9. Perform Segmentation using

- Simple thresholding
- Otsu Binarization
- Adaptive Thresholding

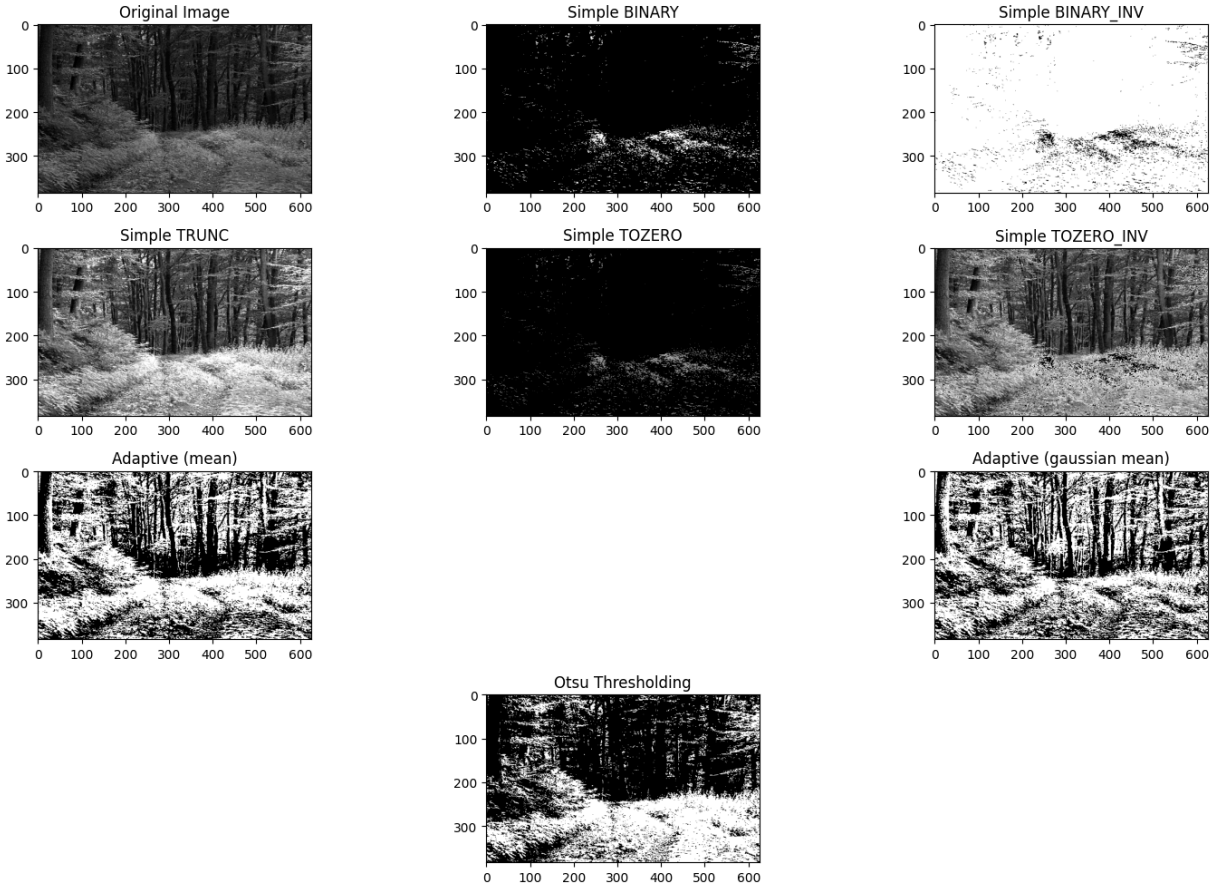
In [6]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img=cv2.imread("log.png")
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

In [7]:
```
#simple thresholding

thresh1 = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)[1]
thresh2 = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV)[1]
thresh3 = cv2.threshold(img, 127, 255, cv2.THRESH_TRUNC)[1]
thresh4 = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO)[1]
thresh5 = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO_INV)[1]
```

In [8]:
```
#adaptive thresholding

athresh1 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BI

athresh2 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRE
```

In [9]:
```
#otsu thresholding

othresh = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)[1]
```

In [10]:
```
plt.figure(figsize=(15,10))
plt.subplot(4, 3, 1), plt.imshow(img, cmap='gray'), plt.title('Original Image')
plt.subplot(4, 3, 2), plt.imshow(thresh1, cmap='gray'), plt.title('Simple BINARY')
plt.subplot(4, 3, 3), plt.imshow(thresh2, cmap='gray'), plt.title('Simple BINARY_IN
plt.subplot(4, 3, 4), plt.imshow(thresh3, cmap='gray'), plt.title('Simple TRUNC')
plt.subplot(4, 3, 5), plt.imshow(thresh4, cmap='gray'), plt.title('Simple TOZERO')
plt.subplot(4, 3, 6), plt.imshow(thresh5, cmap='gray'), plt.title('Simple TOZERO_IN
plt.subplot(4, 3, 7), plt.imshow(athresh1, cmap='gray'), plt.title('Adaptive (mean)
plt.subplot(4, 3, 9), plt.imshow(athresh2, cmap='gray'), plt.title('Adaptive (gauss
plt.subplot(4,3,11), plt.imshow(othresh, cmap='gray'), plt.title('Otsu Thresholding

plt.tight_layout()
plt.show()
```



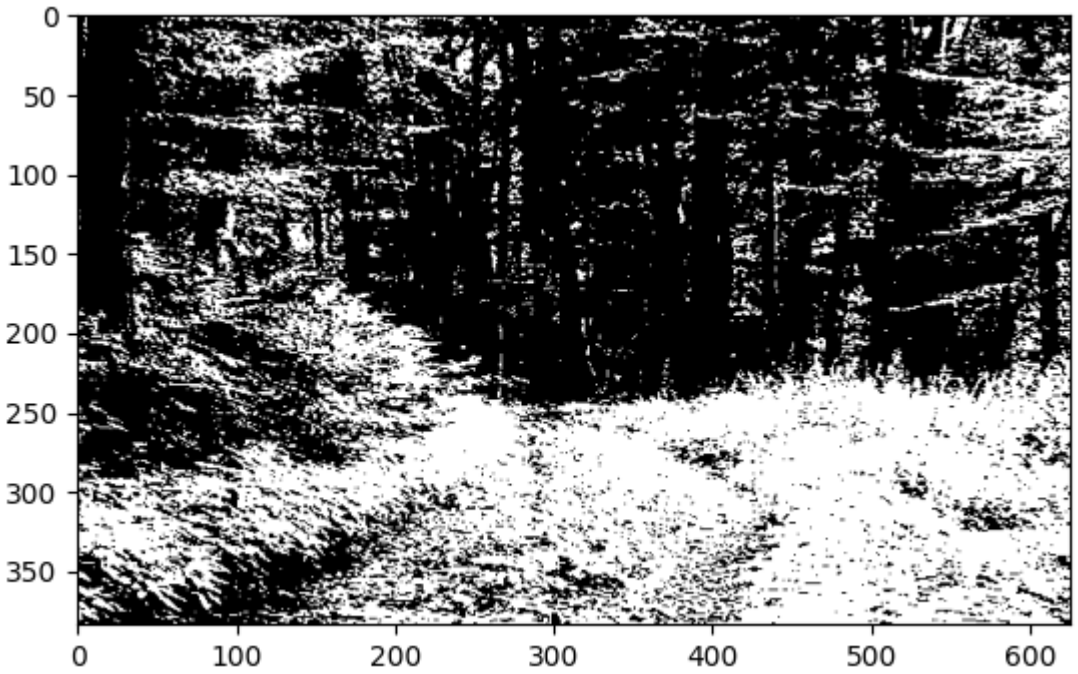
\f[dst(x,y) = \fork{\texttt{maxValue}}{if (src(x,y) > T(x,y))}{0}{otherwise}\f]

In [12]:

```python
import numpy as np
import cv2

def otsu_thresholding(image):
    # Step 1: Calculate histogram
    hist, _ = np.histogram(image.flatten(), bins=256, range=[0,256])

    # Step 2: Calculate probability density function (PDF)
    hist = hist.astype(np.float32)
    pdf = hist / np.sum(hist)

    # Step 3: Calculate cumulative distribution function (CDF)
    cdf = np.cumsum(pdf)

    # Step 4: Calculate cumulative mean (CM)
    cm = np.cumsum(np.arange(256) * pdf)

    # Step 5: Calculate between-class variance (BCV)
    bcv = np.zeros(256)
    for i in range(1, 256):
        if cdf[i] == 0:
            continue
        p1 = cdf[i]
        p2 = 1 - p1
        if p2 == 0:
            continue
```

```
        mu1 = cm[i] / p1
        mu2 = (cm[-1] - cm[i]) / p2
        bcv[i] = p1 * p2 * (mu1 - mu2) ** 2

    # Step 6: Find optimal threshold
    threshold = np.argmax(bcv)

    # Step 7: Apply threshold to image
    thresholded_image = (image > threshold).astype(np.uint8) * 255

    return thresholded_image, threshold

# Example usage:
# image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)
thresholded_image, threshold_value = otsu_thresholding(img)
# cv2.imwrite('output_image.jpg', thresholded_image)
plt.imshow(thresholded_image,cmap='gray')
```

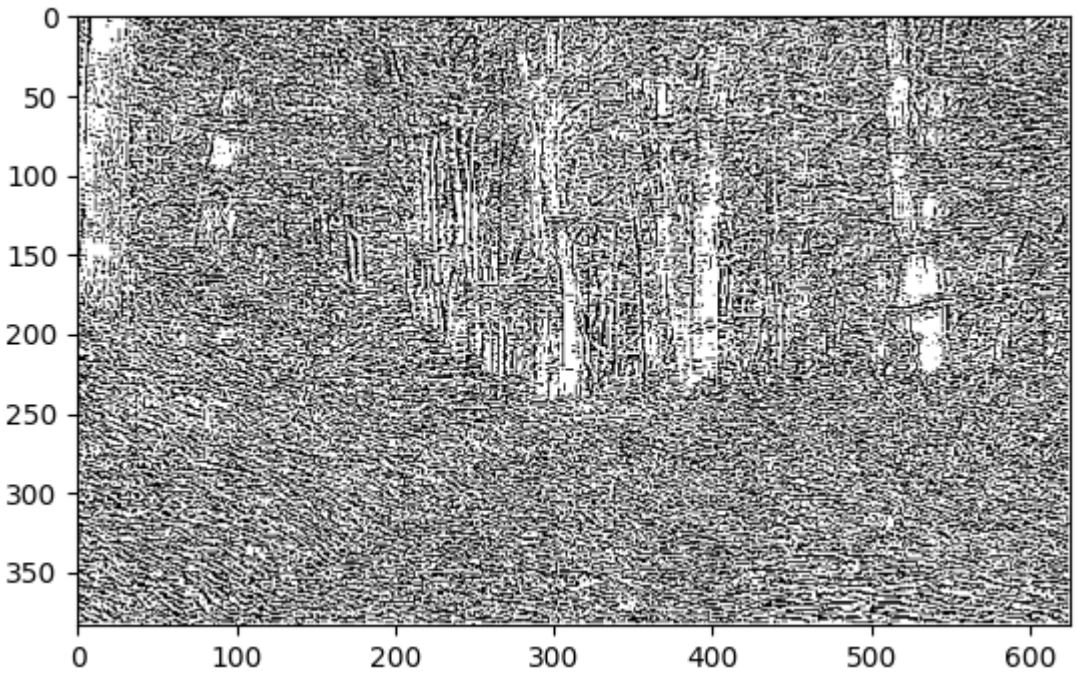
Out[12]: <matplotlib.image.AxesImage at 0x19b579c4250>



In [13]:
```
import numpy as np

def adaptive_thresholding(image, block_size=3, constant=2):
    # Convert image to grayscale if it's not already
    if len(image.shape) == 3:
        gray = image.mean(axis=2).astype(np.uint8)
    else:
        gray = image.astype(np.uint8)

    # Apply padding to handle borders
    pad_width = block_size // 2
    padded_image = np.pad(gray, pad_width, mode='constant', constant_values=0)

    # Initialize output image
    thresholded_image = np.zeros_like(gray)
```

```
    # Iterate over image pixels
    for i in range(pad_width, gray.shape[0] + pad_width):
        for j in range(pad_width, gray.shape[1] + pad_width):
            # Compute local mean
            local_mean = np.mean(padded_image[i-pad_width:i+pad_width+1, j-pad_widt

            # Apply threshold
            if gray[i-pad_width, j-pad_width] > local_mean - constant:
                thresholded_image[i-pad_width, j-pad_width] = 255
            else:
                thresholded_image[i-pad_width, j-pad_width] = 0

    return thresholded_image

# Example usage:
# Load your image here
# image = cv2.imread('input_image.jpg')
thresholded_image1 = adaptive_thresholding(img)
# cv2.imwrite('output_image.jpg', thresholded_image)
plt.imshow(thresholded_image1,cmap='gray')
```

Out[13]: <matplotlib.image.AxesImage at 0x19b5808cc10>

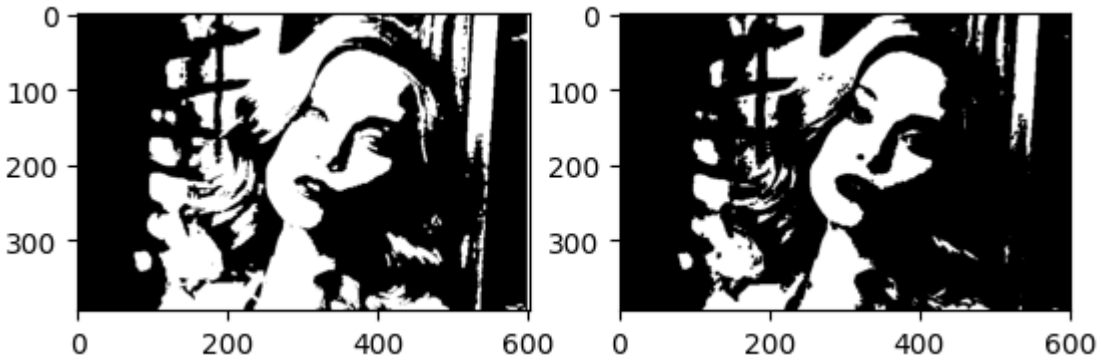


In [19]:
```
import numpy as np

def dilation(image, kernel):
    # Get image dimensions
    height, width = image.shape

    # Get kernel dimensions
    kernel_height, kernel_width = kernel.shape

    # Create output image
    dilated_image = np.zeros_like(image)
```

```
    # Iterate over image pixels
    for i in range(height):
        for j in range(width):
            # Initialize maximum value
            max_val = 0

            # Iterate over kernel
            for m in range(kernel_height):
                for n in range(kernel_width):
                    # Check if kernel is within image bounds
                    if i + m < height and j + n < width:
                        # Apply kernel
                        if kernel[m, n] == 1:
                            max_val = max(max_val, image[i + m, j + n])

            # Set output pixel value
            dilated_image[i, j] = max_val

    return dilated_image

def erosion(image, kernel):
    # Get image dimensions
    height, width = image.shape

    # Get kernel dimensions
    kernel_height, kernel_width = kernel.shape

    # Create output image
    eroded_image = np.zeros_like(image)

    # Iterate over image pixels
    for i in range(height):
        for j in range(width):
            # Initialize minimum value
            min_val = 255

            # Iterate over kernel
            for m in range(kernel_height):
                for n in range(kernel_width):
                    # Check if kernel is within image bounds
                    if i + m < height and j + n < width:
                        # Apply kernel
                        if kernel[m, n] == 1:
                            min_val = min(min_val, image[i + m, j + n])

            # Set output pixel value
            eroded_image[i, j] = min_val

    return eroded_image

# Example usage:
# Load your binary image here (should be in grayscale, with foreground pixels as wh
# image = np.array([[0, 0, 0, 0, 0],
#                   [0, 255, 255, 255, 0],
#                   [0, 255, 255, 255, 0],
```

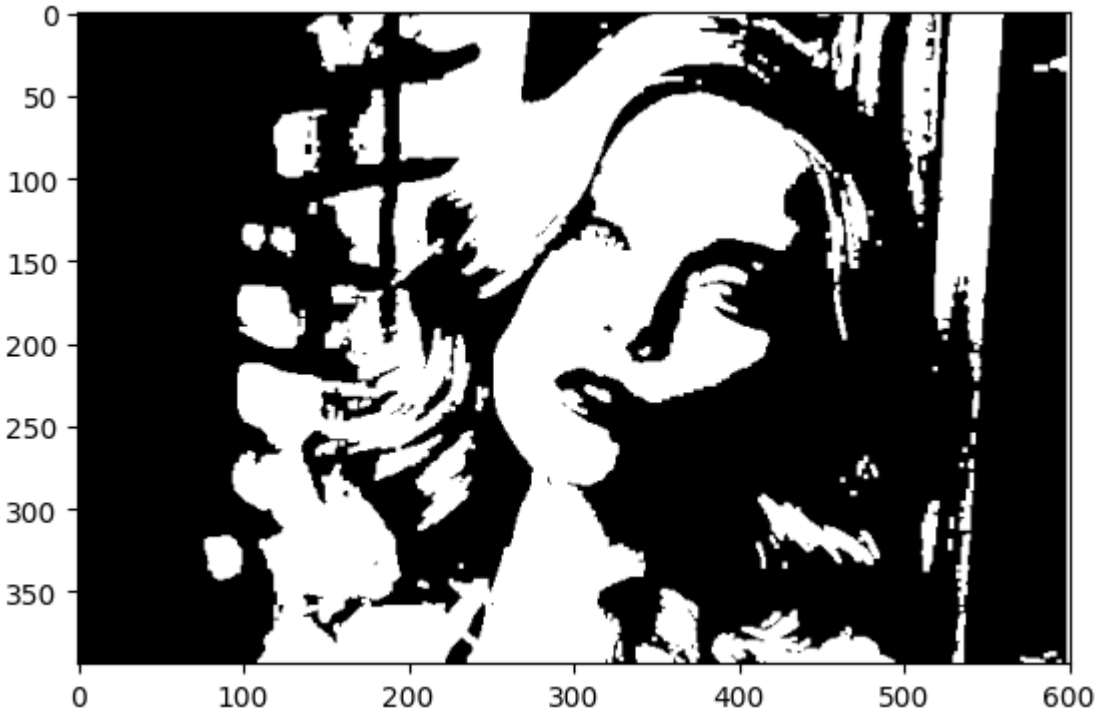
```
#                    [0, 255, 255, 255, 0],
#                    [0, 0, 0, 0, 0]], dtype=np.uint8)
# Define a kernel (structuring element)
kernel = np.array([[1, 1, 1],
                   [1, 1, 1],
                   [1, 1, 1]], dtype=np.uint8)
# Perform dilation
im = cv2.imread('edge.png',0)
binary = cv2.threshold(im,127,255,cv2.THRESH_BINARY)[1]
dilated_image = dilation(binary, kernel)
# Perform erosion
eroded_image = erosion(binary, kernel)
# You can adjust the kernel size and shape according to your requirements
plt.subplot(121)
plt.imshow(dilated_image,cmap='gray')
plt.subplot(122)
plt.imshow(eroded_image,cmap='gray')
```

Out[19]: <matplotlib.image.AxesImage at 0x19b57485250>



In [21]:
```
di = cv2.dilate(binary,kernel)
plt.imshow(di,cmap='gray')
```

Out[21]: <matplotlib.image.AxesImage at 0x19b5a527590>



In [ ]: