MTE 203 Project 2: Canny Edge Detection

By: Nikunj Patel (#20957142)

Introduction

Canny Edge Detection is a major image processing method that is primarily used to find and detect edges in images. It was first developed by John Canny in 1986 and has been used widely due to its accuracy and reliability. The primary goal is to find cracks within buildings and bridges to ensure enhanced safety, risk reduction, and timely repairs. Early detection of cracks is essential to ensure timely maintenance and repair while also avoiding structural failures. Furthermore, a clear understanding of the Canny edge detection being utilized for crack detection will be shown through the stages such as Gaussian blurring, gradient computation, non-maximum suppression, double thresholding, and edge tracking by hysteresis. These stages will be critical to see how they play a role in detecting cracks from images of buildings and bridges. The focus of this study is to show how the use of Canny edge detection can help civil engineers enhance crack detection and promote safety.

Background

In image processing, most algorithms often face challenges with large images and data. With the use of edge detection techniques like the Canny edge detector it can help find the boundaries and sharp changes in intensity or color which are known as "edges" within an image to reduce data and focus on critical information to allow for a faster processing time [1]. Canny edge detection algorithm is a multi-stage algorithm used to detect/identify various edges inside an image.

Step 1: Smoothing using a Gaussian filter:

To start, the input image is converted to a grayscale image since edge detection doesn't rely on colors and then a Gaussian filter kernel is convolved with the image to reduce noise and smooth the image. The equation for a Gaussian filter kernel of size (2k+1) × (2k+1) is given by:

$$G_{xy} = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{\left(x-(k+1)\right)^2+\left(y-(k+1)\right)^2}{2\sigma^2}\right); \quad 1 \le x, y \le (2k+1)$$

where:

- x and y are the coordinates of the pixel in the kernel.
- $(k+1)$ is the size of the kernel.
- σ (sigma) is the standard derivation of the Gaussian, controlling the spread of the distribution.

The result after applying Gaussian filter will be a blurred/smooth version of the input image since doing so reduces noise while maintaining the critical features of the image [2]. This technique is used to also control the amount of detail that appears in the edge image.

Step 2: Finding Intensity Gradients:

The smoothened image is then filtered with a Sobel operator 3x3 kernel in both x and y directions of an image to compute the gradients. The Sobel x-operator detects the horizontal edges, while the Sobel y-operator detects the vertical edges. By performing convolution of these operators with the image, it highlights areas where there are major changes in intensity and brightness along with their directions [3] [4]. The Sobel x-operator is of the form:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A$$

where G_x represents the gradient intensity in the x-direction while being convolved with A which is the input image. The Sobel y-operator is of the form:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$$

where G_y represents the gradient intensity in the y-direction while being convolved with A which is the input image. After finding the gradients in both directions you can calculate the gradient magnitude which represents the strength of edges in the images. The gradient magnitude (G) is given by the equation:

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

To find the direction of edges at each pixel in the image the gradient orientation (θ) is computed which is given by the equation:

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

where:

- G_y is the Sobel y-operator
- G_x is the Sobel x-operator

The edge direction angle is rounded to one of the four angles demonstrating vertical, horizontal and the two diagonals ($0°, 45°, 90°, 135°$ degrees) for simplicity and ease of computation [3] [4].

Step 3: Non-Maximum Suppression:

After finding the gradient magnitude and direction, the entire image is scanned to remove any unwanted pixels that don't contribute to producing thin and well-defined edges. This process is known as non-maximum suppression, and it is done by checking each pixel's gradient magnitude and comparing it with its neighbors along the gradient orientation. If the magnitude at the current pixel is greater than both neighbors, it's known as a local maximum and stored as a potential edge or else it is suppressed and set to 0 [4] [5]. An example of this can be seen in Figure 1, Point A is on the edge in the vertical direction while the gradient direction is normal to the edge. Points B and C are in gradient direction which means that Point A is checked with B and C to verify if it's a local maximum. If so, it is considered for the next stage, or else it will be suppressed and given a value of 0 [5].

