c = 1$



## Exercício 3

Carregue *Mapa* (*mapa.png*) em tons de cinza.

1. Experimente diversas operações morfológicas com diferentes elementos estruturantes (caixas, cruzes, barras) e veja como eles afetam as estruturas na imagem.
2. Tente encontrar uma sequência de operações morfológicas que seja capaz de **identificar a grade** (latitudes e longitudes) na imagem.

# Exercício

## Exercício 4

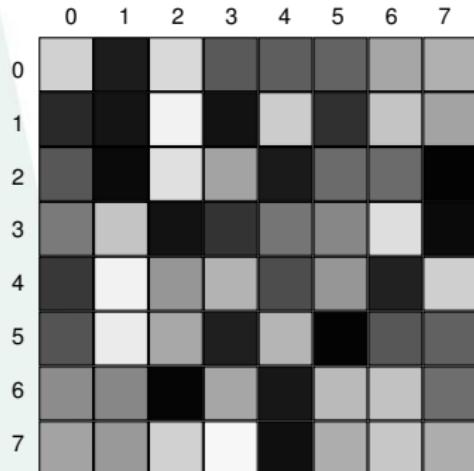
Carregue *Caneca 1* e *Caneca 2* como imagens em tons de cinza

1. Faça a diferença absoluta entre as imagens. Além dos pixels pertencentes aos dois novos objetos, quais outros pixels são evidenciados e por quê?
2. Binarize a imagem, utilizando limiarização, para demarcar os objetos.
3. Utilize operações morfológicas para se livrar dos artefatos indesejáveis.
4. Utilize *flood fill* para identificar os dois objetos. Que critério pode ser utilizado para remover artefatos indesejáveis que ainda estejam na cena?

# Pirâmides - downsampling

C++      `cv::pyrDown(src, dst[, dstsize])`

Python    `cv2.pyrDown(src[, dst[, dstsize]])`



1. Convolução com um kernel Gaussiano:

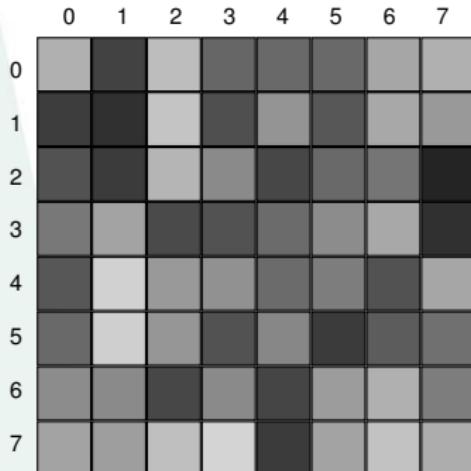
\*  $G_{5 \times 5}$

$$G_{5 \times 5} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

2. Remova as linhas e colunas pares

# Pirâmides - *downsampling*

C++      cv2::pyrDown(src, dst[, dstsize])  
Python    cv2.pyrDown(src[, dst[, dstsize]])



1. Convolução com um kernel Gaussiano:

$$G_{5 \times 5} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

2. Remova as linhas e colunas pares

# Pirâmides - *downsampling*

C++      `cv2::pyrDown(src, dst[, dstsize])`  
Python    `cv2.pyrDown(src[, dst[, dstsize]])`

0	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

1. Convolução com um kernel Gaussiano:

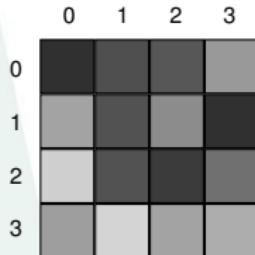
$$G_{5 \times 5} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

2. Remova as linhas e colunas pares

# Pirâmides - *downsampling*

C++      cv2::pyrDown(src, dst[, dstsize])

Python    cv2.pyrDown(src[, dst[, dstsize]])



1. Convolução com um kernel Gaussiano:

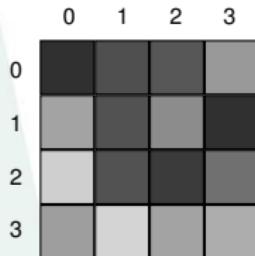
$$G_{5 \times 5} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

2. Remova as linhas e colunas pares

# Pirâmides - *upsampling*

C++      `cv2::pyrUp(src, dst[, dstsize])`

Python    `cv2.pyrUp(src[, dst[, dstsize]])`



1. Adicione linhas e colunas pares
2. Convolução com um kernel Gaussiano x4:

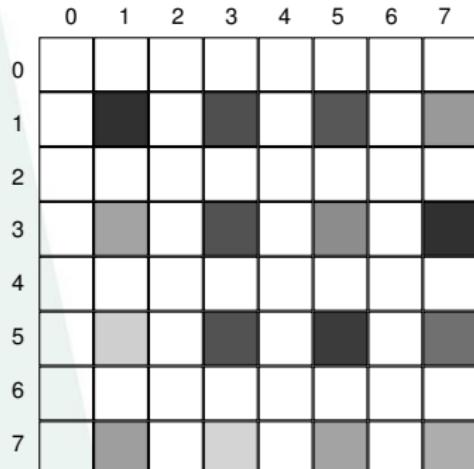
$$G_{5 \times 5} = \frac{4}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

# Pirâmides - *upsampling*

C++      cv2::pyrUp(src, dst[, dstsize])

Python    cv2.pyrUp(src[, dst[, dstsize]])

0	1	2	3	4	5	6	7
0							
1							
2							
3							
4							
5							
6							
7							



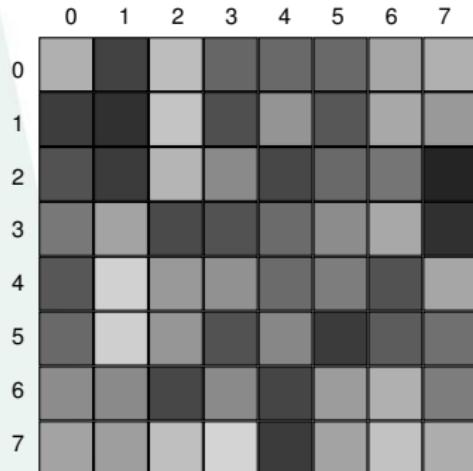
1. Adicione linhas e colunas pares
2. Convolução com um kernel Gaussiano x4:

$$G_{5 \times 5} = \frac{4}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

# Pirâmides - *upsampling*

C++      cv2::pyrUp(src, dst[, dstsize])

Python    cv2.pyrUp(src[, dst[, dstsize]])



1. Adicione linhas e colunas pares
2. Convolução com um kernel Gaussiano x4:

$$G_{5 \times 5} = \frac{4}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

# Pirâmides



Nível 0



Nível 1

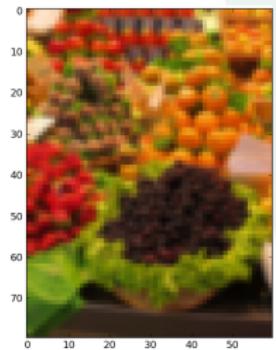
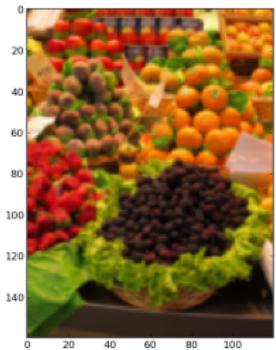


Nível 2



Nível 3

# Pirâmides



# Outline

Introdução à OpenCV

Operações com matrizes

Cor

Operações básicas em processamento de imagens

Transformações de imagem

Histogramas

# Deteção de bordas: Sobel

C++      cv2::Sobel(src, dst, ddepth, dx, dy[, ksize[, scale[, delta[, borderType]]]]))  
Python    cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]]))

