An Investigation of The Effect Smoothing Has on Edge Detection Algorithms

To what extent does Smoothing Reduce False Edge Detection from Blurry Areas of Images on Various Edge Detection Algorithms?

Computer Science Extended Essay

Word count: 3999

CS EE World
https://cseeworld.wixsite.com/home
May 2023
23/34
B

Submitter info:
Name: Haichuan Wang
Contact: pridak [at] foxmail [dot] com
University: UCSD

Table of Contents

INTRODUCTION....4

EDGE DETECTION PROCESS: AN OVERVIEW....7

SETTING A UNIFIED STANDARD: GRAYSCALE....7

THE NEED FOR EXCLUDING NOISES: SMOOTHING....7

EDGE DETECTION TECHNIQUE: DIFFERENTIATION....9

NON-MAXIMUM SUPPRESSION (NMS)....11

THRESHOLDING....12

ALGORITHMS....14

BASIC CONCEPTS: KERNEL, CONVOLUTION, AND OPERATORS....14

SOBEL OPERATOR....15

ROBERTS CROSS OPERATOR....16

PREWITT'S OPERATOR....17

LAPLACIAN OF GAUSSIAN OPERATOR....17

CANNY EDGE DETECTION ALGORITHM....18

EXPERIMENTATION....20

METHODOLOGY....20

HYPOTHESIS....23

RESULTS....25

EVALUATION ... 27

STATISTICS ... 27

VISUAL ... 28

METHODOLOGY ... 32

CONCLUSION ... 34

BIBLIOGRAPHY ... 35

APPENDIX ... 37

SAMPLE IMAGE ... 37

CODE FOR EDGE DETECTION ALGORITHMS ... 38

Introduction

For photographers, it's important to determine if the focus is accurate before taking a picture. To meet the demands of such a group of customers, camera manufacturers would integrate functions that assist photographers in determining the focal length into their products. One of the very useful functions is manual focus peaking[1] (MF peaking). When MF peaking is enabled, it will automatically start detecting areas within the photo where areas of high contrast exist and marking them with a colored silhouette[2]. This would allow users to distinguish whether the effects the current focal length presents fit their needs. In the field of computer science, MF peaking belongs to a larger topic, edge detection.

Edge detection is essentially the technique for computers to capture and identify the "significant properties of objects"[3] in images. Such properties could be caused by "discontinuity in photometrical, geometrical and physical characteristics of objects."[4] These discontinuities would eventually be displayed in the way of grey level (also known as grayscale), a pattern that shows the color intensity of pixels in a digital

---

1 CANON INC.2019. (2019). *PowerShot G7 X Mark III Advanced User Guide* [Press release]. https://gdlp01.c-wss.com/gds/8/0300035728/04/psg7x-mk3-ug4-en.pdf

2 *Nakahara, K. (2021, March 2). Focus Guide & MF Peaking: Making Manual Focus Easier. SNAPSHOT. Retrieved September 20, 2022, from https://snapshot.canon-asia.com/reg/article/eng/focus-guide-mf-peaking-making-manual-focus-easier*

3 *Ziou, D., & Tabbone, S. (1998). Edge detection techniques-an overview. Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii, 8, 537-559.*

4 *Ziou, D., & Tabbone, S. (1998). Edge detection techniques-an overview. Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii, 8, 537-559.*

image in the form of black and white[5]. Pixels with a great variance of color intensity with nearby pixels would be considered to have high contrast in the image. During the process of edge detection, computers are given the task to identify and visualize those pixels, eventually forming images that consist of only pixels with high contrast. shows the result of edge detection is shown on top of the image, assisting photographers to determine appropriate focal lengths.

However, MF peaking also has problems. For example, a blur too shallow might not create enough color confusion, deceiving the computer, and resulting in a visualization that covers areas where they are not supposed to be marked[6]. Such mistakes could be fatal for a professional photographer. Different algorithms would also affect the quality of the final products.

The fact that such a problem exists with MF peaking intrigued the topic of this essay. One solution to reduce this type of problem is “smoothing”, a technique that blurs images to reduce color contrast so fewer edges would be detected. This essay aims to conduct an experiment, in which different edge detection algorithms would be tested with the application of smoothing being the dependent factor to see to what extent

---

[5] *Johnson, S. (2006). Stephen Johnson on Digital Photography. Van Duuren Media.*

[6] *(2019, February 6). just hold there, with peak MF level anywhere is in focus, and with peak MF level nowhere is in focus, I am used to zooming in on the focus [Comment on “The MF peaking that will change your life, a must-see tip for novice photographers!”]. https://www.bilibili.com/video/BV1Xb411k7AL*

smoothing improves the effectiveness of edge detection. Furthermore, since little is known about the algorithm MF peaking utilizes, various algorithms would be tested.

Edge Detection Process: an Overview

Setting a Unified Standard: Grayscale

The most basic concept of edges is the high contrast between pixels which results in a great difference in color intensities between nearby pixels. The first thing computers do when processing an edge detection is to identify those pixels that have a large difference in grayscale value from that of nearby ones. To set a unified standard and increase efficiency, colored images are transferred into the form of grayscale. When a typical RGB (one that uses different intensity values of red, green, and blue to present colors) image is converted into a grayscale image, the values of its RGB intensity are converted into a unified value of black and white: the weakest value, 0, represents black and the strongest, 255, represents white, with 254 different shades of gray in between them.[7] When a computer reads the information of pixels in an image in the format of grayscale, it only has to compare values in one channel rather than in three, which simplifies the logic. The formula for converting RGB images into grayscale ones is shown below:

$$grayscale = (0.299 * R) + (0.587 * G) + (0.114 * B)$$[8]

The Need for Excluding Noises: Smoothing

After an image is converted from colored into grayscale, the computer needs to make the grayscale image “smooth”. The process of smoothing is required due to the

[7] Johnson, S. (2006). Stephen Johnson on Digital Photography. Van Duuren Media.

[8] Saravanan, C. (2010, March). Color image to grayscale image conversion. In 2010 Second International Conference on Computer Engineering and Applications (Vol. 2, pp. 196-199). IEEE.

presence of “noise”. By definition, “image noise” stands for “random variation of image intensity and visible as grains in the image”[9]. The reason for the emergence of image noise may vary[10], but the damage they cause is the same: edge detection is originally about identifying a continued discontinuity that forms lines; however, with image noise comes an irregular pattern of discontinuity in color intensity that may lead to a visualization of edge detection with obvious deficiency. If image noise appears in an area out of a camera’s focus, MF peaking may take it as an area with high contrast, creating confusion for its user, and failing to deliver its original purpose. That is why “smoothing” may be particularly important for MF peaking, and becomes the independent factor that will be examined in the later part of this essay. Generally, smoothing is done by applying blur filters onto images. When blur filters are applied, each pixel will go through a series of computations in which “pixels in a small neighborhood of the concerned pixel are involved”[11] The computation in this context refers to averaging. A pixel of image noise shows a great color intensity discontinuity with its nearby pixels, and averaging reduces such discontinuity by making the color intensity difference decrease. Overall, smoothing provides an option to reduce the noise in images. However, this also comes at a cost: images may experience a loss of

---

[9] *Verma, R., & Ali, J. (2013). A comparative study of various types of image noise and efficient noise removal techniques. International Journal of advanced research in computer science and software engineering, 3(10).*

[10] *Owotogbe, J. S., Ibiyemi, T. S., & Adu, B. A. (2019). A comprehensive review on various types of noise in image processing. int. J. Sci. eng. res, 10(11), 388-393.*

[11] *LEE, J. S. (1983). Digital Image Smoothing and the Sigma Filter. COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING, 24, 255-269.*

valid information as well.[12] When smoothing is applied, true edges are also blurred, and the confusion created would result in the exclusion of these true edges.

Edge Detection Technique: Differentiation

A good way of understanding how computers deal with inputs is to try to visualize the process. In an ideal situation, computers look for discontinuity in the grayscale values in images.

