Disciplina:

Análise de Imagem e Visão Computacional

Professor: Octavio Santana

PUC Minas

Aula 3

Recuperação de Imagens com base no conteúdo visual

Professor: Octavio Santana

PUC Minas

Objetivos da Aula

Ao final desta aula, os alunos serão capazes de:

1. Utilizar técnicas de segmentação e detecção de contornos.
2. Extrair descritores de imagem para identificação de padrões e objetos.
3. Recuperação de imagens com base no conteúdo visual

Segmentação e Limiarização

Segmentação de objetos

- A segmentação de objetos em imagem é uma das principais etapas de um sistema baseado em Visão Computacional. Essa etapa consiste em separar somente a área que representa o objeto de interesse em uma nova imagem, excluindo também o segundo plano dessa região.
- O objeto a ser estudado é considerado o **primeiro plano da imagem**. Os pixels que não fazem parte de sua representação são denominados como **segundo plano**. A segmentação é o primeiro passo para que possamos extrair características do objeto.
- Somente com ele segmentado, é possível obter informações e detalhes que tornam possível classificá-lo.
- Veremos quatro métodos para segmentação de objetos em imagens:
  - Segmentação por binarização.
  - Segmentação por cor
  - Segmentação por bordas
  - Segmentação por movimento

Segmentação por binarização

A segmentação por binarização, também conhecida como aplicação de limiar de intensidade, é um dos métodos mais simples e eficazes para separar objetos de interesse do fundo em uma imagem. Esse processo consiste em converter uma imagem em tons de cinza para uma imagem binária, onde os pixels são classificados como objeto ou fundo com base em um limiar predefinido.

Binarização Global

O processo de binarização global usa um limiar fixo para classificar os pixels da imagem. Quando um pixel tem um valor superior ao limiar, ele é definido como branco (255); caso contrário, é definido como preto (0). Na biblioteca OpenCV, essa operação pode ser realizada com a função *cv2.threshold*, que recebe quatro parâmetros principais:

- Imagem de entrada: deve estar em tons de cinza.
- Valor do limiar: definido manualmente, geralmente por tentativa e erro.
- Valor máximo: intensidade atribuída aos pixels acima do limiar.
- Tipo de limiar: define se o objeto será representado na cor preta (THRESH_BINARY) ou branca (THRESH_BINARY_INV).

PUC Minas

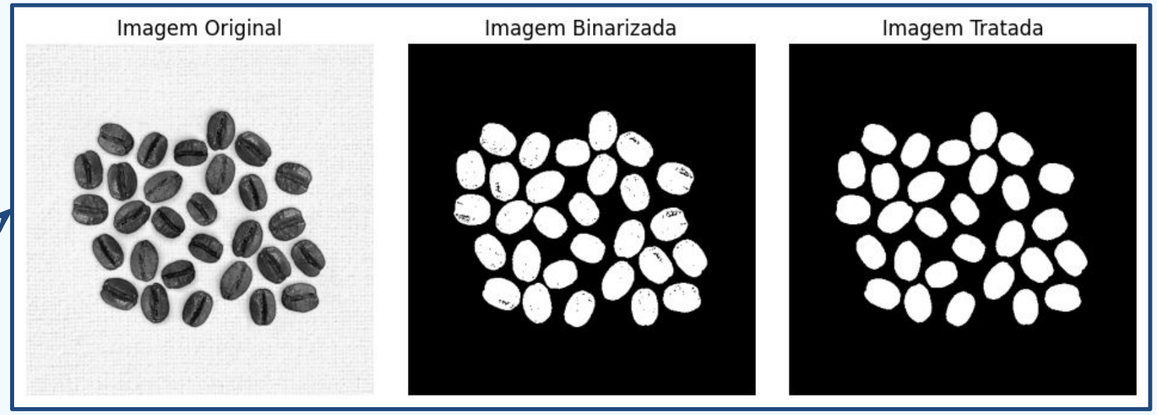
Segmentação por binarização

Binarização Global

```
img = cv2.imread("graos_de_cafe.png")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

th, img_binarizada = cv2.threshold(img_gray, 140, 255, cv2.THRESH_BINARY_INV)
```

Nessa figura, note também que as áreas que representam os grãos de café estão falhadas, possuindo pontos pretos quando deveriam ser completamente brancas. Essas falhas são provocadas pelo procedimento de binarização. Elas ocorrem quando pixels pertencentes ao objeto de interesse são representados por um valor inferior ao limiar.

Falhas como estas são consideradas ruídos na imagem, entretanto, podem ser facilmente tratadas com operações morfológicas.



Segmentação por binarização

Binarização Adaptativa

Quando a imagem possui iluminação desigual, a binarização adaptativa se torna uma alternativa melhor. Nesse método, o limiar é calculado localmente para diferentes regiões da imagem, garantindo uma segmentação mais precisa.

A função adaptiveThreshold da OpenCV permite realizar esse processo e requer seis parâmetros:

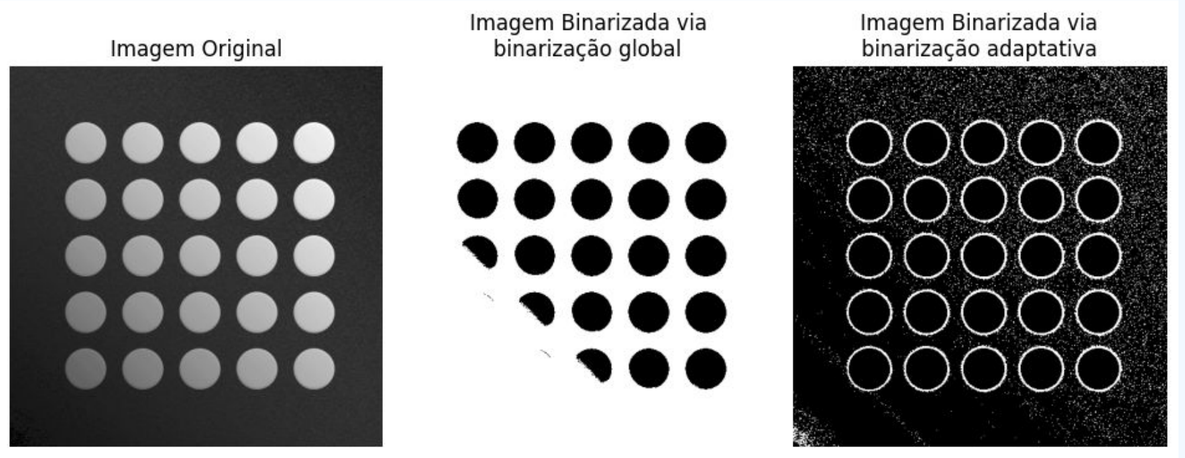
- Imagem de entrada
- Valor máximo da intensidade do pixel
- Método de cálculo do limiar (ADAPTIVE_THRESH_MEAN_C ou ADAPTIVE_THRESH_GAUSSIAN_C)
- Tipo de limiar (THRESH_BINARY ou THRESH_BINARY_INV)
- Tamanho da máscara (número ímpar que define a região analisada para o cálculo do limiar)
- Constante subtraída do limiar calculado