- ▶ **Bordas** correspondem a pontos onde há **descontinuidade**
- ▶ Considere a imagem como uma **função de intensidade  $f$** 
  - ▶ bordas surgem como **máximos locais** no gradiente de  $f$
- ▶ **Imagen digital**
  - ▶ amostragem de uma função de intensidade
- ▶ Desejamos aproximar  $\frac{\delta f}{\delta x}$  e  $\frac{\delta f}{\delta y}$

# Deteção de bordas: Sobel

C++      `cv2::Sobel(src, dst, ddepth, dx, dy[, ksize[, scale[, delta[, borderType]]]])`  
Python    `cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]])`

- ▶  $\frac{\delta f}{\delta x}$  pode ser aproximada por  $G_x = f * h_x$  em que

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- ▶  $\frac{\delta f}{\delta y}$  pode ser aproximada por  $G_y = f * h_y$  em que

$$h_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{bmatrix}$$

- ▶ O **gradiente** pode ser aproximado por  $G = \sqrt{G_x^2 + G_y^2}$

# Detectção de bordas: Sobel

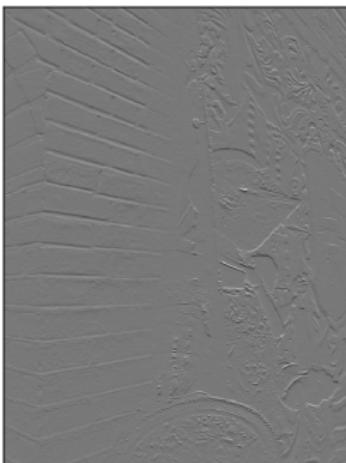
C++      `cv2::Sobel(src, dst, ddepth, dx, dy[, ksize[, scale[, delta[, borderType]]]])`

Python    `cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]])`

$G_x$



$G_y$



$$G = \sqrt{G_x^2 + G_y^2}$$



# Deteção de bordas: Sobel

C++      cv2::Sobel(src, dst, ddepth, dx, dy[, ksize[, scale[, delta[, borderType]]]]))  
Python    cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]]))

## sobel.py

```
22 Gx = cv2.Sobel(src, cv2.CV_32FC1, 1, 0)
23 Gy = cv2.Sobel(src, cv2.CV_32FC1, 0, 1)
24 G = sqrt(Gx**2 + Gy**2)
```

- ▶  $dx = 1, dy = 0$  produz  $G_x \sim \frac{\delta f}{\delta x}$
- ▶  $dx = 0, dy = 1$  produz  $G_y \sim \frac{\delta f}{\delta y}$
- ▶  $dx = 1, dy = 1$  produz  $G_{xy} \sim \frac{\delta^2 f}{\delta x \delta y}$

# Deteção de bordas: Sobel

C++      cv2::Sobel(src, dst, ddepth, dx, dy[, ksize[, scale[, delta[, borderType]]]]])  
Python    cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]]))

- ▶ Kernels de Sobel podem ser definidos para qualquer tamanho
- ▶ Kernels maiores geram aproximações melhores às derivadas

## Exemplo

$$h_x^{7 \times 7} = \begin{bmatrix} -1 & -6 & -15 & -20 & -15 & -6 & -1 \\ -4 & -24 & -60 & -80 & -60 & -24 & -4 \\ -5 & -30 & -75 & -100 & -75 & -30 & -5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 30 & 75 & 100 & 75 & 30 & 5 \\ 4 & 24 & 60 & 80 & 60 & 24 & 4 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}$$

# Detecção de bordas: Canny

C++      `cv2::Canny(src, edges, threshold1, threshold2[, apertureSize=3[, L2gradient]])`  
Python    `cv2.Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]])`

- ▶ Canny criou um detector de bordas que produz **contornos**
- ▶ A orientação da borda em um pixel pode ser aproximada por  $\theta = \arctan \frac{G_y}{G_x}$
- ▶ Dois limiares são utilizados,  $t_1 < t_2$
- ▶ Pixels onde  $G(i, j) > t_2$  são imediatamente selecionados
- ▶ Pixels onde  $G(i, j) < t_1$  são imediatamente descartados
- ▶ Partindo-se dos pixels selecionados e seguindo a orientação da borda, os pixels onde  $G(i, j) > t_1$  são selecionados

# Detecção de bordas: Canny

C++

```
cv::Canny(src, edges, threshold1, threshold2[, apertureSize=3[, L2gradient]])
```

Python

```
cv2.Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]])
```

/



Canny



# Detecção de bordas: Canny

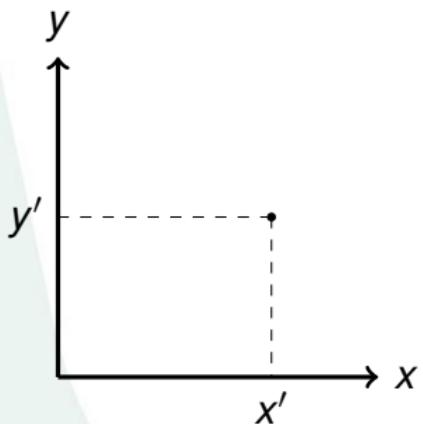
C++      cv2::Canny(src, edges, threshold1, threshold2[, apertureSize=3[, L2gradient]])  
Python    cv2.Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]])

## canny.py

```
22 Gx = cv2.Sobel(src, cv2.CV_32FC1, 1, 0, ksize=7)
23 Gy = cv2.Sobel(src, cv2.CV_32FC1, 0, 1, ksize=7)
24 G = sqrt(Gx**2 + Gy**2)
25
26 # Vamos tentar encontrar bons valores para os thresholds
27 # Considere t = media + desvio_padrao
28 [[mu]], [[sigma]] = cv2.meanStdDev(G)
29 t = mu + sigma
30
31 # Como sugerido por Canny, uma relacao 3:1 entre os
32 # thresholds
33 C = cv2.Canny(src, t/3, t, apertureSize=7)
```

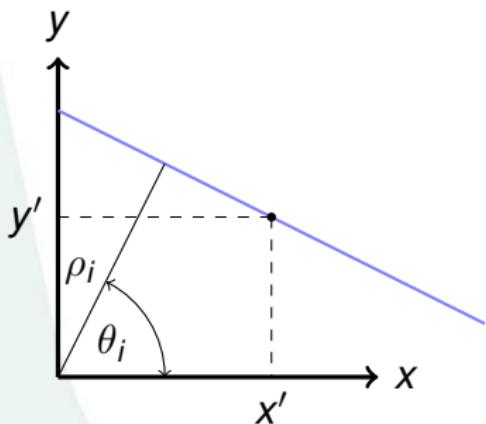
# Transformada de Hough

- ▶ A Transformada de Hough identifica objetos representados em um **espaço paramétrico**
- ▶ Exemplo: retas parametrizadas por  $\rho$  e  $\theta$
- ▶ Hough implementa uma “**votação**”



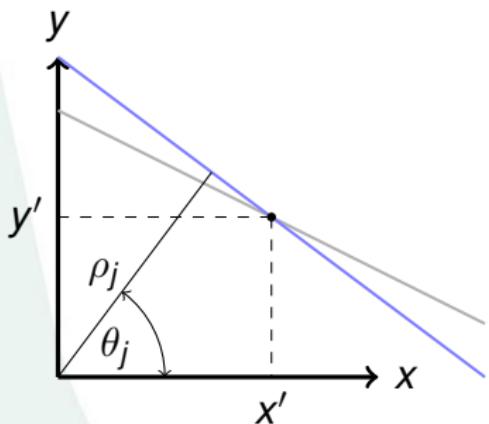
# Transformada de Hough

- ▶ A Transformada de Hough identifica objetos representados em um **espaço paramétrico**
- ▶ Exemplo: retas parametrizadas por  $\rho$  e  $\theta$
- ▶ Hough implementa uma “**votação**”



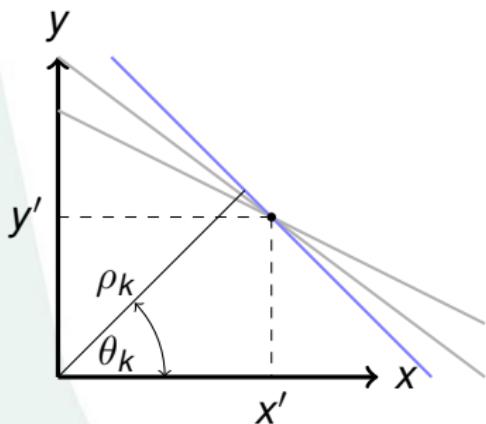
# Transformada de Hough

- ▶ A Transformada de Hough identifica objetos representados em um **espaço paramétrico**
- ▶ Exemplo: retas parametrizadas por  $\rho$  e  $\theta$
- ▶ Hough implementa uma “**votação**”



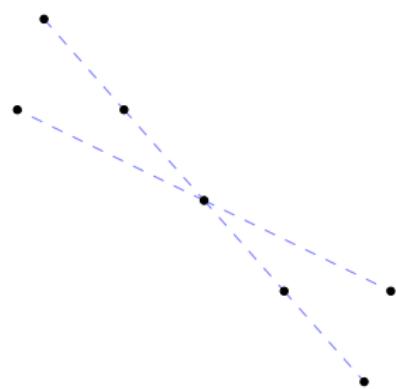
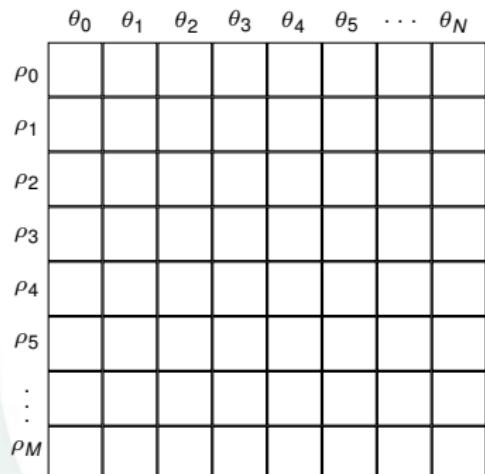
# Transformada de Hough

- ▶ A Transformada de Hough identifica objetos representados em um **espaço paramétrico**
- ▶ Exemplo: retas parametrizadas por  $\rho$  e  $\theta$
- ▶ Hough implementa uma “**votação**”



# Transformada de Hough

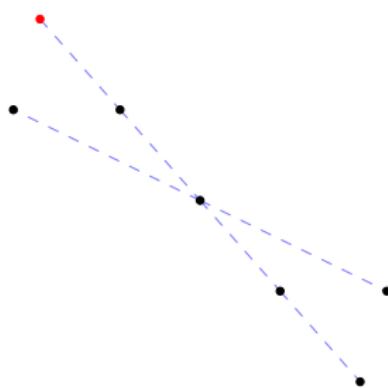
Exemplo com limiar = 4



# Transformada de Hough

Exemplo com limiar = 4

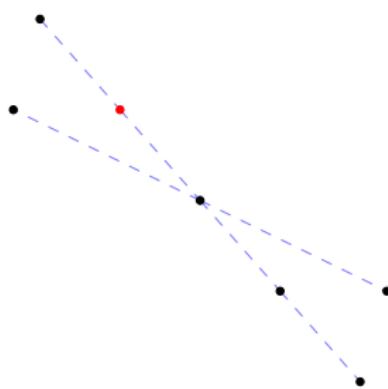
	$\theta_0$	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	$\dots$	$\theta_N$
$\rho_0$								
$\rho_1$								
$\rho_2$			1					
$\rho_3$								
$\rho_4$								
$\rho_5$					0			
:								
$\rho_M$								



# Transformada de Hough

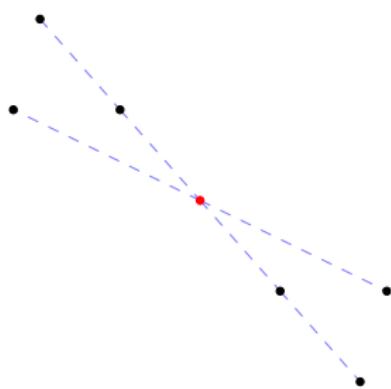
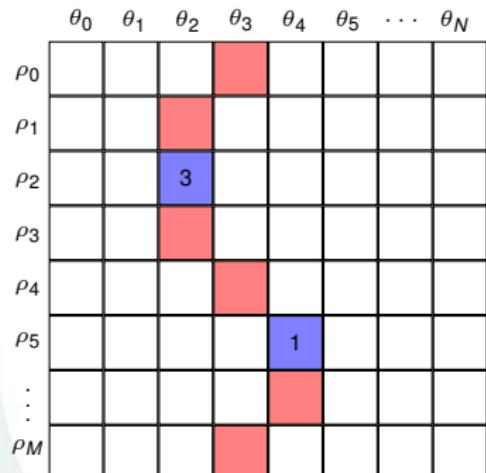
Exemplo com limiar = 4

	$\theta_0$	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	$\dots$	$\theta_N$
$\rho_0$								
$\rho_1$								
$\rho_2$			2					
$\rho_3$								
$\rho_4$								
$\rho_5$					0			
:								
$\rho_M$								



# Transformada de Hough

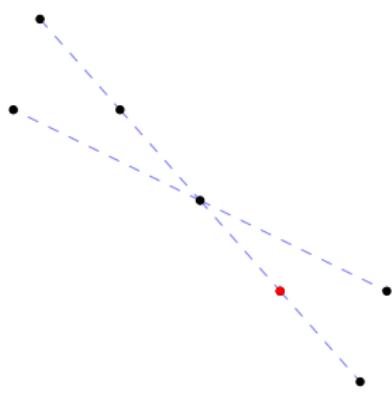
Exemplo com limiar = 4



# Transformada de Hough

Exemplo com limiar = 4

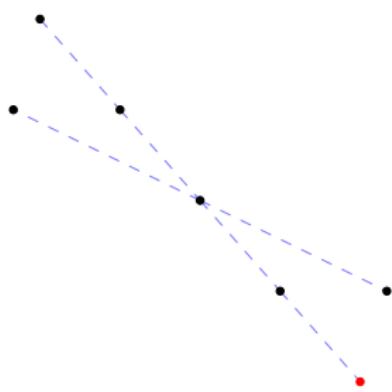
	$\theta_0$	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	$\dots$	$\theta_N$
$\rho_0$								
$\rho_1$								
$\rho_2$			4					
$\rho_3$								
$\rho_4$								
$\rho_5$					1			
:								
$\rho_M$								



# Transformada de Hough

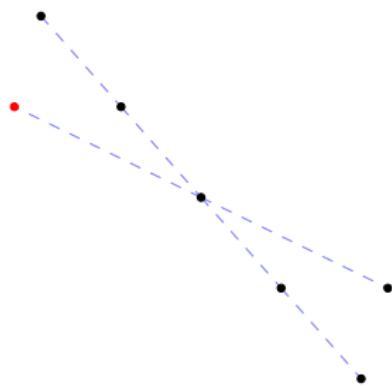
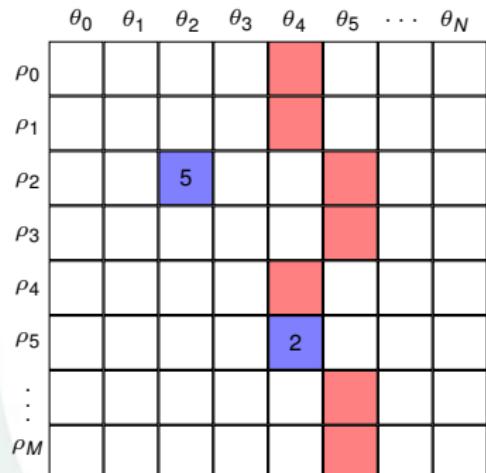
Exemplo com limiar = 4

