Disciplina:

Análise de Imagem e Visão Computacional

Professor: Octavio Santana

PUC Minas

Aula 2:

Processamento e Extração de Caractéristicas de imagens

Professor: Octavio Santana

PUC Minas

Objetivos da Aula

Ao final desta aula, os alunos serão capazes de:

1. Aplicar transformações geométricas em imagem.
2. Aplicar operações aritméticas e operações bit a bit em imagens.
3. Filtros e Suavização em imagens.
4. Transformações Morfológicas.

PUC Minas

Introdução ao Processamento de Imagens

Transformações Geométricas em Imagens no OpenCV

As transformações geométricas são operações que modificam a posição, escala, orientação ou perspectiva de uma imagem. No OpenCV, essas transformações são realizadas principalmente por duas funções:

- *cv2.warpAffine()*: Usa uma matriz de transformação 2×3 para realizar translações, rotações e transformações afins.
- *cv2.warpPerspective()*: Usa uma matriz de transformação 3×3 para realizar transformações de perspectiva.

As fórmulas dessas transformações são:

- Transformação Afim (*cv2.warpAffine()*):

$$dst(x, y) = src(M11x + M12y + M13, M21x + M22y + M23)$$

- Transformação de Perspectiva (*cv2.warpPerspective()*):

$$dst(x, y) = src((M11x + M12y + M13)/(M31x + M32y + M33), (M21x + M22y + M23)/(M31xM32y + M33))$$

PUC Minas

Redimensionamento de Imagens (Scaling)

O redimensionamento altera o tamanho da imagem, aumentando ou reduzindo suas dimensões. Isso é feito com a função *cv2.resize()*.

- Redimensionamento para um tamanho específico
- Redimensionamento proporcional (fatores fx e fy)

```
resized_image = cv2.resize(image, (largura * 2, altura * 2), interpolation=cv2.INTER_LINEAR)
```

```
dst_image = cv2.resize(image, None, fx=0.5, fy=0.5, interpolation=cv2.INTER_AREA)
```

Métodos de Interpolação

O OpenCV oferece diferentes métodos para interpolação ao redimensionar imagens.

- *cv2.INTER_NEAREST* → Vizinho mais próximo (mais rápido, menor qualidade)
- *cv2.INTER_LINEAR* → Bilinear (padrão para redução).
- *cv2.INTER_AREA* → Aproximação por área (melhor para reduzir imagens).
- *cv2.INTER_CUBIC* → Bicúbica (melhor para aumentar imagens, mais lento).
- *cv2.INTER_LANCZOS4* → Interpolação senoidal (alta qualidade, mais lento).

Recomendação

- Para aumentar a imagem → *cv2.INTER_CUBIC* ou *cv2.INTER_LINEAR*
- Para reduzir a imagem → *cv2.INTER_AREA*

Translação de Imagem (Translation)

A translação desloca a imagem em uma direção específica (horizontal ou vertical). Isso é feito multiplicando a imagem por uma matriz de translação *M.*

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

Onde:

- *tx* é o deslocamento horizontal
- ty é o deslocamento vertical.

Criando a Matriz de Translação

Podemos criar a matriz M usando Numpy

```
M = np.float32([[1, 0, x], [0, 1, y]])
```

Depois aplicamos a translação com *cv2.warpAffine()*

```
dst_image = cv2.warpAffine(image, M, (largura, altura))
```

Rotação de Imagens

A rotação da uma imagem é feita utilizando a função *cv2.getRotationMatrix2D()*, que gera uma matriz 2×3 de transformação. Essa matriz define o ângulo de rotação (em graus), o ponto central da rotação e um fator de escala.

- Ângulos positivos ⟶ Rotação anti-horária
- Ângulos negativos ⟶ Rotação horária

Exemplo: Rotacionando uma imagem em 180º sem alterar a escala

```
height, width = image.shape[:2]
M = cv2.getRotationMatrix2D((width / 2.0, height / 2.0), 180, 1)
dst_image = cv2.warpAffine(image, M, (width, height))
```

A matriz *M* define a rotação em torno do centro da imagem com um fator de escala 1 (sem redimensionamento). A transformação é então aplicada com *cv2.warpAffine()*.

Transformação Afim (Affine Transformation)

Uma transformação afim altera a imagem mantendo linhas retas e paralelas intactas, mas não preserva ângulos e distâncias.

A matriz de transformação 2×3 é gerada pela função *cv2.getAffineTransform()*, a partir de três pontos de referência na imagem original e seus respectivos novos locais na imagem transformada.

Exemplo: Aplicando uma transformação afim

```
pts_1 = np.float32([[135, 45], [385, 45], [135, 230]])
pts_2 = np.float32([[135, 45], [385, 45], [150, 230]])
M = cv2.getAffineTransform(pts_1, pts_2)
dst_image = cv2.warpAffine(image, M, (width, height))
```

Aqui, três pontos da imagem original (*pts_1*) são mapeados para novos pontos (*pts_2*), gerando a matriz *M*, que é então aplicada à imagem.

Transformação de Perspectiva

A transformação de perspectiva altera a projeção da imagem, corrigindo distorções ou simulando um novo ponto de vista. A matriz de transformação 3×3 é gerada com *cv2.getPerspectiveTransform()* a partir de quatro pontos de referência.

Exemplo: Aplicando a perspectiva de uma imagem

```
pts_1 = np.float32([[450, 65], [517, 65], [431, 164], [552, 164]])
pts_2 = np.float32([[0, 0], [300, 0], [0, 300], [300, 300]])
M = cv2.getPerspectiveTransform(pts_1, pts_2)
dst_image = cv2.warpPerspective(image, M, (300, 300))
```

Aqui, quatro pontos da imagem original são mapeados para uma nova posição, criando uma transformação que corrige a perspectiva.

Operações aritméticas em imagens

- Permitem combinar ou modificar imagens por meio de soma, subtração, multiplicação e divisão de pixels.
- Para operações como adição e mistura, as imagens devem ter a mesma dimensão e tipo (ex.: 8 bits).
- Os valores de pixel variam de 0 a 255, o que pode causar overflow ao ultrapassar esses limites.

Aritmética de Saturação

- Tipo de operação aritmética em que os valores resultantes são limitados a um intervalo fixo, evitando que saiam desse intervalo.
- No caso de uma imagem de 8 bits, os valores de pixel variam de 0 a 255.
- Para armazenar um valor de r (resultado de uma operação), aplica-se a equação:

$$I(x, y) = \min(\max(\text{round}(r), 0), 255)$$

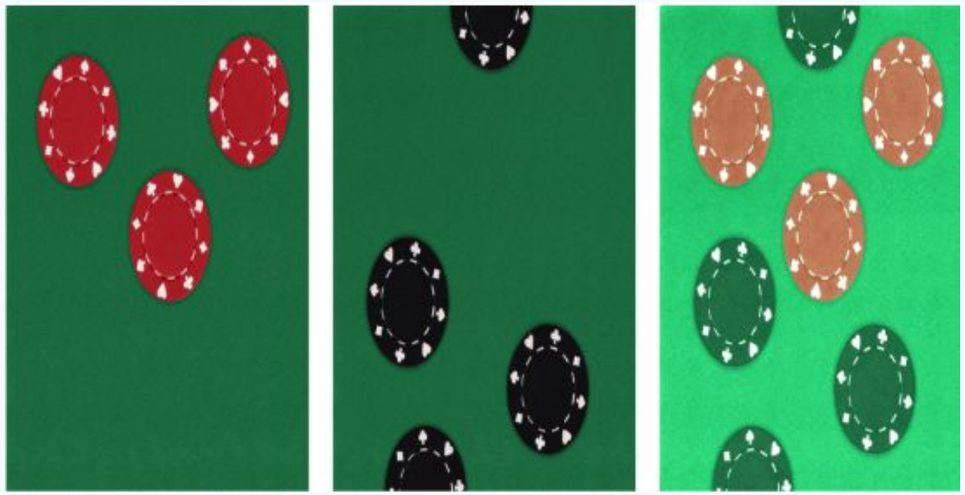
  - round(r) arredonda o valor de r.
  - max(round(r), 0) garante que o valor mínimo não seja menor que 0.
  - min(..., 255) garante que o valor máximo não ultrapasse 255.

