hough-circular-wemerson

March 2, 2025

1 Transformada de Hough Circular

1.1 Atividade Pontuada 04 - Processamento de Imagens

1.1.1 I - Introdução

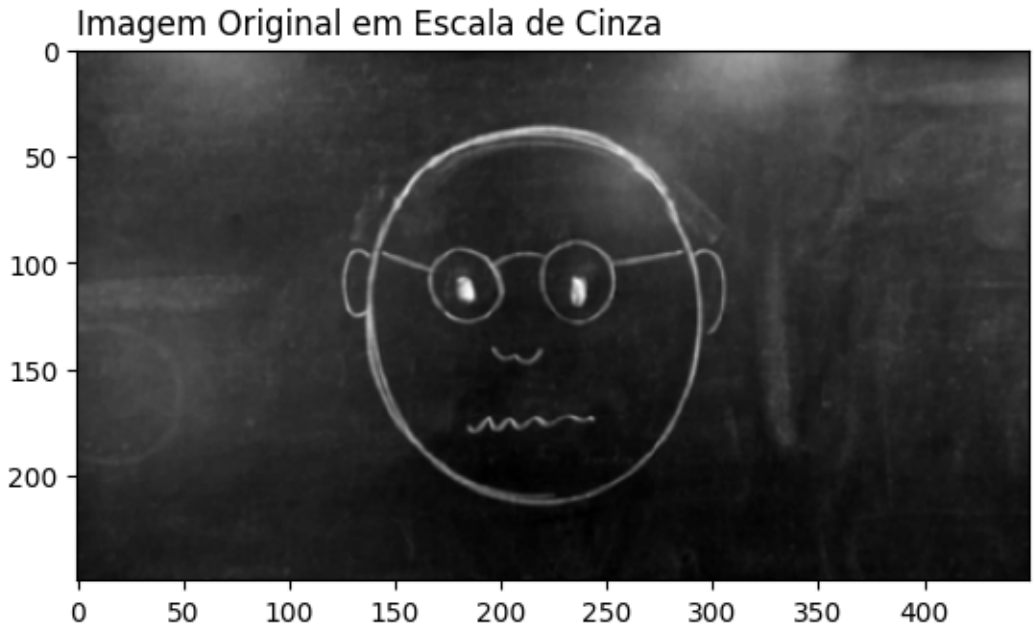
Este trabalho tem como objetivo implementar o algoritmo da Transformada de Hough Circular (CHT) e suas funções auxiliares para detectar círculos em imagens, com base no método descrito por Perdini. A implementação, realizada no ambiente Jupyter Notebook, seguirá a mesma lógica empregada nas aulas de Processamento de Imagens - COMP0432, substituindo funções prontas do skimage, como hough_circle_peaks e hough_circle, por uma versão própria do algoritmo de Transformada de Hough.

A detecção de círculos será feita em duas etapas: a construção da grade de acumuladores (transformação da imagem em um espaço de parâmetros) e a identificação dos picos de acumuladores, que indicam os círculos presentes. Para otimizar a detecção, um processo de refinamento é aplicado para reduzir ruídos.

Imagem Referência (Capturada da Aula 18/02) Para reproduzir o Jupyter Notebook apresentado pelo professor, capturei a imagem diretamente da apresentação. Antes de iniciar as operações, fiz alguns ajustes para compatibilizá-la com o processamento: redimensionei para aproximadamente 450x250 px, mantive a proporção da original e removi a camada de transparência (RGBA), deixando a imagem no formato RGB para evitar erros. Além disso, como só tinha acesso à imagem em nível de cinza, mantive a conversão para escala de cinza apenas a título de ilustração no código.