	$\theta_0$	$\theta_1$	$\theta_2$	$\theta_3$	$\theta_4$	$\theta_5$	$\dots$	$\theta_N$
$\rho_0$	Red							
$\rho_1$		Red						
$\rho_2$			5					
$\rho_3$				Red				
$\rho_4$		Red						
$\rho_5$	Red				1			
:								
$\rho_M$		Red						



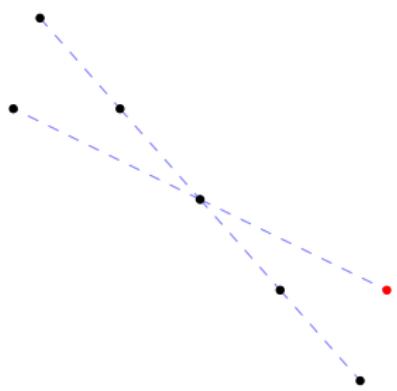
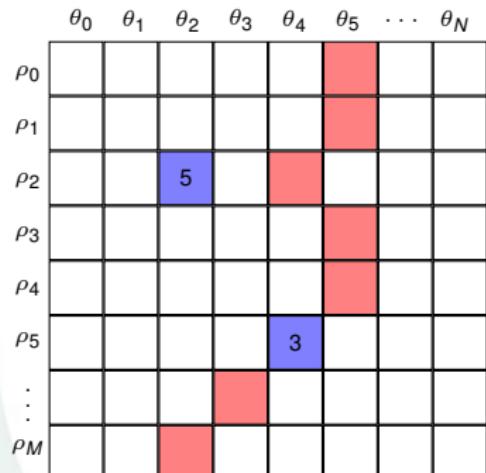
# Transformada de Hough

Exemplo com limiar = 4



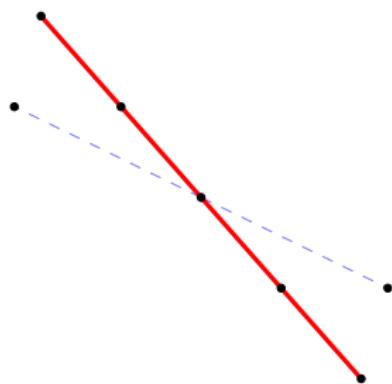
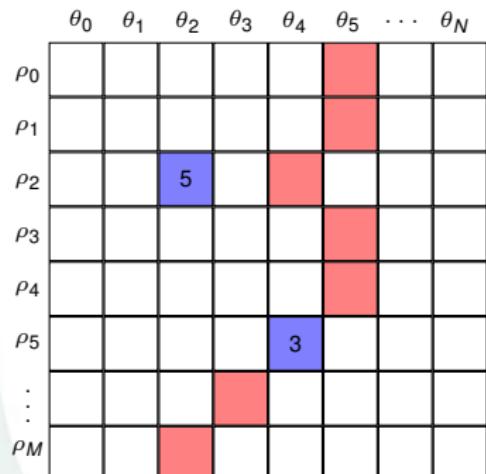
# Transformada de Hough

Exemplo com limiar = 4



# Transformada de Hough

Exemplo com limiar = 4



# Transformada de Hough

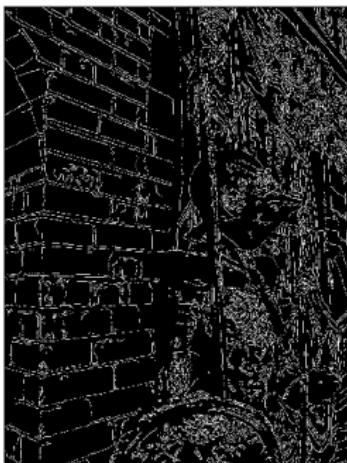
C++      `cv2::HoughLines(image, lines, rho, theta, threshold[, srn[, stn]])`

Python    `lines = cv2.HoughLines(image, rho, theta, threshold[, lines[, srn[, stn]]])`

/



Canny



Hough



# Transformada de Hough

C++      cv2::HoughLines(image, lines, rho, theta, threshold[, srn[, stn]])

Python    lines = cv2.HoughLines(image, rho, theta, threshold[, lines[, srn[, stn]]])

## hough.py

```
48 # Rho com resolucao de 1 pixel
49 # Theta com resolucao de 1o (PI/180 radianos)
50 [lines] = cv2.HoughLines(C, 1, pi/180, acc_threshold)
51
52 for rho, theta in lines:
53     # 60o < theta < 120o
54     if (pi/3 < theta and theta < 2*pi/3):
55         y = lambda x : int((rho - x*cos(theta))/sin(theta))
56         cv2.line(hough, (0, y(0)), (N, y(N)), (0, 0, 255),
57                  thickness=1, lineType=cv2.CV_AA)
```

## Exercício 5

Carregue *Pivôs* (`pca-1_cmp_from_5_bands.pgm`) em tons de cinza.

1. Veja a função `HoughCircles` na documentação da OpenCV.
2. Tente recuperar a localização dos pivôs (áreas de cultivo hirrigadas de padrão circular) utilizando `HoughCircles`. Há algum pré-processamento que você pode realizar na imagem para auxiliar o processo?
3. Veja a função `circle` e desenhe circunferências coloridas demarcando a localização das arruelas. Lembre-se de utilizar uma imagem colorida RGB como saída.

# Outline

Introdução à OpenCV

Operações com matrizes

Cor

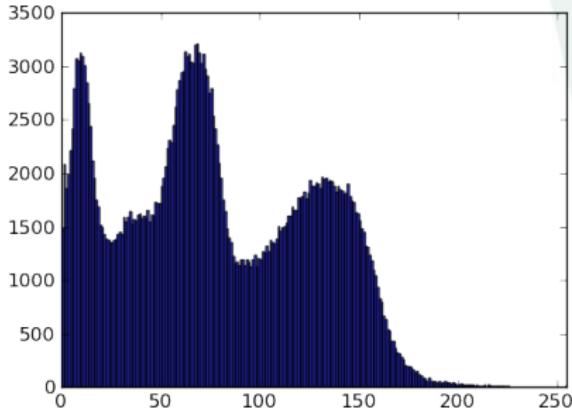
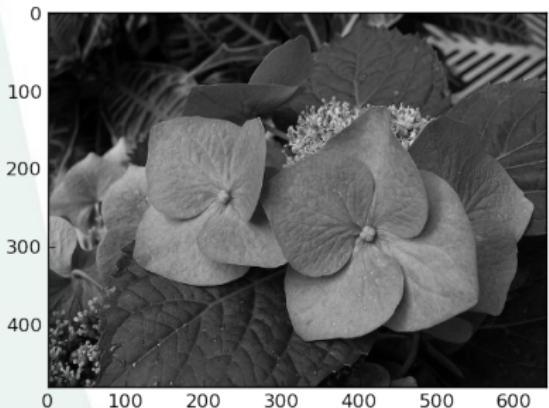
Operações básicas em processamento de imagens

Transformações de imagem

Histogramas

# Histogramas

- ▶ **Histogramas** são tabelas de frequência, uma estimativa para uma função de probabilidade
- ▶ O histograma de uma imagem consiste na contagem do números de pixels encontrados para cada cor