PUC Minas

Segmentação por binarização

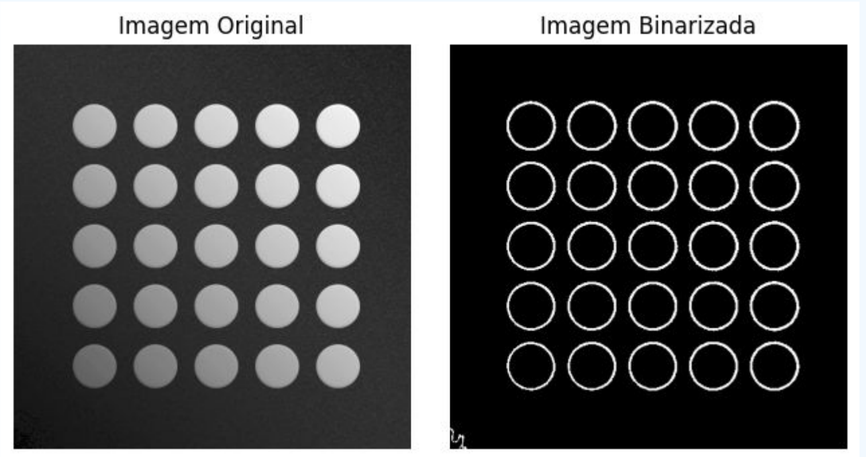
Binarização Adaptativa

```
img = cv2.imread('comprimidos.png', 0)

th, img_binarizada_global = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV)

img_binarizada_adaptativa = cv2.adaptiveThreshold(
    img, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY_INV, 11, 5
)
```



Desafio:

Implemente um código que carregue a imagem "comprimidos.jpeg", realize a binarização adaptativa e exiba o resultado. Para melhorar a qualidade da segmentação, utilize um filtro de mediana antes da binarização.



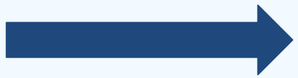
Segmentação por binarização

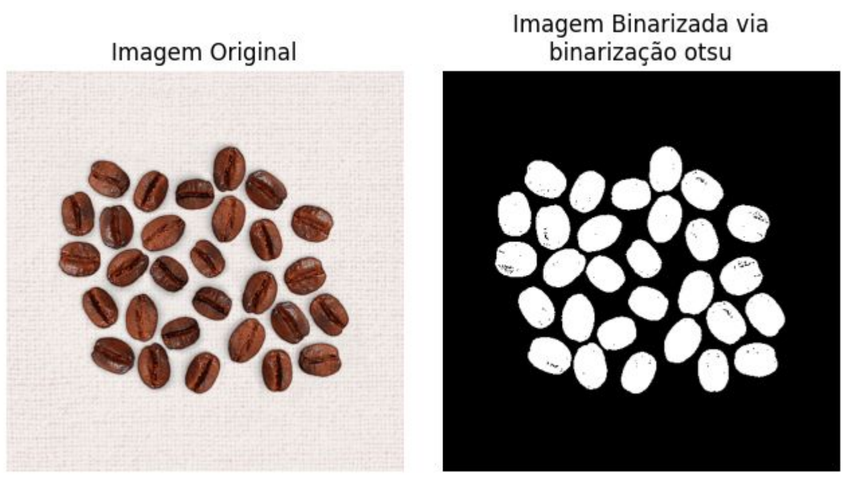
Binarização OTSU

O método de Otsu automatiza a escolha do limiar ideal, analisando o histograma da imagem e escolhendo um valor que maximize a separação entre os pixels do objeto e do fundo.

Para usar o método de Otsu com a função *cv2.threshold*, basta adicionar *cv2.THRESH_OTSU* ao tipo de limiar.

```
img = cv2.imread("graos_de_cafe.png")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

th, img_binarizada = cv2.threshold(
    img_gray, 0, 255,
    cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU
)
```





O limiar é calculado automaticamente pelo algoritmo de Otsu, reduzindo a necessidade de tentativa e erro.

Segmentação por cor

A segmentação por cor é uma técnica amplamente utilizada em processamento de imagens para identificar e separar regiões de interesse baseadas em suas cores. O espaço de cores HSV é frequentemente empregado nesse tipo de segmentação, pois representa a matiz (cor), a saturação e o valor (brilho) de forma independente.

O espaço HSV facilita a segmentação por cor ao concentrar as informações de matiz em um único canal, permitindo definir intervalos de cor mais precisos.

Uso do método *cv2.inRange* para segmentação

A biblioteca OpenCV oferece a função inRange para realizar a segmentação por cor. Essa função recebe como entrada:

- Uma imagem no espaço HSV.
- Um vetor com os valores do limite inferior da cor desejada.
- Um vetor com os valores do limite superior da cor desejada.

```
img_segmentada = cv2.inRange(
    img_hsv,
    limite_inferior,
    limite_superior
)
```

Segmentação por cor

Podemos converter uma cor do formato RGB para HSV utilizando o OpenCV.

```python
import cv2
import numpy as np

verdeRGB = np.uint8([[[0, 255, 0]]])
verdeHSV = cv2.cvtColor(verdeRGB, cv2.COLOR_BGR2HSV)
print(verdeHSV)  # Exemplo de saída: [[[60 255 255]]]
```

Para definir os limites de segmentação, podemos ajustar o canal de matiz somando e subtraindo um valor, normalmente 20 unidades.

```python
# Limites para a cor verde
limite_inferior = np.array([40, 100, 100])
limite_superior = np.array([80, 255, 255])
```

Exercício:

Desenvolva uma função chamada *calcular_limites_cor_hsv* que receba como entrada um array de uma cor específica no formato RGB e retorne os limites inferior e superior dessa cor no espaço HSV.

