UNIVERSIDADE FEDERAL DE LAVRAS-UFLA

DISCIPLINA PGM848- VISÃO COMPUTACIONAL NO MELHORAMENTO DE PLANTAS

CAROLINE MARCELA DA SILVA

EWERTON LÉLYS RESENDE

MARIANA ANDRADE DIAS

THIAGO TAVARES BOTELHO

PROCESSAMENTO DE IMAGENS EM PYTHON- REO 3

Relatório apresentado como requisito avaliativo da disciplina de Visão Computacional no Melhoramento de Plantas, do Programa de Pós Graduação em Genética e Melhoramento de Plantas.

Prof. Dr. Vinícius Quintão Carneiro

Lavras-MG

Agosto/2020

EXERCÍCIO 01:

Selecione uma imagem a ser utilizada no trabalho prático e realize os seguintes processos utilizando as bibliotecas OPENCV e Scikit-Image do Python:

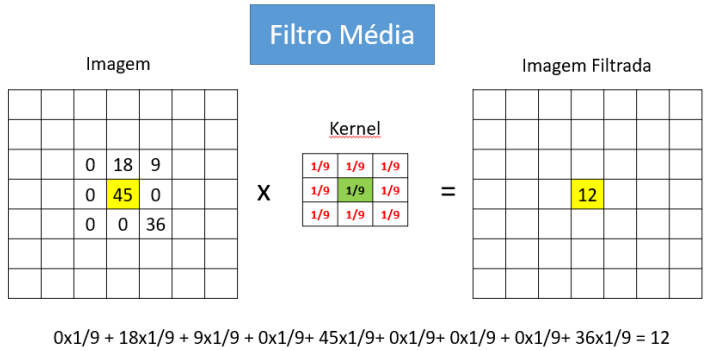
A imagem selecionada para a utilização neste roteiro será a mesma utilizada no roteiro anterior (REO2), a qual representa lesões características de antracnose em folhas de feijoeiro. A imagem está representada na figura 1.

Figura 1: Imagem selecionada.

a) Aplique o filtro de média com cinco diferentes tamanhos de kernel e compare os resultados com a imagem original;

A partir do processo de convolução, altera-se o valor do pixel de interesse, de modo a modificar a imagem original para utilizar posteriormente em diferentes tipos de processamento. Esta alteração do valor do pixel pode ser feita por meio da utilização de kernels, que levam em consideração os valores vizinhos ao seu valor de interesse.

O filtro de média consiste em um dos diferentes tipos de filtros existentes. Ele se baseia, para a alteração de um determinado pixel, na média dos valores vizinhos dos pixels da região de interesse. O cálculo da média é feito a partir do somatório da multiplicação do valor do kernel correspondente ao valor de cada pixel vizinho. Dessa forma, o kernel tem que ter dimensão semelhante a sua região de interesse e valores que propiciem a obtenção da média, como mostrado o exemplo do quadro a seguir:



Com o novo valor obtido, do pixel de interesse, o processo de convolução é realizado em toda a imagem, ou seja, o novo valor obtido, a partir da média dos vizinhos, vai passar por todos os valores da imagem, os alterando dessa forma. O resultado é a redução da quantidade de variação entre um pixel e outro, gerando uma nova imagem com menos ruídos, suavizada, ou na prática, mais embaçada.

Para a confecção desse exercício, chamamos a imagem.

```
#Preparação da imagem
arquivo = ("trabalho.png")
```

```
img_bgr = cv2.imread(arquivo,1)
img_rgb = cv2.cvtColor(imagem, cv2.COLOR_BGR2RGB)
```

A aplicação do filtro de média foi realizada a partir de 5 tamanhos kernels com valores distintos, pela função “cv.blur”, informando a imagem utilizada RGB (*img_rgb)* e a dimensão do kernel (*11 , 11)*. Por característica, o kernel deve apresentar valores ímpares.

```
#Filtros de média
img_fmedia_1 =  cv2.blur ( img_rgb , ( 11 , 11 ))
img_fmedia_2 =  cv2.blur ( img_rgb , ( 21 , 21 ))
img_fmedia_3 =  cv2.blur ( img_rgb , ( 31 , 31 ))
img_fmedia_4 =  cv2.blur ( img_rgb , ( 41 , 41 ))
img_fmedia_5 =  cv2.blur ( img_rgb , ( 51 , 51 ))
```

Em seguida, apresentamos as imagens resultantes dos diferentes tamanhos de kernels em uma figura “Imagens - Filtro de média”, a partir do comando “plt.subplot”.

```
#Apresentação das imagens
plt.figure('Imagens- Filtro de média')
plt.subplot(231)
plt.imshow(img_rgb)
plt.title("RGB")
plt.xticks([])
plt.yticks([])

plt.subplot(232)
plt.imshow(img_fmedia_1)
plt.title("KERNEL 11x11")
plt.xticks([])
plt.yticks([])

plt.subplot(233)
plt.imshow(img_fmedia_2)
plt.title("KERNEL 21x21")
plt.xticks([])
plt.yticks([])

plt.subplot(233)
```

```
plt.imshow(img_fmedia_3)
plt.title("KERNEL 31x31")
plt.xticks([])
plt.yticks([])

plt.subplot(235)
plt.imshow(img_fmedia_4)
plt.title("KERNEL 41x41")
plt.xticks([])
plt.yticks([])

plt.subplot(236)
plt.imshow(img_fmedia_5)
plt.title("KERNEL 51x51")
plt.xticks([])
plt.yticks([])

plt.show()
```

O resultado é apresentado na figura 2. Como podemos observar, a medida que aumentamos a dimensão do kernel, menor são o foco e a nitidez da imagem, chegando em um ponto, que não nos permite sequer distinguir seus objetos. Isso acontece porque aumentamos o número de vizinhos usados para o cálculo da média, alterando-a. Essa aplicação é útil para remover ruído e/ou conteúdos de alta frequência, como as bordas, por exemplo.

Figura 2: Filtros de média.

b) Aplique diferentes tipos de filtros com pelo menos dois tamanhos de kernel e compare os resultados entre si e com a imagem original.

Como anteriormente mencionado, outros filtros, além do filtro de média, estão disponíveis para alteração da imagem a partir do valor de um determinado pixel. Com esses filtros, podemos trabalhar a quantidade de ruídos, presença de bordas, nitidez, entre outras características da imagem, de acordo com um determinado objetivo.

Para isso, vamos aplicar em nossa imagem os filtros de média, gaussiano, mediana e bilateral, com dois tamanhos de kernel padronizados em todos os casos. As imagens 3 e 4, representam a aplicação dos filtros com kernel de 33x33 e 61x61, respectivamente. O código para essa aplicação está descrito a seguir para cada um dos filtros.

- *Filtro de Média*

O filtro de média está descrito na questão “a”. Aqui, só alteramos o argumento da função correspondente ao tamanho do kernel para 33x33 e 61x61.

```
# Filtro de média
img_filtro_media1 = cv2.blur(img_rgb,(33,33))
img_filtro_media2 = cv2.blur(img_rgb,(61,61))
```

- *Filtro Gaussiano*

O filtro Gaussiano, diferentemente do anterior que calcula a média simples, é baseado no cálculo da média ponderada dos vizinhos do pixel de interesse, de modo que os vizinhos mais próximos recebem peso maior e os mais distantes, menor. A função utilizada para essa aplicação é a “cv2.GaussianBlur” e como parâmetros utilizamos a imagem RGB, o tamanho do kernel e valor determinante do peso da média ponderada. Neste caso, o zero permite a determinação automática dos pesos, a partir de um desvio padrão da função gaussiana.

```
# Filtro Gaussiano
img_filtro_gaussiano1 = cv2.GaussianBlur(img_rgb,(33,33),0)
img_filtro_gaussiano2 = cv2.GaussianBlur(img_rgb,(61,61),0)
```

- *Filtro de Mediana*

Nos filtros anteriores, um novo valor de pixel foi calculado para a aplicação na imagem. Esse não é o caso do filtro de mediana. Aqui, a função “cv2.medianBlur” calcula a mediana dos valores de pixel que estão na vizinhança de interesse e o seleciona, ou seja, gera o filtro a partir de um valor já existente da região. O filtro de mediana remove melhor os ruídos do que os filtros de média e gaussiana. O filtro mediano preserva as bordas de uma imagem, mas não lida com ruído de manchas. Como parâmetros utilizamos a imagem RGB e o valor do kernel.

```
# Filtro de Mediana
img_filtro_mediana1 = cv2.medianBlur(img_rgb,33)
img_filtro_mediana2 = cv2.medianBlur(img_rgb,61)
```

- *Filtro Bilateral*

Os filtros anteriores tendem a borrar a borda, já no filtro bilateral isso não ocorre. Esse utiliza duas funções gaussianas, na primeira é levado em conta a ponderação dos pixels da vizinhança e na segunda função, a eliminação dos valores de pixels discrepantes, *outliers*, da vizinhança. A função aplicada foi “ cv2.bilateralFilter”, com a imagem RGB, o tamanho do kernel, o desvio padrão da primeira e segunda função gaussiana, ou seja, quanto de *outliers* vai ser eliminado na vizinhança considerando a intensidade dos pixels e sua distância na vizinhança.

```
# Filtro Bilateral
img_filtro_bilateral1 = cv2.bilateralFilter(img_rgb,33,33,21)
img_filtro_bilateral2 = cv2.bilateralFilter(img_rgb,61,61,21)
```

Como resultado, notamos que quanto maior o valor do kernel, mais perdemos em detalhes dos objetos da imagem. Mas de modo geral, o filtro bilateral, apresentou mais eficiente em eliminar os ruídos, sem perder o detalhismo das bordas, o que foi importante no nosso caso, onde são encontrados parte das lesões de antracnose.

Figura 3: Diferentes filtros com tamanho de kernel 33x33.

Figura 4: Diferentes filtros com tamanho de kernel 61x61.

Além dos filtros mostrados anteriormente, existem filtros específicos para detecção de bordas de imagens. Entre eles, os filtros Laplacian, Sobel e a técnica de Canny. Como em nossa imagem, as nervuras atingidas pela antracnose foram melhor destacadas com o filtro bilateral, que por característica, tende a suavizar as bordas, resolvemos investigar tais filtros mais específicos.

- *Laplacian*

A função “cv2.Laplacian” trabalha para essa aplicação, com o argumento da imagem inteiro de 64 bitz. Nesse caso, como difere da imagem inicial de 8 bitz, é necessário uma conversão para esse formato, transformando inicialmente a matriz em números absolutos, inteiros, pela função numpy “np.absolute” e em seguida para inteiro de 8 bitz, “np.uint8”.

```
#Laplacian
img_1 = cv2.Laplacian(img_rgb,cv2.CV_64F)
abs_164f = np.absolute (img_1)
img_1 = np.uint8(abs_164f)
```

- *Sobel*

A mesma transformação do item anterior precisa ser realizada para a aplicação deste filtro. O filtro de Sobel é aplicado separadamente nos dois eixos da imagem, x e y, através da função “cv2.Sobel”, utilizando como argumentos a imagem RGB, o parâmetro de 64 bitz, a indicação de qual eixo será analisado, com 0 e 1, e o tamanho do kernel.

```
#Sobel
img_sx = cv2.Sobel(img_rgb,cv2.CV_64F,0,1,ksize=5)  abs_sx64f
= np.absolute(img_sx)
img_sx = np.uint8(abs_sx64f)

img_sy = cv2.Sobel(img_rgb,cv2.CV_64F,1,0,ksize=5)
abs_sy64f = np.absolute(img_sy)
img_sy = np.uint8(abs_sy64f)
```

- *Técnica de Canny*

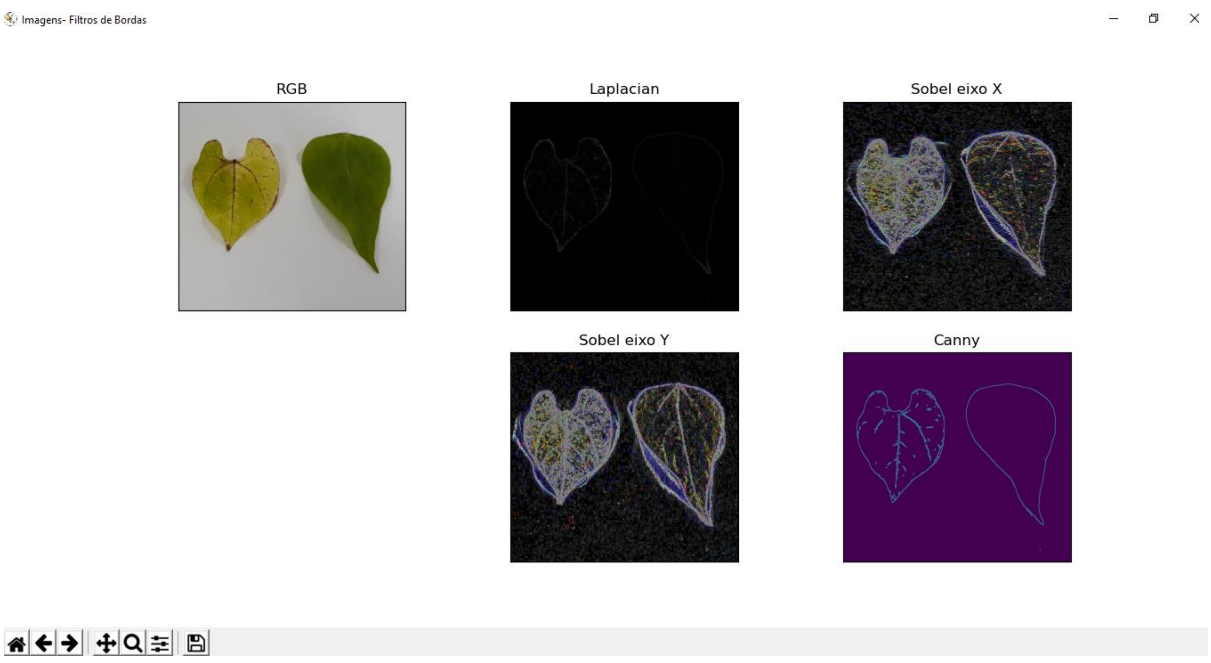
A técnica de Canny é baseada nos diferentes gradientes que os objetos de uma imagem recebem, visando distinguir, por esses, as bordas da imagem. A aplicação é feita com a função “cv2. Canny” que leva como argumentos a imagem RGB, o valor mínimo de gradiente (abaixo dele não é considerado borda) e o número máximo de

gradiente (acima dele, todos os objetos são considerados bordas). Os valores intermediários entre o valor mínimo e máximo são analisados pelo algoritmo por conectividade com os valores máximos, podendo ser reconhecidos como bordas ou não.

```
#Bordas Canny
edges = cv2.Canny(img_rgb, 100,200)
```

A figura 5 mostra o resultado da aplicação dos filtros. A partir dela, podemos afirmar a técnica de Canny com os valores de mínimo e máximo de 100 e 200, respectivamente, foi eficiente em distinguir não só as bordas, mas também as nervuras lesionadas, características da doença.

Figura 5: Filtros de borda.



c) Realize a segmentação da imagem utilizando o processo de limiarização. Utilizando o reconhecimento de contornos, identifique e salve os objetos de interesse. Além disso, acesse as bibliotecas Opencv e Scikit-Image, verifique as variáveis que podem ser mensuradas e extraia as informações pertinentes (crie e salve uma tabela com estes dados). Apresente todas as imagens obtidas ao longo deste processo.

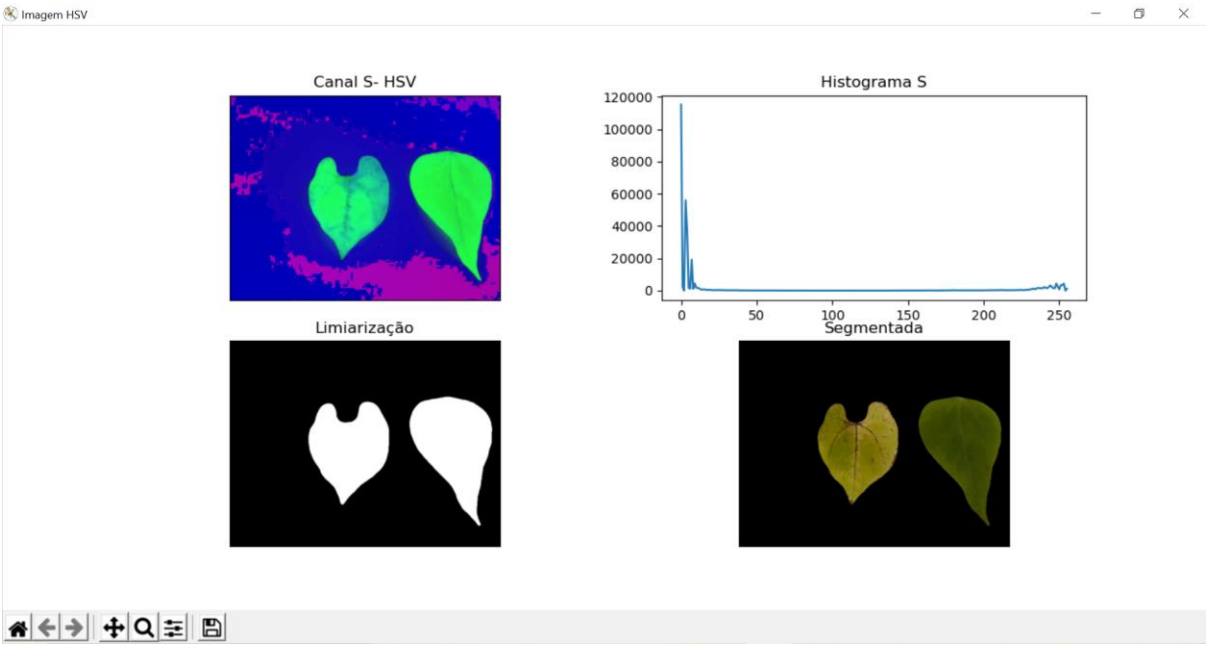
A segmentação é um processo que permite remover o fundo da imagem, destacando outros objetos de interesse. É possível segmentar uma imagem com filtro, de modo que a segmentação fique mais precisa, devido a eliminação de possíveis ruídos da imagem.

