IFCE - Visão Computacional - Processamento Digital de Imagens

FILTRAGEM E SEGMENTAÇÃO

Leonardo Costa de Sousa

Trabalho 2 - Filtragem e Segmentação

1. Filtragem (espacial ou na freqüência)

Nesse item foi feito uma filtragem no domínio da freqüência, que consiste em obter a transformada de Fourier de uma imagem, aplicar um filtro a essa transformada e calcular a inversa para obter o resultado.

$$Transformada\ de\ Fourier\ para\ uma\ imagem\ discreta:$$

$$F(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

$$para\ u = (0,1,2,\dots,M-1)\ e\ v = (0,1,2,\dots,N-1)$$

$$Transformada\ inversa:$$

$$f(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} F(x,y)e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

$$para\ x = (0,1,2,\dots,M-1)\ e\ y = (0,1,2,\dots,N-1), onde\ \Delta u = 1/(M\Delta x)\ e\ \Delta v = 1/(N\Delta y)$$

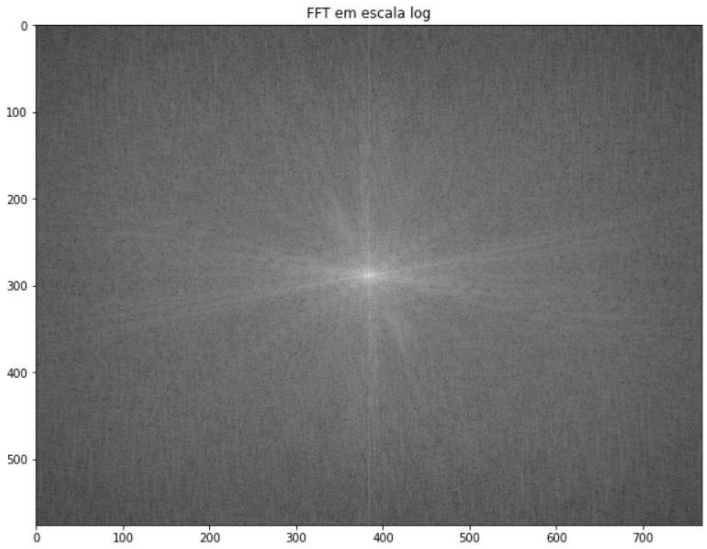
Abaixo temos a imagem em tons de cinza em que vamos aplicar o filtro.

```
img = cv2.imread('imgs/pinguins2.jpeg', cv2.IMREAD_GRAYSCALE)
```

A imagem é transformada para o domínio de Fourier ou FTT (transformada discreta), representada por F(u,v), utilizando a função ff2 da biblioteca scipy.fftpack que é então tratado, a transformada de Fourier e sua inversa geram números complexos isso é resolvido pela função np.absolute que gera os valores absolutos, e exibido em escala logarítmica, como podemos ver abaixo:

```
F = fp.fft2(img) #Calculando a FFT 2D da imagem
```

```
#tratando a FFT para os gráficos
Fm = np.absolute(F)
Fm /= Fm.max()
Fm = fp.fftshift(Fm)
Fm = np.log(Fm) #FFT em escala logaritma
```

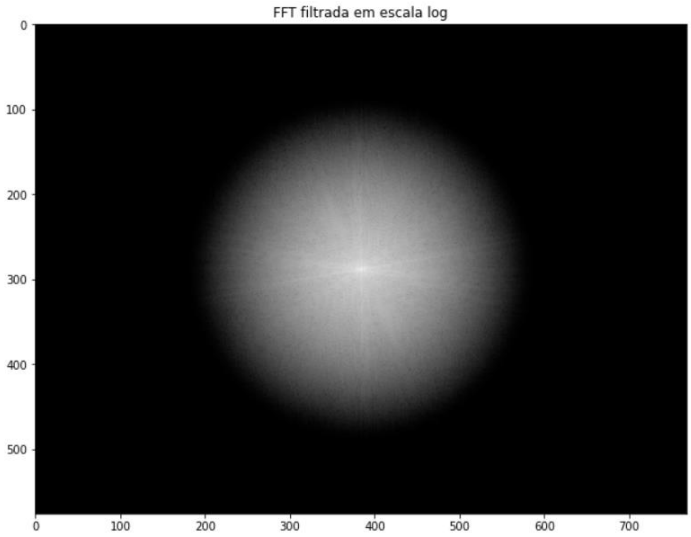


Agora precisamos aplicar o filtro H(u,v) na imagem no domínio da freqüência F(u,v) que nesse caso trata-se de um filtro passa baixa gaussiano, um filtro de desfoque (gaussian blur), que atua eliminando as altas freqüências da imagem ou seja vai ser exibido somente as transições suaves. Para gerar esse filtro foi usada a função abaixo:

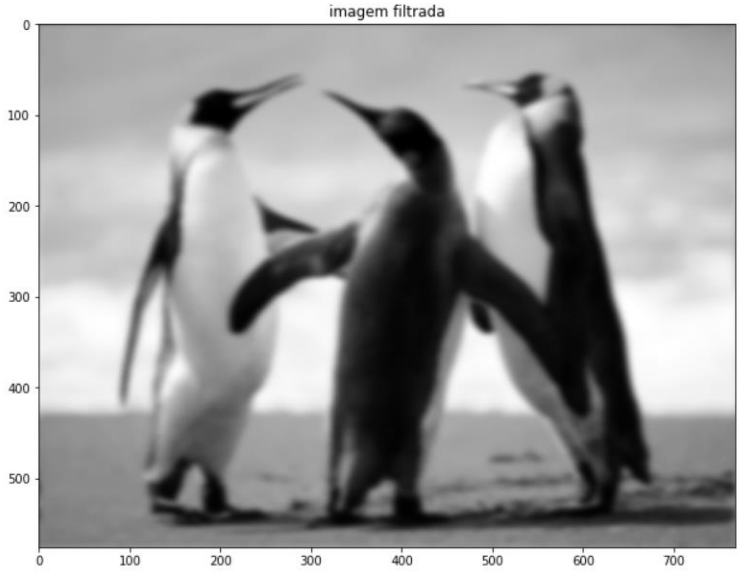
```
def gauss(S, sigma=1, vmax= 0.05):

    Nu, Nv = S.shape
    u = Nu * np.linspace(-vmax, vmax, Nu)
    v = Nv * np.linspace(-vmax, vmax, Nv)
    U, V = np.meshgrid(v, u)

    G = np.exp(-(U*U + V*V) / 2.0 / sigma**2)

    G = fp.fftshift(G)

    return G / sigma**2
```

$$Fórmula\ da\ função\ gaussiana: g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

O filtro é aplicado na FFT da imagem, obtemos então H(u,v)F(u,v) que após ser devidamente tratado é exibido abaixo:



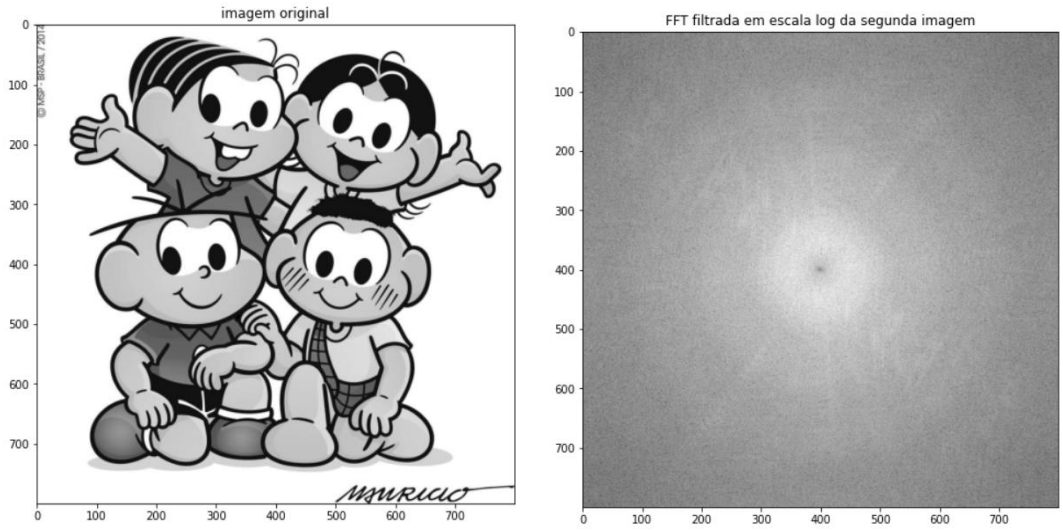
Obtemos a transformada inversa através da função ifft2 de scipy.fftpack, a imagem e tratada e por fim exibida. Onde o desfoque da imagem apos a aplicação do filtro pode ser observado.



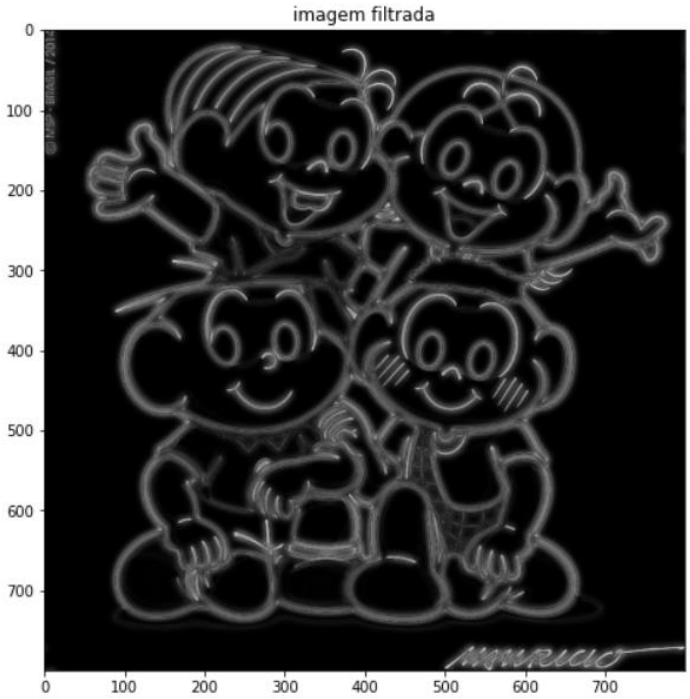
Fazendo uma modificação na função Gauss para que ela possa ser usada para filtrar as baixas freqüências obtendo assim um filtro passa-alta.

```
G = 1 - np.exp(-(U*U + V*V) / 2.0 / sigma**2)
```

E aplicando na imagem abaixo, temos sua FFT filtrada:



O resultado final trata-se de um filtro passa-alta, como pode ser observado na imagem abaixo que somente mostras os contornos ou transições fortes.



2. Morfologia Matemática (Erosão e dilatação)

Erosão e dilatação são operações elementares da morfologia, com a erosão podemos diminuir partículas, eliminar componentes menores que o elemento estruturante, aumenta buracos e permitir a separação de componentes conectados e com a dilatação podemos aumenta particular, preencher buracos e conecta componentes. Formalmente temos:

$$Erosão: A \ominus B = \{x | B_x \subseteq A\}$$

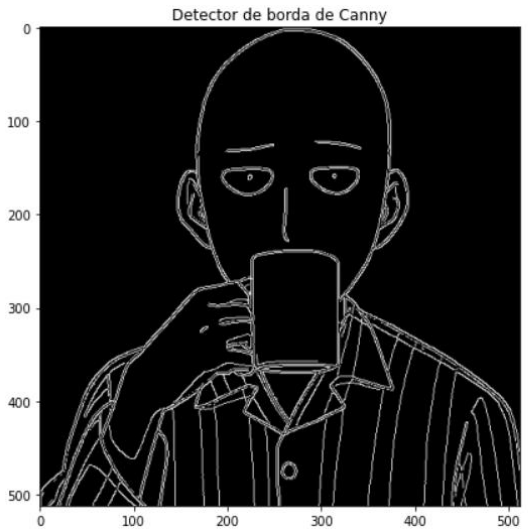
$$dilatação: A \otimes B = \{x | B_x \cap A \neq \emptyset\}$$

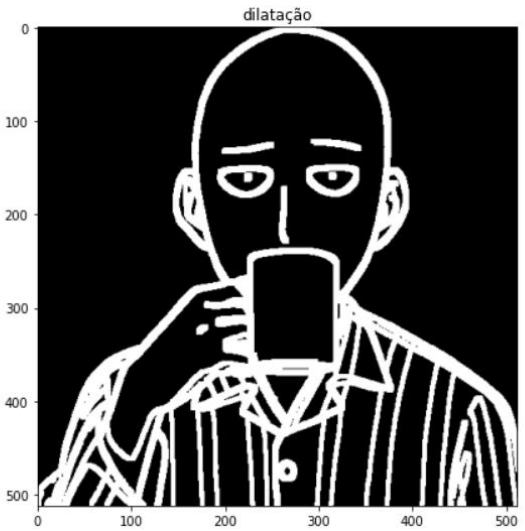
Onde A é a imagem e B o elemento estruturante, que conforme varia de tamanho e forma define a ação da operação.

Para o primeiro exemplo foi aplicado o detector de borda de Canny, para obtenção das bordas da imagem do saitama, então e aplicado dilatação com a função cv2.dilate e um elemento estruturante 5x5.

```
canny = cv2.Canny(img,100,200, 15) #Detector de borda de Canny

ele = np.ones((5,5),np.uint8)

dilate = cv2.dilate(canny, ele, 1)
```

A dilatação tem como efeito o engrossamento das bordas como pode ser observado nas imagens abaixo:

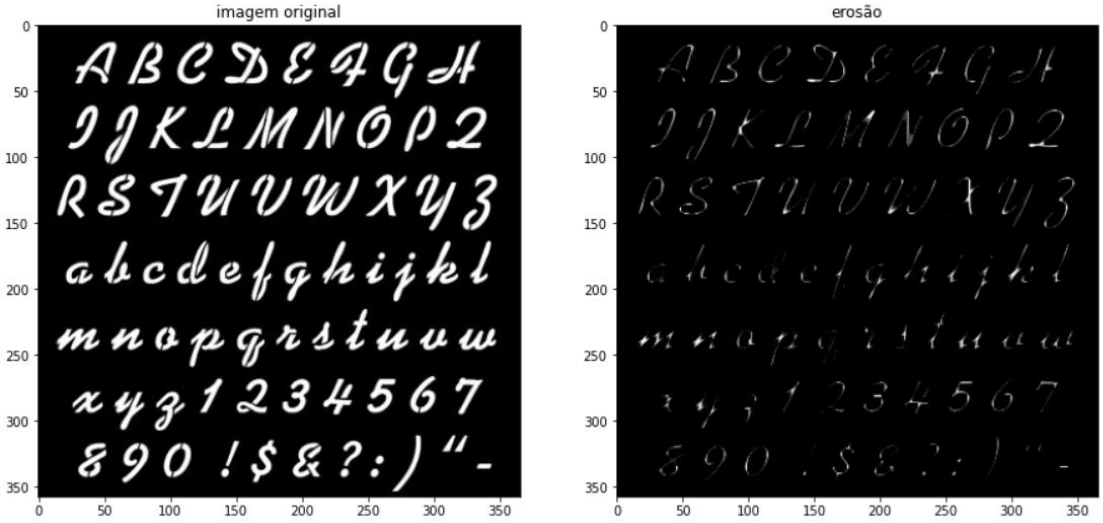




A erosão é aplicada diretamente uma imagem compostas do alfabeto branco no fundo preto com utilizando a função cv2.erode e um elemento estruturante 2x2, mais que isso as letras ficam imperceptíveis.

```
letras = cv2.imread('imgs/letras.jpg', cv2.IMREAD_GRAYSCALE)

#ele = cv2.getStructuringElement(cv2.MORPH_CROSS,(2, 2))
ele = np.ones((4,4),np.uint8)

erode = cv2.erode(letras, ele, 1)
```

O efeito principal que pode ser observado na erosão é o afinamento das letras como pode ser observado no resultado abaixo.

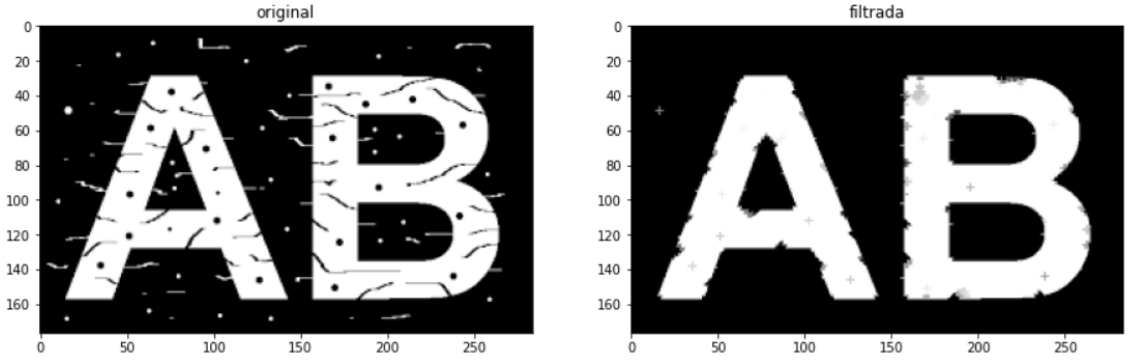


Abertura é uma erosão seguido de dilatação (utilizando o mesmo elemento estruturante) sendo útil na remoção de ruído da imagem, dilatação seguida de erosão chama-se fechamento.

Umas letras com ruído e carregado é aplicada a abertura morfológica com um elemento estruturante 5x5 o que elimina os ruídos de fora da imagem, então ela e invertida utilizando a função cv2.bitwise onde outra vez e aplicada a abertura pondo fim nos ruídos internos, por fim a imagem e invertida para seu estado inicial com redução considerável dos ruídos.

```
img2 = cv2.imread('imgs/ruido.png', cv2.IMREAD_GRAYSCALE)
ele = cv2.getStructuringElement(cv2.MORPH_CROSS,(5, 5))

data = cv2.erode(img2, ele)
data = cv2.dilate(data, ele)

data = cv2.bitwise_not(data) #inverte a imagem pra filtrar a parte interna

data = cv2.erode(data, ele)
data = cv2.dilate(data, ele)
```

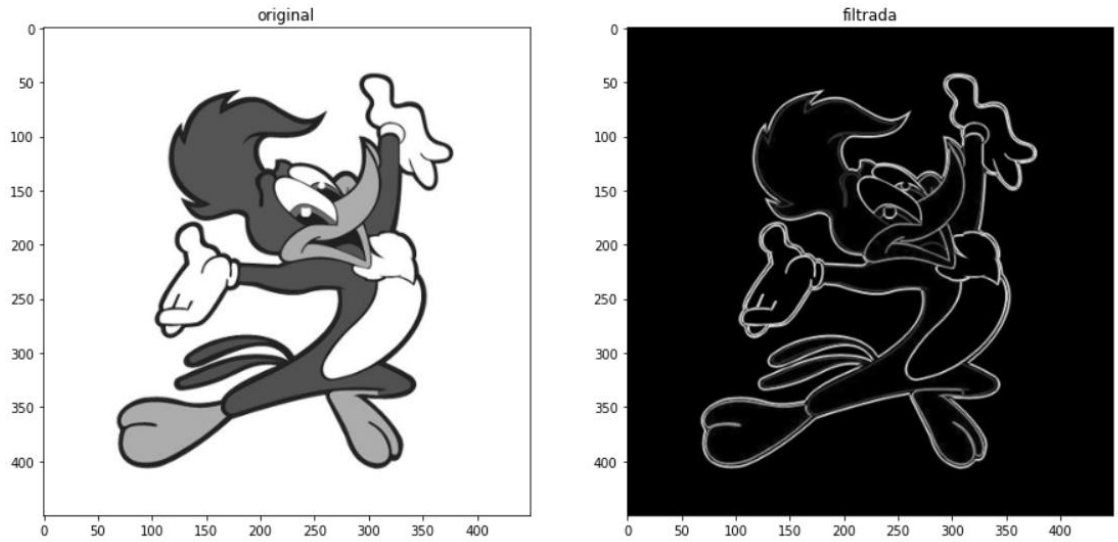
Como pode ser observado abaio:



Erosão e dilatação podem ser usadas para detecção de contorno, no exemplo a seguir foi feito a dilatação da imagem do pica-pau e posteriormente o resultado é subtraída da imagem original.

```
img3 = cv2.imread('imgs/picapau.jpg', cv2.IMREAD_GRAYSCALE)

ele = cv2.getStructuringElement(cv2.MORPH_CROSS,(5, 5))
dilate = cv2.dilate(img3, ele)

contorno = dilate - img3
```

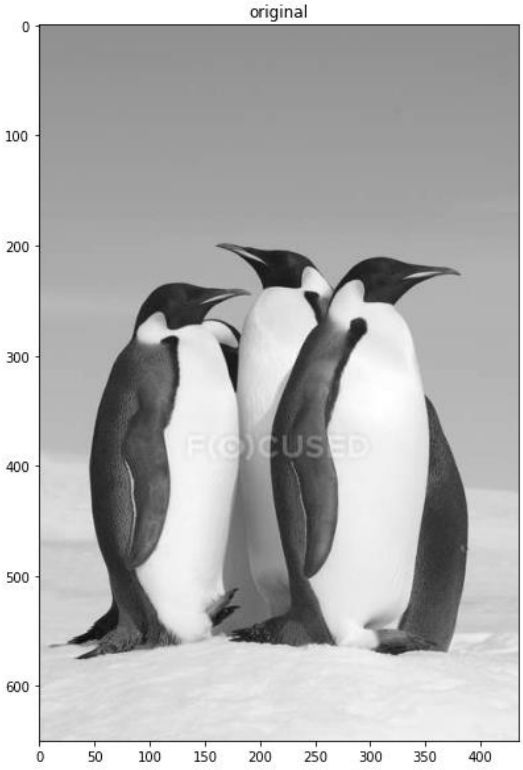
Como a dilatação engrossa as bordas, ao se subtrair a imagem original do resultado fica evidente somente o contorno:

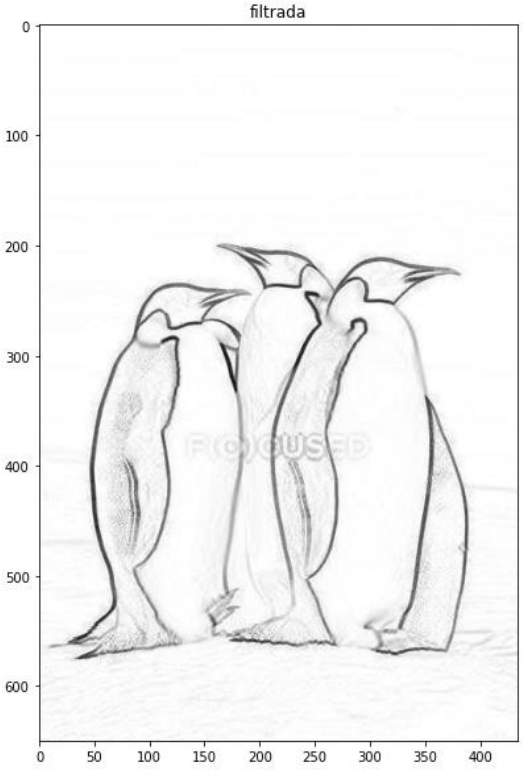


Como pode ser observado também na imagem dos pingüins logo abaixo

```
img4 = cv2.imread('imgs/pinguins.jpg', cv2.IMREAD_GRAYSCALE)
ele = cv2.getStructuringElement(cv2.MORPH_CROSS,(8, 8))
dilate = cv2.dilate(img4, ele)

contorno = dilate - img4

contorno = cv2.bitwise_not(contorno)
```

Aqui o resultado foi invertido para se obter esse resultado de desenho.





3. Limiarização por média móvel

Limiarização por média móvel pode ser usado para segmentar uma imagem quando a intensidade entre o fundo e os objetos são suficientemente diferentes.

A função abaixo gera um limiar global tendo com parâmetros a imagem um limar estimado e um limite para o numero máximo de interações.

1. Ele divide a imagem em duas partes a partir do limiar passado por parâmetro.
2. Encontra a media entre as duas partes.
3. Calcula o novo limiar.
4. Verifica o critério de parada, caso contrario volta para a instrução 1.

Ou seja ele aplica o algoritmo de limiarização global simples por media móvel.

```python
def thres_finder(img, thres=20,delta_T=1.0):

    #Divide as imagens em duas partes
    x_low, y_low = np.where(img<=thres)
    x_high, y_high = np.where(img>thres)

    #Encontra a média das duas partes
    mean_low = np.mean(img[x_low,y_low])
    mean_high = np.mean(img[x_high,y_high])

    #Calcula o novo limite
    new_thres = (mean_low + mean_high)/2

    #Critérios de parada, caso contrário iterar
    if abs(new_thres-thres)< delta_T:
        return new_thres
    else:
        return thres_finder(img, thres=new_thres,delta_T=1.0)
```

A função limiarização limiariza a imagem através de um limiar passado por parâmetro e a função calcHist geta o histograma da imagem.

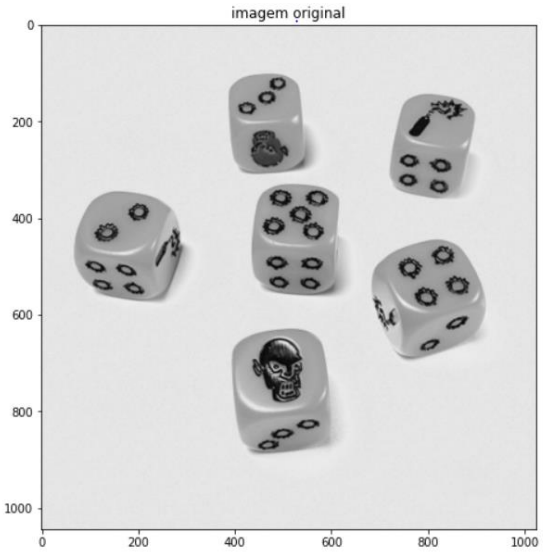
```python
def limiarizacao(img, limiar):

    img_saida = np.zeros(img.shape)

    Tmin = np.min(img)
    Tmax = np.max(img)

    altura, largura = img_saida.shape

    for y in range(0, altura):
        for x in range(0, largura):
            if img[y][x] < limiar:
                img_saida[y, x] = Tmin
            else:
                img_saida[y, x] = Tmax

    img_saida = np.uint8(img_saida)

    return img_saida
```

Função para gerar o histograma:

```python
def calcHist(img):
    import numpy as np
    import matplotlib.pyplot as plt
    import math as m

    altura, largura = img.shape

    if img.dtype == 'uint8':
        buffer = np.zeros((256))

    for y in range(0, altura):
        for x in range(0, largura):

            buffer[(img[y, x])] += 1

    return 100*buffer/(altura*largura)
```

A imagem que vamos segmentar e carregada na variável img em tons de cinza.

```python
img = cv2.imread('imgs/dados2.jpg', cv2.IMREAD_GRAYSCALE)
```



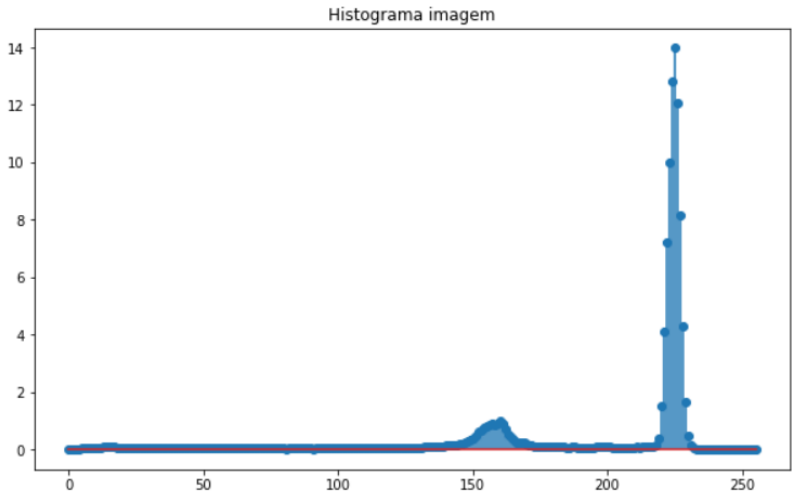
Posteriormente tem seu histograma gerado pela função calcHist, seu limiar é calculado por fim a imagem e limiarizada por esse limiar calculado.

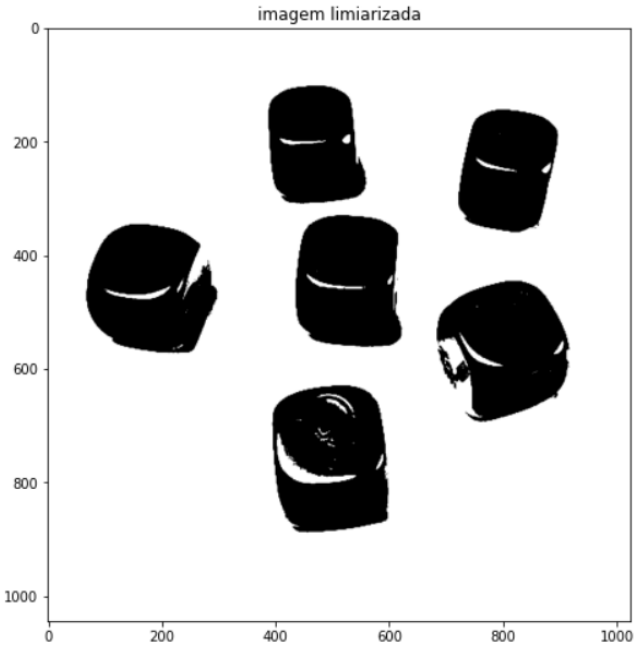
```
hist = calcHist(img)

thresh = thres_finder(img)
print(f'threshold = {thresh}')

img_seg = limiarizacao(img, thresh)
```

Limiar global retornado pela função:

```
threshold = 179.31688356840647
```

Histograma da imagem e resultado da limiarização:

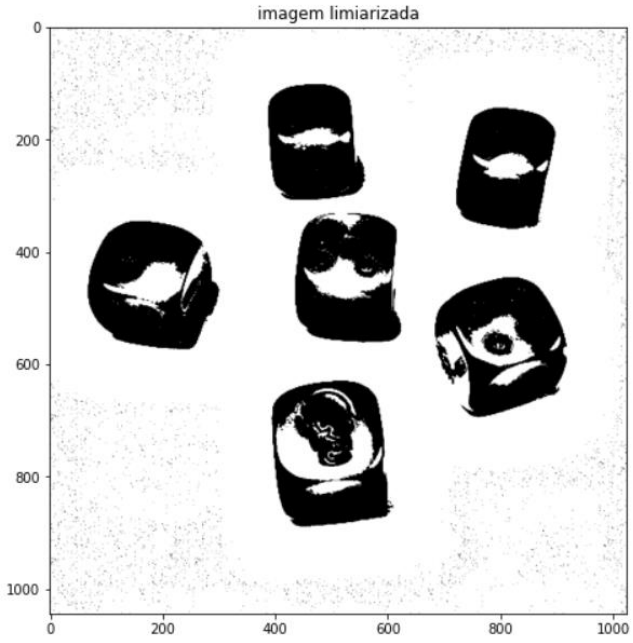




A segmentação nesse caso apresenta algumas falhas como pode ser observado, onde parte da sombra do objeto foi adicionado a ela por ter uma intensidade próxima ao do objeto.

O equivalente usando a biblioteca do opencv:

```
img_seg2 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 199, 5)
```



4. Crescimento de regiões

Técnica de segmentação por similaridade que agrupa pixels em regiões cada vez maiores iniciados por um conjunto de pontos semente, manual ou automaticamente definidos, baseados em um critério de crescimento. Em cada interação um pixel similar ao pixel semente é adicionado se for vizinho e não tiver sido designado para outra região mantendo no final a região conexa.

O algoritmo a seguir, baseado em crescimento de regiões, permite pintar áreas brancas delimitadas por outra cor. Ele recebe a imagem, as coordenadas da semente e a cor, e de inicio guarda o pixel em um fila, retira da fila, verifica se é branco então pinta e depois guarda seus vizinhos na fila e recomeça o processo ate não ter mais pixels na fila.

```
def pintar(a,li,ci,cor):
    import queue
    b=a.copy()
    q=queue.Queue()
    q.put(li)
    q.put(ci)
    while not q.empty():
        l=q.get()
        c=q.get()
        if all(b[l,c,:]==[255,255,255]):
            b[l,c]=cor
            q.put(l-1); q.put(c)
            q.put(l+1); q.put(c)
            q.put(l); q.put(c+1)
            q.put(l); q.put(c-1)

    return b;
```

O próximo algoritmo usado em imagens binárias recebe uma imagem as coordenadas da semente e retorna uma nova imagem somente com a área segmentada. Ao contrario do anterior em vez de uma fila usa um algoritmo auxiliar para adicionar novos pixels à região.

```
def region_growing(img, seed):
    seed_points = []
    outimg = np.zeros_like(img)
    seed_points.append((seed[0], seed[1]))
    processed = []
    while(len(seed_points) > 0):
        pix = seed_points[0]
        outimg[pix[0], pix[1]] = 255
        for coord in get8n(pix[0], pix[1], img.shape):
            if img[coord[0], coord[1]] != 0:
                outimg[coord[0], coord[1]] = 255
                if not coord in processed:
                    seed_points.append(coord)
                processed.append(coord)
        seed_points.pop(0)

    return outimg
```

Função auxiliar:

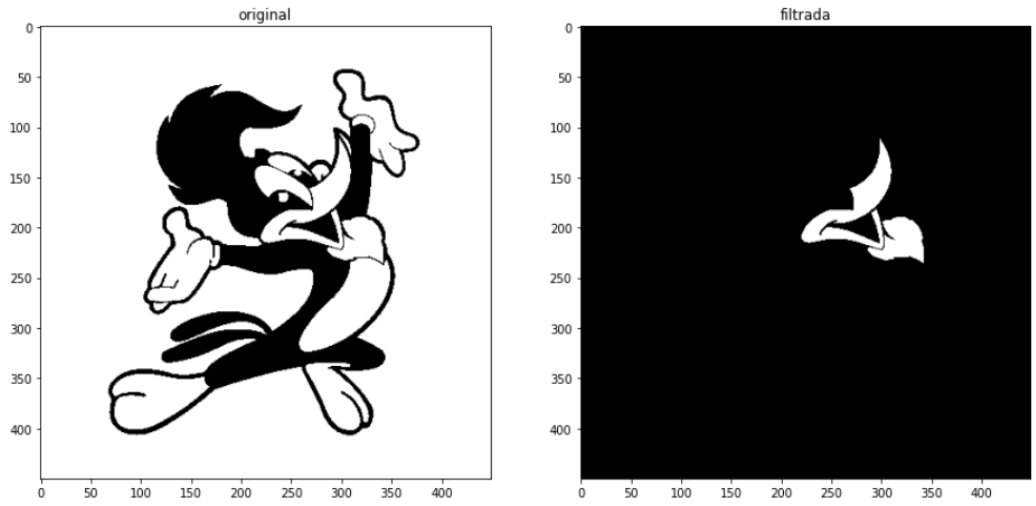
```
def get8n(x, y, shape):
    out = []
    maxx = shape[1]-1
    maxy = shape[0]-1

    #top left
    outx = min(max(x-1,0),maxx)
    outy = min(max(y-1,0),maxy)
    out.append((outx,outy))
```

```
#top center
outx = x
outy = min(max(y-1,0),maxy)
out.append((outx,outy))

#top right
outx = min(max(x+1,0),maxx)
outy = min(max(y-1,0),maxy)
out.append((outx,outy))

#left
outx = min(max(x-1,0),maxx)
outy = y
out.append((outx,outy))

#right
outx = min(max(x+1,0),maxx)
outy = y
out.append((outx,outy))

#bottom left
outx = min(max(x-1,0),maxx)
outy = min(max(y+1,0),maxy)
out.append((outx,outy))

#bottom center
outx = x
outy = min(max(y+1,0),maxy)
out.append((outx,outy))

#bottom right
outx = min(max(x+1,0),maxx)
outy = min(max(y+1,0),maxy)
out.append((outx,outy))

return out
```

A imagem do pica-pau depois de ser limiarizada pela função cv2.threshold e segmentada utilizando a função region_growing descrita anteriormente, a semente e adicionada nas coordenadas [166, 287] no interior do bico.

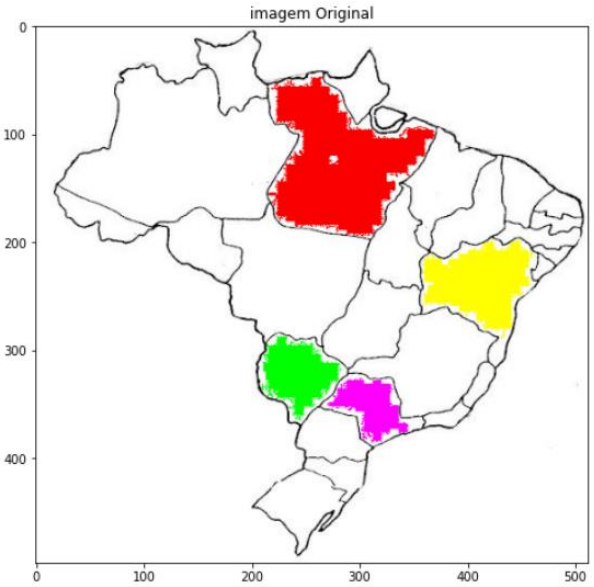
```
image = cv2.imread('imgs/picapau.jpg', 0)
ret, img = cv2.threshold(image, 128, 255, cv2.THRESH_BINARY)

out = region_growing(img, [166, 287])
```

Abaixo o resultado da segmentação, foi pintado regiões fora do bico devido alguma descontinuidade da linha do bico. O algoritmo leva um tempo considerável de processamento e varia conforme o tamanho da área a ser segmentada obviamente.



Resultado da função ‘pintar’ sobre a imagem do mapa do Brasil, onde quatro sementes espalhadas em alguns estados com cores diferentes

```
a = cv2.imread('imgs/mapa2.jpg', cv2.IMREAD_COLOR)
b = pintar(a,100,250,[250,0,0])
b = pintar(b,300,250,[0,255,0])
b = pintar(b,350,300,[255,0,255])
b = pintar(b,250,400,[255,255,0])

plt.figure(figsize=(12,8))
plt.title('imagem Original')
plt.imshow(b)
```

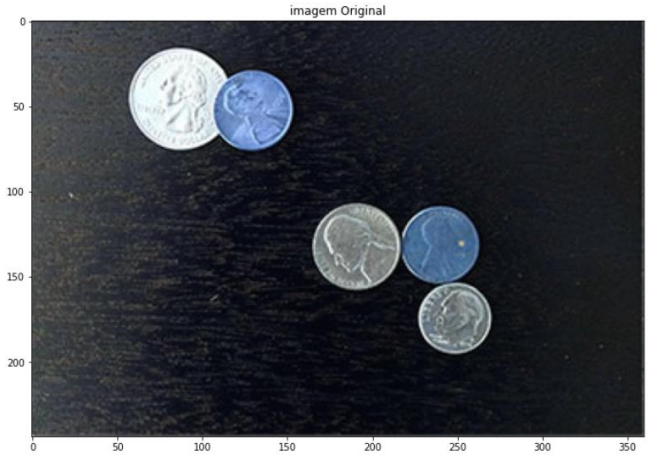


5. Watershed

Watershed, divisor de águas ou limites das bacias hidrográficas, é uma técnica de segmentação por similaridade baseada em crescimento de região aplicado a topografia, sua idéia principal consiste em preencher um relevo topográfico com água, de forma que a superfície seja dividida em regiões devido aos domínios de atração exercida em cada região (Vincent e Soille, 1991). As águas se acumulam nas bacias de captação, áreas do relevo mais baixas equivalentes a regiões mais escuras da imagem e conforme o nível de água vai aumentando mais evidente ficam os picos ou áreas mais claras da imagem e quando áreas com águas diferentes se encontram formam diques que são linhas divisórias separando essas regiões na imagem.

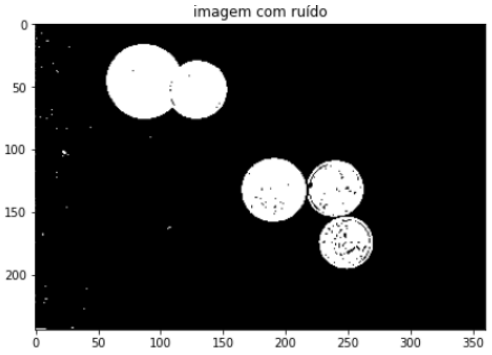
Para esse item foi usada uma imagem composta por moedas de tamanhos diferentes e em contato uma com as outras.

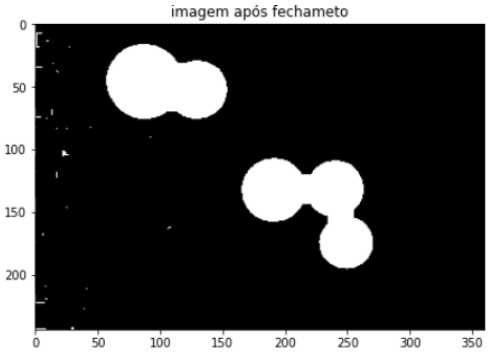
```
img = cv2.imread('imgs/coins.png')
img_gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```



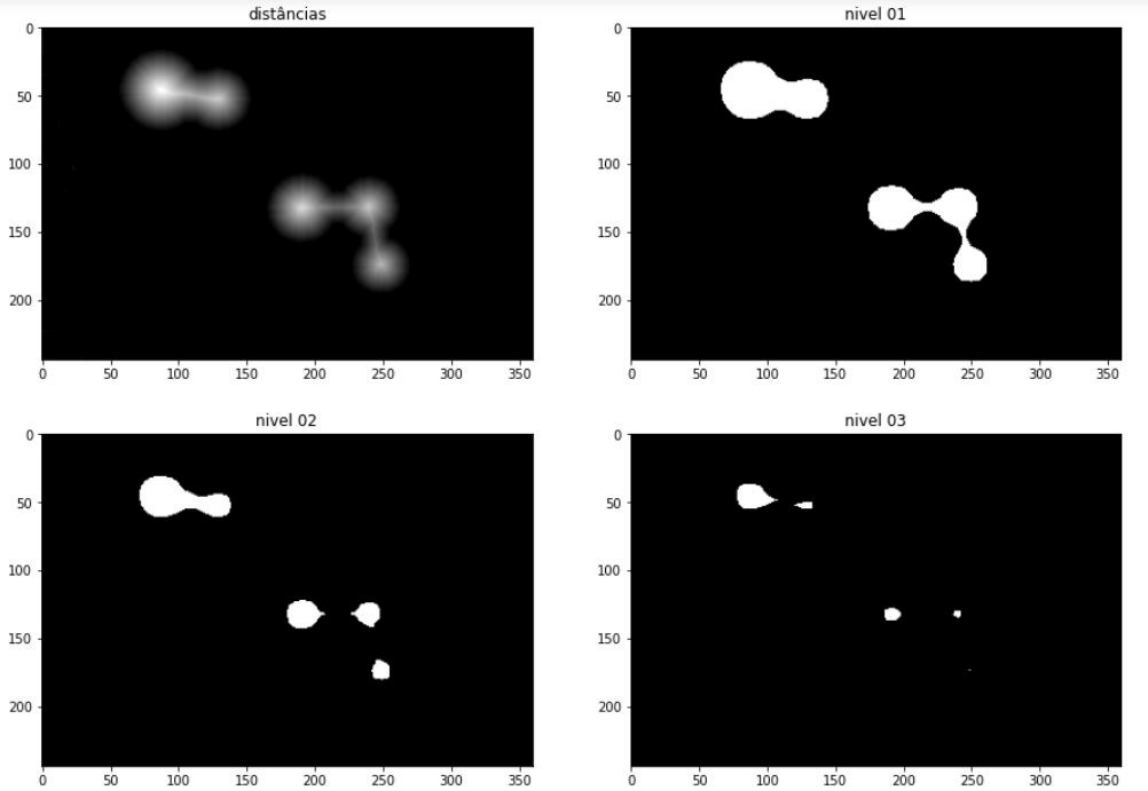
A imagem é linearizada através do método de Otsum, logo a seguir são removido os ruídos usando fechamento morfológico com um elemento estruturante de 2x2 e 3 interações.

```
ret, thresh = cv2.threshold(img_gray,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

kernel = np.ones((3,3),np.uint8)
closing = cv2.morphologyEx(thresh,cv2.MORPH_CLOSE, kernel, iterations = 3)
```





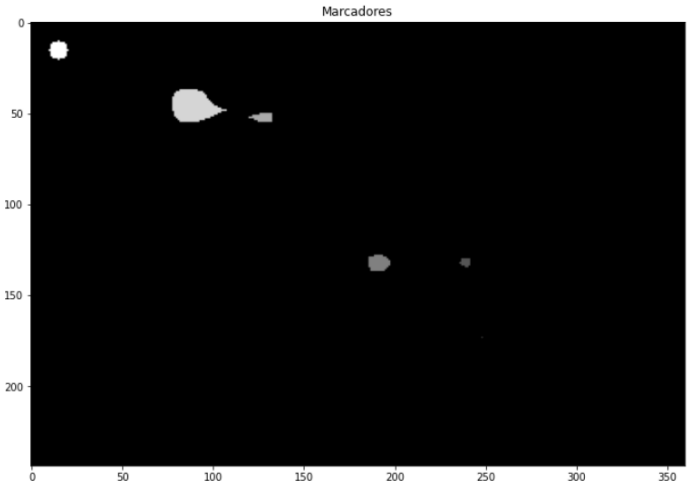
Em seguida é feito o calculo das distâncias euclidianas das moedas através do método cv2.distanceTransform (imagem superior esquerda) então é aplicado três limiarizações pra obter a posição dos marcadores variando o limiar (imagens nível 01, nível 02 e nível 03 abaixo), com o valor do limiar controlamos um nível da “inundação” para obter os marcadores mais ideais para o experimento.



O trecho abaixo cria os marcadores necessários para a aplicação do método cv2.watershed:

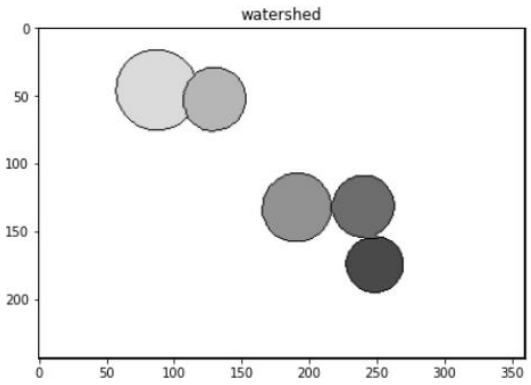
```
markers = np.zeros(dist.shape, dtype=np.int32)
dist_8u = dist3.astype('uint8')
contours, _ = cv2.findContours(dist_8u, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
for i in range(len(contours)):
    cv2.drawContours(markers, contours, i, (i+1), -1)

markers = cv2.circle(markers, (15,15), 5, len(contours)+1, -1)
```

Como pode ser observado na imagem abaixo foi usado o nível três que separa as duas moedas que estão sobrepostas em áreas diferentes para que seja formada a linha divisória (dique) entre elas. A marcação da menor moeda na parte inferior direita da imagem fica quase imperceptível mas esta lá.



Abaixo o resultado da aplicação da função cv2.watershed com os marcadores definidos acima necessário para separar as moedas sobrepostas na imagem

```
markers = cv2.watershed(img, markers)
img[markers == -1] = [255,0,0]
```



6. Transformada de Hough

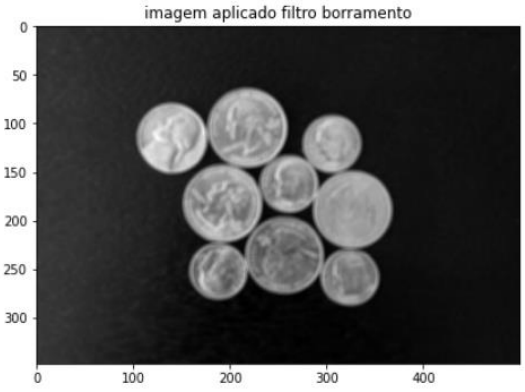
A Transformada de Hough é um método padrão para detecção de formas que são facilmente parametrizadas (linhas, círculos, elipses, etc.) em imagens digitalizadas baseada em segmentação em descontinuidades e de baixo custo computacional.

Abaixo e carregada à imagem que seja utilizada nesse item, trata-se de moedas de diferentes tamanhos em contato umas com as outras. A transformada de Hough vai permitir segmentar as moedas e individualizá-las pela sua forma circular.

```
img = cv2.imread('imgs/coins2.jpg', cv2.IMREAD_COLOR)
```

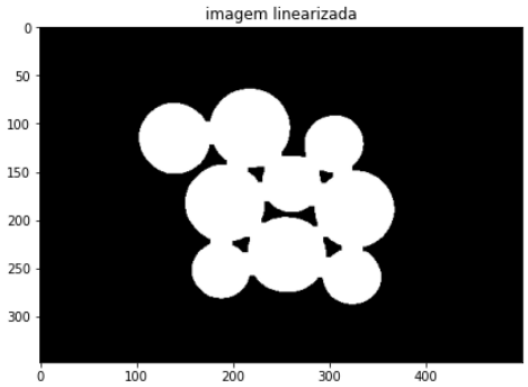
Para que seja aplicada a transformada de Hough e necessário a imagem seja tratada de forma a destacar seu contorno. Para isso ela foi primeiramente desfocada utilizando a função cv2.blur com um kernel 5x5.

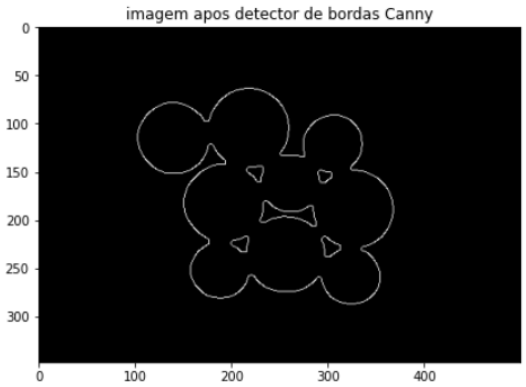
```
kernel = np.ones((5,5),np.uint8)

grey = cv2.cvtColor(img, cv2.COLOR_BGRA2GRAY)
grey_blurred = cv2.blur(grey, (5, 5))
```



Logo em seguida a imagem desfocada e linearizada, utilizando o método de Otsu, passa por fechamento morfológico para que sejam retirados os ruídos com elemento estruturante 3x3 e então e aplicada o detector de bordas de Canny.

```
ret, thresh = cv2.threshold(grey_blurred,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

kernel = np.ones((3,3),np.uint8)
closing = cv2.morphologyEx(thresh,cv2.MORPH_CLOSE, kernel, iterations = 2)

canny = cv2.Canny(closing,100,200)
```





De posse dos contornos da imagem, e então aplicado o método cv2.HoughCircles responsável por encontrar todas as formas circulares da imagem utilizando a transformada de Hough.

```
circles = cv2.HoughCircles(canny, cv2.HOUGH_GRADIENT, dp=1, minDist=50, param1=100, param2=10, minRadius=10, maxRadius=50)

if circles is not None:

    circles = np.uint16(np.around(circles))
    cimg = img.copy()

    for pt in circles[0, :]:
        a, b, r = pt[0], pt[1], pt[2]

        cv2.circle(cimg, (a, b), r, (255, 0, 0), 2)

        print("Centro ({:}, {:}), raio = {:}".format(a, b, r))

        cv2.circle(cimg, (a, b), 1, (0, 0, 255), 5)
```

Os círculos são detectados e impresso sobre uma copia da imagem original utilizando esse for acima, abaixo temos a posição dos círculos encontrados representando todas as moedas da imagem.

```
Centro (218, 104), raio = 40
Centro (140, 116), raio = 37
Centro (254, 236), raio = 40
Centro (192, 184), raio = 40
Centro (326, 188), raio = 40
Centro (304, 122), raio = 30
Centro (324, 256), raio = 30
Centro (186, 254), raio = 29
Centro (260, 162), raio = 30
```

A aplicação correta desse método foi tentativa e erro para encontrar os valores corretos para usar como parâmetros. Com destaque para o parâmetro dp que é razão inversa da resolução do acumulador em relação a imagem, funciona melhor entre 1 e 2, e o param2 que é o limite do acumulador, muito importante também sumir com os ruídos pois o esse método traça círculos através de conjuntos de pontos e curvas por isso a necessidade do pré-processamento.

Resultado da localização dos círculos na imagem original: