Edge and Circle Detection: A Comparative Analysis of Different Approaches

Fraidakis Ioannis
Department of Electrical and Computer Engineering
Aristotle University of Thessaloniki

May 2025

Abstract

This report presents the implementation, detailed analysis, and comparative evaluation of fundamental image processing techniques for edge and circle detection. It covers 2D FIR convolution as a foundational operation, edge detection using the Sobel operator and the Laplacian of Gaussian (LoG) operator, and circle detection via the Hough transform. The methodologies are implemented in Python and rigorously tested on the `basketball_large.png` image. The study emphasizes understanding algorithmic principles, the critical impact of parameter tuning on detection results, and a comparative analysis of the Sobel and LoG edge detectors.

1 Introduction

Edge detection and feature extraction are foundational tasks in digital image processing, crucial for identifying significant structural information and enabling higher-level scene understanding and object recognition. This report focuses on the implementation and analysis of key algorithms for these tasks, addressing edge detection (Section 2.2) through gradient-based (Sobel) and Laplacian-based (LoG) methods, and circle detection (Section 2.3) using the Hough transform. This study specifically addresses the following objectives:

- Implement a function for 2D Finite Impulse Response (FIR) convolution, serving as a core component for the edge detection algorithms.
- Develop and implement the Laplacian of Gaussian (LoG) operator for edge detection, including the creation of LoG kernels, adaptive zero-crossing detection, and connected component analysis for noise reduction, while investigating the impact of kernel size.
- Develop and implement the Sobel operator for edge detection, analyzing the effect of varying thresholds and the utility of post-processing steps.
- Implement the Hough transform for detecting circular shapes in binary edge maps
- Demonstrate the implemented functions on a test image (`basketball_large.png`).

2 Theoretical Background

2.1 2D FIR Convolution

Convolution is a fundamental operation in image processing, used for applying filters to images. For a 2D image I and a kernel h (mask), the discrete convolution $(I * h)$ at a point (x, y) is defined as:

$$(f * h)(x, y) = \sum_j \sum_i f(x - i, y - j)h(i, j) \tag{1}$$

The kernel h is often referred to as a Finite Impulse Response (FIR) filter. The origin of the mask and the input image are important for determining the alignment during convolution and the origin of the output image. In this implementation, explicit convolution is performed without relying on FFT-based methods.

2.2 Edge Detection

Edge detection is a fundamental image processing technique that identifies significant local changes in image intensity. These changes typically correspond to boundaries of objects, or changes in surface properties. The primary methods for edge detection are:

- **Laplacian-based methods:** These methods use the second-order derivative of the image intensity. Edges are located at zero-crossings in the output of the Laplacian operator. The Laplacian of Gaussian (LoG) method is a common technique in this category. It first applies a Gaussian smoothing filter to reduce noise and then applies the Laplacian operator to detect regions of rapid intensity change, pinpointing edges at the zero-crossings of the result.
- **Gradient-based methods:** These methods operate by calculating the first-order derivative of the image intensity. Edges are then identified at pixels where the gradient magnitude is maximal. A prominent example is the Sobel operator, which utilizes two 3×3 convolution kernels to approximate the gradient.

2.3 Hough Transform for Circles

The Hough transform is a feature extraction technique adept at identifying instances of specific shapes, such as circles, even when they are imperfect or partially obscured. This method operates by a voting procedure in a parameter space. For circle detection, the parametric equation $(x - a)^2 + (y - b)^2 = r^2$ is utilized, where (a, b) denotes the center coordinates and r is the radius. Edge pixels, identified from a binary edge map, contribute votes to all potential circles they could form part of. This is achieved by iterating through a range of possible radii; for each edge pixel and radius, potential centers (a, b) are computed. These votes are then accumulated in a 3D parameter space, often called Hough space, corresponding to (a, b, r). Peaks in this accumulator, exceeding a predefined minimum vote threshold (V_{min}), signify detected circles.

3 Implementation Framework

3.1 Code Architecture

The project is structured into a collection of Python modules, each dedicated to specific functionalities within the image processing pipeline. This modular design enhances code organization, reusability, and maintainability. The core modules are:

- `fir_conv.py`: This module implements the foundational 2D Finite Impulse Response (FIR) convolution. The primary function, `fir_conv`, is detailed in Section **??**. It serves as a core building block for subsequent edge detection algorithms.
- `log_edge.py`: This module provides the functionality for Laplacian of Gaussian (LoG) edge detection. The main function is `log_edge`, which identifies edges at zero-crossings of the LoG-filtered image. It also includes helper functions such as `create_log_kernel` for generating the LoG filter and `find_zero_crossings` for locating edge pixels. The LoG methodology is further explained in Section **??**.
- `sobel_edge.py`: This module is dedicated to edge detection using the Sobel operator. It contains the `sobel_edge` function, which processes an image to highlight edges based on gradient magnitudes. The algorithmic details are discussed in Section **??**.
- `circ_hough.py`: This module implements the Hough transform specifically tailored for circle detection. The `circ_hough` function analyzes binary edge maps to identify circular shapes. See more in Section **??**.
- `edge_refinement.py`: This module houses various helper functions aimed at post-processing and refining the outputs of edge and circle detection algorithms. Key functions include `remove_small_components` and `non_maximum_suppression`.
- `demo.py`: This script serves as the main demonstration and testing environment.

3.2 Technical Dependencies

The implementation relies on the following Python libraries:

- **NumPy:** Core library for numerical computations and array manipulations, used for image representation and all algorithmic calculations.
- **SciPy:** Used for its `ndimage` functionalities, for example in `edge_refinement.py` for connected component analysis via the `remove_small_components` function.
- **Pillow (PIL):** Used in `demo.py` for loading images.
- **Matplotlib:** Employed for plotting and displaying images and results in `demo.py`.
- **OpenCV (cv2):** Utilized in `demo.py` for image resizing and color space conversions.
- **os, time:** Standard libraries used for file operations and timing in `demo.py`.

4 Proposed Algorithm Implementations

4.1 FIR Convolution

The FIR convolution function (`fir_conv`) computes the 'full' 2D convolution of an input image with a kernel.

Algorithm 1 2D FIR Convolution ('Full' Output)

Flip the kernel h horizontally and vertically to get $h_{flipped}$.
Let input image dimensions be $M \times N$ and kernel dimensions be $m \times n$.
Calculate output image dimensions: $M_{out} = M + m - 1$, $N_{out} = N + n - 1$.
Initialize the output image I_{out} of size $M_{out} \times N_{out}$ with zeros.
for each row r_k from 0 to $m - 1$ and column c_k from 0 to $n - 1$ in $h_{flipped}$ **do**
    if $h_{flipped}[r_k, c_k] \neq 0$ **then**
        Let I_{in} be the input image.
        $I_{out}[r_k : r_k + M, c_k : c_k + N] \leftarrow I_{out}[r_k : r_k + M, c_k : c_k + N] + I_{in} \times h_{flipped}[r_k, c_k]$.
    end if
end for
Return convolved image I_{out} and output origin O_{out}.

The `fir_conv` algorithm involves the following key stages:

1. **Kernel Flipping**: The input kernel h is flipped both horizontally and vertically.
2. **Output Dimension Calculation**: The dimensions of the output image I_{out} are determined to accommodate a 'full' convolution. If the input image I_{in} has dimensions $M \times N$ and the kernel h has dimensions $m \times n$, the output image will have dimensions $(M + m - 1) \times (N + n - 1)$.
3. **Kernel-Centric Computation**: The algorithm iterates through each element of the flipped kernel $h_{flipped}$.
    - For each non-zero kernel element $h_{flipped}[r_k, c_k]$:
    - The entire input image I_{in} is scaled by the value of this kernel element.
    - This scaled version of the input image is then added to a corresponding sub-region of the output image I_{out}. The sub-region starts at (r_k, c_k) and has the same dimensions as the input image.
4. **Output Origin Handling**: The origin of the input image and the kernel are used to determine the origin of the resulting convolved image I_{out}.

This kernel-centric approach effectively performs a 'full' convolution by summing scaled and shifted versions of the input image, corresponding to each kernel element's influence. The core computation, where the input image is scaled by a kernel element and added to the output, leverages NumPy's vectorized operations for efficient array manipulation. This avoids explicit pixel-by-pixel loops in Python, significantly improving performance.

4.2 Sobel Edge Detection

The Sobel operator uses two 3×3 kernels to compute gradients in the x and y directions. The gradient magnitude is thresholded to produce a binary edge map.

Algorithm 2 Sobel Edge Detection

function SOBEL_EDGE(in_img_array, $threshold$, min_size)
  $G_x \leftarrow \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ ▷ Horizontal gradient kernel
  $G_y \leftarrow -G_x^T$ ▷ Vertical gradient kernel
  $g_x \leftarrow$ fir_conv(in_img_array, G_x)
  $g_y \leftarrow$ fir_conv(in_img_array, G_y)
  $magnitude \leftarrow \sqrt{g_x^2 + g_y^2}$ ▷ Gradient magnitude
  $binary\_edge\_map \leftarrow (magnitude \geq thres)$ ▷ Threshold to create binary map
  $refined_edge_map \leftarrow$remove_small_components($bin\_edge\_map$, min_size)
  return $refined\_edge\_map$
end function

The Sobel edge detection algorithm, involves the following key stages:

1. **Kernel Definition**: Two 3×3 convolution kernels are defined. The horizontal kernel (G_x) detects vertical edges by approximating the first derivative with respect to columns. detects horizontal edges by approximating the first derivative with respect to rows, created by transposing and negating G_x. Both kernels employ a central difference scheme with additional weighting to reduce noise sensitivity.

2. **Gradient Approximation**: The input image (`in_img_array`) is convolved separately with the G_x and G_y kernels using the `fir_conv` function. This process yields two gradient component images, g_x and g_y, which represent the intensity changes in the horizontal and vertical directions, respectively.

3. **Gradient Magnitude Calculation**: The overall gradient magnitude at each pixel is computed from the g_x and g_y components using the Euclidean distance formula: $magnitude = \sqrt{g_x^2 + g_y^2}$. This magnitude signifies the strength of the edge at each pixel, irrespective of its orientation. Large magnitude values indicate strong edges, while small values suggest either weak edges or homogeneous regions.

4. **Binary Edge Map Creation**: The computed gradient magnitude image is thresholded using the specified `threshold` parameter. Pixels where the gradient magnitude is greater than or equal to this `threshold` are classified as edge pixels (set to 1), while those below are classified as non-edge pixels (set to 0). This step results in a `binary_edge_map`. The `threshold` value directly influences the sensitivity of the edge detection and the density of the resulting edge map.

5. **Edge Refinement**: The `binary_edge_map` is further processed by the `remove_small_components` function. This function uses the `min_size` parameter to identify and eliminate small, isolated connected components of edge pixels, which often correspond to noise. The output is a `refined_edge_map`.

4.3 Laplacian of Gaussian Edge Detection

The Laplacian of Gaussian (LoG) edge detector combines Gaussian smoothing with the Laplacian operator to identify edges at zero-crossings in the filtered image. To enhance robustness against noise, adaptive thresholding is applied to the zero-crossings, and small, isolated edge segments are removed using connected component analysis.

Algorithm 3 Laplacian of Gaussian Edge Detection

function LOG_EDGE(in_img_array, $kernel\_size$)
    Determine $sigma$, $thresh\_mult$, min_size based on $kernel\_size$.
    $log\_kernel \leftarrow$ CREATE_LOG_KERNEL($sigma$, $kernel\_size$)
    $filtered\_img \leftarrow$ FIR_CONV(in_img_array, log_kernel)
    $binary\_edge\_map \leftarrow$ FIND_ZERO_CROSSINGS($filtered\_img$, $thresh\_mult$)
    $refined\_edge\_map \leftarrow$ REMOVE_SMALL_COMPONENTS($binary\_edge\_map$, min_size)
    return $refined\_edge\_map$
end function

The LoG edge detection algorithm proceeds in the following stages:

1. **Parameter Initialization:** The primary function, `log_edge`, accepts the input image and a `kernel_size`. From `kernel_size`, several parameters are derived, which have been optimized for typical performance:
    - `sigma`: The standard deviation for the Gaussian component of the LoG filter. It dictates the scale of smoothing; larger `sigma` values result in more smoothing.
    - `thresh_mult`: A multiplier used in calculating the adaptive threshold .
    - `min_size`: The minimum number of pixels for a connected component to be considered a valid edge segment. This is used to filter out noise after detection.
2. **Kernel Creation:** The `create_log_kernel` function generates the LoG kernel. It uses the specified `kernel_size` and the derived `sigma`. The resulting kernel is normalized so that its elements sum to approximately zero.
3. **Convolution:** The input image is convolved with the LoG kernel using `fir_conv`.
4. **Adaptive Zero-Crossing Detection:** Edges are provisionally identified at pixels where the intensity in the `filtered_img` changes sign (crosses zero) with respect to their 8-connected neighbors. The `find_zero_crossings` logic first collects all absolute intensity differences that occur at zero-crossings between pixels and their neighbors. An `adaptive_threshold` is then computed using the formula: $\text{mean}(\text{differences}) + thres\_mul \times \text{std}(\text{differences})$. A pixel is confirmed as an edge point if a zero-crossing is detected with any of its neighbors AND the absolute intensity difference across that zero-crossing exceeds this `adaptive_threshold`. This adaptive approach helps to preserve significant edges while suppressing weak or spurious zero-crossings.

5. **Edge Map Refinement:** Finally, the `remove_small_components` function cleans the binary edge map. It uses connected component analysis to identify contiguous regions of edge pixels. Any component (segment) containing fewer pixels than `min_size` is removed, effectively eliminating small, isolated noise artifacts.

4.4 Hough Circle Transform

This circle detection algorithm uses Hough transform to identify circles in binary edge maps:

Algorithm 4 Hough Circle Transform

function CIRC_HOUGH(in_img_array, R_max, dim, V_min)
    Initialize parameter space (accumulator) with dimensions dim
    Extract edge pixels from input image
    for each radius r in discretized range from R_min to R_max **do**
        Generate sampling angles based on radius size
        Calculate potential circle centers for all edge points
        Increment accumulator bins for valid center-radius combinations
    end for
    Extract circle candidates where votes $\geq V\_min$
    Apply non-maximum suppression to eliminate overlapping detections
    return circle centers, radii, and vote counts
end function

The key steps are:

1. **Parameter Space Definition:** An accumulator array tracks votes for all possible combinations of circle parameters (center coordinates a, b and radius r).
2. **Voting Process:** For each edge pixel and each potential radius:
    - The algorithm calculates possible circle centers using the parametric equation: $a = x_e - r\cos\theta$ and $b = y_e - r\sin\theta$ for various angles θ.
    - The number of sampling angles is adaptively increased for larger radii to maintain consistent coverage.
    - All potential circle parameters receive votes in the accumulator, with the process vectorized for efficiency.
3. **Candidate Circle Extraction:** Accumulator cells with votes greater than or equal to `V_min` are identified as candidate circles.
4. **Non-Maximum Suppression:** Overlapping detections are eliminated by retaining only the highest-voted circle within proximity thresholds for center location & radius.
5. **Result Processing:** The algorithm returns sorted circles with their centers, radii, and corresponding vote counts, allowing visualization ranked by detection confidence.

5 Experimental Results and Analysis

Test Image

All experiments are conducted using the `basketball_large.png`. Contains both natural curved shapes and clear circular objects, making it ideal for testing both edge detection and circle identification algorithms.

Figure 1: Basketball test image used in the experiments.

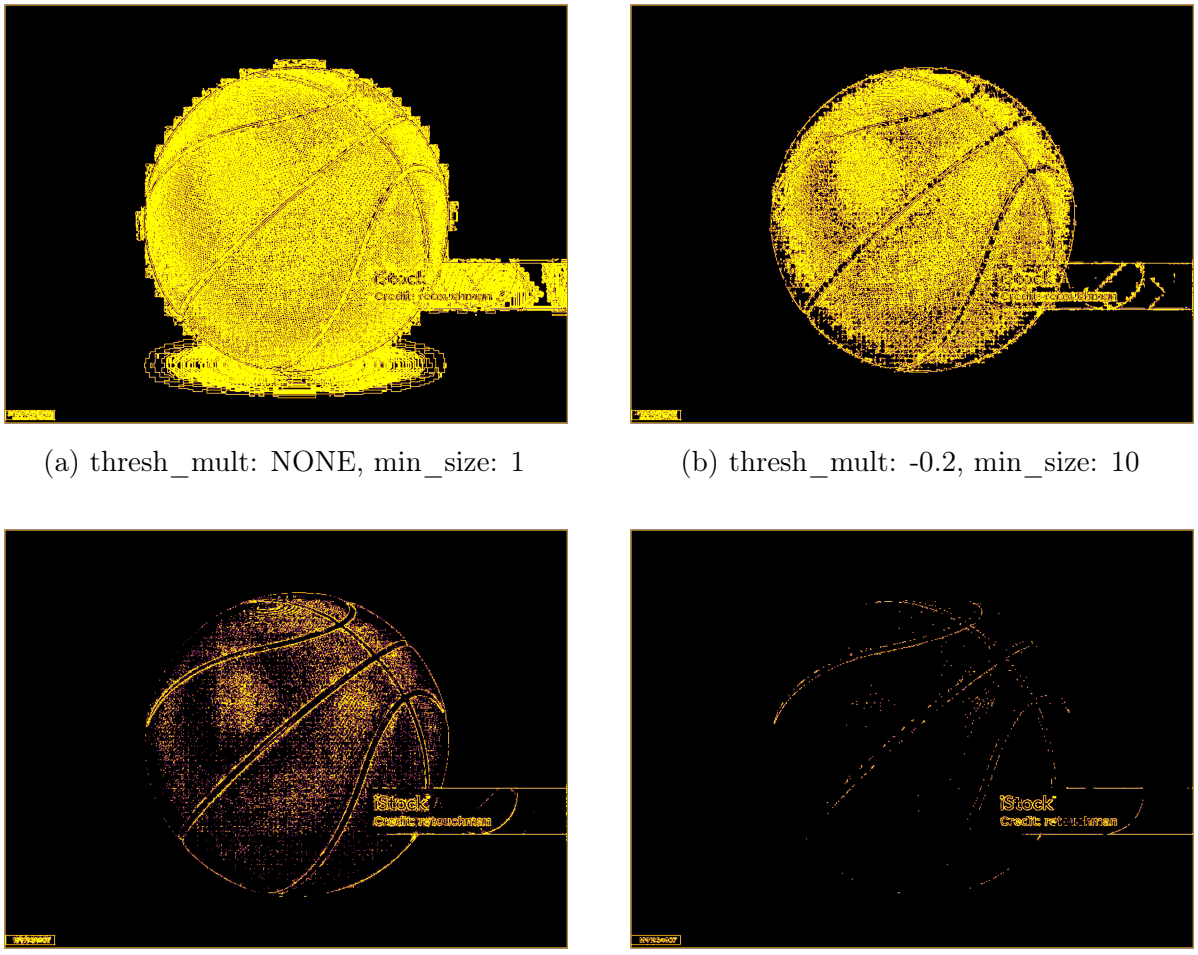
5.1 LoG Edge Detection

The `process_log_edge` function in `demo.py` applies LoG edge detection, systematically testing kernel sizes of [5, 7, 9, 11, 13].

Visual Results

Kernel size: 5

With a 5x5 kernel, the LoG detector captures even the finest details of the image, including the textured pebbles on the basketball's surface. While this level of detail might be desirable for some applications requiring intricate feature extraction, it becomes problematic for our subsequent circle detection task. These fine-grained features, such as the pebble textures, essentially function as noise when attempting to identify the primary circular shape of the basketball. The Hough transform, used for circle detection, can be misled by these numerous small edges, potentially identifying spurious circles or failing to robustly detect the main object. Therefore, subsequent experiments will explore methods to mitigate this high-frequency noise, by increasing the kernel size for more aggressive smoothing, while retaining essential edge information critical for robust circle detection.



(a) thresh_mult: NONE, min_size: 1

(b) thresh_mult: -0.2, min_size: 10

(c) thresh_mult: 0.1, min_size: 1

(d) thresh_mult: 0.5, min_size: 5

Figure 2: Effects of varying threshold multiplier and minimum component size on LoG detection with kernel size 5. Higher threshold values reduce noise but miss subtle edges.

Recursive Parameter Tuning Process

To optimize LoG edge detection, an iterative parameter tuning strategy was employed. This process commenced with an initial kernel size and an adaptive threshold mechanism. If crucial edges appeared missing or fragmented, the thresholding sensitivity was increased (by adjusting `thres_mul` of the adaptive threshold) to improve edge continuity. Conversely, if this adjustment introduced an unacceptable level of noise for the subsequent circle detection, the kernel size was increased to enhance smoothing. This, in turn, often necessitated a readjustment of the threshold parameters. This iterative refinement of kernel size, threshold sensitivity, and minimum component size for post-processing continued until an optimal balance between edge preservation and noise suppression was achieved.

This approach led to the exploration of kernel sizes from 5 up to 13. As illustrated in Figure 3, larger kernel sizes yielded progressively smoother edge maps with reduced noise, thereby emphasizing the primary outline of the basketball while diminishing textural details. Kernel sizes beyond 13 showed diminishing returns in edge quality improvement, establishing it as a practical upper limit for this application.

Notably, the execution time remained efficient even with kernel dimensions up to 13×13. This efficiency is attributable to the vectorized implementation, which performs operations on entire arrays rather than relying on slower, explicit loops. This configuration produced clean, continuous edges that accurately delineated the basketball's circular boundary while effectively eliminating internal texture noise, making it well-suited for the subsequent circle detection phase.

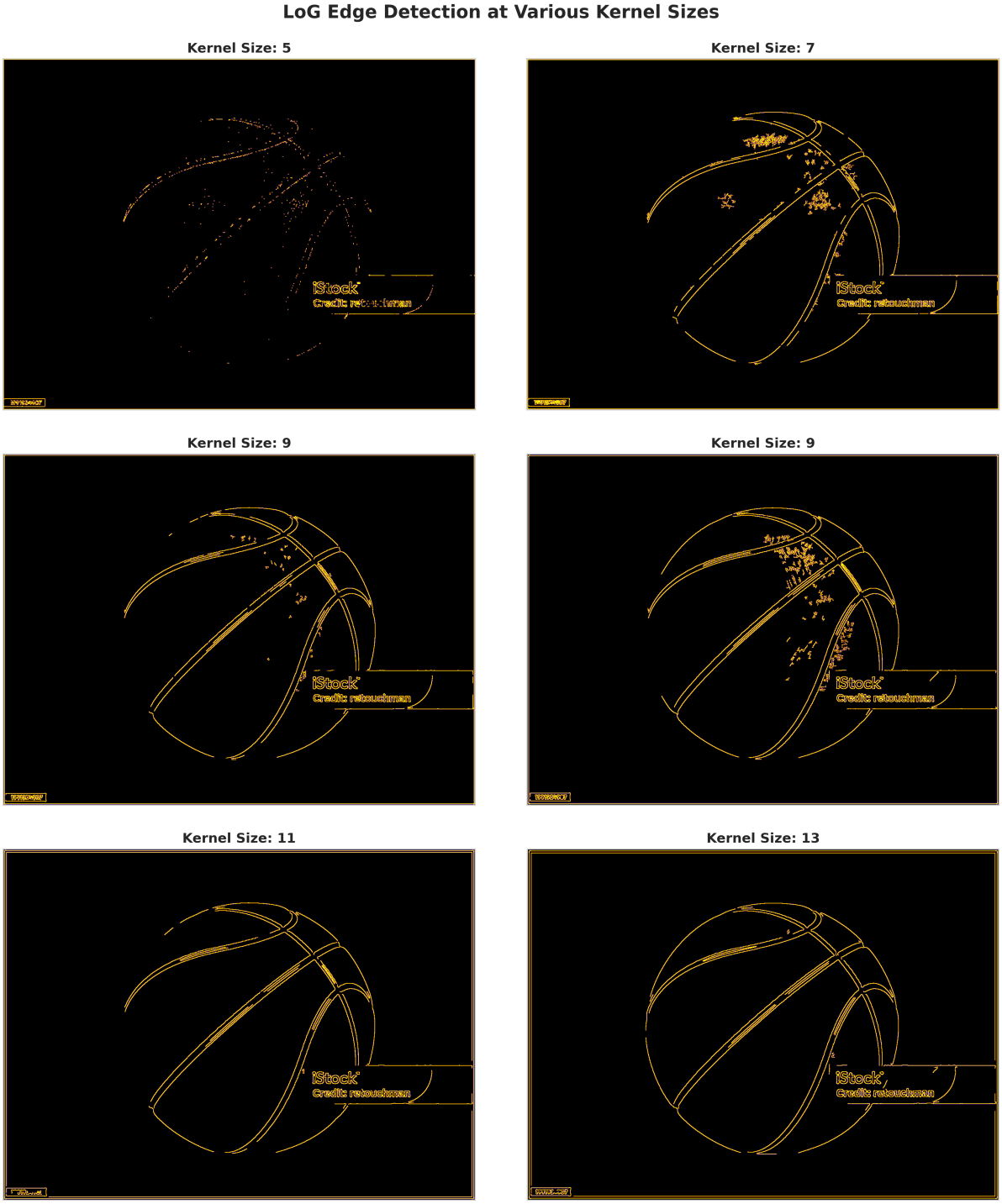


Figure 3: Progression of LoG edge detection results with increasing kernel sizes

5.2 Sobel Edge Detection

The `process_sobel_edge` function, located in `demo.py`, implements Sobel edge detection. This section analyzes its performance with thresholds ranging from 0.2 to 0.8.

Visual Results

The Sobel operator, being gradient-based, responds strongly to sharp intensity transitions. As illustrated in Figure 4, lower thresholds (e.g., 0.1-0.2) produce dense edge maps, capturing fine details but also incorporating more noise. Conversely, higher thresholds (e.g., 0.8-1.0) result in sparser edge maps, retaining only the most prominent edges. An optimal balance, effectively capturing primary object boundaries while suppressing noise, appears to be achieved with thresholds around 0.4-0.6. The images in Figure 4 are rendered using the 'gnuplot' colormap for distinct visualization of edge intensities. Notably, bilinear interpolation, a method that calculates pixel values by taking a weighted average of the four nearest pixel values, is deliberately applied during display, which serves an important analytical purpose: it causes isolated noise pixels to appear as distinctive purple artifacts.

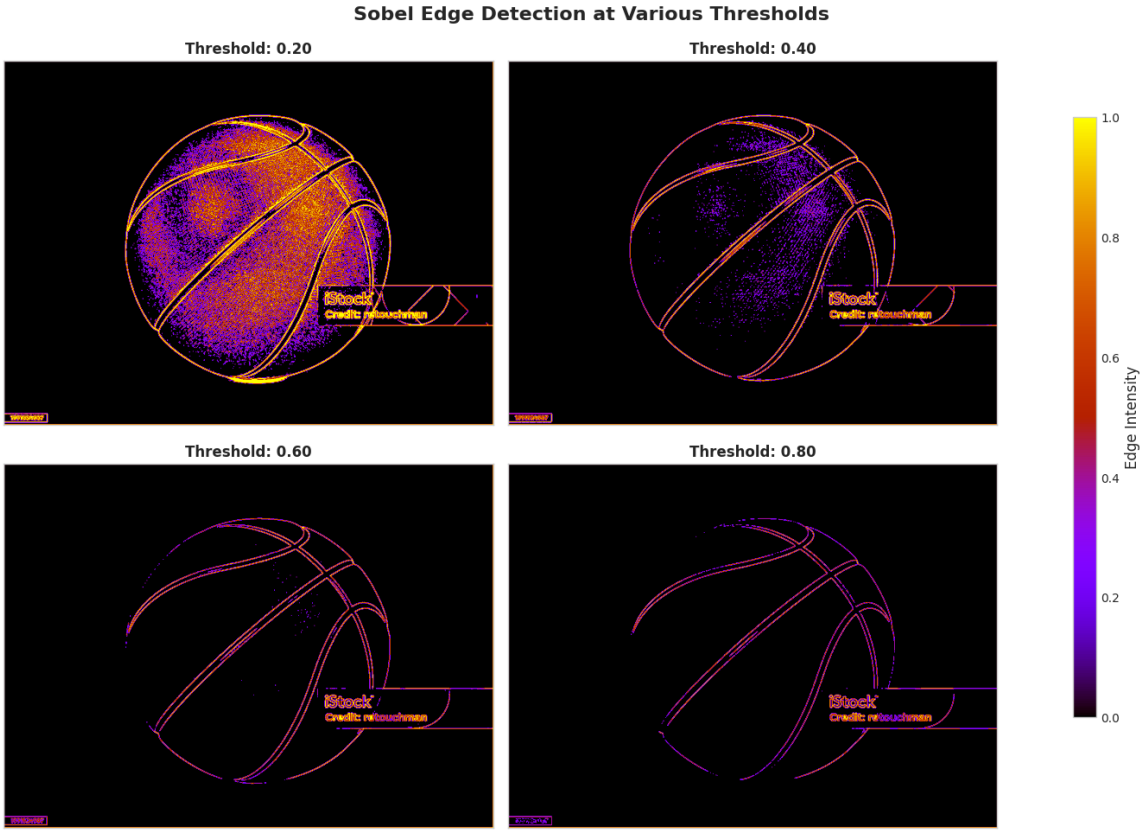
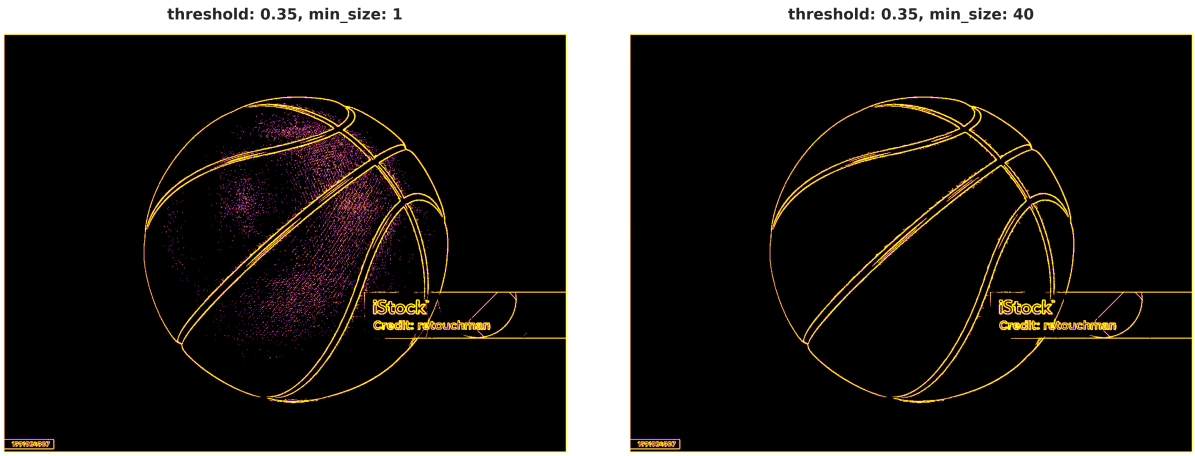


Figure 4: Sobel: Varying thresholds, showing the trade-off between detail and noise

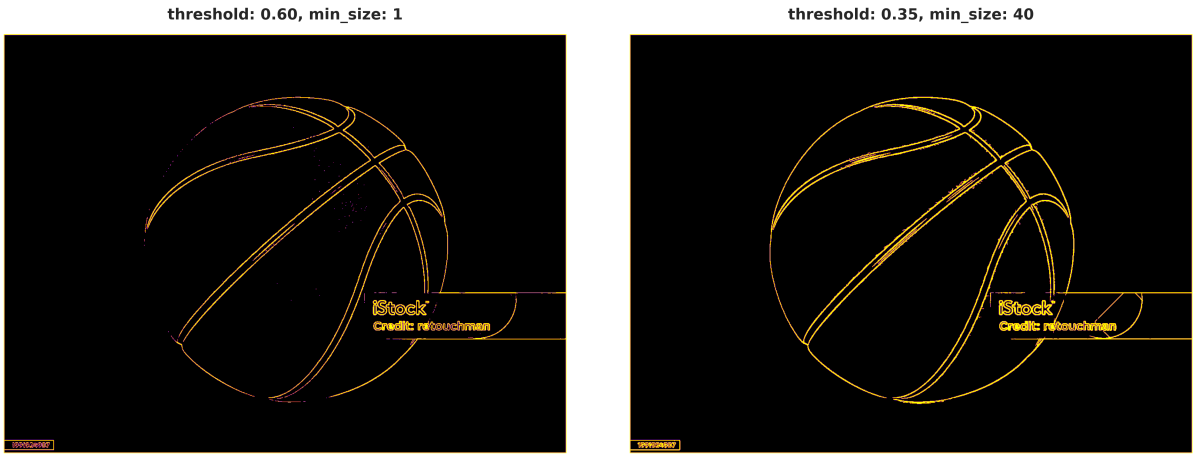
Further refinement can be achieved by addressing textural noise. For instance, at a threshold of 0.4, many of the basketball's pebbles manifest as isolated pixels. Applying the `remove_small_components` function effectively eliminates these isolated pixels. As shown in Figure 5, this post-processing step yields a cleaner edge map that preserves significant structural edges while eliminating completely the textural noise.



(a) Before `remove_small_components()` (b) After `remove_small_components()`

Figure 5: Effect of `remove_small_components` on Sobel edge detection. Significant improvement in edge quality is observed after removing small, isolated components.

Figure 6 compares two approaches that achieve complete noise elimination. The first uses a raw edge map with a higher threshold (0.6) chosen specifically to eliminate noise through thresholding alone. The second employs a lower threshold (0.35) followed by small component removal, which effectively eliminates all noise while preserving more edge detail. This comparison demonstrates that the two-step approach (lower threshold with cleaning) produces superior results by retaining more structural information.



(a) Sobel with threshold 0.6, no cleaning (b) Sobel with threshold 0.35, after cleaning

Figure 6: Comparison: Sobel edge detection at an optimal raw threshold (0.6) versus a lower threshold (0.35) with subsequent small component removal.

Threshold vs. Edge Count

The relationship between the threshold value and the total number of detected edge pixels is depicted in Figure 7. The plot shows a decreasing trend: as the threshold increases, fewer pixels are classified as edges. This characteristic exponential decline reflects the distribution of gradient magnitudes within the image and underscores the critical role of threshold selection in balancing detail retention against noise suppression.

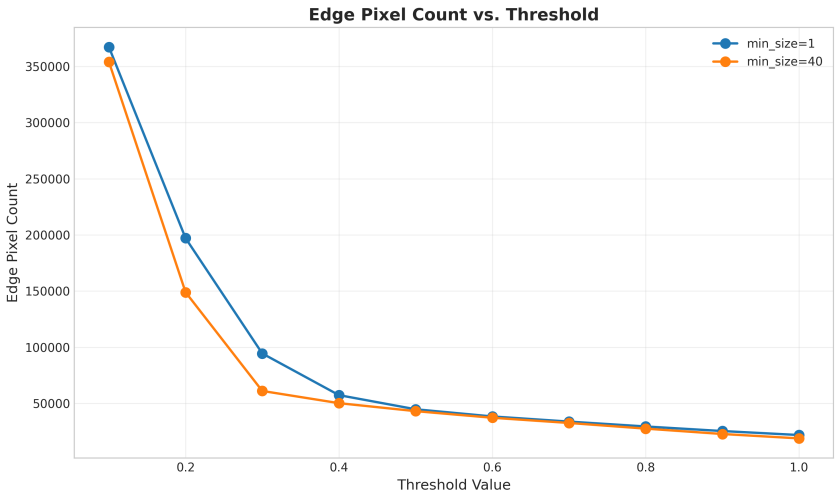
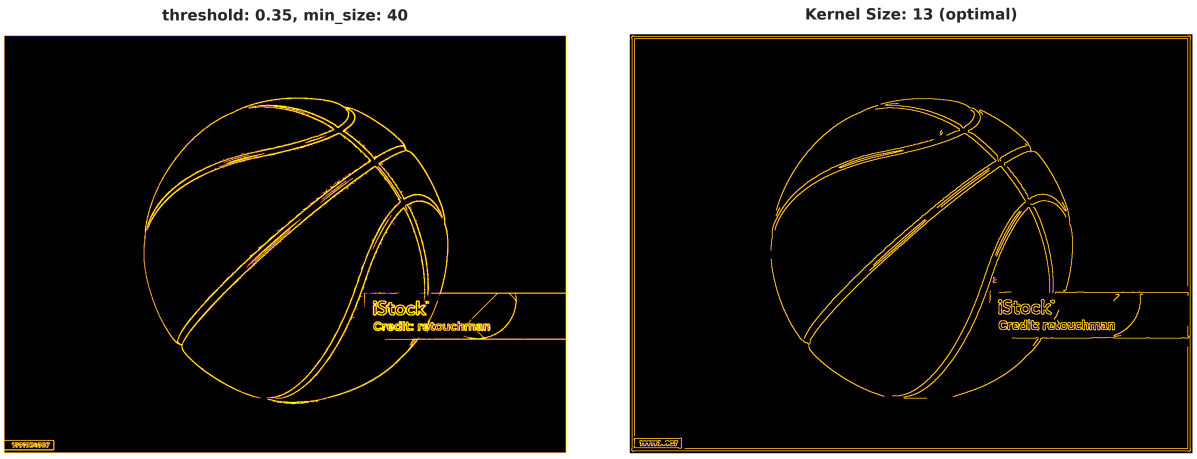


Figure 7: Edge pixel count as a function of threshold for the Sobel operator. The exponential decline highlights the sensitivity of edge detection to the chosen threshold.

5.3 Comparison: Sobel vs. LoG

For a direct comparison, optimal parameters, derived from prior experiments, were used for both the Sobel and Laplacian of Gaussian (LoG) methods. The Sobel operator was configured with a threshold of 0.35 and a minimum component size of 40; the LoG operator utilized a kernel size of 13.



(a) Sobel (b) LoG

Figure 8: Comparison of Sobel and LoG. Note Sobel's thicker but more continuous edges.

Figure 8 highlights distinct performance characteristics of Sobel and LoG under these optimized configurations:

Sobel Operator (threshold 0.35, min_size 40): With this parameter configuration, the Sobel operator demonstrates significantly improved edge continuity and is capable of capturing more subtle or "difficult" edges. The low threshold of 0.35 increases its sensitivity, allowing for the detection of finer details and weaker Post-processing with a 'min_size' of 40 effectively removes noise associated with low thresholds, preserving meaningful edge segments and resulting in a comprehensive edge map. A known characteristic of Sobel, especially at low thresholds, is the production of thicker edges, as gradient magnitudes can exceed the threshold across a wider pixel band around the true edge. This thickness is reflected in the Sobel edge map's 55,000 edge pixels.

Laplacian of Gaussian (kernel size 13): The LoG method aims for thinner, precisely localized edges by detecting zero-crossings of the second derivative. Integrated Gaussian smoothing offers excellent noise immunity, yielding clean edge lines. This precision results in 33,000 edge pixels, about 40% fewer than Sobel, despite detecting similar features. However, LoG may exhibit less continuity on challenging, low-contrast edges if they don't produce strong zero-crossings after smoothing. This can happen if zero-crossings from subtle features are weak or fall below the adaptive zero-crossing detection threshold. While LoG targets single-pixel-wide edges, variations in edge strength can break these thin lines, making them appear less continuous than Sobel's inherently thicker edges.

Complexity: Sobel is computationally lighter, using small, fixed kernels and direct gradient calculation. LoG is more demanding, involving convolution with a larger Gaussian kernel and zero-crossing detection, although our vectorized implementation mitigates this.

The choice between Sobel and LoG depends on the specific requirements of subsequent image processing tasks. For tasks needing robust, though less precise, edge segments (e.g., tracking large structures), Sobel's output may be preferable. For high-precision measurements requiring fine, single-pixel thick edges, LoG is theoretically superior, but achieving consistent continuity demands careful parameter tuning.

5.4 Hough Circle Detection

The Hough Circle Transform is implemented to detect circles, capable of using either Sobel or LoG edge maps as input. Edge detection is performed at full image resolution.

Computational Optimization: When processing large edge maps, automatic downscaling is applied based on the `max_edge_pixels` parameter. This manages computational cost, potentially impacting effective resolution, but helps ensure a more consistent processing load regardless of the edge detection method used and input image.

5.4.1 Critical Parameters

The detection process is governed by several key parameters that can be organized into three main categories:

Dimensional Parameters

- **Radius Search Range (`R_min`, `R_max`)**: Defines the minimum and maximum radii to search for in the edge map. This range should be chosen based on the expected sizes of circles in the image, accounting for any downscaling applied.
- **Maximum Edge Pixels (`max_edge_pixels`)**: Controls computational load by limiting the number of edge pixels processed. If an input edge map exceeds this threshold, it is downscaled by a dynamically calculated factor.

Accumulator Configuration

- **Accumulator Dimensions (`dim`)**: This parameter governs the resolution of the 3D Hough accumulator for center coordinates (a, b) and radius r. The actual number of bins for a, b, and r are derived from the (potentially downscaled) edge map's spatial extent and the selected `R_max`, using an internal scaling factor (`pixels`). This factor determines how coarsely or finely the parameter space is quantized.
- **Minimum Vote Threshold (`V_min`)**: Sets the minimum number of votes an (a, b, r) triplet must receive in the accumulator to be considered a candidate circle. This directly impacts detection sensitivity.

Non-Maximum Suppression Controls

- `nms_center_thresh_perc`: Specifies the maximum allowable distance between circle centers (as a percentage of `R_min`) for them to be considered overlapping.
- `nms_radius_thresh_perc`: The maximum relative percentage difference in radii for two circles to be considered overlapping.

The interplay between these parameters significantly affects the sensitivity, precision, and computational cost of the detection process.

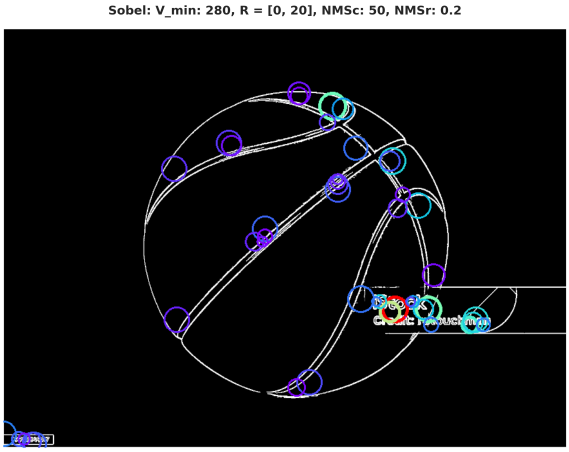
5.4.2 Visualization Approach

Detected circles are visualized with the following conventions:

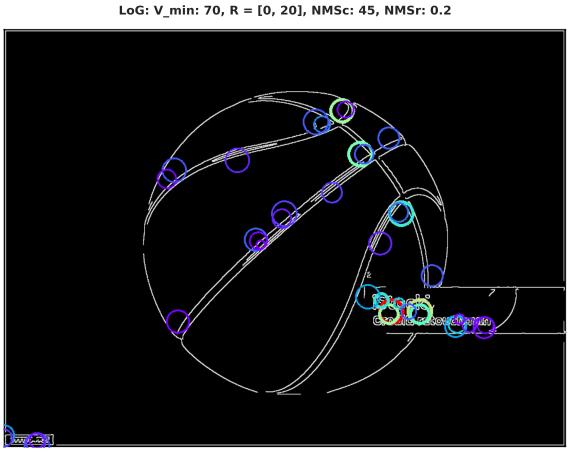
- Color coding based on vote strength using a rainbow colormap, where higher vote counts (stronger circles) receive distinct colors.
- Line thickness proportional to the normalized vote value, causing stronger detections to be drawn with thicker lines.

Radius Range Analysis

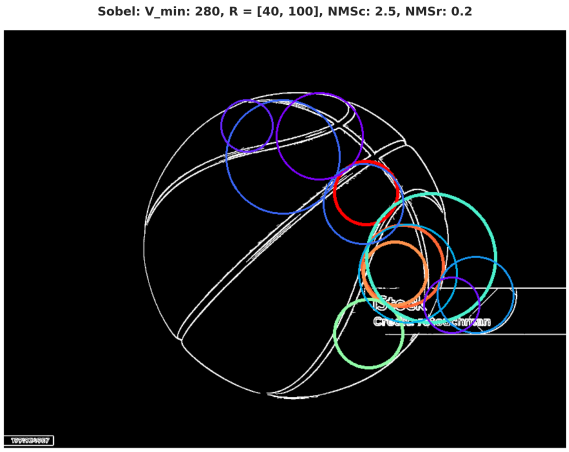
As shown in Figure 9, the detection sensitivity varies significantly across different radius:



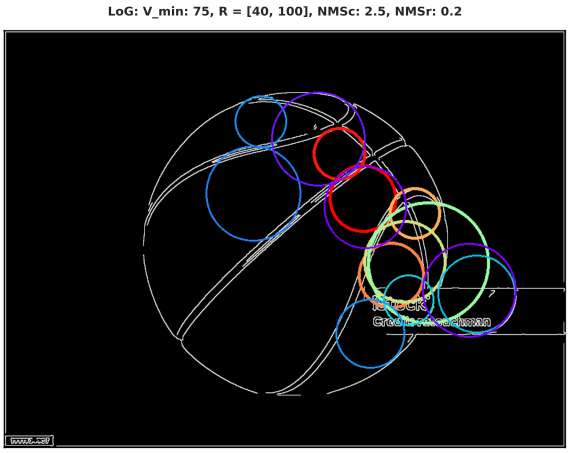
(a) Sobel: Small radius



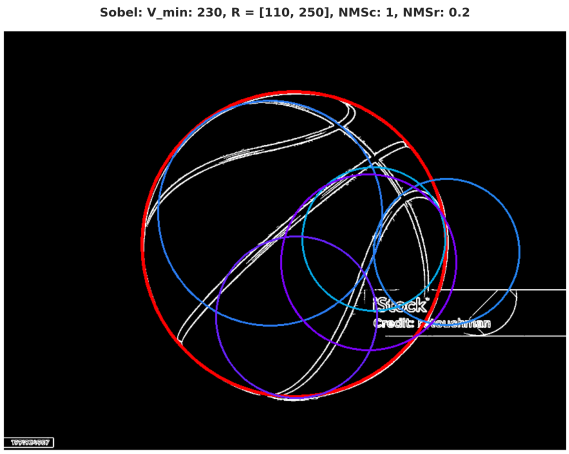
(b) LoG: Small radius



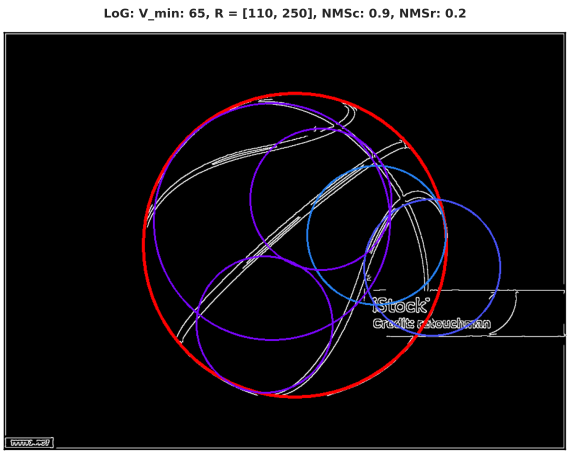
(c) Sobel: Medium radius



(d) LoG: Medium radius



(e) Sobel: Large radius

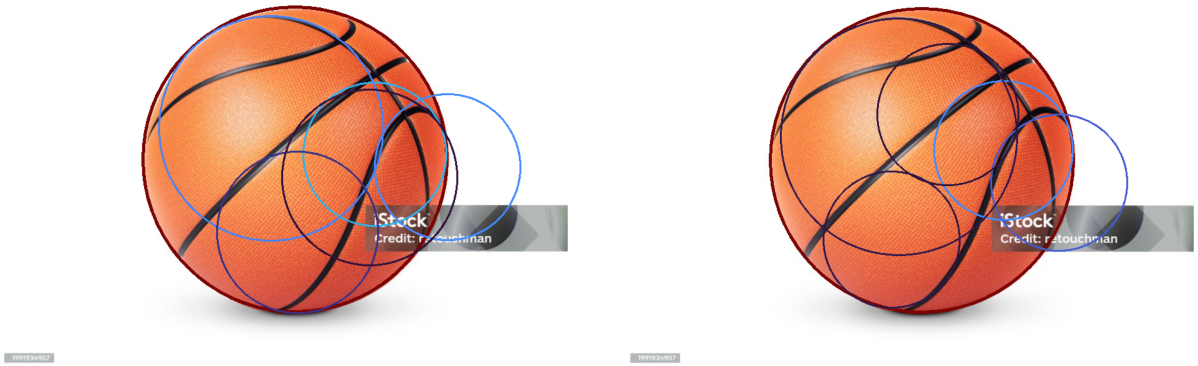


(f) LoG: Large radius

Figure 9: Circle detection: Sobel (left) vs LoG (right) edge maps across different radius.

Edge Detection Method Comparison

Figure 10 illustrates the largest circles detected using both Sobel and LoG edge maps, overlaid on the original image:



(a) Sobel-based circle detection (b) LoG-based circle detection

Figure 10: Largest circles detected by the Hough transform, using Sobel and LoG edge maps, overlaid on the original color image.

Edge Detector Influence : LoG detection generally required lower vote thresholds (`V_min`) compared to Sobel due to the thinner, more precise nature of LoG edge maps.

6 Conclusion

This study has implemented and analyzed key image processing techniques for edge and circle detection. The comparative analysis reveals distinct characteristics of each approach:

Edge Detection Insights

The **Sobel operator** demonstrates computational efficiency with a straightforward implementation using fixed 3×3 kernels. It produces robust, continuous edge maps but with characteristically thicker edges. Lower thresholds (0.35) combined with small component removal (min_size=40) yielded optimal results, balancing sensitivity to subtle features with noise suppression.

The **Laplacian of Gaussian** detector achieves superior edge localization with precisely defined, single-pixel-wide edges. The investigation of kernel sizes (5-13) revealed that larger kernels provided necessary smoothing while preserving structural boundaries. The zero-crossing detection with adaptive thresholding effectively identified true edges while suppressing spurious responses.

Cycle Detection Insights

The **Hough circle transform** successfully bridged edge detection and object recognition, with performance directly influenced by edge map quality. The experiments demonstrated that:

- Effective parameter tuning of radius ranges and accumulator dimensions was critical for accurate circle identification
- Non-maximum suppression strategies effectively eliminated redundant detections
- LoG-based detection required lower vote thresholds compared to Sobel due to the thinner, more precise nature of LoG edge maps

Future Directions

Based on the challenges encountered and opportunities identified during implementation:

- **Adaptive parameter selection** could leverage machine learning approaches to automatically determine optimal thresholds, kernel sizes, and accumulator parameters based on image characteristics
- **Geometric shape expansion** beyond circles to detect ellipses, lines, and arbitrary curves would enhance detection capabilities for complex scenes
- **Performance optimization** particularly for the Hough transform, to enhance suitability for real-time processing applications in video processing
- **Investigation of more sophisticated edge refinement techniques** beyond simple component removal.

The implemented framework provides a solid foundation for these extensions, with clearly defined interfaces between processing stages that facilitate experimentation and further development in practical image processing applications.