3. Sobel and Canny Edge Detection

Table of Contents

1. Libraries
2. Sobel Edge Detection
3. Canny Edge Detection
4. Homework

Importing Libraries

```
import cv2
import skimage
import numpy as np
from scipy import ndimage
from skimage import exposure
from PIL import Image, ImageOps
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy.ndimage import convolve
from scipy.ndimage import gaussian_filter as gauss
from scipy.ndimage import median_filter as med
```

Sobel Edge Detection

As a first step in extracting features, you will apply the Sobel edge detection algorithm. This finds regions of the image with large gradient values in multiple directions. Regions with high omnidirectional gradient are likely to be edges or transitions in the pixel values.

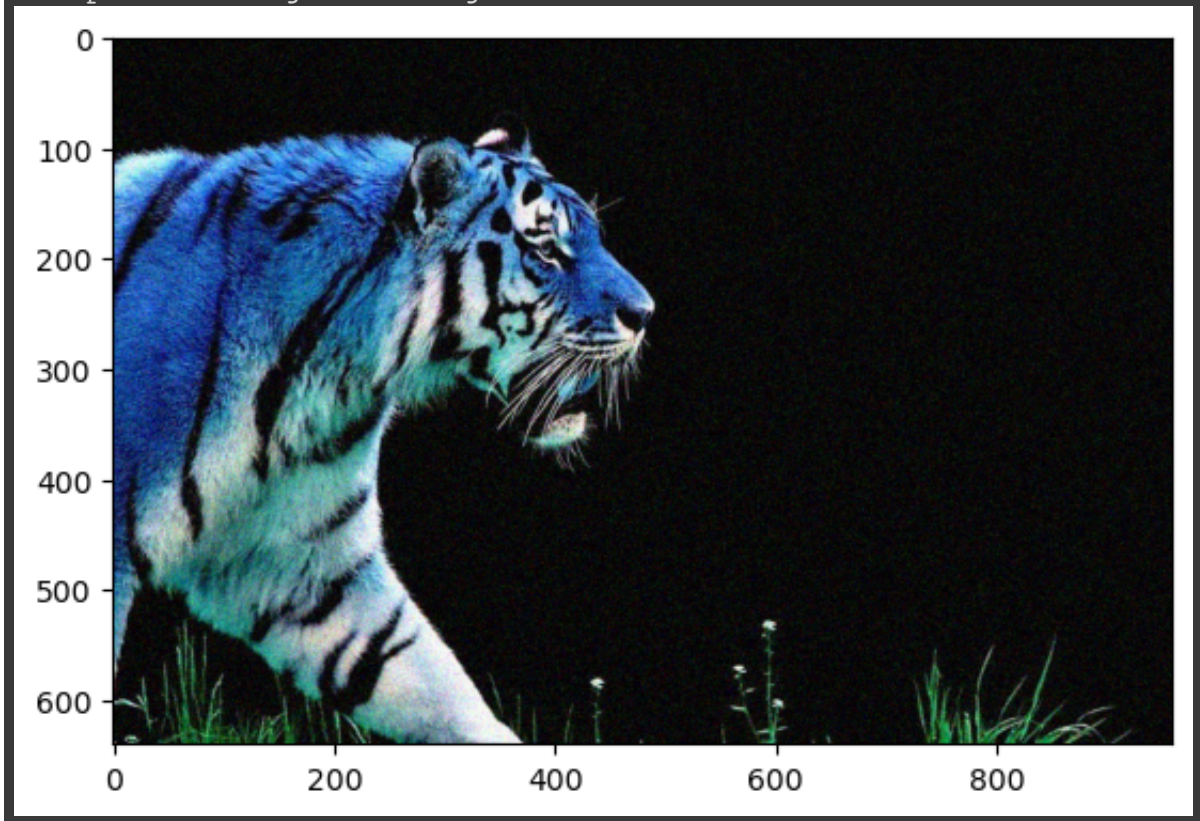
The code in the cell below applies the Sobel algorithm to the median filtered image, using these steps:

1. Convert the color image to grayscale for the gradient calculation since it is two dimensional.
2. Compute the gradient in the x and y (horizontal and vertical) directions.
3. Compute the magnitude of the gradient.
4. Normalize the gradient values.

```
def edge_sobel(image):
    from scipy import ndimage
    import skimage.color as sc
    import numpy as np
    image = sc.rgb2gray(image) # Convert color image to gray scale
    dx = ndimage.sobel(image, 1)  # horizontal derivative
    dy = ndimage.sobel(image, 0)  # vertical derivative
    mag = np.hypot(dx, dy)  # magnitude
    mag *= 255.0 / np.amax(mag)  # normalize (Q&D)
    mag = mag.astype(np.uint8)
    return mag
```

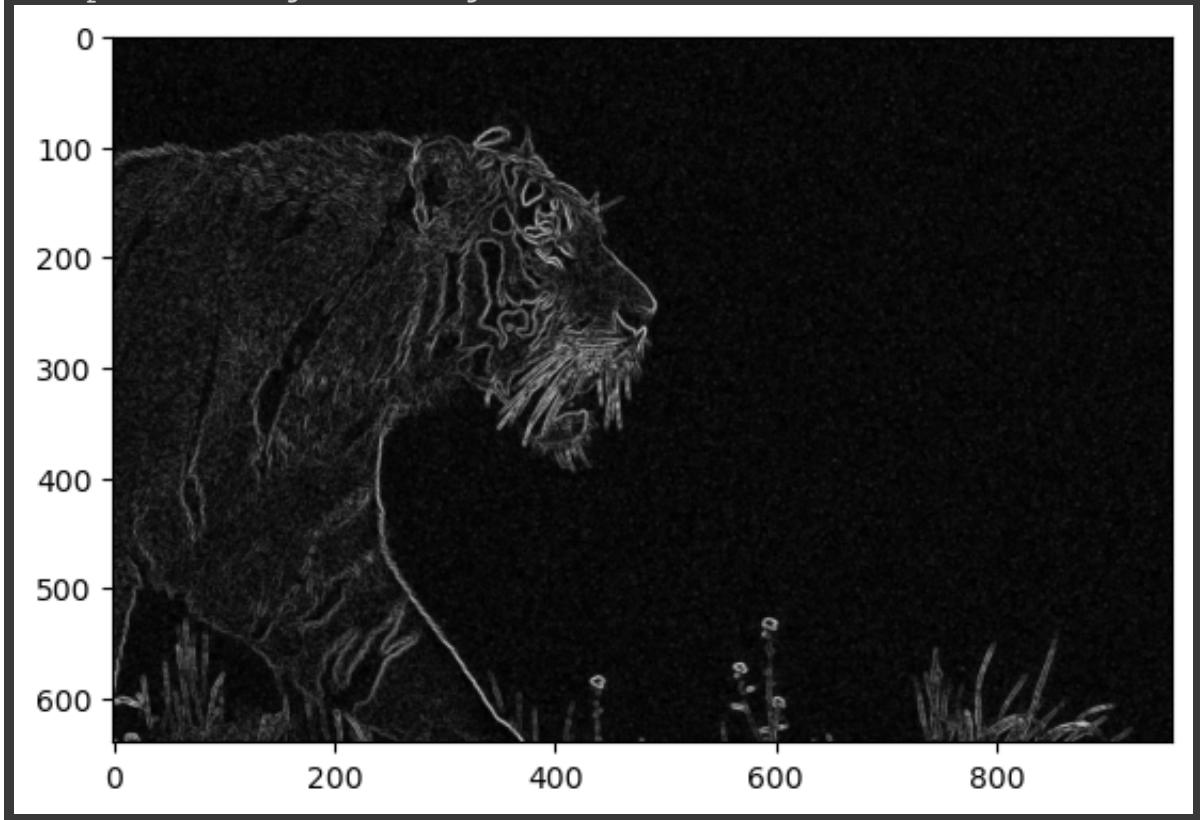
```
original_image = cv2.imread("data/img.jpg")

original_image_array = np.array(original_image)

img = skimage.util.random_noise(original_image_array)
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x14c6946b0>
```



```
img_med = med(img, size=2)
img_edge = edge_sobel(img_med)
plt.imshow(img_edge, cmap="gray")
```