PUC Minas

Operação de adição

- Soma os valores dos pixels de duas imagens, resultando em uma nova imagem.
- Pode causar saturação de pixels, tornando regiões muito claras ou distorcidas.
- Pode ser usada para ajustar o brilho e contraste ao adicionar valores constantes.

```
imagem = cv2.add(imagemFichasVermelhas, imagemFichasPretas)
```



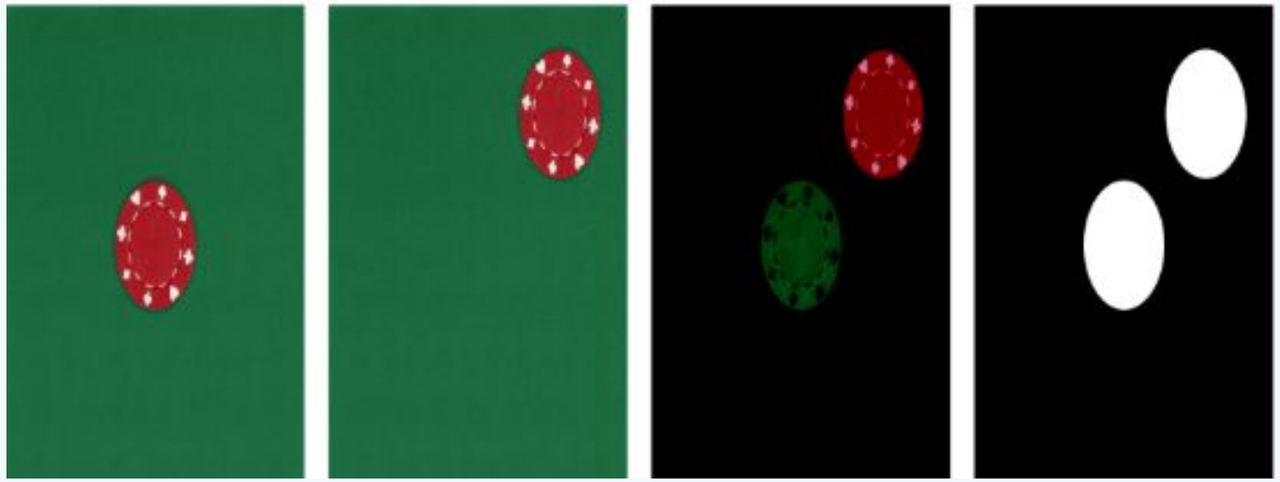
```
imagem = cv2.add(imagem, 40)
```

```
imagem = cv2.add(imagem, -40)
```

Operação de subtração

- Destaca as diferenças entre duas imagens, útil para detectar movimentos e mudanças.
- Aplicável em processamento de vídeo para rastreamento de objetos.
- Pode ser combinada com binarização para segmentar objetos em movimento.
- Exemplo de código:

```
imagem = cv2.subtract(imagemFichaPosicao1, imagemFichaPosicao2)
```

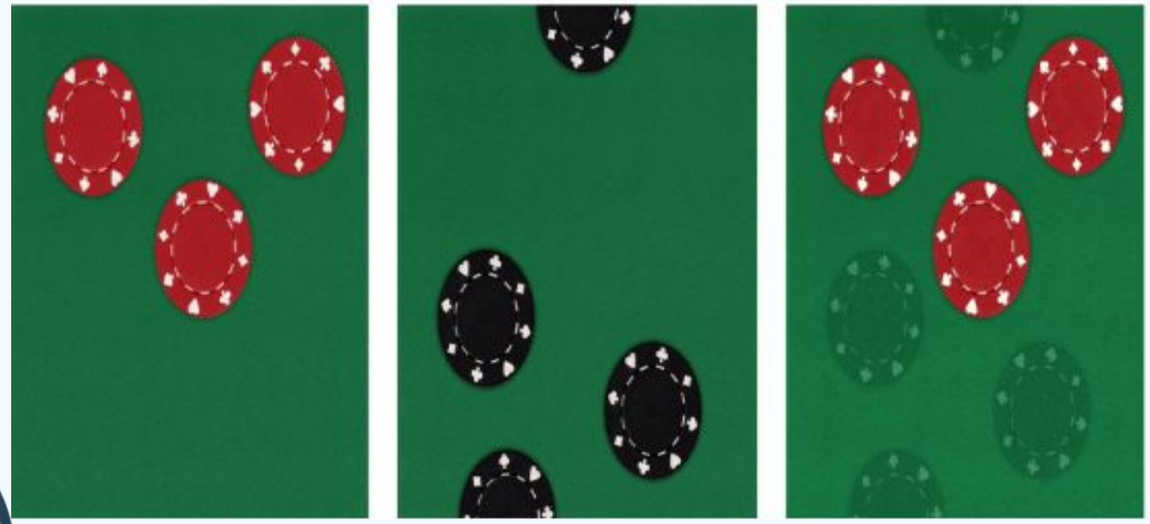


Operação de mistura

- Combina duas imagens com pesos diferentes, criando um efeito de transparência ou fusão.
- São 5 parâmetros obrigatórios:
  - src1 e src2: imagens a serem combinadas.
  - alpha e beta: pesos das imagens (valores entre 0.0 e 1.0).
  - gamma: ajuste opcional de brilho.
- Exemplo:

```
imagem = cv2.addWeighted(src1, alpha, src2, beta, gamma)
```

```
imagem = cv2.addWeighted(imagemFichasPretas, 0.2, imagemFichasVermelhas, 1.0, 0)
```



Operações aritméticas em imagens

Operação de multiplicação (*cv2.multiply()*)

- Multiplica os valores dos pixels de duas imagens, útil para realçar ou reduzir intensidades.
- Também pode ser usada para multiplicar uma imagem por um número escalar.

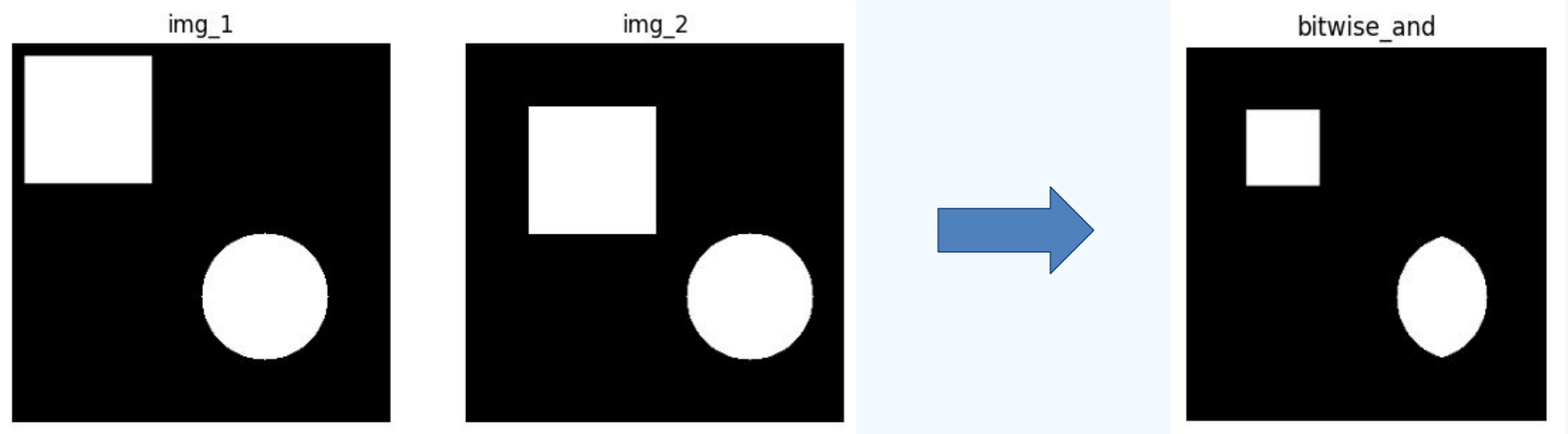
Operação de divisão (*cv2.divide()*)

- Divide os pixels de uma imagem pelos pixels de outra.
- Pode ser usada para normalizar imagens ou verificar se são idênticas.
- Menos eficiente que a subtração para detecção de diferenças.

Operações bit a bit

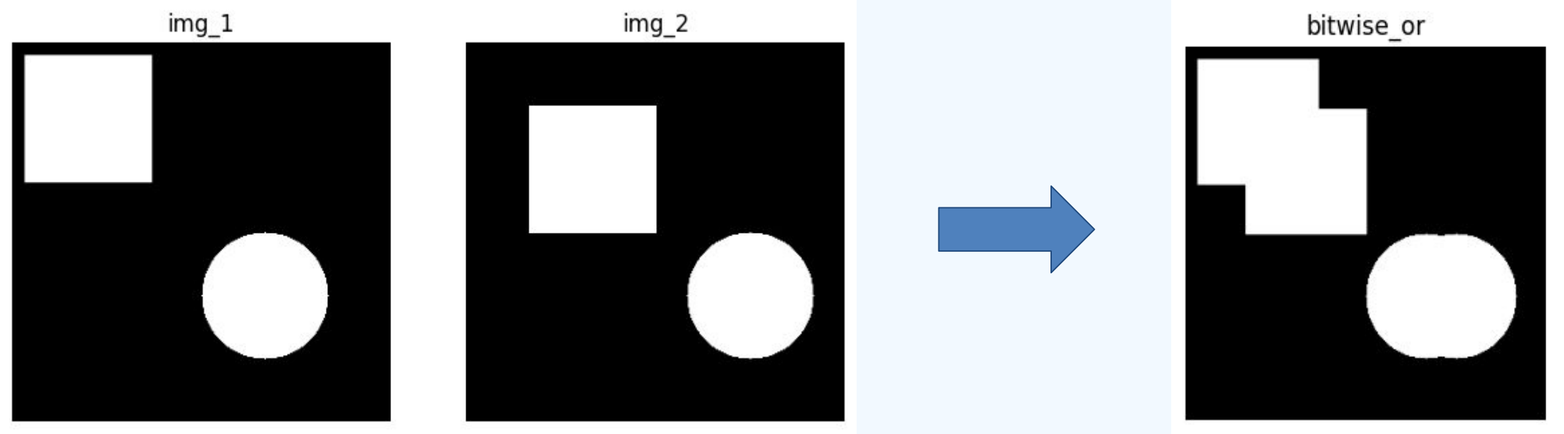
- Operações bit a bit são operações realizadas diretamente nos bits que compõem os valores dos pixels de uma imagem, permitindo manipulações rápidas e eficientes.
- OpenCV fornece quatro operações principais de manipulação bit a bit: AND, OR, XOR e NOT.

- Bitwise AND (*cv2.bitwise_and(img_1, img_2)*)
  - Retorna 1 apenas onde ambos os bits correspondentes nas imagens de entrada são 1.
  - Útil para mascarar regiões de interesse, permitindo que apenas os pixels compartilhados entre as duas imagens sejam mantidos.



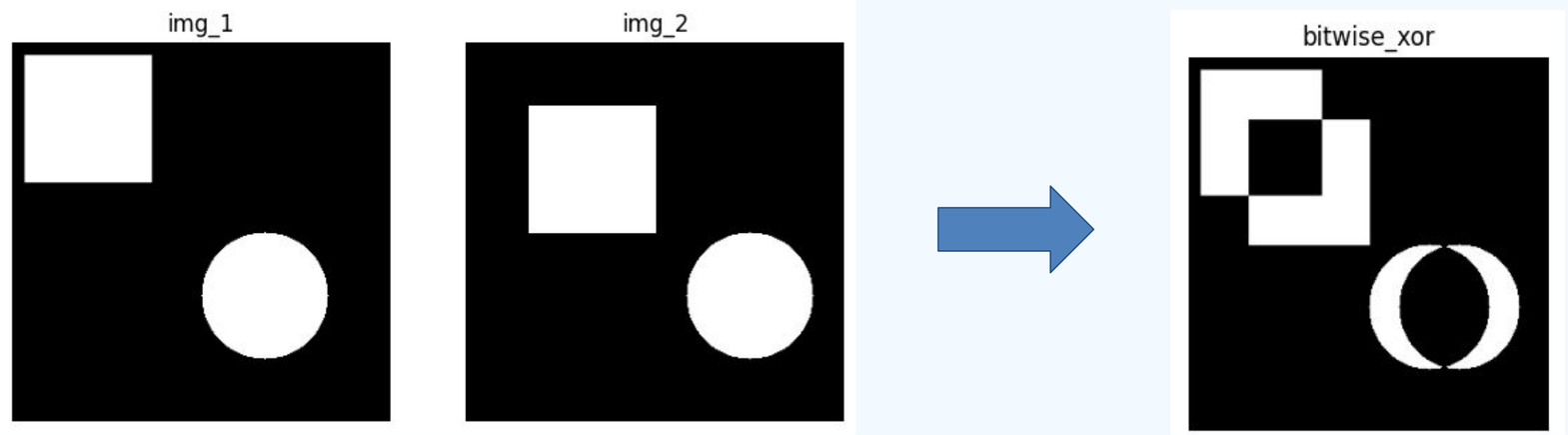
Operações bit a bit

- Bitwise OR (*cv2.bitwise_or(img_1, img_2)*)
  - Retorna 1 onde pelo menos um dos bits correspondentes nas imagens de entrada são 1.
  - Útil para combinar imagens ou máscaras.



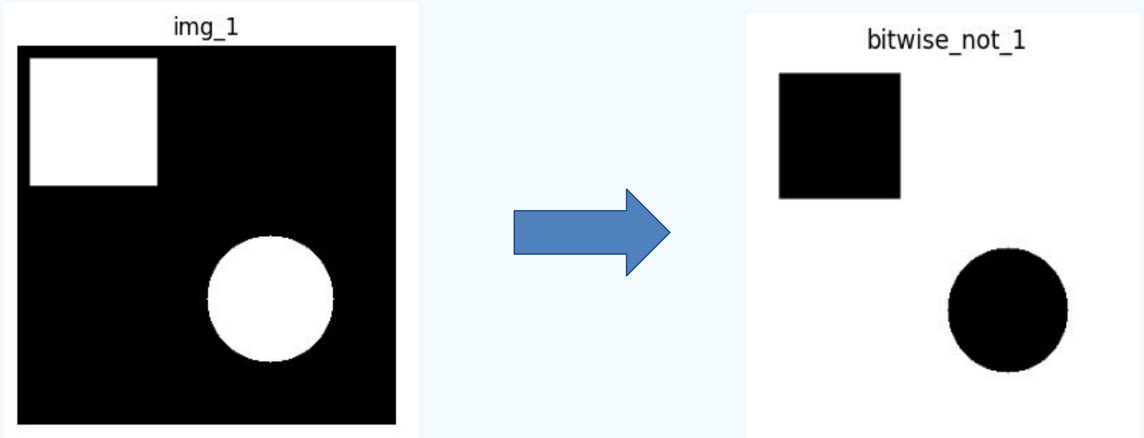
Operações bit a bit

- Bitwise XOR (*cv2.bitwise_xor(img_1, img_2)*)
  - Retorna 1 onde os bits correspondentes nas imagens de entrada são diferentes (um é 1 e o outro é 0).
  - Útil para detectar diferenças entre duas imagens.



Operações bit a bit

- Bitwise NOT (*cv2.bitwise_not(img_1)*)
  - Inverte os bits da imagem: trasforma pixels brancos (255) em pretos (0) e vice-versa.
  - Útil para inverter máscaras binárias.



PUC Minas

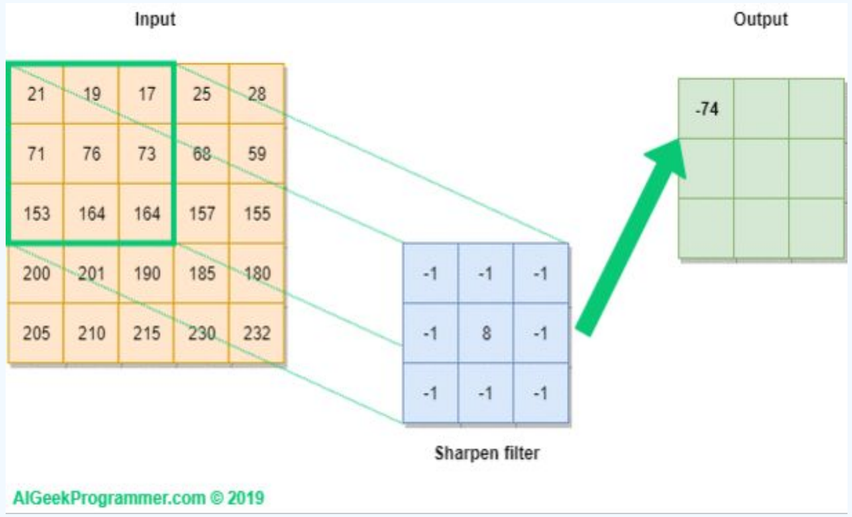
Filtros e Suavização de Imagens no OpenCV

PUC Minas

Conceito de Kernels e Operação de Filtragem

Um kernel, também chamado de filtro ou máscara, é uma pequena matriz de valores numéricos que é aplicada sobre uma imagem para modificar seus pixels de acordo com um determinado efeito.

A aplicação de um kernel a uma imagem é feita por um processo chamado convolução.



Passos da Convolução

1. O kernel percorre a imagem pixel a pixel.
2. Para cada posição, os valores do kernel são multiplicados pelos valores dos pixels vizinhos correspondentes.
3. Os produtos são somados e o resultado é armazenado no pixel central da nova imagem.
4. O processo se repete até toda a imagem ser processada.

💡 A convolução é a base de operações como borramento, realce de bordas e detecção de padrões.

Tipos de kernels e seus efeitos

Diferentes **kernels** produzem diferentes efeitos na imagem:

Tipo de kernel	Efeito	Método no OpenCV
Média (Blur)	Suaviza a imagem, reduzindo ruídos	*cv2.blur()* ou *cv2.boxFilter()*
Gaussiano	Suavização mais suave e natural	*cv2.GaussianBlur()*
Mediana	Remove ruídos do tipo “sal e pimenta”	*cv2.medianBlur()*
Bilateral	Reduz ruídos preservando as bordas	*cv2.bilateralFilter()*
Detecção de Bordas	Realça bordas e detalhes na imagem	*cv2.Laplacian()*, *cv2.Sobel()*, *cv2.Canny()*
Realce (Sharpening)	Aumentar a nitidez	*cv2.Filter2D()* com um kernel de nitidez

Aplicação de kernels arbitrários

A função *cv2.filter2D()* permite aplicar um kernel (ou filtro) personalizado à imagem, utilizando a operação de convolução.

A convolução consiste em percorrer a imagem pixel a pixel, aplicando uma máscara (kernel) sobre os valores vizinhos e computando um novo valor para o pixel central.

Exemplo: Aplicando um kernel de média 5x5

```
kernel_averaging_5_5 = np.ones((5, 5), np.float32) / 25
smooth_image_f2D = cv2.filter2D(image, -1, kernel_averaging_5_5)
```

- O que esse kernel faz?
  - Esse kernel de média substitui cada pixel pela média dos 25 pixels ao seu redor (matriz 5×5), resultando em um efeito de suavização.

Aplicação de kernels arbitrários

A função *cv2.filter2D()* permite aplicar um kernel (ou filtro) personalizado à imagem, utilizando a operação de convolução.

A convolução consiste em percorrer a imagem pixel a pixel, aplicando uma máscara (kernel) sobre os valores vizinhos e computando um novo valor para o pixel central.

Exemplo: Aplicando um kernel para realce de bordas

```
kernel_sharpening = np.array([[ 0, -1,  0],
                              [-1,  5, -1],
                              [ 0, -1,  0]])
sharpened_image = cv2.filter2D(image, -1, kernel_sharpening)
```

- O que esse kernel faz?
  - Esse kernel de realce de bordas acentua os contornos da imagem, tornando-a mais nítida.

Filtro de Média (Averaging Filter)

A filtragem por média é realizada aplicando um kernel de média sobre a imagem. Esse processo reduz variações bruscas de intensidade, suavizando a imagem e removendo ruídos.