Tabela de Cores e seus Limites no Espaço HSV

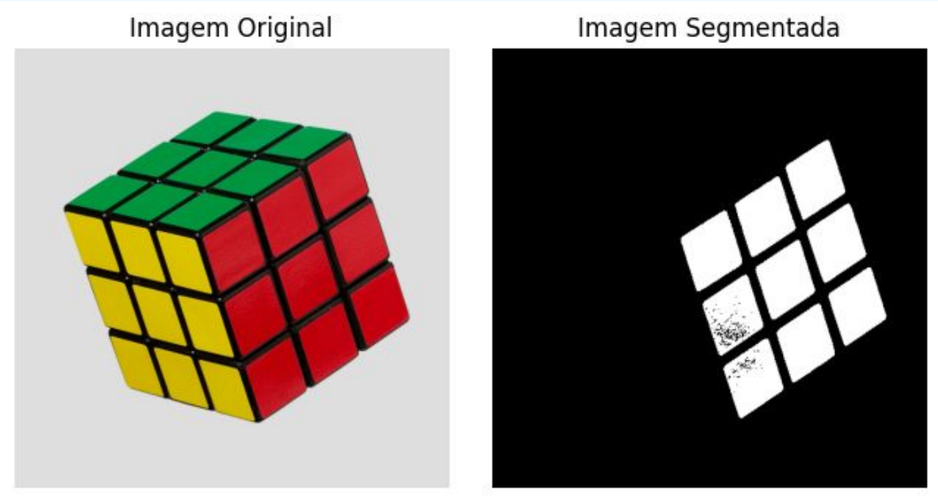
Cor	Limite Inferior (HSV)	Limite Superior (HSV)
Amarelo	[10, 100, 100]	[50, 255, 255]
Azul	[100, 100, 100]	[140, 255, 255]
Verde	[40, 100, 100]	[80, 255, 255]

Segmentação por cor

O caso específico da cor vermelha

- Na teoria das cores, o vermelho puro tem um valor de matiz próximo de 0° (360° também é vermelho, pois o círculo se fecha). No OpenCV, isso significa que a matiz para o vermelho está dividida em dois extremos da escala HSV:
  - Parte Superior: 160 a 179
  - Parte Inferior: 0 a 20
- Quando queremos segmentar o vermelho, precisamos considerar essa divisão, pois uma única faixa contínua de matiz não cobre toda a gama de tons vermelhos.

Exemplo quando não é considerado todos os tons de vermelho na segmentação



Segmentação por cor

O caso específico da cor vermelha

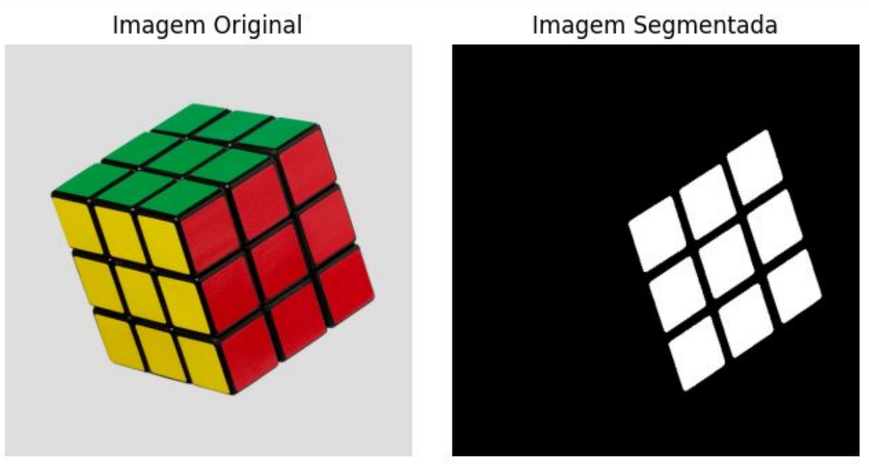
O OpenCV não trata automaticamente essa discontinuidade no HSV, então precisamos manualmente definir dois intervalos e combiná-los. Então é necessário verificar se h_min e h_max cruzam o limite de 0 ou 179.

Se H estiver dentro de um único intervalo:

- Retornamos apenas um par (limite_inferior, limite_superior).

Se H atravessar o limite do espaço HSV:

- Dividimos o intervalo em duas faixas separadas:
  - Do valor mínimo até 179
  - De 0 até o valor máximo



Exemplo quando é considerado todos os tons de vermelho na segmentação

PUC Minas

Segmentação por cor

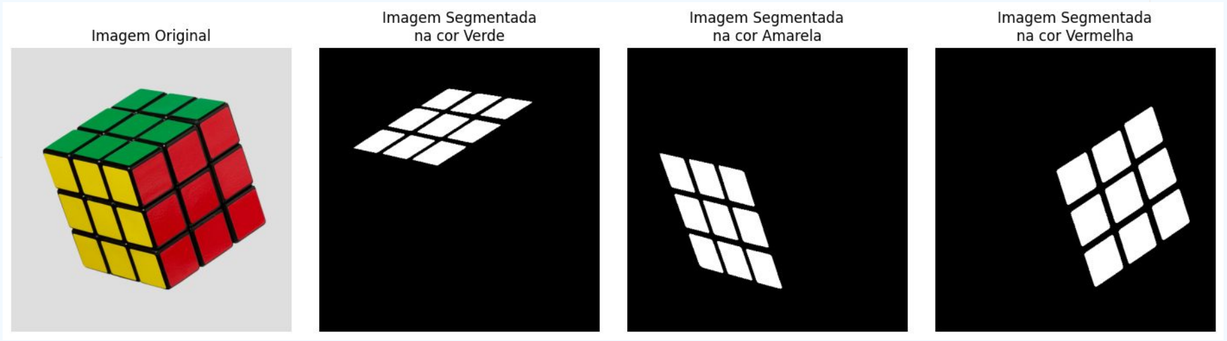
Desafio: Segmentação de Cores no Cubo Mágico

Você recebeu a imagem de um cubo mágico e deve implementar um código que segmente as cores verde, amarelo e vermelho utilizando o espaço de cores HSV e a função cv2.inRange da OpenCV.

Objetivos:

- Carregar a imagem do cubo mágico.
- Converter para o espaço HSV para facilitar a segmentação das cores.
- Implementar a função calcular_limites_cor_hsv, que recebe um array RGB de uma cor e retorna os limites inferior e superior no espaço HSV.
- Aplicar a segmentação para cada cor de interesse (verde, amarelo e vermelho).
- Exibir a imagem original e as segmentadas.