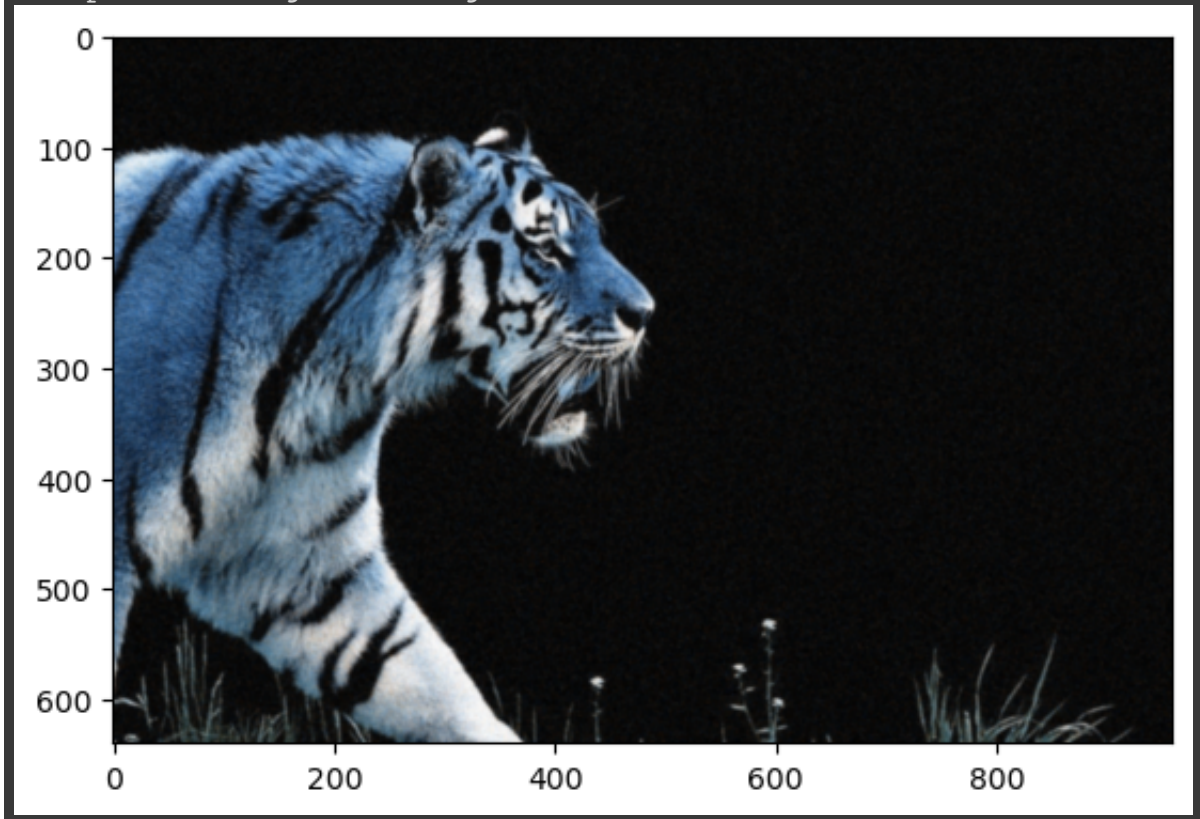
```
<matplotlib.image.AxesImage at 0x14c720530>
```



Now let's try with the more blurred gaussian filtered image.

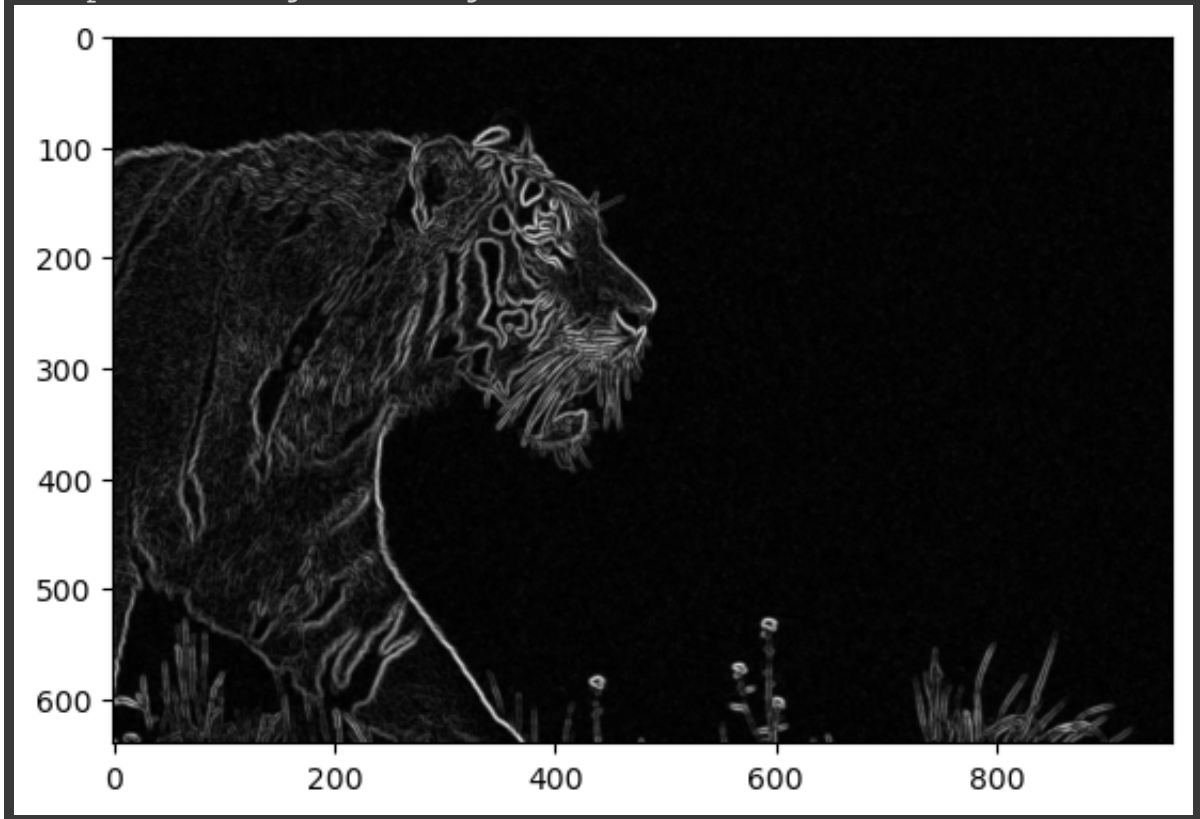
```
img_gauss = gauss(img, sigma=1)
plt.imshow(img_gauss)
```

```
<matplotlib.image.AxesImage at 0x14c7e1010>
```



```
img_edge = edge_sobel(img_gauss)
plt.imshow(img_edge, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14c8128d0>
```



Canny Edge Detection

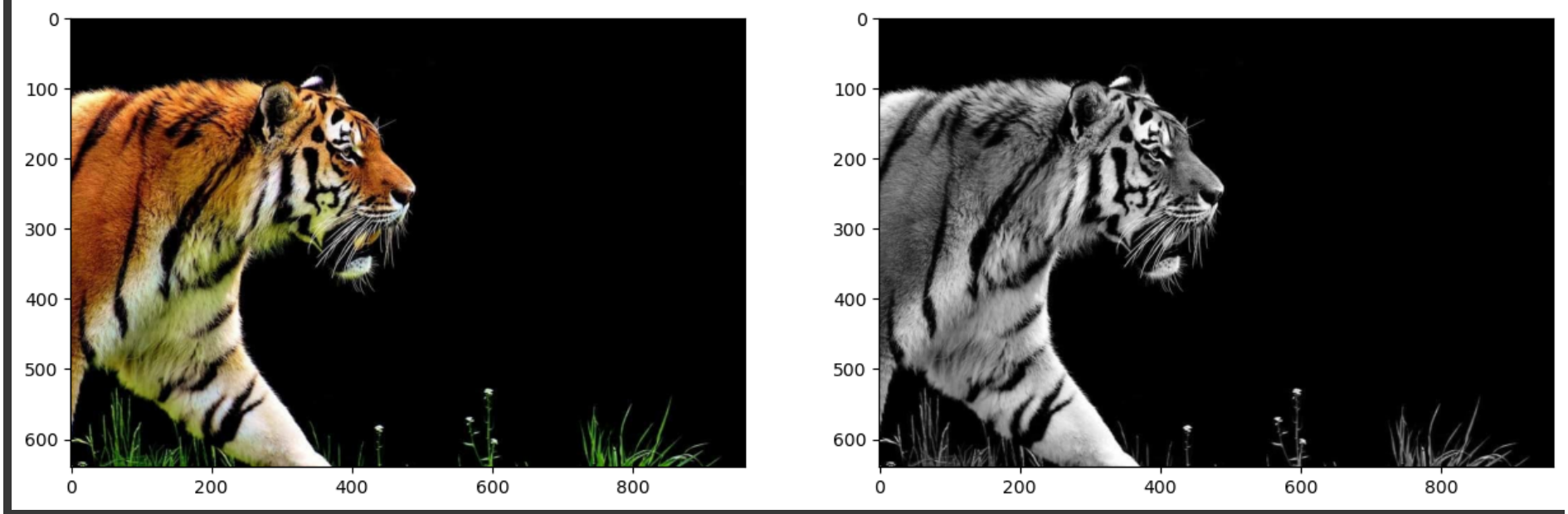
Steps:

1. Noise Reduction
2. Gradient Calculation
3. Non-maximum Supression
4. Double Threshold
5. Edge Tracking by Hysteresis

Pre-requisite: Convert the image to grayscale before algorithm.

```
img = original_image

img_color = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(img_color)
plt.subplot(1, 2, 2)
plt.imshow(img_gray, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14c908b90>
```



1. Noise Reduction

Edge detection are highly sensitive to image noise due to the derivatives behind the algorithm.