No OpenCV utiliza-se as funções *cv2.blur()* e *cv2.boxFilter()*. Ambas as funções aplicam a média dos pixels dentro de uma janela (kernel) ao redor de cada pixel. Sendo que o pixel central da janela é atualizado com o valor médio

- *cv2.blur()* : Sempre usa um kernel normalizado (a soma dos elementos do kernel é igual a 1)
- *cv2.boxFilter() :* Pode ser usado com um kernel normalizado ou não (quando *normalize=False*)

Parâmetros Importantes:

- Tamanho do kernel (*ksize*):
  - Define a área sobre o qual a média será calculada
  - Um kernel maior resulta em um borramento mais intenso
- Âncor (*anchor*)
  - Define o ponto de referência do kernel ao aplicar a convolução
  - O padrão (-1,-1) posiciona a âncora no centro do kernel.

Filtro Gaussiano

Diferente do filtro de média simples, o filtro Gaussiano (*cv2.GaussianBlur()*) usa uma matriz gerada por uma função Gaussiana (distribuição normal). Isso faz com que os pixels mais próximos do centro tenham mais peso na média do que os mais distantes, resultando em um efeito mais natural.

Parâmetros Importantes:

- Tamanho do kernel (*ksize*):
  - Deve ser ímpar e positivo (ex.: 3x3, 5x5, 7x7)
  - Define a área de influência da suavização.
- Desvios Padrão (*sigmaX* e *sigmaY)*
  - *sigmaX*: Controla a dispersão do filtro na direção X.
  - *sigmaY*: Controla a dispersão na direção Y.
  - Se apenas *sigmaX* for especificado, *sigmaY* assumirá o mesmo valor.
  - Se ambos forem zero (0,0), os valores serão calculados automaticamente com base no tamanho do kernel.

Filtro da Mediana (Median Filtering)

O filtro da mediana (*cv2.medianBlur()*) funciona de maneira diferente: ele substitui o pixel central pela mediana dos valores dos vizinhos. Isso faz com que seja extremamente eficaz para remover ruídos do tipo "sal e pimenta" (pixels muito claros ou escuros).

Quando usar o filtro da mediana?

- Para remover ruído “sal e pimenta”
- Quanto quiser manter bordas mais preservadas do que no filtro de média

Obs.: O tamanho do kernel deve ser ímpar e maior que 1.

Filtro Bilateral (Bilateral Filtering)

Os filtros anteriores borram a imagem inteira, inclusive as bordas. O filtro bilateral (*cv2.bilateralFilter()*) resolve esse problema ao aplicar um peso maior para pixels próximos e com cores similares, preservando bordas enquanto suaviza outras regiões.

A Filtragem Bilateral usa dois filtros gaussianos

- Filtro Gaussiano Espacial: Considera apenas pixels próximos ao ponto central
- Filtro Gaussiano baseado na diferença de intensidade: Dá mais peso a pixels com intensidade semelhantes ao pixel central, garantindo que bordas sejam preservadas.

Parâmetros da função

```
smooth_image_bf = cv2.bilateralFilter(image, d=5, sigmaColor=10, sigmaSpace=10)
```

- d: Tamanho da vizinhança para considerar na filtragem.
- sigmaColor: Controla a influência da diferença de intensidade. Valores maiores resultam em uma filtragem mais forte entre pixels com diferenças de intensidade maiores.
- sigmaSpace: Define o alcance da vizinhança espacial. Quanto maior, mais pixels ao redor do ponto central serão considerados a filtragem.

Realce de bordas

Vimos filtros para tratamento de ruído em imagens. Porém, o procedimento de filtragem em imagens possui duas outras aplicações tão importantes quanto o tratamento de ruído: **extração de características, e realce de arestas, bordas ou contornos em imagens.**

PUC Minas

Operador de Sobel

- Filtro não linear para realce de contornos, destacando linhas verticais e horizontais.
- A filtragem pode ser feita na direção horizontal ou vertical.
- Produz bordas mais grossas comparadas a outros métodos.
- Implementado na OpenCV através da função *cv2.Sobel()*.
- Parâmetros principais:
  - *cv2.CV_8U*: define o tipo de variável para armazenar os pixels.
  - Direção da filtragem: (dx=1, dy=0) para horizontal, (dx=0, dy=1) para vertical.
  - ksize: tamanho da máscara de filtragem.

```python
import cv2

imagem = cv2.imread("estacionamento.jpeg", 0)

sobelx = cv2.Sobel(imagem, cv2.CV_8U, 1, 0, ksize = 3)
sobely = cv2.Sobel(imagem, cv2.CV_8U, 0, 1, ksize = 3)

cv2.imshow("Original", imagem)
cv2.imshow("Sobel X", sobelx)
cv2.imshow("Sobel Y", sobely)

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

PUC Minas

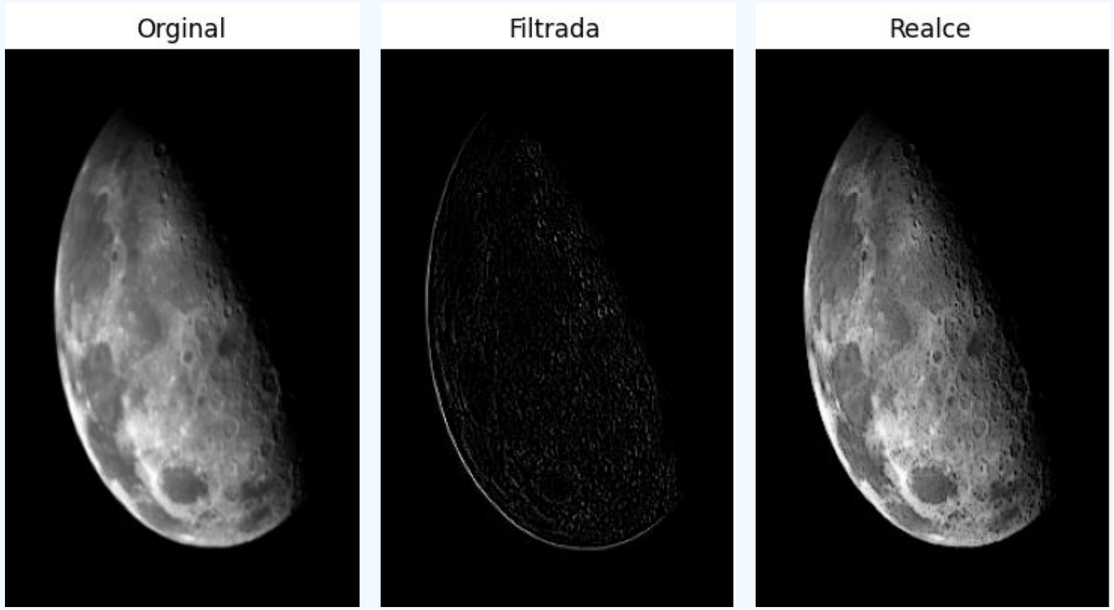
Operador Laplaciano

- Filtro não linear para realce de contornos, destacando linhas verticais e horizontais.
- Popular para realce de bordas, utilizando uma máscara de ordem 3.
- Calcula a média ponderada dos pixels vizinhos e eleva ao quadrado.
- Produz bordas mais finas do que o Sobel, mas é altamente sensível a ruído.
- Implementado na OpenCV através da função *cv2.Laplacian()*.
- Aplicação comum de filtro passa-baixas (ex: gaussiano) antes do laplaciano para reduzir ruídos.

Aguçamento de Bordas

- Feito pela subtração da imagem original com sua versão processada pelo Laplaciano.
- Realça detalhes finos, podendo gerar pixels negativos.
- Para evitar pixels negativos, utiliza-se *cv2.subtract()*, que trata esses valores automaticamente

```
import cv2

img_original = cv2.imread('lua.jpg', 0)
img_filtrada = cv2.Laplacian(img_original, cv2.CV_8U)
img_realce = cv2.subtract(img_original, img_filtrada)
```



PUC Minas

Filtro Máscara de Desaguçamento

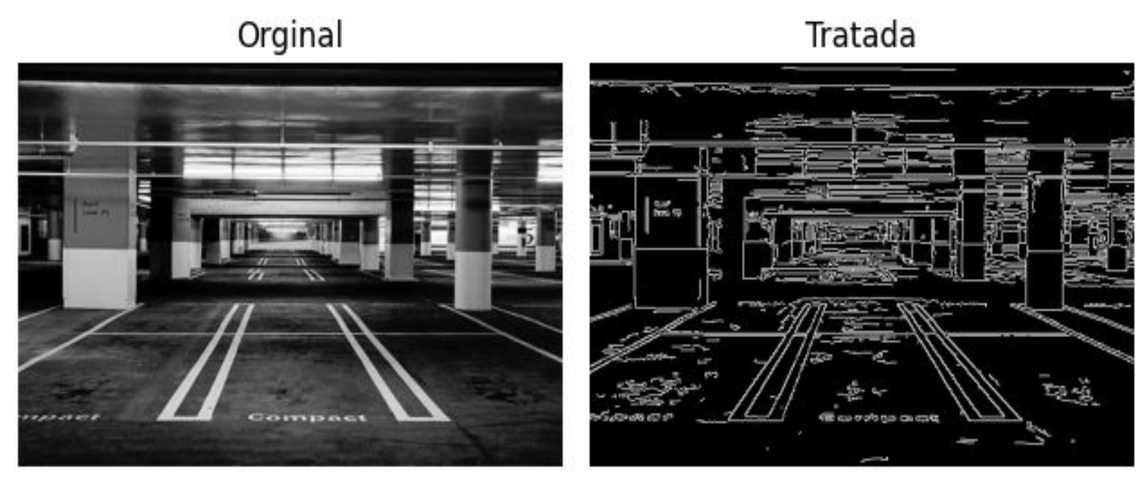
- Técnica baseada na subtração da imagem original por uma versão suavizada dela mesma.
- A suavização é realizada por um filtro passa-baixas (ex: Gaussiano).
- Após a subtração, a imagem com bordas realçadas é somada à original.
- Implementado na OpenCV utilizando *cv2.GaussianBlur()*, *cv2.subtract()* e *cv2.add()*.

```python
import cv2

img_original = cv2.imread("radiografia.jpg", 0)
img_suavizada = cv2.GaussianBlur(img_original, (13, 13), 3)
img_detalhes = 3 * cv2.subtract(img_original, img_suavizada)
img_realcada = cv2.add(img_original, img_detalhes)
```

Detector de Bordas Canny

- Um dos mais eficientes métodos para detecção de bordas.
- Desenvolvido por John Canny, baseado em três princípios:
  - Detectar todas as bordas possíveis.
  - Manter as bordas próximas às reais.
  - Evitar a criação de bordas falsas.
- Utiliza filtro gaussiano e operador de Sobel.
- Implementado na OpenCV com a função *cv2.Canny()*, que requer:
  - Imagem em tons de cinza.
  - Dois limiares para controle da intensidade de detecção.
  - Valores mais altos nos limiares resultam em menos bordas detectadas.

```
import cv2

img_original = cv2.imread("estacionamento.png", 0)
img_tratada = cv2.Canny(img_original, 100, 200)
```



PUC Minas

Transformações Morfológicas

Transformações Morfológicas

- Operações aplicadas principalmente em imagens binárias, baseadas na forma dos objetos presentes na imagem.
- Utilizam um elemento estruturante (kernel) para definir a natureza da transformação.
- As operações básicas são dilatação e erosão.
- Operações derivadas incluem abertura e fechamento.
- Outras transformações utilizam a diferença entre as operações básicas.

Elemento Estruturante (kernel)

- Esse elemento pode ser visto como uma imagem binária, menor que a imagem original, armazenado geralmente em uma matriz quadrada. Ele é a base para que qualquer operação morfológica seja executada.
- Assim como os filtros espaciais, a matriz que representa o elemento estruturante percorrerá toda a imagem a ser tratada. Nesse processo, uma determinada operação morfológica será realizada pixel a pixel na imagem, alterando o valor de cada um deles para seguir o padrão definido por esse elemento.
- Tipos disponíveis no OpenCV:
  - Retangular (cv2.MORPH_RECT) – adequado para estruturas regulares.
  - Elíptico (cv2.MORPH_ELLIPSE) – útil para formas arredondadas.
  - Cruz (cv2.MORPH_CROSS) – enfatiza conexões em cruz.

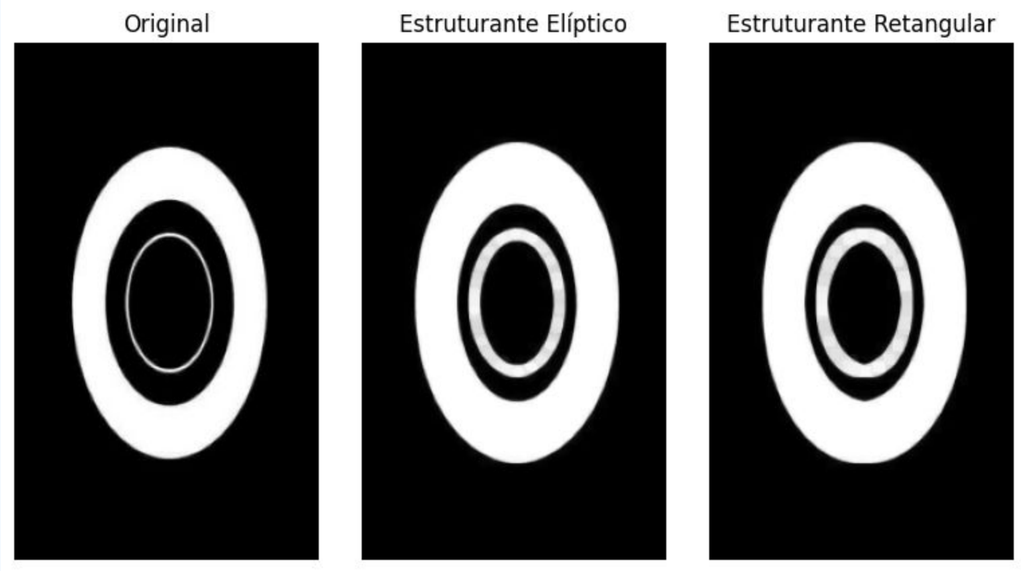
```
# Retangular

>>> cv2.getStructuringElement(cv2.MORPH_RECT,(5,5))
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)
```

```
# Elíptico

>>> cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

```
# Cruz

>>> cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))
array([[0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

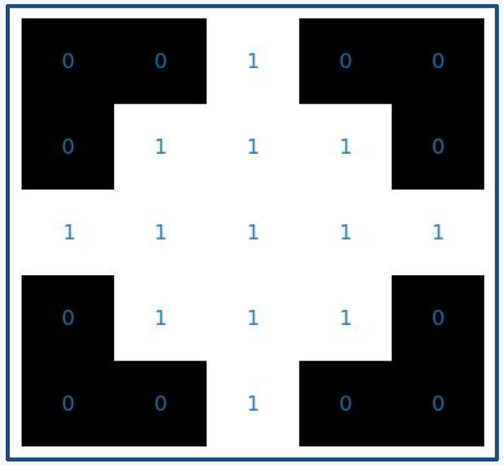
Elemento Estruturante (kernel)

A fim de exemplificar o uso desses diferentes elementos estruturantes, a figura adiante apresenta um objeto submetido a uma operação morfológica para dilatá-lo.



Em casos especiais, quando nenhum dos elementos estruturantes predefinidos apresentam bons resultados, podemos criar um elemento estruturante personalizado.

