In [61]:
```
import os
import cv2
import numpy as np
from matplotlib import pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
```

The `%matplotlib inline` magic command in the provided code uses the default Matplotlib coordinate system, which assumes the origin at the bottom-left corner, causing potential differences in image orientation compared to OpenCV's coordinate system. To align the coordinate systems and prevent y-axis inversion, the `plt.imshow` function should be used with the `origin='upper'` parameter when displaying images in a Jupyter Notebook or Jupyter Lab environment.

In [62]:
```
ds_path = "leaf_image.jpg"
```

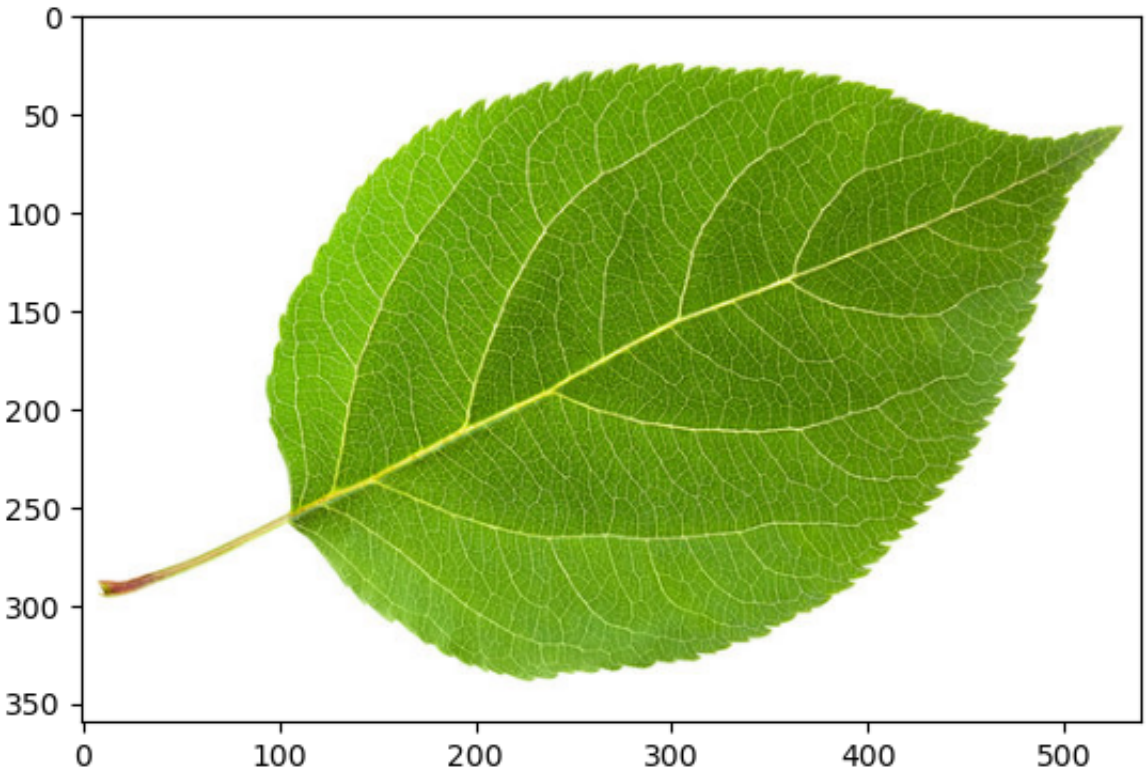
In [63]:
```
test_img_path = ds_path
```

In [64]:
```
test_img_path
```

Out[64]:
```
'leaf_image.jpg'
```

In [65]:
```
main_img = cv2.imread(test_img_path)
img = cv2.cvtColor(main_img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
```

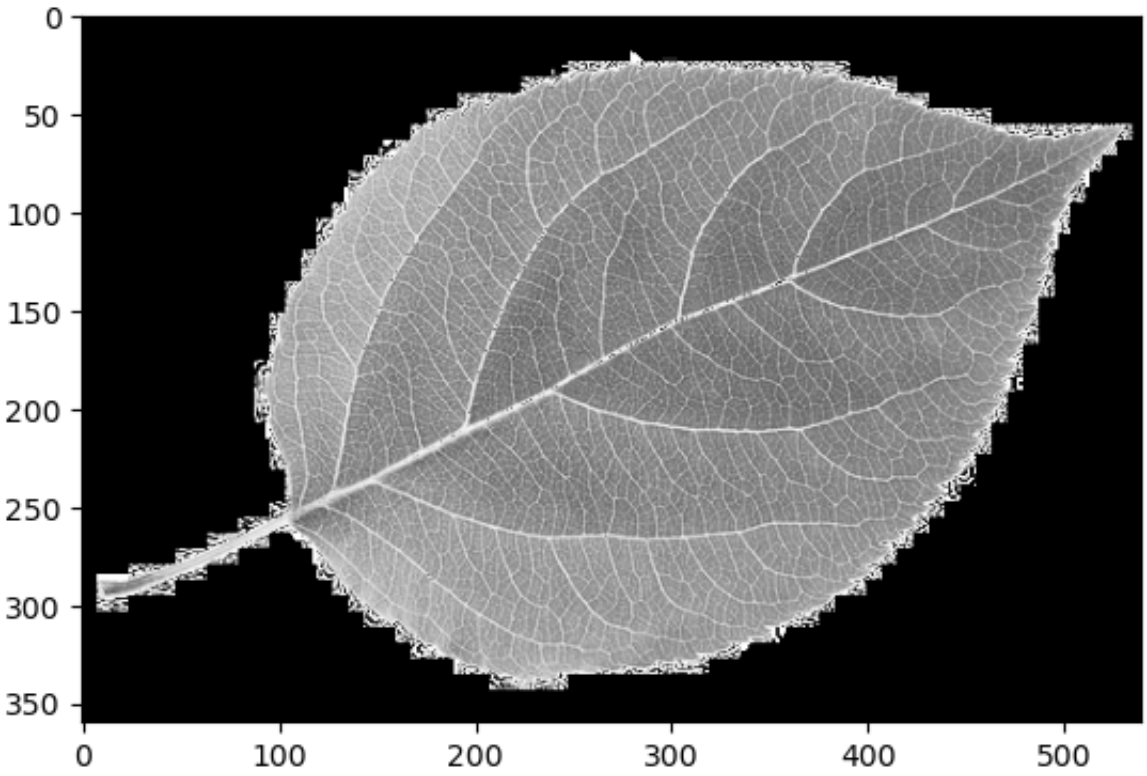
Out[65]: <matplotlib.image.AxesImage at 0x14bfc2b10>



OpenCV convention: In OpenCV, the origin (0,0) is at the top-left corner of the image. The y-axis increases downward, and the x-axis increases to the right.

In [111]:
```
gs = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
plt.imshow(gs,cmap='Greys_r')
```

Out[111]:
```
<matplotlib.image.AxesImage at 0x14c6ed050>
```



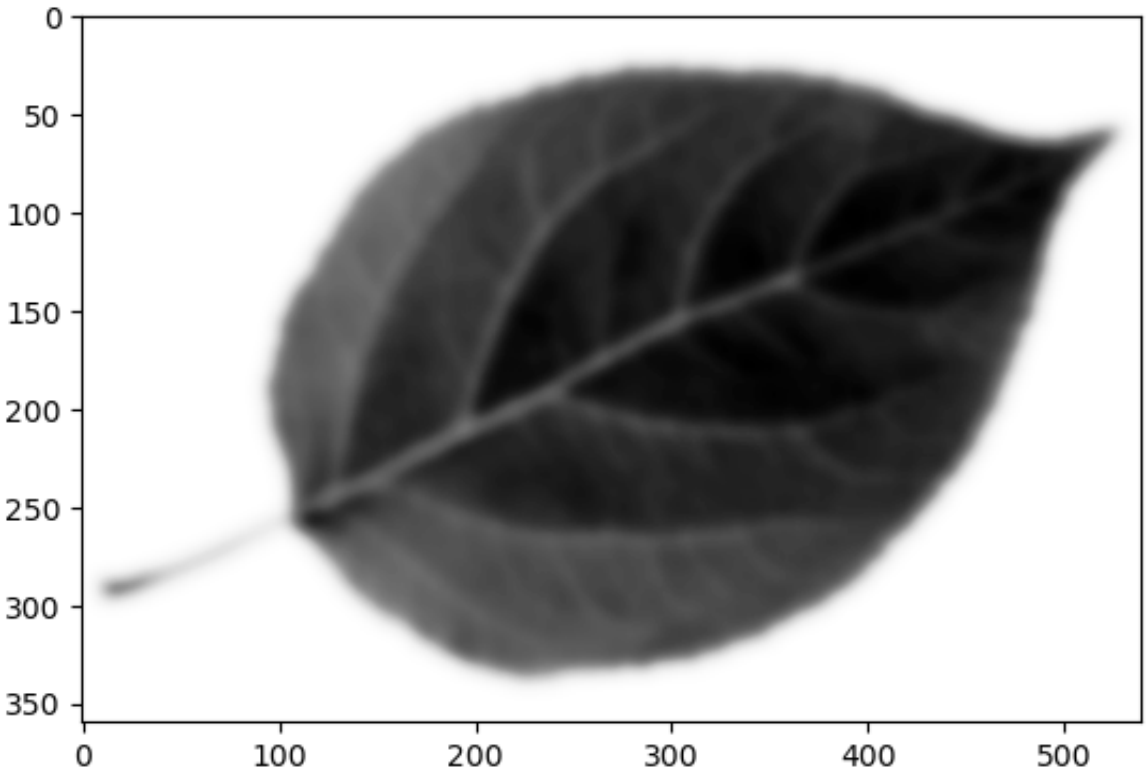
The inversion of the image axis in the provided code may arise from the default Matplotlib coordinate system, which expects the origin at the bottom-left corner, while OpenCV uses the top-left corner as the origin. To correct this, the `plt.imshow` function should be used with the `origin='upper'` parameter when displaying a grayscale image, ensuring proper alignment of the coordinate systems.