# Equalização de histogramas

C++      `cv::equalizeHist(src, dst)`  
Python    `dst = cv2.equalizeHist(src)`

- ▶ Se a maioria dos pixels se concentra em torno de um determinado valor, a imagem apresenta **baixo contraste**
- ▶ A **equalização de histograma** define um mapeamento de valores que expande o histograma
- ▶ O histograma é normalizado de forma que a soma dos *bins* seja 255.
- ▶ O **histograma acumulado**  $H'(I)$  é computado:

$$H'(I) = \sum_{0 \leq k < I} H(k)$$

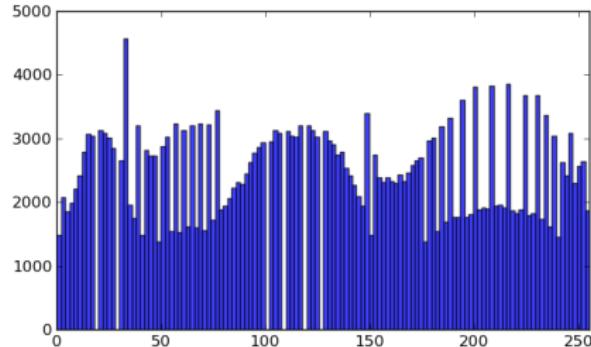
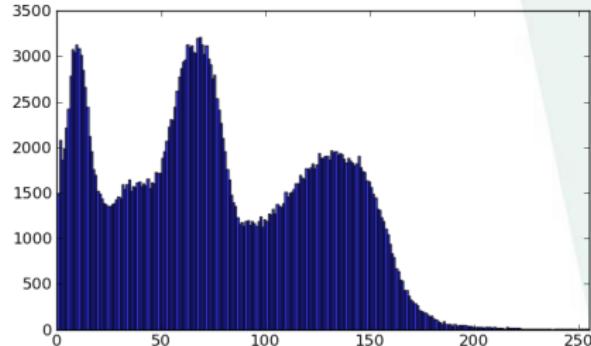
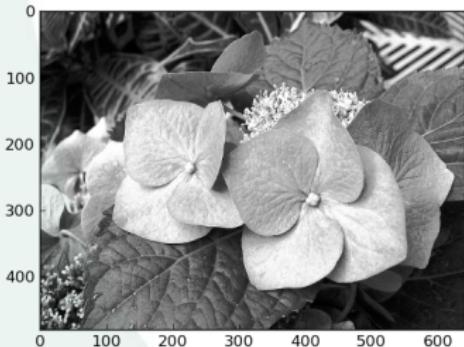
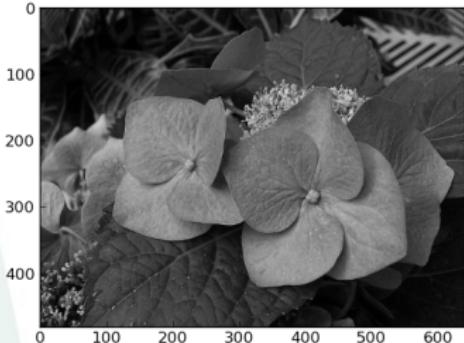
- ▶  $H'(I)$  é utilizado como uma *look-up table*:

$$\text{dst}(i, j) = H'(\text{src}(i, j))$$

# Equalização de histogramas

C++      `cv::equalizeHist(src, dst)`

Python    `dst = cv2.equalizeHist(src)`



# Exercício

## Exercício 6

Carregue uma **imagem colorida** como *Frutas* ou *Flores*.

1. Use `cv2.split` para obter os canais R, G e B
2. Equalize cada um dos três canais. Use `cv2.merge` para produzir uma nova imagem colorida com os três canais equalizados.
3. Converta a imagem original para o espaço La\*b\* (ou HSV). Use `cv2.split` para separar os canais.
4. Equalize **somente o canal L (ou H)**. Use `cv2.merge` para obter uma nova imagem colorida com os três canais.
5. Compare os resultados. Qual seria a forma “correta” para equalizar uma imagem colorida?

# Cálculo de histogramas

C++      `cv::calcHist(arrays, narrays, channels, mask, hist, dims, histSize, ranges, uniform, accumulate)`

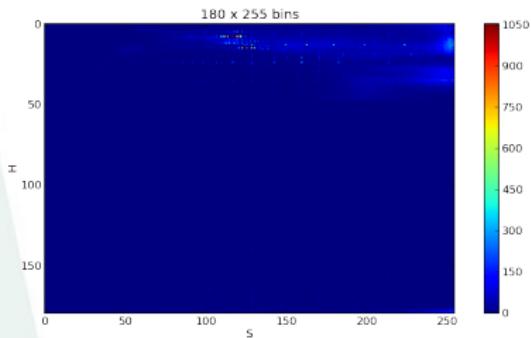
Python    `hist = cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])`

- ▶ Computação de histogramas N-dimensionais
- ▶ Exemplos:
  - ▶ um histograma 1D de L;
  - ▶ um histograma 2D de H e S
  - ▶ um histograma 3D de R, G, B.
- ▶ Questões de **quantização devem ser consideradas**
- ▶ Histogramas muito esparsos podem apresentar bins com apenas uma observação

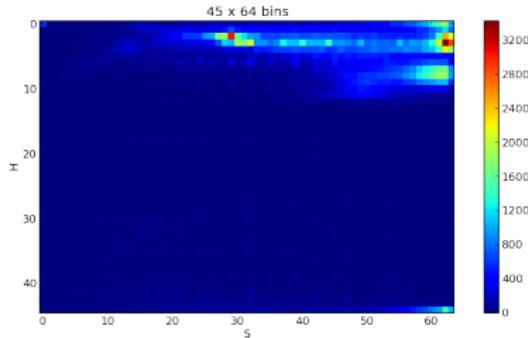
# Cálculo de histogramas

O problema com histogramas esparsos

H em 180 bins  
S em 255 bins



H em 45 bins  
S em 64 bins



# Cálculo de histogramas

C++      cv::calcHist(arrays, narrays, channels, mask, hist, dims, histSize, ranges, uniform, accumulate)  
Python    hist = cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])

## histograms.py

```
8     image = cv2.imread(argv[1], cv2.CV_LOAD_IMAGE_GRAYSCALE)
9
10    # Histograma 1D
11    h = cv2.calcHist([image],channels=[0],mask=None, histSize
12                  =[255], ranges=[0,255])
13    pylab.plot(h)
14    pylab.show()
```

# Cálculo de histogramas

C++      cv::calcHist(arrays, narrays, channels, mask, hist, dims, histSize, ranges, uniform, accumulate)  
Python    hist = cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])

## histograms.py

```
15 # Histograma 2D envolvendo H e S
16 image = cv2.imread(argv[1], cv2.CV_LOAD_IMAGE_COLOR)
17 hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
18 h = cv2.calcHist([hsv],
19                 channels=[0,1],           # Utilize hue (0) e
20                 sat (1)
21                 mask=None,
22                 histSize=[36,50],        # Quantizacao 36 bins
23                               # (hue) e 50 bins (sat)
24                 ranges=[0,180,0,255])
25 pylab.imshow(h, interpolation='nearest')
26 pylab.colorbar()
27 pylab.show()
```

# Comparação de histogramas

C++      `cv::compareHist(H1, H2, method)`  
Python    `retval = cv2.compareHist(H1, H2, method)`

- ▶ Imagens podem ser comparadas através de seus histogramas
- ▶ A distribuição de cores é separada, não a maneira como as cores se organizam na imagem
  - ▶ Exemplo: bandeira da França vs. bandeira da Holanda
- ▶ Há 4 métodos diferentes para comparar histogramas

Bhattacharyya distance

`cv.CV_COMP_BHATTACHARYYA`

Chi-Quadrado

`cv.CV_COMP_CHISQR`

Correlação

`cv.CV_COMP_CORREL`

Intersecção

`cv.CV_COMP_INTERSECT`

# Reprojeção de histograma

C++      `cv::calcBackProject(images, narrays, channels, hist, backProject, ranges, 1, uniform)`

Python    `dst = cv2.calcBackProject(images, channels, hist, ranges[, dst[, scale]])`

- ▶ Considere um histograma representando uma distribuição de interesse, por exemplo, a cor de uma família de objetos
- ▶ A reprojeção encontra o *bin* de cada pixel  $(i, j)$
- ▶ O valor do pixel  $(i, j)$  na imagem reprojetada corresponde ao valor do *bin*

# Exercício

## Exercício 7

Utilizando uma câmera e um editor de imagens, vamos produzir uma imagem contendo **amostras de cor de pele**.

