Marco Galindo ID: 310540013
2021, December 10th Image Processing

Programming Assignment #2

Selected task

The task selected for this programming assignment was the number one, corresponding to the implementation of the Canny edge detector algorithm.

Introduction

Humans are generally great and fast at image classification tasks, being able to accurately recognize what is inside an image. This process is so good, that images do not need to be highly detailed for a human to know what they are. A big percentage of what humas use to identify things are their edges. We will define an edge inside an image as the separation between segments or objects. Edge detection algorithms give a simpler representation of the image, by just showing the most important edges inside it. The purpose of the assignment is to explore and implement the Canny edge detector, one of the most important and popular edge detection algorithms, given its performance and an implementation that is not too complicated.

Objectives

The main objectives of this assignment are the following:

- Implement the Canny edge detector algorithm that was discussed in the lectures.
- Tune the Canny edge detector to work on multiple pictures showing very different objects.
- Research some advancements made to the algorithm over time and explore their contributions to the original idea.

Review of the methods used

Canny edge detector

The Canny edge detector, further referred just as "Canny", is one of the best edge detector algorithms out there. The approach of the algorithm is to accomplish the following three objectives:

1. Low error rate. All edges should be found.
2. Edge points should all be well localized. The edges from the algorithm need to be as close as possible to the true edge in the image.
3. Single edge point response. The output edges must be one-pixel wide.

To achieve these three tasks, Canny follows a series of steps that involve some pre-processing of the image, the detection of preliminary edges and the refinement of these edges to get a very good edge detection result.

It is important to note that this algorithm does not work with color images, so, the first true step required by Canny is to convert the original image to a grayscale version of it.

The following four steps of the algorithm are detailed below:

1. Image smoothing

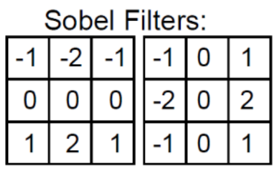
One of the downsides of this edge detection process is that it is very sensitive to noise in the image, even the slightest noise will yield a bad result, with many edges that correspond to noise and not real edges in the image. To counter this downside, a smoothing filter is applied to the original image; usually the applied algorithm is a Gaussian smoothing filter.

The Gaussian smoothing filter is a 2-D convolutional filter that blurs the original image to remove some details and noise. The resulting value of each pixel will be defined by a weighted sum of pixels in its neighborhood. The resulting formula can be defined as:

$$g(x,y) = \sum_i \sum_j \frac{1}{2\pi\sigma^2} e^{-\frac{x_i^2 + y_j^2}{2\sigma^2}}$$

2. Calculate gradient magnitude and direction

After smoothing the image, the gradient magnitude and direction for each pixel in the image is calculated by using the Sobel filters for example. The Sobel horizontal and vertical filters can be seen in Figure 1.


Sobel Filters:

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Figure 1 Sobel filters for vertical and horizontal edge gradients.

3. Non-maxima suppression

With the gradient magnitudes and directions from the Sobel filters, a non-maxima suppression is applied to only keep the strongest pixels. The process consists of scanning each pixel and if the pixel's value is lower than any of its neighbors in the direction of the gradient, the pixel's value becomes zero (see Figure 2).

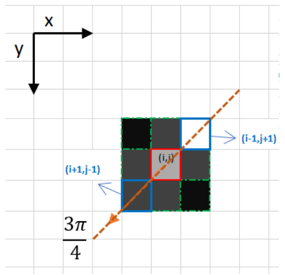


Figure 2 Example of non-maxima suppression. The red line indicates the direction of the gradient and the target pixel in this window corresponds to that located in (i,j). Because the pixel in (i,j) has a lower value than the pixel in (i-1,j+1), the pixel in (i,j) becomes zero.

4. Double thresholding

The next step consists of defining a set of strong and weak edge points. For this task, two thresholds are decided, T_1 and T_2. If a pixel's value is above T_1, it is defined as a strong edge and its value is converted to 255 (the highest value of a white pixel in an image). If the pixel's value is below T_2, it is classified as irrelevant and it becomes 0 (a black pixel). However, if a pixel's value is between $T_1$and T_2, the pixel is classified as "weak" and it is processed on the next step.

5. Edge Tracking by Hysteresis

This process is very useful for the overall result of the algorithm. Its objective is to decide whether to keep or discard the weak pixels identified during the double thresholding. The edge tracking by hysteresis consists in turning weak points to strong points only if they have other strong points in their vicinity, otherwise, they get discarded, meaning their values become 0. The result of this process helps join strong pixels and remove undesired gaps in lines.

<u>Improved Canny edge detector algorithm</u>

One of the main problems of the original Canny algorithm is its high sensitivity to noise, and that it involves two thresholds that must be manually changed for each different picture, making the algorithm a little annoying to work with. In response to these limitations, W. Rong et. Al proposed an improvement to the base Canny algorithm that involves two modifications.

The first difference is that this new algorithm replaces the image gradient by using a gravitational field of intensity as they call it. The idea is based on Newton's law of universal gravity, and applying its formula to the current setting, ultimately resulted in the creation of the gravitational field intensity operator seen in Figure 3.

$$G_x = \begin{pmatrix} -\frac{\sqrt{2}}{4} & 0 & \frac{\sqrt{2}}{4} \\ -1 & 0 & 1 \\ -\frac{\sqrt{2}}{4} & 0 & \frac{\sqrt{2}}{4} \end{pmatrix} \quad G_y = \begin{pmatrix} \frac{\sqrt{2}}{4} & 1 & \frac{\sqrt{2}}{4} \\ 0 & 0 & 0 \\ -\frac{\sqrt{2}}{4} & -1 & -\frac{\sqrt{2}}{4} \end{pmatrix}$$

Figure 3 Gravitational field intensity horizontal and vertical operators

The second difference is that by some equations using the total gravitational field intensity of the pixels in the image, the algorithm can automatically define the two thresholds for the double-thresholding process. However, it replaces the need to manually set the two thresholds with the need to manually define a parameter *k*, that ultimately defines the level of detail the result retains after the double-thresholding. Even if there is still a parameter to be set, it is easier to experiment with this parameter instead of modifying the two thresholds in the original algorithm.