Figure one: an abrupt change in color intensity of a grayscale image

Visualizing those abrupt changes in color intensity would become a graph like this:

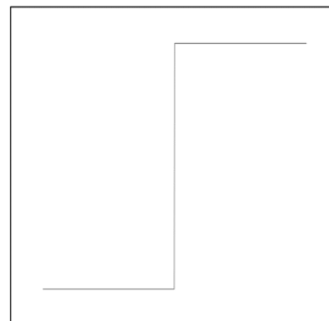

Figure two: a visualization of an abrupt change in color intensity of a grayscale image using graph[13]

[12] *Ziou, D., & Tabbone, S. (1998). Edge detection techniques-an overview. Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii, 8, 537-559.*

[13] *Ziou, D., & Tabbone, S. (1998). Edge detection techniques-an overview. Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii, 8, 537-559.*

However, in real-life practices, a discontinuity in color intensity would not look this drastic. Instead, there shall be a rather gradual change:

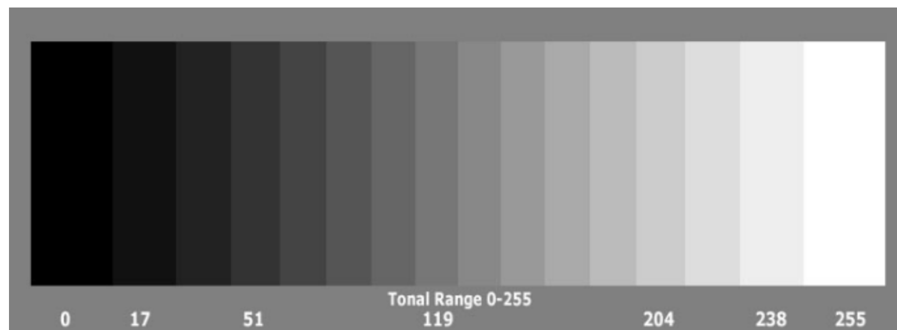

Figure three: A gradual change in color intensity of a grayscale image[14]

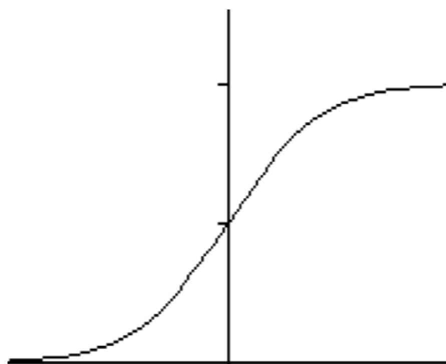

Figure four: a visualization of a gradual change in color intensity of a grayscale image using graph [15]

This implies that instead of an absolutely clear edge between two objects in an image, it's more common to see an edge with some thickness (of several pixels). Still, the most obvious difference in color intensity is when the change in the intensity is at maximum, which can be identified via the first derivative of the grayscale curve:

[14] Antoniadis, P. (n.d.). How to Convert an RGB Image to a Grayscale. Baeldung. https://www.baeldung.com/cs/convert-rgb-to-grayscale

[15] Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.

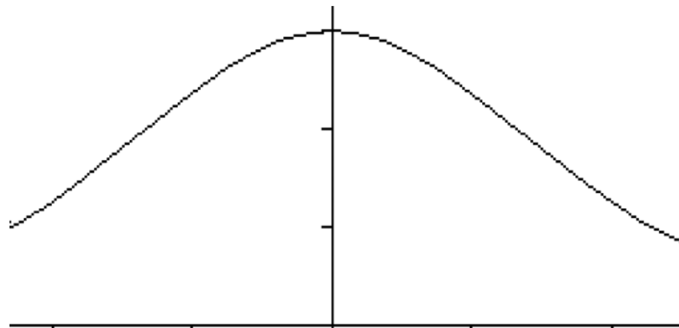

Figure five: the maximum of the first derivative[16]

Another way to get to that point can be done by finding the location of the pixel when the second derivative is zero:

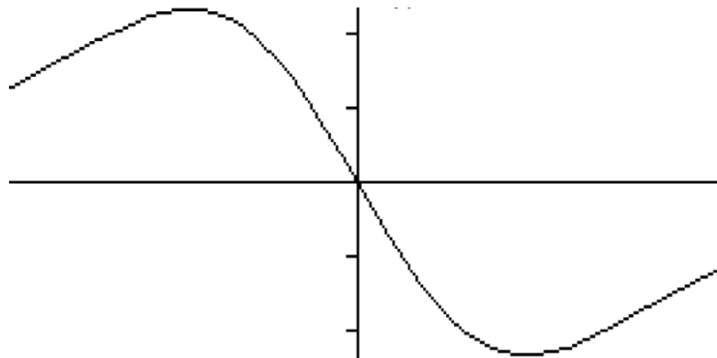

Figure five: the zero crossing of the second derivative[17]

Non-Maximum Suppression (NMS)

Non-maximum suppression is often used with the first-derivative test as it needs information on the gradient direction to work. Since the first derivative test measures the magnitude of change if color intensity horizontally and vertically, the approximate direction where the change in color intensity can also be revealed using trigonometry:

---

[16] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

[17] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

$$\theta = \tan^{-1}\left(\frac{Gy}{Gx}\right)$$ [18]

NMS is a technique used to thin out the edges detected by the algorithm and only keep the strongest ones. When applying NMS to a pixel on an edge, color intensity values along the gradient direction of that pixel are compared to each other, and the one pixel with the highest gradient magnitude would be kept, with all other gradient magnitudes of pixels in that direction dropped, leaving the edge itself one-pixel thick. NMS allows the output to be more precise, though also bears the risk of information loss, as weak edges are further weakened in this process.

Thresholding

For a grayscale image that has gone through a derivative test, its output would be a matrix with its size corresponding to the original image, and each unit location (corresponding to the original pixel location) is placed with a value showing the magnitude of change in color intensity, horizontally and vertically combined. If such output is taken as the final result of edge detection, then it would be an undesirable outcome, as any pixel where there is a change in color intensity in its neighboring pixels would be considered an edge, contradicting edge detection's original intention that only "significant properties of objects"[19] are marked. This means a unified standard in the form of a specific value, or a range of values needs to be used to judge the initial output so that only the "true" edges, edges that are wanted, are preserved

---

18 *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

19 *Ziou, D., & Tabbone, S. (1998). Edge detection techniques-an overview. Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii, 8, 537-559.*

and “false” edges or image noise can be ignored[20]. The process of applying this system is called thresholding, where a threshold value is chosen to separate the pixels into two groups: one group with pixel values above the threshold, and the other with pixel values below the threshold. The thresholding process is commonly used in edge detection to distinguish the edge pixels from the non-edge pixels by setting a threshold on the gradient magnitude or other edge detection operator. The pixels above the threshold value are considered edge pixels and the pixels below the threshold value are considered non-edge pixels.

[20] *Gonzalez, R. C., & WOODS 3rd, R. E. (2008). Edition. Digital Image Processing. Upper Saddle River, USA: Prentice Hall.*

Algorithms

Basic Concepts: Kernel, Convolution, and Operators

The way of locating discontinuities with the color intensities (whether by the first derivative or by the second derivative) needs to be further explained.

A kernel, in the field of image processing, is a function based on a matrix. A kernel would be applied to a pixel by applying the function to the intensity value of that pixel.[21] To help visualize that, here's a graph of a kernel:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure six: a kernel used to blur an image[22]

When this kernel is applied to a pixel, it adds up all the color intensity values of all eight pixels around it to that of itself, each value is weighted by one. After that, the sum of the nine values is divided by nine, getting the average.

Convolution, in an academically rigorous tone, stands for "a mathematical operation on two functions (f and g) that produces a third function ($f * g$) that

---

[21] *Gonzalez, R. C., & WOODS 3rd, R. E. (2008). Edition. Digital Image Processing. Upper Saddle River, USA: Prentice Hall.*

[22] *Kernel (image processing). (2012). In Wikipedia. Retrieved September 22, 2022, from https://en.wikipedia.org/wiki/Kernel_(image_processing)*

expresses how the shape of one is modified by the other."[23] However, in the field of image processing, convolution can be simply explained as the process of applying a kernel, a small matrix of numbers, to an image with a larger size than that of the kernel, to obtain a filtered image. The convolution operation involves sliding the kernel over the input image, performing weighted multiplication at each pixel, and summing the results to produce a single output value. The new values of color intensity would be presented on a brand-new canvas that has the same size as the original image, where the new value is painted in the same position the old value is located.

With different kernels and different processes of convolutions, different types of algorithms, or to be stricter, "operators", are created.

Sobel Operator

The Sobel operator uses the first derivative of color intensity to locate discontinuities. Locating process would run two times on an image, with one time horizontally, and the other time vertically. This means two kernels would be applied in two directions, with one being the 90° rotated version of another. Here's the visualization of the

[23] *Wikipedia contributors. (2023, February 4). Convolution. Wikipedia. https://en.wikipedia.org/wiki/Convolution*

Sobel Operator's kernels:

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

Gx

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

Gy

Figure seven: the two kernels used by Sobel Operator, Gx would apply in the horizontal direction, Gy would apply in the vertical direction[24]

The final value of intensity on a single pixel can be calculated by:

$$|G| = \sqrt{Gx^2 + Gy^2}$$ [25]

Roberts Cross Operator

The Roberts cross operator follows the same rules as the Sobel operator. What sets these two operators apart is that the Roberts cross operator has two simpler kernels, "designed to respond maximally to edges running at 45° to the pixel grid"[26]:

+1	0
0	-1

0	+1
-1	0

Gx Gy

Figure eight: the two kernels used by Roberts cross operator[27]

[24] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

[25] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

[26] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

[27] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

Roberts cross operator also follows the same function to calculate the final color intensity value of a pixel:

$$|G| = \sqrt{Gx^2 + Gy^2}$$ [28]

Prewitt's Operator

Prewitt's operator stands as an even more similar operator to the Sobel operator, with the only difference being the kernels:

-1	0	+1
-1	0	+1
-1	0	+1

Gx

+1	+1	+1
0	0	0
-1	-1	-1

Gy

Figure nine: kernels of the Prewitt's operator[29]

Laplacian of Gaussian Operator

The Laplacian of Gaussian operator uses the second derivative test to locate discontinuities in color intensity. However, the Laplacian of Gaussian operator is extremely sensitive to noises. While operators using first derivative tests detect edges by finding local maximums, the Laplacian filter applies second derivative tests, identifying edges by looking for zero crossings. For images with noises, the first derivative test could avoid taking them into account by setting a threshold to exclude small variations in the change in color intensities. However, the second derivative test takes pixels into account whenever there is a zero crossing. Therefore, noises that can

[28] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

[29] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

be excluded by the first derivative test will be seen as edges by the second derivative test. Although thresholding can solve part of this problem by excluding false edges detected, the second derivative test still needs further optimization due to its high sensitivity to noise. Hence, before applying the Laplacian operator to an image, a Gaussian filter needs to be applied first to smooth the original image. The kernel of Gaussian, though, is a mathematical function that describes a normal distribution of values: the size and the numbers in the kernel rely on the standard deviation. After convoluting the Gaussian kernel onto the original image, the image would be smoothed, therefore suitable to apply the Laplacian kernel. This process can also be done so the two steps are processed in one round: convolve the Gaussian kernel with the Laplacian kernel to create a new kernel, which is the Laplacian of Gaussian (LoG) kernel. The formula of the LoG kernel is written as:

$$LoG(\mathrm{x},\mathrm{y}) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2 + y^2}{2\sigma^2}\right] \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$ [30]

Canny Edge Detection Algorithm

The Canny edge detection algorithm is a mix of different algorithms: First, the original image is convoluted by the Gaussian kernel, then the output image is processed again by the Sobel operator, including thresholding and non-maximum suppression. Next, two thresholds are applied to the gradient magnitude to identify the potential edge pixels. Any pixel with a gradient magnitude above the high threshold is marked as a strong edge pixel, while any pixel with a gradient magnitude below the

[30] *Maini, R., & Aggarwal, H. (2009). Study and comparison of various image edge detection techniques. International journal of image processing (IJIP), 3(1), 1-11.*

low threshold is marked as a non-edge pixel. Pixels with a gradient magnitude between the high and low threshold are marked as weak edge pixels. Lastly, edges (whether strong or weak) are connected (known as edge tracking by hysteresis), where the weak edge pixels that are connected to strong edge pixels are included as part of the edge. The weak edge pixels that are not connected to strong edge pixels are discarded.

Experimentation

Methodology

In this experiment, one image with a deep depth of field presenting a clear object will be tested by the algorithms mentioned above twice. In the first test, the image will go through all algorithms without smoothing. Then, the same image would go through all algorithms again with smoothing (in this case, Gaussian blur). This will create five groups of outputs: each algorithm would have two output images, with one being the output without smoothing, and the other one with smoothing. The code of these algorithms comes from Jason Altschuler's repository on GitHub.[31] After collecting outputs, the original image would be processed so that the silhouette of the object within the image would be marked. Next, all the outputs would be upon the original input, which allows real edges (edges that are detected within the clear area, also known as the area the object in the test image covers) in the output images to overlap with the object in the input. The pixels from the outputs within the silhouette would then be deleted. By doing so, only false edges (edges detected within the blurry areas of the input image) are detected. Then, color range selection would be performed, which selects the pixels whose color (in this case, black) is used to visualize edges. As edges in the clear areas of images are already deleted, color range selection would only take pixels in the blur areas of images into account. Lastly, I would run image analyses to record the number of pixels that are wrongly identified as edges for further

---

[31] *J. (n.d.). EdgeDetector/detectors at master · JasonAltschuler/EdgeDetector. GitHub. https://github.com/JasonAltschuler/EdgeDetector/tree/master/detectors*

comparison. To help visualize this process, a series of screenshots would be presented below:

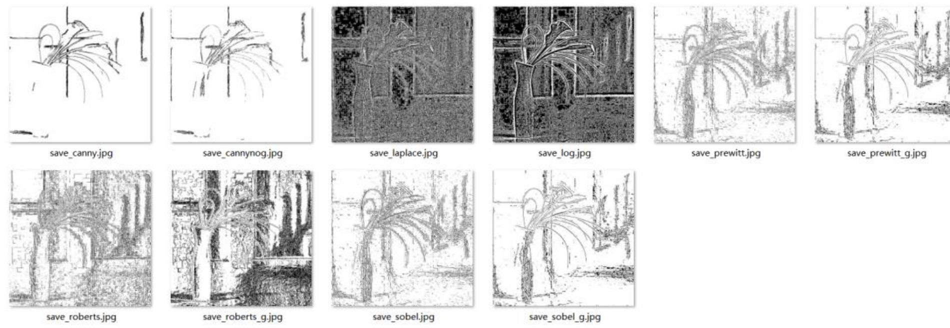

Figure ten: an image showing the outputs of all algorithms

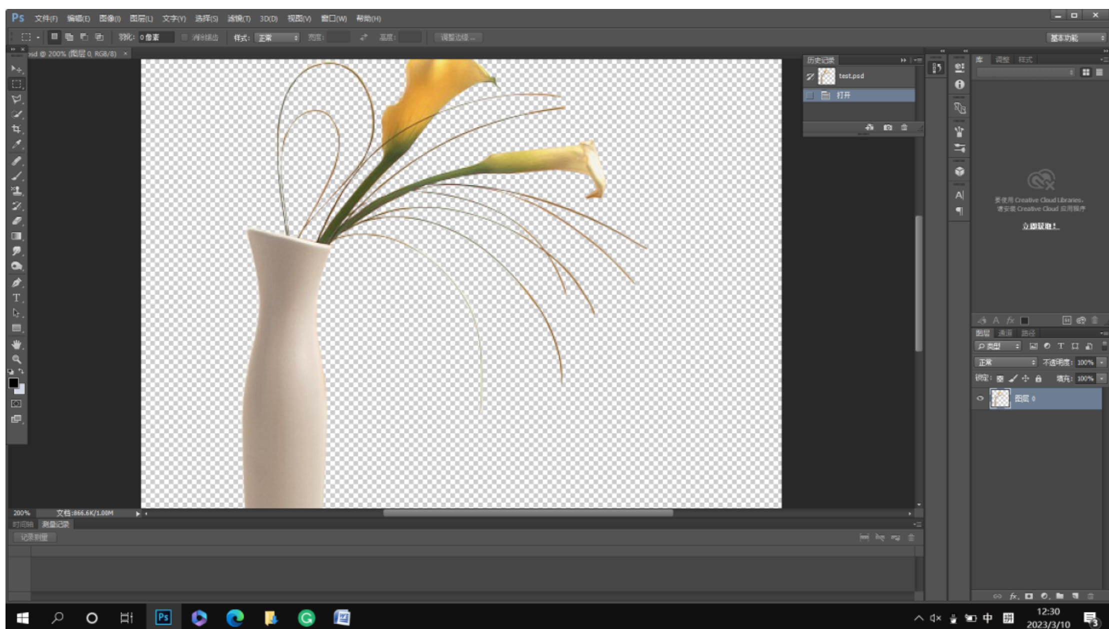

Figure eleven: an image showing the original input, whose pixels out of the clear object in the center are removed to obtain the object's silhouette so the silhouette can be easily marked

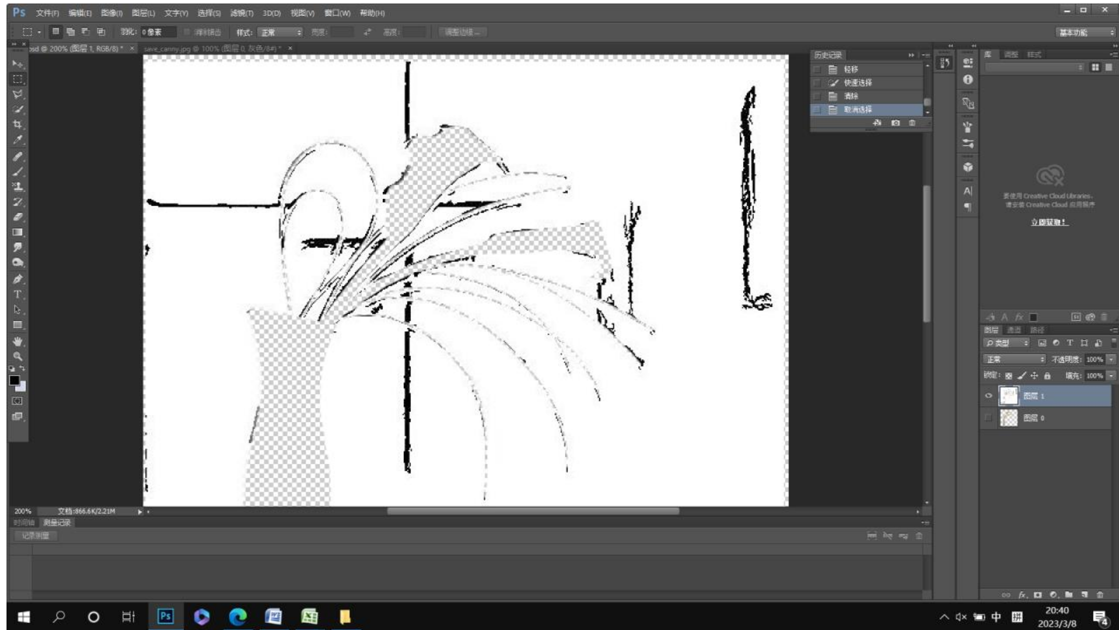

Figure twelve: an example of an output whose pixels of valid edges are removed

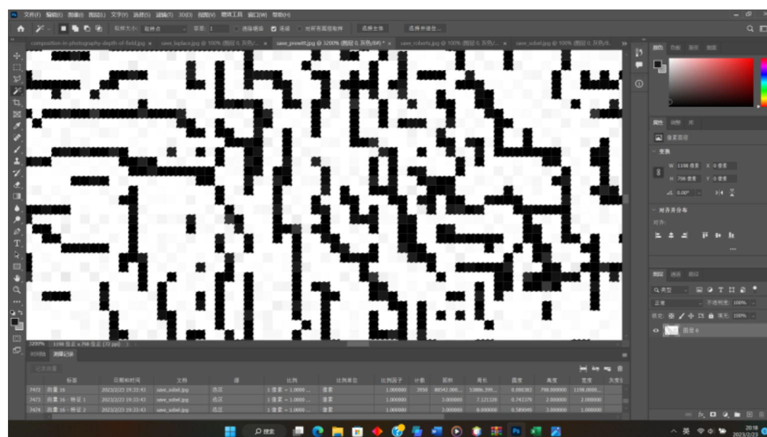

Figure thirteen: color range selection is performed

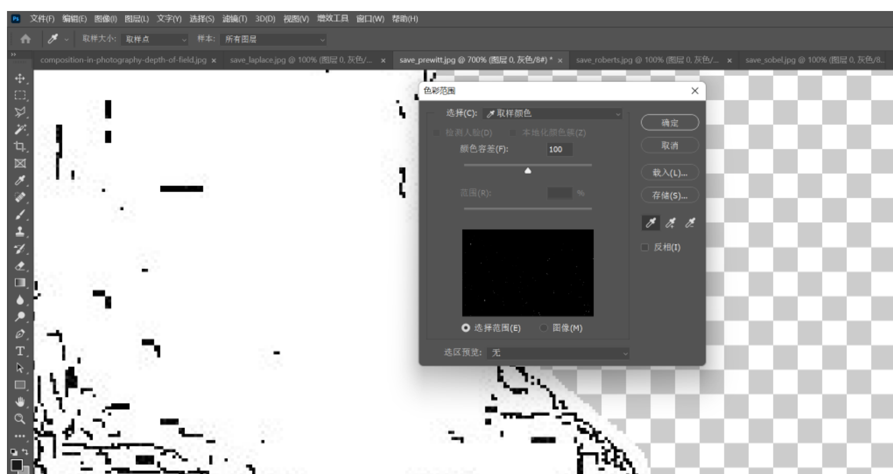

Figure fourteen: after color range selection, pixels representing invalid edges are selected.

文档	源	比例	比例单位	比例因子	计数
save_prewitt.jpg	选区 源 - 测量的源	1 像素 = 1.0000 ...	像素	1.000000	3942

Figure fifteen: the number of pixels wrongly identified as edges is shown in the final statistics

After data collection, the change in the detection results due to Gaussian blur for each algorithm would be compared to their counterparts. The change in accuracy will then be calculated in the form of a percentage, to provide a way to compare the effectiveness of smoothing across all algorithms. A comparison between the effectiveness of different algorithms will also be conducted to answer the question at the start. Furthermore, a visual evaluation would also be conducted, for some of the changes in the detection outputs cannot be revealed through statistical data.

Hypothesis

Performing Canny edge detection without Gaussian blur would likely lead to a more noisy and inaccurate edge map. The Gaussian blur is initially a built-in step in Canny edge detection, as it helps to reduce the impact of noise in the image. Without the Gaussian blur, the Sobel operator used to calculate the gradient of the image would amplify the noise, resulting in spurious edges being detected. This would lead to a less accurate edge map, with many false edges. However, with double thresholding, it's also possible that false edges can still be excluded as edges in blurred areas are likely going to be filtered out. In summary, not performing Gaussian blur before canny edge

detection would lead to a less accurate and noisier edge map. But even after smoothing is applied, the numerical statistical data may not be as appealing as that of other algorithms due to the complexity of the Canny edge detection procedure that will still improve the outcome.

Performing LoG edge detection without Gaussian blur would lead to a far noisier and less accurate output, as the second-derivative approach the Laplacian filter takes is sensitive to noise. The Gaussian blur is also a built-in step in LoG edge detection as it helps to smooth out the image and reduce the impact of noise. Therefore, the application of Gaussian blur would likely greatly improve the effectiveness of reducing false edge detection.

Since Canny edge detection uses the Sobel operator. The effects of smoothing on solely the Sobel operator itself may make the output resemble that of Canny edge detection. Nevertheless, one significant difference between these two approaches is the number of steps involved in each algorithm. Sobel edge detection with Gaussian blur is a simpler algorithm that involves only two steps: blurring the image with a Gaussian filter and then applying the Sobel operator to detect edges. Canny edge detection, on the other hand, is a more complex algorithm that involves multiple steps, including NMS and hysteresis thresholding. Another difference is the type of edges that are detected by each algorithm. Sobel edge detection with Gaussian blur tends to keep all edges detected in an image, including those that are not significant. Canny

edge detection, on the other hand, is more selective in the edges it detects and tends to identify only the most prominent edges in an image. Hence, Canny edge detection is likely producing smoother and more continuous edge maps, with fewer artifacts and false edges. Sobel edge detection with Gaussian blur, on the other hand, can produce a noisier and less accurate edge map, as it is a simpler algorithm. However, the Sobel operator may achieve better numerical results, also due to its lack of complexity.

Furthermore, due to the similar characteristics between the Sobel detector, Pewitt's detector, and Robert cross detector, the numerical data is likely going to be similar.

Results

The following diagram, in alphabetic order of different algorithms, shows the numbers of pixels wrongly taken into account as edges, when Gaussian blur is or is not applied:

Algorithm / Gaussian	Canny Edge Detection without Gaussian Blur	Laplacian	Prewitt's	Roberts Cross	Sobel
Applied	8308	112698	24979	49194	24272
Not Applied	6705	100508	19699	41529	19418

The next diagram shows the reduction of false edge detection in percentage after Gaussian blur is applied.

Algorithm	Canny Edge Detection	Laplacian of Gaussian	Prewitt's with Gaussian	Roberts Cross with Gaussian	Sobel with Gaussian
Reduction	19.3%	10.8%	21.1%	15.6%	20.0%

Evaluation

Statistics

According to the statistics, the algorithm that produced the highest reduction is Prewitt's with Gaussian (21.1%). The second highest reduction is obtained with Sobel with Gaussian(20.0%). Laplacian of Gaussian produced the smallest reduction (10.8%) and Roberts Cross with Gaussian and Canny edge detection with Gaussian in between LoG and Canny (15.6% and 19.3%, respectively).

It is quite surprising that the LoG operator, ended up being the least-enhanced algorithm by numerical data, as smoothing in previous prediction is considered a necessary step for a good output of LoG as it would greatly reduce false edge detection caused by image noise. The current situation is possibly caused by the lack of image noise: in the sample image, there is little, or even no image noise, making the primary reason for false edge detection being gradient values in blurry areas covered within set threshold values.

On the other hand, previous assumptions that the Sobel operator with Gaussian blur may achieve better numerical results than Canny edge detection is confirmed by the result.

While Sobel with Gaussian blur did gain a reduction percentage similar to that of Prewitt’s with Gaussian, the result of Roberts cross with Gaussian varies from the

previous two algorithms. The possible reason for that could be the kernels used: Sobel and Prewitt's both use 3x3 kernels, yet Roberts cross uses 2x2 kernels.

Compared by quantity, the output of Canny edge detection has the least number count of pixels wrongly detected as edges, making it the most suitable detection method for MF peaking.

Visual

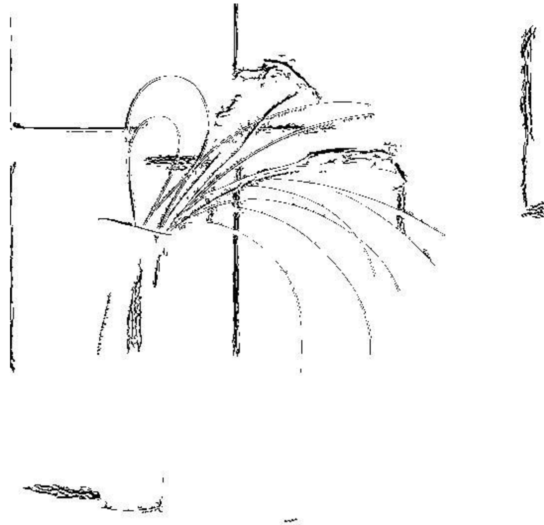

Figure sixteen: output of Canny without smoothing

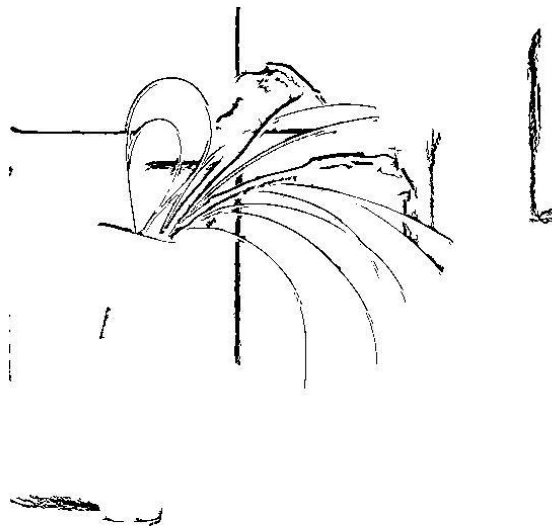

Figure seventeen: output of Canny

In the output of Canny edge detection without smoothing, there are more false edges that are detected, and true edges are disconnected. In the output of Canny edge detection, fewer false edges are detected as a result of Gaussian blur. Furthermore, the effects of edge tracking are also enhanced by smoothing so the edges are better connected, forming consecutive lines. While smoothing also removed some true edges from the objects, such loss in valid information is little and outweighed by the positive effects.

Figure eighteen: output of Laplacian

Figure nineteen: output of LoG

Although LoG received the lowest numerical improvement, the visual outputs indicated a great improvement by Gaussian blur, where the output of LoG has a much clearer silhouette, distinguishing the boundary between clear and blurry areas in the image. The LoG does a better job preserving valid information than any other algorithm, though also leaving the quantity of image noise pixels detected greater than that of any other algorithm as well.

Figure twenty: output of Prewitt's

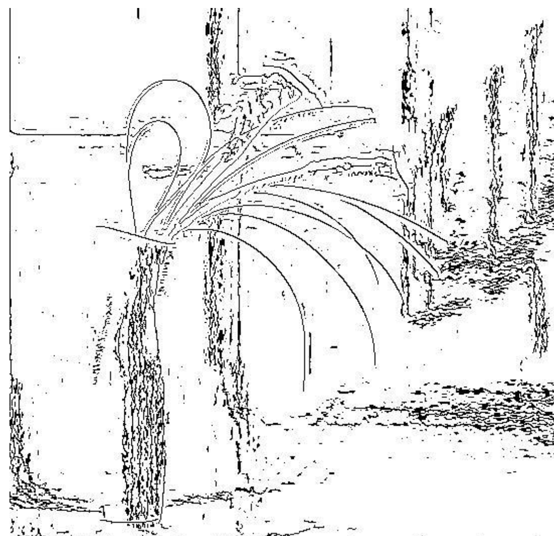

Figure twenty-one: output of Prewitt's with Gaussian blur

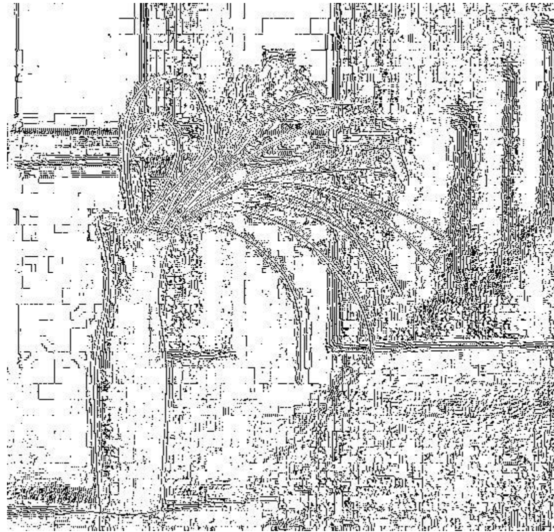

Figure twenty-two: output of Roberts cross

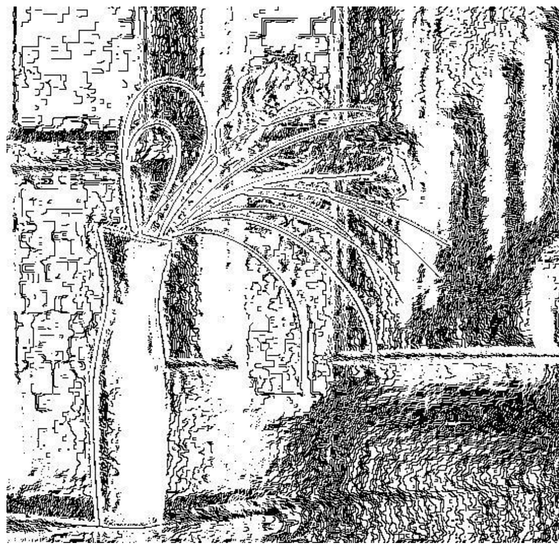

Figure twenty-three: output of Roberts cross with Gaussian blur

Figure twenty-four: output of Sobel

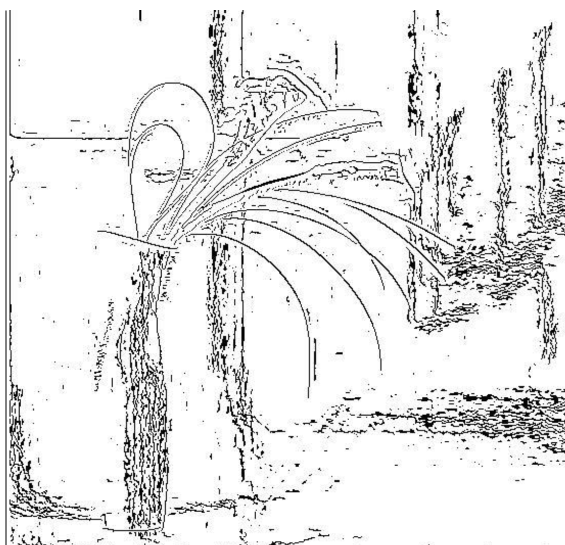

Figure twenty-five: output of Sobel with Gaussian blur

The rest outputs of the three first-derivative operators though varied in numerical data, demonstrated a similar pattern in terms of visualization. The outputs of these operators without smoothing all displayed a significant amount of unorganized false edges in the background, and the distinction between blurry areas and clear areas is not obvious. Such problems are all greatly improved when the input image is smoothed before being processed by these operators: false images, though reduced by a limited quantity, are now more organized, actually showing an understandable forming in the background; the distinction between clear and blur areas are also easier to identify; similar to LoG, lines of edges are now more consecutive likely due to the averaging effect of Gaussian blur. Though there’s a loss in detecting true edges. The final output, in general, is greatly improved.

Methodology

For the methodology of the experiment, several aspects can be further improved.

The code used to carry out different operators can be optimized: the current code poses a limit on the storage size of images, as an image that takes too much storage may increase the number of recursions, leading to stack overflow errors. With this constraint, the image sample has to be compressed so that edge detection can be performed on them.

Furthermore, there are more ways to indicate the effects algorithms have on input. For example, the F1 score is also a measure of a model's accuracy that combines precision (In this case, the proportion of true edge pixels detected among all edge pixels detected) and recall (In this case, the proportion of true edge pixels detected among all actual edge pixels) into a single score.

Finally, the experiment failed to consider one dependent variable that has a practical value in edge detection's real-life application: when edge detection is used as MF peaking on cameras, edges need to be formed in real-time. With too much time consumed to detect the edges, such algorithms can be seen as ineffective solutions to real-life problems. If further investigation is conducted, the time elapsed should also be recorded.

Conclusion

As a practical function, MF peaking serves to provide an effective way to distinguish objects in and out of the focal point. However, MF peaking would also lose its ability to distinguish such areas sometimes, due to discontinuity in color intensity being too great. This essay provides an overview of a solution to this problem as well as several existing edge detection techniques. By experimenting, this essay investigated the effectiveness such solution have on different edge detection algorithms. From the results of the experiment, a conclusion can be drawn, that Prewitt's operator benefits the most from smoothing, and the Canny edge detection, with the least false edges detected, becomes the most suitable technique for MF peaking.

Bibliography

[1] - CANON INC.2019. (2019). PowerShot G7 X Mark III Advanced User Guide [Press release].
https://gdlp01.c-wss.com/gds/8/0300035728/04/psg7x-mk3-ug4-en.pdf

[2] - Nakahara, K. (2021, March 2). Focus Guide & MF Peaking: Making Manual Focus Easier. SNAPSHOT. Retrieved September 20, 2022, from
https://snapshot.canon-asia.com/reg/article/eng/focus-guide-mf-peaking-making-manual-focus-easier

[3] - Ziou, D., & Tabbone, S. (1998). Edge detection techniques-an overview. Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii, 8, 537-559.

[4] - Johnson, S. (2006). Stephen Johnson on Digital Photography. Van Duuren Media.

[5] - (2019, February 6). just hold there, with peak MF level anywhere is in focus, and with peak MF level nowhere is in focus, I am used to zooming in on the focus [Comment on “The MF peaking that will change your life, a must-see tip for novice photographers!”]. https://www.bilibili.com/video/BV1Xb411k7AL

[6] - Saravanan, C. (2010, March). Color image to grayscale image conversion. In 2010 Second International Conference on Computer Engineering and Applications (Vol. 2, pp. 196-199). IEEE.

[7] - Verma, R., & Ali, J. (2013). A comparative study of various types of image noise and efficient noise removal techniques. International Journal of advanced research in computer science and software engineering, 3(10).

[8] - Owotogbe, J. S., Ibiyemi, T. S., & Adu, B. A. (2019). A comprehensive review on various types of noise in image processing. int. J. Sci. eng. res, 10(11), 388-393.

[9] - Antoniadis, P. (n.d.). How to Convert an RGB Image to a Grayscale. Baeldung. https://www.baeldung.com/cs/convert-rgb-to-grayscale

[10] - Gonzalez, R. C., & WOODS 3rd, R. E. (2008). Edition. Digital Image Processing. Upper Saddle River, USA: Prentice Hall.

[11] - Kernel (image processing). (2012). In Wikipedia. Retrieved September 22, 2022, from https://en.wikipedia.org/wiki/Kernel_(image_processing)

[12] - Wikipedia contributors. (2023, February 4). Convolution. Wikipedia. https://en.wikipedia.org/wiki/Convolution

[13] - J. (n.d.). EdgeDetector/detectors at master · JasonAltschuler/EdgeDetector. GitHub. https://github.com/JasonAltschuler/EdgeDetector/tree/master/detectors

[14] - LEE, J. S. (1983). Digital Image Smoothing and the Sigma Filter. COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING, 24, 255-269.

[15] - (no date) Visual China Group. Available at: httpswww.vcg.comcreative806935445.jpg (Accessed: October 9, 2022).

Appendix

Sample Image[32]

[32] (no date) Visual China Group. Available at: httpswww.vcg.comcreative806935445.jpg (Accessed: October 9, 2022).

Code[33] for Edge Detection Algorithms

Canny edge detector:

```
package edgedetector.detectors;


import java.awt.image.BufferedImage;

import java.io.File;

import java.io.IOException;

import java.util.HashSet;

import java.util.Stack;


import javax.imageio.ImageIO;


import kmeans.KMeans;

import ui.ImageViewer;

import util.CSVwriter;

import util.Hypotenuse;

import util.Threshold;
```

[33] J. (n.d.). EdgeDetector/detectors at master · JasonAltschuler/EdgeDetector. GitHub.

https://github.com/JasonAltschuler/EdgeDetector/tree/master/detectors

```
import edgedetector.imagederivatives.ConvolutionKernel;

import edgedetector.imagederivatives.ImageConvolution;

import edgedetector.util.NonMaximumSuppression;

import grayscale.Grayscale;


public class CannyEdgeDetector {

    private static final double[][] X_KERNEL = {{-1, 0, 1},

    {-2, 0, 2},

    {-1, 0, 1}};

    private static final double[][] Y_KERNEL = {{1, 2, 1},

    {0, 0, 0},

    {-1, -2, -1}};


    private boolean L1norm;


    private boolean calcThreshold;


    private int highThreshold;


    private int lowThreshold;
```

```
private int minEdgeSize;

private boolean[][] edges;

private boolean[][] strongEdges;

private boolean[][] weakEdges;

private int numEdgePixels;

private int numStrongEdgePixels;

private int numWeakEdgePixels;

private int rows;
private int columns;

private CannyEdgeDetector() {
}
```

```
private CannyEdgeDetector(Builder builder) {
    // set user information from builder
    this.L1norm = builder.L1norm;
    this.minEdgeSize = builder.minEdgeSize;
    if (!(this.calcThreshold = builder.calcThreshold)) {
        this.lowThreshold = builder.lowThreshold;
        this.highThreshold = builder.highThreshold;
    }
    findEdges(builder.image);
}

public static class Builder {
    private int[][] image;
    private boolean calcThreshold = true;
    private int lowThreshold;
    private int highThreshold;
    private boolean L1norm = false;
    private int minEdgeSize = 0;
```

```
        public Builder(int[][] image) {
            this.image = image;
        }

        public Builder thresholds(int lowThreshold, int highThreshold) {
            if (lowThreshold > highThreshold || lowThreshold < 0 ||
highThreshold > 255) {
                throw new IllegalArgumentException("Invalid threshold values");
            }
            this.calcThreshold = false;
            this.lowThreshold = lowThreshold;
            this.highThreshold = highThreshold;
            return this;
        }

        public Builder L1norm(boolean L1norm) {
            this.L1norm = L1norm;
            return this;
        }
```

```
            public Builder minEdgeSize(int minEdgeSize) {

                  this.minEdgeSize = minEdgeSize = 0;

                  return this;

            }



            public CannyEdgeDetector build() {

                  return new CannyEdgeDetector(this);

            }

      }



      private void findEdges(int[][] image) {



            ImageConvolution gaussianConvolution = new ImageConvolution(image,
ConvolutionKernel.GAUSSIAN_KERNEL);

            int[][] smoothedImage = gaussianConvolution.getConvolvedImage();



            ImageConvolution x_ic = new ImageConvolution(smoothedImage,
X_KERNEL);

            ImageConvolution y_ic = new ImageConvolution(smoothedImage,
Y_KERNEL);
```

```
        int[][] x_imageConvolution = x_ic.getConvolvedImage();
        int[][] y_imageConvolution = y_ic.getConvolvedImage();

        rows = x_imageConvolution.length;
        columns = x_imageConvolution[0].length;

        int[][] mag = new int[rows][columns];
        NonMaximumSuppression.EdgeDirection[][] angle = new
    NonMaximumSuppression.EdgeDirection[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                mag[i][j] = hypotenuse(x_imageConvolution[i][j],
y_imageConvolution[i][j]);
                angle[i][j] = direction(x_imageConvolution[i][j],
y_imageConvolution[i][j]);
            }
        }

        edges = new boolean[rows][columns];
```

```
        weakEdges = new boolean[rows][columns];

        strongEdges = new boolean[rows][columns];




        for (int i = 0; i < rows; i++) {

            for (int j = 0; j < columns; j++) {

                if (NonMaximumSuppression.nonMaximumSuppression(mag,
angle[i][j], i, j)) {

                    mag[i][j] = 0;

                }

            }

        }


        if (calcThreshold) {

            int k = 3;

            double[][] points = new double[rows * columns][1];

            int counter = 0;

            for (int i = 0; i < rows; i++) {

                for (int j = 0; j < columns; j++) {

                    points[counter++][0] = mag[i][j];
```

```
            }
        }

        KMeans clustering = new KMeans.Builder(k, points)
                .iterations(10)
                .pp(true)
                .epsilon(.01)
                .useEpsilon(true)
                .build();
        double[][] centroids = clustering.getCentroids();

        boolean b = centroids[0][0] < centroids[1][0];
        lowThreshold = (int) (b ? centroids[0][0] : centroids[1][0]);
        highThreshold = (int) (b ? centroids[1][0] : centroids[0][0]);
    }

    HashSet<Integer> strongSet = new HashSet<Integer>();
    HashSet<Integer> weakSet = new HashSet<Integer>();

    int index = 0;
```

```
numWeakEdgePixels = 0;
numStrongEdgePixels = 0;
for (int r = 0; r < rows; r++) {
    for (int c = 0; c < columns; c++) {
        if (mag[r][c] >= highThreshold) {
            strongSet.add(index);
            strongEdges[r][c] = true;
            numStrongEdgePixels++;
        } else if (mag[r][c] >= lowThreshold) {
            weakSet.add(index);
            weakEdges[r][c] = true;
            numWeakEdgePixels++;
        }
        index++;
    }
}

boolean[][] marked = new boolean[rows][columns];
Stack<Integer> toAdd = new Stack<Integer>();
```

```
            for (int strongIndex : strongSet) {
                  dfs(ind2sub(strongIndex, columns)[0], ind2sub(strongIndex,
columns)[1], weakSet, strongSet, marked, toAdd);

                  if (toAdd.size() >= minEdgeSize) {
                        for (int edgeIndex : toAdd) {
                              edges[ind2sub(edgeIndex, columns)[0]][ind2sub(edgeIndex,
columns)[1]] = true;
                        }
                  }

                  toAdd.clear();
            }
      }

      private void dfs(int r, int c, HashSet<Integer> weakSet, HashSet<Integer>
strongSet, boolean[][] marked, Stack<Integer> toAdd) {

            if (r < 0 || r >= rows || c < 0 || c >= columns || marked[r][c]) {
                  return;
```

```
        }

        marked[r][c] = true;

        int index = sub2ind(r, c, columns);
        if (weakSet.contains(index) || strongSet.contains(index)) {
            toAdd.push(sub2ind(r, c, columns));

            dfs(r - 1, c - 1, weakSet, strongSet, marked, toAdd);
            dfs(r - 1, c, weakSet, strongSet, marked, toAdd);
            dfs(r - 1, c + 1, weakSet, strongSet, marked, toAdd);
            dfs(r, c - 1, weakSet, strongSet, marked, toAdd);
            dfs(r, c + 1, weakSet, strongSet, marked, toAdd);
            dfs(r + 1, c - 1, weakSet, strongSet, marked, toAdd);
            dfs(r + 1, c, weakSet, strongSet, marked, toAdd);
            dfs(r + 1, c + 1, weakSet, strongSet, marked, toAdd);
        }
    }

    private static int[] ind2sub(int index, int columns) {
```

```
        return new int[]{index / columns, index - columns * (index / columns)};
    }

    private static int sub2ind(int r, int c, int columns) {
        return columns * r + c;
    }

    private int hypotenuse(int x, int y) {
        return (int) (L1norm ? Hypotenuse.L1(x, y) : Hypotenuse.L2(x, y));
    }

    private NonMaximumSuppression.EdgeDirection direction(int G_x, int G_y) {
        return NonMaximumSuppression.EdgeDirection.getDirection(G_x, G_y);
    }

    public static double[][] getX_KERNEL() {
        return X_KERNEL;
    }

    public static double[][] getyKernel() {
```

```
        return Y_KERNEL;
    }

    public boolean isL1norm() {
        return L1norm;
    }

    public int getHighThreshold() {
        return highThreshold;
    }

    public int getLowThreshold() {
        return lowThreshold;
    }

    public boolean[][] getEdges() {
        return edges;
    }

    public boolean[][] getStrongEdges() {
```

```
        return strongEdges;
    }

    public boolean[][] getWeakEdges() {
        return weakEdges;
    }

    public int getNumEdgePixels() {
        return numEdgePixels;
    }

    public int getStrongEdgePixels() {
        return numStrongEdgePixels;
    }

    public int getWeakEdgePixels() {
        return numWeakEdgePixels;
    }

    public int getRows() {
```

```
        return rows;
    }


    public int getColumns() {
        return columns;
    }
}
```

Gaussian operator:

```
package edgedetector.detectors;

import edgedetector.imagederivatives.ImageConvolution;
import edgedetector.util.NonMaximumSuppression;
import util.Hypotenuse;
import util.Threshold;

public abstract class GaussianEdgeDetector {

    protected boolean[][] edges;

    protected int threshold;

    protected boolean L1norm;

    protected abstract double[][] getXkernel();
    protected abstract double[][] getYkernel();

    protected void findEdges(int[][] image, boolean L1norm) {
```

```
double[][] x_kernel = getXkernel();
double[][] y_kernel = getYkernel();


ImageConvolution x_ic = new ImageConvolution(image, x_kernel);
ImageConvolution y_ic = new ImageConvolution(image, y_kernel);


int[][] x_imageConvolution = x_ic.getConvolvedImage();
int[][] y_imageConvolution = y_ic.getConvolvedImage();


int rows = x_imageConvolution.length;
int columns = x_imageConvolution[0].length;


int[][] mag = new int[rows][columns];
NonMaximumSuppression.EdgeDirection[][] angle = new
NonMaximumSuppression.EdgeDirection[rows][columns];
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < columns; j++) {
        mag[i][j] = (int) (L1norm ? Hypotenuse.L1(x_imageConvolution[i][j],
y_imageConvolution[i][j]) :
```

```
Hypotenuse.L2(x_imageConvolution[i][j], y_imageConvolution[i][j]));
                    angle[i][j] =
NonMaximumSuppression.EdgeDirection.getDirection(x_imageConvolution[i][j],

y_imageConvolution[i][j]);
                }
            }

            edges = new boolean[rows][columns];
            threshold = Threshold.calcThresholdEdges(mag);
            for (int i = 0; i < rows; i++)
                for (int j = 0; j < columns; j++)
                    edges[i][j] = (mag[i][j] < threshold) ? false :
NonMaximumSuppression.nonMaximumSuppression(mag, angle[i][j], i , j);
    }

        public boolean[][] getEdges() {
            return edges;
    }
```

```
    public int getThreshold() {
        return threshold;
    }

    public boolean getL1norm() {
        return L1norm;
    }
}
```

Laplacian operator:

```
package edgedetector.detectors;



import java.awt.image.BufferedImage;

import java.io.File;

import java.io.IOException;



import javax.imageio.ImageIO;



import ui.ImageViewer;

import util.Threshold;

import edgedetector.imagederivatives.ConvolutionKernel;

import edgedetector.imagederivatives.ImageConvolution;

import grayscale.Grayscale;




public class LaplacianEdgeDetector {

   private boolean[][] edges;



   private int threshold;
```

```
private double[][] kernel = {{-1, -1, -1},
                            {-1, 8, -1},
                            {-1, -1, -1}};

public LaplacianEdgeDetector(String filePath) {
    BufferedImage originalImage;
    try {
        originalImage = ImageIO.read(new File(filePath));
        findEdges(Grayscale.imgToGrayPixels(originalImage));
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public LaplacianEdgeDetector(int[][] image) {
    findEdges(image);
}

private void findEdges(int[][] image) {
```

```
        ImageConvolution gaussianConvolution = new ImageConvolution(image,
ConvolutionKernel.GAUSSIAN_KERNEL);
        int[][] smoothedImage = gaussianConvolution.getConvolvedImage();

        ImageConvolution ic = new ImageConvolution(smoothedImage, kernel);

        int[][] convolvedImage = ic.getConvolvedImage();
        int rows = convolvedImage.length;
        int columns = convolvedImage[0].length;

        threshold = Threshold.calcThresholdEdges(convolvedImage);

        edges = new boolean[rows][columns];
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < columns; j++)
                edges[i][j] = Math.abs(convolvedImage[i][j]) == 0.0;
    }

    public boolean[][] getEdges() {
        return edges;
```

```
    }


    public int getThreshold() {

        return threshold;

    }

}
```

Prewitt’s operator:

```
package edgedetector.detectors;



import grayscale.Grayscale;



import java.awt.image.BufferedImage;

import java.io.File;

import java.io.IOException;



import javax.imageio.ImageIO;



import ui.ImageViewer;

import util.Threshold;




public class PrewittEdgeDetector extends GaussianEdgeDetector {

    private final static double[][] X_kernel = {{-1, 0, 1},

                                                {-1, 0, 1},

                                                {-1, 0, 1}};
```

```
private final static double[][] Y_kernel = {{1, 1, 1},
                                            {0, 0, 0},
                                            {-1, -1, -1}};

 * @Override
 * {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}}
 */
public double[][] getXkernel() {
    return PrewittEdgeDetector.X_kernel;
}

 * @Override
 * {{1, 1, 1}, {0, 0, 0}, {-1, -1, -1}}
 */
public double[][] getYkernel() {
    return PrewittEdgeDetector.Y_kernel;
}

public PrewittEdgeDetector(String filePath) {
    BufferedImage originalImage;
```

```
        try {
            originalImage = ImageIO.read(new File(filePath));
            findEdges(Grayscale.imgToGrayPixels(originalImage), false);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public PrewittEdgeDetector(int[][] image) {
        findEdges(image, false);
    }

    public PrewittEdgeDetector(int[][] image, boolean L1norm) {
        findEdges(image, L1norm);
}
```

Roberts cross operator:

```
package edgedetector.detectors;

import grayscale.Grayscale;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

import ui.ImageViewer;
import util.Threshold;

public class RobertsCrossEdgeDetector extends GaussianEdgeDetector {
    private final static double[][] X_kernel = {{1, 0},
                                                {0, -1}};

    private final static double[][] Y_kernel = {{0, -1},
```

```
                                                  {1, 0}};

  * @Override
  * {{1, 0}, {0, -1}}
  */
public double[][] getXkernel() {
    return RobertsCrossEdgeDetector.X_kernel;
}

  * @Override
  * {{0, -1}, {1, 0}}
  */
public double[][] getYkernel() {
    return RobertsCrossEdgeDetector.Y_kernel;
}

public RobertsCrossEdgeDetector(String filePath) {
    BufferedImage originalImage;
    try {
```

```
            originalImage = ImageIO.read(new File(filePath));

            findEdges(Grayscale.imgToGrayPixels(originalImage), false);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }



    public RobertsCrossEdgeDetector(int[][] image) {

        findEdges(image, false);

    }



    public RobertsCrossEdgeDetector(int[][] image, boolean L1norm) {

        findEdges(image, L1norm);

    }
```

Sobel operator:

```
package edgedetector.detectors;

import grayscale.Grayscale;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

import ui.ImageViewer;
import util.Threshold;

public class SobelEdgeDetector extends GaussianEdgeDetector {
    private final static double[][] X_kernel = {{-1, 0, 1},
                                                {-2, 0, 2},
                                                {-1, 0, 1}};
```

```
private final static double[][] Y_kernel = {{1, 2, 1},
                                            {0, 0, 0},
                                            {-1, -2, -1}};

 * @Override
 * {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}}
 */
public double[][] getXkernel() {
    return SobelEdgeDetector.X_kernel;
}

 * @Override
 * {{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}}
 */
public double[][] getYkernel() {
    return SobelEdgeDetector.Y_kernel;
}

public SobelEdgeDetector(String filePath) {
    BufferedImage originalImage;
```

```
        try {
            originalImage = ImageIO.read(new File(filePath));
            findEdges(Grayscale.imgToGrayPixels(originalImage), false);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public SobelEdgeDetector(int[][] image) {
        findEdges(image, false);
    }

    public SobelEdgeDetector(int[][] image, boolean L1norm) {
        findEdges(image, L1norm);
    }
```