[1]:
```
import matplotlib.pyplot as plt #para exibir as imagens e gráficos
from matplotlib.image import imread #para ler a imagem a partir de um arquivo␣
 ↪(PNG por exemplo)
import numpy as np #para manipulação de arrays e operações matemáticas (como␣
 ↪arrays)
from skimage import color #para conversões entre diferentes representações de␣
 ↪cores (como escala de cinza)
from skimage.draw import circle_perimeter #para desenhar o perímetro (contorno)␣
 ↪de círculo na imagem

#diretório da imagem
imagem = imread(r"C:\Users\Wemerson\Downloads\rosto.png")
```

```
#converter para escala de cinza
imagem_cinza = color.rgb2gray(imagem)

#exibir a imagem
plt.imshow(imagem_cinza, cmap="gray")
plt.title("Imagem Original em Escala de Cinza", loc="left")
plt.show()
```



1.1.2 II - Processamento de Imagem

Filtro Sobel A detecção de bordas é um pré-requisito para a detecção de círculos. O filtro Sobel é usado para calcular o gradiente da imagem, destacando as bordas através de convoluções em duas direções: horizontal e vertical. A magnitude do gradiente resultante indica a presença de bordas na imagem. Isso é necessário porque os círculos a se detectar geralmente estão nas bordas de objetos na imagem.

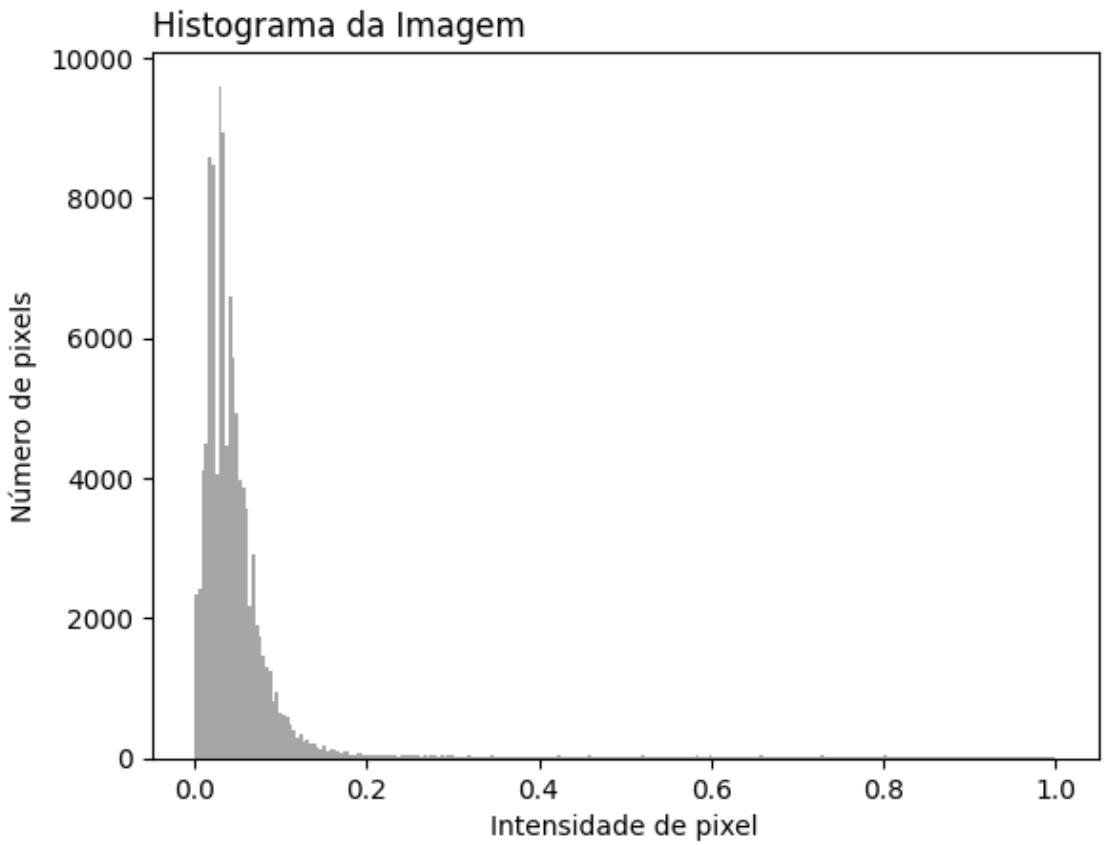
O Sobel ajuda a identificar essas transições de intensidade, criando um mapa de bordas, uma etapa importante pois sem bordas bem definidas, a Transformada de Hough Circular não funcionaria corretamente.

[2]:
```
#filtro Sobel
def filtro_sobel(imagem):
    #máscara Sobel para a direções horizontal(Gx)
    sobel_x = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]])
    #máscara Sobel para a direções vertical(Gy)
```

```
    sobel_y = np.array([[-1,-2,-1],
                        [ 0, 0, 0],
                        [ 1, 2, 1]])

    #obter dimensões da imagem
    m, n = imagem.shape

    #imagens de saída para Gx e Gy
    gx = np.zeros_like(imagem)
    gy = np.zeros_like(imagem)

    #convolução com as máscaras Sobel
    for i in range(1, m - 1): #ignorar bordas da imagem
        for j in range(1, n - 1):
            regiao = imagem[i - 1:i + 2, j - 1:j + 2] #região 3x3 ao redor do␣
 ↪pixel (i, j)
            gx[i, j] = np.sum(regiao * sobel_x) #aplicar máscara Sobel na␣
 ↪direção x
            gy[i, j] = np.sum(regiao * sobel_y) #aplicar máscara Sobel na␣
 ↪direção y

    #calcular magnitude do gradiente (combinação de Gx e Gy)
    magnitude_gradiente = np.sqrt(gx**2 + gy**2)

    return magnitude_gradiente

#aplicando filtro Sobel
bordas = filtro_sobel(imagem_cinza)

#exibir a imagem de bordas
plt.imshow(bordas, cmap='gray')
plt.title("Detecção de Bordas (Sobel)", loc="left")
plt.show()
```

Histograma O histograma de uma imagem mostra como os valores de pixel estão distribuídos, o que pode ser útil para ajustes de contraste ou limiar de binarização. Ele serve como uma ferramenta para analisar a distribuição das intensidades de pixels e pode ajudar a ajustar processos, como a binarização, que vem a seguir.

[3]:
```
#histograma
def exibir_histograma(imagem):
    #calcular histograma da imagem (intensidades de pixel)
    plt.hist(imagem.ravel(), bins=256, range=(0, 1), color='grey', alpha=0.7)
    plt.xlabel('Intensidade de pixel')
    plt.ylabel('Número de pixels')
    plt.show()

plt.title("Histograma da Imagem", loc="left")
#histograma da imagem de bordas
exibir_histograma(bordas)
```

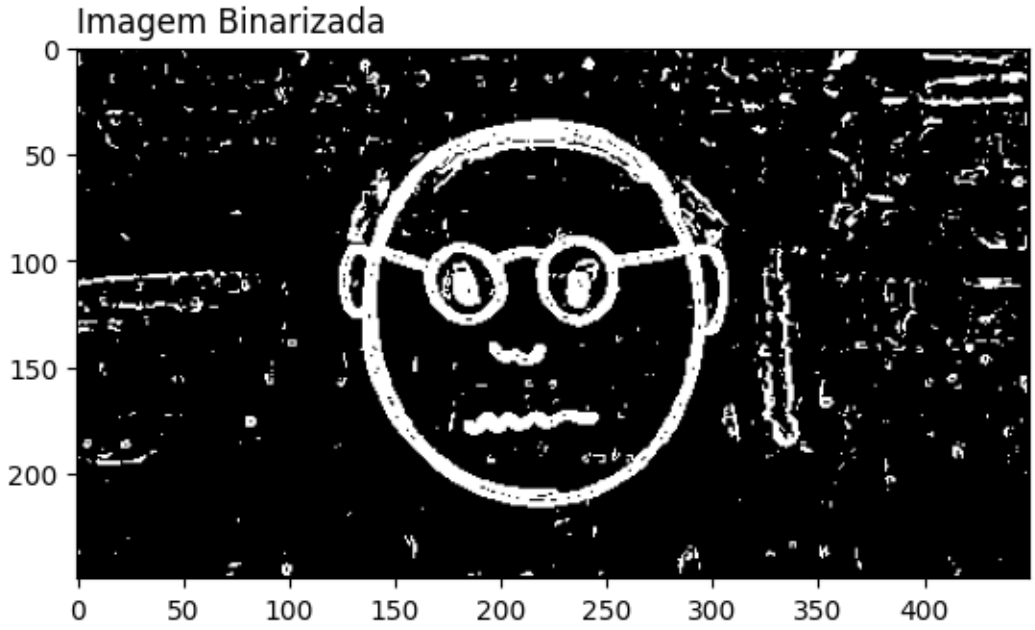


Binarização A binarização transforma a imagem em uma representação simplificada, onde os pixels de borda se tornam "1" (branco) e o fundo se torna "0" (preto). Ela é feita com base em um limiar de intensidade. O principal benefício aqui é que a imagem se torna mais fácil de processar para a detecção de círculos, pois o espaço de parâmetros da Transformada de Hough Circular pode se concentrar nos pixels de borda.

Embora simples, essa etapa é fundamental porque, sem ela, a CHT teria muito mais dados para processar, o que tornaria a detecção mais imprecisa.

[4]:
```
#binarização
def binarizacao(imagem, limiar=0.1):
    #limite de intensidade para binarizar a imagem
    imagem_binaria = np.zeros_like(imagem)

    #imagem binária será 1 (branca) se o valor da borda for maior que o limiar
    imagem_binaria[imagem > limiar] = 1

    return imagem_binaria

#binarização da imagem de bordas
```

```
binaria = binarizacao(bordas)

#exibir imagem binarizada
plt.imshow(binaria, cmap='gray')
plt.title("Imagem Binarizada", loc="left")
plt.show()
```



[5]:
```
#histograma da imagem de Binarizada
plt.title("Histograma da Imagem Binarizada", loc="left")
exibir_histograma(binaria) #terá apenas 0s e 1s
```

1.1.3 III - Operações Morfológicas

As operações morfológicas são aplicadas para melhorar a qualidade da imagem binarizada. Elas são úteis para eliminar ruídos (como pequenos pontos não desejados) ou preencher buracos nas bordas.

- `Erosão` - Remove os pixels das bordas das formas.
- `Dilatação` - Expande as regiões brancas.
- `Abertura` - Aplica erosão seguida de dilatação, removendo pequenos ruídos.
- `Fechamento` - Aplica dilatação seguida de erosão, preenchendo pequenos buracos.

Essas operações foram reaproveitadas do Trabalho 03, em que implementei para melhorar o processamento das imagens antes de realizar uma Esqueletização. No caso da Transformada de Hough Circular, elas são essenciais para garantir que a imagem esteja limpa e com bordas bem definidas, o que facilita a detecção dos círculos de forma precisa.

Erosão (A B) A `erosão` remove pixels das bordas das formas brancas da imagem. A função realiza isso verificando, para cada pixel da imagem, se o elemento estruturante se ajusta à região ao redor dele. Se a região for completamente compatível com o elemento estruturante (no caso, se todos os pixels de valor 1 coincidirem com o valor do elemento), o pixel central é mantido.

[6]:
```
#função de erosão (A  B): erosão de uma imagem A com o elemento estruturante B
def erosao(imagem, elemento_estruturante):
    m, n = imagem.shape #dimensões da imagem
    h, w = elemento_estruturante.shape #dimensões do elemento estruturante
    resultado = np.zeros_like(imagem) #imagem de saída
    pad_h, pad_w = h//2, w//2 #padding para bordas

    #iteração sobre a imagem (ignorando as bordas)
    for i in range(pad_h, m - pad_h):
        for j in range(pad_w, n - pad_w):
            regiao = imagem[i - pad_h:i + pad_h + 1, j - pad_w:j + pad_w + 1]␣
 ↪#região da imagem ao redor do pixel (i, j)

            #a erosão só mantém o pixel se a região for totalmente compatível␣
 ↪com o elemento estruturante
            if np.all(regiao[elemento_estruturante == 1] == 1):
                resultado[i, j] = 1
    return resultado #retorna a imagem erodida
```

Dilatação (A B) A `dilatação` é o oposto da erosão. Ela expande as regiões brancas na imagem. Para cada pixel da imagem, a dilatação mantém o pixel central se algum pixel da região ao redor (definida pelo elemento estruturante) for branco. Assim, as formas da imagem se expandem.

[7]:
```
#função de dilatação (A  B): dilatação de uma imagem A com o elemento␣
 ↪estruturante B
def dilatacao(imagem, elemento_estruturante):
    m, n = imagem.shape #dimensões da imagem
    h, w = elemento_estruturante.shape #dimensões do elemento estruturante
    resultado = np.zeros_like(imagem) #imagem de saída
    pad_h, pad_w = h//2, w//2 #padding para bordas

    #iteração sobre a imagem (ignorando as bordas)
    for i in range(pad_h, m - pad_h):
        for j in range(pad_w, n - pad_w):
            regiao = imagem[i - pad_h:i + pad_h + 1, j - pad_w:j + pad_w + 1]␣
 ↪#região da imagem ao redor do pixel (i, j)

            #a dilatação acontece se pelo menos um pixel da região corresponder␣
 ↪ao elemento estruturante
            if np.any(regiao[elemento_estruturante == 1] == 1):
                resultado[i, j] = 1
    return resultado #retorna a imagem dilatada
```

Abertura A `abertura` é uma operação composta pela erosão seguida da dilatação. Ela é útil para remover pequenos ruídos ou detalhes finos da imagem. Basicamente, ela primeiro reduz a imagem (erosão) e depois expande as regiões remanescentes (dilatação).

[8]:
```
#abertura (A  B)  B: primeiro a erosão (A  B) e depois a dilatação ((A  B) ␣
↪B)
def abertura(imagem, elemento_estruturante):
    return dilatacao(erosao(imagem, elemento_estruturante),␣
↪elemento_estruturante)
```

Fechamento O `fechamento` realiza a operação inversa da abertura: primeiro a dilatação e depois a erosão. Isso ajuda a preencher pequenos buracos ou lacunas nas regiões brancas da imagem.

[9]:
```
#fechamento (A  B)  B: primeiro a dilatação (A  B) e depois a erosão ((A  B)␣
↪ B)
def fechamento(imagem, elemento_estruturante):
    return erosao(dilatacao(imagem, elemento_estruturante),␣
↪elemento_estruturante)
```

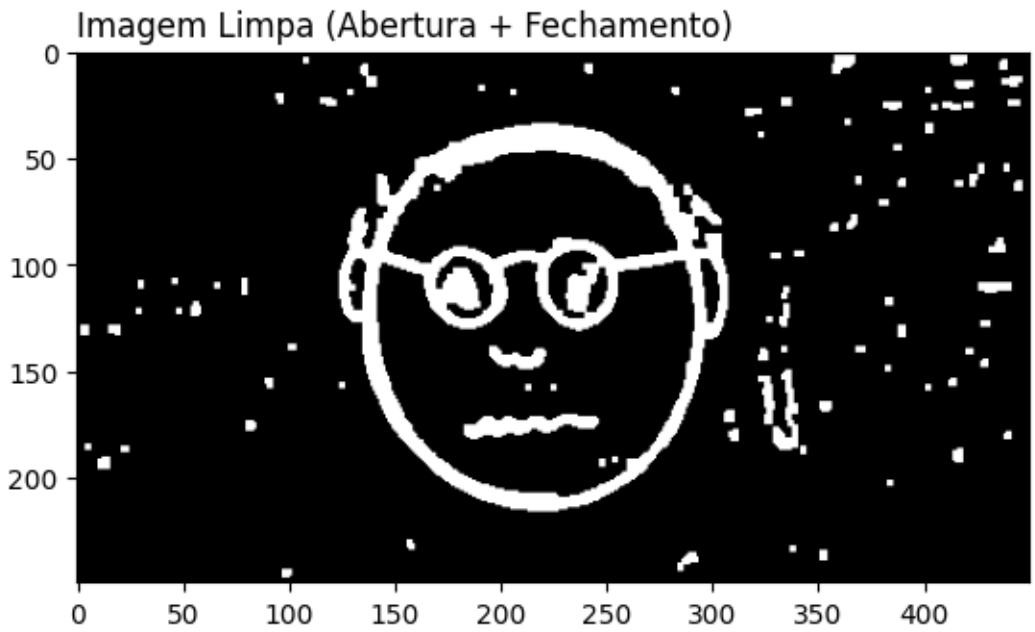
Limpeza da Imagem Após a binarização, a imagem pode conter ruídos ou buracos indesejados que prejudicam a detecção de círculos. Para resolver isso, aplicamos operações morfológicas que ajudam a melhorar a qualidade da imagem e a precisão da detecção.

Processo de Limpeza A limpeza da imagem binária é realizada utilizando operações morfológicas de `abertura` e `fechamento`. O fechamento preenche buracos nas formas, enquanto a abertura remove ruídos e detalhes indesejados. Essas operações ajudam a melhorar a qualidade da imagem e a precisão na detecção dos círculos.

No caso específico da `Transformada de Hough Circular`, a limpeza é essencial para garantir que a imagem binária esteja livre de artefatos indesejados antes de aplicar a detecção de círculos.

O elemento estruturante (kernel 3x3) usado para essas operações define a área ao redor de cada pixel que será afetada pelas transformações. Através dessa combinação de operações, conseguimos uma imagem binária limpa, com bordas mais definidas e com menores chances de erros durante a detecção dos círculos.

[10]:
```
#elemento estruturante (kernel 3x3)
elemento_estruturante = np.array([[1, 1, 1],
                                  [1, 1, 1],
                                  [1, 1, 1]], dtype=np.uint8)

#limpeza da imagem binária com abertura e fechamento
binaria_fechada = fechamento(binaria, elemento_estruturante) #remover pequenos␣
↪ruídos
binaria_limpa = abertura(binaria_fechada, elemento_estruturante) #preencher␣
↪pequenos buracos

#exibir imagem limpa
plt.imshow(binaria_limpa, cmap='gray')
plt.title("Imagem Limpa (Abertura + Fechamento)", loc="left")
plt.show()
```



1.1.4 IV - Transformada de Hough Circular (CHT)

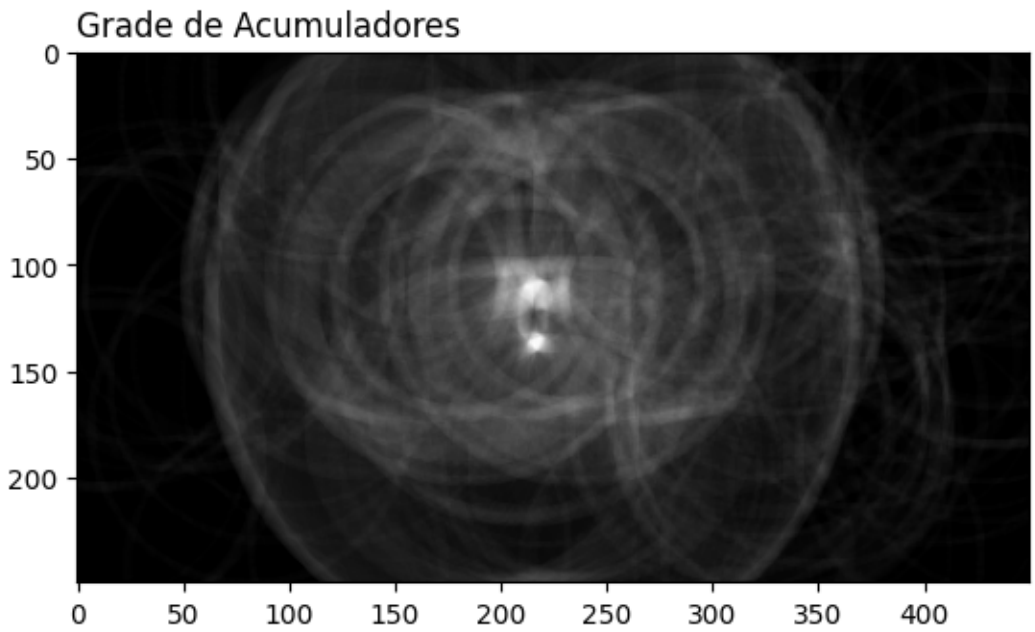
A **Transformada de Hough Circular (CHT)** é o passo central do processo. Ela transforma as bordas da imagem em um espaço de parâmetros, onde cada ponto de borda "vota" em possíveis centros e raios de círculos. O resultado dessa transformação é uma matriz acumuladora, onde os picos indicam a presença de círculos.

A vantagem dessa técnica é que ela pode detectar círculos em situações de ruído ou sobreposição de formas. O problema é que ela exige muita memória e poder computacional, por isso é importante otimizar o espaço de parâmetros e a detecção de picos.

Implementação da CHT

[11]:
```
#Transformada de Hough Circular (CHT)
def hough_circular(imagem, intervalo_raio):
    m, n = imagem.shape #obter dimensões da imagem (m: altura, n: largura)
    acumulador = np.zeros((m, n, len(intervalo_raio))) #inicializa matriz do␣
 ↪acumulador

    #obter índices dos pixels de borda (onde o valor é maior que zero)
    pixels_borda = np.argwhere(imagem > 0)

    #para cada pixel de borda, calcular os possíveis centros dos círculos
    for (i, j) in pixels_borda:
        for r_index, r in enumerate(intervalo_raio): #para cada raio no␣
 ↪intervalo de raios
            #calcular os centros do círculo para cada ângulo de 0 a 360 graus
```

```
            thetas = np.arange(0, 360, 1) #cria vetor de ângulos de 0 a 360␣
 ↪graus
            a = (i - r * np.cos(np.deg2rad(thetas))).astype(int) #coordenada x␣
 ↪(horizontal) do centro
            b = (j - r * np.sin(np.deg2rad(thetas))).astype(int) #coordenada y␣
 ↪(vertical) do centro

            #filtrar índices da imagem para garantir que o centro está dentro␣
 ↪dos limites
            mascara_valida = (0 <= a) & (a < m) & (0 <= b) & (b < n)
            a_valido = a[mascara_valida]
            b_valido = b[mascara_valida]

            #acumular votos nos centros válidos (dentro da imagem)
            for ai, bi in zip(a_valido, b_valido):
                acumulador[ai, bi, r_index] += 1  #incrementa valor do␣
 ↪acumulador para o centro (ai, bi)

    return acumulador #retorna a matriz acumuladora

#intervalo de raios (definido de 15 a 75, com intervalo 1)
intervalo_raio = np.arange(15, 76, 1)

#executando a Transformada de Hough Circular na imagem binária limpa
acumulador = hough_circular(binaria_limpa, intervalo_raio)

#visualizar a grade de acumuladores para o raio 75 (índice 60 pois começa em 0)
plt.imshow(acumulador[:, :, 60], cmap='gray')
plt.title("Grade de Acumuladores", loc="left")
plt.show()
```



Identificação dos Círculos Após a execução da CHT, o próximo passo é identificar os picos na matriz acumuladora, que correspondem aos centros dos círculos. Esses picos são considerados apenas se tiverem uma intensidade significativa, ou seja, se estiverem acima de um certo limiar.

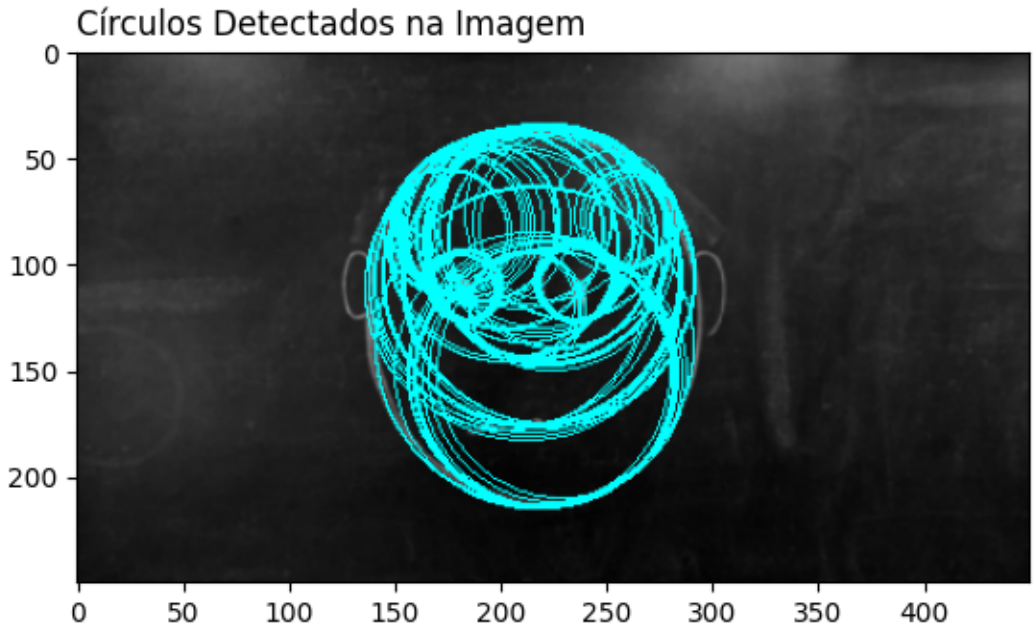
A detecção dos círculos depende do limiar escolhido. Se o limiar for muito baixo, muitos falsos positivos podem ocorrer. Se for muito alto, círculos reais podem ser ignorados.

[12]:
```
#função para encontrar os centros dos círculos na matriz acumuladora com base em um limiar
def encontrar_centros_circulos(acumulador, intervalo_raio, limiar):
    centros = []
    #para cada raio, procura-se os pontos máximos na matriz acumuladora
    for r_index in range(acumulador.shape[2]):
        max_acum = np.max(acumulador[:, :, r_index])

        #só considera os picos com votos acima de um certo limiar
        if max_acum >= limiar * max_acum:
            #obter índices de todos os pontos com valor máximo no acumulador
            pontos_maximos = np.argwhere(acumulador[:, :, r_index] == max_acum)
            for (x, y) in pontos_maximos:
                centros.append((x, y, intervalo_raio[r_index])) #adiciona o centro e o raio correspondente

    return centros
```

Desenho dos Círculos na Imagem Depois de identificar os *centros* e *raios* dos círculos, a última etapa é desenhá-los na imagem. Isso é feito com a função `circle_perimeter`, que traça o perímetro dos círculos com base nas coordenadas do centro e raio.

Esse passo tem como objetivo simplesmente visualizar os círculos encontrados na imagem original, para verificar que a detecção foi realizada corretamente.

[13]:
```
#função para desenhar círculos na imagem
def desenhar_circulos(imagem, centros_circulos):
    imagem_com_circulos = imagem.copy() #cópia da imagem original para evitar␣
 ↪alterações diretas

    #verificar se a imagem é do tipo float e os valores estão no intervalo [0,␣
 ↪1]
    if imagem_com_circulos.dtype == np.float32 or imagem_com_circulos.dtype ==␣
 ↪np.float64:
        if imagem_com_circulos.max() <= 1.0: #se os valores estão no intervalo␣
 ↪[0, 1]
            imagem_com_circulos = (imagem_com_circulos * 255).astype(np.uint8)␣
 ↪#convertendo para o intervalo [0, 255]

    for (x, y, r) in centros_circulos:
        #calcular perímetro do círculo usando a função circle_perimeter
        rr, cc = circle_perimeter(x, y, r, shape=imagem.shape)

        #garantir que os índices estão dentro da imagem
        rr = np.clip(rr, 0, imagem.shape[0] - 1)
        cc = np.clip(cc, 0, imagem.shape[1] - 1)

        #desenhar o círculo (marcando pixels do perímetro)
        imagem_com_circulos[rr, cc] = [0, 255, 255] #círculo verde água (RGB)

    return imagem_com_circulos

#encontra os centros dos círculos com votos acima de 99% do máximo no acumulador
centros = encontrar_centros_circulos(acumulador, intervalo_raio, limiar=0.99)
#desenhar os círculos na imagem original
imagem_com_circulos = desenhar_circulos(imagem, centros)

#exibir imagem com círculos
plt.imshow(imagem_com_circulos)
plt.title("Círculos Detectados na Imagem", loc="left")
plt.show()
```



1.1.5 V - Conclusão

A `Transformada de Hough Circular` é eficaz na detecção de círculos, mas apresenta limitações, especialmente quando implementada manualmente. Além da sensibilidade a ruídos e da dependência da qualidade da imagem inicial, a implementação sem funções prontas ainda pode resultar em resíduos menos significativos na imagem mesmo após a limpeza e em um número maior de círculos detectados em comparação com bibliotecas otimizadas. Embora ofereça mais controle sobre os parâmetros, essa abordagem exige um equilíbrio entre desempenho e precisão, tornando essencial a escolha criteriosa de limiar e das operações morfológicas aplicadas.

Wemerson Soares | 202300084020