Explanation of the experiments done

- **Original Canny edge detection algorithm**

The implemented algorithm was tested on different pictures, each taken under different settings and needing a different tuning of the model parameters to give the best result. The output of each step detailed in the previous section can be observed for the pictures. Figure 5 and Figure 6 show the final output of the Canny algorithm, as well as the output from the different stages throughout the process applied to the image shown in Figure 4.

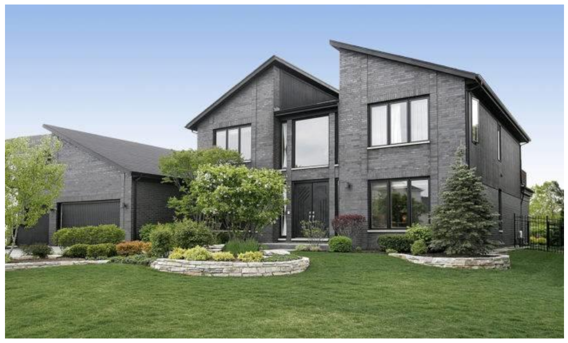


Figure 4 Original source image for Canny algorithm plotting of steps

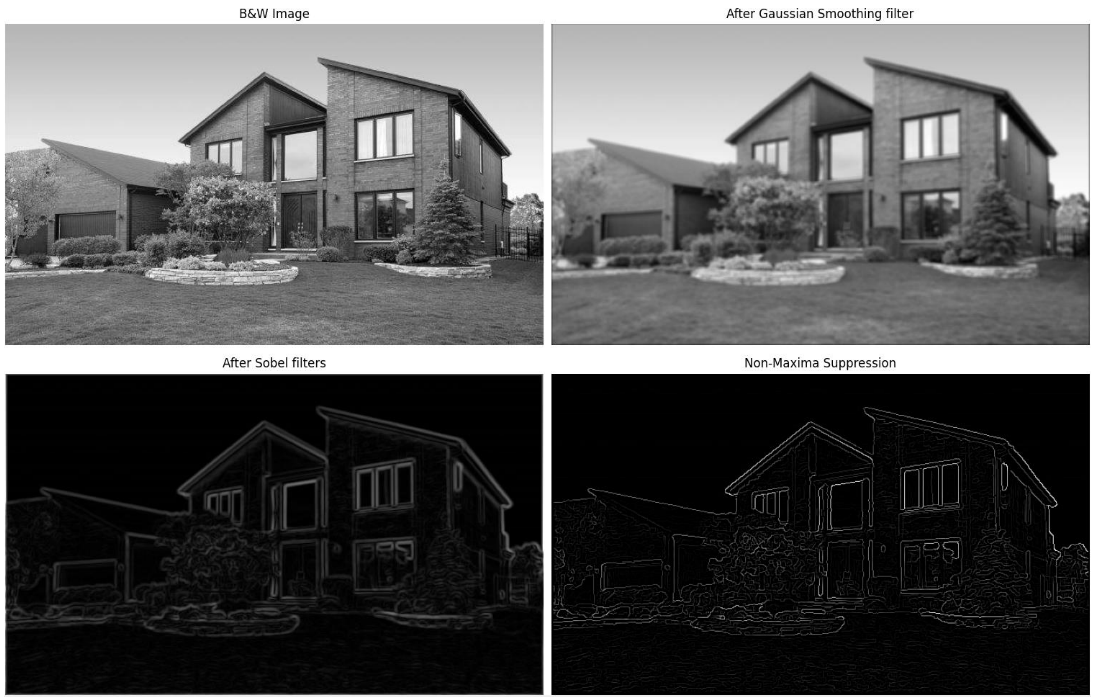


Figure 5 From left to right, top to bottom: (a) B&W version of the original image. (b) Image after a 5x5 Gaussian smoothing filter is applied. (c) Image after the application of Sobel filters. (d) Non-maxima suppression result.

In Figure 5(c) we can see that the edges are highlighted, but we can observe that they are not one-pixel wide, but they appear blurry. The non-maxima suppression solves this issue as seen on Figure 5(d), where edges appear better and thinner. However, pixels in this final image are not yet as defined as the algorithm wants them to be.

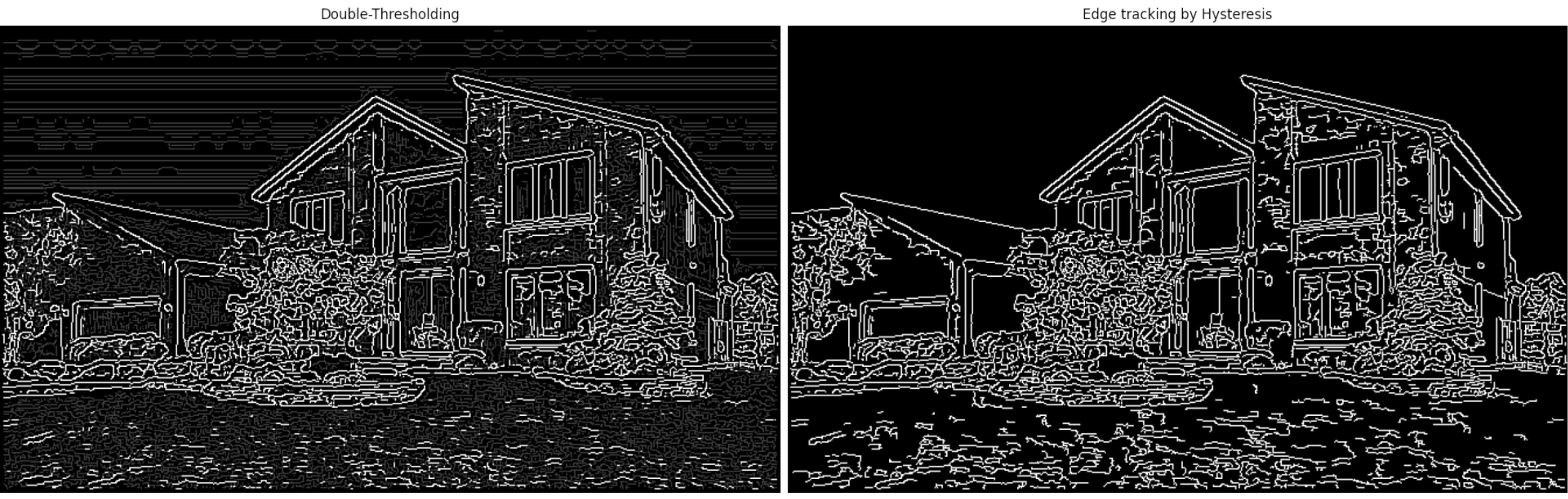


Figure 6 From left to right: (a) Double-thresholding result (b) Final result after Edge tracking by Hysteresis. For easy visualization, the image here has enhanced brightness.