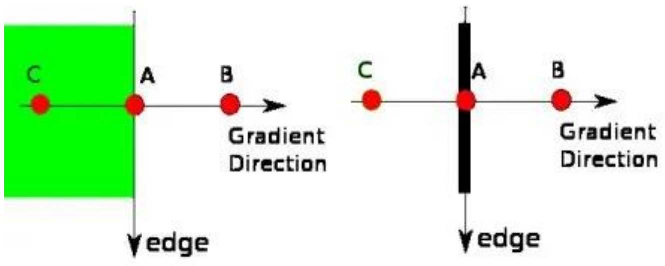


Figure 1: Performing non-max suppression on A, B, and C which are pixels (red dots). The green box represents a pixel of an image which was not suppressed since A is a local maximum which validates A, B and C as a potential edge [6].

Step 4: Double Threshold:

Upon removing the unwanted pixels, the image consists of a mix of main edges that are required and other edge pixels that were generated by either color variation or noise. Double threshold helps get rid of unwanted pixels by assigning the edge pixels as either strong edges or weak edges. The first step is for the user to pass two thresholds, lowerVal and upperVal. If an edge pixel's gradient value (G) is higher than the upperVal, it is classified as a strong edge pixel. If the edge pixel's gradient value (G) is less than the upperVal but greater than the lowerVal, it is classified as a weak edge pixel which is preserved for the next stage. Lastly, if the edge pixel's gradient value (G) is less than the lowerVal, it is suppressed and given a value of 0 [7] [8]. Any of the pixels that lie between the high and low threshold are only taken into consideration but if a weak edge is connected to a strong edge pixel it is kept [9]. In Figure 2, A is considered as a strong edge pixel since it's above the high threshold (upperVal) while B is considered as a weak edge pixel since it's below the high threshold (upperVal) but greater than the low threshold (loweVal). For this case, C will not be removed from the next stage but rather taken into consideration since it is connected to A which is a strong edge pixel.

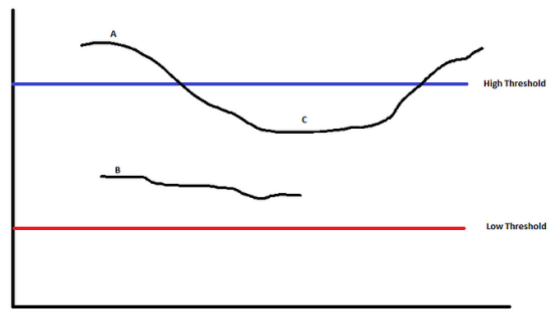


Figure 2: Double threshold being applied to pixels A, B and C. The blue line represents the high threshold while the red line represents the low threshold. Pixel A is a strong edge pixel since it's greater than the high threshold while Pixel B is a weak edge pixel. Pixel C is not removed but taken to next stage since its connected to Pixel A [9].

Step 5: Track edge by hysteresis:

As shown in Figure 2, the process of considering C since it's connected to a strong edge A is also known as Edge tracking by hysteresis. This algorithm allows you to verify if the weak edges that were preserved from the previous stage are actual edges or not to form continuous edges in the final output. Weak edges that are not connected to a strong edge will be removed while the edges that are connected to a strong edge will be shown in the final output [9] [10].

Approach

The intention of this section is to show that the use of Canny edge detection for identifying cracks in buildings and bridges to promote safety, timely maintenance and avoid structural failures.

Step 1: Approach
The first step is to gather a dataset of buildings and bridges with known cracks while making sure that it has variation in crack types, orientations and lighting.

Grayscale: Convert the input color images to grayscale.

Gaussian Smoothing: Apply a Gaussian filter kernel to the grayscale images to reduce noise and smooth the images for more accurate edge detection.

Sobel Operators: Calculate the gradient magnitude and direction of the smoothed image using Sobel operators in both the x and y directions.

Non-Maximum Suppression: Perform non-maximum suppression to thin out the detected edges and keep only the local maxima of gradient magnitude and the gradient direction.

Double Thresholding: Apply double thresholding to classify edge pixels as strong, weak, or non-edges based on their gradient magnitude values.

Edge Tracking by Hysteresis: Perform edge tracking by hysteresis to connect weak edges to strong edges and form continuous edges.

Final Output: After completing all the stages, the final image will be produced which would accurately show cracks on building and bridges.

Step 2: Methodology

Implement the Canny algorithm in Python using NumPy and matplotlib. Each of the steps mentioned above will be programmed sequentially while the output at each stage will be outputted to observe the results.