Em análises anteriores, verificamos o potencial dos canais “a” do sistema de cor Lab e do “Cb”, do sistema de cor YCrCb, para destacar os sintomas da antracnose nas nervuras da folha, nosso objeto principal. Entretanto, para essa questão, nosso objetivo foi identificar como objetos na imagem, a partir da técnica de contorno, as duas folhas, somente. Sem ainda fazer inferências sobre a área lesionada.

Para isso, vamos prosseguir com a utilização do filtro HSV para uma primeira segmentação, utilizando principalmente o canal S, como no relatório anterior. E vamos adicionar o filtro de mediana para suavizar os ruídos e permitir melhor segmentação. O código utilizado e as imagens resultantes estão apresentadas a seguir:

```
# Conversão para  o canal HSV:
img_HSV = cv2.cvtColor(img_bgr,cv2.COLOR_BGR2HSV)

# Partição dos canais
H,S,V = cv2.split(img_HSV)

# Filtro de mediana
S = cv2.medianBlur(S,5)

# Histograma do canal informativo
hist_S = cv2.calcHist([S],[0], None, [256],[0,256])

# Limiarização - Thresholding
(L, img_limiar) =
cv2.threshold(S,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
(Linv, img_limiar_inv) =
cv2.threshold(S,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

# Obtendo imagem segmentada
img_segmentada =
cv2.bitwise_and(img_rgb,img_rgb,mask=img_limiar)
```

Figura 6: Segmentação do canal S com filtro de mediana pela técnica de OTSU.



A partir da função “cv2.findContours” nós obtemos os contornos dos objetos da imagem. Criamos uma variável para abrigar a cópia da imagem segmentada, para que possa ser modificada, sem causar perdas dos processos anteriores, e aplicamos a função, que recebeu como parâmetros a nova variável “máscara”, informação sobre a hierarquia dos objetos dentro da imagem, “cv2.RETR_TREE” e por último, a informação de como o contorno deve ser obtido, “cv2.CHAIN_APPROX_SIMPLE”. Essa função gera duas informações de saída, todos os contornos da imagem, “cnts” e a hierarquia dos objetos da imagem, “h”.

```
# Contorno das folhas de cada objeto da imagem
mascara = img_limiar_inv.copy()
cnts,h = cv2.findContours(mascara,
cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

Para enquadrar, recortar e salvar os diferentes objetos encontrados, individualmente, utilizamos a estrutura em loop “for.. in..” do numpy associado com a função “enumerate”. Dessa forma, conseguimos ler a informação de cada contorno “c” e sua posição “i”.

A função “cv2.boundingRect” permite separar os objetos da imagem e salvá-los individualmente. Esse “recorte” é feito por um retângulo que engloba o contorno dos objetos. Como parâmetro da função indicamos o contorno “c” do objeto em cada posição, obtido pelo “for”. A função nos retorna quatro variáveis, sendo elas:

x- onde começa o objeto no eixo x;

y- onde começa o objeto no eixo y;

w- largura;

h- altura;

Essas informações facilitam a delimitação do recorte do objeto na imagem. Realizamos o recorte na imagem binarizada e na imagem segmentada RGB. Em todas elas delimitamos o objeto da seguinte forma:

y:y + h, x:x + w - Ou seja, para o recorte abranger todo o eixo Y do objeto, em toda sua altura, e todo o eixo X em toda sua largura.

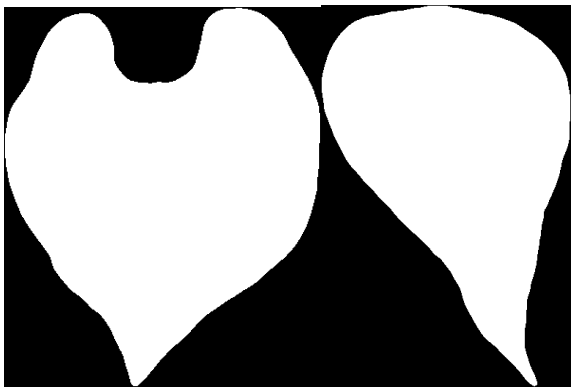
Por questões de compatibilidade com o formato salvo pelo programa “Open CV”, é preciso converter a imagem segmentada RGB e seus recortes para o formato BGR. Para salvar as imagens no computador, utilizamos a função “cv2.write” com os argumentos, nome do arquivo, sua extensão e o objeto utilizado.

```
for (i, c) in enumerate(cnts):

  (x, y, w, h) = cv2.boundingRect(c)
  obj = img_limiar[y:y+h,x:x+w]
  obj_rgb = img_segmentada[y:y+h,x:x+w]
  obj_bgr = cv2.cvtColor(obj_rgb,cv2.COLOR_RGB2BGR)
  cv2.imwrite('s'+str(i+1)+'.png',obj_bgr)
  cv2.imwrite('sb'+str(i+1)+'.png',obj)
```

As figuras 7 e 8 mostram os objetos que foram reconhecidos e recortados a partir da imagem.

FIGURA 7: Imagem original e objetos recortados salvos em formato BGR (sb).

FIGURA 8: Imagem original e objetos recortados e salvos em formato binarizado (s).



Para coletar informações sobre os objetos recortados utilizamos a função “regionprops” do pacote Scikit image. Tal função permite calcular características da imagem como perímetro, área, eixo maior e menor, entre outras. Para calcular a dimensão do recorte utilizamos a função “np.shape”. Para obtenção de medidas, quando utilizado funções do Opencv, o material analisado é a imagem com os objetos contornados. Já para as características obtidas com o Regionprops tem como imagem base, a imagem binária. Utilizamos também a biblioteca “Pandas” para criar uma tabela que abrigasse essas características, com a função “pd.dataframe”. O resultado da caracterização física das medidas dos objetos é apresentado na tabela 1.

```
  area = cv2.contourArea(c)
razao = round((h/w), 2)
perim = round(cv2.arcLength(c, True), 2)
tam_ret = np.shape(obj)
```

```
regiao = regionprops(obj)
rm = round(regiao[0].minor_axis_length, 2)
rmai = round(regiao[0].major_axis_length, 2)
cen = regiao[0].centroid
dimen += [[str(i + 1), str(h), str(w), str(area), str(razao),
        str(perim), str(tam_ret), str(rm), str(rmai),
str(cen)]]

dados_folhas = pd.DataFrame(dimen)
dados_folhas = dados_folhas.rename(columns={0: 'FOLHA', 1:
'ALTURA', 2: 'LARGURA', 3: 'AREA', 4: 'RAZAO',
                                                    5:
'PERIMETRO', 6: 'TAMANHO RET', 7:'EIXO MENOR', 8:'EIXO
MAIOR',

     9:'CENTROIDE'})
dados_folhas.to_csv('medidas.csv', index=False)
```

Tabela 1: Informações sobre medidas físicas dos objetos encontrados.

FOLHA	ALTURA	LARGURA	ÁREA	RAZÃO	PERÍMETRO	TAMANHO RETÂNGULO	EIXO MENOR	EIXO MAIOR	CENTROIDE
1	250	195	31258	1.28	799.92	(250, 195)	198.77	219.86	(109.35, 98.57)
2	314	200	35947.5	1.57	865.34	(314, 200)	178.47	286.4	(114.87, 108.50)

Além das características sobre tamanho e dimensão da imagem, podemos obter também informações em nível de cor das mesmas. Essas informações são apresentadas a seguir, na imagem 9.

```
  print('Medidas de Cor')
  min_val_r, max_val_r, min_loc_r, max_loc_r =
cv2.minMaxLoc(obj_rgb[:,:,0], mask=obj)
  print('Valor Mínimo no R: ', min_val_r, ' - Posição: ',
min_loc_r)
  print('Valor Máximo no R: ', max_val_r, ' - Posição: ',
max_loc_r)
  med_val_r = cv2.mean(obj_rgb[:,:,0], mask=obj)
  print('Média no Vermelho: ', med_val_r)

  min_val_g, max_val_g, min_loc_g, max_loc_g =
cv2.minMaxLoc(obj_rgb[:, :, 1], mask=obj)
  print('Valor Mínimo no G: ', min_val_g, ' - Posição: ',
min_loc_g)
```

```
  print('Valor Máximo no G: ', max_val_g, ' - Posição: ',
max_loc_g)
  med_val_g = cv2.mean(obj_rgb[:,:,1], mask=obj)
  print('Média no Verde: ', med_val_g)

  min_val_b, max_val_b, min_loc_b, max_loc_b =
cv2.minMaxLoc(obj_rgb[:, :, 2], mask=obj)
  print('Valor Mínimo no B: ', min_val_b, ' - Posição: ',
min_loc_b)
  print('Valor Máximo no B: ', max_val_b, ' - Posição: ',
max_loc_b)
  med_val_b = cv2.mean(obj_rgb[:,:,2], mask=obj)
  print('Média no Azul: ', med_val_b)
  print('-'*50)
```

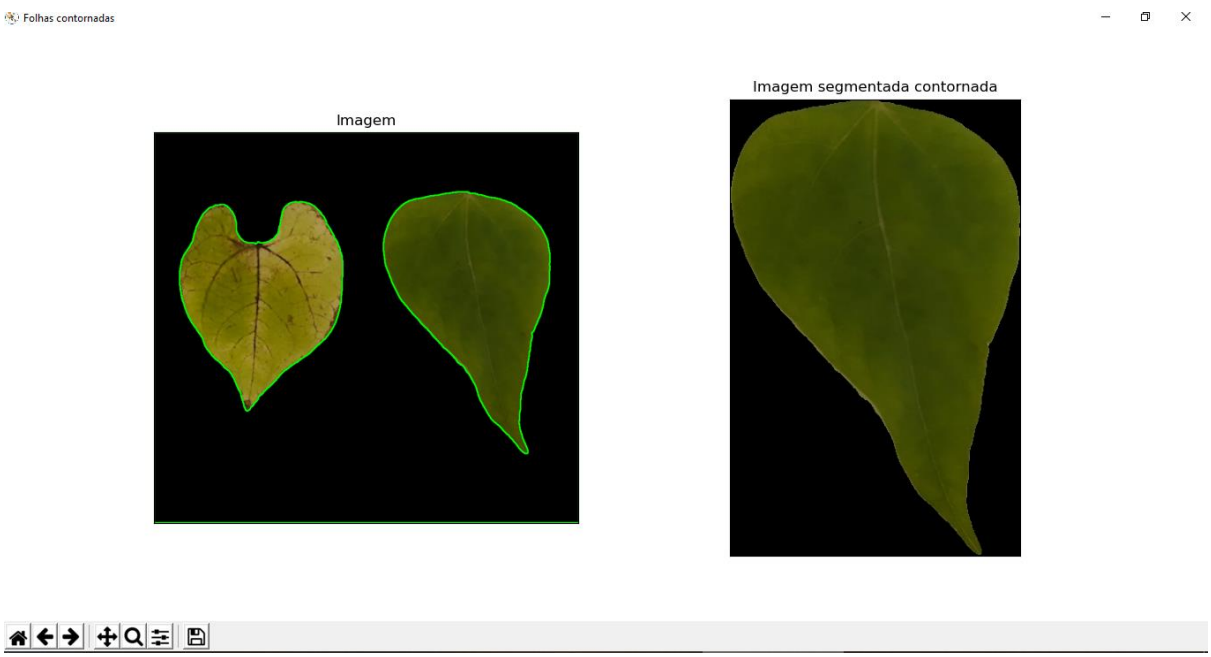
Imagem 9: Medidas do sistema de cor dos objetos selecionados.

```
Medidas de Cor
Valor Mínimo no R:  33.0  - Posição:  (94, 49)
Valor Máximo no R:  173.0  - Posição:  (178, 35)
Média no Vermelho:  (128.03560577288263, 0.0, 0.0, 0.0)
Valor Mínimo no G:  18.0  - Posição:  (94, 50)
Valor Máximo no G:  156.0  - Posição:  (118, 38)
Média no Verde:  (115.47297126218508, 0.0, 0.0, 0.0)
Valor Mínimo no B:  0.0  - Posição:  (51, 11)
Valor Máximo no B:  88.0  - Posição:  (118, 35)
Média no Azul:  (18.844695531079882, 0.0, 0.0, 0.0)
Medidas de Cor
Valor Mínimo no R:  45.0  - Posição:  (89, 84)
Valor Máximo no R:  125.0  - Posição:  (178, 169)
Média no Vermelho:  (63.84324577344567, 0.0, 0.0, 0.0)
Valor Mínimo no G:  54.0  - Posição:  (88, 85)
Valor Máximo no G:  127.0  - Posição:  (190, 140)
Média no Verde:  (74.7953631807919, 0.0, 0.0, 0.0)
Valor Mínimo no B:  0.0  - Posição:  (44, 14)
Valor Máximo no B:  75.0  - Posição:  (178, 169)
Média no Azul:  (4.748141417478936, 0.0, 0.0, 0.0)
Total de objetos:  2
```

Em seguida, apresentamos os contornos a partir da função “plt.figure”. Para fazer essa demarcação, utilizamos a função “cv.2drawContours”, indicando a imagem, os objetos identificados, a indicação para contornar todos os objetos (-1), uma tupla indicando os valores do sistema RGB que devem ser utilizados no contorno, ou seja, qual será a cor da linha e por último o valor da espessura da mesma. O resultado é mostrado na figura 10.

```
#Apresentando os contornos
seg = img_segmentada.copy()
cv2.drawContours(seg,cnts,-1,(0,255,0),2)

plt.figure('Objetos')
plt.subplot(1,2,1)
plt.imshow(seg)
plt.xticks([])
plt.yticks([])
plt.title('Objetos')

plt.subplot(1,2,2)
plt.imshow(obj_rgb)
plt.xticks([])
plt.yticks([])
plt.title('Objetos')
plt.show()
```

Figura 10: Objetos contornados.



d) Utilizando máscaras, apresente o histograma somente dos objetos de interesse.

Com os objetos contornados e recortados é possível criar histogramas específicos para cada um deles, através do uso de máscaras. Em nossas análises, foram reconhecidos dois principais objetos, além da imagem inteira, referente as duas folhas da imagem, como mostrado nas figuras 7 e 8.

Para a obtenção dos histogramas, criamos uma figura para abriga-los, e a partir da estrutura de repetição “for.. in..”, e com a função “cv2.boundingRect” aplicamos um retângulo ao redor dos objetos e os recortamos, como na questão anterior. A diferença é que após esse processo, criamos histogramas, com a função “cv2.calcHist”, para cada um dos canais da imagem RGB segmentada, utilizando a imagem binarizada obtida pela limiarização de OTSU.

```
plt.figure('Objetos')
for (i, c) in enumerate(cnts):

  (x,y,w,h) = cv2.boundingRect(c)
  print('Objeto #%d' % (i+1))
  print(cv2.contourArea(c))
  obj = img_limiar[y:y+h,x:x+w]
  obj_rgb = img_segmentada[y:y+h,x:x+w]
  grafico = True
  if grafico == True:

     hist_r =
cv2.calcHist([obj_rgb[:,:,0]],[0],obj,[256],[0,256])
     hist_g = cv2.calcHist([obj_rgb[:,:,1]], [0],obj, [256],
[0, 256])
     hist_b = cv2.calcHist([obj_rgb[:,:,2]], [0],obj, [256],
[0, 256])

     plt.subplot(3,3,2)
     plt.imshow(obj_rgb)
     plt.title('Objeto: ' + str(i+1))

     plt.subplot(3, 3, 4)
     plt.imshow(obj_rgb[:,:,0],cmap='gray')
     plt.title('Objeto: ' + str(i + 1))

     plt.subplot(3, 3, 5)
     plt.imshow(obj_rgb[:,:,1],cmap='gray')
     plt.title('Objeto: ' + str(i + 1))
```

```
      plt.subplot(3, 3, 6)
      plt.imshow(obj_rgb[:,:,2],cmap='gray')
      plt.title('Objeto: ' + str(i + 1))

      plt.subplot(3, 3, 7)
      plt.plot(hist_r, color='r')
      plt.title("Histograma - R")
      plt.xlim([0, 256])
      plt.xlabel("Valores Pixels")
      plt.ylabel("Número de Pixels")

      plt.subplot(3, 3, 8)
      plt.plot(hist_g, color='g')
      plt.title("Histograma - R")
      plt.xlim([0, 256])
      plt.xlabel("Valores Pixels")
      plt.ylabel("Número de Pixels")

      plt.subplot(3, 3, 9)
      plt.plot(hist_r, color='r')
      plt.title("Histograma - B")
      plt.xlim([0, 256])
      plt.xlabel("Valores Pixels")
      plt.ylabel("Número de Pixels")

      plt.show()

  else:
      pass
```

Os histogramas dos canais do sistema de cor RGB de cada objeto é apresentado nas figuras 11 e 12. Em ambos os casos, para todos os canais, foi observado o mesmo padrão no histograma, com um único pico, representando o objeto recortado.

Figura 11: Histograma do objeto 1 da imagem.

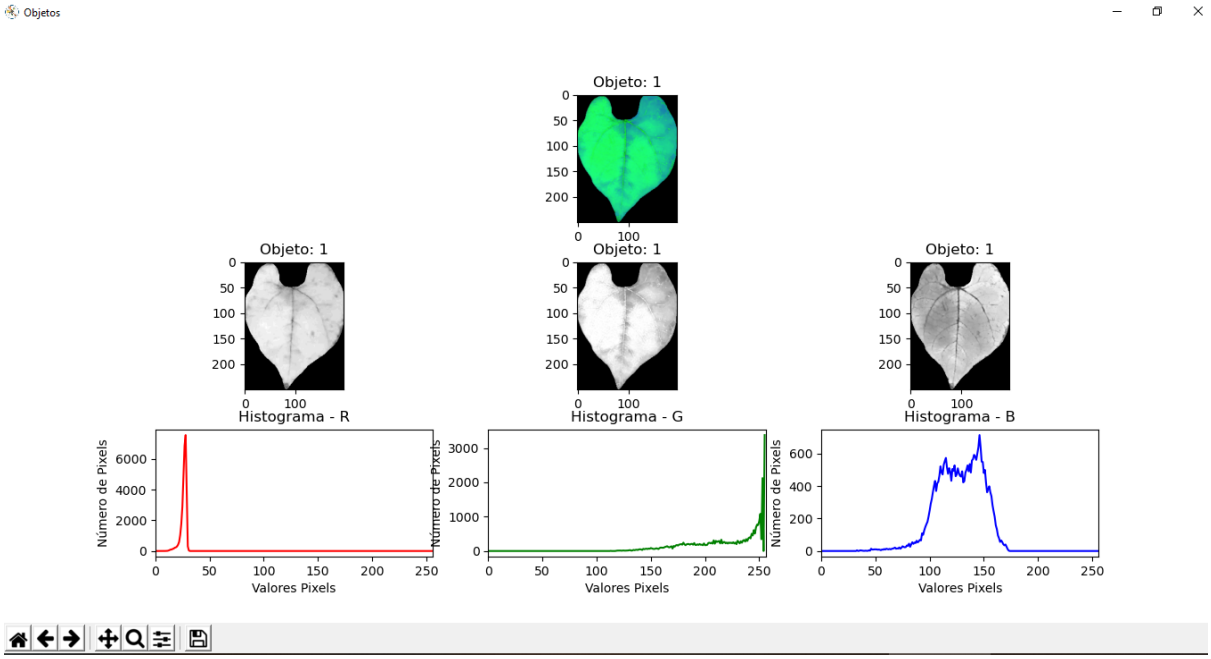
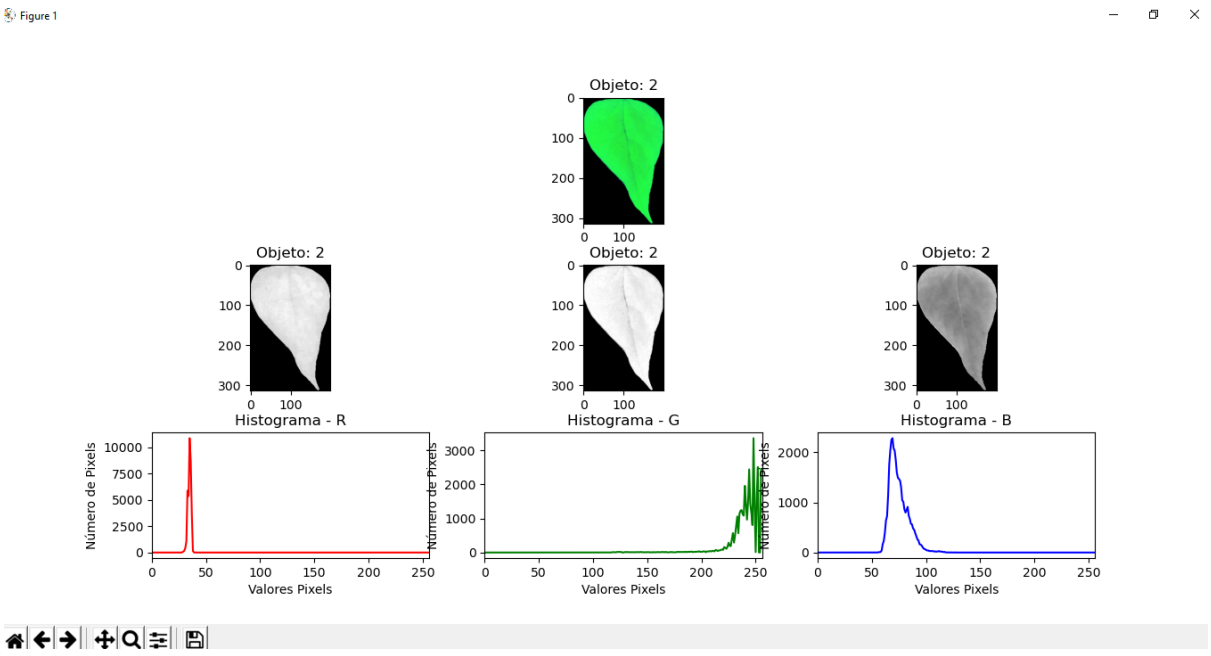


Figura 12: Histograma do objeto 2 da imagem.



e) Realize a segmentação da imagem utilizando a técnica de k-means. Apresente as imagens obtidas neste processo.

K-mens consiste em uma técnica baseada em aprendizado de máquina, sendo assim, mais robusta e utilizada quando não se é possível segmentar com técnicas mais simples, como a de limiarização, por exemplo. A diferença entre essas duas, é

a técnica k-means permite separar a imagem em até cinco regiões diferentes, a partir do agrupamento dos pixels que apresentam cores semelhantes.

O primeiro passo para sua utilização, foi realizar a formatação da imagem, para que ela pudesse ser trabalhada no k-means. Nesta técnica, se trabalha nas linhas cada pixel e na coluna as variáveis que são os canais verde, vermelho e azul, do sistema RGB. O primeiro passo para tal formatação foi criar uma matriz, chamada de “pixel_values”. Ela recebeu a nossa imagem RGB, e com o auxílio da função “reshape” transformamos o formato da nossa imagem. O primeiro parâmetro informado (-1) se refere as linhas, ele irá pegar todos os pixels da imagem e colocar em formato de linha e 3 colunas, correspondentes aos 3 canais de cores. Essa matriz é um inteiro de 8 bits, posteriormente, fizemos a conversão dessa matriz para decimal, com o auxílio da função da biblioteca numpy, “np.float32”, e por fim, fizemos a observação da dimensão dessa matriz gerada.

```
pixel_values = img_rgb.reshape((-1, 3))
pixel_values = np.float32(pixel_values)
print('-'*80)
print('Dimensão Matriz: ',pixel_values.shape)
print('-'*80)
```

Em seguida, como essa técnica consiste em um processo interativo, precisamos estabelecer um critério para cessar seu andamento, ou de modo contrário, ela rodaria infinitamente. Para isso, utilizamos dois critérios de parada, um critério de precisão em relação ao centróide (EPS) e o número máximo de interações que ele vai rodar. Posteriormente, estabelecemos o número de grupos, no caso, dois grupos, o fundo e a folha de interesse. Então utilizamos a função “cv2.kmeans”, que é uma função que irá receber alguns parâmetros, como: a matriz gerada anteriormente, o número de grupos, o nome dos grupos (informamos “None” pois não iremos informar o nome desses grupos, ele vai chamar de 0 e 1, como temos dois grupos) e o critério de parada que vamos utilizar. Em seguida, estabelecemos o número de vezes que vamos executar a iniciação (10 vezes) para encontrar o melhor centróide que vai ser chutado aleatoriamente. A partir desse comando, teremos três respostas, a soma de quadrado da distância de cada pixel ao centróide (dist), o

número de labels, que está relacionado com a classificação dos pixels e de como eles estão agrupados, podemos utilizar essa informação de grupos para tentar enxergar cada grupo separadamente. Em seguida, faremos a conversão desses labels para um vetor único,que agora é visto, como se fosse uma lista. Cada posição dessa lista vai me dar 0 ou 1.

```
criterio = (cv2.TERM_CRITERIA_EPS +
cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
# Número de Grupos (k)
k = 2
dist, labels, (centers) = cv2.kmeans(pixel_values, k, None,
criterio, 10, cv2.KMEANS_RANDOM_CENTERS)
print('-'*80)
print('SQ das Distâncias de Cada Ponto ao Centro: ', dist)
print('-'*80)
print('Dimensão labels: ', labels.shape)
print('Valores únicos: ',np.unique(labels))
print('Tipo labels: ', type(labels))

labels = labels.flatten()
print('-'*80)
print('Dimensão flatten labels: ', labels.shape)
print('Tipo labels (f): ', type(labels))
print('-'*80)
```

Para enxergarmos os valores de cada labels precisaremos utilizar a função “np.unique” que vai retornar quais são os valores únicos da imagem (0 e 1) e vai retornar a contagem de valores de cada grupo. Posteriormente, transformamos os valores únicos e as contagens que estavam em vetor linhas para vetor colunas, com auxílio da função “np.reshape”. Em seguida, fizemos a concatenação dos valores de únicos e de contagens e imprimimos o histograma (Figura 13). Outra informação que podemos imprimir, seriam os centróides encontrados, cada linha representa uma classe (0 e 1) e as colunas representam as informações de cada canal (vermelho, verde e azul), esses valores estão em decimais, então faremos a transformação para inteiros de oito bits com auxílio da função “np.uint8” (figura y).

```
val_unicos,contagens = np.unique(labels,return_counts=True)
```

```
val_unicos = np.reshape(val_unicos,(len(val_unicos),1))
contagens = np.reshape(contagens,(len(contagens),1))
hist = np.concatenate((val_unicos,contagens),axis=1)
print('Histograma')
print(hist)
print('-'*80)
print('Centroides Decimais')
print(centers)
print('-'*80)
# Conversão dos centroides para valores de interos de 8
digitos
centers = np.uint8(centers)
print('-'*80)
print('Centroides uint8')
print(centers)
print('-'*80)
```

Para se obter uma matriz de cada pixel, contendo as informações dos centróides criamos uma matriz segmentada que irá conter os centers e dentro dela os labels. Pedimos então, para que naquelas posições em que os labels fossem zero, substituir os centers de 0 (179,178,176) e assim respectivamente para o 1. Naquela posição que o pixel for rotulado de zero, ele recebeu o valor do centróide dele. Pedimos para imprimir a dimensão e uma parte da matriz segmentada (figura 14) e observamos que estamos pegando a cor média e impondo em cada um dos pixels.

```
# Conversão dos pixels para a cor dos centroides
matriz_segmentada = centers[labels]
print('-'*80)
print('Dimensão Matriz Segmentada: ',matriz_segmentada.shape)
print('Matriz Segmentada')
print(matriz_segmentada[0:5,:])
print('-'*80)
```

Essa matriz segmentada não está em formato de imagem, então precisamos fazer essa transformação. Para isso, utilizamos a função “reshape”, e essa imagem, voltou a ter a dimensão da imagem que carregamos no início (img_rgb.shape) que é a imagem denominada de rótulos, nessa imagem podemos observar os dois grupos (figura 15).

Para capturar cada imagem de cada região separadamente, primeiro fizemos uma cópia da imagem original , e transformamos ela em formato de linha e 3 colunas, correspondentes aos 3 canais de cores. Com essa matriz nova, criamos uma nova matriz onde pegamos todos os pixels diferentes de 0 para um grupo e todos os pixels diferentes de 1 para o outro grupo. Em seguida, transformamos essas matrizes em imagem.

```
# Reformatar a matriz na imagem de formato original
img_segmentada1 = matriz_segmentada.reshape(img_rgb.shape)

# Grupo 1
original_01 = np.copy(img_rgb)
matriz_or_01 = original_01.reshape((-1, 3))
matriz_or_01[labels != 0] = [0, 0, 0]
img_final_01 = matriz_or_01.reshape(img_rgb.shape)

# Grupo 2
original_02 = np.copy(img_rgb)
matriz_or_02 = original_02.reshape((-1, 3))
matriz_or_02[labels != 1] = [0, 0, 0]
img_final_02 = matriz_or_02.reshape(img_rgb.shape)
```

Em seguida fazemos a apresentação das imagens.

```
# Apresentar Imagem
plt.figure('Imagens')
plt.subplot(2,2,1)
plt.imshow(img_rgb)
plt.title('ORIGINAL')
plt.xticks([])
plt.yticks([])

plt.subplot(2,2,2)
plt.imshow(img_segmentada1)
plt.title('ROTULOS')
plt.xticks([])
plt.yticks([])

plt.subplot(2,2,3)
plt.imshow(img_final_01)
plt.title('Grupo 1')
plt.xticks([])
```

```
plt.yticks([])

plt.subplot(2,2,4)
plt.imshow(img_final_02)
plt.title('Grupo 2')
plt.xticks([])
plt.yticks([])

plt.show()
```

Figura 13- Informações sobre as matrizes e histograma dos valores de únicos e de contagens.

```
--------------------------------------------------------------------------------
Dimensão Matriz:  (326529, 3)
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
SQ das Distâncias de Cada Ponto ao Centro:  231133496.35343647
--------------------------------------------------------------------------------
Dimensão labels:  (326529, 1)
Valores únicos:  [0 1]
Tipo labels:  <class 'numpy.ndarray'>
--------------------------------------------------------------------------------
Dimensão flatten labels:  (326529,)
Tipo labels (f):  <class 'numpy.ndarray'>
--------------------------------------------------------------------------------
Histograma
[[     0 257114]
 [     1  69415]]
--------------------------------------------------------------------------------
```

Figura 14- Informações sobre os centróides e Matriz segmentada.

```
Centroides Decimais
[[179.06982  178.2428   176.72485 ]
 [ 94.18222   94.16587   12.788259]]
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Centroides uint8
[[179 178 176]
 [ 94  94  12]]
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Dimensão Matriz Segmentada:  (326529, 3)
Matriz Segmentada
[[179 178 176]
 [179 178 176]
 [179 178 176]
 [179 178 176]
 [179 178 176]]
```

O grupo 1 provavelmente está relacionado com o fundo da imagem, e o grupo dois relacionado com as folhas. Podemos observar pela imagem, que quando tratamos do grupo 1, todos os pixel que foram diferentes de 0, foram apagados, e quando observamos o grupo dois, todos os pixels que forem diferentes de 1 foram apagados, no nosso caso, o fundo.

Figura 15- Objetos separados em cada grupo.

f) Realize a segmentação da imagem utilizando a técnica de watershed. Apresente as imagens obtidas neste processo.

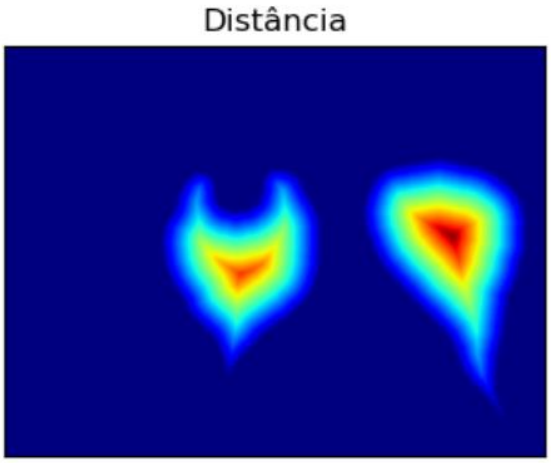
A técnica de watershed permite, por característica, fazer a segmentação de objetos conectados. Para sua aplicação, convertemos a imagem para o sistema HSV (“cv2.cvtcolor”), particionamos os seus canais (“cv2.split”) e aplicamos com a função “cv2.threshold” o limiar de OTSU na imagem.

```
img_HSV = cv2.cvtColor(img_bgr,cv2.COLOR_BGR2HSV)
h,s,v = cv2.split(img_HSV)
limiar, mascara =
cv2.threshold(s,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
```

Para realizar a segmentação de Watershed é necessário encontrar os picos dentro da nossa imagem. Esses picos, são encontrados com base na distância dos pontos brancos da nossa máscara em relação ao fundo preto. Para isso, utilizaremos o comando “ndimage.distance_transform_edt” que realiza o cálculo da distância euclidiana de cada ponto da folha aos pontos mais escuros. Com isso, obtivemos uma imagem de distâncias na escala de cores “JET”, onde os pontos com vermelho mais intenso são aqueles do centro das folhas.

```
img_dist = ndimage.distance_transform_edt(mascara)
```

Figura 16: Imagem em escala de cor JET



Para encontrar dentro da região vermelha mais intenso, o local de maior valor, ou seja, o pico, utilizaremos a função “peak_local_max”. Essa função irá nos retornar uma matriz booleana da mesma dimensão da nossa imagem, com verdadeiros e falsos. Nos picos, o valor da matriz será verdadeiro e fora deles será falso.

```
max_local = peak_local_max(img_dist, indices=False,
min_distance=100, labels=mascara)

print('Número de Picos')
print(np.unique(max_local,return_counts=True))
print('-'*50)
```

Figura 17: Número de picos encontrados na imagem.

```
Número de Picos
(array([False,  True]), array([326527,      2], dtype=int64))
```

De acordo com a imagem acima (Figura 17), podemos observar que nossa matriz booleana apresenta valores falsos e verdadeiro sendo que, são 326527 valores falsos e 2 valores verdadeiro. Como são dois pixels verdadeiros, a princípio, temos que nossa imagem apresenta 2 picos, ou seja, duas folhas.

Tendo identificado o número de picos, agora temos que fazer a marcação de cada pico. Desta forma, cada pico será demarcado com um valor e para isso

utilizaremos a função “ndimage.label”. O comando “structure” irá fazer uma análise de conectividade.

```
marcadores,n_marcadores = ndimage.label(max_local,
structure=np.ones((3, 3)))

print('Análise de conectividade - Marcadores')
print(np.unique(marcadores,return_counts=True))
print('-'*50)
```

Figura 18: Análise de conectividade.

```
Análise de conectividade - Marcadores
(array([0, 1, 2]), array([326527,      1,      1], dtype=int64))
```

Para cada pico, temos o resultado de sua análise de conectividade (Figura 18). Para o pico 0, que provavelmente é nosso fundo, foram observados 326527 pixels conectados. Para os picos 1 e 2, foram observados somente um pico conectado para cada.

Agora, de posse dos nossos marcadores, podemos executar a segmentação usando a função “watershed”. Como parâmetro dessa função, temos que indicar nossa imagem de distância, nossos marcadores e uma máscara, que é nossa imagem binária. Desta forma, nossas folhas que eram representadas por picos, serão representadas por valores, por isso colocamos o sinal de negativo antes da nossa imagem de distâncias (img_dist) no comando. Desta forma, iremos inverter nossa matriz de distância, sendo que todos os valores passarão a ser negativos.

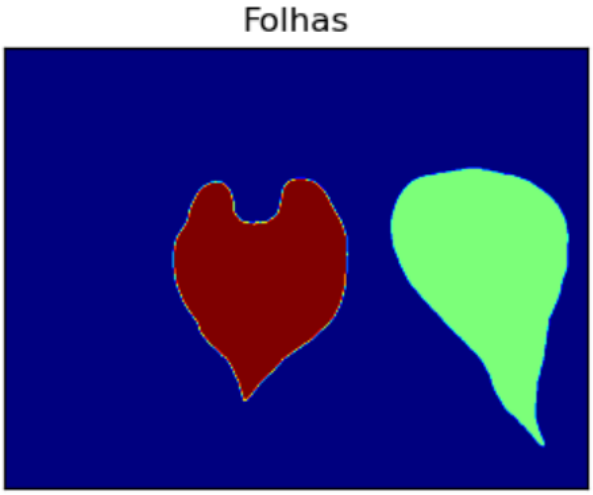
```
img_ws = watershed(-img_dist, marcadores, mask=mascara)

print('Imagem Segmentada - Watershed')
print(np.unique(img_ws,return_counts=True))
print("Número de Folhas: ", len(np.unique(img_ws)) - 1)
```

Figura 19: Matriz Watershed.

```
Imagem Segmentada - Watershed
(array([0, 1, 2]), array([258602,  36324,  31603], dtype=int64))
Número de Folhas:  2
```

O “img_ws” é uma imagem rotulada, onde que os pontos de uma inundação serão marcados com um valor. A imagem segmentada Watershed é composta pelos valores 0, 1 e 2, sendo que cada valor representa uma região da imagem. Observando o resultado acima (Figura 19) temos que a região 1 é marcada com 258602 pixels, a região 2 é marcada com 36324 pixels e a terceira região é marcada com 31603 pixels.

Figura 20: Imagem Segmentada - Watershed.



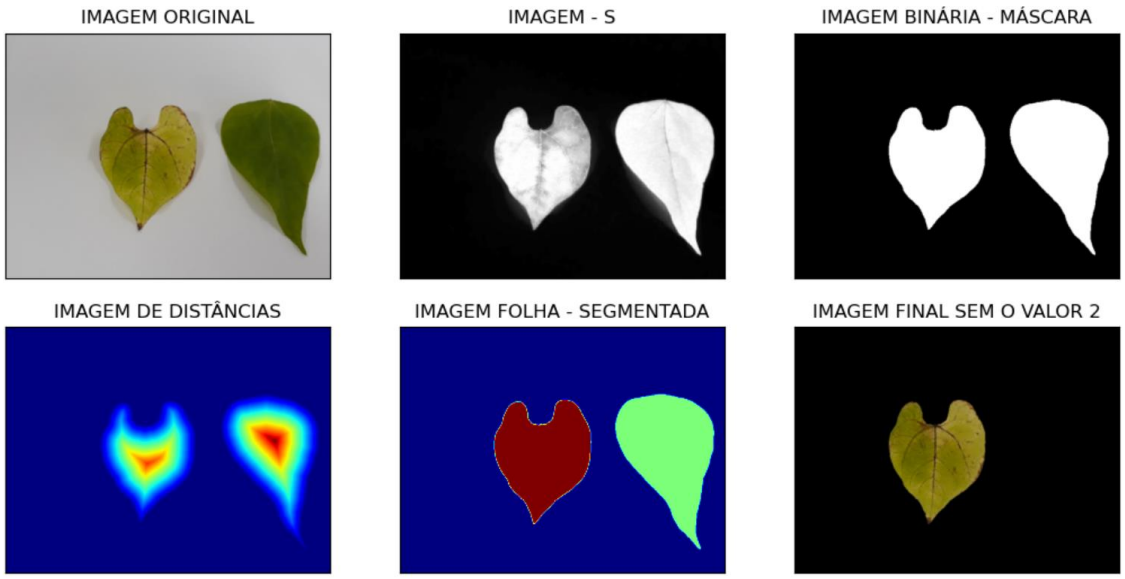
Além disso tudo que já foi feito, podemos também acessar cada folha individualmente. Para isso, criamos uma cópia (img_final) da nossa imagem original em RGB. Utilizando o comando abaixo, iremos acessar todos os valores diferentes de 2, ou seja, estaremos pegando tudo que não é a “folha 2”. No caso, estaremos acessando o nosso fundo da imagem e a folha 1.

```
img_final = np.copy(img_rgb)
img_final[img_ws != 2] = [0,0,0] # Acessando a folha 2
```

Ao final do exercício, criamos uma figura que contém a imagem original, imagem no sistema de cores HSV (Canal S), imagem binarizada (máscara), imagem de distâncias, imagem segmentada pela técnica de watershed e nossa imagem final sem o valor 2.

```
plt.figure('Watershed')
plt.subplot(2,3,1)
plt.imshow(img_rgb)
plt.xticks([])
plt.yticks([])
plt.title('IMAGEM ORIGINAL')

plt.subplot(2,3,2)
plt.imshow(s,cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title('IMAGEM - S')

plt.subplot(2,3,3)
plt.imshow(mascara,cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title('IMAGEM BINÁRIA - MÁSCARA')

plt.subplot(2,3,4)
plt.imshow(img_dist,cmap='jet')
plt.xticks([])
plt.yticks([])
plt.title('IMAGEM DE DISTÂNCIAS')

plt.subplot(2,3,5)
plt.imshow(img_ws,cmap='jet')
plt.xticks([])
plt.yticks([])
plt.title('IMAGEM FOLHA - SEGMENTADA')

plt.subplot(2,3,6)
plt.imshow(img_final)
plt.xticks([])
plt.yticks([])
plt.title('IMAGEM FINAL SEM O VALOR 2')

plt.show()
```

Figura 21: Segmentação Watershed.



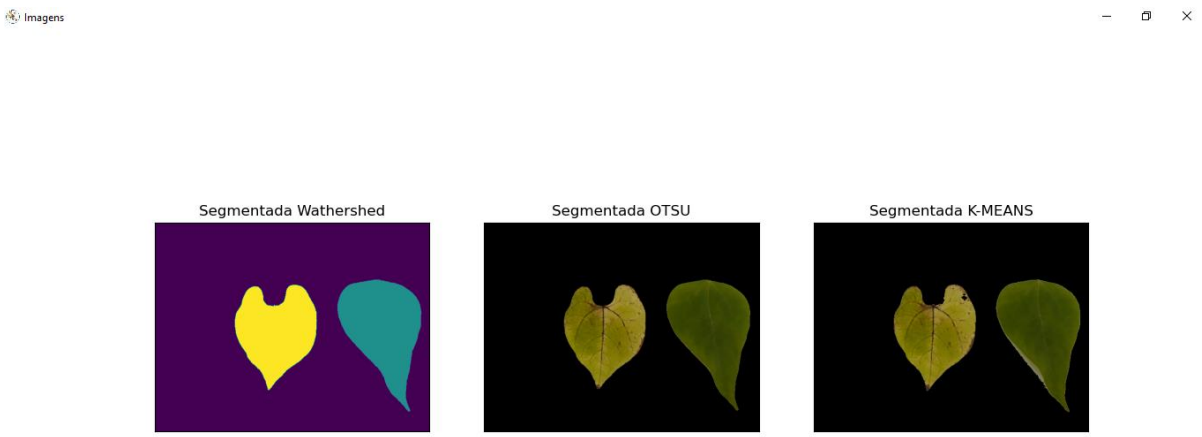
g) Compare os resultados das três formas de segmentação (limiarização, k-means e watershed) e identifique as potencialidades de cada delas.

Comandos para plotagem das três imagens obtidas anteriormente.