```
import cv2
import numpy as np

elementoEstruturante = np.matrix([
  [0, 0, 1, 0, 0],
  [0, 1, 1, 1, 0],
  [1, 1, 1, 1, 1],
  [0, 1, 1, 1, 0],
  [0, 0, 1, 0, 0]
], np.uint8)
```

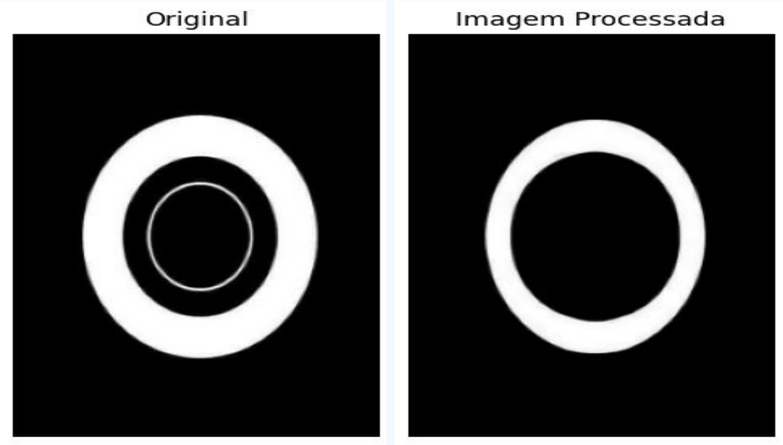


PUC Minas

Operações de erosão e dilatação

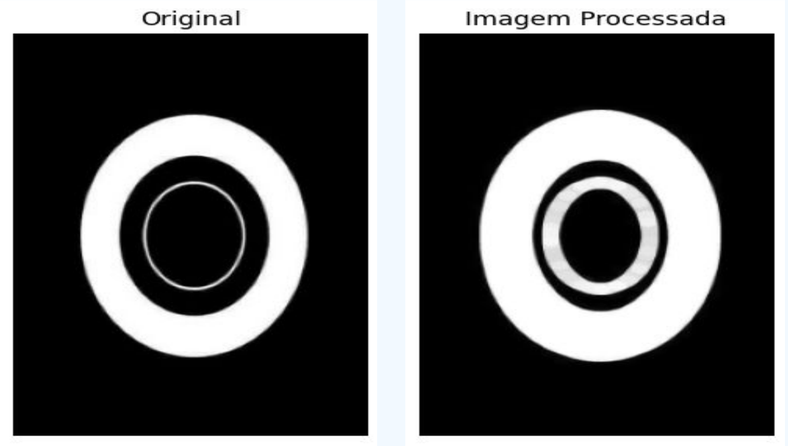
- Erosão (*cv2.erode*)
  - Reduz as regiões de primeiro plano (foreground) da imagem.
  - Objetos diminuem de tamanho, e buracos internos aumentam.

```
img = cv2.imread('circle_binario.jpg', 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
img_processada = cv2.erode(img, kernel, iterations=2)
```



- Dilatação (*cv2.dilate*)
  - Expande as regiões de primeiro plano (foreground) da imagem.
  - Aumenta o tamanho dos objetos enquanto reduz buracos internos

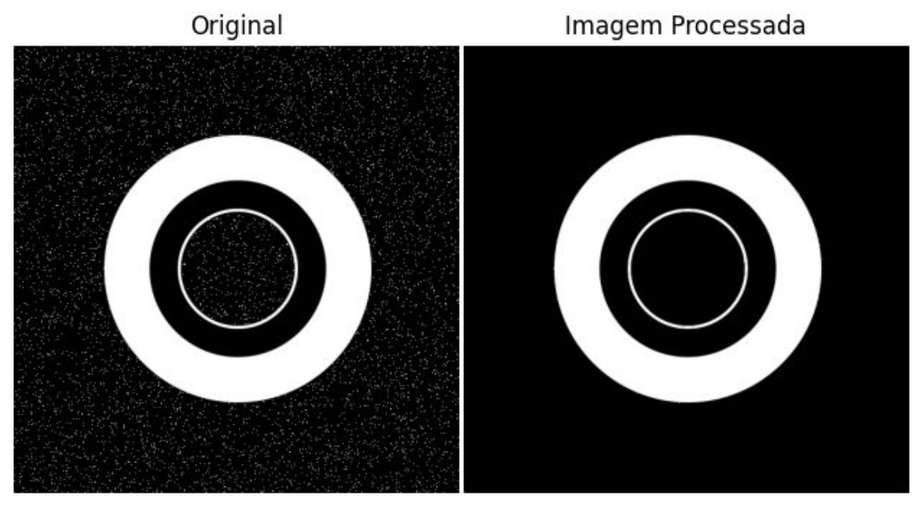
```
img = cv2.imread('circle_binario.jpg', 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
img_processada = cv2.dilate(img, kernel, iterations=2)
```



Operações de abertura e fechamento

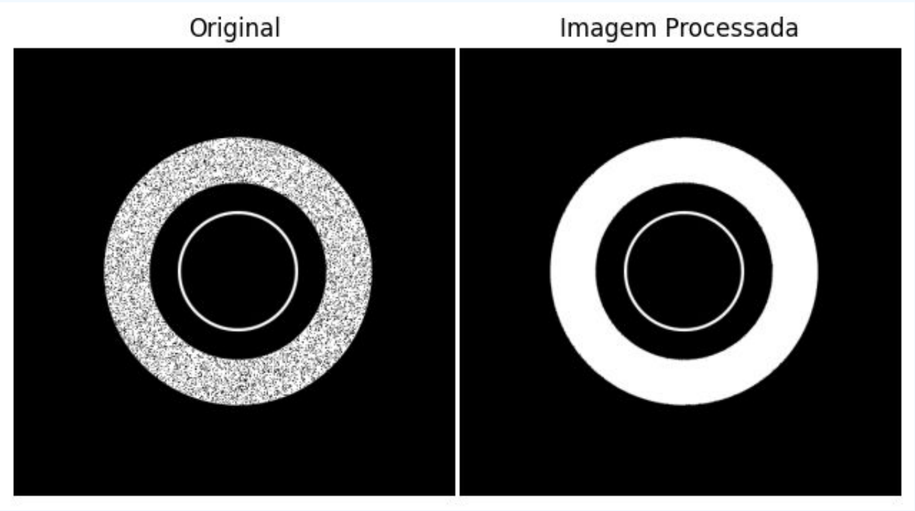
- Abertura (*cv2.morphologyEx* com *cv2.MORPH_OPEN*)
  - Consiste em erosão seguida de dilatação.
  - Remove ruídos pequenos como salt-and-pepper noise.