The pixels in Figure 6(a) only have three possible values, 0 for irrelevant pixels, 25 for weak pixels and 255 for strong pixels. We can observe a lot of weak pixels, mostly around the sky area and the grass below. Most, if not all, of the sky weak pixels are gone in Figure 6(b), but unfortunately, a lot of weak pixels that appear in the grass section are converted to strong pixels, due to the noise in the area. The grass section of the image would ideally be clear of edges since they do not actually appear on the original image. Moreover, a lot of noise can be observed in the trees and walls of the image.

- **Improved Canny edge detection algorithm**

The improved Canny edge algorithm was applied to the same image shown in Figure 4 and later compared with the base Canny algorithm. The result is seen in Figure 7.

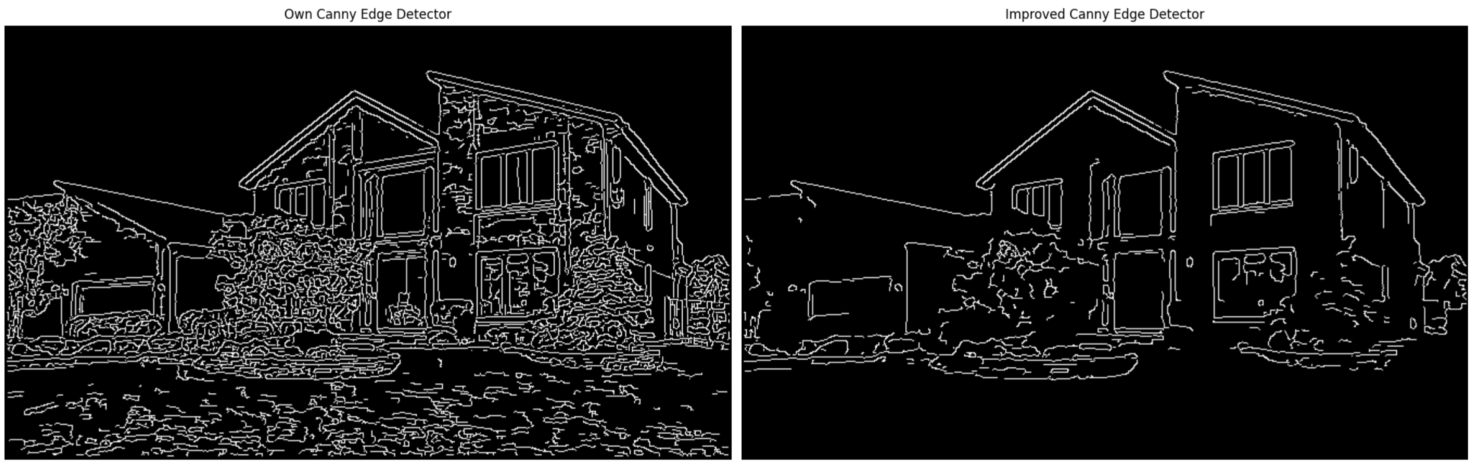


Figure 7 Comparison from base Canny and improved Canny edge detector

The changes made in the improved algorithm manage to capture only the essential information inside the picture, removing all noise present in the grass and walls from the original result from the base Canny algorithm. The value of *k* for this image was 1.1. The paper that proposes the improvement of the Canny algorithm mentions that their *k* value is between 1.2 and 1.6, but the usage of 1.1 specifically benefited this image, so it was chosen.

- The use of a sharpening filter

In this set of experiments, a Laplacian filter was applied to the image before being processed by the Canny detector, the sharpened image can be seen in Figure 8. The Laplacian sharpening filter highlights edges in the image, but at the same time, it also highlights the noise, so it makes the result of the Canny algorithm worse (Figure 9). It is interesting to note that even if the sharpening of the original image is not too intense, the result of the Canny algorithm is very bad, compared to the result previously shown in Figure 7.

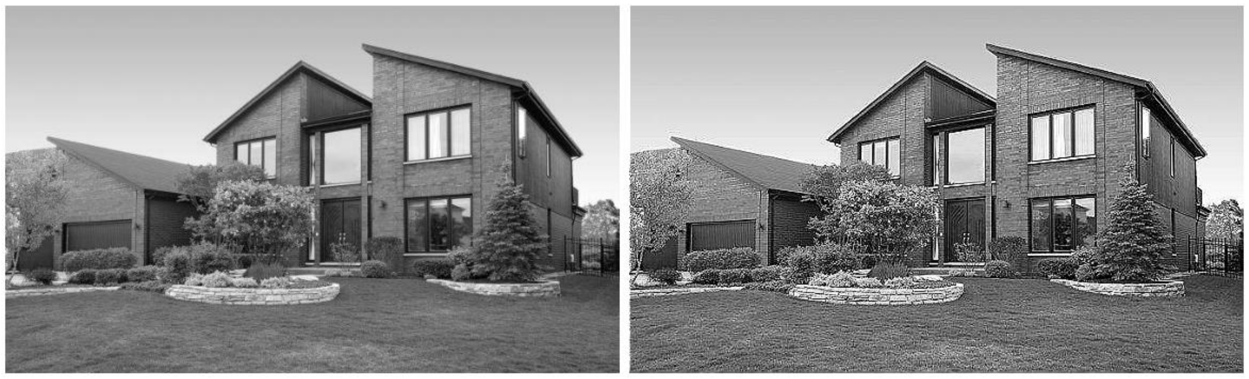


Figure 8 Original and sharpened image

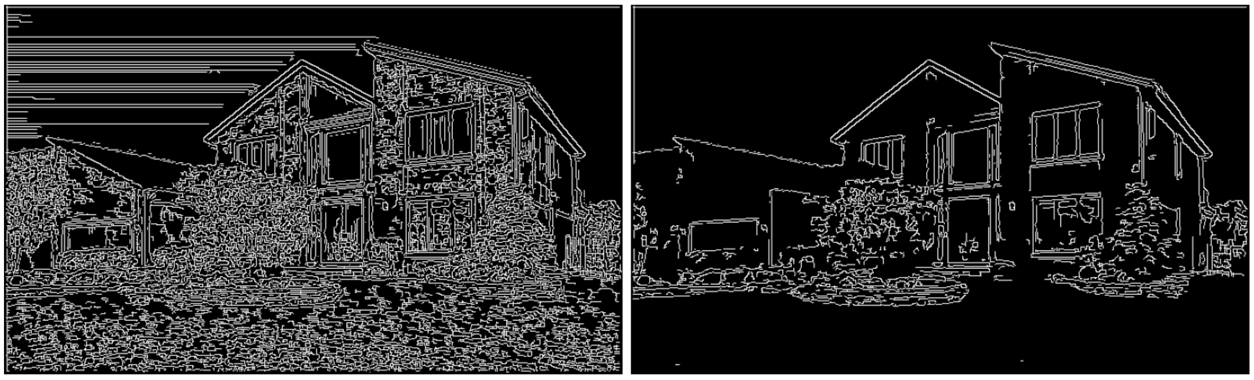


Figure 9 Result of the base Canny and improved Canny on the sharpened image

- More results and comparisons

The images in Figure 11, Figure 12, Figure 13 and Figure 14 were also compared with the results given by the Canny edge detector provided by the open-source library OpenCV.

During all these examples, because of the convenience of the improved Canny algorithm's manual parameter *k*, its value was manually adjusted to fit each specific image, while the parameters in the base Canny and the OpenCV algorithms were the same for all examples, due to their specific and more complex nature, requiring more time and tests to fine tune.

As an example of the effects of the *k* parameter, in Figure 10 we can see how the value of the k parameter affects an image's result.

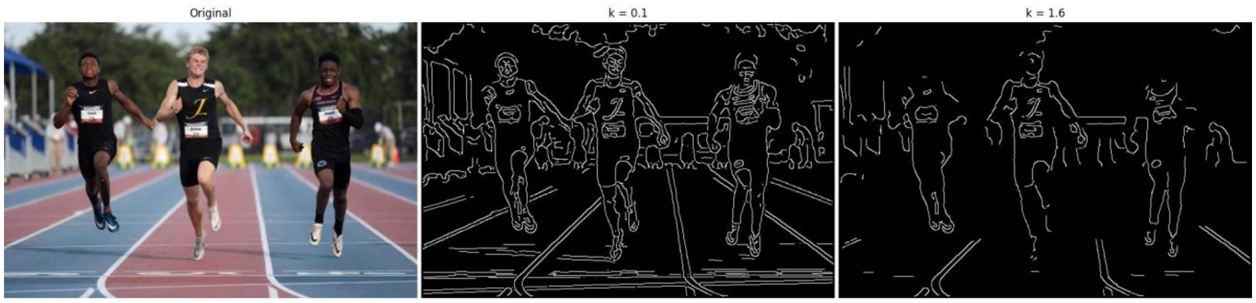


Figure 10 An image's result of the improved Canny with different k values

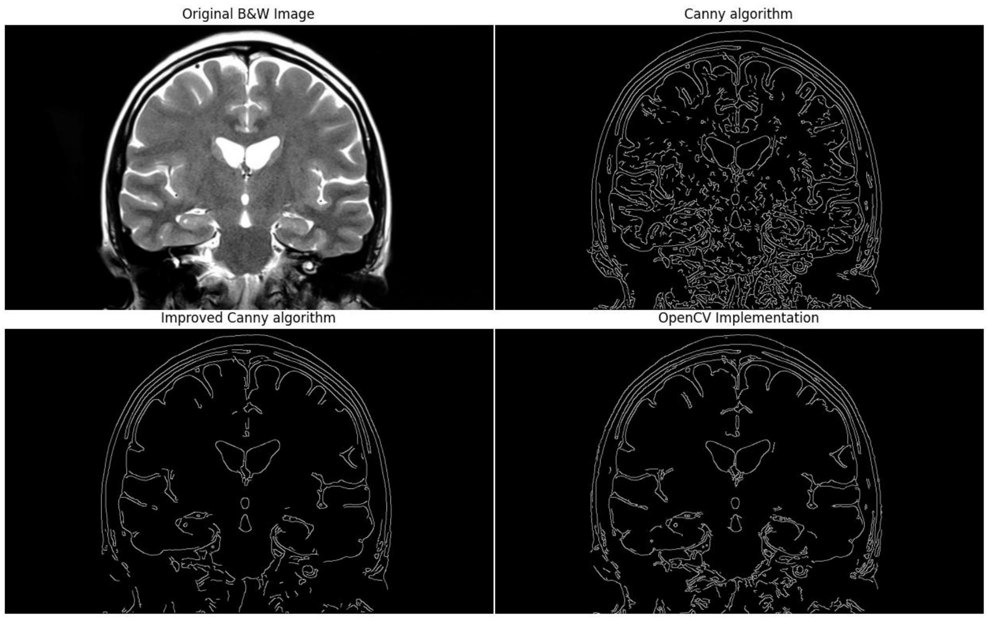


Figure 11 Example where both, the improved version and the OpenCV implementation, have similar and good results.