```
plt.figure('Imagens')
plt.subplot(1,3,1)
plt.imshow(img_ws)
plt.xticks([])
plt.yticks([])
plt.title('Segmentada Wathershed')

plt.subplot(1,3,2)
plt.imshow(img_segmentada)
plt.xticks([])
plt.yticks([])
plt.title('Segmentada OTSU')

plt.subplot(1,3,3)
plt.imshow(img_final_02)
plt.title('Segmentada K-MEANS')
plt.xticks([])
plt.yticks([])

plt.show()
```

Podemos observar, que os três tipos de segmentação na imagem utilizada, apresentaram um resultado semelhante. A imagem em questão, apresentou uma relativa facilidade de segmentação, não sendo observada uma grande diferença entre os três tipos. Sendo a segmentação de Wathershed e OTSU mais semelhantes.

Figura 23: Diferentes tipos de segmentação aplicadas as folhas de feijão.



A partir da nossa imagem, não conseguimos distinguir muita diferença com a aplicação das três segmentações. Para obter tal comparação realizamos a segmentação em duas imagens de folhas de soja com sintomas de ferrugem asiática (*Phakopsora pachyrhizi*). Estas folhas apresentam diferentes quantidades de dados, ou seja, furos devido o ataque de insetos, como mostra a figura 24.

Figura 24: Folhas de soja com lesões de ferrugem asiática e danos por insetos.

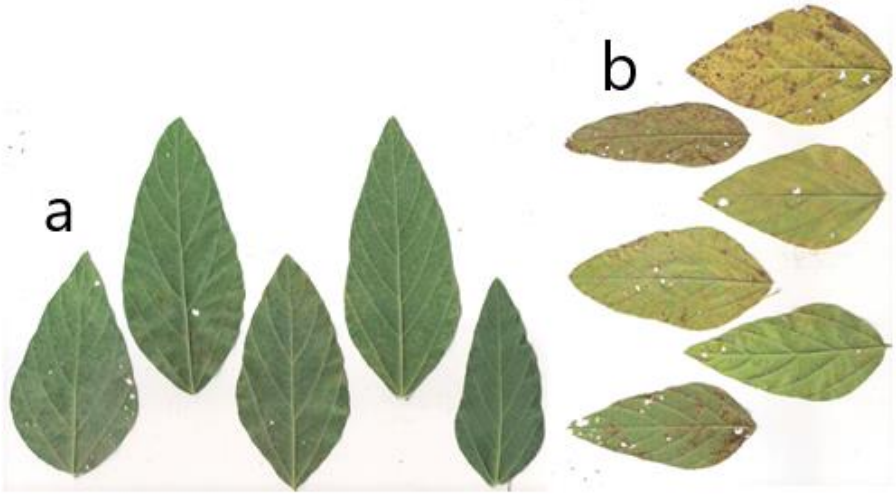
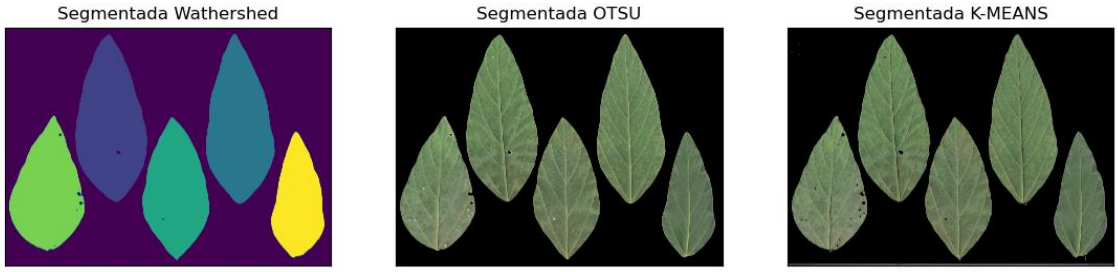
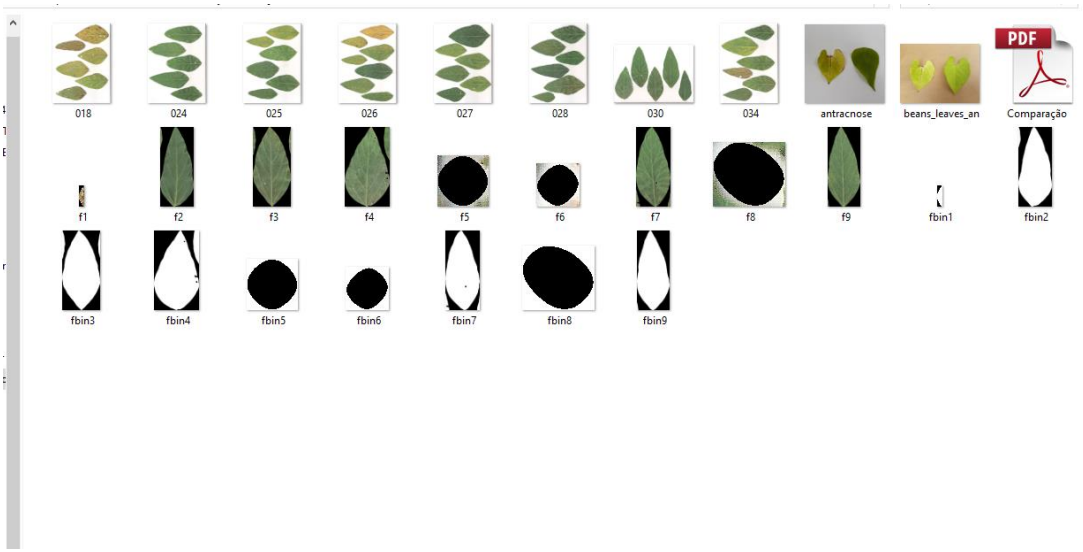


Figura 25: Diferentes segmentações aplicadas a folhas de soja (Imagem A).



Para a imagem (a) contendo menor quantidade furos a segmentação de OTSU, mesmo usando filtro de mediana, segmentou alguns buracos como folha, como mostra a figura 26.

Figura 26: Objetos encontrados na imagem A, a partir da segmentação de OTSU.



Já utilizando a segmentação de Watershed, foi possível separar eficientemente as cinco folhas usando como valor mínimo de distância 700 (*valormin_distance=700)*. A segmentação de K-means também foi eficiente.

Agora considerando a imagem (b), a qual apresenta maior quantidade de dados devido a pragas a segmentação foi eficiente apenas pelo método de K-means, visto que na segmentação de OTSU gerou grande quantidade de imagens. Pois cada buraco foi considerado como uma folha. A figura 27 mostra a grande quantidade de imagens geradas ao salvar as folhas.

Figura 27: Objetos segmentados da imagem B, a partir da segmentação de OTSU.

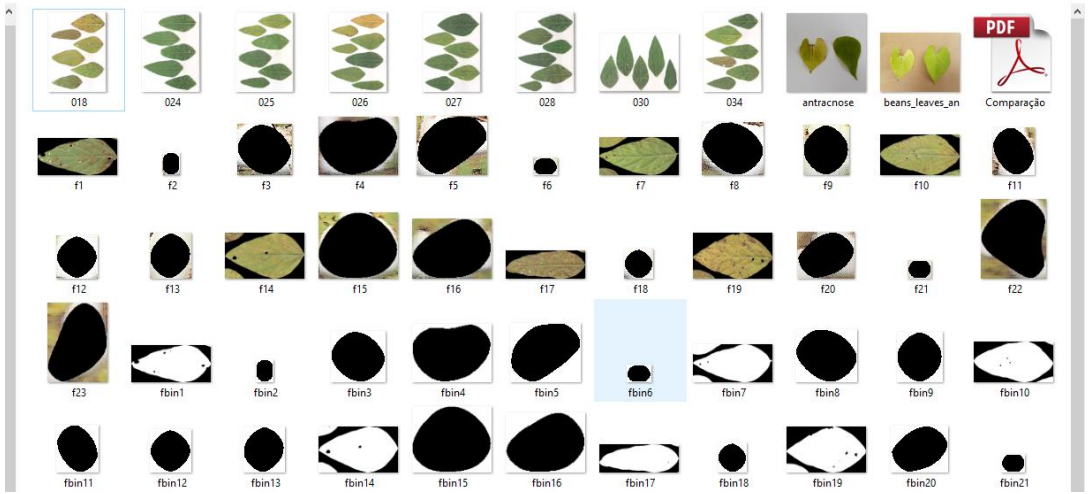
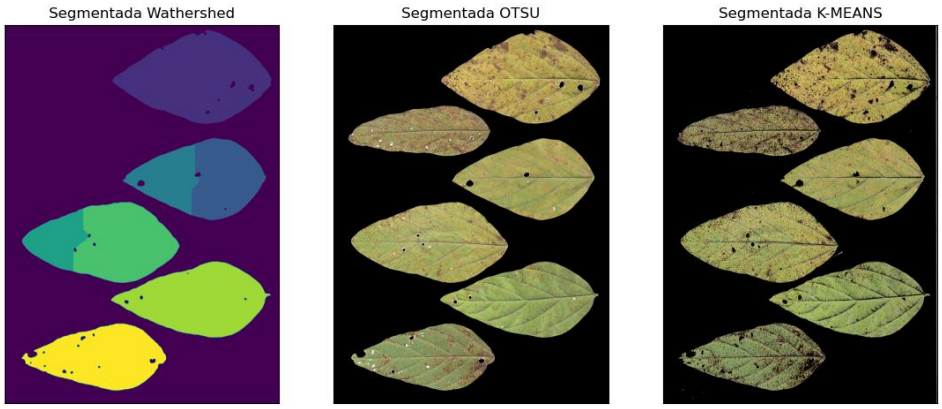


Figura 28: Diferentes tipos de segmentação aplicadas a folha de soja (Imagem B).



Com relação ao Watershed observou-se que os furos prejudicaram a segmentação, pois estes foram confundidos com as bordas das folhas. Assim não houve um valor de distância mínima para separar as folhas sem que perdesse o objeto de interesse (folha).