```
img = cv2.imread('circle_binario_ruido_1.png', 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
img_processada = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```



- Fechamento (*cv2.morphologyEx* com *cv2.MORPH_CLOSE*)
  - Consiste em dilatação seguida de erosão.
  - Preenche pequenos buracos na imagem.

```
img = cv2.imread("circle_binario_ruido_2.png", 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
img_processada = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

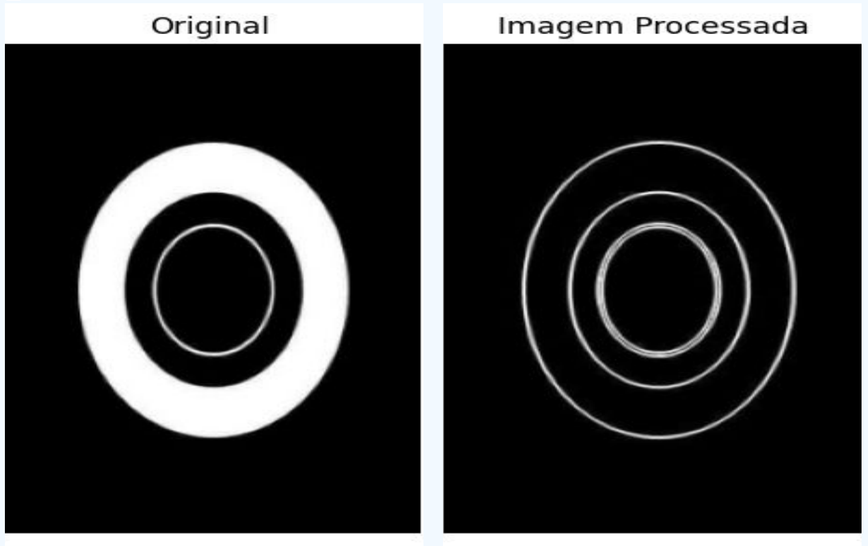


PUC Minas

Principais Operações Morfológicas no OpenCV

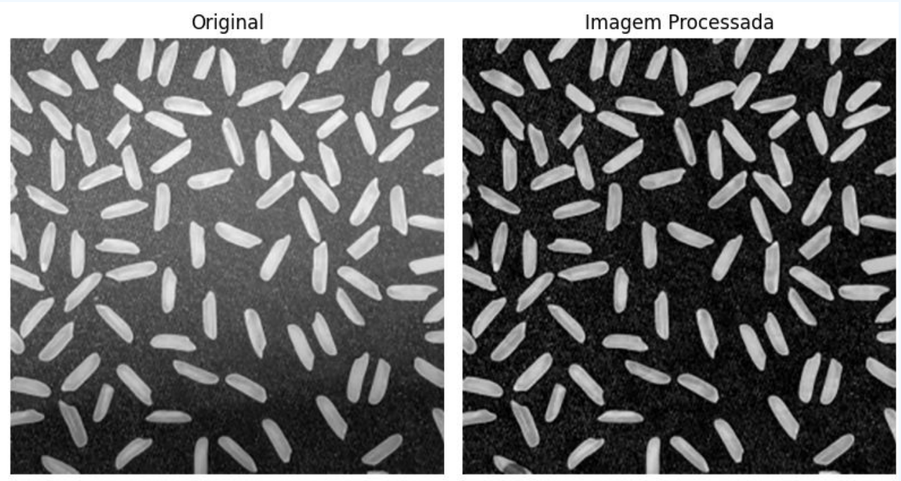
- Gradiente Morfológico (*cv2.morphologyEx* com *cv2.MORPH_GRADIENT*)
  - Diferença entre dilatação e erosão da imagem original.
  - Destaca bordas e contornos de objetos na imagem.

```
img = cv2.imread("circle_binario.jpg", 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
img_processada = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
```



- Top Hat (*cv2.morphologyEx* com *cv2.MORPH_TOPHAT*)
  - Diferença entre a imagem original e sua abertura.
  - Realça detalhes brilhantes em fundos escuros.

```
img = cv2.imread("arroz.png", 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (25,25))
img_processada = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel)
```

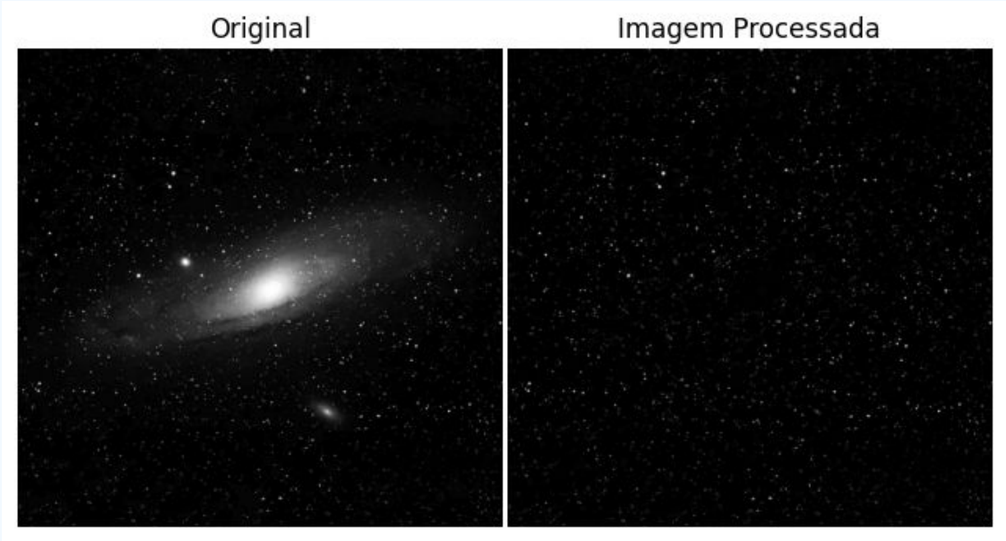


PUC Minas

Principais Operações Morfológicas no OpenCV

A operação Top Hat, quando aplicada com elementos estruturantes menores, permite suprimir grandes regiões na imagem.

```
img = cv2.imread("galaxia.png", 0)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
img_processada = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel)
```



Conclusão

Nesta aula, exploramos técnicas fundamentais para o processamento de imagens usando OpenCV e Python.

Agora os alunos são capazes de:

- Aplicar transformações geométricas em imagens, como rotação, translação e redimensionamento, permitindo ajustes precisos na composição visual.
- Realizar operações aritméticas e bit a bit, essenciais para combinar imagens, ajustar brilho, contraste e criar máscaras avançadas.
- Utilizar filtros e suavização para realçar características ou reduzir ruídos, melhorando a qualidade da imagem para análises posteriores.
- Executar transformações morfológicas, como erosão e dilatação, para manipulação de formas e estruturas em imagens binárias.