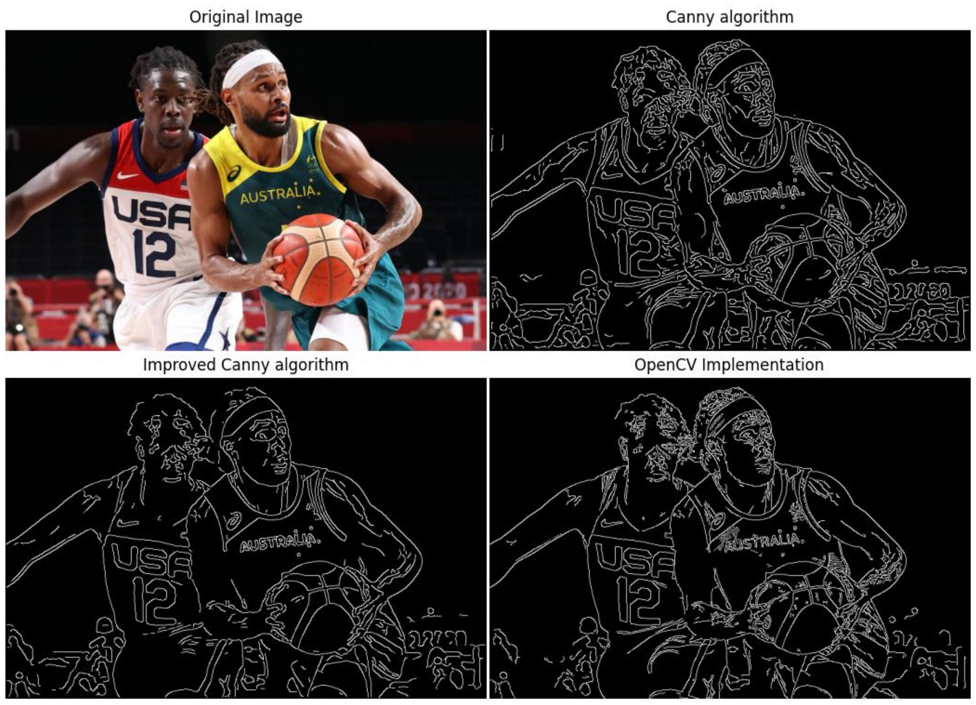


Figure 12 Example where all three results are good, with the improved version having a cleaner detection of the edges in the shirts of the players.

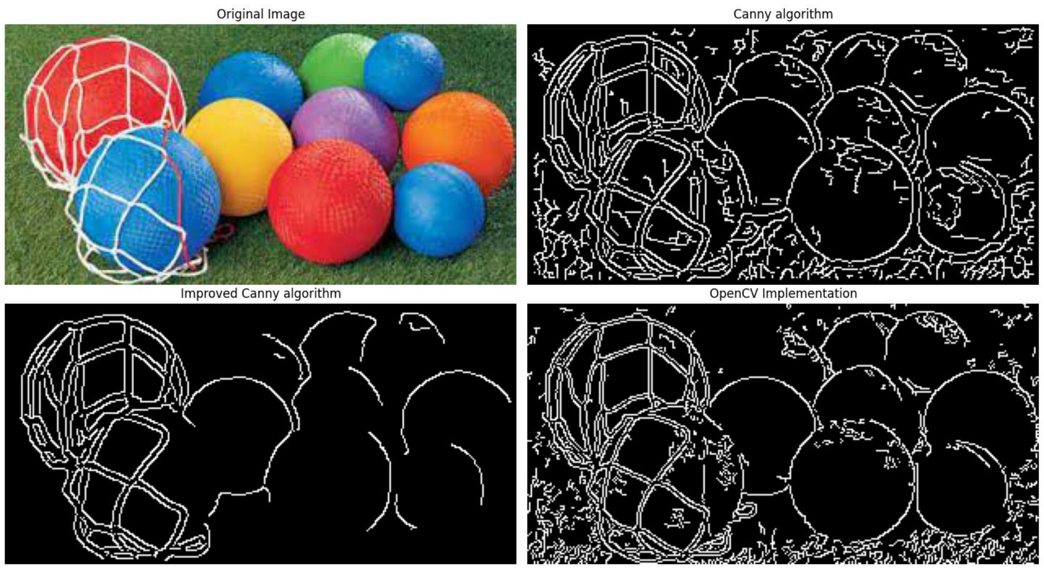


Figure 13 Example where even if the improved algorithm removes all the background noise, it also misses edges around some balls.

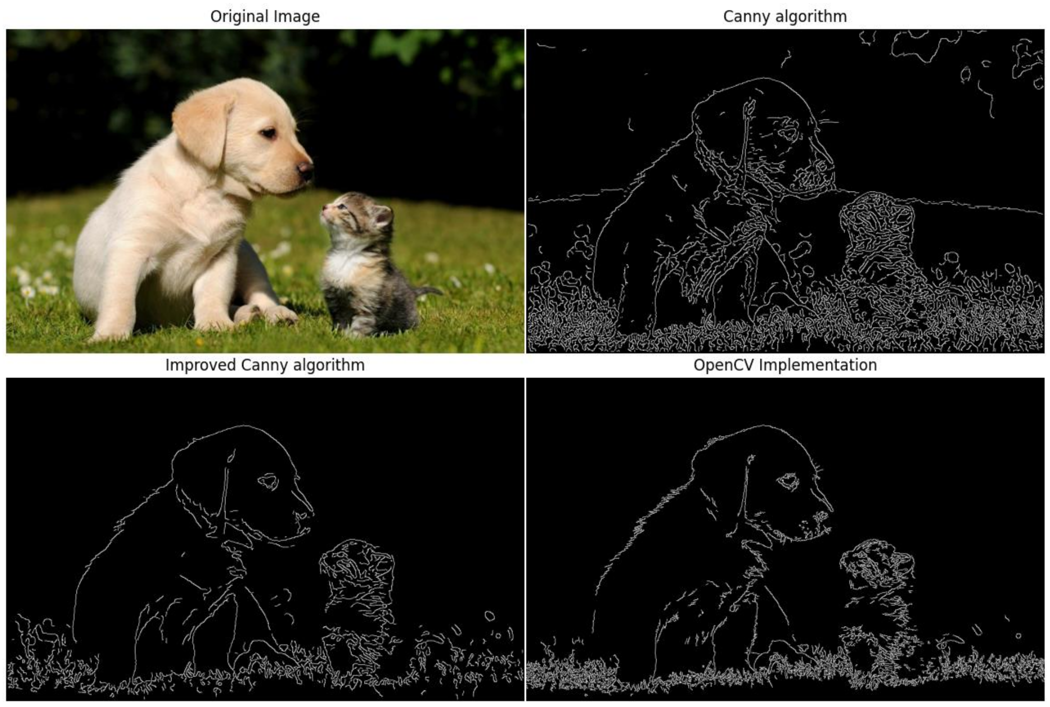


Figure 14 Example where the base algorithm's result has a lot of noise around the cat's face, while the improved and OpenCV implementations show a very good result.

Other improvements not implemented in this homework

As this algorithm is very popular, it has several other improvements that can be implemented, but their software implementation and fully understanding requires the knowledge of more complex details that were beyond the scope of this homework. However, the basic idea of some of these works can be discussed in this section.