We can apply a Gaussian Kernel, the size of the kernel depends on the expected blurring effect. The smaller the less blurring effect.

Equation for Gaussian Kernel of size $(2k + 1) \times (2k + 1)$

$$H_{i,j} = \frac{1}{2\pi\sigma^2} exp(-\frac{(i - (k + 1))^2 + (j - (k + 1))^2}{2\sigma^2}); 1 \leq i, j \leq (2k + 1)$$

```
def gaussian_kernel(size, sigma=1):
    size = int(size) // 2
    x, y = np.mgrid[-size:size+1, -size:size+1]
    normal = 1 / (2.0 * np.pi * sigma**2)
    g =  np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
    return g
```

1.1 Sigma Parameter σ

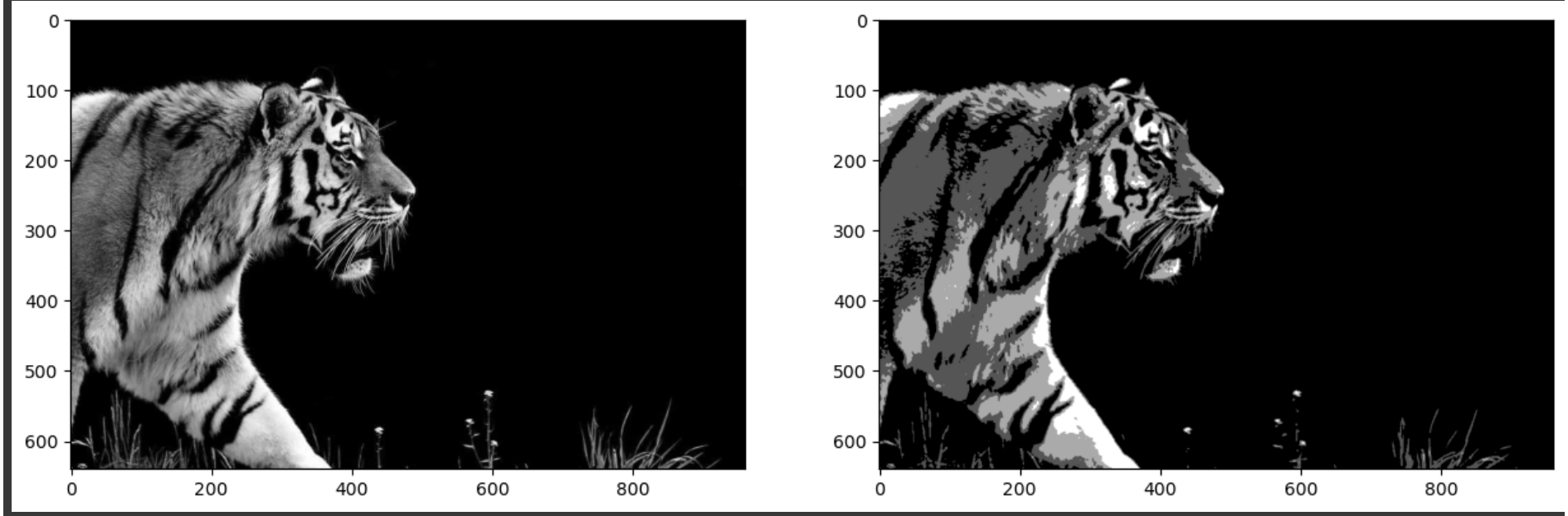
```
# Change this parameter
sigma = 10
```

1.2 Kernel Size Parameter

```
kernel_size = 3
```

```
img_gaussian = convolve(img_gray, gaussian_kernel(kernel_size, sigma))

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(img_gaussian, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14c9beed0>
```



2. Gradient Calculation

Edges correspond to a change of pixels intensity.

To detect it, the easiest way is to apply filters that highlight this intensity change in both directions:

- horizontal (x)
- and vertical (y)

It can be implemented by convoling I with *Sobel kernels* Kx and Ky

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

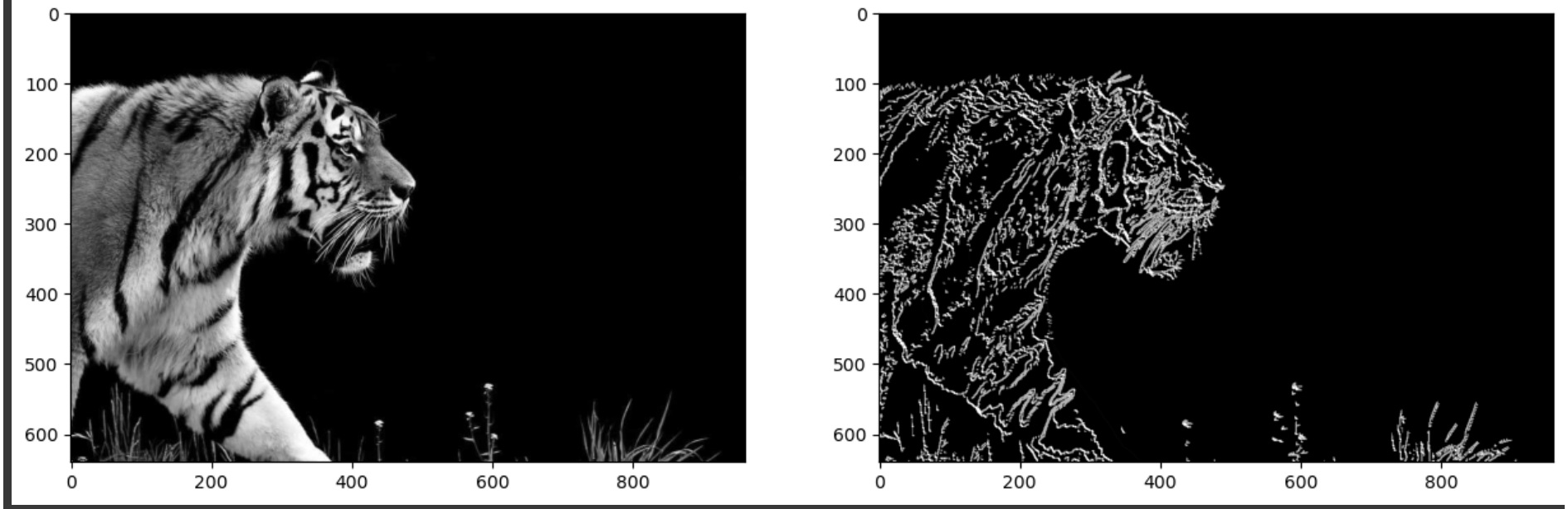
Then, the magnitude G and the slope θ of the gradient are calculated as follow:

$$|G| = \sqrt{I_x^2 + I_y^2}, \theta(x, y) = arctan(\frac{I_y}{I_x})$$

```
def sobel_filters(img):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)

    Ix = convolve(img, Kx)
    Iy = convolve(img, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)

    return (G, theta)
```

```
G, theta = sobel_filters(img_gaussian)

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(G, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14ca6a1b0>
```



3. Non-Maximum supression

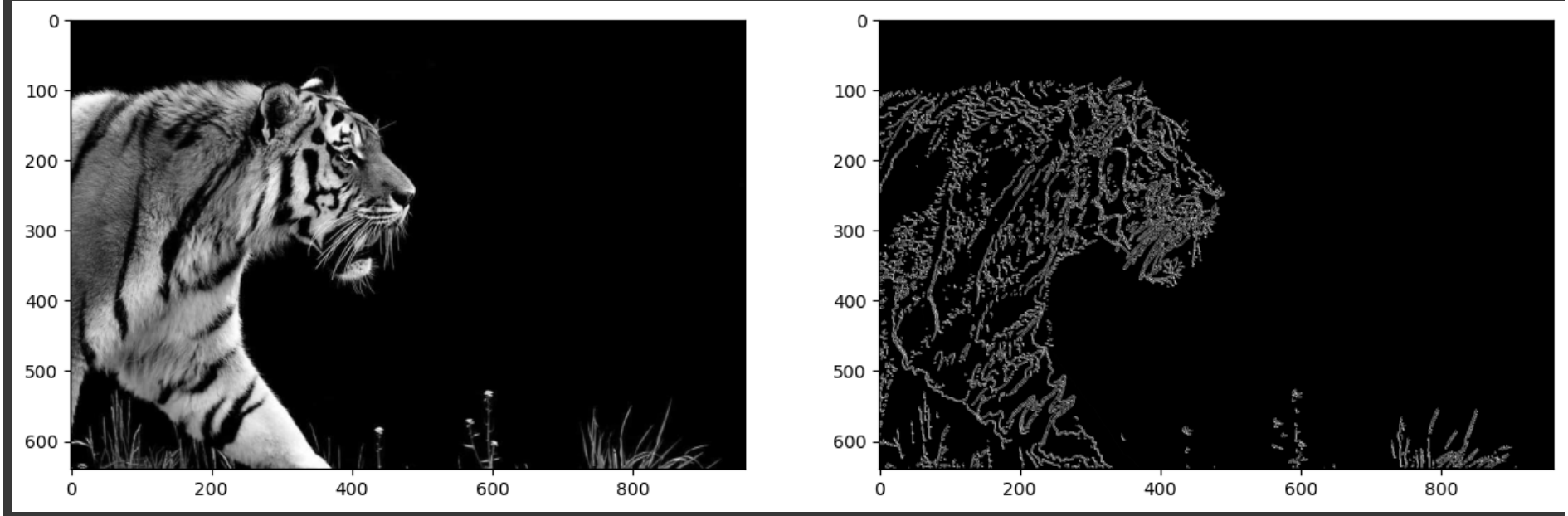
1. Create a matrix initialized to 0 of the same size of the original gradient intensity matrix
2. Identify the edge direction based on the angle value from the angle matrix
3. Check if the pixel in the same direction has a higher intensity than the pixel that is currently processed
4. Return the image processed with the non-max suppression algorithm.

```
def non_max_suppression(img, D):
    M, N = img.shape
    Z = np.zeros((M,N), dtype=np.int32)
    angle = D * 180. / np.pi
    angle[angle < 0] += 180


    for i in range(1,M-1):
        for j in range(1,N-1):
            try:
                q = 255
                r = 255

               #angle 0
                if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                    q = img[i, j+1]
                    r = img[i, j-1]
                #angle 45
                elif (22.5 <= angle[i,j] < 67.5):
                    q = img[i+1, j-1]
                    r = img[i-1, j+1]
                #angle 90
                elif (67.5 <= angle[i,j] < 112.5):
                    q = img[i+1, j]
                    r = img[i-1, j]
```

```
                #angle 135
                elif (112.5 <= angle[i,j] < 157.5):
                    q = img[i-1, j-1]
                    r = img[i+1, j+1]

                if (img[i,j] >= q) and (img[i,j] >= r):
                    Z[i,j] = img[i,j]
                else:
                    Z[i,j] = 0

            except IndexError as e:
                pass

    return Z
```

```
img_nonmax = non_max_suppression(G, theta)

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(img_nonmax, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14cb1fef0>
```



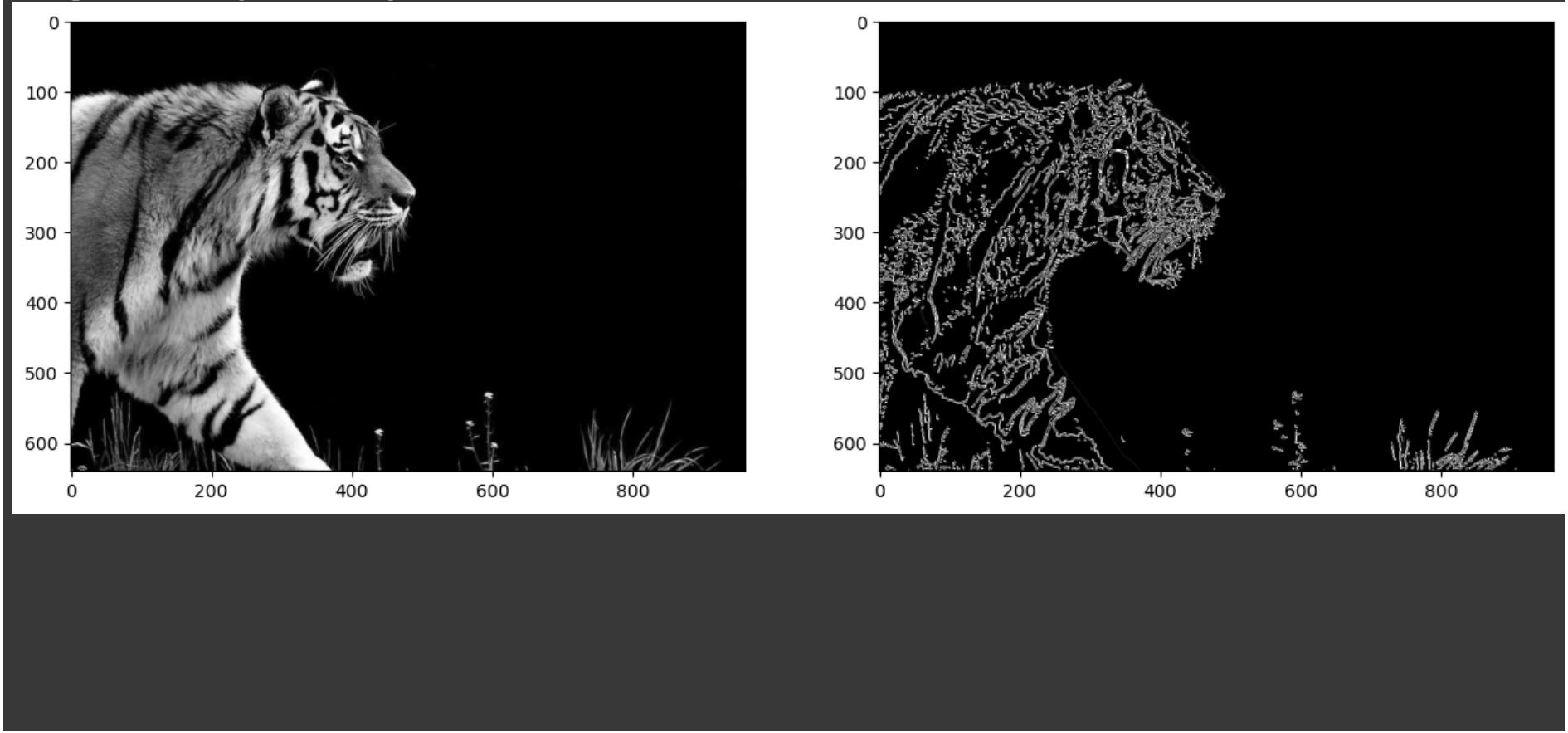
4. Double threshold

- Strong pixels are pixels that have an intensity so high that we are sure they contribute to the final edge.
- Weak pixels are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as non-relevant for the edge detection.
- Other pixels are considered as non-relevant for the edge.

```
def threshold(img, lowThresholdRatio=0.05, highThresholdRatio=0.09):

    highThreshold = img.max() * highThresholdRatio;
    lowThreshold = highThreshold * lowThresholdRatio;

    M, N = img.shape
    res = np.zeros((M,N), dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)

    strong_i, strong_j = np.where(img >= highThreshold)
    zeros_i, zeros_j = np.where(img < lowThreshold)

    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThreshold))

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak

    return (res)
```

```
img_threshold = threshold(img_nonmax)

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(img_threshold, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14cc03a40>
```



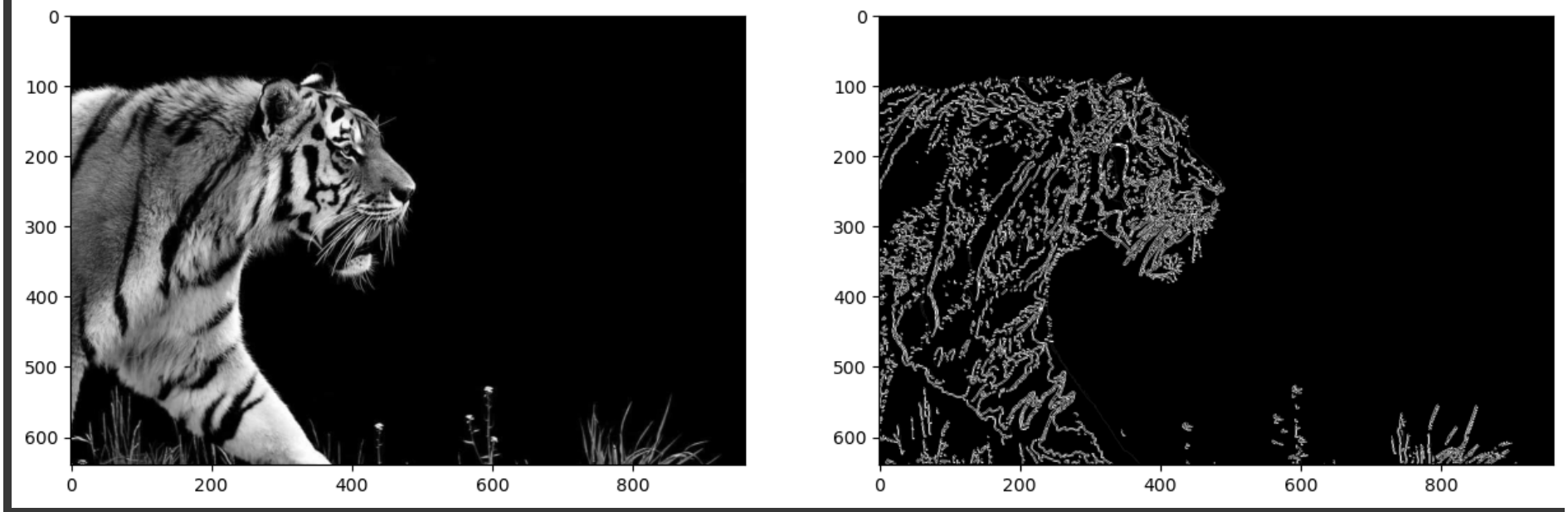
5. Edge Tracking by Hysteresis

The hysteresis consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one

```
def hysteresis(img, weak = 75, strong=255):
    M, N = img.shape
    for i in range(1, M-1):
        for j in range(1, N-1):
            if (img[i,j] == weak):
                try:
                    if ((img[i+1, j-1] == strong) or (img[i+1, j] == strong) or (img[i+1, j+1] == strong)
                        or (img[i, j-1] == strong) or (img[i, j+1] == strong)
                        or (img[i-1, j-1] == strong) or (img[i-1, j] == strong) or (img[i-1, j+1] == strong)
                        img[i, j] = strong
                    else:
                        img[i, j] = 0
                except IndexError as e:
                    pass
    return img
```

```
img_final = hysteresis(img_threshold)

plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(img_final, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x14c695520>
```



Homework

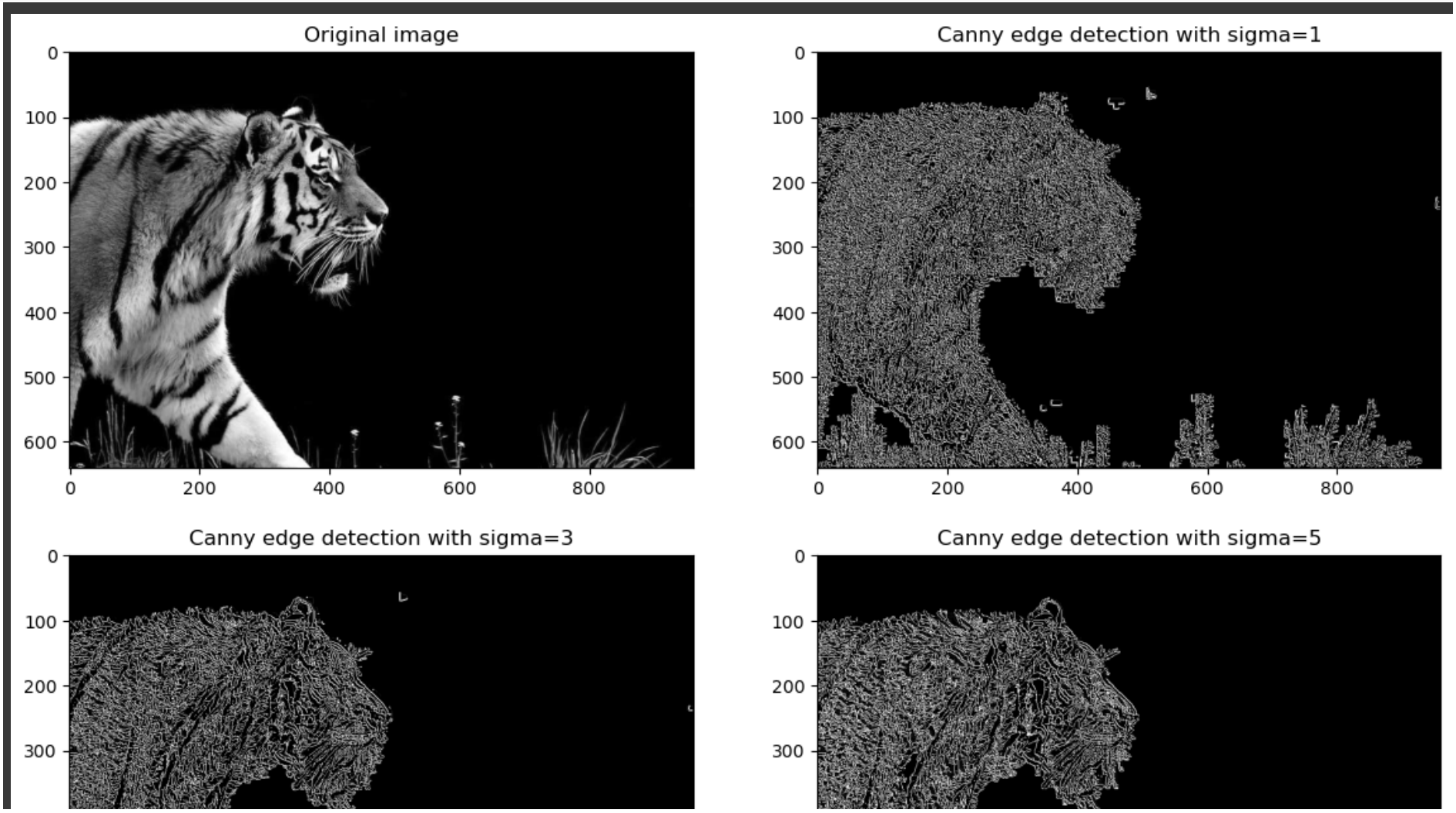
```
def canny_edge_detection(img_gray, kernel_size, sigma, remove_noise):
    """
    Common function that encompases the entire notebook's canny
    edge detection algorithm.
    """
    img_to_process = convolve(img_gray, gaussian_kernel(kernel_size, sigma)) if remove_noise else img_gray
    G, theta = sobel_filters(img_to_process)
    img_nonmax = non_max_suppression(G, theta)
    img_threshold = threshold(img_nonmax)
    img_final = hysteresis(img_threshold)
    return img_final

def load_gray_image(image_path):
    """
    Loads a gray version of an image given the image path.
    """
    img = cv2.imread(image_path)
    return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

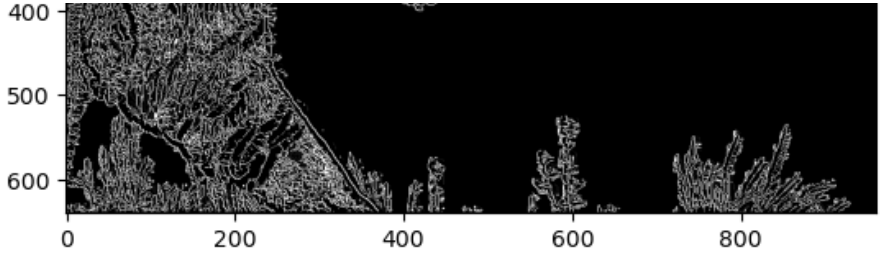
```
img_gray = load_gray_image("data/img.jpg")
```

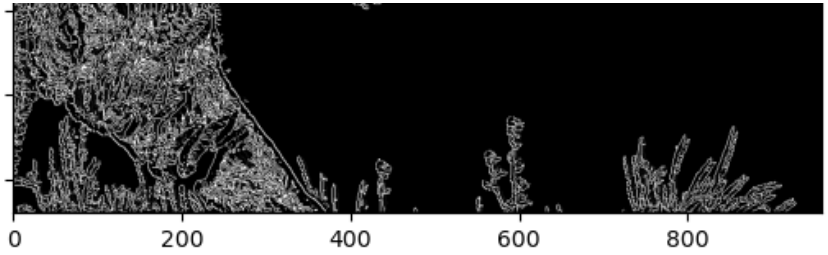
a) Modify the sigma value and observe the changes in the lines you detect

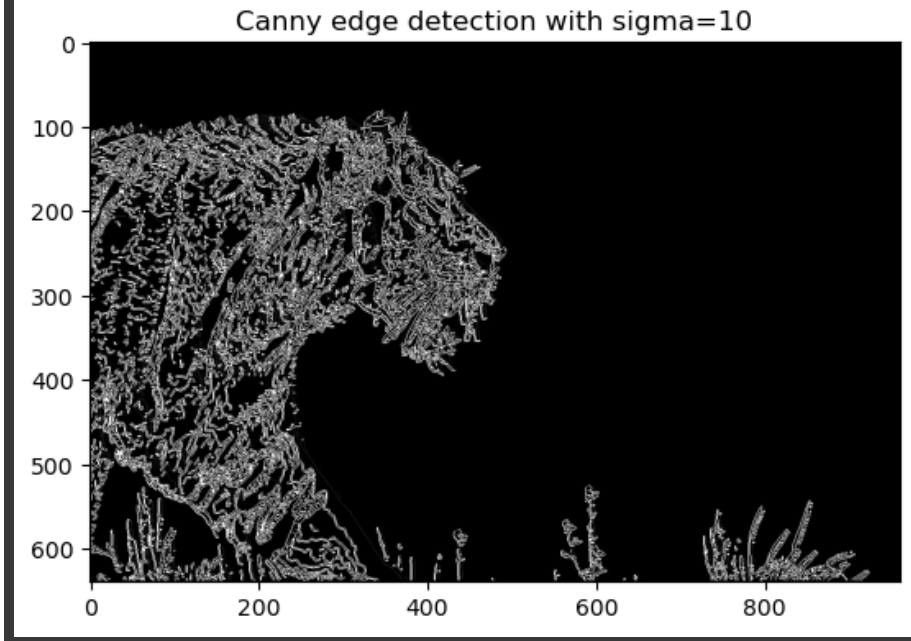
```
sigmas = [1, 3, 5, 10, 20]

kernel_size = 5 # This kernel value provides better results for the tiger image.