In [67]:
```
gs.shape
```

Out[67]:
```
(360, 540)
```

In [68]:
```
blur = cv2.GaussianBlur(gs, (25,25),0)
plt.imshow(blur,cmap='Greys_r')
```

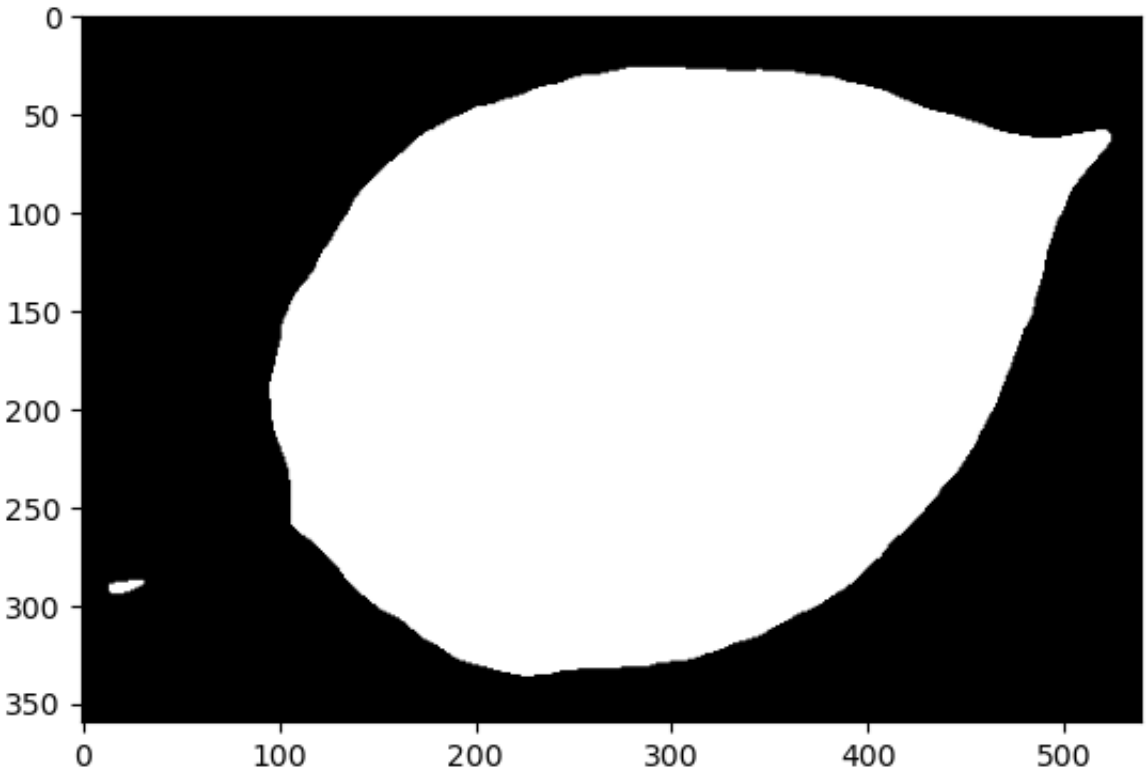
Out[68]: <matplotlib.image.AxesImage at 0x14d0b2b10>



The inversion of the image axis in the provided code may arise from the default Matplotlib coordinate system, which expects the origin at the bottom-left corner, while OpenCV uses the top-left corner as the origin. To correct this, the `plt.imshow` function should be used with the `origin='upper'` parameter when displaying a grayscale image, ensuring proper alignment of the coordinate systems.

In [69]:
```
ret_otsu,im_bw_otsu = cv2.threshold(blur,0,255,cv2.THRESH_BINARY_INV+c
plt.imshow(im_bw_otsu,cmap='Greys_r')
```

Out[69]:
```
<matplotlib.image.AxesImage at 0x14d157f10>
```



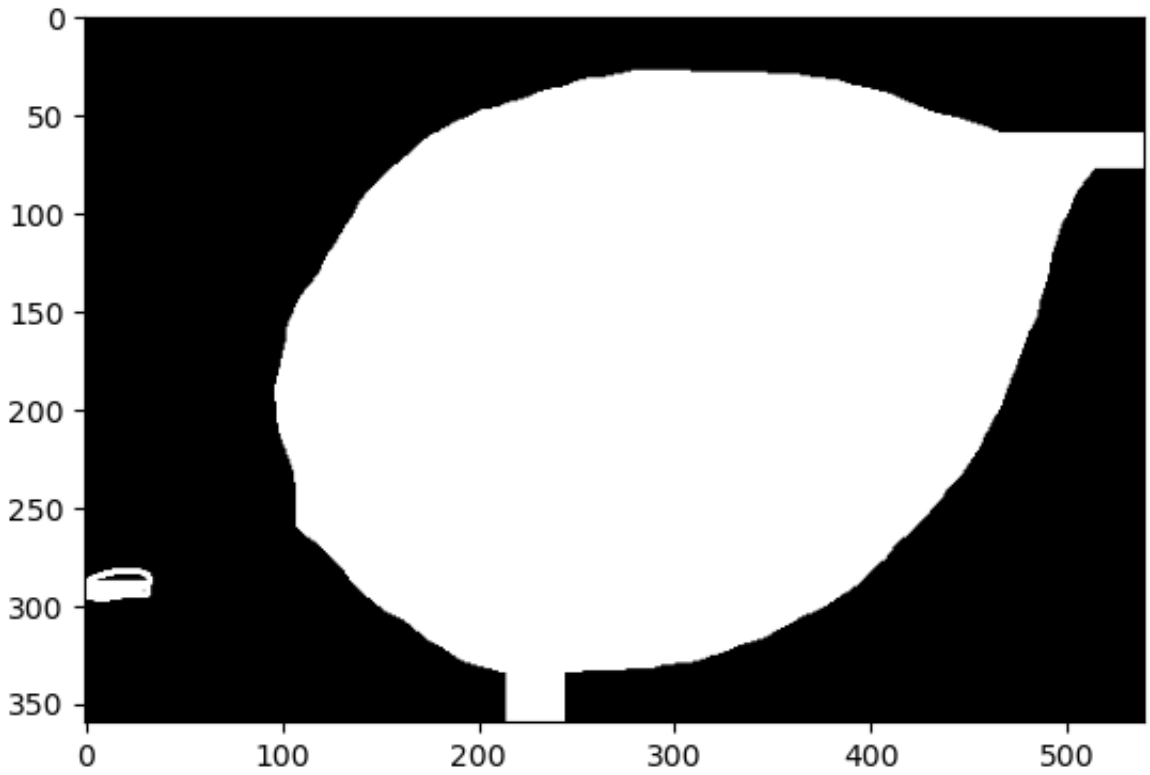
The inversion is a result of the way images are typically displayed in computer graphics, where the origin is at the top-left corner. When using cv2.THRESH_BINARY_INV, black and white regions are inverted, making the background white and the foreground black. This inversion is correctly displayed by Matplotlib without the need for the origin='upper' parameter.

In [70]:
```
# Creating a 50x50 matrix filled with ones, which serves as the kernel

kernel = np.ones((50,50),np.uint8)

# Applying a morphological closing operation to the binary image 'im_b


closing = cv2.morphologyEx(im_bw_otsu, cv2.MORPH_CLOSE, kernel)
```

In summary, the code is creating a 50x50 square structuring element (kernel) filled with ones and then applying a morphological closing operation to the binary image im_bw_otsu. This operation helps in closing small gaps or holes in the binary image, making the foreground regions more connected and compact. The size of the kernel (50x50 in this case) can be adjusted based on the specific characteristics of the image and the desired effect of the morphological operation.

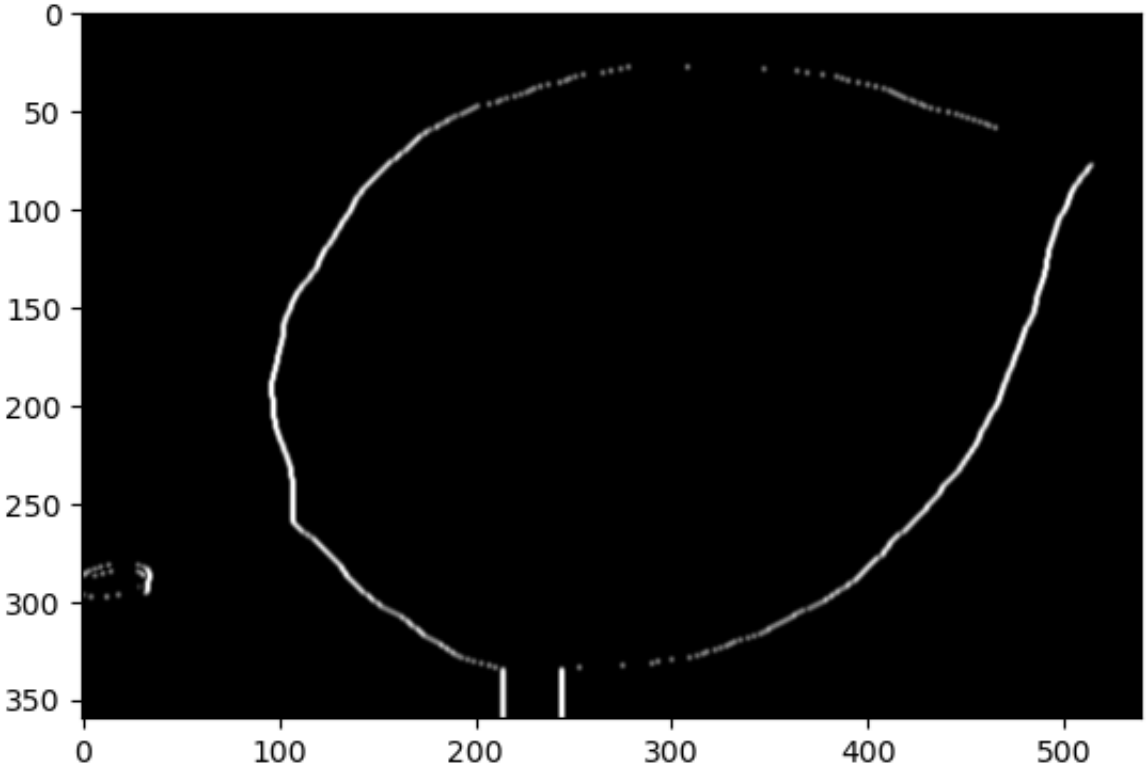
In [112]:
```
#Displaying the result of the morphological closing operation using Ma

plt.imshow(closing,cmap='Greys_r')
```

Out[112]:
```
<matplotlib.image.AxesImage at 0x14d786110>
```



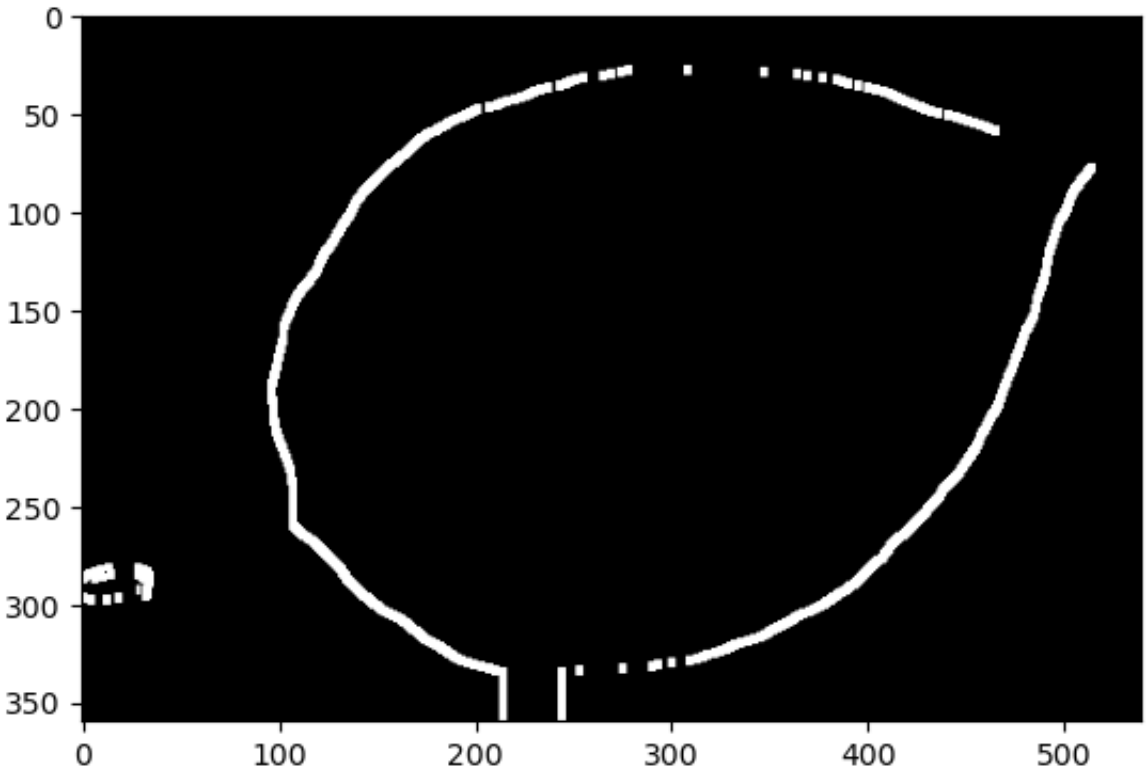
In [115]:
```
# Applying Sobel edge detection along the x-axis to the binary image '
sobelx64f = cv2.Sobel(closing,cv2.CV_64F,1,0,ksize=5)
# Taking the absolute value of the Sobel result to get magnitude
abs_sobel64f = np.absolute(sobelx64f)
# Converting the magnitude values to 8-bit unsigned integers
sobel_8u = np.uint8(abs_sobel64f)
# Displaying the Sobel edge detection result using Matplotlib
plt.imshow(abs_sobel64f,cmap='Greys_r')
```

Out[115]: <matplotlib.image.AxesImage at 0x14d8df1d0>



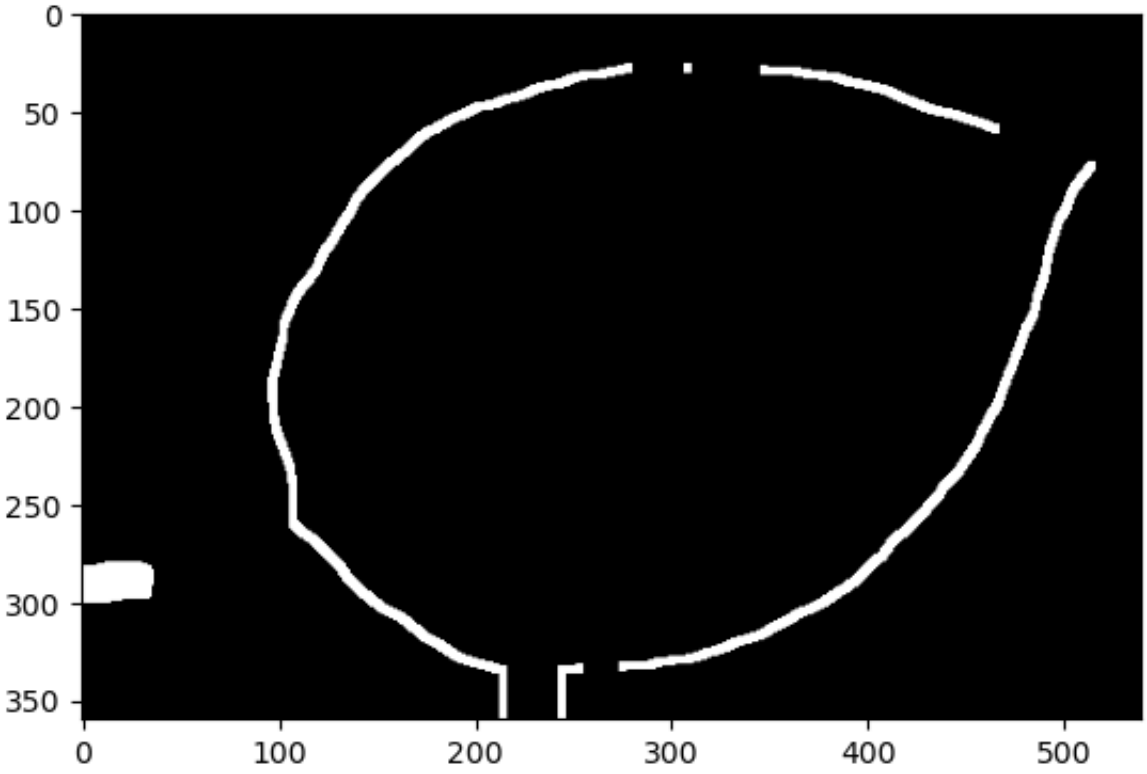
In [116]:
```
# Thresholding the Sobel edge detection result to obtain a binary imag
ret_sobel,im_bw_sobel = cv2.threshold(sobel_8u,1,255,cv2.THRESH_BINARY
# Displaying the binary image obtained after thresholding
plt.imshow(im_bw_sobel,cmap='Greys_r')
```

Out[116]: <matplotlib.image.AxesImage at 0x14d92df50>



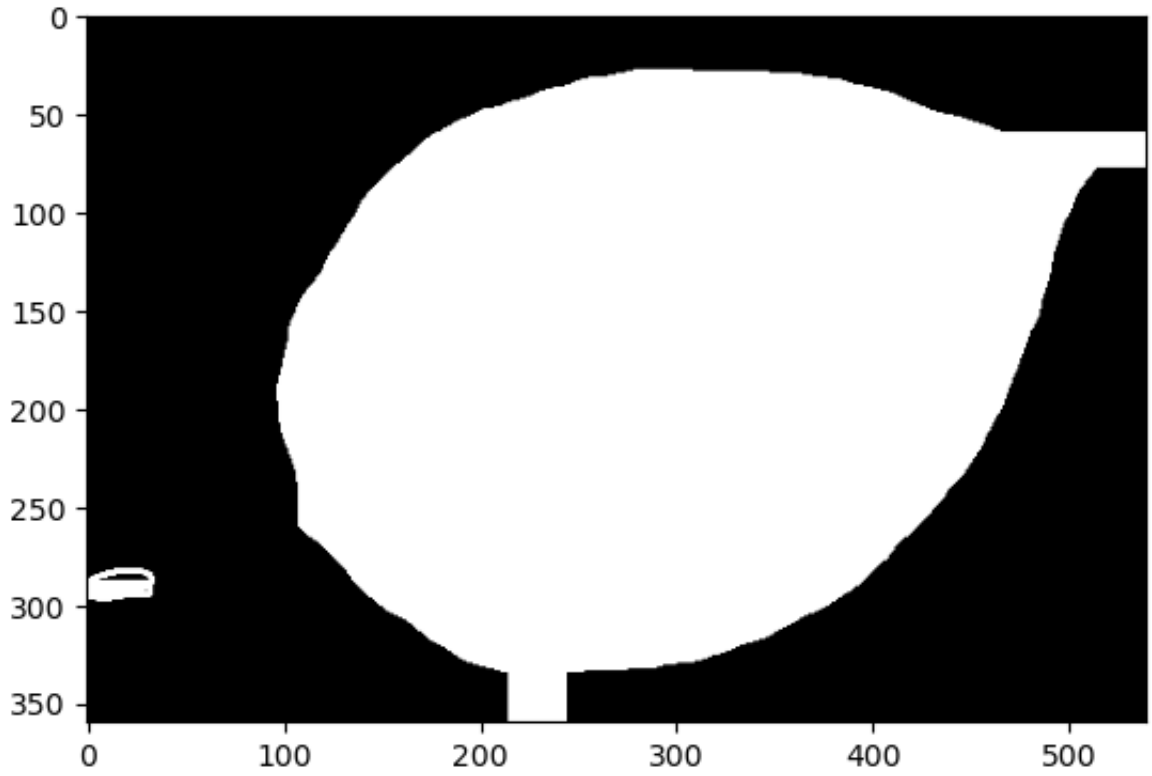
In [118]:
```
# Creating a 15x15 square structuring element (kernel) filled with one
kernel_edge = np.ones((15, 15), np.uint8)

# Applying a morphological closing operation to the binary image 'im_b
closing_edge = cv2.morphologyEx(im_bw_sobel, cv2.MORPH_CLOSE, kernel_e

# Displaying the result of the morphological closing operation on the
plt.imshow(closing_edge, cmap='Greys_r')
```

Out[118]: <matplotlib.image.AxesImage at 0x14da40050>



In [119]:
```
# Displaying the result of the previous morphological closing operatic
plt.imshow(closing, cmap="Greys_r")
```

Out[119]: <matplotlib.image.AxesImage at 0x14daab550>



In [120]:
```
# Finding contours in the binary image obtained after the morphologica
contours, hierarchy = cv2.findContours(closing, cv2.RETR_TREE, cv2.CHA
```

In [121]:
```
len(contours)
```

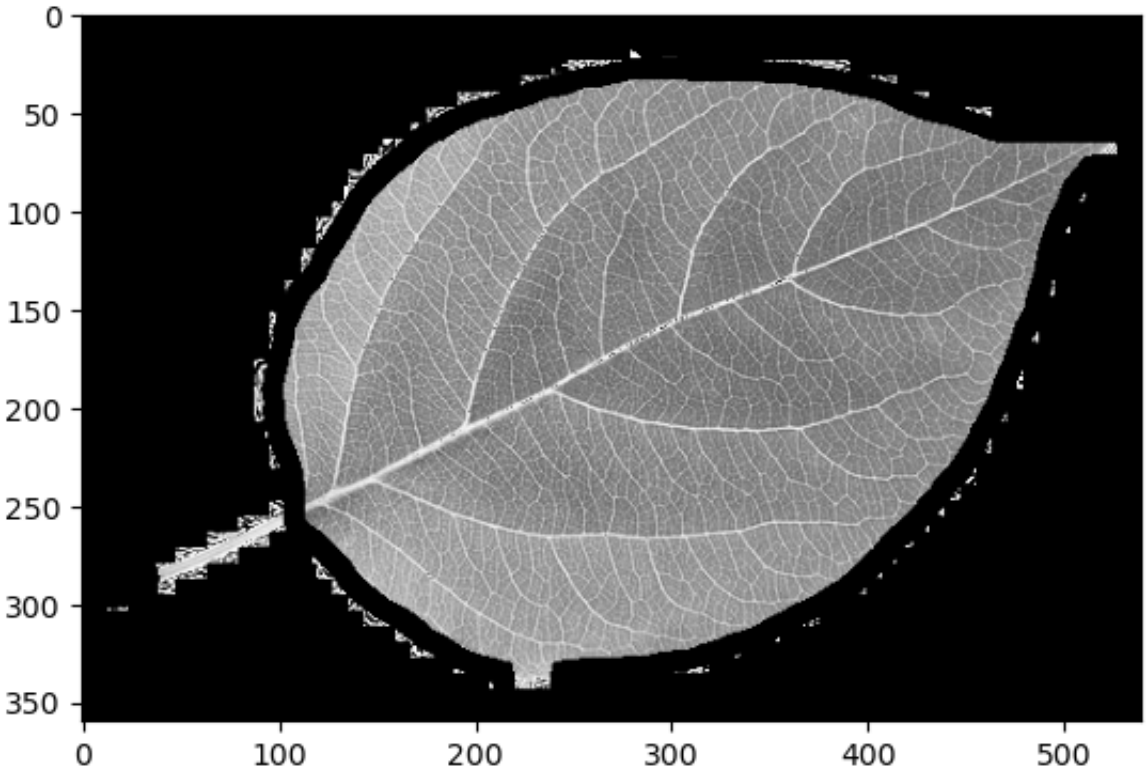
Out[121]: 4

In [125]:
```
# Selecting the first contour from the list of contours
cnt = contours[0]

# Printing the number of points in the selected contour
len(cnt)
```

Out[125]: 29

In [124]:
```
# Drawing contours on the grayscale image
plottedContour = cv2.drawContours(gs, contours, -1, (0, 255, 0), 10)

# Displaying the image with drawn contours
plt.imshow(plottedContour, cmap="Greys_r")
```

Out[124]: <matplotlib.image.AxesImage at 0x14db97e50>



In [126]:
```
# Calculating the moments of the selected contour
M = cv2.moments(cnt)
M
```

Out[126]:
```
{'m00': 450.5,
 'm10': 7627.833333333333,
 'm01': 130612.0,
 'm20': 167350.0833333333,
 'm11': 2208125.625,
 'm02': 37875363.58333333,
 'm30': 4125965.95,
 'm21': 48418798.7,
 'm12': 639337963.2333333,
 'm03': 10985400454.1,
 'mu20': 38196.1624429646,
 'mu11': -3387.288059563376,
 'mu02': 7451.166019231081,
 'mu30': -1062.8317132764496,
 'mu21': 14239.883590120822,
 'mu12': -815.7167177142255,
 'mu03': -1125.7857723236084,
 'nu20': 0.18820455970349684,
 'nu11': -0.016690238418348224,
 'nu02': 0.036714249030149415,
 'nu30': -0.00024673315737902,
 'nu21': 0.0033057457686025337,
 'nu12': -0.00018936616095885297,
 'nu03': -0.00026134775117077537}
```

In [131]:
```
# Calculating the area of the selected contour
area = cv2.contourArea(cnt)
area
```

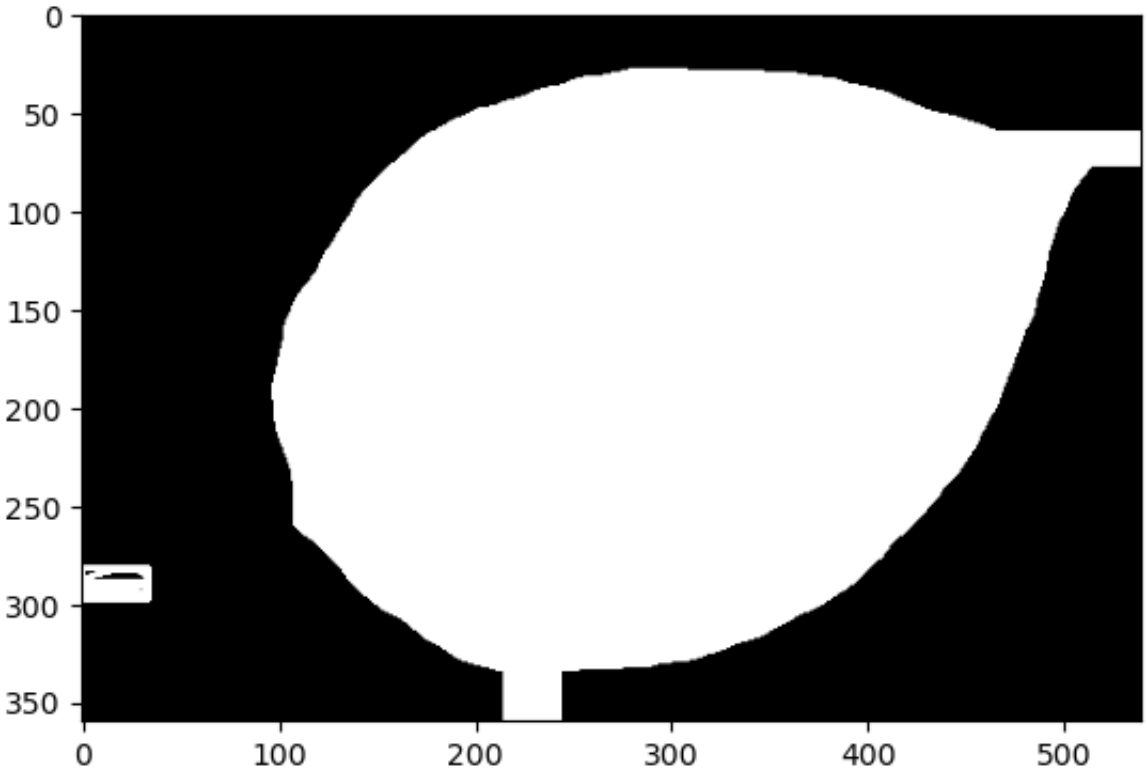
Out[131]:
```
450.5
```

In [132]:
```
# Calculating the perimeter (arc length) of the selected contour
perimeter = cv2.arcLength(cnt, True)
perimeter
```

Out[132]:
```
91.21320307254791
```

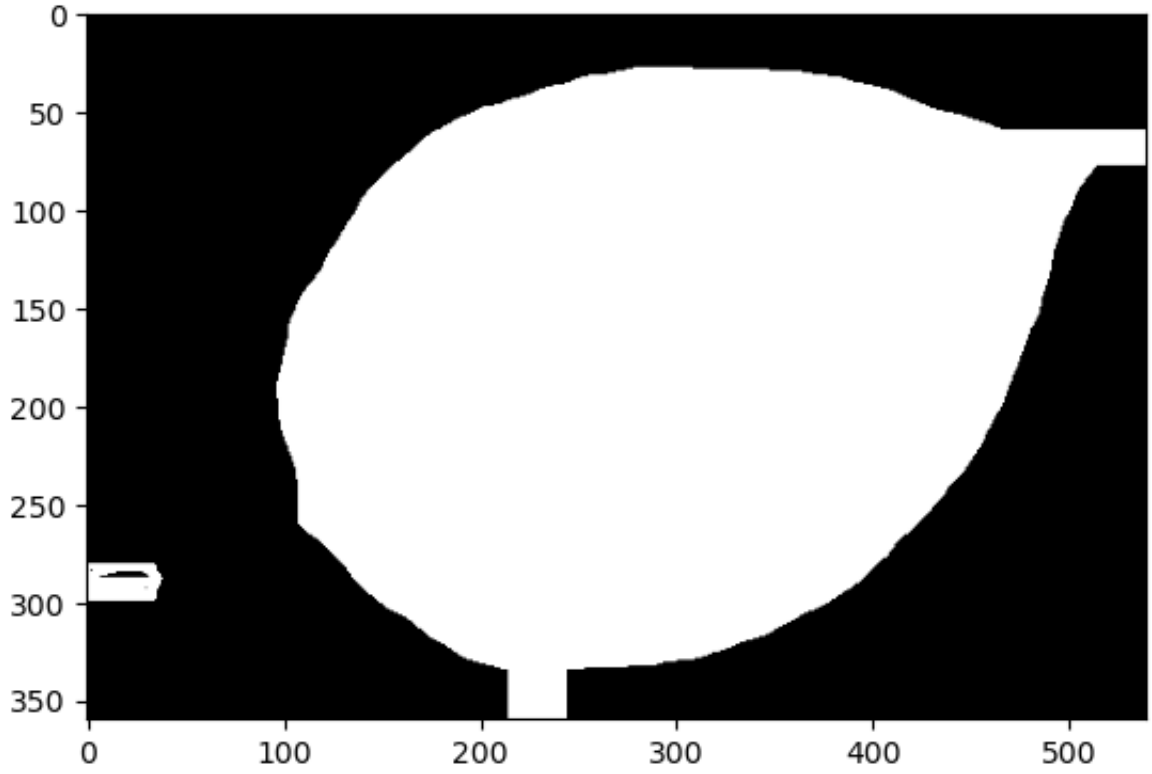
In [133]:
```
# Finding the minimum area bounding rectangle (rotated rectangle) for
rect = cv2.minAreaRect(cnt)

# Extracting the four corner points of the rectangle
box = cv2.boxPoints(rect)
box = np.intp(box)

# Drawing the bounding box on the binary image after closing operation
contours_im = cv2.drawContours(closing, [box], 0, (255, 255, 255), 2)

# Displaying the image with the drawn bounding box
plt.imshow(contours_im, cmap="Greys_r")
```

Out[133]: <matplotlib.image.AxesImage at 0x14dc0d750>



In [134]:
```
# Fitting an ellipse to the selected contour
ellipse = cv2.fitEllipse(cnt)

# Drawing the fitted ellipse on the binary image after closing operati
im = cv2.ellipse(closing, ellipse, (255, 255, 255), 2)

# Displaying the binary image with the drawn ellipse
plt.imshow(closing, cmap="Greys_r")
```

Out[134]: <matplotlib.image.AxesImage at 0x14dc8cd90>



In [135]:
```
# Obtaining the bounding rectangle (upright rectangle) for the selecte
x, y, w, h = cv2.boundingRect(cnt)

# Calculating the aspect ratio of the bounding rectangle
aspect_ratio = float(w) / h

# Displaying the calculated aspect ratio
aspect_ratio
```

Out[135]: 2.0588235294117645

In [136]:
```
# Calculating rectangularity, the ratio of the area of the bounding re
rectangularity = w * h / area

# Displaying the calculated rectangularity
rectangularity
```

Out[136]:
```
1.320754716981132
```

In [87]:
```
# Calculating circularity, a measure of how close the contour is to a
circularity = ((perimeter)**2) / area

# Displaying the calculated circularity
circularity
```