1. Use `cv2.calcHist` para produzir um histograma 2D, matriz  $\times$  saturação ou  $a * \times b *$ , que represente a distribuição de cores de pele.
2. Carregue uma imagem do grupo. Use `cv2.calcBackProject` para encontrar faces, braços e mãos das pessoas em cena.
3. Utilize operadores morfológicos para corrigir falhas nas regiões encontradas.

A transformada de Fourier faz a decomposição de uma imagem em componentes formadas por senos e cossenos. Ela transforma a imagem do **domínio espacial** para o **domínio da frequência**:

$$F(u, v) = \sum_{x=0}^N \sum_{y=0}^M f(x, y) e^{-i2\pi(ux/N + vy/M)}$$

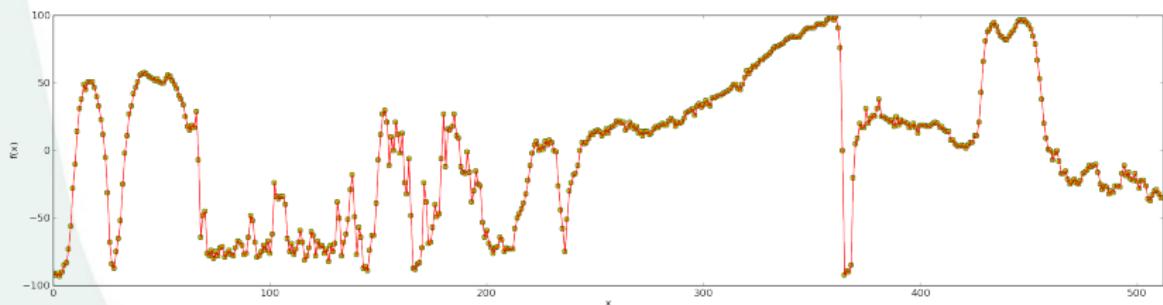
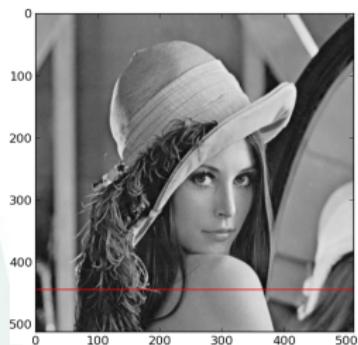
# Transformada Discreta de Fourier

## Domínio espacial

- ▶ Podemos pensar na nossa imagem como um **sinal**
- ▶ Domínio: posição no espaço,  $x$
- ▶ Imagem: valor do pixel naquela posição do espaço:  $f(x)$

# Transformada Discreta de Fourier

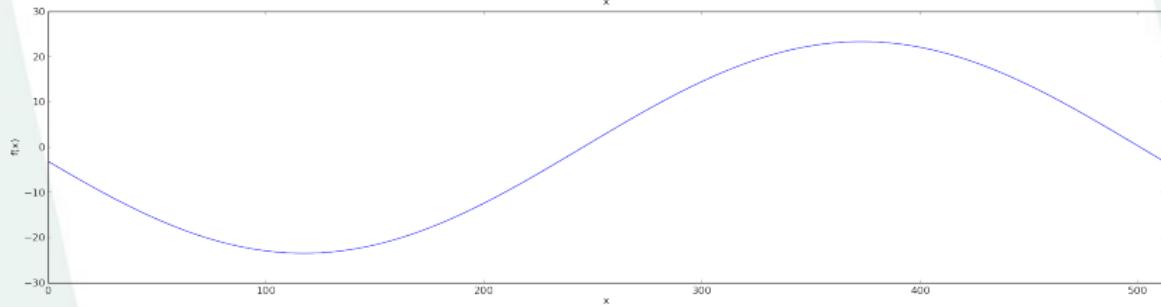
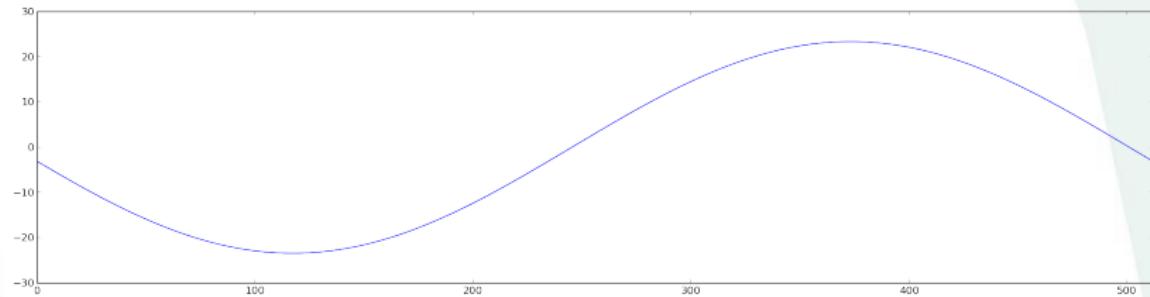
## Domínio espacial



# Transformada Discreta de Fourier

Intuição

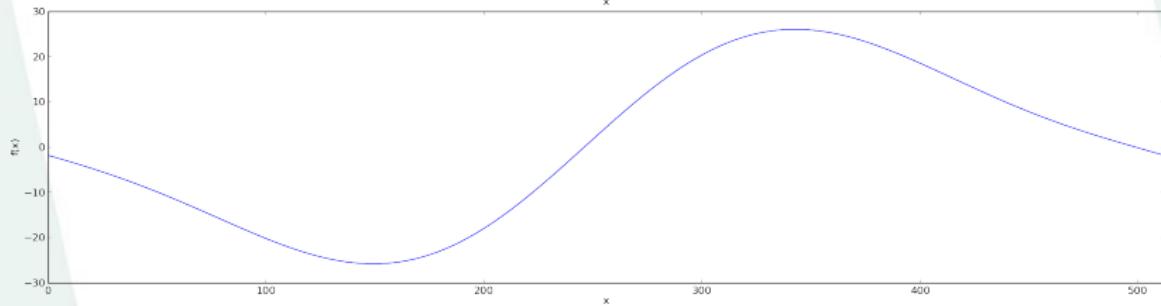
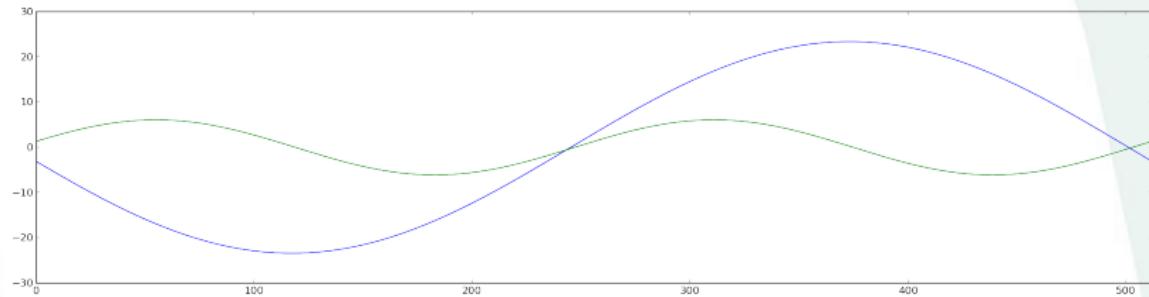
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

## Intuição

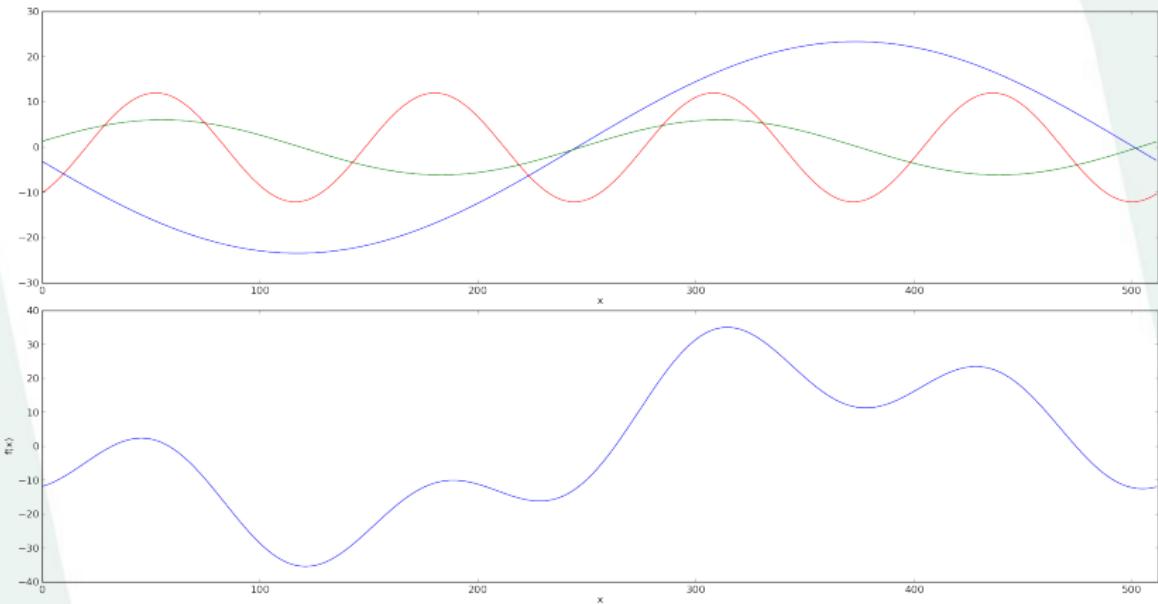
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

## Intuição

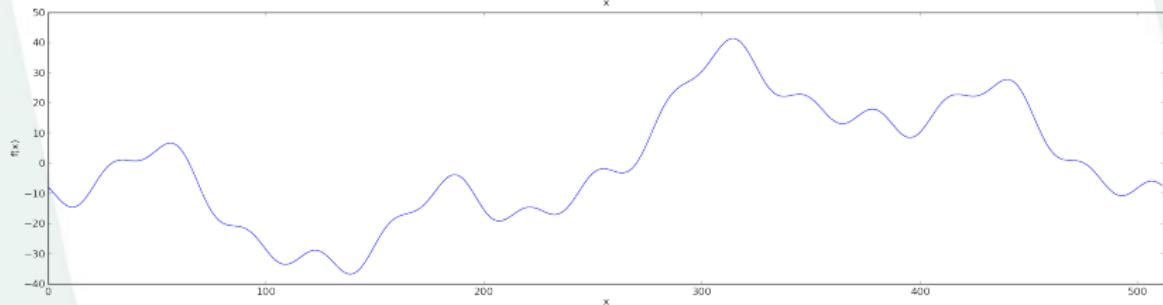
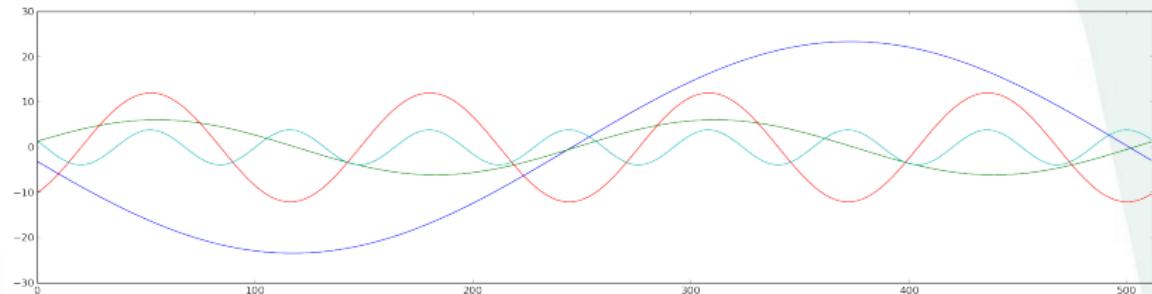
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

Intuição

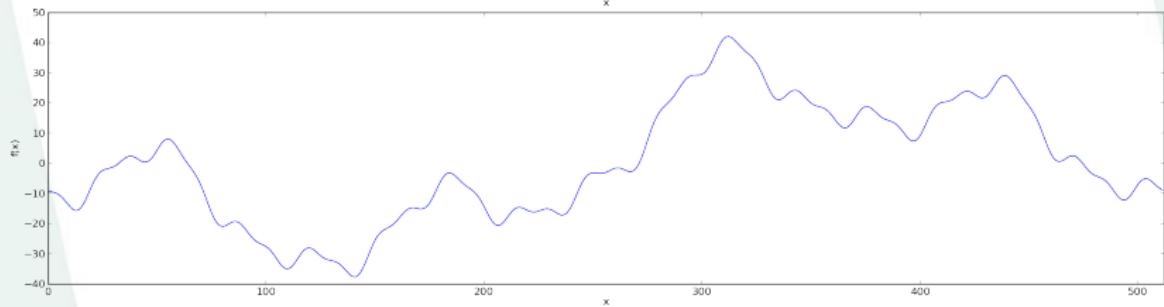
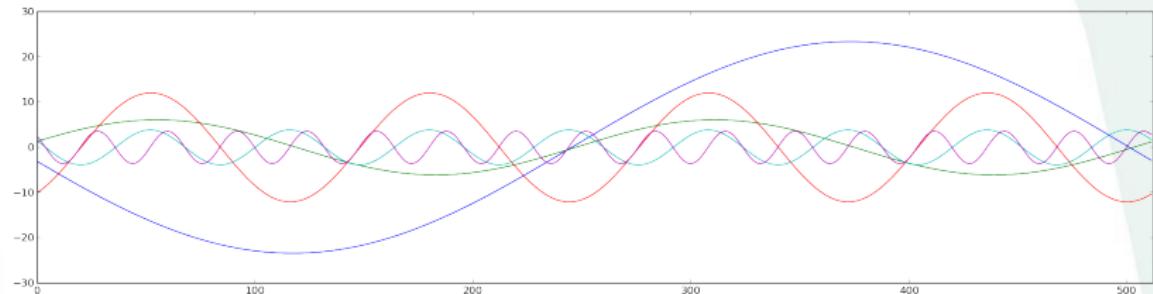
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

Intuição

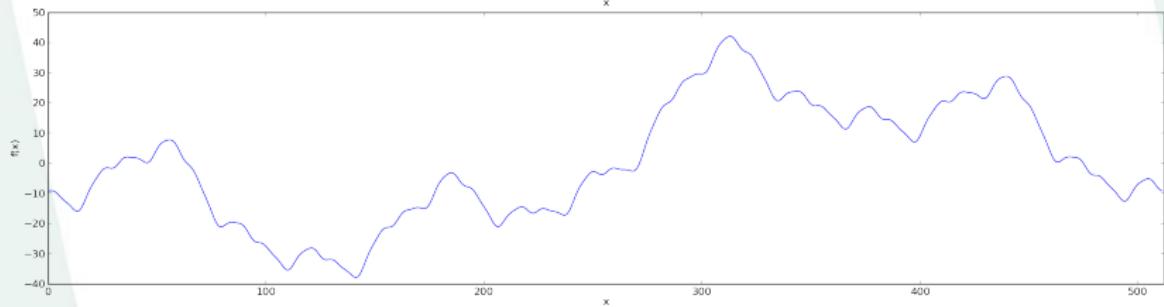
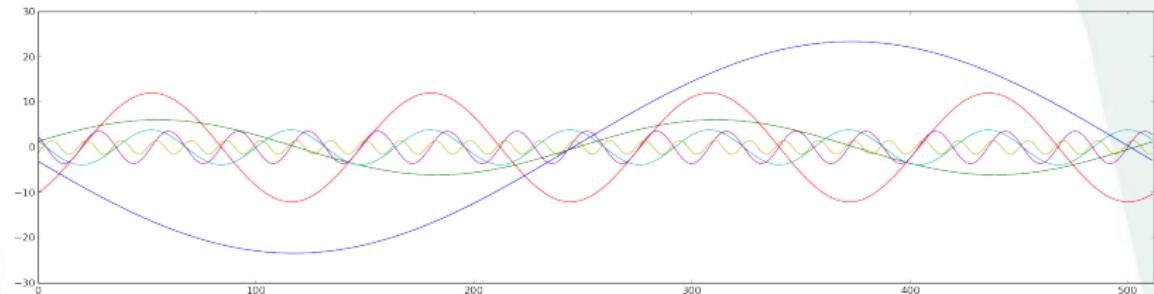
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

Intuição

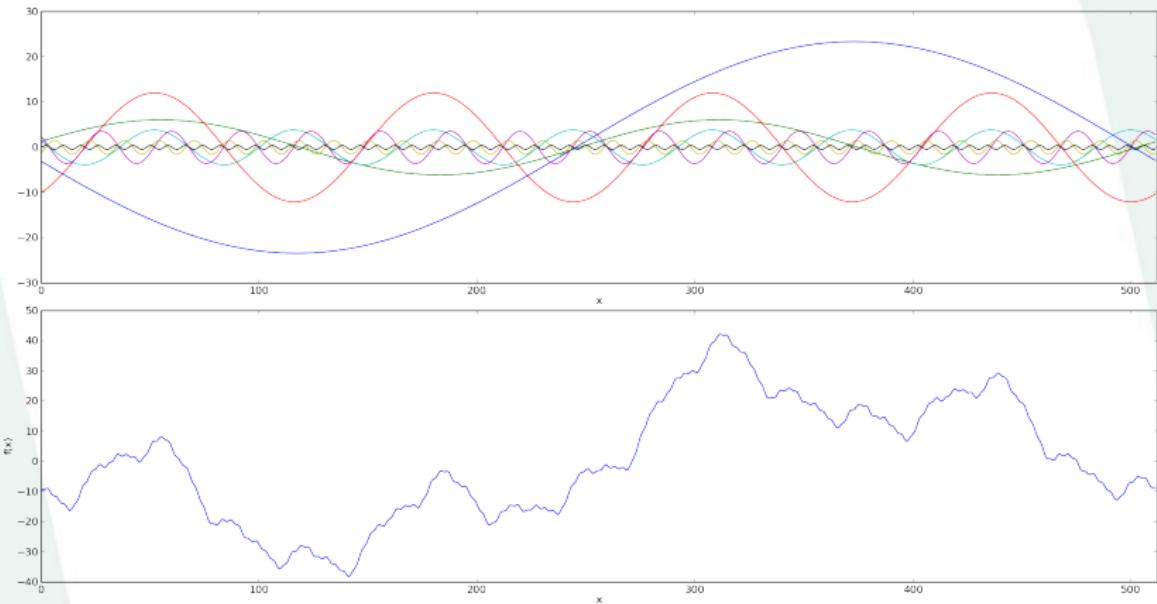
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

Intuição

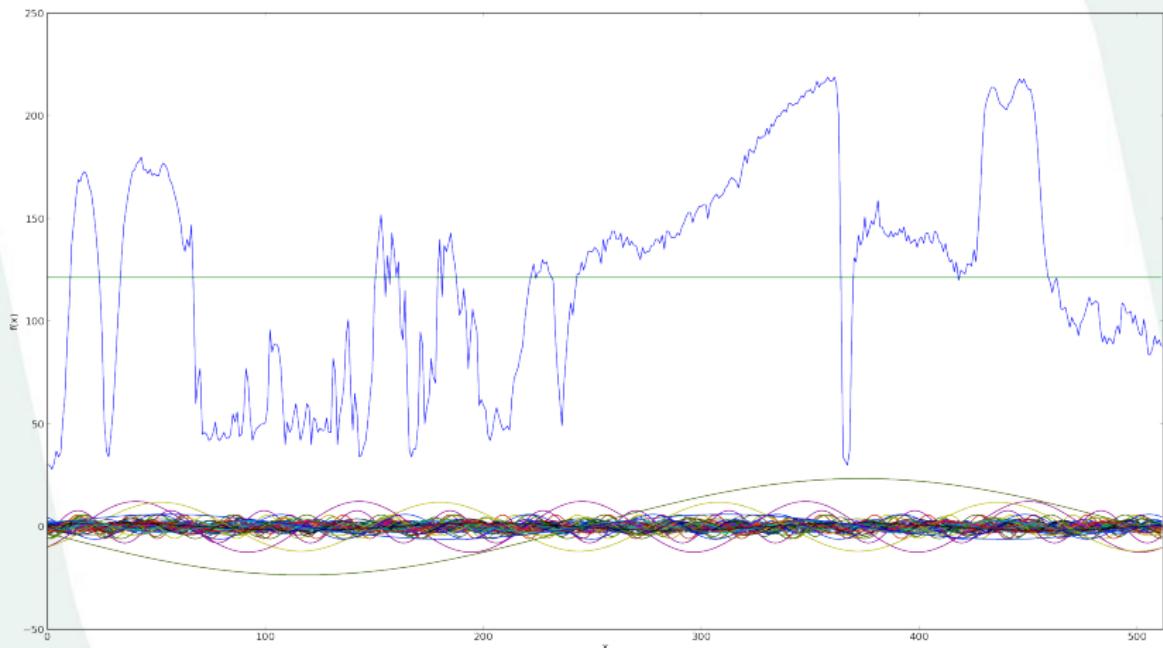
<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

Intuição

<http://goo.gl/MN1DJ>



# Transformada Discreta de Fourier

## Exemplo de utilização

### dft.py

```
11 # Expande a imagem para um tamanho otimo
12 M, N = image.shape
13 Mpad = cv2.getOptimalDFTSize(M)
14 Npad = cv2.getOptimalDFTSize(N)
15 padded = cv2.copyMakeBorder(image, 0, Mpad - M, 0, Npad - N,
16                             cv2.BORDER_CONSTANT)
17
18 # Aloca espaço tanto para valores reais como imaginarios
19 padded = np.array(padded, dtype=np.float32)
20 complexI = cv2.merge([padded, np.zeros((Mpad, Npad), dtype=
21                      np.float32)])
22
23 # Computa a Transformada Discreta de Fourier
24 complexI = cv2.dft(complexI)
25
26 # Para exibicao, computa a magnitude
27 re, im = cv2.split(complexI)
28 mag = np.sqrt(re**2 + im**2)
```

# Transformada Discreta de Fourier

## Exemplo de utilização

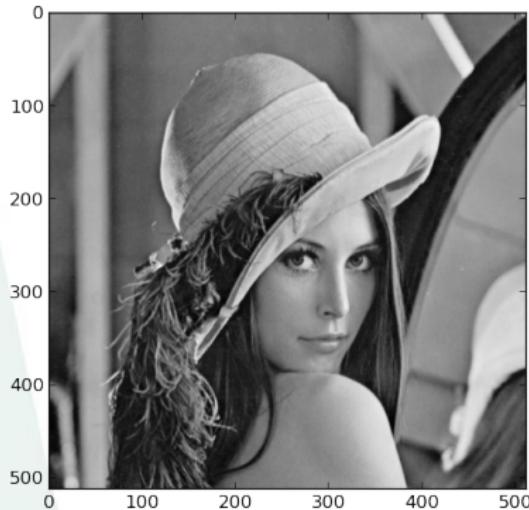
### dft.py

```
27  
28     # Utiliza escala logaritmica para apresentar os resultados  
29     lmag = np.log(1.0 + mag)  
30     pylab.imshow(lmag, cmap=pylab.cm.gray)  
31     pylab.show()
```

# Transformada Discreta de Fourier

Exemplo de utilização

$f(x, y)$



$\log(|F(u, v)| + 1)$

