南京信息工程大学
Nanjing University of Information Science & Technology

University of Reading

Digital Image Processing
Final Project Report

Project ID : ☐ **01** ☑ **02**

NUIST Student Number: 202283710048

UoR Student Number: 31808308

Student Name: Pan yuxuan

Image Enhancement and Restoration

Abstract: In the realm of digital image processing, the blur poses a significant challenge to the restoration of photographic clarity, particularly in images of moving subjects such as people. This study presents an approach to address the restoration of blurred image of a boy, emphasizing the importance of accurate and efficient image recovery. This methodology involves the combination of frequency domain analysis by using Fast Fourier Transform algorithm and Histogram Equalization after Wiener filter. The results of our study are more promising than other methods through contrast experiment and sharpness evaluation, with the main findings indicating a substantial enhancement in image quality post-restoration. The restored images show a significant reduction in blur, with improved edge preservation and texture detail. We conclude that our algorithm is effective in mitigating the effects of blurred images.

Keywords: Digital Image Processing; Blurred Image; Image Restoration; Image Enhancement; Fast Fourier Transform Algorithm; Wiener Filter; Histogram Equalization; Sharpness Evaluation

---

1. Introduction

In our modern world filled with cameras and digital images, one common problem keeps troubling both professional photographers and casual users alike: blurry pictures. This issue becomes especially frustrating when we try to capture moving subjects, like children playing or athletes in action. Despite the advanced technology in today's cameras, motion blur [1] remains a persistent challenge that can ruin otherwise perfect shots. The problem is particularly noticeable in low-light conditions or when capturing fast-moving subjects, where even the most sophisticated camera systems struggle to maintain image clarity.

Over the years, scientific researchers have developed various methods to fix blurry images, each with its own strengths and weaknesses. Some approaches focus on using deep learning methods [2] to restore images, leveraging the power of neural networks to learn complex patterns of image degradation. While these methods have shown impressive results, they require numerous datasets to train the model effectively and demand significant computational resources, making them less accessible for real-time applications or resource-constrained environments. On the other hand, traditional image processing methods, while computationally more efficient, often struggle to achieve satisfactory results without additional information about the blur characteristics or the original scene.

Our study takes a practical approach to this problem by combining two techniques in a new way, aiming to bridge the gap between computational efficiency and restoration quality. We begin by using a mathematical tool called Fast Fourier Transform [3] to understand and analyze the type of blur present in the image. This frequency-domain analysis allows us to identify the specific characteristics of the motion blur. Based on this analysis, we generate a Point Spread Function [4] that mathematically describes how the blur has affected the image. Using this function, we apply several filter methods [5], including but not limited to inverse filtering and Wiener filtering, to restore the image. Through systematic comparison of these methods, we identify the most effective approach for each specific case. Finally, we employ Histogram Equalization [6] after the Wiener filter [7] to enhance the visual quality of the restored image, carefully analyzing various parameters to generate the best possible result.

What makes our approach special is its focus on real-world scenarios, particularly images of people in motion. While our primary goal is to solve the problems of motion-blur images, the methodology we've developed is versatile enough to address other types of blurs as well. The combination of frequency-domain analysis and spatial-domain enhancement provides a robust framework that can adapt to different blur conditions. This adaptability, coupled with the method's computational efficiency, makes it particularly suitable for practical applications where both speed and quality are important considerations.

2. Methods

Our approach to restoring blurred images follows a clear, step-by-step process that builds well-established techniques while introducing some innovative combinations. The method is designed to be both effective and practical, working well with different kinds of blur and producing good results without requiring perfect knowledge of prior information. **Figure 1** shows the flow chart of total process.

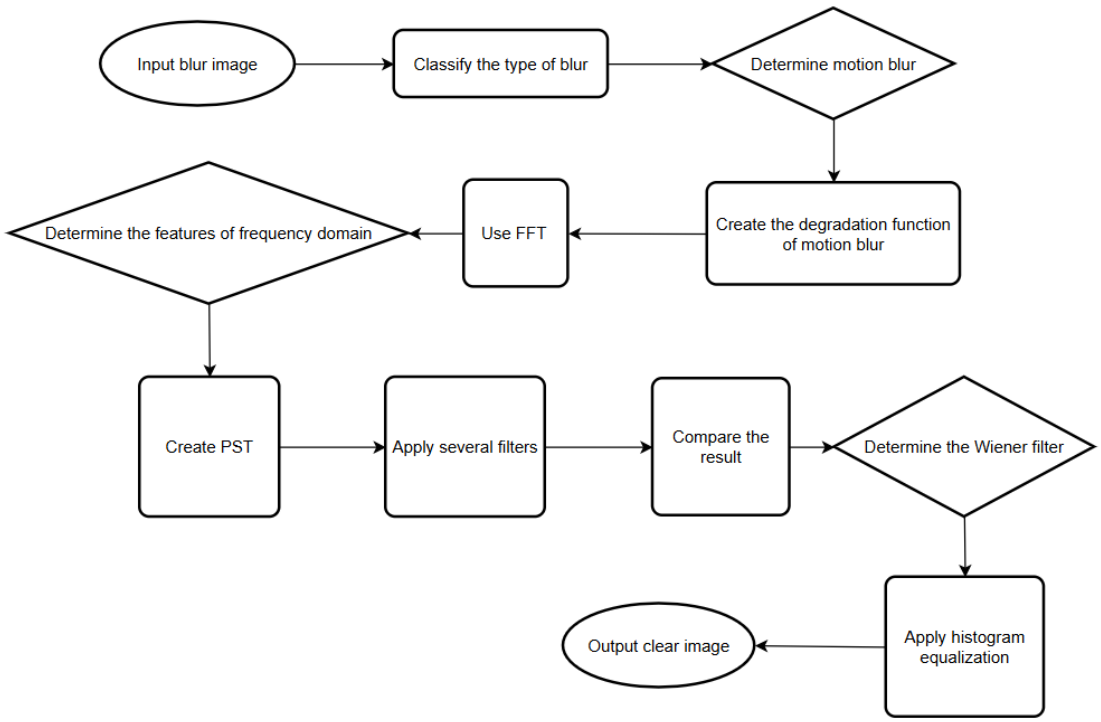


Figure 1. This is a flow chart. It demonstrates all steps during this experiment.

We start by examining the blurred image to understand what kind of blur we're dealing with. This initial analysis is crucial because there is no prior information for reference. Through analysis of different types of blur images and classification of them, we can know that this boy-blur image is probably motion blur. This point will be verified later by analyzing frequency domain. For motion blur, we look for telltale signs like directional smearing – where objects appear stretched in a particular direction. We also examine how edges and textures appear in the image, as motion blur tends to create characteristic patterns in these features. To complement our visual inspection, we use mathematical tools to analyze the blur's characteristics more objectively. This combination of visual and mathematical analysis helps us confirm that we're indeed working with motion blur. We can use some angle and distance measurements tools to determine the range of our parameters.

Once we've identified the blur type, we create a mathematical model that describes how this blur affects the image. Think of this as creating a blueprint of the blur – it helps us understand how the original sharp image got transformed into the blurry version we see. This model needs to capture both the direction and the extent of the blur, which we determine by looking at how the blur affects different parts of the image.

The next step takes us into the frequency domain – a different way of looking at the image that reveals patterns we can't easily see in the normal view. Using a mathematical tool called the Fast Fourier Transform (FFT), we convert the image into this frequency view. The Fast Fourier Transform (FFT) is an efficient algorithm that converts a signal or image from the spatial domain (pixel intensities) to the frequency domain (frequency components). It decomposes an image into a sum of sine and cosine waves of different

frequencies, revealing hidden patterns that are not visible in the spatial domain. In the context of motion blur, FFT helps identify the blur's characteristics by analyzing directional streaks in the frequency spectrum. These streaks are perpendicular to the blur direction, and their spacing indicates the blur length. Here, the motion blur shows up as distinct patterns that tell us about the direction and strength of the blur. This information is crucial for the next steps.

Based on that, we create what's called a Point Spread Function (PSF) – essentially a mathematical description of how a single point of light would get blurred in our image. The Point Spread Function (PSF) is a mathematical model that describes how a single point of light is spread out in an image due to blurring. For motion blur, the PSF is typically a straight line whose direction and length correspond to the blur's direction and extent. The PSF acts as a "blueprint" of the blur process, allowing us to mathematically reverse the blurring through deconvolution. By analyzing the frequency domain (via FFT), we can estimate the PSF's parameters, such as its direction and length. Once the PSF is known, it becomes the foundation for designing restoration filters, such as the Wiener filter, which can effectively recover the original image by counteracting the blur's effects. This PSF becomes the key to reversing the blurring process. We then try three different methods to use this PSF to restore the image:

Inverse Filtering

The inverse filtering approach aims to directly reverse the blurring process by applying the inverse of the degradation function in the frequency domain. Mathematically, this involves dividing the Fourier transform of the blurred image by the Fourier transform of the Point Spread Function (PSF). While theoretically straightforward, this method is highly sensitive to noise, as it tends to amplify high-frequency components, including noise artifacts. To mitigate this issue, a thresholding mechanism is often applied to suppress noise-dominated frequencies, ensuring that the restoration process focuses on meaningful signal components. Despite its limitations, inverse filtering serves as a foundational concept in image restoration, providing insights into the direct reversal of blurring effects.

Constrained Least Squares Filtering

The constrained least squares filtering method addresses the ill-posed nature of image restoration by incorporating regularization techniques. This approach seeks to minimize the difference between the observed blurred image and the restored image while imposing constraints on the solution's smoothness. By introducing a regularization term, the method effectively balances the trade-off between image sharpness and noise suppression. This regularization is particularly valuable in handling noise amplification, a common issue in inverse filtering. The constrained least squares approach is adaptive, allowing it to adjust to local image characteristics, thereby preserving important features while reducing artifacts. This method is widely regarded for its robustness and ability to produce visually plausible results.

Wiener Filtering

Wiener filtering represents a statistically optimal approach to image restoration, minimizing the mean square error between the original and restored images. This method leverages prior knowledge of both the image and noise statistics to design a frequency-dependent filter. By incorporating the power spectra of the original image and the noise, the Wiener filter adaptively adjusts its response to different frequency components, enhancing signal recovery while suppressing noise. This adaptive nature makes Wiener filtering particularly effective in scenarios where noise characteristics are non-uniform or partially known. The method's ability to predict the original image based on statistical properties makes it a powerful tool for achieving a balanced restoration, combining sharpness with noise reduction in a computationally efficient manner.

Through comprehensive evaluation of the three filtering methods, Wiener filtering was identified as the most effective approach, consistently yielding superior results in terms of qualitative visual assessment. Because the result of sharpness evaluation showed that the Wiener filtering has the highest scores. The Wiener filter's ability to adaptively balance noise suppression and detail preservation makes it particularly suitable for motion blur restoration.

To further enhance the restored image, a contrast adjustment technique known as histogram equalization was applied. This post-processing step aims to optimize the image's intensity distribution, thereby improving overall visibility and enhancing fine details. Histogram equalization works by redistributing pixel intensities across the available dynamic range, effectively increasing local contrast in regions where details may have been obscured. However, to prevent over-enhancement and the introduction

of artifacts, a controlled approach was implemented, ensuring that the natural appearance of the image was preserved.

The combination of Wiener filtering and histogram equalization provides a robust framework for image restoration, effectively addressing both the removal of motion blur and the enhancement of visual quality.

3. Experiments

This section provides a detailed description of the experimental methodology, results, and conclusions. The experiments are structured to address blur restoration and enhancement in images, following a systematic workflow.

3.1. Blur Identification and Degradation Modeling

3.1.1. Blur Type Classification

The objective of this step was to identify the type of blur present in the input image. A dataset of images with various types was collected, and a classification method was used to analyze the input image. The result indicated that the input image suffered from motion blur, characterized by a linear trajectory of pixel displacement. **Figure 2** shows some samples in the dataset.

(a) **(b)** **(c)**

Figure 2. This is a figure showing three common types of blurs: (a)Motion blur. (b)Atmospheric blur(heatwave). (c)Defocus blur.

3.1.2. Degradation Function and Frequency Domain Analysis

The goal here was to model the motion blur degradation process and analyze its frequency domain characteristics. The Fast Fourier Transform (FFT) was then applied to convert the degraded image into the frequency domain, revealing the direction and length of the motion blur pulse. The degradation function accurately simulated the motion blur effect, and the frequency domain analysis provided clear insights into the blur trajectory. **Function (1)** is the degradation function. where u and v are frequency coordinates, a and b represent motion velocities in the horizontal and vertical directions, and T is the exposure time.

$$H(u,v) = \frac{T}{\pi(ua+vb)} \sin[\pi(ua+vb)]\, e^{-j\pi(ua+vb)} \tag{1}$$

3.2. Image Restoration and Enhancement

3.2.1. PSF Creation

The objective of this step was to restore the degraded image using a Point Spread Function (PSF) and compare the effectiveness of different filters. Based on the frequency domain analysis, a PSF was created to model the motion blur kernel. L is the length and θ is the angle of the motion blur. The features of PSF can be known from the frequency domain which is as shown in **Figure 3**.

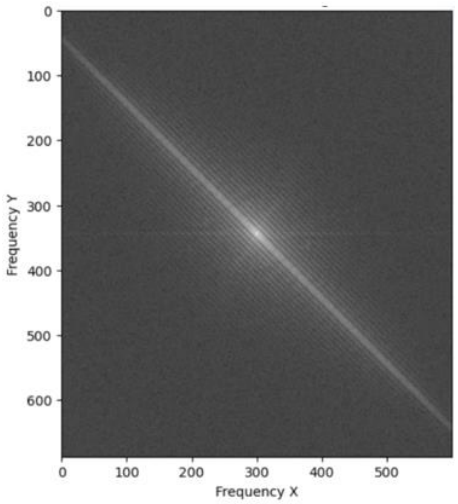


Figure 3. This is a figure showing the image of frequency domain.

3.2.2. Restoration and Enhancement

Three filters were applied: the inverse filter, which directly inverts the degradation function but is sensitive to noise; the constrained least squares filter, which minimizes noise amplification during restoration; and the Wiener filter, which optimizes the trade-off between noise reduction and image restoration. The Wiener filter outperformed the others in noise suppression and detail preservation. Then we can add Histogram Equalization method to enhance this image. **Figure 4** shows the image results and **Table 1** shows the evaluation of each result by using Mean Gradient Magnitude.

Image	Mean Gradient Magnitude
Original	12.2187
Inverse	34.2198
Constrained Least Squares	37.2785
Wiener	75.6408
Wiener + HE	102.2973

Table 1. This is a table which shows the MGM of different results.

(a) **(b)** **(c)** **(d)**

Figure 4. This figure shows four results of different filters: (a)Inverse. (b)CLS. (c)Wiener. (d)Wiener + HE

4. Discussion

In this study, we systematically evaluated the performance of different filters for motion blur restoration and employed the mean gradient magnitude method to assess the effectiveness of restoration and enhancement. The results demonstrated that the combination of Wiener filtering and histogram equalization achieved a satisfactory outcome in processing the motion-blurred image. The Wiener filter, known for its ability to balance noise suppression and detail preservation, outperformed other filters such as the inverse filter and the constrained least squares filter. This finding aligns with previous studies that highlight the Wiener filter's robustness in handling degraded images with noise. Furthermore, the application of histogram equalization significantly enhanced the visual quality of the restored image by improving contrast and brightness, which is consistent with the broader literature on post-processing techniques in image restoration.

A key of this work lies in the effective utilization of the Point Spread Function (PSF) and Fast Fourier Transform (FFT) to analyze the blur type and infer the degradation function. By constructing a PSF based on the motion direction and length observed in the frequency domain, we were able to accurately model the motion blur degradation process. This approach not only provided a clear understanding of the blur characteristics but also laid a solid foundation for the subsequent restoration steps.

In a broader context, the proposed methodology can be extended to address more complex image restoration challenges. For instance, in cases where the blur type is unknown or involves multiple degradation factors, machine learning-based approaches could be employed to enhance the robustness and accuracy of blur identification and restoration. Recent advances in deep learning, particularly convolutional neural networks (CNNs), have shown great potential in handling complex and diverse degradation scenarios. Future research could explore the development of hybrid models that combine traditional image processing techniques with deep learning algorithms to achieve superior restoration performance. Additionally, the application of these techniques in real-time systems, such as video restoration or autonomous driving, could further expand their practical relevance and impact.

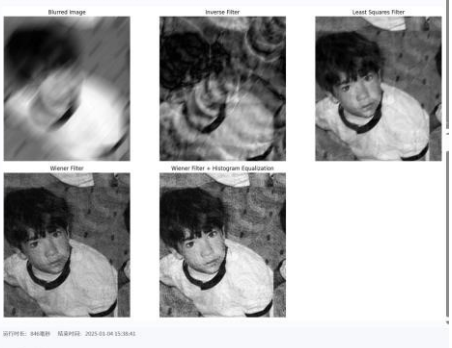
References

1. **Jia, J.** Single image motion deblurring using transparency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007, pp. 1-8.
2. **Kupyn, O.; Budzan, V.; Mykhailych, M.; Mishkin, D.; Matas, J.** DeblurGAN: Blind motion deblurring using conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 8183-8192.
3. **Cooley, J.W.; Tukey, J.W.** An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 1965, 19, 297-301.
4. **Goodman, J.W.** *Introduction to Fourier Optics*. McGraw-Hill, 1996.
5. **Buades, A.; Coll, B.; Morel, J.M.** A non-local algorithm for image denoising. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005, pp. 60-65.
6. **Stark, J.A.** Adaptive image contrast enhancement using generalizations of histogram equalization. *IEEE Transactions on Image Processing*, 2000, 9(5), 889-896.
7. **Wiener, N.** *Extrapolation, interpolation, and smoothing of stationary time series with engineering applications*. MIT Press, 1949.

Appendix：

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

# Generate the Point Spread Function (PSF) for motion blur
def getMotionDsf(img, angle, dist):
    xCenter = img.shape[0] // 2
    yCenter = img.shape[1] // 2
    sinVal = np.sin(angle * np.pi / 180)
    cosVal = np.cos(angle * np.pi / 180)
    PSF = np.zeros(img.shape[:2])
    for i in range(dist):
        xOffset = round(sinVal * i)
        yOffset = round(cosVal * i)
        if (xCenter - xOffset >= 0 and xCenter - xOffset < img.shape[0] and
            yCenter + yOffset >= 0 and yCenter + yOffset < img.shape[1]):
            PSF[int(xCenter - xOffset), int(yCenter + yOffset)] = 1
    return PSF / PSF.sum()

# Wiener filter
def wienerFilter(img, PSF, eps, K):
    fftImg = np.fft.fft2(img)
    fftPSF = np.fft.fft2(PSF) + eps
    fftWiener = np.conj(fftPSF) / (np.abs(fftPSF)**2 + K)
    imgWienerFilter = np.fft.ifft2(fftImg * fftWiener)
    imgWienerFilter = np.abs(np.fft.fftshift(imgWienerFilter))
    return imgWienerFilter

# Inverse filtering
def inverseFilter(img, PSF, eps=1e-3):
    fftImg = np.fft.fft2(img)
    fftPSF = np.fft.fft2(PSF) + eps
    imgInverseFilter = np.fft.ifft2(fftImg / fftPSF)
    imgInverseFilter = np.abs(np.fft.fftshift(imgInverseFilter))
    return imgInverseFilter

# Least squares constraint (pseudo-inverse filtering)
def leastSquaresFilter(img, PSF, eps=1e-3):
    fftImg = np.fft.fft2(img)
    fftPSF = np.fft.fft2(PSF) + eps
    fftLS = np.conj(fftPSF) / (np.abs(fftPSF)**2 + eps)
    imgLeastSquares = np.fft.ifft2(fftImg * fftLS)
    imgLeastSquares = np.abs(np.fft.fftshift(imgLeastSquares))
    return imgLeastSquares

# Histogram equalization
def histogram_equalization(img):
    return cv2.equalizeHist(img)

# Sharpness evaluation (based on gradient)
def calculate_sharpness(image):
    # Calculate the gradient of the image (Sobel operator)
    gradient_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    gradient_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    # Return the mean of the gradient as the sharpness metric
    return np.mean(gradient_magnitude)

# Read the blurred image
blurred_image = cv2.imread("data/boy-blurred.tif", 0)  # Read the image in grayscale mode

# Blur direction and length
angle = 45  # Blur direction (45 degrees)
length = 91  # Blur length (91 pixels)

# Generate PSF
PSF = getMotionDsf(blurred_image, angle, length)

# Apply Inverse Filtering
imgInverseFilter = inverseFilter(blurred_image, PSF)

# Apply Least Squares Filtering
imgLeastSquares = leastSquaresFilter(blurred_image, PSF)

# Apply Wiener Filtering
imgWienerFilter = wienerFilter(blurred_image, PSF, eps=1e-5, K=0.000005)

# Normalize Wiener Filtered Image to the range [0, 255] and convert to uint8
imgWienerFilter = cv2.normalize(imgWienerFilter, None, 0, 255, cv2.NORM_MINMAX)
imgWienerFilter = imgWienerFilter.astype(np.uint8)  # Ensure the image is uint8

# Apply Histogram Equalization to Wiener Filtered Image
imgWienerEqualized = histogram_equalization(imgWienerFilter)

# Ensure all images are of type uint8
imgInverseFilter = cv2.normalize(imgInverseFilter, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
imgLeastSquares = cv2.normalize(imgLeastSquares, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

# Evaluate image quality
def evaluate_image_quality(image, title):
    sharpness = calculate_sharpness(image)
    print(f"{title} - Sharpness: {sharpness:.4f}")

# Evaluate the blurred image
evaluate_image_quality(blurred_image, "Blurred Image")

# Evaluate the Inverse Filtered Image
evaluate_image_quality(imgInverseFilter, "Inverse Filtered Image")

# Evaluate the Least Squares Filtered Image
evaluate_image_quality(imgLeastSquares, "Least Squares Filtered Image")

# Evaluate the Wiener Filtered Image
evaluate_image_quality(imgWienerFilter, "Wiener Filtered Image")

# Evaluate the Wiener Filtered + Histogram Equalized Image
evaluate_image_quality(imgWienerEqualized, "Wiener Filtered + Histogram Equalized Image")

# Display the images
plt.figure(figsize=(15, 10))

# Display the blurred image
plt.subplot(2, 3, 1)
plt.imshow(blurred_image, cmap='gray')
plt.title('Blurred Image')
plt.axis('off')

# Display the Inverse Filtered Image
plt.subplot(2, 3, 2)
plt.imshow(imgInverseFilter, cmap='gray')
plt.title('Inverse Filter')
plt.axis('off')

# Display the Least Squares Filtered Image
plt.subplot(2, 3, 3)
plt.imshow(imgLeastSquares, cmap='gray')
plt.title('Least Squares Filter')
plt.axis('off')

# Display the Wiener Filtered Image
plt.subplot(2, 3, 4)
plt.imshow(imgWienerFilter, cmap='gray')
plt.title('Wiener Filter')
plt.axis('off')

# Display the Wiener Filtered + Histogram Equalized Image
plt.subplot(2, 3, 5)
plt.imshow(imgWienerEqualized, cmap='gray')
plt.title('Wiener Filter + Histogram Equalization')
plt.axis('off')

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

```
Blurred Image - Sharpness: 12.2187
Inverse Filtered Image - Sharpness: 34.2198
Least Squares Filtered Image - Sharpness: 17.2785
Wiener Filtered Image - Sharpness: 75.6888
Wiener Filtered + Histogram Equalized Image - Sharpness: 102.2073
```



运行时长: 84毫秒 结束时间: 2025-03-04 15:38:41

Image Segmentation and Morphology

Abstract: Image segmentation and morphology are fundamental techniques in digital image processing, playing a crucial role in object detection and boundary delineation. This study focuses on the detection and separation of circles of varying sizes, particularly large circles, small circles, and those in contact with image boundaries, using the Hough Transform. The Hough Transform is employed to identify circular shapes within complex images, while morphological operations such as erosion, opening, and closing are utilized to refine the segmentation process. Specifically, these operations help distinguish boundaries between large and small circles, enhancing the accuracy of the segmentation results. The proposed method demonstrates robustness in handling overlapping and boundary-contacting circles, which are common challenges in image analysis. Experimental results validate the effectiveness of the approach, showing improved precision in circle detection and boundary division.

Keywords: Image Segmentation; Hough Transform; Morphological Operation; Circle Detection; Boundary Division

---

1. Introduction

Image segmentation [1] and morphological processing are crucial research directions in the field of digital image processing, with wide applications in medical image analysis, object detection, and computer vision. In these applications, accurately distinguishing different objects and extracting their boundary information in images is a significant challenge. Particularly in complex scenes involving overlapping objects or objects touching image boundaries. This study focuses on a specific scenario: an image containing numerous circular objects, where large circles are located on the periphery and small circles are enclosed within. To address this problem, we propose a segmentation method that combines the Hough transform and morphological operations, aiming to efficiently detect large circles, small circles, and circles touching the image boundary, as well as to precisely delineate the boundaries between large and small circles.

The significance of this study lies in its ability to handle object segmentation in these kinds of scenes, especially when objects are in contact or overlapping. The innovation of this work is the adoption of a phased processing strategy, integrating edge detection, the Hough transform [2], and morphological operations to achieve efficient segmentation of circular objects in complex scenes. Specifically, the proposed process consists of four parts: first, detecting circles touching the image boundary using Canny edge detection [3] and contour analysis; second, detecting all small circles using the Hough circle transform; third, extracting large circles by excluding small circles and those touching the boundary from the complete set of circles; and finally, precisely delineating the boundaries between large and small circles through morphological operations [4] such as erosion, opening, and closing.

This study presents a feasible solution for addressing image segmentation challenges in circles scenes by integrating edge detection, the Hough transformation, and morphological operations. Experimental results demonstrate that the proposed method achieves better performance than single approach in terms of both accuracy and computational efficiency. Through this work, we aim to provide a practical technical reference for the application of image segmentation and morphological processing techniques.

2. Methods

Image segmentation and morphology is one of the core tasks in digital image processing, aiming to divide an image into regions or objects with specific meanings. In complex scenes, such as images containing numerous overlapping or boundary-touching circular objects, traditional segmentation methods often struggle to achieve satisfactory results. This study focuses on a specific scenario: an input image

containing numerous circular objects, where large circles are located on the periphery and small circles are enclosed within. To achieve precise segmentation of this scenario, a phased processing approach is proposed, combining edge detection, the Hough transform, and morphological operations to progressively detect targets and delineate boundaries. Specifically, the proposed method consists of four parts: detecting circles touching the image boundary, detecting small circles, detecting large circles (excluding those touching the boundary), and delineating boundaries between large and small circles. Each part is described in detail below.

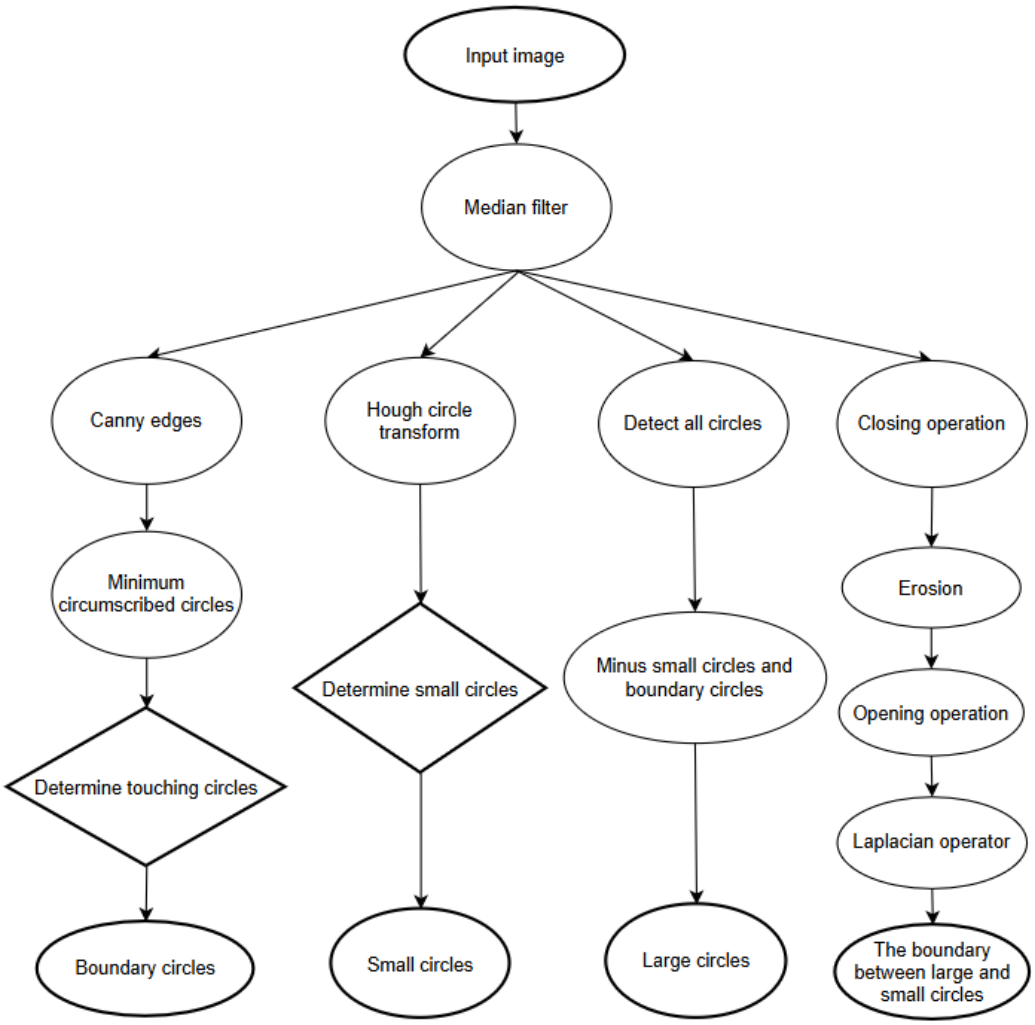


Figure 1. This is a flow chart. It demonstrates all steps during this experiment.

First, the input image is preprocessed. The color image is converted to grayscale to reduce computational complexity while retaining key structural information. Then, median filtering is applied to remove salt-and-pepper noise from the image, thereby improving the accuracy of subsequent edge detection. Next, the Canny edge detection algorithm is used to extract edge information from the image. The Canny algorithm, through dual-threshold detection and edge linking, effectively extracts significant edges in the image. After obtaining edge information, contour detection is used to extract all contours, and the minimum enclosing circle is calculated for each contour. The minimum enclosing circle provides a good fit for the geometric shape of the contour, especially for near-circular objects. By determining whether each minimum enclosed circle touches the image boundary (i.e., whether the distance from the center to the boundary is less than the radius), circles touching the boundary can be accurately identified. These circles are often incomplete, possibly truncated due to image acquisition or target distribution. Finally, the detected boundary-touching circles are labeled and drawn on the image for further processing in subsequent steps.

In the second part, the Hough Circle Transform is used to detect all small circles in the image. The Hough Circle Transform is a parameter space-based method for detecting circular objects, capable of effectively identifying circular targets in images. First, the input image is preprocessed, including grayscale

conversion and Gaussian filtering, to reduce noise interference in the detection results. Then, the parameter range for the Hough Circle Transform is set to ensure only small circles are detected. The Hough Circle Transform, through an accumulator space voting mechanism, robustly detects small circles in the image, even in the presence of partial overlaps or noise. Detected small circles must satisfy the condition that both their centers and circumferences lie within the image boundaries to ensure target integrity. For each detected small circle, its center and contour are drawn, and it is labeled as a "small circle" for further processing in subsequent steps.

The goal of the third part is to extract large circles from all detected circles while excluding those touching the boundary. First, contour detection is used to detect all circles in the image, and the minimum enclosing circle is calculated for each contour. This step ensures comprehensive coverage of circular objects in the image, including both large and small circles. Then, small circles detected in the second part and "boundary-touching circles" labeled in the first part are excluded from the complete set of circles. The exclusion operation is achieved by comparing center positions and radii, ensuring only complete large circles are retained. The key to this step lies in accurately distinguishing large circles from small circles and excluding boundary-touching circles to avoid misclassification. Finally, the detected large circles are labeled and drawn on the image, providing a foundation for subsequent boundary delineation.

Finally, morphological operations are used to delineate boundaries between large and small circles. Morphological operations are commonly used techniques in image processing, capable of locally modifying images through the shape and size of structuring elements. First, a closing operation is applied to the input image to remove small circles inside. The closing operation, consisting of dilation followed by erosion, fills gaps between small circles, thereby separating large and small circles. Next, an erosion operation is performed to make large circle regions more prominent. The erosion operation, by eliminating boundary pixels, effectively shrinks target regions and removes small-area noise. Then, an opening operation is applied to remove noise. The opening operation, consisting of erosion followed by dilation, smooths boundaries and removes small-area noise, further optimizing the segmentation results. Finally, boundaries are extracted using the Laplacian operator and overlaid on the original image in red. The Laplacian operator highlights edge information in the image, clearly delineating boundaries between large and small circles. The key to this step lies in selecting appropriate morphological operations and structuring element sizes to ensure accurate and robust boundary delineation.

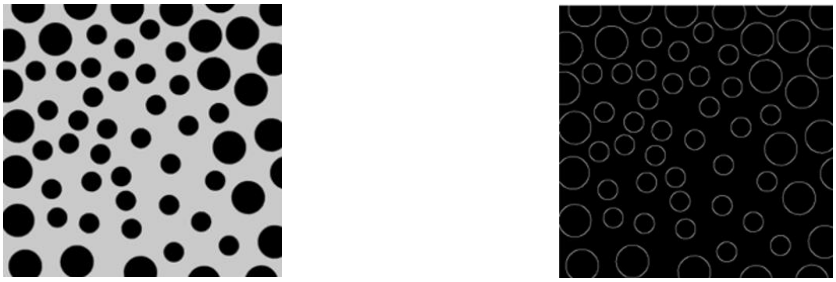
3. Experiments

This section provides a detailed description of the experimental design, implementation, and analysis of the results. The experiments are divided into four parts: detecting circles touching the image boundary, detecting small circles, detecting large circles (excluding those touching the boundary), and delineating boundaries between large and small circles. Each part is implemented through code, and the effectiveness of the methods is validated through visual results.

3.1. Detecting Circles Merged with the Image Boundary

3.1.1. Image Preprocessing and Edge Detection

The input image is converted to grayscale to reduce computational complexity. Median filtering (5x5 kernel) is applied to remove salt-and-pepper noise, which improves the accuracy of subsequent edge detection. **Figure 2 (a)** shows the image after processing. The Canny edge detection algorithm is used to extract edge information from the image. The algorithm employs dual-threshold detection and edge linking to identify significant edges. **Figure 2 (b)** shows the Canny edge detection result.



(a) **(b)**

Figure 2. This is a figure. (a)result of grayscale converting and median filtering. (b)result of Canny edges

3.1.2. Contour Detection and Minimum Enclosing Circle Calculation

Contours are detected from the edge map, and the minimum enclosing circle is calculated for each contour. This step ensures that the geometric shape of each contour is approximated by a circle.

3.1.3. Boundary Contact Detection

For each minimum enclosing circle, it is determined whether the circle touches the image boundary by checking if the distance from the circle's center to the boundary is less than its radius. Circles touching the boundary are labeled as "boundary-touching circles." **Figure 3** shows the result.

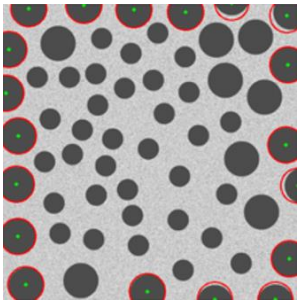


Figure 3. This figure shows all the circles which merged with the boundary.

3.2. Detecting Small Circles

Hough Circle Transform and Circle Validation: The Hough Circle Transform is used to detect circular objects in the image. The parameters are set to ensure that only small circles are detected. Detected circles are validated to ensure that both their centers and circumferences lie within the image boundaries. **Figure 4** shows the result.

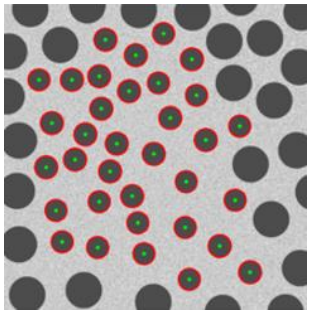
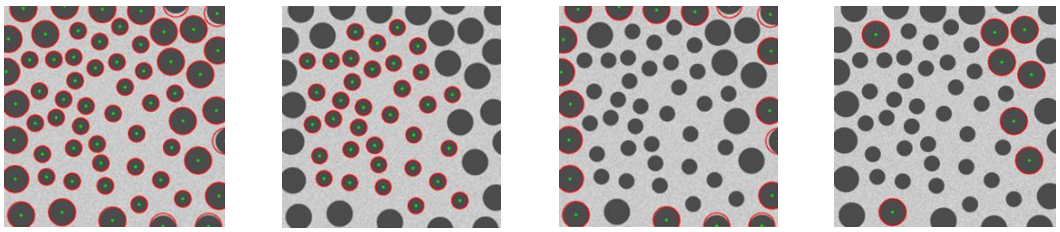


Figure 4. This figure shows all small circles.

3.3. Detecting Large Circles

Contour Detection and Exclusion of Small and Boundary-Touching Circles: Contours are detected from the preprocessed image, and the minimum enclosing circle is calculated for each contour. Small circles detected in the second part and boundary-touching circles labeled in the first part are excluded from the complete set of circles. The centers (green dots) and contours (red lines) of the remaining large circles are drawn on the original image for visualization. The main idea is minus. Using a for-loop to seek out all large circles. **Figure 5** shows the process.

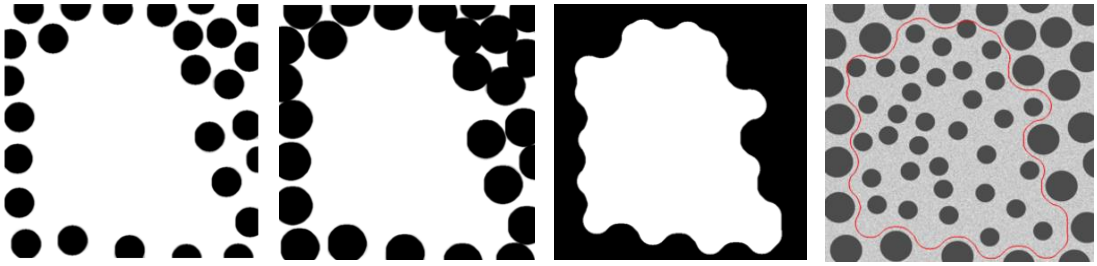


(a) **(b)** **(c)** **(d)**

Figure 5. This is a figure. (a)all circles. (b)small circles. (c)boundary circles. (d)large circles.

3.4. Sketch the Boundary between Small and Large Circles

Morphology Methods to Sketch the Boundary: The final part of the experiment aims to delineate boundaries between large and small circles using morphological operations. The process is implemented as follows: Closing Operation: A closing operation is applied to the input image to remove small circles inside. The operation uses an elliptical structuring element to fill gaps between small circles. Erosion Operation: An erosion operation is performed to make the large circle regions more prominent. This step eliminates boundary pixels and reduces noise. Opening Operation: An opening operation is applied to further remove noise and smooth the boundaries. Binarization: The result is binarized using Otsu's thresholding method to separate the foreground and background. Boundary Extraction: The Laplacian operator is used to extract boundaries, which are then overlaid on the original image in red for visualization. **Figure 6** shows the process.



(a) (b) (c) (d)

Figure 6. This is a figure. (a)closing operation. (b)erosion. (c)opening and binarization. (d)boundary.

4. Discussion

The experimental results demonstrate the effectiveness of the proposed method in detecting and segmenting circular objects in complex images. By combining edge detection, the Hough transform, and morphological operations, the method achieves high accuracy in identifying large circles, small circles, and boundary-touching circles, as well as delineating boundaries between them. This phased approach addresses the limitations of traditional methods, particularly in handling overlapping objects and boundary effects.

The findings align with previous studies that highlight the challenges of circle detection in noisy and complex scenes. While traditional methods often struggle with such scenarios, the proposed method leverages a combination of techniques to improve robustness and accuracy. For example, the use of morphological operations for boundary delineation builds on existing research but introduces a novel sequence of closing, erosion, and opening operations to enhance performance. The method's adaptability also suggests potential for addressing similar problems involving other shapes or more complex geometries.

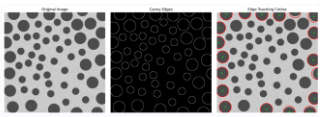
Future research could explore extending the method to handle non-circular shapes, integrating machine learning for improved robustness, and optimizing computational efficiency for real-time applications. Testing on diverse datasets will further validate the method's generalizability and identify areas for improvement. In conclusion, the proposed method provides a robust and accurate solution for detecting and segmenting circular objects in complex images. By addressing the limitations of traditional approaches and leveraging a combination of techniques, this work contributes to the advancement of image segmentation.

References

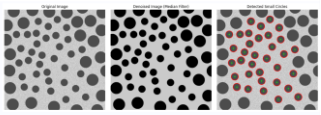
1. **Csurka, G.** Semantic Image Segmentation: Two Decades of Research. *arXiv* 2023, 2302.06378.
2. **Duda, R.O.; Hart, P.E.** Use of the Hough Transformation to Detect Lines and Curves in Pictures. *Commun. ACM* 1972, 15, 11–15.
3. **Canny, J.** A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 1986, 8, 679–698.
4. **Serra, J.** Image Analysis and Mathematical Morphology. *Academic Press* 1982, 1, 1–512.

Appendix：

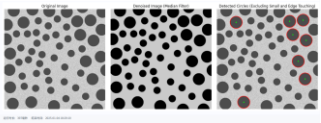
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read .tif file
img_path = 'data/fig4b04.tif' # Replace with your .tif file path
img_original = cv2.imread(img_path, cv2.IMREAD_COLOR) # Read image in color mode

# Convert to grayscale
img_gray = cv2.cvtColor(img_original, cv2.COLOR_BGR2GRAY)

# Remove salt-and-pepper noise (median filtering)
img_denoised = cv2.medianBlur(img_gray, 7) # Use a 5x5 median filter

# Canny edge detection
edges = cv2.Canny(img_denoised, threshold1=30, threshold2=150)

# Find contours
contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Detect incomplete circles
min_radius = 10  # Minimum radius for a short-circle touching the edge
for contour in contours:
    epsilon = 0.01 * cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, epsilon, True)
    if len(approx) > 5:  # Ensure enough points to form a circle
        (x, y), radius = cv2.minEnclosingCircle(contour)
        center = (int(x), int(y))
        radius = int(radius)

        # Check if the circle touches the image edge
        if (x - radius <= 0 or x + radius >= img_gray.shape[1] or  # Circle exceeds image width
            y - radius <= 0 or y + radius >= img_gray.shape[0]):  # Circle exceeds image height
            edge_touching_circles.append((center, radius))
            # Draw the circle on the original image
            cv2.circle(img_original, center, radius, (0, 0, 255), 2)  # Draw circle in green
            cv2.circle(img_original, center, radius, (0, 0, 255), 2)  # Draw circle in red

# Display results
plt.figure(figsize=(15, 5))
images = [img_gray, edges, img_original]
titles = ['Original Image', 'Canny Edges', 'Edge Touching Circles']
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB) if i > 1 else images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.tight_layout()
plt.show()
```




```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read .tif file
img_path = 'data/fig4b04.tif' # Replace with your .tif file path
img_original = cv2.imread(img_path, cv2.IMREAD_COLOR) # Read image in color mode

# Convert to grayscale
img_gray = cv2.cvtColor(img_original, cv2.COLOR_BGR2GRAY)

# Remove salt-and-pepper noise (median filtering)
img_denoised = cv2.medianBlur(img_gray, 7) # Use a 5x5 median filter

# Detect circles using Hough Circle Transform
circles = cv2.HoughCircles(img_denoised, cv2.HOUGH_GRADIENT, dp=1.2, minDist=20, param1=50, param2=30, minRadius=5, maxRadius=15)

# Check if circles are detected
if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        # Draw the outer circle (to complete (center and circumference are within image boundaries)
        h, w = img_gray.shape[:2]
        if 0 <= i[0] < w and 0 < i[1] < h:  # Circle is within image width
            if 0 <= i[0] - i[2] < w and 0 < i[1] - i[2] < h:  # Circle is within image height
                # Draw the circle on the original
                cv2.circle(img_original, (i[0], i[1]), i[2], (0, 0, 0), 2)
                # Draw the center of the circle
                cv2.circle(img_original, (i[0], i[1]), 2, (255, 0, 0), 3)