Out[87]:
```
31.68772067347585
```

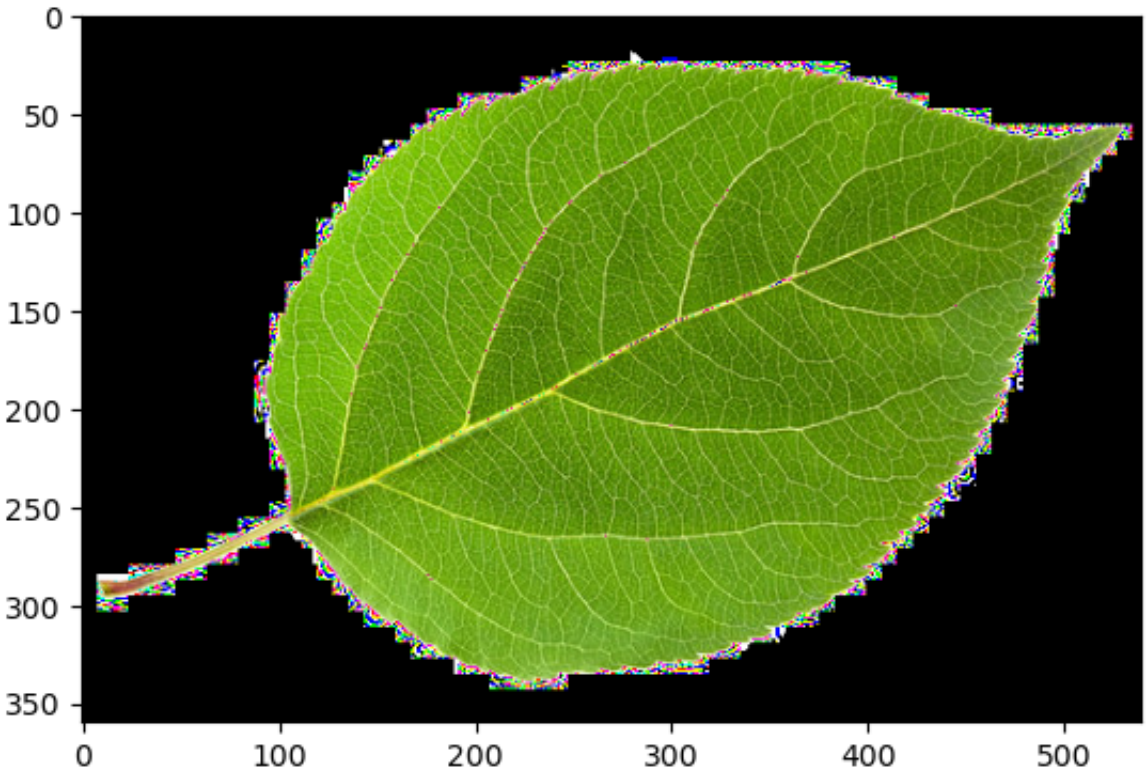
In [88]:
```
# Calculating the equivalent diameter, the diameter of a circle with t
equi_diameter = np.sqrt(4 * area / np.pi)

# Displaying the calculated equivalent diameter
equi_diameter
```

Out[88]:
```
14.647213499088313
```

In [140]:
```
# Fitting an ellipse to the selected contour and extracting ellipse pa

(x,y),(MA,ma),angle = cv2.fitEllipse(cnt)
```

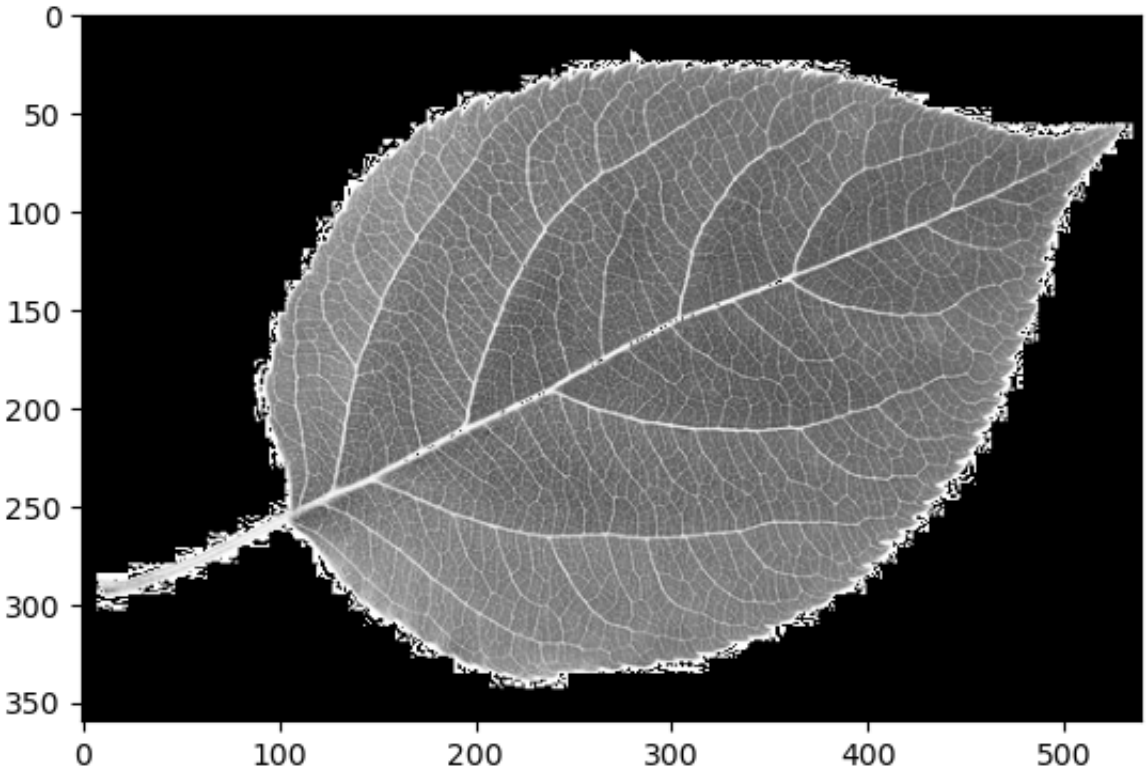
In [142]:
```
# Displaying the original image

plt.imshow(img,cmap="Greys_r")
```

Out[142]: <matplotlib.image.AxesImage at 0x14dce2b10>



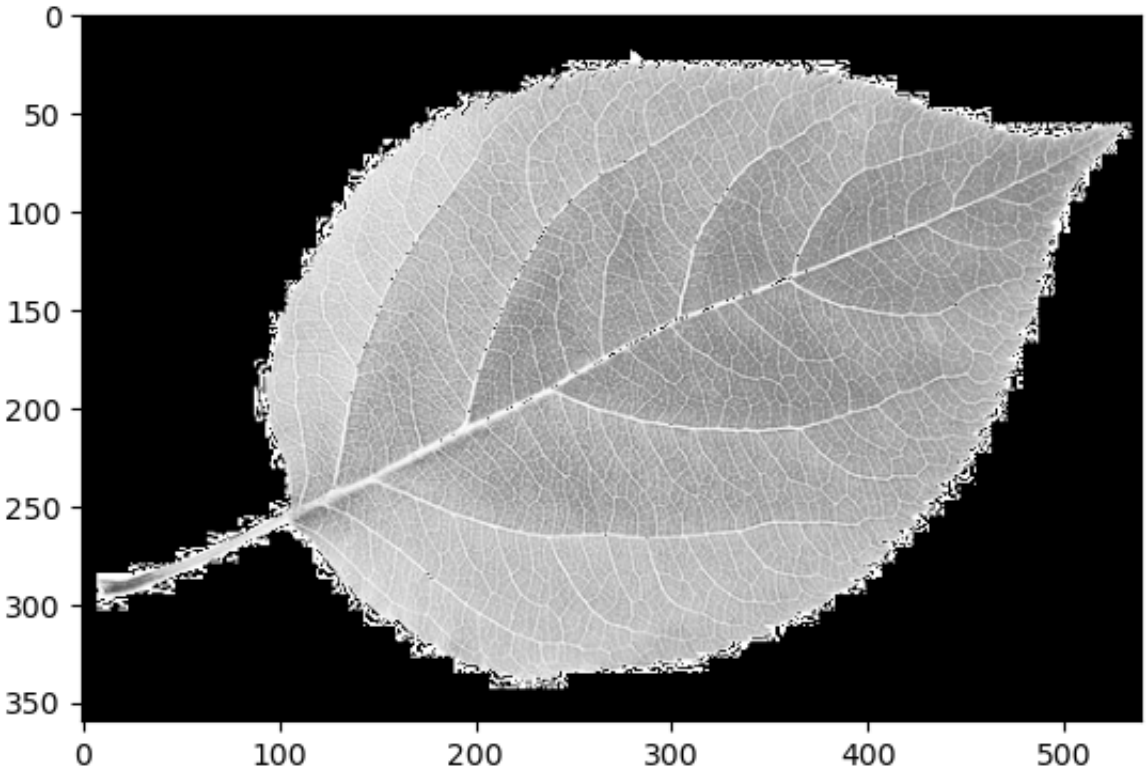
In [147]:
```
# Extracting the red channel from the original image
red_channel = img[:, :, 0]

# Displaying the red channel as a grayscale image
plt.imshow(red_channel, cmap="Greys_r")
```

Out[147]: <matplotlib.image.AxesImage at 0x169718910>



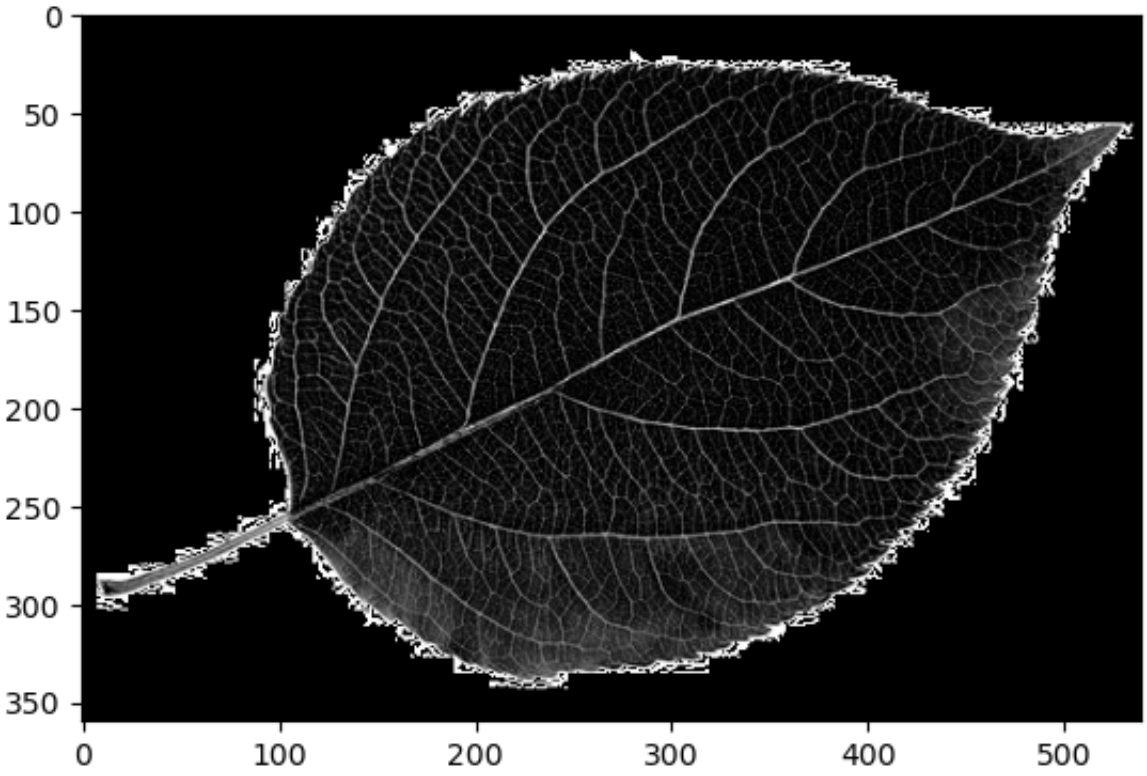
In [148]:
```
# Extracting the green channel from the original image
green_channel = img[:, :, 1]

# Displaying the green channel as a grayscale image
plt.imshow(green_channel, cmap="Greys_r")
```

Out[148]: <matplotlib.image.AxesImage at 0x1696cab10>



In [149]:
```
# Extracting the blue channel from the original image
blue_channel = img[:, :, 2]

# Displaying the blue channel as a grayscale image
plt.imshow(blue_channel, cmap="Greys_r")
```

Out[149]: <matplotlib.image.AxesImage at 0x1697efa50>



In [150]:
```
# Calculating the mean intensity value of the pixel values in the blue
np.mean(blue_channel)

np.mean(blue_channel)
```

Out[150]: 17.947469135802468

In [154]:
```
# Calculating the mean intensity value of the pixel values in the gree
np.mean(green_channel)
```

Out[154]: 83.69838477366255

In [156]:
```
# Calculating the mean intensity value of the pixel values in the red
np.mean(red_channel)
```

Out[156]: 59.917705761316874

In [152]:
```
# Setting white pixels to black (0) in the blue channel
blue_channel[blue_channel == 255] = 0

# Setting white pixels to black (0) in the green channel
green_channel[green_channel == 255] = 0

# Setting white pixels to black (0) in the red channel
red_channel[red_channel == 255] = 0
```

In [157]:
```
# Calculating the mean intensity value of the pixel values in the red
red_mean = np.mean(red_channel)
red_mean
```

Out[157]: 59.917705761316874

In [158]:
```
# Calculating the mean intensity value of the pixel values in the gree

green_mean = np.mean(green_channel)
green_mean
```

Out[158]: 83.69838477366255

In [159]:
```
# Calculating the mean intensity value of the pixel values in the blue

blue_mean = np.mean(blue_channel)
blue_mean
```

Out[159]: 17.947469135802468

In [163]:
```

# Calculating the standard deviation (variability) of the pixel values
red_var = np.std(red_channel)
red_var
```

Out[163]: 64.41775179387123

In [164]:
```
# Calculating the standard deviation (variability) of the pixel values

blue_var=np.std(blue_channel)
blue_var
```

Out[164]: 41.31255498460539

In [165]:
```
# Calculating the standard deviation (variability) of the pixel values

green_var=np.std(green_channel)
green_var
```

Out[165]:
```
83.75357123682492
```

In [166]:
```
red_channel
```

Out[166]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

In [103]:
```
blue_channel
```

Out[103]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

In [104]:
```
green_channel
```

Out[104]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

In [169]:
```
# Importing the mahotas library with the alias 'mt'
import mahotas as mt

# Calculating Haralick textures features for the grayscale image
textures = mt.features.haralick(gs)

# Calculating the mean Haralick textures features along axis 0
ht_mean = textures.mean(axis=0)
ht_mean
```

Out[169]:
```
array([ 2.75682680e-01,  7.64649573e+02,  9.12246218e-01,  4.35666372
e+03,
        5.51148353e-01,  1.18270043e+02,  1.66620053e+04,  4.57863303
e+00,
        7.21181551e+00,  1.12287666e-03,  3.78798779e+00, -2.62567124
e-01,
        9.41635100e-01])
```

In [170]:
```
# Printing specific Haralick texture features from the mean values
print(ht_mean[1])  # Contrast
print(ht_mean[2])  # Correlation
print(ht_mean[4])  # Inverse Difference Moments
print(ht_mean[8])  # Entropy
```

```
764.6495731048951
0.9122462180243582
0.5511483532281869
7.2118155055769195
```

1. **Entropy:**

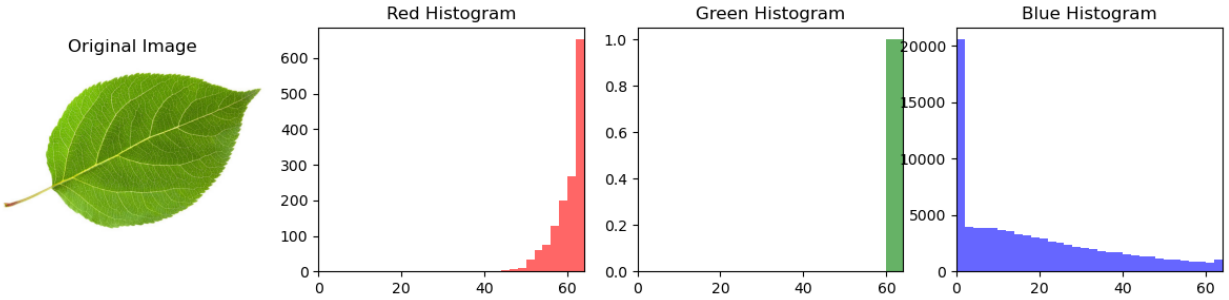
    - **Definition:** Entropy measures the randomness or disorder in an image.
    - **In Context:** In the context of texture analysis, high entropy indicates a more complex or random texture, while low entropy suggests a more ordered or uniform texture.
2. **Inverse Difference Moments (IDM):**

    - **Definition:** IDM is a measure of local homogeneity or smoothness in an image.
    - **In Context:** High IDM values suggest a more homogeneous or smooth texture, while lower values indicate more variation or heterogeneity.
3. **Correlation:**

    - **Definition:** Correlation measures the linear dependence between the intensity values of a pixel and its neighbors at a given offset.
    - **In Context:** High correlation values suggest a more structured or predictable texture, while low values indicate a less structured or more complex texture.
4. **Contrast:**

    - **Definition:** Contrast measures the local variations in pixel intensities.
    - **In Context:** High contrast values indicate regions with sharp intensity transitions, contributing to a more visually distinct or textured appearance. Lower contrast values suggest smoother transitions between intensity levels.