plt.figure(figsize=(14, 5 * len(sigmas)))

plt.subplot(len(sigmas), 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.title("Original image")
```

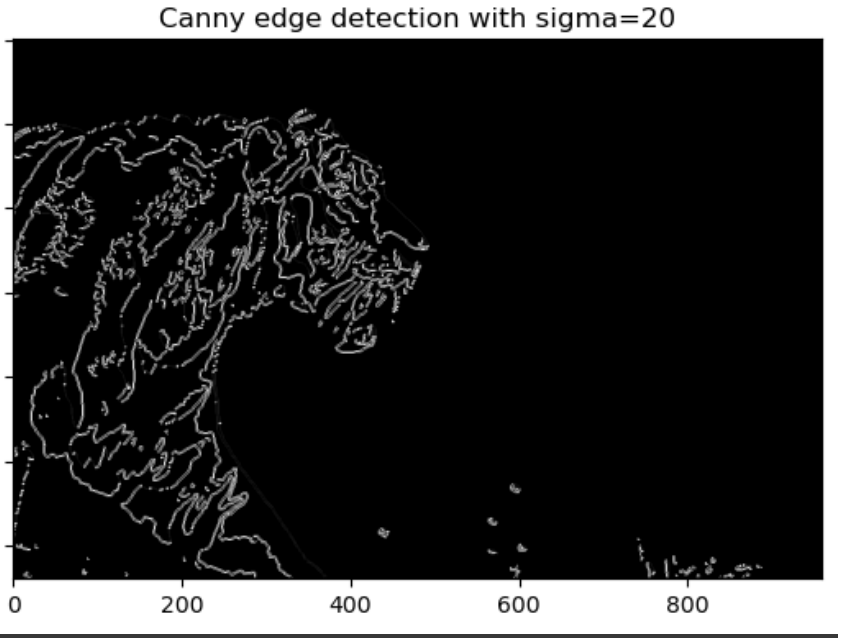
```
for i, sigma in enumerate(sigmas, 1):

    img_canny_edge_detection = canny_edge_detection(img_gray, kernel_size, sigma, True)

    plt.subplot(len(sigmas), 2, i+1)
    plt.imshow(img_canny_edge_detection, cmap="gray")
    plt.title(f"Canny edge detection with sigma={sigma}")

plt.show()
```




400
500
600
0
200
400
600
800


400
500
600
0
200
400
600
800

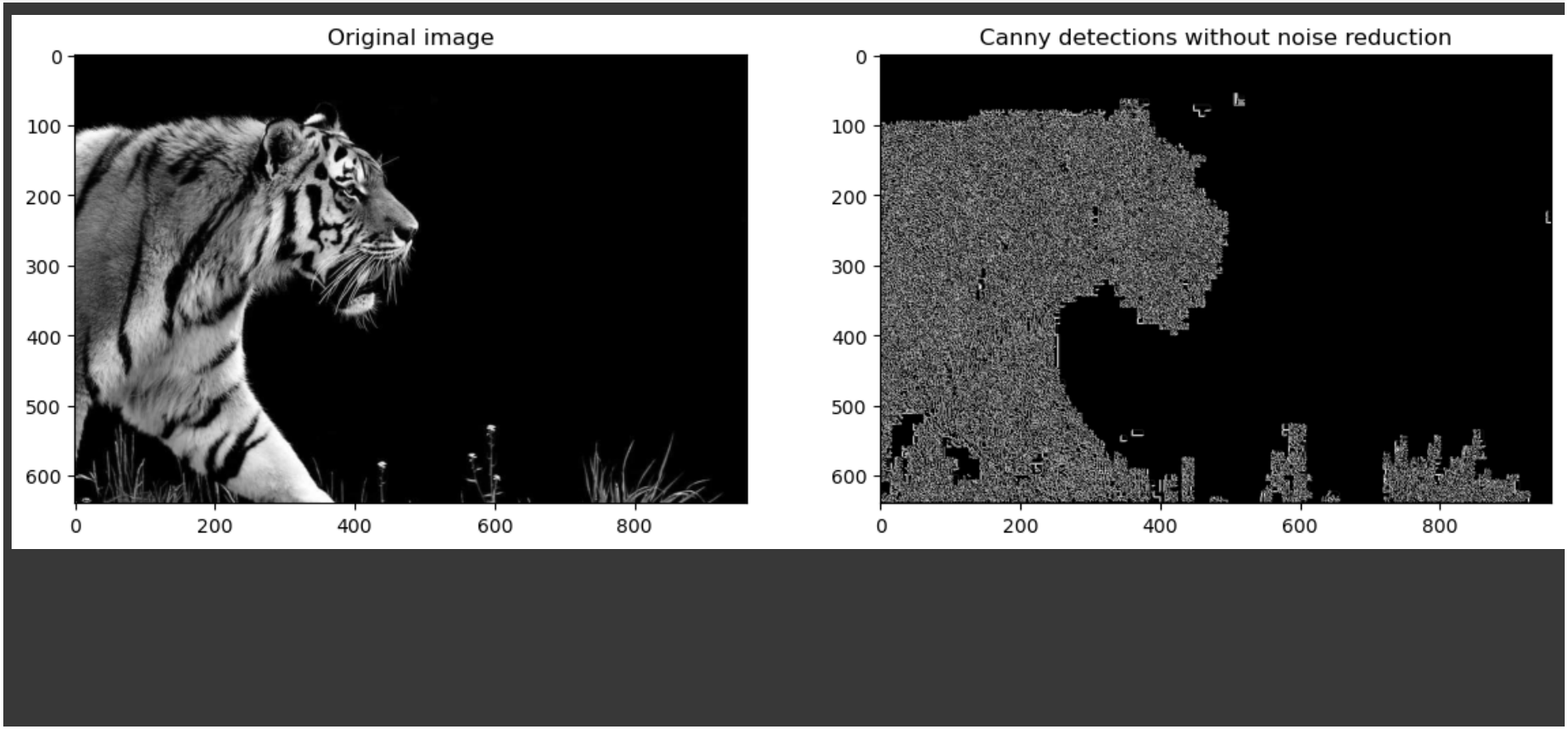

Canny edge detection with sigma=10
0
100
200
300
400
500
600
0
200
400
600
800


Canny edge detection with sigma=20
0
100
200
300
400
500
600
0
200
400
600
800

Using a sigma value of 5 with the Canny algorithm shows slight improvements in capturing fine details like fur and whiskers in the image. This setting strikes a balance between highlighting these subtle edges without adding too much noise. However, when we increase the sigma value to 20, we notice that the algorithm becomes better at identifying larger and more prominent details, such as the distinct stripes on a tiger's fur. With this higher sigma value, the algorithm suppresses noise even more effectively, resulting in sharper and clearer edges. So, the choice of sigma value affects how well the algorithm detects different features in the image – lower values are better for finer details, while higher values are more suited to capturing larger and more noticeable features.

b) Analyze the behavior of the Canny Detector with and without noise removal

```
img_canny_edge_detection = canny_edge_detection(img_gray, kernel_size, None, False)

plt.figure(figsize=(14, 5 * len(sigmas)))

plt.subplot(len(sigmas), 2, 1)
plt.imshow(img_gray, cmap="gray")
plt.title("Original image")

plt.subplot(len(sigmas), 2, 2)
plt.imshow(img_canny_edge_detection, cmap="gray")
plt.title("Canny detections without noise reduction")

plt.show()
```



Without noise reduction, the Canny algorithm faces difficulties because it identifies edges solely based on sudden shifts in brightness. When an image contains noise, which causes minor fluctuations in brightness, the algorithm may incorrectly interpret these fluctuations as edges. This oversight results in a less precise edge detection outcome, with numerous false edges generated due to the misinterpretation of noise. Consequently, the edge detection image exhibits an abundance of edges, many of which are not genuine features but artifacts caused by noise.

Hence, while using the image of the tiger without noise, the Canny algorithm basically detected that the entire body of the tiger was comprised of edges.

c) Experiment with images with different numbers of lines and texture to observe the behavior of the algorithm

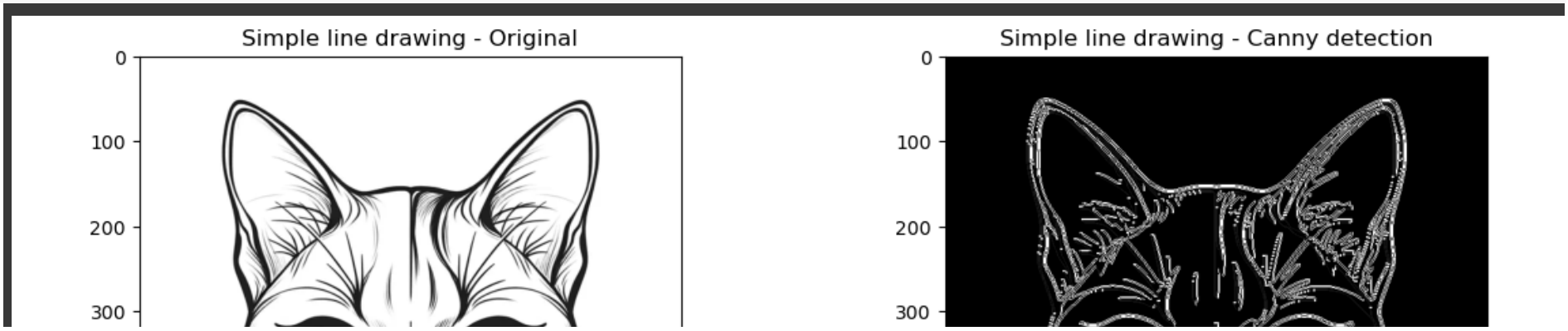
We'll be using he following images in order to test how the algorithm detect edges under different circumstances:

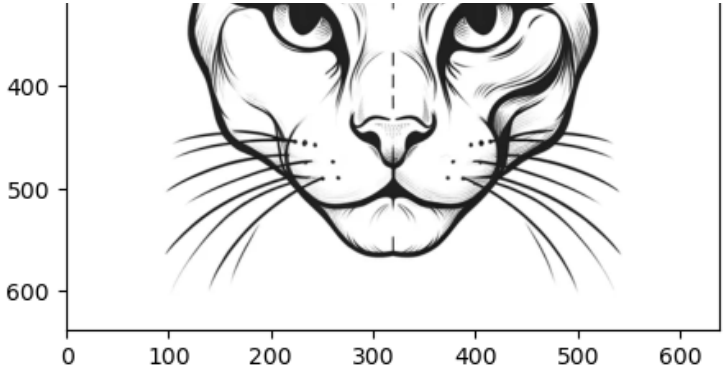
- Simple drawings: Images with clear, straight lines can help see how well the algorithm detects sharp edges.
- Natural scenery: Photos of nature, like forests or fields, show how the algorithm handles complex textures and edges.
- Repetitive patterns: Pictures of things like brick walls or fabrics demonstrate how it deals with repetitive patterns.
- Artificial patterns: Images with grids or mazes help understand how it detects edges in structured patterns.
- Gradients: Pictures with smooth color changes show how it handles gradual transitions.

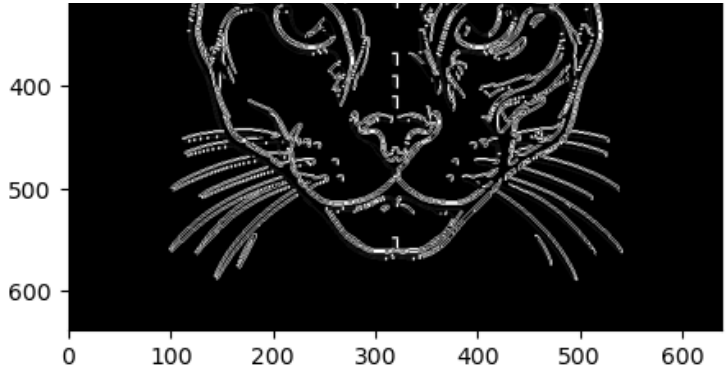
```
img_simple_line_drawing = load_gray_image("data/img_simple_line_drawing.jpg")
img_natural_landscape = load_gray_image("data/img_natural_landscape.jpg")
img_tile_pattern = load_gray_image("data/img_tile_pattern.jpg")
img_maze = load_gray_image("data/img_maze.jpg")
img_gradient = load_gray_image("data/img_gradient.jpg")
```

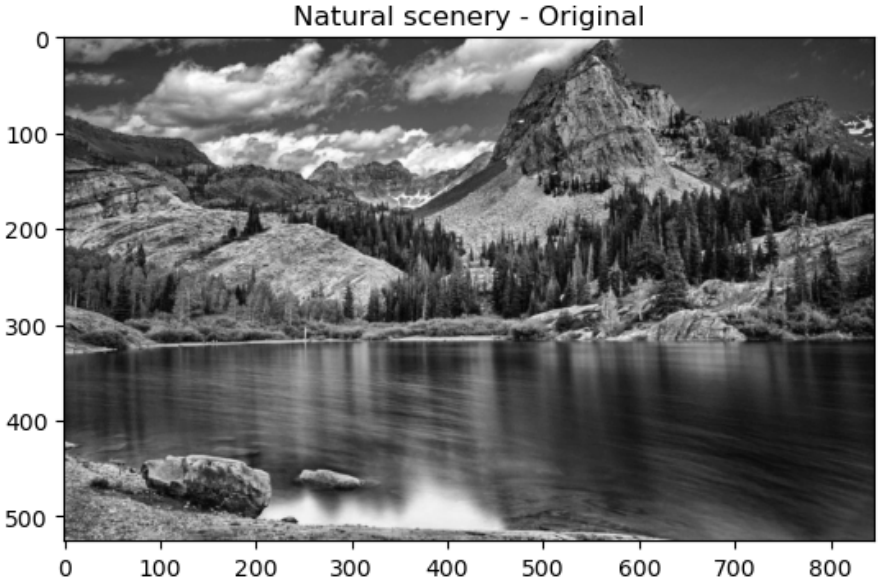
```
image_data_array = [
    ("Simple line drawing", img_simple_line_drawing),
    ("Natural scenery", img_natural_landscape),
    ("Repetitive pattern", img_tile_pattern),
    ("Artificial pattern", img_maze),
    ("Gradient", img_gradient)
]
```

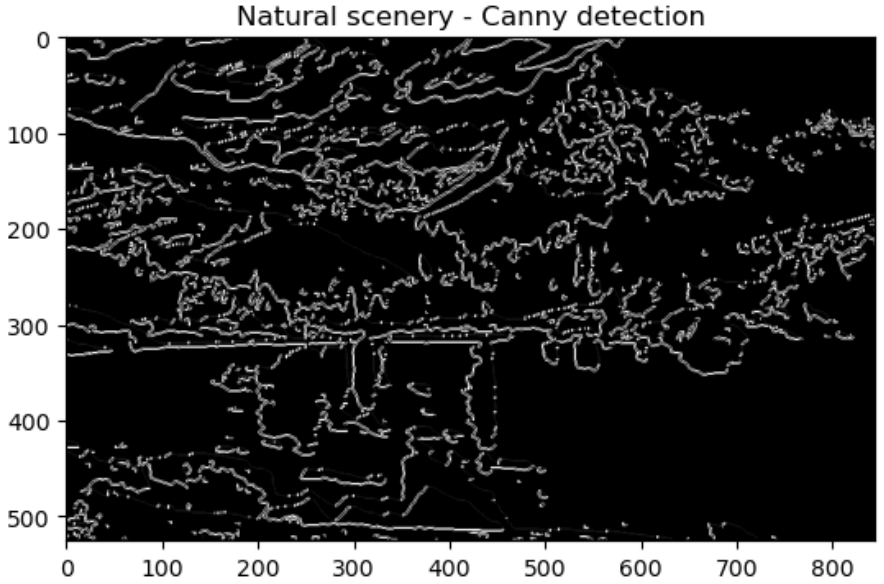
```
plt.figure(figsize=(14, 6 * len(image_data_array)))

for i, image_data in enumerate(image_data_array, 1):
    title = image_data[0]
    img = image_data[1]

    img_canny_edge_detection = canny_edge_detection(img, kernel_size, sigma, True)

    plt.subplot(len(image_data_array), 2, 2*i-1)
    plt.imshow(img, cmap="gray")
    plt.title(f"{title} - Original")

    plt.subplot(len(image_data_array), 2, 2*i)
    plt.imshow(img_canny_edge_detection, cmap="gray")
    plt.title(f"{title} - Canny detection")

plt.show()
```

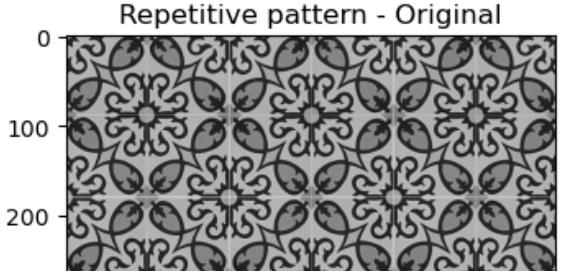


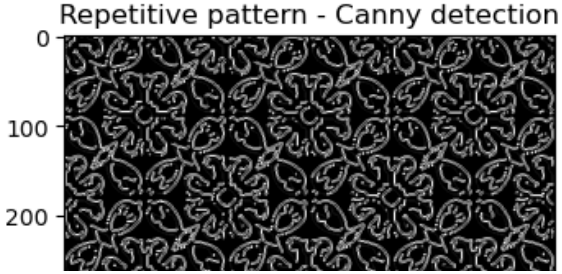

0
100
200
300
400
500
600

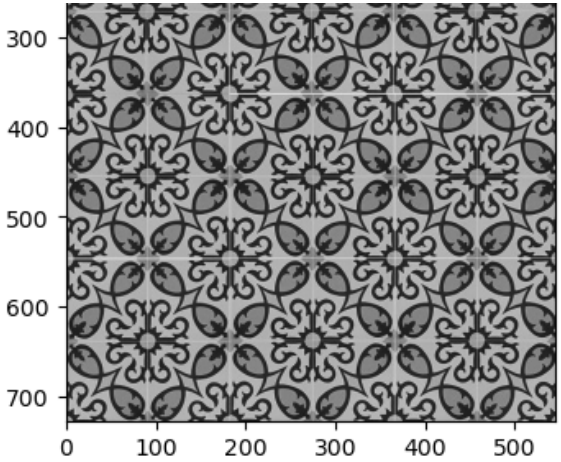

0
100
200
300
400
500
600

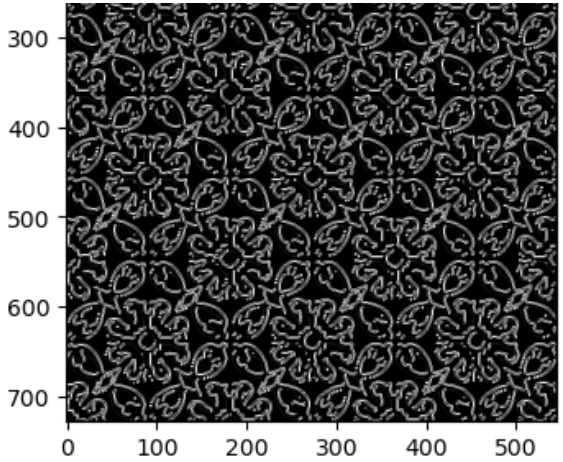

Natural scenery - Original
0
100
200
300
400
500
0
100
200
300
400
500
600
700
800


Natural scenery - Canny detection
0
100
200
300
400
500
0
100
200
300
400
500
600
700
800

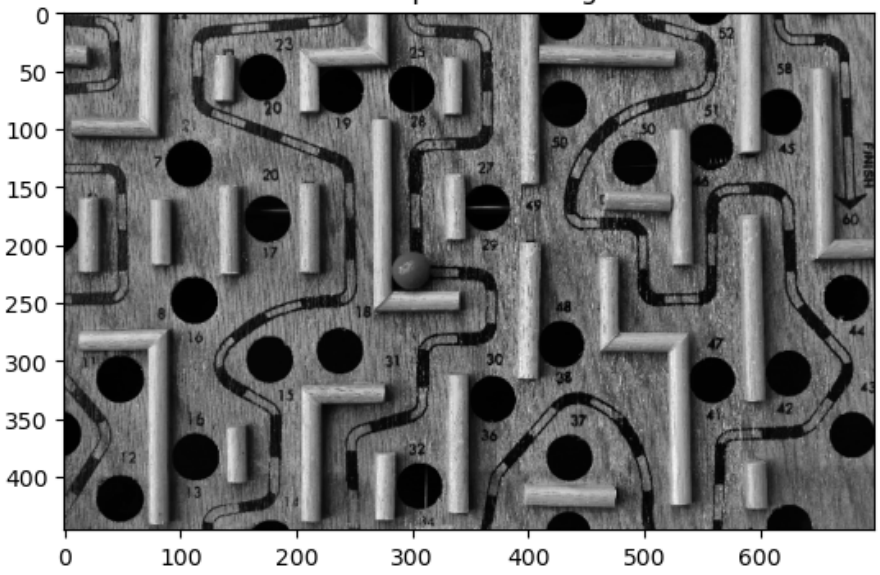

Repetitive pattern - Original
0
100
200


Repetitive pattern - Canny detection
0
100
200

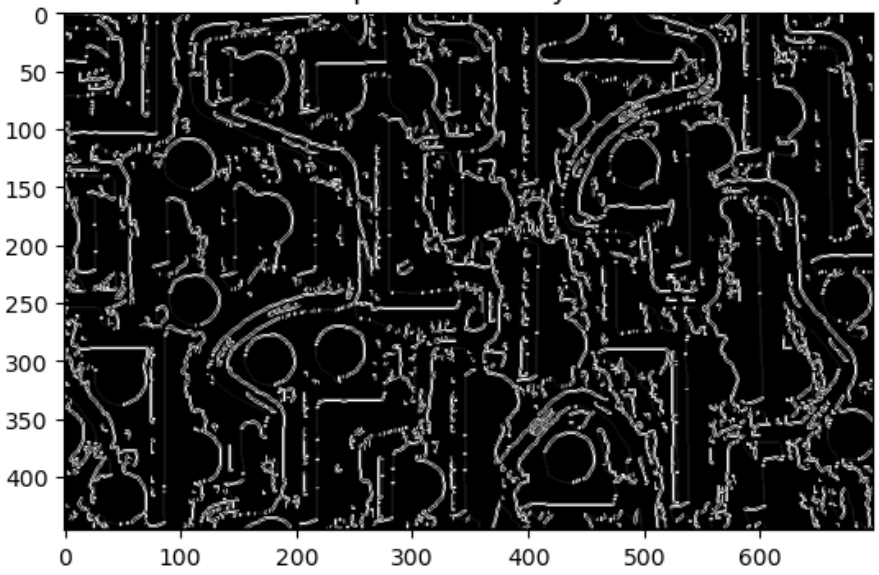




Artificial pattern - Original

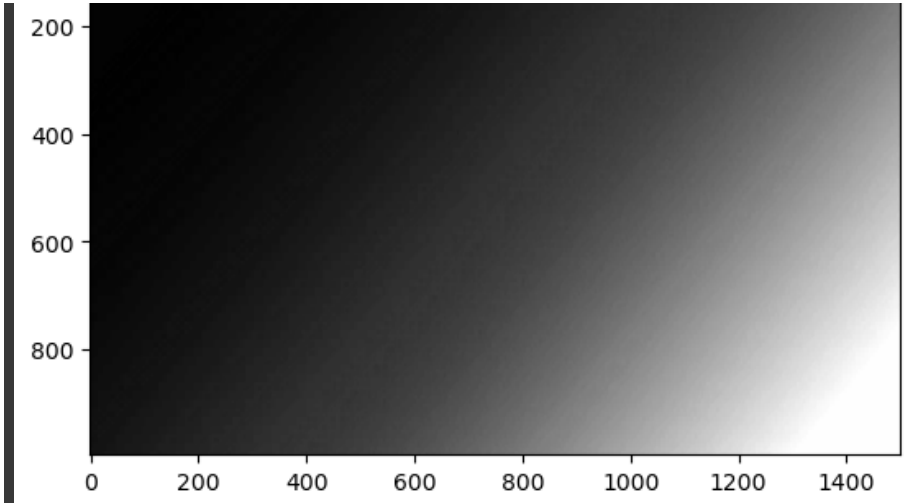


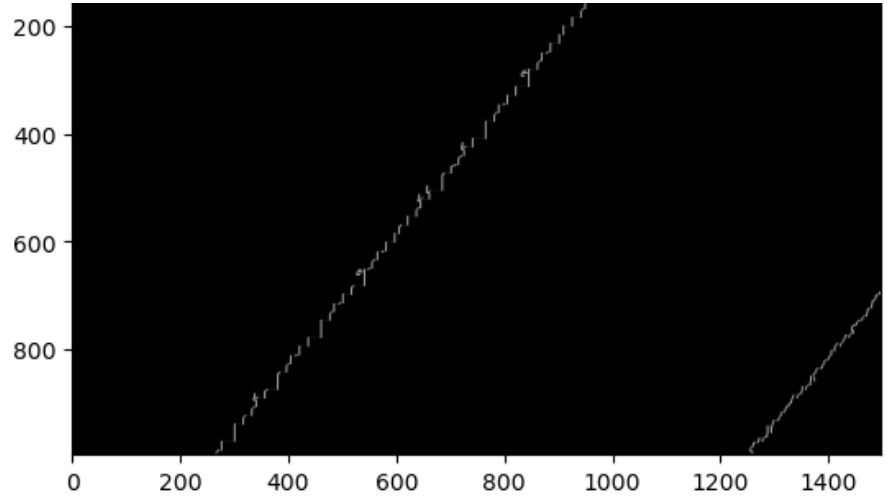
Artificial pattern - Canny detection



Gradient - Original

Gradient - Canny detection


200
400
600
800
0
200
400
600
800
1000
1200
1400


200
400
600
800
0
200
400
600
800
1000
1200
1400

Experimenting with various types of images helps us understand the effectiveness of the Canny algorithm in different scenarios.

Simple drawings provide a clear indication of the algorithm's ability to detect distinct edges accurately. Natural scenes reveal its performance with complex textures and irregular edges commonly found in real-world images. Textured patterns demonstrate how well it handles repetitive structures, while artificial patterns showcase its capability to identify edges in structured environments. Lastly, gradient images highlight its proficiency in detecting smooth transitions between different regions of varying intensity.