Step 3: Experiments

To first test the system, the Canny edge detection will be applied to images of buildings and bridges with cracks to observe the results. Also, to check whether the cracks are found effectively. The second test would be to adjust the parameters such as the Gaussian kernel, and high and low thresholds, and see the effects it has on the accuracy of crack detection. Lastly, to verify the implementation of the Canny algorithm, the gradient magnitude and orientation will be calculated externally for a few pixels in the images and compared to the values found by the algorithm.

Step 4: Metrics

Visual Inspection: Compare the output images at each stage of the Canny algorithm with the original images to ensure that there are well defined edges represent cracks.

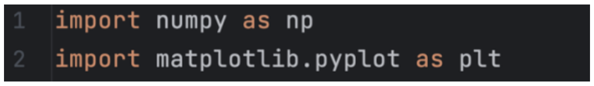
Accuracy and Precision: Adjust the parameters of the Canny algorithm to check the accuracy and precision of the cracks.

Adaptability: Test the Canny algorithm on multiple images with different lighting, crack types and angles to check its adaptability in detecting cracks under various scenarios.

Implementation

The implementation of all the stages of the Canny edge algorithm in Python using NumPy and matplotlib is shown sequentially below. The code was written with the assistance of ChatGPT [11].

The first step was to import the two libraries as seen in Figure 3. NumPy is used to convert the input image to grayscale, create the Gaussian kernel, padding the image with zeros and manipulating arrays for finding magnitude and orientation for Sobel gradients. Matplotlib is used to display the images specifically the output at each stage.

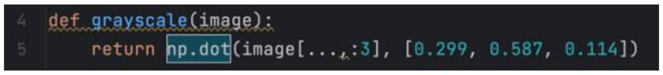


```python
import numpy as np
import matplotlib.pyplot as plt
```

Figure 3: Importing NumPy and matplotlib libraries

Grayscale input image:

As seen in Figure 4, the grayscale function takes image parameter as an input and returns a grayscale image. The 'image' parameter is the colored input image which is assumed to be a NumPy array. The '…,:3' is used to select the first three color channels of the image (red, green and blue channels). Next '[0.299, 0.587, 0.114]' are the weights used for the RGB channels. The 'np.dot()' function performs the dot product between the RGB channels and the weight array which results in a grayscale image.



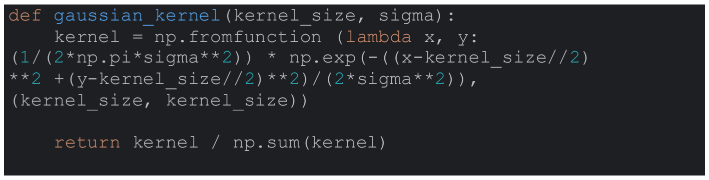
```python
def grayscale(image):
    return np.dot(image[...,:3], [0.299, 0.587, 0.114])
```

Figure 4: Implementing the grayscale function

Gaussian Smoothing:

Figure 5 and Figure 6 shows two functions being implemented that perform Gaussian smoothing on an input image. The first function takes two parameters which are 'kernel_size' (an odd integer) and 'sigma' (standard deviation). The kernel_size is 5x5 which is the standard size used for Canny edge detection. It uses 'np.fromfunction()' to create a 2D array that represents the Gaussian kernel. The lambda function inside 'np.fromfunction()' calculates the value for each of the element in the kernel based on the Gaussian formula shown earlier. The return statement normalizes the kernel to make sure that the elements sum up to 1 and then returns the resulting Gaussian kernel.