# Display results
plt.figure(figsize=(15, 5))
images = [img_gray, img_denoised, img_original]
titles = ['Original Image', 'Denoised Image (Median Filter)', 'Detected Small Circles']
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB) if i > 1 else images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.tight_layout()
plt.show()
```




```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read .tif file
img_path = 'data/fig4b04.tif' # Replace with your .tif file path
img_original = cv2.imread(img_path, cv2.IMREAD_COLOR) # Read image in color mode

# Convert to grayscale
img_gray = cv2.cvtColor(img_original, cv2.COLOR_BGR2GRAY)

# Remove salt-and-pepper noise (median filtering)
img_denoised = cv2.medianBlur(img_gray, 7) # Use a 5x5 median filter

# Third code block: detect all circles
def detect_all_circles(img):
    # Apply edge detection
    edges = cv2.Canny(img, threshold1=30, threshold2=150)

    # Find contours
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Store detected circles
    detected_circles = []

    # Detect incomplete circles
    for contour in contours:
        epsilon = 0.01 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        if len(approx) > 5:  # Ensure enough points to form a circle
            (x, y), radius = cv2.minEnclosingCircle(contour)
            center = (int(x), int(y))
            radius = int(radius)
            detected_circles.append((center, radius))

    return detected_circles

# Second code block: Detect small circles
def detect_small_circles(img):
    # Detect circles using Hough Circle Transform
    circles = cv2.HoughCircles(img, cv2.HOUGH_GRADIENT, dp=1.2, minDist=20, param1=50, param2=30, minRadius=5, maxRadius=15)

    # Store detected small circles
    detected_small_circles = []

    if circles is not None:
        circles = np.uint16(np.around(circles))
        for i in circles[0, :]:
            h, w = img.shape[:2]
            if 0 <= i[0] < w and 0 < i[1] < h:
                detected_small_circles.append((i[0], i[1]
                                               ))

    return detected_small_circles

# Third code block: Detect circles touching the image boundary
def detect_edge_touching_circles(img):
    # Apply edge detection
    edges = cv2.Canny(img, threshold1=30, threshold2=150)

    # Find contours
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Store circles touching the edge
    edge_touching_circles = []

    # Detect incomplete circles
    for contour in contours:
        epsilon = 0.01 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        if 5 <= len(approx) < 8:  # If the curve is close to a circle
            (x, y), radius = cv2.minEnclosingCircle(contour)
            center = (int(x), int(y))
            radius = int(radius)

            # Check if the circle touches the image edge
            if (x - radius <= 0 or x + radius >= img.shape[1] or  # Circle exceeds image width
                y - radius <= 0 or y + radius >= img.shape[0]):  # Circle exceeds image height
                edge_touching_circles.append((center, radius))

    return edge_touching_circles

# Detect all circles
all_circles = detect_all_circles(img_denoised)

# Detect small circles
small_circles = detect_small_circles(img_denoised)

# Detect circles touching the image boundary
edge_touching_circles = detect_edge_touching_circles(img_denoised)

# Remove small circles and edge-touching circles from all circles
remaining_circles = []
for circle in all_circles:
    is_small = False
    is_edge_touching = False

    # Check if it's a small circle
    for small_circle in small_circles:
        distance = np.sqrt((circle[0] - small_circle[0]) ** 2 + (circle[1] - small_circle[1]) ** 2)
        if distance < 10 and abs(circle[1] - small_circle[1]) < 5:  # Center distance and radius difference within threshold
            is_small = True
            break

    # Check if it touches the image boundary
    for edge_circle in edge_touching_circles:
        distance = np.sqrt((circle[0] - edge_circle[0]) ** 2 + (circle[1] - edge_circle[1]) ** 2)
        if distance < 10 and abs(circle[1] - edge_circle[1]) < 10:  # Center distance and radius difference within threshold
            is_edge_touching = True
            break

    # If it's neither a small circle nor touches the edge, keep it
    if not is_small and not is_edge_touching:
        remaining_circles.append(circle)

# Draw the remaining circles
for center, radius in remaining_circles:
    cv2.circle(img_original, center, radius, (0, 255, 0), 2)  # Draw center in green
    cv2.circle(img_original, center, radius, (0, 0, 255), 2)  # Draw circle in red

# Display results
plt.figure(figsize=(15, 5))
images = [img_gray, img_denoised, img_original]
titles = ['Original Image', 'Denoised Image (Median Filter)', 'Detected Circles (Excluding Small and Edge Touching)']
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB) if i > 1 else images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.tight_layout()
plt.show()
```




```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read the image
image = cv2.imread('data/fig4b04.tif', cv2.IMREAD_GRAYSCALE)

# Perform closing operation to remove inner small circles
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (25, 25))
closed = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)

# Define circular structure to remove inner small circles
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (25, 25))

# Invert to make the large circle region dark problems
inv = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (25, 25))
_, binary = cv2.threshold(inv, 127, 255, cv2.THRESH_BINARY_INV)

# Perform opening operation to remove noise
opened = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (25, 25))
_, binary = cv2.threshold(opened, 127, 255, cv2.THRESH_BINARY)

# Invert the image
inverted = 255 - cv2.bitwise_not(binary, cv2.THRESH_BINARY_INV)

# Extract boundaries
boundary = cv2.Canny(inverted, 50, 150)
_, thresh = cv2.threshold(boundary, 127, 255, 0)

# Convert grayscale image to color
filled = cv2.cvtColor(thresh, cv2.COLOR_GRAY2BGR)

# Overlay Boundary in red
red_color = (0, 0, 255)  # BGR format, red in (0, 0, 255)
filled[thresh > 0] = red_color  # Set boundary regions to red

# Display results
plt.figure(figsize=(15, 15))

plt.subplot(221), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.axis('off')

plt.subplot(222), plt.imshow(closed, cmap='gray'), plt.title('Closing Operation')
plt.axis('off')

plt.subplot(223), plt.imshow(opened, cmap='gray'), plt.title('Erosion')
plt.axis('off')

plt.subplot(224), plt.imshow(filled, cmap='gray'), plt.title('Opening Operation and Dilatation')
plt.axis('off')

plt.subplot(225), plt.imshow(cv2.cvtColor(filled,cv2.COLOR_BGR2RGB)), plt.title('Draw the Boundary')
plt.axis('off')

plt.show()
```