Resultado esperado

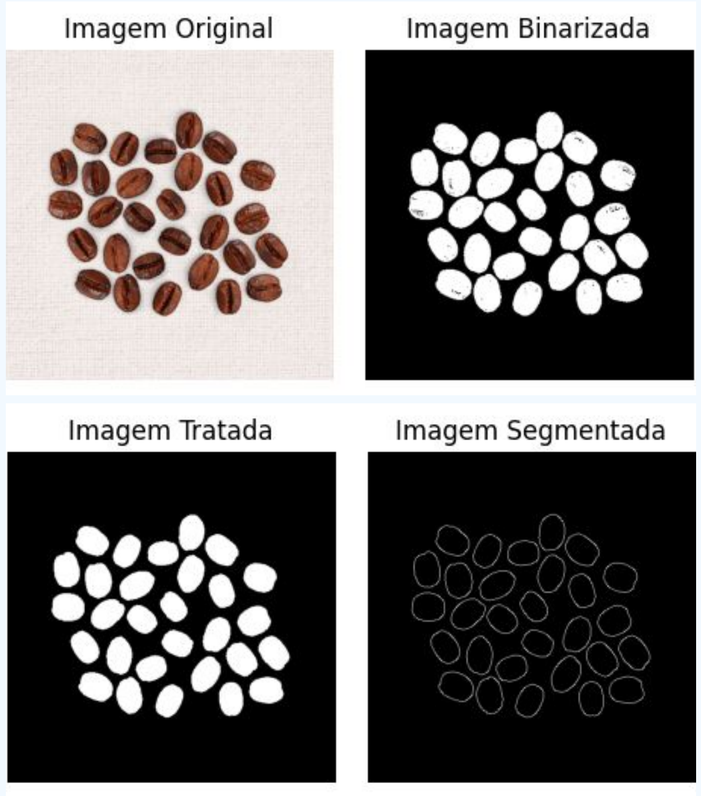


Segmentação por bordas

A segmentação por bordas é um método utilizado para identificar os contornos dos objetos em uma imagem. O detector de bordas Canny é um dos mais eficientes para esse tipo de tarefa. Ele opera em diferentes etapas para garantir uma segmentação precisa, reduzindo ruídos e preservando as bordas principais.

Processo de segmentação de bordas

- **Conversão para Tons de Cinza**: A imagem original é convertida para escala de cinza para simplificar o processamento.
- **Binarização:** Aplica-se um limiar para segmentar os pixels de interesse.
- **Operações Morfológicas:** São aplicadas operações de fechamento e erosão para remover ruídos e refinar a segmentação.
- **Detecção de Bordas com Canny:** O algoritmo de Canny é usado para destacar os contornos dos objetos.



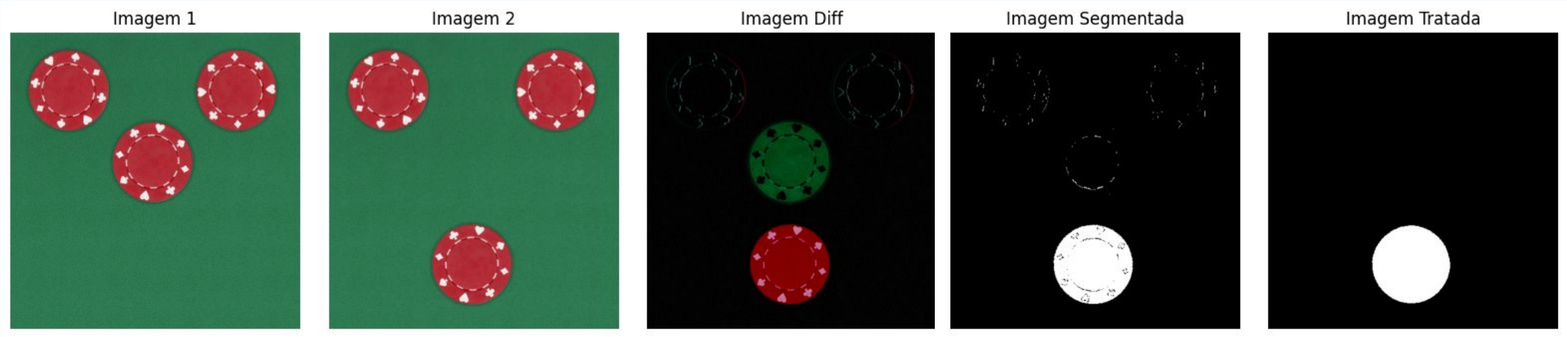
PUC Minas

Segmentação por movimento

A segmentação por movimento é um método que detecta alterações entre duas imagens capturadas em momentos diferentes. Esse processo é muito utilizado em monitoramento e rastreamento de objetos.

Processo de segmentação por movimento

- **Captura de Duas Imagens:** Uma imagem antes da movimentação e outra após.
- **Subtração das Imagens:** Os pixels que permaneceram inalterados são removidos, restando apenas os objetos que se moveram.
- **Conversão para HSV e Segmentação por Cor:** Aplica-se a conversão para HSV e a segmentação para isolar o objeto em movimento.
- **Operações Morfológicas:** São utilizadas para refinar a segmentação e remover ruídos.



Detecção de Contornos

Contornos

Contornos podem ser vistos como curvas que une todos os pontos ao longo da borda de uma determinada forma em uma imagem. Como eles definem os limites dos objetos, a análise desses pontos pode revelar informações essenciais para a **análise de forma**, **detecção** e **reconhecimento de objetos**.

A biblioteca OpenCV oferece diversas funções para detectar e processar contornos de forma eficiente. Antes de explorar essas funções, é importante compreender a estrutura de um contorno amostral.

Função que simula a detecção de um contorno em ma imagem.

```
def get_one_contour():
    """Retorna um contorno fixo"""
    cnts = [np.array([
        [[600, 320]], [[563, 460]], [[460, 562]], [[320, 600]],
        [[180, 563]],[[78, 460]], [[40, 320]], [[77, 180]],
        [[179, 78]], [[319, 40]], [[459, 77]], [[562, 179]]
    ], dtype=np.int32)]
    return cnts
```

O OpenCV fornece a função *cv2.drawContours()*, que desenha o contorna na imagem.

contourIdx: corresponde ao índice de contornos (útil ao desenhar contornos individuais. Para desenhar todos os contornos, passe -1)

```
cv2.drawContours(
    img,
    contours,
    contourIdx,
    color,
    thickness
)
```

Detecção de Contornos com OpenCV

A função *cv2.findContours()* é responsável por identificar contornos em imagens binárias:

```
contours, hierarchy = cv2.findContours(image, modo, metodo)
```

Os modos de detecção incluem:

- cv2.RETR_EXTERNAL: Retorna apenas os contornos externos.
- cv2.RETR_LIST: Retorna todos os contornos sem hierarquia.
- cv2.RETR_TREE: Retorna todos os contornos e estabelece relações hierárquicas.

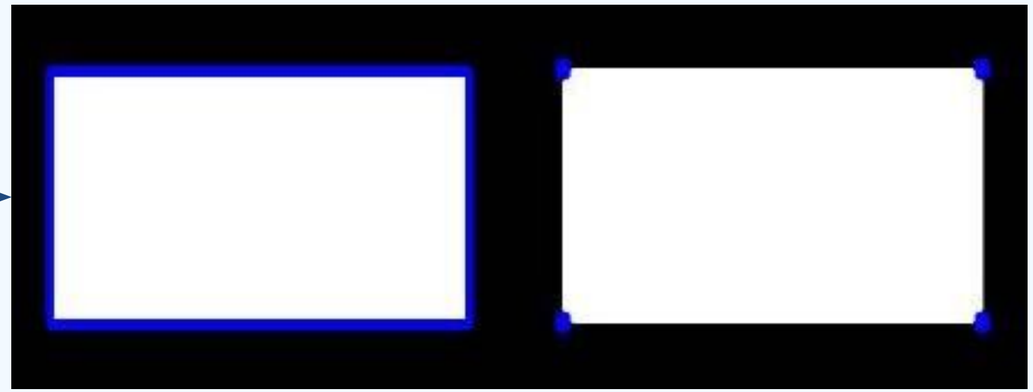
A estrutura **hierarchy** armazena informações sobre as relações entre os contornos, incluindo:

- Próximo contorno no mesmo nível
- Contorno anterior no mesmo nível
- Primeiro contorno filho
- Contorno pai

Detecção de Contornos com OpenCV

Os contornos podem conter muitos pontos desnecessários. O OpenCV fornece diferentes métodos para reduzi-los:

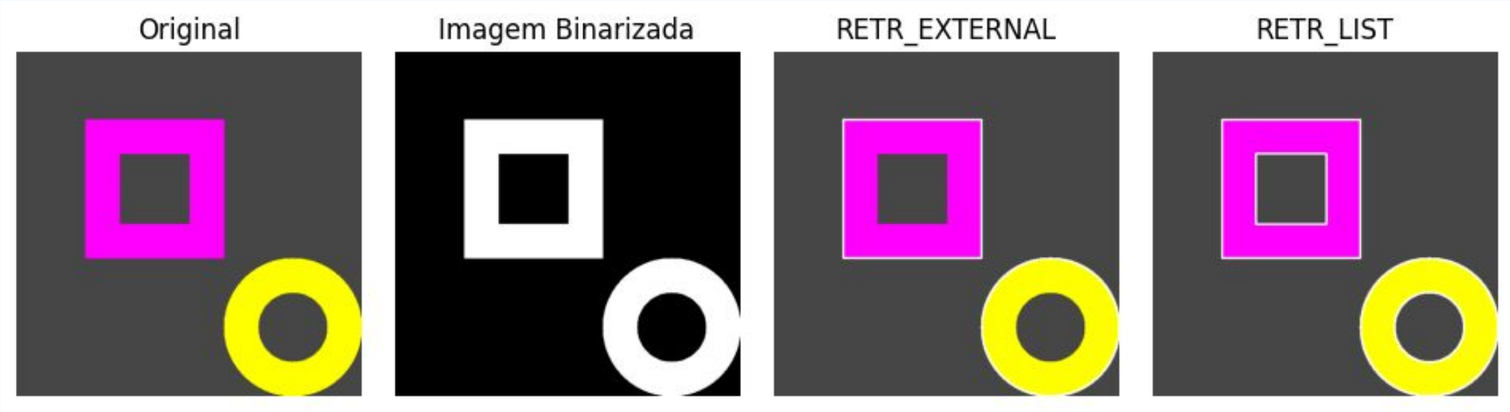
- *cv2.CHAIN_APPROX_NONE*: Mantém todos os pontos do contorno.
- *cv2.CHAIN_APPROX_SIMPLE*: Remove pontos redundantes e armazena apenas os vértices.
- *cv2.CHAIN_APPROX_TC89_L1* e *cv2.CHAIN_APPROX_TC89_KCOS*: Utilizam o algoritmo Teh-Chin para uma compressão baseada em curvatura.



PUC Minas

Detecção de Contornos com OpenCV

Exercício Prático

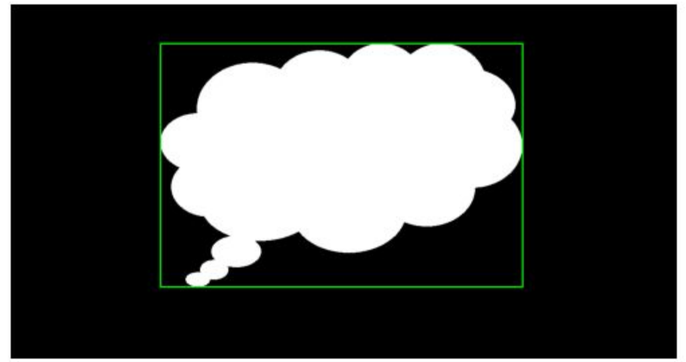


Funções úteis no OpenCV para contornos

Depois de detectar os contornos, podemos usar funções do OpenCV para descrever melhor cada um deles.

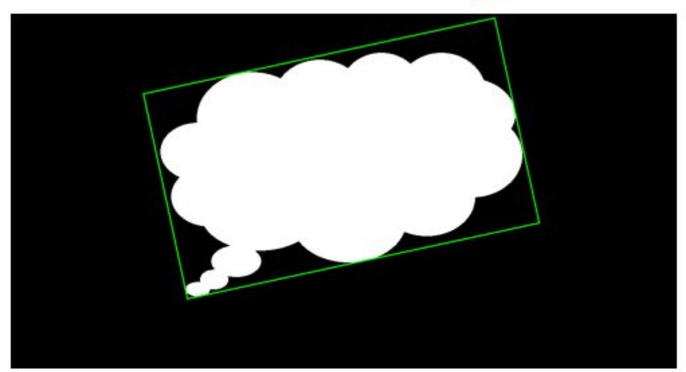
cv2.boundingRect()

- O que faz: Retorna o menor retângulo “alinhado ao eixo” (sem rotação) que envolve todo o contorno.
- Retorna: x, y, w, h → canto superior esquerdo e dimensões do retângulo.
- Uso: Pode ser usado para recortar o objeto da imagem ou calcular o apect ratio



cv2.minAreaRect()

- O que faz: Retorna o menor retângulo possível, mas agora permitindo rotação.
- Retorna: centro, dimensões e ângulo de rotação.
- Uso: Para obter os vértices reais do retângulo



Funções úteis no OpenCV para contornos

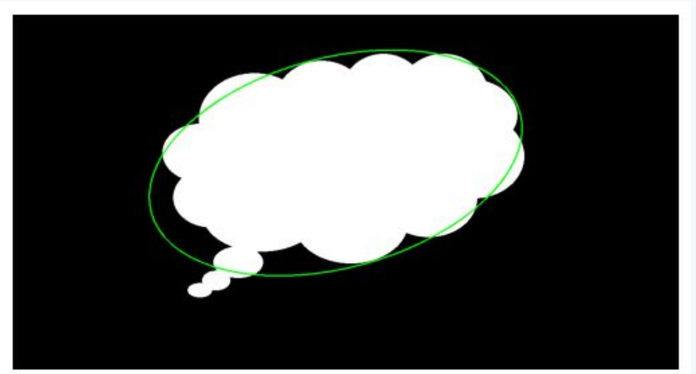
Depois de detectar os contornos, podemos usar funções do OpenCV para descrever melhor cada um deles.

cv2.minEnclosingCircle()

- O que faz: Calcula o menor círculo que envolve completamente o contorno.
- Retorna: centro (x, y) e o raio.
- Uso: Pode ser usado para medir simetria circular ou usar como filtro geométrico.

cv2.fitEllipse()

- O que faz: Ajusta uma elipse aos pontos de contorno, minimizando o erro quadrático.
- Uso: Ideal para medir excentricidade ou orientação da forma.
- Obs.: A elipse só é ajustada se o contorno tiver ao menos 5 pontos.