The second function takes two parameters: 'image' and 'kernel'. It calculates the 'kernel_size' based on the size of the provided kernel. To handle edge pixels during convolution it pads the input image with zeroes on all sides. After padding, a new output array ('result') with the same dimensions as the input image but initialized with zeros. Next, using nested loops the function performs convolution for each pixel by taking the region of the image that overlaps with the kernel, taking the element-wise product with the kernel, and then storing the absolute value of the sum in the matching location of the 'result' array. This function returns the 'result' array that has the filtered image found by convolving the input image with the given kernel.



```python
def gaussian_kernel(kernel_size, sigma):
    kernel = np.fromfunction (lambda x, y:
(1/(2*np.pi*sigma**2)) * np.exp(-((x-kernel_size//2)
**2 +(y-kernel_size//2)**2)/(2*sigma**2)),
(kernel_size, kernel_size))

    return kernel / np.sum(kernel)
```

Figure 5: Implementing Gaussian smoothing with the use of first function which is 'gaussian_kernel'

```
def convolution2d(image, kernel):
    kernel_size = kernel.shape[0]
    pad = kernel_size // 2
    height, width = image.shape
    padded_image = np.pad(image,
pad_width=((pad, pad), (pad, pad)),
mode='constant', constant_values=0)
    result = np.zeros_like(image)
    for y in range(height):
        for x in range(width):
            result[y, x] =
np.abs(np.sum(padded_image[y:y + kernel_size,
x:x + kernel_size] * kernel))

    return result
```

Figure 6: Implementing Gaussian smoothing with the use of two functions which are 'convolution2d'

Sobel gradients:

The function shown in Figure 7 is the implementation of the Sobel gradients to find the intensity gradients of the image. Two Sobel filter kernels 'sobel_x' for detecting horizontal edges and 'sobel_y' for detecting vertical edges. Both are 3x3 matrices that represent gradient values in the x and y directions. Next, 'sobel_x' and 'sobel_y' are convolved with the input image to find the gradients in the x and y directions for each pixel of the image. The gradient magnitude and orientation are then calculated for each pixel using their respective formulas shown earlier. Lastly, it returns two 2D arrays: 'gradient_magnitude' and 'gradient_orientation' which will be used in the next stages.

```
def sobel_gradients(image):
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2],
[-1, 0, 1]])
    sobel_y = np.array([[-1, -2, -1], [0, 0, 0],
[1, 2, 1]])
    gradient_x = convolution2d(image, sobel_x)
    gradient_y = convolution2d(image, sobel_y)
    gradient_magnitude = np.sqrt(gradient_x**2 +
gradient_y**2)
    gradient_orientation =
np.arctan2(gradient_y, gradient_x)
    return gradient_magnitude,
gradient_orientation
```

Figure 7: Implementation of Sobel operators

Non-Max Suppression:

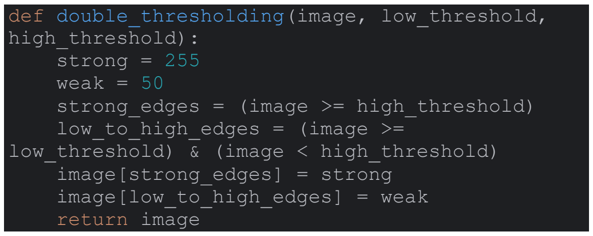
The implementation of Non-Max Suppression can be seen in Figure 8. This function takes two input arrays: 'gradient_magnitude' and 'gradient_orientation'. The width and height of 'gradient_magnitude' is found and used to help find the dimensions for the 'suppressed' output array. The 'angle_quantized' is found by converting the 'gradient_orientation' array from radians to degress and then rounding the values between $(0°, 45°, 90°, and\ 135°)$. Next the function goes through each pixel in the 'gradient_magnitude' using nested for loops while ignoring the border to make sure that edges of the images are unaffected. For each pixel, it gets the quantized angle then determines which neighboring pixels to compare with to check for local maxima. For this function, 'q' and 'r' represent the gradient magnitudes of neighboring pixels and to calculate the values of them the function uses the gradient magnitude of the corresponding magnitude value. After finding the values, the gradient magnitude of the current pixel is compared with 'q' and 'r' and if the current pixel is greater than or equal to both 'q' and 'r', the pixel is a local maximum, and its value is stored in the 'suppressed' array or else it's assigned a value of 0. Lastly the function returns the 'suppressed' array which has thinned and well-defined edges.

```
def non_max_suppress(gradient_magnitude, gradient_orientation):
    height, width = gradient_magnitude.shape
    suppressed = np.zeros((height, width), dtype=np.float32)
    angle_quantized = (gradient_orientation * 180.0 / np.pi) %
180.0
    for y in range(1, height - 1):
        for x in range(1, width - 1):
            angle = angle_quantized[y, x]
            q, r = 0, 0
            if 0 <= angle < 22.5 or 157.5 <= angle < 180:
                q = gradient_magnitude[y, x + 1]
                r = gradient_magnitude[y, x - 1]
            elif 22.5 <= angle < 67.5:
                q = gradient_magnitude[y + 1, x - 1]
                r = gradient_magnitude[y - 1, x + 1]
            elif 67.5 <= angle < 112.5:
                q = gradient_magnitude[y + 1, x]
                r = gradient_magnitude[y - 1, x]
            elif 112.5 <= angle < 157.5:
                q = gradient_magnitude[y - 1, x - 1]
                r = gradient_magnitude[y + 1, x + 1]
            if gradient_magnitude[y, x] >= q and
gradient_magnitude[y, x] >= r:
                suppressed[y, x] = gradient_magnitude[y, x]
    return suppressed
```

Figure 8: This function performs non-maximum suppression

Double Thresholding:

As seen in Figure 9, this function implements Double thresholding, and it takes three inputs: ‘image’, ‘low_threshold’, and ‘high_threshold’. Firstly, two intensity values were defined ‘strong’ and ‘weak’ which will be used to label the edge pixels. Next two masks are created ‘strong_edges’ for elements ≥ high_threshold and ‘low_to_high_edges’ for elements ≥ ‘lower_threshold’ and less than ‘high_threshold’. The ‘high_threshold’ and ‘low_threshold’ are declared in main with values 110 and 50 respectively since it produced valid outputs and they’re used to classify the pixels as strong, weak or non-edges. Now, the function will assign ‘strong’ value to ‘strong_edges’ and ‘weak’ value to ‘low_to_high_edges’ while the rest are labelled as non-edges and given a value of 0. This function returns an image with strong, weak and non-edges.

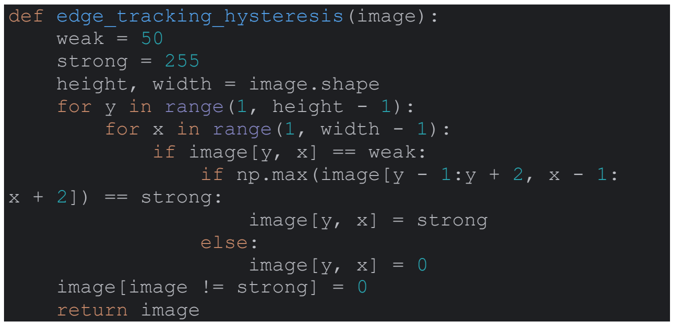


```
def double_thresholding(image, low_threshold,
high_threshold):
    strong = 255
    weak = 50
    strong_edges = (image >= high_threshold)
    low_to_high_edges = (image >=
low_threshold) & (image < high_threshold)
    image[strong_edges] = strong
    image[low_to_high_edges] = weak
    return image
```

Figure 9: Implementation of Double Thresholding

Edge Tracking by Hysteresis:

As seen in Figure 10, this function takes the image as input which is the result after double thresholding. To start, two variables are initialized as ‘weak’ and ‘strong’ which will be used to label the edges. Using nested for loops, the function iterates over the image and checks for pixels marked as ‘weak’. After finding a ‘weak’ pixel, the 8 neighboring pixels are also checked and if one of them has a value of ‘strong’, the weak pixel is promoted to a ‘strong’ value since they’re connected. Otherwise, the weak pixel is set to 0 and considered a non-edge. After going through all weak pixels, all non-edges are set to 0 which removes them from the final output. This function returns an output image with strong edges that is continuous and well-defined.

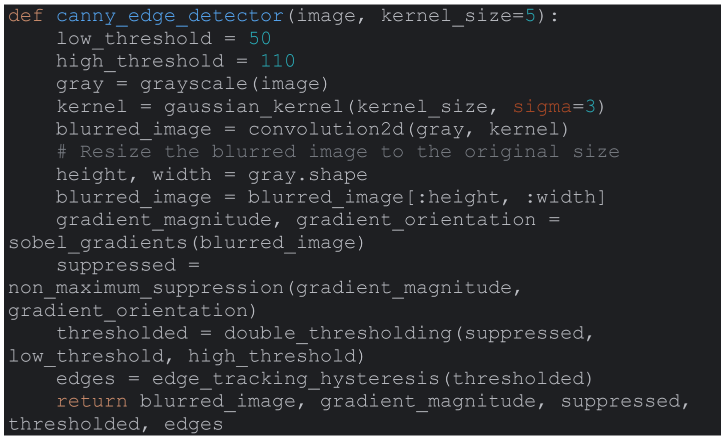


```
def edge_tracking_hysteresis(image):
    weak = 50
    strong = 255
    height, width = image.shape
    for y in range(1, height - 1):
        for x in range(1, width - 1):
            if image[y, x] == weak:
                if np.max(image[y - 1:y + 2, x - 1:
x + 2]) == strong:
                    image[y, x] = strong
                else:
                    image[y, x] = 0
    image[image != strong] = 0
    return image
```

Figure 10: Implementation of Edge Tracking by Hysteresis which connects weak edges (50) to strong edges (255)

Main function/Final Output:

As seen in Figure 11, the main function implements the complete Canny edge detection algorithm. It takes an input image along with kernel size while the low threshold and high threshold are assigned values. This function calls all the previous functions described above sequentially to produce an output.



```
def canny_edge_detector(image, kernel_size=5):
    low_threshold = 50
    high_threshold = 110
    gray = grayscale(image)
    kernel = gaussian_kernel(kernel_size, sigma=3)
    blurred_image = convolution2d(gray, kernel)
    # Resize the blurred image to the original size
    height, width = gray.shape
    blurred_image = blurred_image[:height, :width]
    gradient_magnitude, gradient_orientation =
sobel_gradients(blurred_image)
    suppressed =
non_maximum_suppression(gradient_magnitude,
gradient_orientation)
    thresholded = double_thresholding(suppressed,
low_threshold, high_threshold)
    edges = edge_tracking_hysteresis(thresholded)
    return blurred_image, gradient_magnitude, suppressed,
thresholded, edges
```

Figure 11: Implementation of main function with all the other functions being called in order to complete the algorithm

Results and Discussion

The results show the importance of each stage of the Canny edge detection algorithm and how it affects the input image when it comes to finding edges related to cracks. To start an input image of a bridge with cracks was inputted into the algorithm and the first two stages were performed together which was to grayscale the image and then perform Gaussian Blur as seen in Figure 12. This stage resulted in the input image being transformed into a smooth grayscale image to reduce noise within the image while keeping all the essential features such as the cracks. Figure 13 shows the result after applying Sobel operators to the smoothed grey image which highlights all the major intensity regions with the use of gradient magnitude allowing the next stages to precisely identify and track the edges with cracks. Figure 14 shows the result after the input image goes through non-max suppression which produces an image with more defined edges specifically focused toward the cracks on the bridge. Figure 15 is the result after the Double Thresholding which classified all the strong and weak edges while Figure 16 performs Edge Tracking by Hysteresis and produces the final output that clearly shows the major cracks within the bridge that needs to be resolved immediately.

The results acquired from the algorithm successfully identify the major cracks within the bridge, but the algorithm can be improved further to detect even smaller cracks in buildings and bridges using image enhancement, adjusting parameters for edge detection and incorporating contextual information. Before applying the canny edge detection algorithm, if the input image is enhanced to clearly show the cracks it can significantly increase the edges detected by the algorithm because more pixels can be captured within the image. The current algorithm has a fixed value for the Gaussian sigma, kernel size, low threshold and high threshold but if these values were adjusted based on the image itself and the types of cracks present on the bridges and buildings. It would enhance the number of relevant edges captured specifically the edges that contribute to finding the cracks on bridges and buildings while filtering out all the unnecessary noise. This would make the algorithm far more accurate and reliable for finding major cracks. Lastly, another process that can be added to the algorithm is incorporating information related to typical patterns of cracks on buildings and bridges to help identify different types of cracks and to reduce the number of false edges captured.

The importance of this study is to demonstrate how the Canny edge detection algorithm being used for identifying cracks on buildings and bridges can help Civil engineers identify these major cracks. By detecting the cracks early, engineers can take timely actions such as maintenance on the building or bridge to prevent potential disasters and structural failures while ensuring the safety of the population. This method is also very cost-effective and the algorithm can be improved to detect small cracks before these major cracks occur which would allow timely maintenance on the building or bridge which is way less costly than fixing major cracks. Also, this algorithm only requires an image of the building or bridge to find the cracks which is a very efficient method to finding such a crucial component.

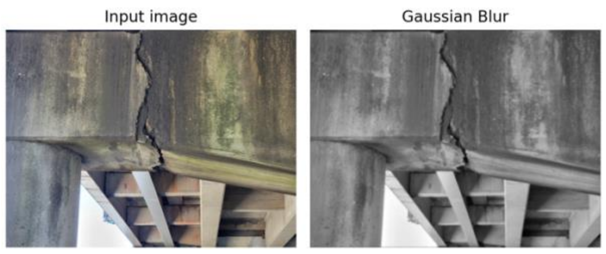


Figure 12: The result achieved after applying Gaussian filter where the original input image (left side) compared to the output after grayscale and Gaussian Blur (right side)

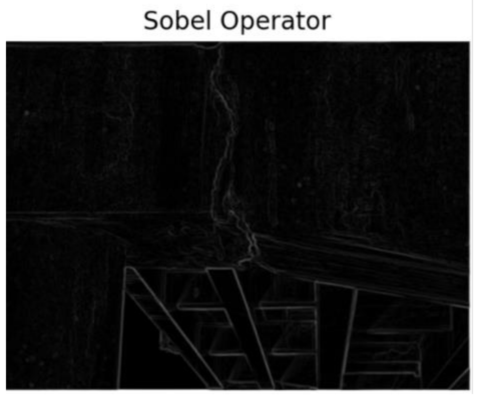


Figure 13: The result after applying Sobel operators to the input image which shows all the major intensity regions within the image.

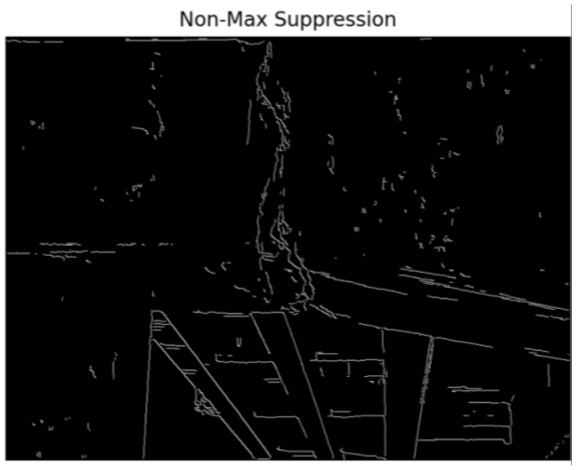


Figure 14: The result after applying Non-Max Suppression to the input image to clearly find the cracks on the bridge

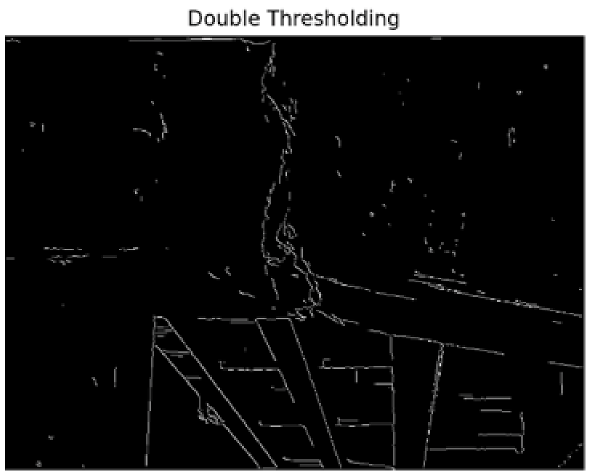


Figure 15: The result after applying Double Thresholding to the input image to classify the weak and strong edges of the image

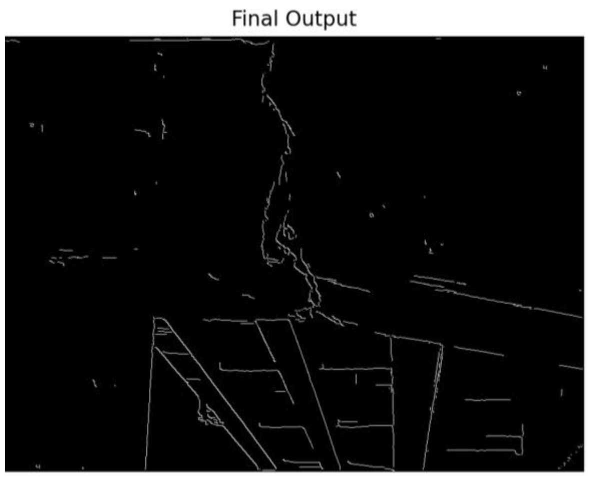


Figure 16: The result after applying Edge Tracking by Hysteresis which gives you the final output.

Conclusion

This paper presented a study conducted on the Canny Edge Detection algorithm for identifying cracks in buildings and bridges. Each stage of the algorithm which included Gaussian smoothing, Sobel operators, Non-Maximum Suppression, Double Thresholding, and Edge Tracking by Hysteresis was explained and displayed to understand the significance of each stage in crack detection. The results showed that the algorithm can identify major cracks on buildings and bridges which shows its potential to increase safety and avoid structural failures. The algorithm can be further improved to enhance its accuracy and effectiveness in detecting cracks in buildings and bridges by implementing image enhancement, adjusting parameters and contextual information. The importance of this study was also discussed since it has the potential to transform crack detection in various structures, allowing for timely intervention to avoid disasters and structural failures while being cost-effective. This helped prove the importance of Canny edge detection for Civil engineers. Future research can be focused on improving the algorithm and adding machine learning techniques to improve the accuracy and reliability of detecting cracks. This algorithm can also be expanded to incorporate more applications in the field of Civil engineering with further research. In conclusion, the study has successfully shown the use of Canny Edge Detection in finding cracks in buildings and bridges making it a potential tool for Civil engineers to promote safe and long-lasting buildings and bridges.

References

[1] V. M, "Comprehensive Guide to Edge Detection Algorithms," 7 August 2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2022/08/comprehensive-guide-to-edge-detection-algorithms/. [Accessed 22 July 2023].

[2] M. N. Teli, "Canny Edge," University of Maryland, 12 May 2016. [Online]. Available: https://www.cs.umd.edu/class/fall2019/cmsc426-0201/files/11_CannyEdgeDetection.pdf. [Accessed 22 July 2023].

[3] E. B. M. M. Ehsan Akbari Sekehravani, "Implementing canny edge detection algorithm for noisy image," 4 August 2020. [Online]. Available: https://www.researchgate.net/publication/341051272_Implementing_canny_edge_detection_algorithm_for_noisy_image. [Accessed 23 July 2023].

[4] M. Kumar, "Canny Edge Detection for Image Processing," 23 January 2022. [Online]. Available: https://indiantechwarrior.com/canny-edge-detection-for-image-processing/. [Accessed 20 July 2023].

[5] S. Singh, Artist, *What is Canny Edge Detection.* [Art]. 2020.

[6] S. Singh, Artist, *Non-max suppression on gradient edges.* [Art]. 2020.

[7] S. V. C. C. F. I. a. L. J. K. Qian Xu, "A Distributed Canny Edge Detector: Algorithm and FPGA Implementation," 7 July 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6774938. [Accessed 20 July 2023].

[8] J. Canny, "A Computational Approach to Edge Detection," November 1986. [Online]. Available: https://ieeexplore.ieee.org/document/4767851. [Accessed 21 July 2023].

[9] V. Italonia, "Canny Edge Detection.," Medium, 14 November 2019. [Online]. Available: https://medium.com/bleep-bloop/canny-edge-detection-712c45ee1431. [Accessed 24 July 2023].

[10] J. Liang, "Canny Edge detection," 24 June 2018. [Online]. Available: https://justin-liang.com/tutorials/canny/#grayscale. [Accessed 22 July 2023].

[11] ChatGPT, "ChatGPT," OpenAI, [Online]. Available: https://chat.openai.com/. [Accessed 20 July 2023].

[12] S. Kaushal, "Beginner's Guide To Image Gradient," Analytics Vidhya, 27 July 2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2022/07/beginners-guide-to-image-gradient/#:~:text=The%20gradient%20can%20be%20defined,the%20presence%20of%20an%20edge.. [Accessed 22 July 2023].