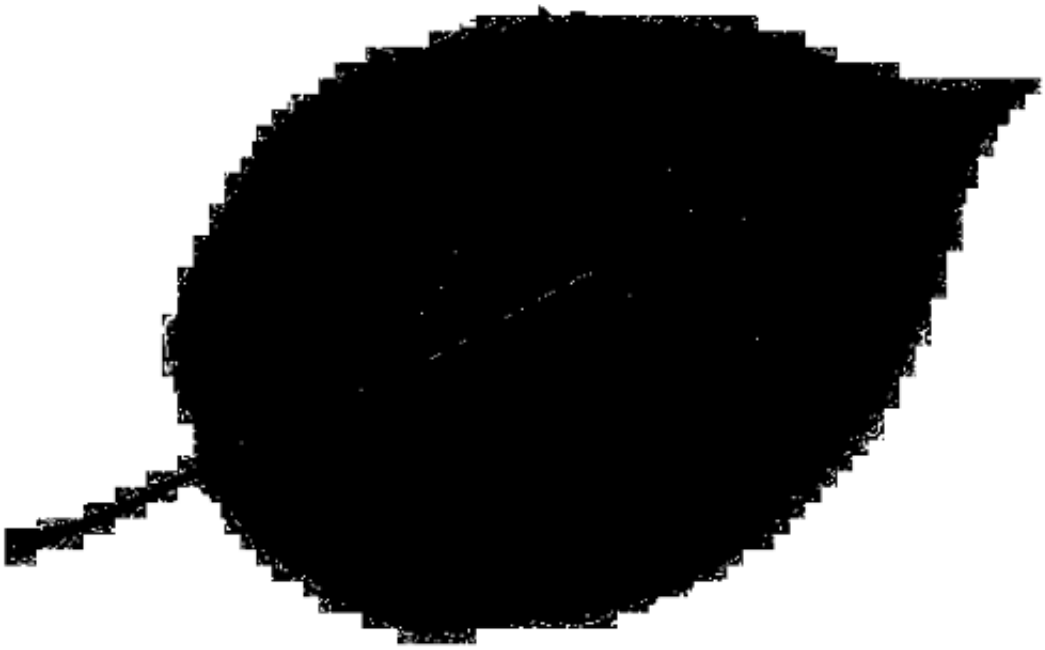
In [216]:
```
# Read a color image
image_color = cv2.imread("leaf_image.jpg")

# Convert the image to RGB (OpenCV uses BGR by default)
image_rgb = cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB)

# Plot histograms for each color channel
fig, axes = plt.subplots(1, 4, figsize=(15, 3))

# Plot the original image
axes[0].imshow(image_rgb)
axes[0].set_title('Original Image')
axes[0].axis('off')

# Plot histograms for each color channel
for i, color in enumerate(['Red', 'Green', 'Blue']):
    axes[i + 1].hist(image_rgb[:,:,i].ravel(), bins=32, range=[0,64],
    axes[i + 1].set_title(f'{color} Histogram')
    axes[i + 1].set_xlim([0, 64])

plt.show()
```



In [191]:
```
# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply Otsu's thresholding to the grayscale image
ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.
print("Threshold limit: " + str(ret))

# Set axis properties to not show ticks and labels
plt.axis('off')

# Display the thresholded image using a grayscale colormap
plt.imshow(thresh, cmap='gray')
```

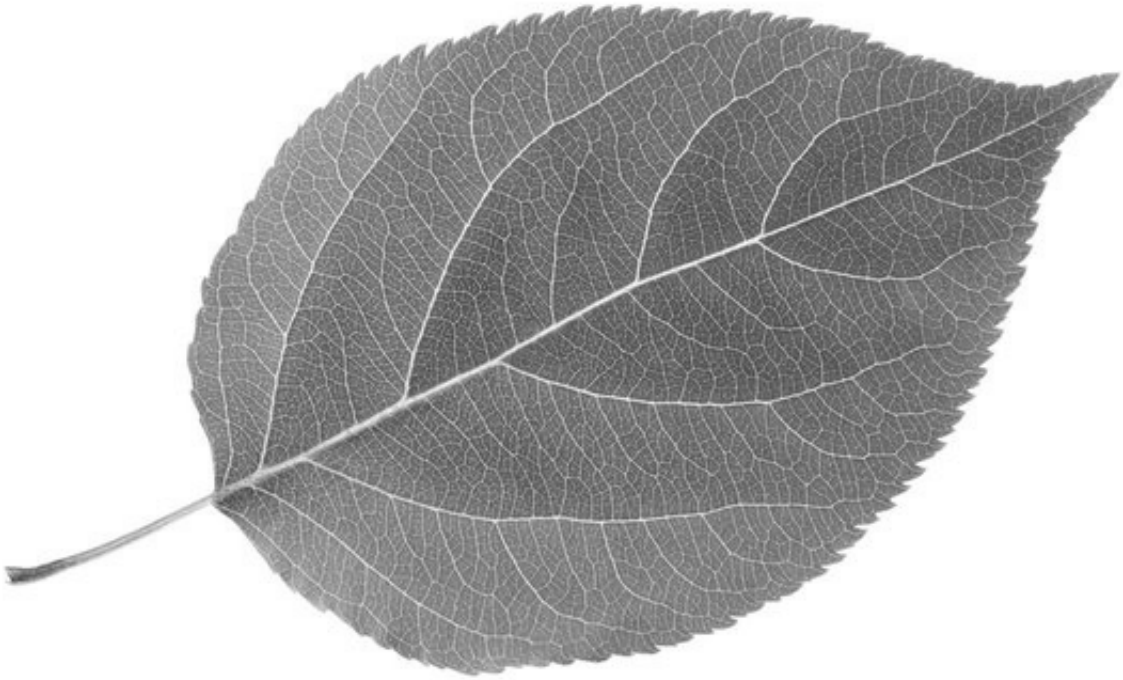
```
Threshold limit: 57.0
```

Out[191]:
```
<matplotlib.image.AxesImage at 0x16ae22b10>
```



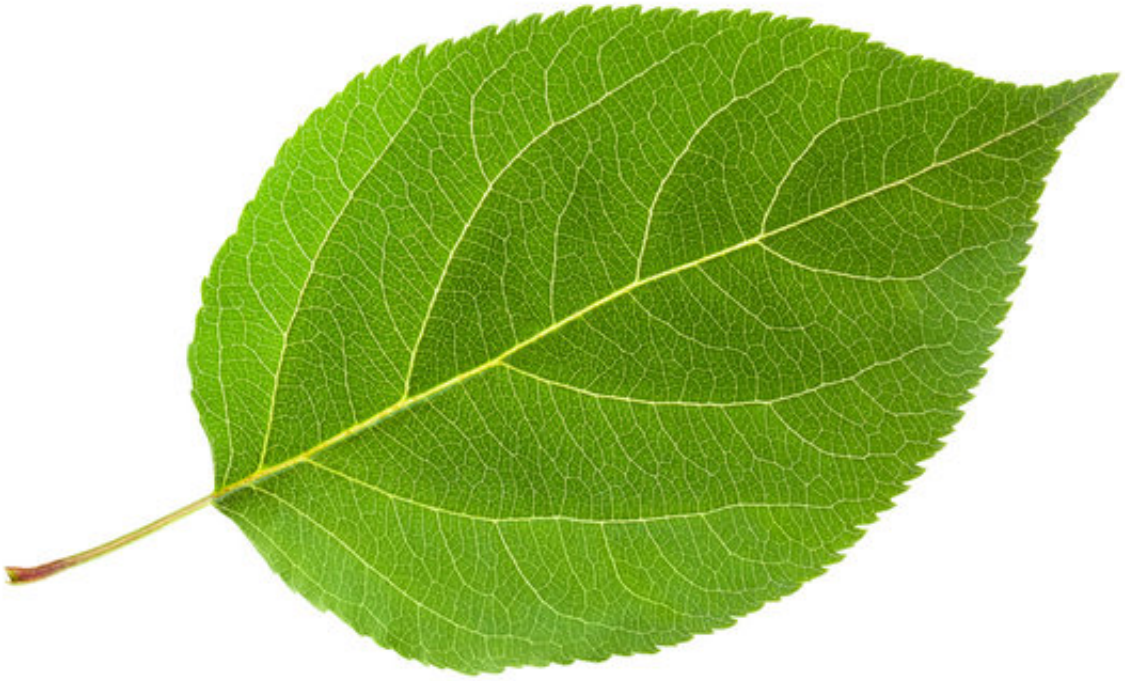
In [177]:
```
im = image.convert('L')

im
```

Out[177]:



In [194]:
```
# Importing the Image class from the Python Imaging Library (PIL)
from PIL import Image

# Enabling inline plotting for Matplotlib
%matplotlib inline

# Opening an image file named "leaf_image.jpg"
image = Image.open("leaf_image.jpg")

# Displaying the image
image
```

Out[194]:



In [195]:
```
array(im)
```

Out[195]:
```
array([[255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       ...,
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255]], dtype=uint8)
```

In [196]:
```
# Noise removal using morphological opening
kernel = np.ones((3, 3), np.uint8)
opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=

# Sure background area
sure_bg = cv2.dilate(opening, kernel, iterations=3)

# Sure foreground area
dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(

# Finding the unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg, sure_fg)
```

In [197]:
```
# Creating subplots with 1 row and 3 columns
f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True)

# Displaying the sure background, dilated
ax1.imshow(sure_bg, cmap='gray')
ax1.set_title('Sure background, dilated')

# Displaying the sure foreground, eroded
ax2.imshow(sure_fg, cmap='gray')
ax2.set_title('Sure foreground, eroded')

# Displaying the subtracted image (unknown region)
ax3.imshow(unknown, cmap='gray')
ax3.set_title('Subtract image')

# Fine-tuning the layout to adjust the spacing between subplots
f.subplots_adjust(wspace=15.9)
```