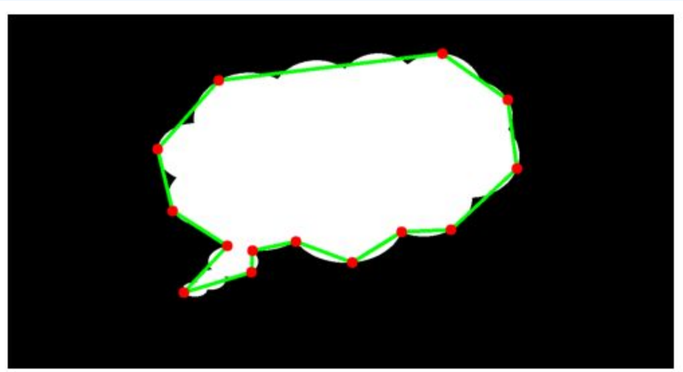
PUC Minas

Funções úteis no OpenCV para contornos

Depois de detectar os contornos, podemos usar funções do OpenCV para descrever melhor cada um deles.

cv2.approxPolyDP()

- O que faz: Aproxima o contorno por uma curva poligonal, reduzindo o número de vértices, com base em um parâmetro epsilon.
- Uso: Útil para identificar formas geometricas simples.



cv2.contourArea()

- O que faz: Calcula a área de um contorno, ou seja, quantos pixels estão dentro da forma delimitada por aquele contorno.
- Uso: Pode ser usado para filtrar contornos pequenos, detectar objetos grandes, ou para classificar objetos por tamanho.

```
pixel_per_cm = 40
area_pixels = cv2.contourArea(contour)
area_cm2 = area_pixels / (pixel_per_cm ** 2)

print(f"Área real estimada: {area_cm2:.2f} cm²")
```

Desafio prático: Ordenando círculos por área

Você recebeu uma imagem contendo vários círculos de tamanhos diferentes espalhados por um fundo preto. Seu objetivo é processar essa imagem e identificar todos os círculos, calculando a área de cada um usando *cv2.contourArea()*. Depois, você deve ordenar os círculos do menor para o maior, e rotular cada um com sua posição na ordem (1 para o menor, 2 para o segundo menor, etc.).

Etapas sugeridas

- Pré-processamento:
  - Leia a imagem.
  - Converta para tons de cinza.
  - Aplique um threshold ou Canny para destacar os círculos.
- Detecção de contornos:
  - Use cv2.findContours() para detectar os contornos.
- Cálculo das áreas:
  - Use cv2.contourArea() para obter a área de cada contorno.
- Ordenação:
  - Ordene os contornos pela área crescente.
- Rotulagem:
  - Calcule o centro (centroide) de cada círculo com momentos.
  - Use cv2.putText() para escrever o número da posição em cima de cada círculo na imagem.

Extração de Caracteristicas

Momentos de Imagem

Na matemática, um momento é uma medida que descreve certas características de uma forma ou função — por exemplo, sua massa, centro, dispersão, etc. Em visão computacional, usamos os momentos de imagem como ferramentas para extrair informações úteis sobre formas detectadas em uma imagem.

Momentos de imagem podem ser entendidos como médias ponderadas das intensidades dos pixels da imagem (ou de uma forma/contorno). Eles ajudam a capturar propriedades geométricas de objetos na imagem.

Por exemplo, com os momentos, podemos descobrir:

- A área de um objeto.
- O centro de massa (também chamado de centroide).
- Informações relacionadas à orientação, simetria e distribuição dos pixels.

PUC Minas

Momentos de Imagem

A função *cv2.moments()* é usada para calcular todos os momentos até a terceira ordem de uma forma vetorial (como um contorno).

```
M = cv2.moments(contour)
```

Aqui, contour é, por exemplo, contours[0], o primeiro contorno detectado por *cv2.findContours()*.

Tipo de Momento	Base de Cálculo	Invariante a	Utilidade principal
Espacial (m)	Origem da imagem	✖	Calcular área, centroide
Centrado (mu)	Centroide do objeto	✅ posição	Estudar forma sem depender da posição
Normalizado (nu)	Escala e centroide considerados	✅ posição e escala	Comparar formas com tamanhos diferentes

```
{'m00': 31134.0,
 'm10': 7783500.0,
 'm01': 7783500.0,
 'm20': 2023013223.6666665,
 'm11': 1945875000.0,
 'm02': 2023013223.6666665,
 'm30': 544322417750.0,
 'm21': 505753305916.6667,
 'm12': 505753305916.6667,
 'm03': 544322417750.0,
 'mu20': 77138223.66666651,
 'mu11': 0.0,
 'mu02': 77138223.66666651,
 'mu30': 0.0001220703125,
 'mu21': 6.103515625e-05,
 'mu12': 6.103515625e-05,
 'mu03': 0.0001220703125,
 'nu20': 0.07957924080046662,
 'nu11': 0.0,
 'nu02': 0.07957924080046662,
 'nu30': 7.137115971104531e-16,
 'nu21': 3.5685579855522656e-16,
 'nu12': 3.5685579855522656e-16,
 'nu03': 7.137115971104531e-16}
```

PUC Minas

Extraindo Características de Objetos

Os momentos de imagem são valores numéricos que descrevem propriedades geométricas de um contorno detectado em uma imagem. Com esses momentos, conseguimos calcular diversas características dos objetos — mesmo que os valores dos momentos por si só não tenham um significado geométrico direto, eles permitem calcular métricas importantes.