- R. Biswas et. Al incorporated theory from Type-2 Fuzzy sets into the algorithm to try and define the threshold automatically. They state that when an image is low-quality or has some issues, defining a clear boundary by looking at the intensity histogram can be complicated and that is why they choose to improve this section of the algorithm. They ultimately change the use of two thresholds to the use of a single and automatically calculated threshold that defines if the pixel should be discarded or kept.
- Y. Feng et. Al decided to tackle the noise sensitivity of the base Canny by proposing an algorithm that is inspired by the base Canny. First, they modify the pre-processing of the image, applying a median filter to remove noise instead of a Gaussian filter. Second, using filters based on Euclidean distance, instead of filters based on the image's gradient (e.g., Sobel filters). Finally, they make the use of other edge detection algorithms to finalize their result, the Frei-Chen algorithm and the Otsu algorithm.

Discussions

The results obtained from the Canny edge detector are good, but the algorithm does not make it simple to adjust its parameters to fit each image. For this reason, the usage of an improved algorithm is highly encouraged, to save time in the fine tuning of the algorithm and get great results.

It is also interesting to note the effect of the Laplacian sharpening filter, that even when the original image seems almost unaffected to the human eye, the Canny algorithm still is affected a lot by the changes made to the image, even around areas that do not seem to have any edge on the original image. Like the sky on Figure 9.

Main References

- Yingke Feng, Jinmin Zhang, and Siming Wang , "A new edge detection algorithm based on Canny idea", AIP Conference Proceedings 1890, 040011 (2017) https://doi.org/10.1063/1.5005213
- Ranita Biswas, & Jaya Sil (2012). An Improved Canny Edge Detection Algorithm Based on Type-2 Fuzzy Sets*. Procedia Technology, 4, 820-824.*
- W. Rong, Z. Li, W. Zhang and L. Sun, "An improved Canny edge detection algorithm," *2014 IEEE International Conference on Mechatronics and Automation*, 2014, pp. 577-582, doi: 10.1109/ICMA.2014.6885761.
- Gonzalez, R. C. (2018). *Digital Image Processing* (pp. 716-737). Pearson.
- Sofiane Sahir, Canny Edge Detection Step by Step in Python — Computer Vision, 2019. https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123. (Accessed December 2021).

Code

```python
import matplotlib.pyplot as plt
import numpy as np
import math
import cv2
import os


def laplacian(image, filter_config, filter_size=3):
    """Method to apply a sharpening Laplacian filter to the image."""
    image_array = np.array(image)
    resulting_array = np.zeros(image_array.shape)
    image_array = np.pad(image_array, 1, mode='constant')
    X_STEP = filter_size
    Y_STEP = filter_size
    height, width = resulting_array.shape

    # The filter config contains the matrix definition for the filter
    laplacian_filter = -1 * np.array(filter_config).reshape((filter_size, filter_size))
    for Xo in range(height):
        for Yo in range(width):
            Xf = Xo + X_STEP
            Yf = Yo + Y_STEP
            if Yf > width:
                continue
            if Xf > height:
                continue
            region = image_array[Xo:Xf, Yo:Yf]

            # Apply the filter
            resulting_array[Xo, Yo] = np.sum(np.multiply(region, laplacian_filter))
    sharpened_image = (image_array[1:-1, 1:-1] + resulting_array)

    # Return the sharpenend image and the laplacian mask used
    return sharpened_image


def get_kernel(size, sigma):
    """ Get the filter with the Gaussian formula applied to its original values. """

    # Creating a vector of the desired size and evenly spaced
    kernel = np.linspace(-(size // 2), size // 2, size)

    # Calculate the gaussian for each vector element
    for i in range(size):
        kernel[i] = 1 / (np.sqrt(2 * np.pi) * sigma) * np.e ** (-np.power((kernel[i]) / sigma, 2)
/ 2)

    # Transform the vector into a matrix, to use in in the convolution process
    kernel = np.outer(kernel.T, kernel.T)

    # Normalizing the kernel
    kernel *= 1.0 / kernel.max()
    return kernel


def gaussian_blur(image, filter_size, color=True):
    """ Perform Gaussian Blur on an image. image_array = GRAY image's array"""
    kernel = get_kernel(filter_size, math.sqrt(filter_size))
    image_array = np.array(image)

    if color:
        # For color images, perform the process on the value channel of an HSV image
        height, width, _ = image_array.shape
        X_STEP, Y_STEP = kernel.shape

        resulting_array = np.zeros(image_array.shape)
        resulting_array[:, :, 0] = image_array[:, :, 0]
        resulting_array[:, :, 1] = image_array[:, :, 1]
        pad_height = int((X_STEP - 1) / 2)
        pad_width = int((Y_STEP - 1) / 2)

        padded_image = np.zeros((height + (2 * pad_height), width + (2 * pad_width)))
        padded_image[pad_height:padded_image.shape[0] - pad_height,
        pad_width:padded_image.shape[1] - pad_width] = image_array[:, :, 2]

        # Perfom the convolutions
        for Xo in range(height):
            for Yo in range(width):
                Xf = Xo + X_STEP
                Yf = Yo + Y_STEP
                resulting_array[Xo, Yo, 2] = np.sum(kernel * padded_image[Xo:Xf, Yo:Yf])
        resulting_array[:, :, 2] = resulting_array[:, :, 2] * 255 / np.max(resulting_array[:, :,
2])

        return resulting_array
    else:
        # For B&W images

        height, width = image_array.shape
        X_STEP, Y_STEP = kernel.shape

        resulting_array = np.zeros(image_array.shape)

        pad_height = int((X_STEP - 1) / 2)
        pad_width = int((Y_STEP - 1) / 2)

        padded_image = np.zeros((height + (2 * pad_height), width + (2 * pad_width)))
        padded_image[pad_height:padded_image.shape[0] - pad_height,
        pad_width:padded_image.shape[1] - pad_width] = image_array

        # Perfom the convolutions
        for Xo in range(height):
            for Yo in range(width):
                Xf = Xo + X_STEP
                Yf = Yo + Y_STEP
                resulting_array[Xo, Yo] = np.sum(kernel * padded_image[Xo:Xf, Yo:Yf])
        resulting_array = resulting_array * 255 / np.max(resulting_array)
        return resulting_array


def sobel_filters(image_array):
    """Apply horizontal and vertical Sobel filters on the image array provided."""
    height, width = image_array.shape
    X_STEP = 3
    Y_STEP = 3

    # Definition of the filters
    Kx = np.array([-1, 0, 1, -2, 0, 2, -1, 0, 1], np.float32).reshape(3, 3)
    Ky = np.array([1, 2, 1, 0, 0, 0, -1, -2, -1], np.float32).reshape(3, 3)

    # The resulting image's array
    resulting_array = np.zeros((height, width, 2))

    # Pad the image with zeros
    pad_height = int((X_STEP - 1) / 2)
    pad_width = int((Y_STEP - 1) / 2)
    padded_image = np.zeros((height + (2 * pad_height), width + (2 * pad_width)))
    padded_image[pad_height:padded_image.shape[0] - pad_height,
    pad_width:padded_image.shape[1] - pad_width] = image_array

    # Apply filters
    for Xo in range(height):
        for Yo in range(width):
            Xf = Xo + X_STEP
            Yf = Yo + Y_STEP
            resulting_array[Xo, Yo, 0] = np.sum(Kx * padded_image[Xo:Xf, Yo:Yf])  # Horizontal
            resulting_array[Xo, Yo, 1] = np.sum(Ky * padded_image[Xo:Xf, Yo:Yf])  # Vertical

    # Get the gradient directions and magnitude
    gradient_directions = np.hypot(resulting_array[:, :, 0], resulting_array[:, :, 1])
    gradient_directions = gradient_directions / gradient_directions.max() * 255
    theta = np.arctan2(resulting_array[:, :, 1], resulting_array[:, :, 0])
    return (gradient_directions, theta)

def gravitational_filters(image_array):
     """Apply the gravitational filters of the improvede Canny algorithm"""
    height, width = image_array.shape
    X_STEP = 3
    Y_STEP = 3

    # Gravitational intensity operators
    Kx = np.array([-(2**(1/2))/4, 0, (2**(1/2))/4, -1, 0, 1, -(2**(1/2))/4, 0, (2**(1/2))/4],
np.float32).reshape(3, 3)
    Ky = np.array([(2**(1/2))/4, 1, (2**(1/2))/4, 0, 0, 0, -(2**(1/2))/4, -1, -(2**(1/2))/4],
np.float32).reshape(3, 3)

    resulting_array = np.zeros((height, width, 2))
    # Pad the image
    pad_height = int((X_STEP - 1) / 2)
    pad_width = int((Y_STEP - 1) / 2)

    padded_image = np.zeros((height + (2 * pad_height), width + (2 * pad_width)))
    padded_image[pad_height:padded_image.shape[0] - pad_height,
    pad_width:padded_image.shape[1] - pad_width] = image_array

    for Xo in range(height):
        for Yo in range(width):
            Xf = Xo + X_STEP
            Yf = Yo + Y_STEP
            resulting_array[Xo, Yo, 0] = np.sum(Kx * padded_image[Xo:Xf, Yo:Yf])  # Horizontal
            resulting_array[Xo, Yo, 1] = np.sum(Ky * padded_image[Xo:Xf, Yo:Yf])  # Vertical

    # Get gradient directions and magnitude
    gradient_directions = np.hypot(resulting_array[:, :, 0], resulting_array[:, :, 1])
    gradient_directions = gradient_directions / gradient_directions.max() * 255
    theta = np.arctan2(resulting_array[:, :, 1], resulting_array[:, :, 0])
    return (gradient_directions, theta)


def non_maxima_supression(image_array, gradient_directions):
    # Apply the non-maxima suprresion on the image's array with the help from the gradient
directions
    height, width = image_array.shape
    resulting_array = np.zeros((height, width))

    # Convert the directions to degrees, and flip negative values
    gradient_directions = gradient_directions * 180 / np.pi
    gradient_directions[gradient_directions < 0] += 180
    for X in range(1, height - 1):
        for Y in range(1, width - 1):
            resulting_array[X, Y] = image_array[X, Y]
            direction = gradient_directions[X, Y]
            #Compare intensities and keep only the strongest pixels

            # angle  0
            if (0 <= direction < 22.5) or (157.5 <= direction <= 180):
                pixel_after = image_array[X, Y + 1]
                pixel_before = image_array[X, Y - 1]
            # angle 45
            elif (22.5 <= direction < 67.5):
                pixel_after = image_array[X + 1, Y - 1]
                pixel_before = image_array[X - 1, Y + 1]
            # angle 90
            elif (67.5 <= direction < 112.5):
                pixel_after = image_array[X + 1, Y]
                pixel_before = image_array[X - 1, Y]
            # angle 135
            elif (112.5 <= direction < 157.5):
                pixel_after = image_array[X - 1, Y - 1]
                pixel_before = image_array[X + 1, Y + 1]

            if (image_array[X, Y] >= pixel_after) and (image_array[X, Y] >= pixel_before):
                resulting_array[X, Y] = image_array[X, Y]
            else:
                resulting_array[X, Y] = 0

    return resulting_array
```

```
def threshold(img, ratio = True, lowThresholdRatio=0.05, highThresholdRatio=0.09):
    # Perform the double threhsolding
    if ratio: # The parameters' values define a percentage of detail to keep
        highThreshold = img.max() * highThresholdRatio
        lowThreshold = highThreshold * lowThresholdRatio
    else: # The parameters' values give absolute intensity values
        highThreshold = highThresholdRatio
        lowThreshold = lowThresholdRatio
    result = np.zeros(img.shape)

    # Define strong and weak pixels
    strong_i, strong_j = np.where(img >= highThreshold)
    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThreshold))

    weak_value = 25
    strong_value = 255

    # Change the intensity of the strong and weak pixels identified
    result[strong_i, strong_j] = strong_value
    result[weak_i, weak_j] = weak_value

    return (result, weak_value, strong_value)

def hysteresis(image, weak=25, strong=255):
     # Transform weak pixels into strong pixels or discard them
    img = image.copy()
    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

def using_improved(image_original, k = 1.3, plots = True):
     # Process the image with the improved version of the Canny edge detector

     # Gaussian smoothing filter
    image_gaussian = gaussian_blur(image_original, 5, False)

    # Gravitational filters
    gravitational_image, gradient_directions = gravitational_filters(np.array(image_gaussian))

    # Non-maxima suppression
    non_maxima_img = non_maxima_supression(gravitational_image, gradient_directions)

    width, height = image_original.shape

    # Get Eave and calculate sigma as defined by the improved Canny paper
    Eave = gravitational_image.sum()/(width*height)
    sigma = 0
    for i in range(width):
        for j in range(height):
            sigma += (gravitational_image[i,j]-Eave)**2
    sigma = (sigma / (width*height))**(1/2)
    # Define the two thresholds for the image
    highThresholdRatio = Eave + k * sigma
    lowThresholdRatio = highThresholdRatio / 2

    # Get the result
    double_threshold_img, weak_value, strong_value = threshold(non_maxima_img, ratio=False, 
lowThresholdRatio=lowThresholdRatio, highThresholdRatio=highThresholdRatio)
    resulting_img = hysteresis(double_threshold_img, weak_value, strong_value)

    # Get the plots of the process
    if plots:
        fig = plt.figure(figsize=(15, 15))
        rows = 2
        columns = 2

        fig.add_subplot(rows, columns, 1)
        plt.imshow(image_original, cmap='gray')
        plt.axis('off')
        plt.title("Original")

        fig.add_subplot(rows, columns, 2)
        plt.imshow(image_gaussian, cmap='gray')
        plt.axis('off')
        plt.title("Gaussian")

        fig.add_subplot(rows, columns, 3)
        plt.imshow(gravitational_image, cmap='gray')
        plt.axis('off')
        plt.title("gravitational_image")

        fig.add_subplot(rows, columns, 4)
        plt.imshow(non_maxima_img, cmap='gray')
        plt.axis('off')
        plt.title("non_maxima_img")

        plt.show()

        fig = plt.figure(figsize=(15, 15))
        rows = 1
        columns = 1

        fig.add_subplot(rows, columns, 1)
        plt.imshow(resulting_img, cmap='gray')
        plt.axis('off')
        plt.title("Non Maxima")

        plt.show()

    return resulting_img

def using_own(image_original, plots=False):
     # Process the image with the original base Canny edge detector algorithm

     # Gaussian smoothing
    image_gaussian = gaussian_blur(image_original, 5, False)

    # Sobel filters
    sobel_image, gradient_directions = sobel_filters(np.array(image_gaussian))

    # Non-maxima suppression
    non_maxima_img = non_maxima_supression(sobel_image, gradient_directions)

    # Double thresholding
    double_threshold_img, weak_value, strong_value = threshold(non_maxima_img)

    # Hysteresis process
    resulting_img = hysteresis(double_threshold_img, weak_value, strong_value)

    # Get process' plots
    if plots:
        fig = plt.figure(figsize=(15, 15))
        rows = 2
        columns = 2

        fig.add_subplot(rows, columns, 1)
        plt.imshow(image_original, cmap='gray', vmin=0, vmax=255)
        plt.axis('off')
        plt.title("B&W Image")

        fig.add_subplot(rows, columns, 2)
        plt.imshow(image_gaussian, cmap='gray')
        plt.axis('off')
        plt.title("After Gaussian Smoothing filter")

        fig.add_subplot(rows, columns, 3)
        plt.imshow(sobel_image, cmap='gray')
        plt.axis('off')
        plt.title("After Sobel filters")

        fig.add_subplot(rows, columns, 4)
        plt.imshow(non_maxima_img, cmap='gray')
        plt.axis('off')
        plt.title("Non-Maxima Suppression")

        # plt.subplots_adjust(wspace=0, hspace=0)
        plt.show()

        fig = plt.figure(figsize=(15, 15))
        rows = 1
        columns = 2

        fig.add_subplot(rows, columns, 1)
        plt.imshow(double_threshold_img, cmap='gray')
        plt.axis('off')
        plt.title("Double-Thresholding")

        fig.add_subplot(rows, columns, 2)
        plt.imshow(resulting_img, cmap='gray')
        plt.axis('off')
        plt.title("Edge tracking by Hysteresis")

        plt.show()

    return resulting_img


def using_cv2(image_original, plots=False):
    #Proces the image with the open-source libray OpenCV2

    edges = cv2.Canny(image_original, 110, 210, False)

    if plots:
        plt.subplot(121), plt.imshow(image_original, cmap='gray')
        plt.title('Original Image'), plt.xticks([]), plt.yticks([])
        plt.subplot(122), plt.imshow(edges, cmap='gray')
        plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
        plt.show()
    return edges


def compare(figure1, figure2, text1, text2):
    # Plot two figures side by side
    fig = plt.figure(figsize=(15, 15))
    rows = 1
    columns = 2

    fig.add_subplot(rows, columns, 1)
    plt.imshow(figure1, cmap='gray')
    plt.axis('off')
    plt.title(text1)

    fig.add_subplot(rows, columns, 2)
    plt.imshow(figure2, cmap='gray', vmin=0, vmax=255)
    plt.axis('off')
    plt.title(text2)

    plt.show()


# All the images to be tested, with their corresponding k value for the improved algorithm
images = np.array(["pic4PR2/brain_mri.jpg",1.6,
          "pic4PR2/basketball.jpg",0.7,
          "pic4PR2/balls.jpg",0.9,
          "pic4PR2/house.jpg",1.1,
          "pic4PR2/pets.jpg",1.3,
          "pic4PR2/toy_story.jpg",0.7]).reshape(6,2)

# Process all the images from the array
```

```
for image, k in images:
    image_originalColor = cv2.imread(os.path.join(os.path.dirname(__file__),
image)).astype('uint8')

    # Get B&W image
    image_original = cv2.cvtColor(image_originalColor, cv2.COLOR_BGR2GRAY)

    # Laplacian sharpening filter
    # sharpened_image = laplacian(image_original, [0, 1, 0, 1, -4, 1, 0, 1, 0])


    # Get different versions of the Canny algorithm
    own_canny = using_own(image_original, False)

    # sharpened_canny = using_own(sharpened_image, False)
    improved_canny = using_improved(image_original, k=float(k), plots=False)
    cv2_canny = using_cv2(image_original, False)

    # Plot results
    fig = plt.figure(figsize=(15, 15))
    rows = 2
    columns = 2

    fig.add_subplot(rows, columns, 1)
    plt.imshow(cv2.cvtColor(image_originalColor, cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.title("Original Image")

    fig.add_subplot(rows, columns, 2)
    plt.imshow(own_canny, cmap='gray', vmin=0, vmax=255)
    plt.axis('off')
    plt.title("Canny algorithm")
    fig.add_subplot(rows, columns, 3)
    plt.imshow(improved_canny, cmap='gray')
    plt.axis('off')
    plt.title("Improved Canny algorithm")

    fig.add_subplot(rows, columns, 4)
    plt.imshow(cv2_canny, cmap='gray')
    plt.axis('off')
    plt.title("OpenCV Implementation")

    plt.show()
```