Área do contorno

```
print(cv2.contourArea(contours[0]))          # Método direto
print(M['m00'])                              # Usando momento m00
```

Centroide (centro de massa)

```
cx = round(M['m10'] / M['m00'])              # Coordenada X
cy = round(M['m01'] / M['m00'])              # Coordenada Y
```

Circularidade do Objeto

É uma medida que diz o quão próximo o contorno está de um círculo perfeito. A fórmula é:

$$k = (\text{perímetro}^2) / (\text{área} * 4 * \pi)$$

- Se o objeto for um círculo perfeito -> = 1
- Quanto maior o valor -> menos circular o objeto é

```
def roundness(contour, moments):
    length = cv2.arcLength(contour, True)
    k = (length * length) / (moments['m00'] * 4 * np.pi)
    return k
```

PUC Minas

Extraindo Características de Objetos

“Alongamento” do Objeto

Indica o quão esticado ou alongado o objeto é

1ª forma - usando elipse ajustada

```python
def eccentricity_from_ellipse(contour):
    (x, y), (MA, ma), angle = cv2.fitEllipse(contour)
    a = ma / 2   # semi-eixo maior
    b = MA / 2   # semi-eixo menor
    ecc = np.sqrt(a ** 2 - b ** 2) / a
    return ecc
```

2ª forma - usando momentos

```python
def eccentricity_from_moments(moments):
    a1 = (moments['mu20'] + moments['mu02']) / 2
    a2 = np.sqrt(4 * moments['mu11'] ** 2 + (moments['mu20'] - moments['mu02']) ** 2) / 2
    ecc = np.sqrt(1 - (a1 - a2) / (a1 + a2))
    return ecc
```

Proporção Largura/Altura

Essa métrica mostra se o objeto é mais largo ou mais alto

```python
def aspect_ratio(contour):
    x, y, w, h = cv2.boundingRect(contour)
    return float(w) / h
```

Ela é baseada no retângulo delimitador do contorno

PUC Minas

Matching de Contornos com Hu Moments

Quando queremos comparar a forma de dois objetos em uma imagem (por exemplo, verificar se dois contornos têm formato semelhante), uma técnica bastante útil é o uso dos Hu Moment Invariants.

O que são os Hu Moments?

São 7 valores calculados a partir dos momentos de imagem que ***são invariantes a rotação, escala e translação***. Ou seja, mesmo que a forma esteja girada, redimensionada ou movida, os valores continuam praticamente os mesmos.

O OpenCV fornece a função *cv2.matchShapes()* que usa esses Hu Moments para comparar dois contornos e retornar uma medida de similaridade.

```
score = cv2.matchShapes(contourA, contourB, method, 0.0)
```

Métodos de comparação disponíveis:

Método	Sensível a	Boa escolha para
cv2.CONTOURS_MATCH_I1	Momentos pequenos	Detalhes finos
cv2.CONTOURS_MATCH_I2	Momentos grandes	Comparação geral
cv2.CONTOURS_MATCH_I3	Diferenças pontuais	Erros criutícos

PUC Minas

Desafio prático: Identificando formas semelhantes a um círculo com Hu Moments

Você recebeu uma imagem chamada match_shapes.png, que contém várias formas geométricas diferentes (círculos, triângulos, quadrados, estrelas, etc.). Seu objetivo é utilizar o método cv2.matchShapes() do OpenCV para comparar cada uma dessas formas com um círculo perfeito:

Tarefa

1. Detectar todos os contornos presentes na imagem match_shapes.png.
2. Comparar cada contorno com um círculo de referência utilizando os três métodos de correspondência disponíveis:
   a. cv2.CONTOURS_MATCH_I1
   b. cv2.CONTOURS_MATCH_I2
   c. cv2.CONTOURS_MATCH_I3
3. Classificar as formas da imagem em ordem de semelhança com o círculo, ou seja, do contorno mais semelhante ao menos semelhante.
4. Gerar uma imagem final com as pontuações de similaridade escritas sobre cada forma.

Features em Visão Computacional

Em termos simples, uma feature (ou característica) é uma pequena região da imagem que contém informações únicas e importantes, como cantos, bordas ou padrões repetitivos. Essas regiões são úteis porque podemos usá-las para identificar, rastrear ou comparar objetos em diferentes imagens.

Por exemplo, se você tira duas fotos do mesmo objeto em ângulos diferentes, uma boa feature é aquela que aparece nas duas fotos, mesmo com rotação, escala ou iluminação diferentes.

Algoritmos para detectar features no OpenCV

- SIFT (Scale-Invariant Feature Transform)
- SURF (Speeded-Up Robust Features)
- ORB (Oriented FAST and Rotated BRIEF)

ORB: Uma alternativa eficiente ao SIFT e SURF

O ORB (Oriented FAST and Rotated BRIEF) é uma técnica poderosa e rápida que junta o melhor de dois mundos:

- FAST é usado para detectar os pontos-chave (**keypoints**).
- BRIEF é usado para gerar os **descritores** desses pontos.

A combinação resulta em um algoritmo leve, rápido e robusto — ideal para aplicações em tempo real, como realidade aumentada ou SLAM.

Keypoints

- São os pontos "interessantes" que o ORB identifica na imagem, como cantos. O ORB usa uma versão modificada do FAST para isso, e ainda calcula a orientação de cada ponto, o que ajuda a manter a robustez mesmo com rotação.

Descritores

- Depois de detectar os keypoints, o ORB gera um vetor de números para cada ponto — esse vetor é chamado de descritor, e contém a "assinatura" daquela região da imagem.

PUC Minas

ORB no Opencv

1) Cria o detector ORB

```
orb = cv2.ORB_create()
```

2) Detectar os keypoints

```
keypoints = orb.detect(image, None)
```

3) Calcula os descritores

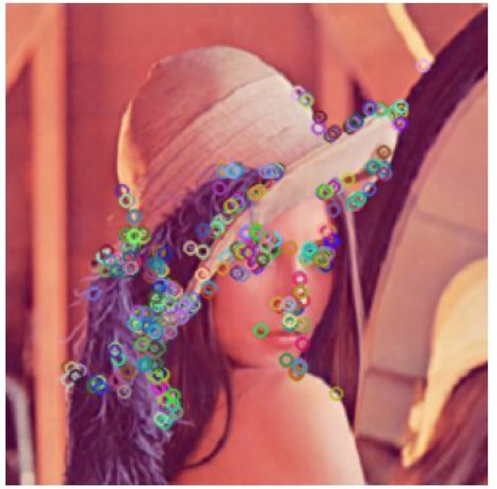
```
descriptors =  orb.compute(image, keypoints)
```

ou tudo de uma vez (detectar os keypoints e calcular os descritores)

```
keypoints, descriptors  = orb.detectAndCompute(image, None)
```

4) Desenha os keypoints na imagem

```
image_keypoints = cv2.drawKeypoints(image, keypoints, None)
```



PUC Minas

Feature Matching: Encontrando pontos em comum entre imagens

Quando detectamos features (características) em duas imagens, o próximo passo é descobrir quais delas são correspondentes. Por exemplo: se temos duas fotos de um mesmo objeto tiradas de ângulos diferentes, queremos saber quais partes da imagem representam o mesmo ponto no mundo real.

Como é feito o Matching no OpenCV

Brute-Force Matcher (BFMatcher)

- É o método mais direto: compara cada descritor de uma imagem com todos os da outra imagem.
- Retorna o par com menor distância (mais parecido).

FLANN Matcher

- Mais rápido que o Brute-Force em bases grandes.
- Usa algoritmos otimizados para aproximação de vizinhos mais próximos.
- Ideal para grandes quantidades de dados ou aplicações em tempo real.

PUC Minas

Etapas para aplicar feature matching com ORB

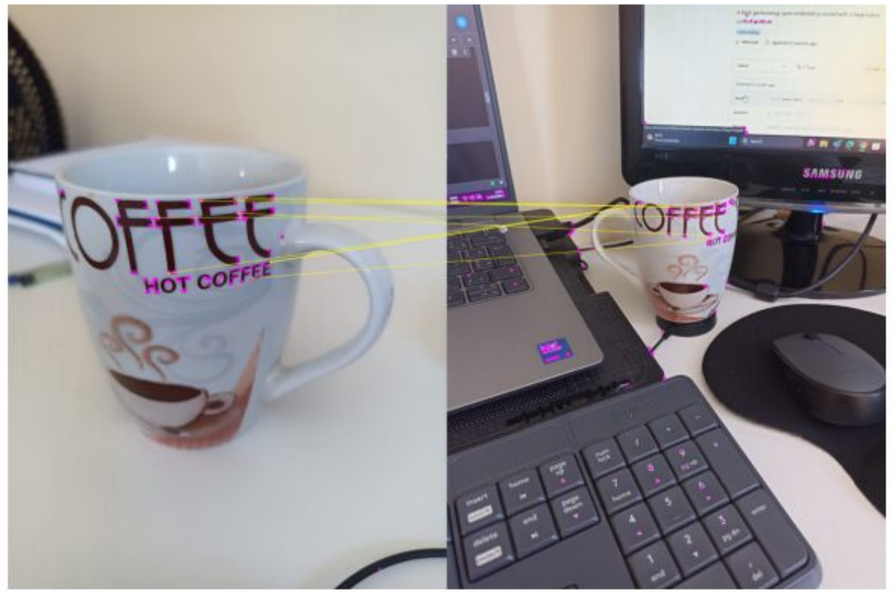
```python
orb = cv2.ORB_create()

# 1. Detecta keypoints e computa descritores nas duas imagens:
kp1, desc1 = orb.detectAndCompute(img1, None)
kp2, desc2 = orb.detectAndCompute(img2, None)

# 2. Cria o matcher e aplica:
matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = matcher.match(desc1, desc2)

# 3. Ordena os matches pela distância (quanto menor, melhor):
matches = sorted(matches, key=lambda x: x.distance)
```

crossCheck=True

- O match() só retorna matches recíprocos (ou seja, de ida e volta), o que geralmente já filtra boa parte dos falsos positivos.

```python
img_result = cv2.drawMatches(img1, kp1, img2, kp2, matches[:20], None,
                             matchColor=(255,255,0), singlePointColor=(255,0,255))
```

Resultado



Etapas para aplicar feature matching com ORB

Se você quiser mais controle e melhor filtragem de falsos positivos, use crossCheck=False e aplique o Lowe's Ratio Test manualmente (isso exige knnMatch):

Para cada descritor da imagem de referência, buscamos os dois descritores mais próximos na imagem da cena (com k=2).

```
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
knn_matches = bf.knnMatch(des1, des2, k=2)
```

Para cada par (m, n) de matches (onde m é o melhor, n é o segundo melhor):

- Verificamos se a distância de m é muito menor que a de n:

```
good_matches = []
for m, n in knn_matches:
    if m.distance < 0.75 * n.distance:
        good_matches.append(m)
```

- Essa verificação significa que m é um match confiável, pois é claramente melhor do que a segunda melhor opção.

PUC Minas

Recuperação de imagens com base no conteúdo visual utilizando BoVW

Método Bag of Visual Words

O Bag of Visual Words (BoVW), ou “Saco de Palavras Visuais”, é uma técnica de visão computacional inspirada no “Bag of Words” usando e processamento de linguagem natural (NLP).

- Analogia com NLP:
  - Em NLP, um texto é representado como um **histograma de palavras**, ignorando a ordem das frases.
  - No BoVW, uma imagem é representada como um histograma de palavras visuais, ignorando a posição exata das características.
- Objetivo Principal:
  - Classificar imagens (ex.: reconhecer objetos, cenas) ou recuperar imagens similares em bancos de dados.

PUC Minas

Etapas do BoVW

Passo 1: Extração de Características (Feature Extraction)

Extraímos ponto-chaves (keypoints) e descritores locais da imagem usando algoritmos como:

- SIFT (Scale-Invariant Feature Transform): Robusto a escala e rotação
- SURF (Speeded-Up Robust Features): Versão mais rápida do SIFT
- ORB (Oriented FAST and Rotated BRIEF): Eficientes em tempo real

São vetores que descrevem regiões locais da imagem. Por exemplo cada keypoint detectado gera um descritos de 128 dimensões (no caso do SIFT)

Passo 2: Construção do Vocabulário Visual

- Agrupamos todos os descritores das imagens de treinamento usando K-means.
- Cada cluster (grupo) do K-means representa uma “palavra visual”
- O centróide do cluster é a representação média daquela palavra.

Etapas do BoVW

Passo 3: Quantização das Características (Features Quantization)

- Para cada descritos de uma nova imagem, encontramos a palavra visual mais próxima (usando distância euclidiana)
- Isso transforma milhares de descritores em um conjunto limitado de palavras visuais.

Passo 4: Construção do Histograma (Image Representation)

- Contamos quantas vezes cada palavra visual aparece na imagem.
- O resultado é um histograma de frequência, que será a representação final da imagem.
- Exemplo de histograma (k=5 palavras visuais)

Palavra Visual	1	2	3	4	5
Frequência	45	12	30	8	5

Conclusão

Nesta aula, exploramos técnicas fundamentais de **Visão Computacional** para o processamento e extração de características de imagens, abordando três grandes tópicos:

Segmentação e Detecção de Contornos

- Aprendemos a utilizar técnicas como:
  - Limiarização (thresholding) para binarizar imagens e separar objetos do fundo.
  - Detecção de bordas com operadores como Sobel, Laplaciano e Canny, essenciais para identificar transições de intensidade.
  - Transformações morfológicas (erosão, dilatação, abertura, fechamento) para refinar segmentações e eliminar ruídos.
  - Detecção de contornos com *cv2.findContours()*, permitindo a extração de formas e objetos em imagens binárias.

Descritores de Imagem para Identificação de Padrões e Objetos

- Estudamos como extrair características robustas para reconhecimento e classificação:
  - Descritores de forma (momentos invariantes, Hu Moments) para representar objetos independentemente de escala, rotação ou posição.
  - Descritores baseados em keypoints (SIFT, SURF, ORB) para extrair pontos de interesse e suas representações matemáticas.

Recuperação de Imagens Baseada em Conteúdo (CBIR - Content-Based Image Retrieval)

- Vimos como sistemas de busca de imagens funcionam por similaridade visual: