IT3212 Assignment 2: Image Preprocessing

Table of Contents

    - 3. Discuss the impact of varying parameters like cell size, block size, and the number of bins on the resulting HOG descriptors.

- Local Binary Patterns

    - 1. Write a Python function to compute the LBP of a given grayscale image (basic 8-neighbor). Your function should output the LBP image, where each pixel is replaced by its corresponding LBP value.
    - 2. Write a Python function to compute the histogram of the LBP image. Plot the histogram and explain what it represents in terms of the texture features of the image.
    - 3. Apply your LBP function to at least three different grayscale images (e.g., a natural scene, a texture, and a face image). Generate and compare the histograms of the LBP images.
    - 4. Discuss the differences in the histograms and what they tell you about the textures of the different images.

- Implement a Blob Detection Algorithm.

    - 1. Apply the contour detection algorithm to the same image dataset. Visualize the detected contours on the original images, marking each contour with a different color.
    - 2. Calculate and display relevant statistics for each image, such as the number of blobs detected, their sizes, and positions.
    - 3. Evaluate and discuss the effect of different parameters in the algorithms on the detection of different blobs.

- Implement a Contour Detection Algorithm

    - 1. Apply the contour detection algorithm to the same image dataset. Visualize the detected contours on the original images, marking each contour with a different color.
    - 2. Calculate and display relevant statistics for each image, such as the number of contours detected, contour area, and perimeter.
    - 3. Compare the results of blob detection and contour detection for the chosen dataset.
    - 4. Discuss the advantages and limitations of each technique.
    - 5. Analyze the impact of different parameters (e.g., threshold values, filter sizes) on the detection results.
    - 6. Provide examples where one technique might be more suitable than the other.

Fourier Transformation

1. Load a grayscale image and apply the 2D Discrete Fourier Transform (DFT) to it Visualize the original image and its frequency spectrum (magnitude). Submit the images, and explanation.

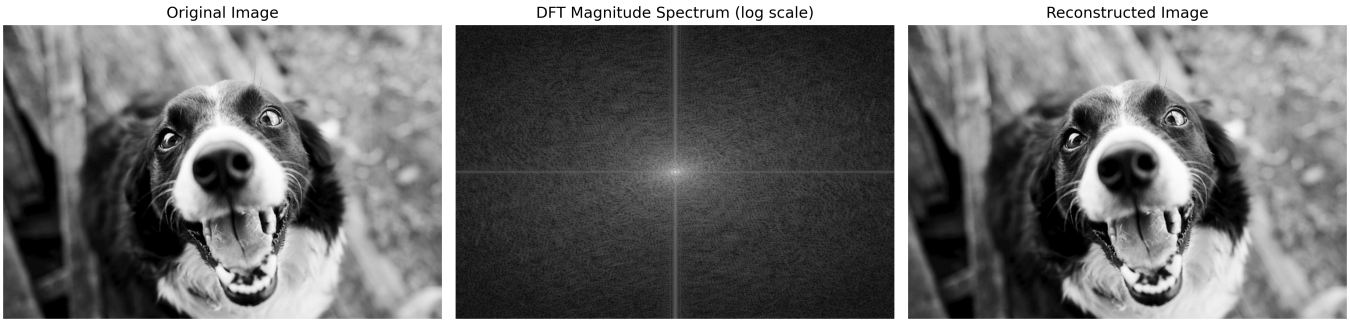


Figure 1: Discrete fourier transformation

The 2D Discrete Fourier Transform (DFT) converts an image from the spatial domain to the frequency domain. In this domain, each point encodes a sinusoidal frequency and orientation, with the center representing the lowest frequencies (average intensity) and the outer points representing higher frequencies (detail and sharp changes). An inverse DFT transforms the frequency representation back into the spatial domain, and reconstruct the image. We see this in figure 1, the log-scale magnitude spectrum is brightest at the center (low frequencies) and sparse toward the edges (high frequencies), and the inverse DFT on the right reconstructs the image accordingly.

2. Implement a low-pass filter in the frequency domain to remove high-frequency noise from an image. Compare the filtered image with the original image. Submit images, and analysis of the results

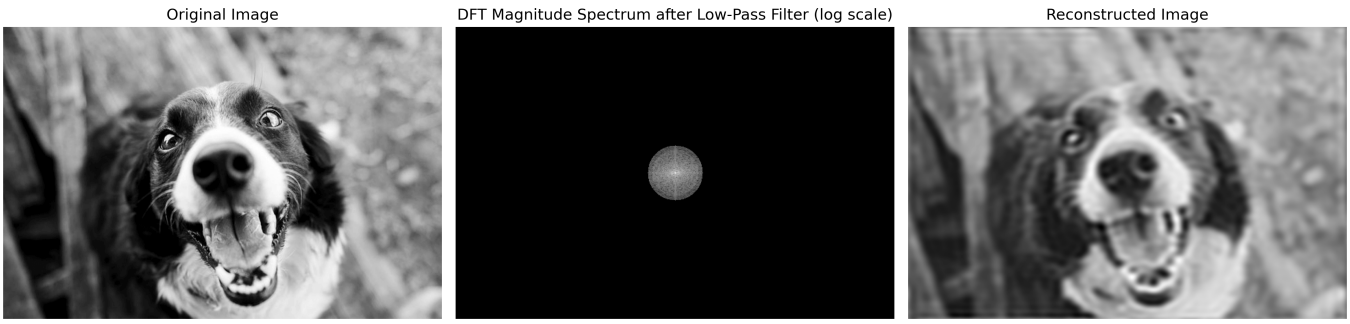


Figure 2: Low-pass filter

A low-pass filter on the DFT keeps the low-frequency components near the spectrum's center and suppresses high-frequency components toward the edges. After applying the inverse DFT, the loss of high-frequency detail like edges and fine textures produces a blurred image, as seen in figure 2.

3. Implement a high-pass filter to enhance the edges in an image. Visualize the filtered image and discuss the effects observed. Submit images, and explanation.

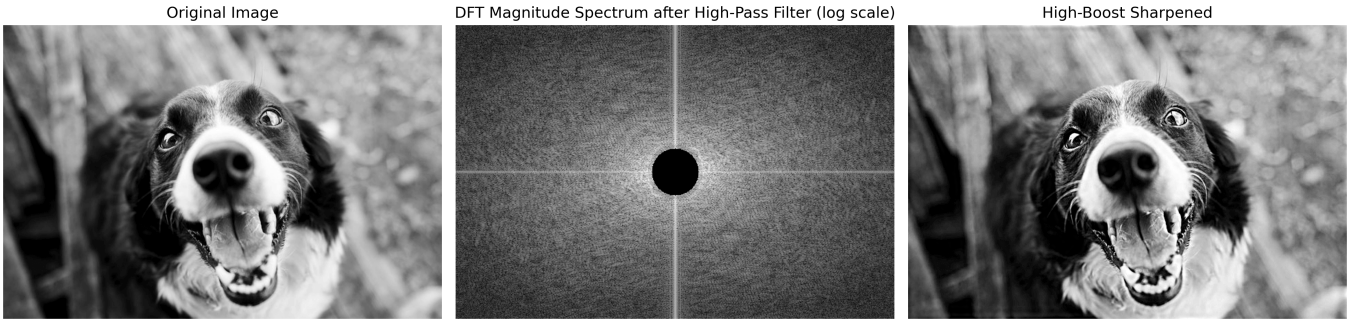
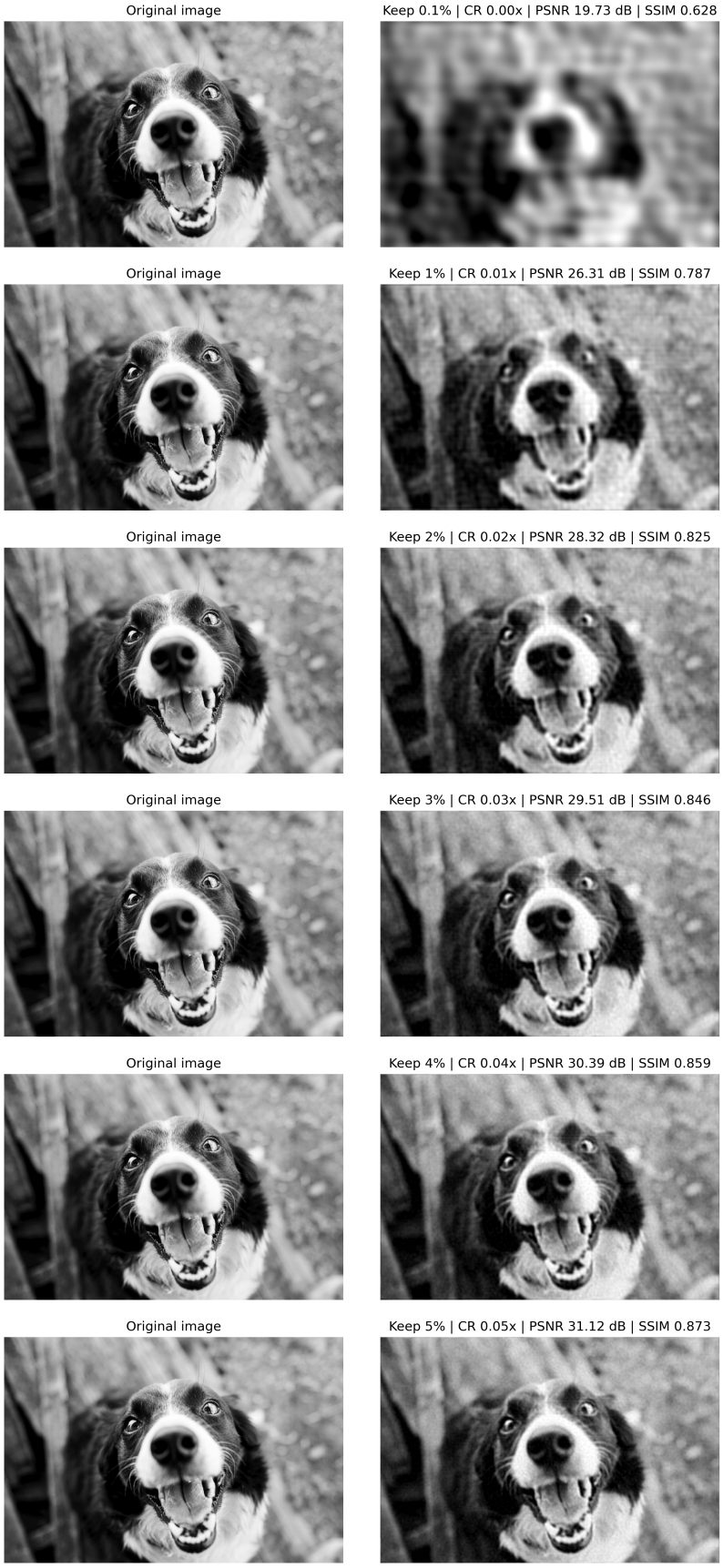


Figure 3: High-pass filter

A DFT high-pass filter preserves the high-frequency components toward the spectrum's edges while suppressing the low-frequency components near the center. After the inverse DFT, the retained high-frequency detail emphasizes edges and fine textures, yielding a sharper image, as shown in figure 3.

4. Implement an image compression technique using Fourier Transform by selectively keeping only a certain percentage of the Fourier coefficients. Evaluate the quality of the reconstructed image as you vary the percentage of coefficients used. Submit the images, and your observations on image quality and compression ratio.



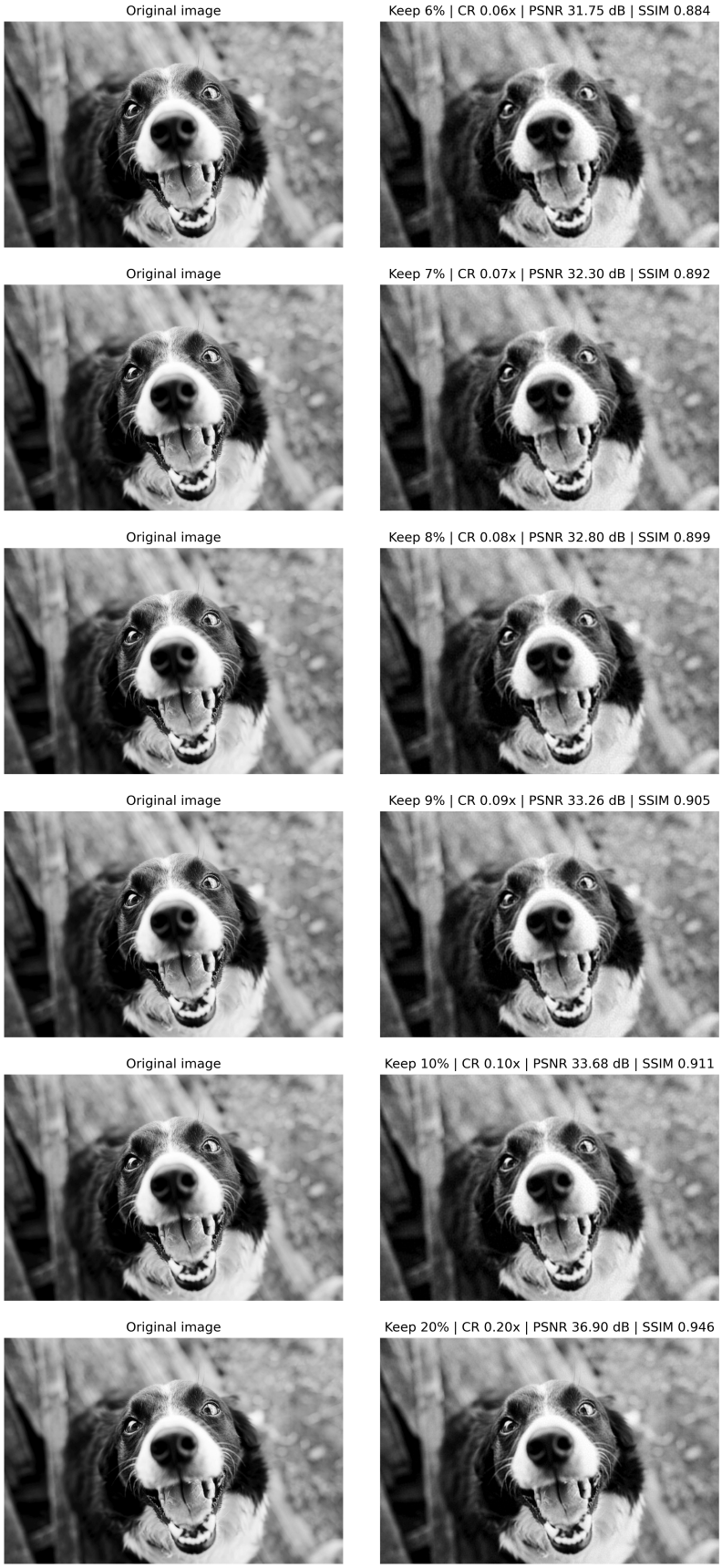


Figure 4: Discrete fourier transformation coeffisients

Keeping only a percentage of the Fourier coefficients means ranking the DFT coefficients by magnitude and retaining just the largest ones, while zeroing the rest. As the proportion of retained Fourier coefficients increases, the visual quality of the reconstructed images improves gradually, as shown in figure 4. At 0.1% of coefficients, the image is barely recognizable, only the dog's rough outline and overall composition are

visible, with fine details lost. At 1-5%, most structures and textures are restored, but the reconstructions still remain blurry. From 5%, the reconstruction becomes nearly indistinguishable from the original image.

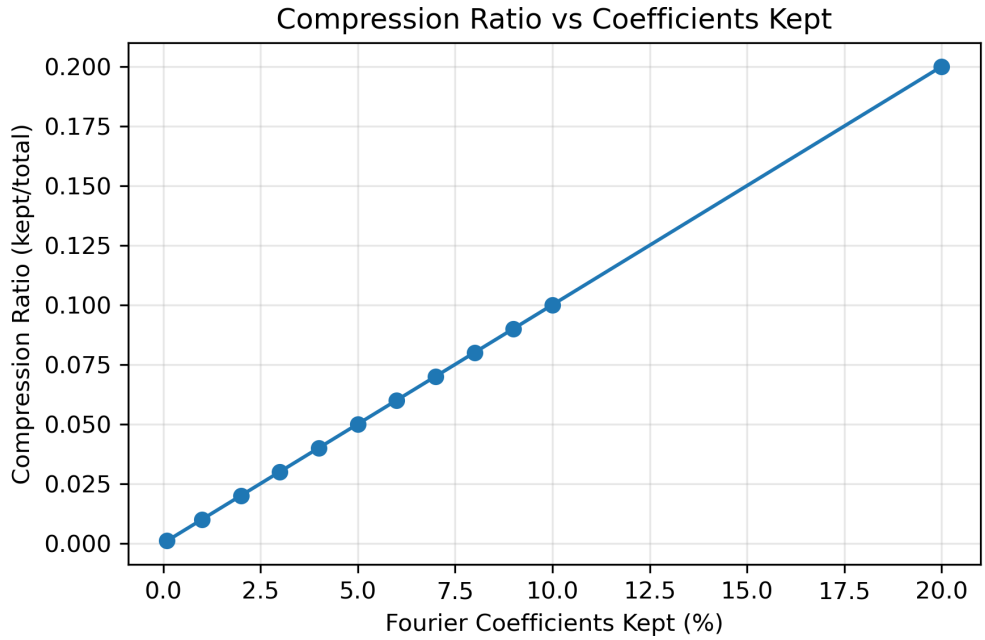


Figure 5: Compression ratio

Compression ratio (CR) measures how much an image is reduced in size, defined as compressed size divided by original size. A lower CR means more compression, and potentially greater quality loss, while a higher CR means less compression. As shown in figure 5, the compression ratio is linear, meaning that keeping more coefficients takes more space.

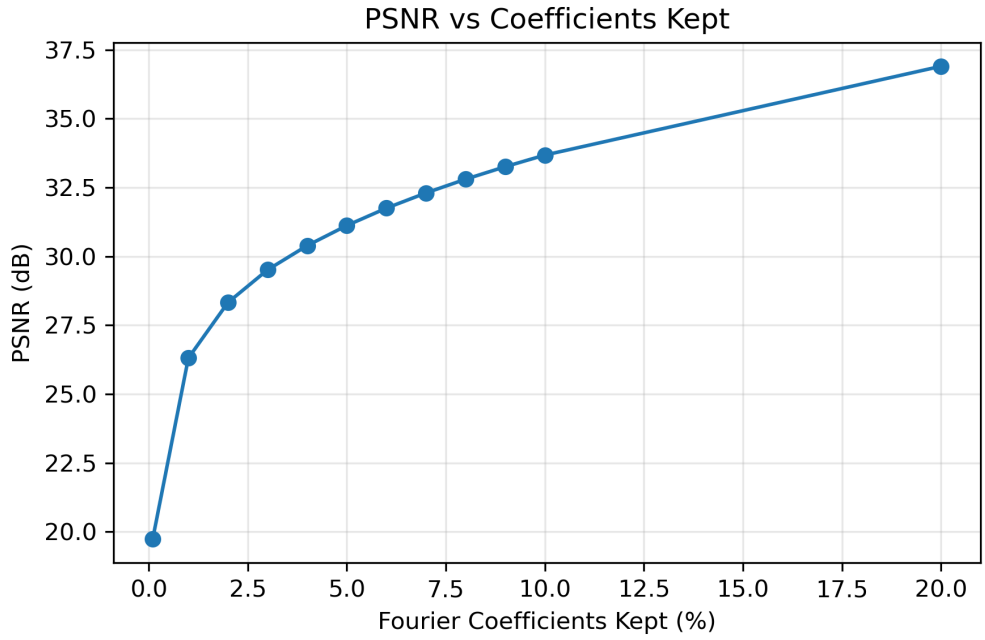


Figure 6: Peak Signal-to-Noise Ratio

Peak Signal-to-Noise Ratio (PSNR) computes the peak signal-to-noise ratio, in decibels, between two images. This ratio is used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image. In figure 6, PSNR rises quickly at very low keep rates, from about 20 dB at 0.1% to around 29–31 dB by 3–5%, then levels off, reaching roughly 37 dB at 20%. As noted in in figure 4, retaining more than 5% of coefficients yields little additional quality.

This aligns with the PSNR curve in figure 6 where increases beyond 5% are marginal, and PSNR values above 30 dB already indicate good quality.

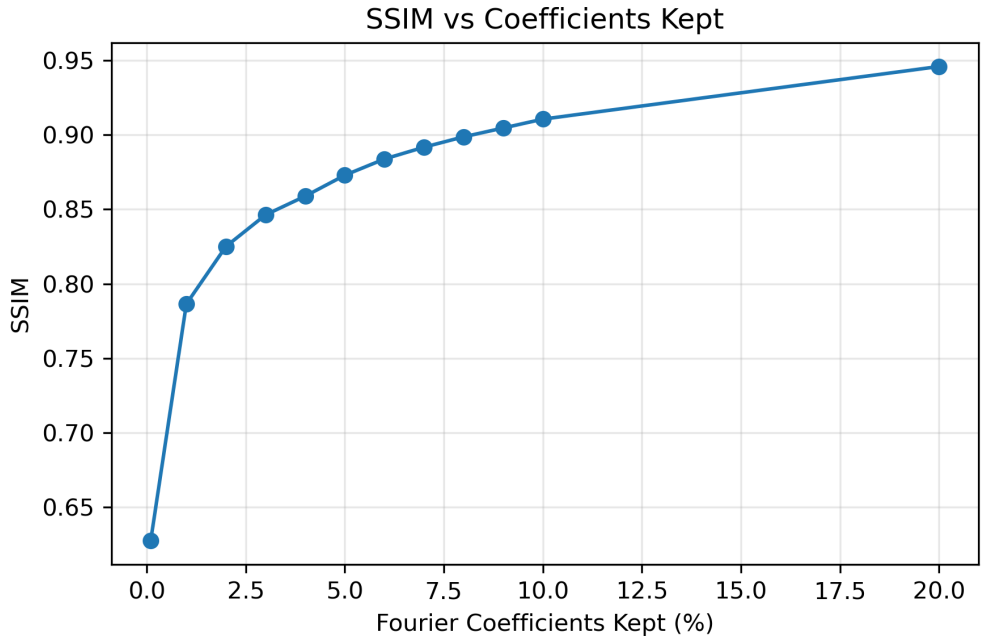


Figure 7: Structural Similarity Index

The Structural Similarity Index (SSIM) is a metric used to measure the similarity between two images by comparing luminance, contrast, and structure, where 1 indicates a perfect match. As shown in figure 7, SSIM rises sharply at very low keep-rates and then converges slowly to 1. Keeping more than 5% of the coeffisients only yield minor gains in similarity, which corresponds with the quality trends shown in figure 6.

Principal Component Analysis

1. PCA Implementation

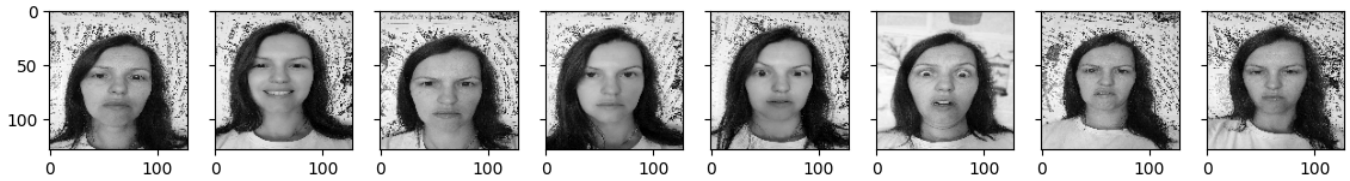


Figure 8: Original images

We loaded the original images in grayscale with a size of 128x128 pixels, then normalized all pixel values between 0 and 1.

The images were converted into a 2D matrix with image as row and pixel value as column.

A covariance matrix was calculated for the image matrix, and it was used to calculate eigenvalues and eigenvectors, sorted in descending order.

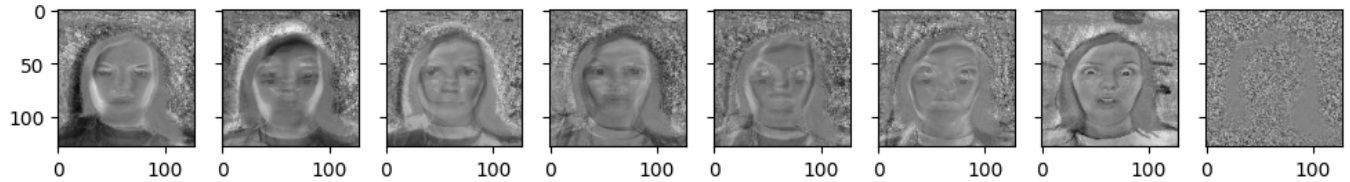


Figure 9: Principal Components

We selected the top k = 6 eigenvectors to start with, and will experiment with the amount later. These were used to create our principal components, as seen in figure 9.

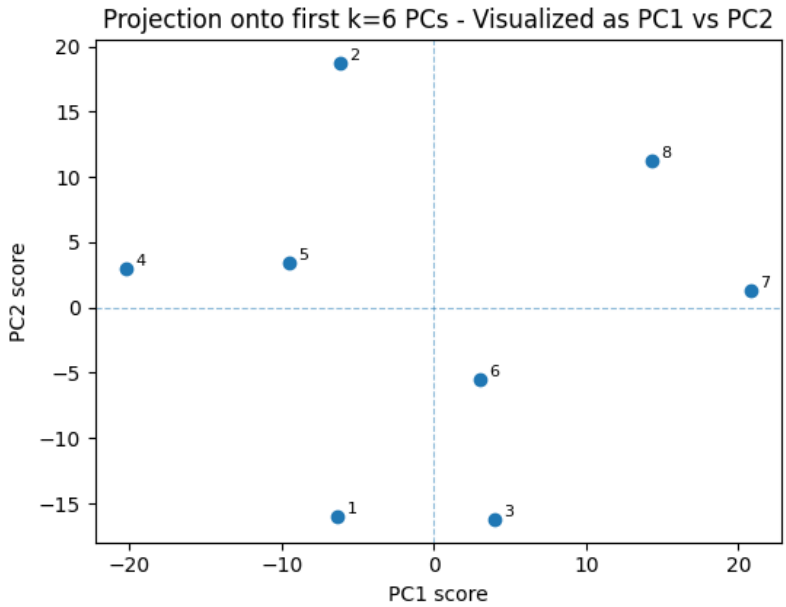


Figure 10: Images visualized as PC1 vs PC2

We visualized all 8 images in the 2-dimensional subspace defined by PC1 and PC2, shown in figure 10.

2. Reconstruction of images

a. Using the selected principal components, reconstruct the images.

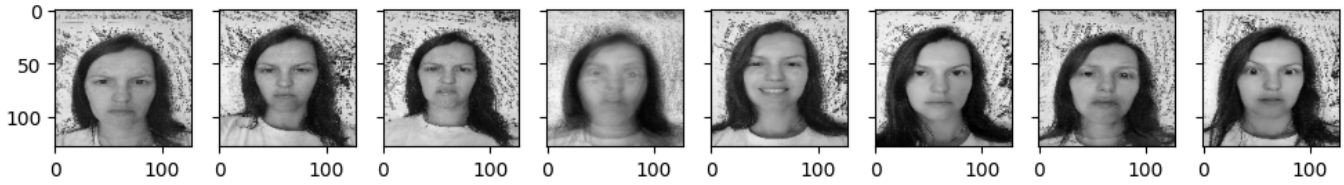


Figure 11: Images reconstructed with k = 6

b. Compare the reconstructed images with the original images to observe the effects of dimensionality reduction.

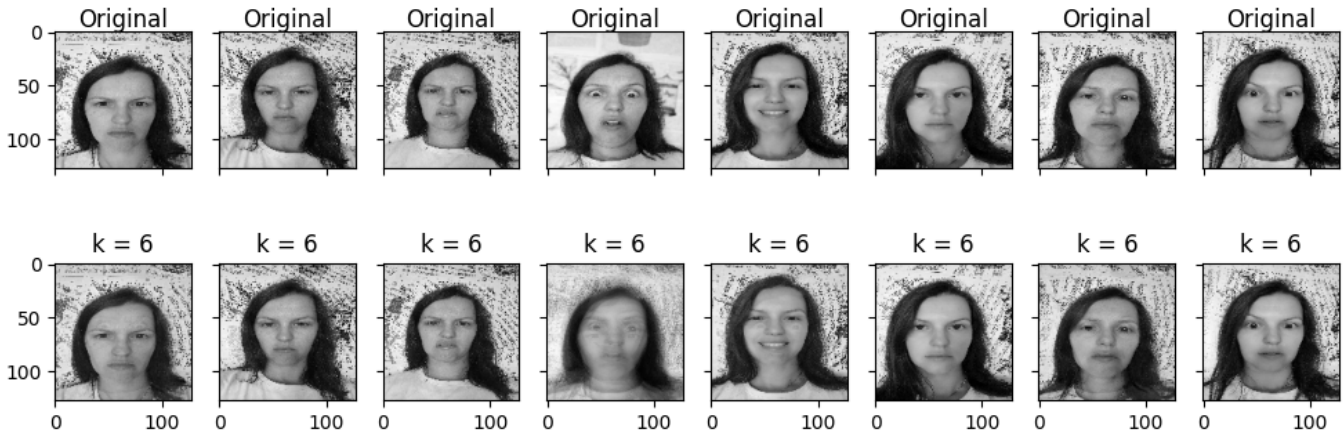


Figure 12: Original images vs reconstructed images with k = 6

3. Experimentation

a. Vary the number of principal components (k) and observe the impact on the quality of the reconstructed images.

Figure 13: Images reconstructed with different K's

We experimented with different k values for the reconstruction of the original images. Figure 13 illustrates how the reconstruction increasingly aporaches an aproximation of the original images with every aditional increase of k.

b. Plot the variance explained by the principal components and determine the optimal number of components that balances compression and quality.

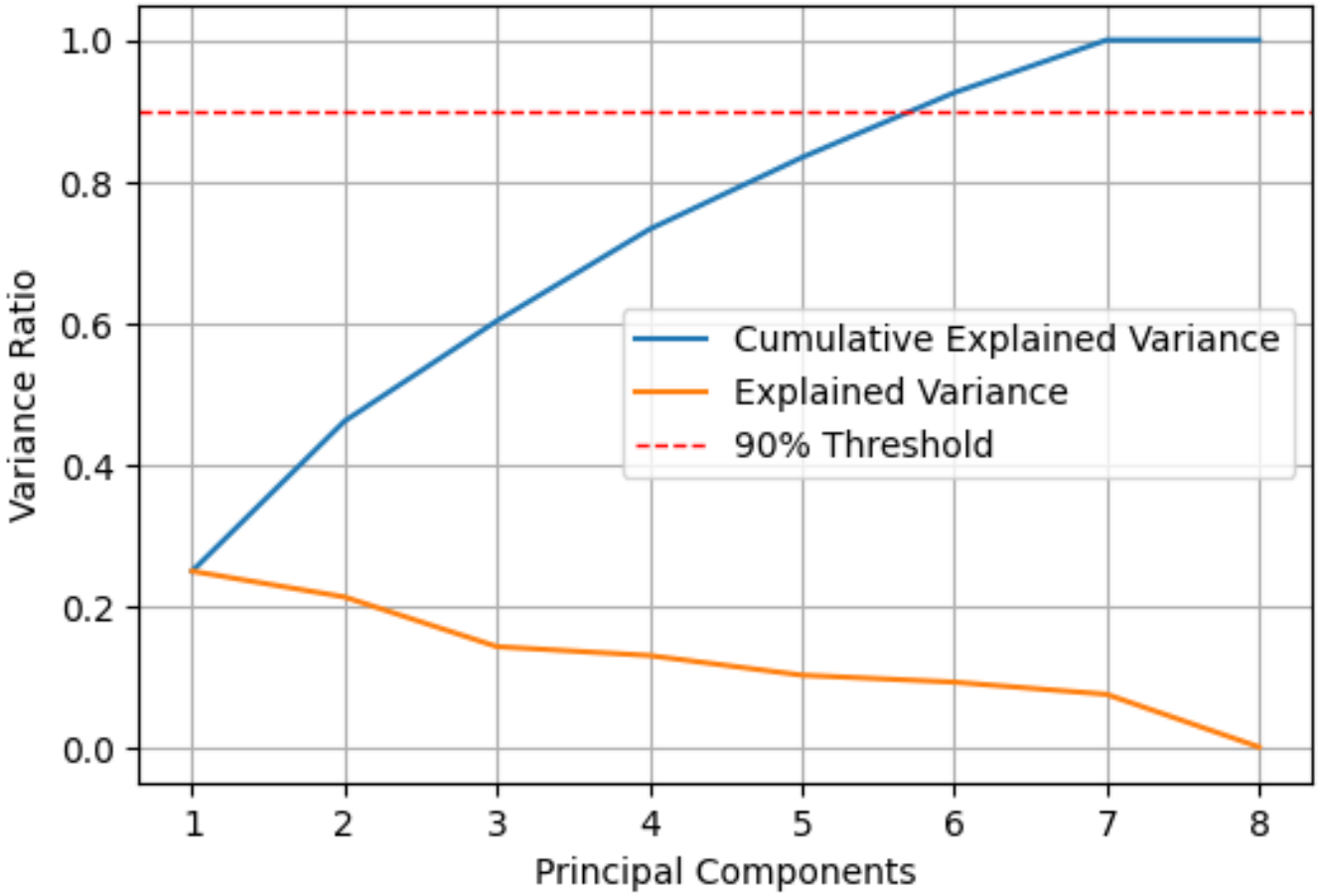


Figure 14: Plot for cumulative variance and individual variance per component

With a threshold of 90% we see that six prinicple components would be needed to reach this level. As seen in figure 13 the images reconstructed with less than six components are considerably more blurry. Given that our dataset is a facial emotions dataset would mean that blurry images are detrimental to the intended purpose of the dataset. However, using all seven components would aproximate a full reconstruction of the original images and would constitute little compression. It can therefore be argued that in our case, if we want to compress our images we could only use six priniciple componets before it would make the subjects emotions difficult to recognize.

4. Visual Analysis

a. Display the original images alongside the reconstructed images for different values of k.

The original images alongside the reconstructed images for different values of k is shown if figure 13.

b. Comment on the visual quality of the images and how much information is lost during compression.

As seen in figure 12, the reconstructed images are very similar to the originals, with the exeption of the fourth image from the left which is very blurry. You can still make out the expression in the image, but it's much harder than with the rest.

This problem can also be seen in figure 13, where reconstructions with lower values for k are more blurry than those with higher values.

You can also see a representation of the quality of the reconstruction using MSE in figure 15, and it will be discussed further in the next section.

5. Error Analysis

a. Compute the Mean Squared Error (MSE) between the original and reconstructed images.

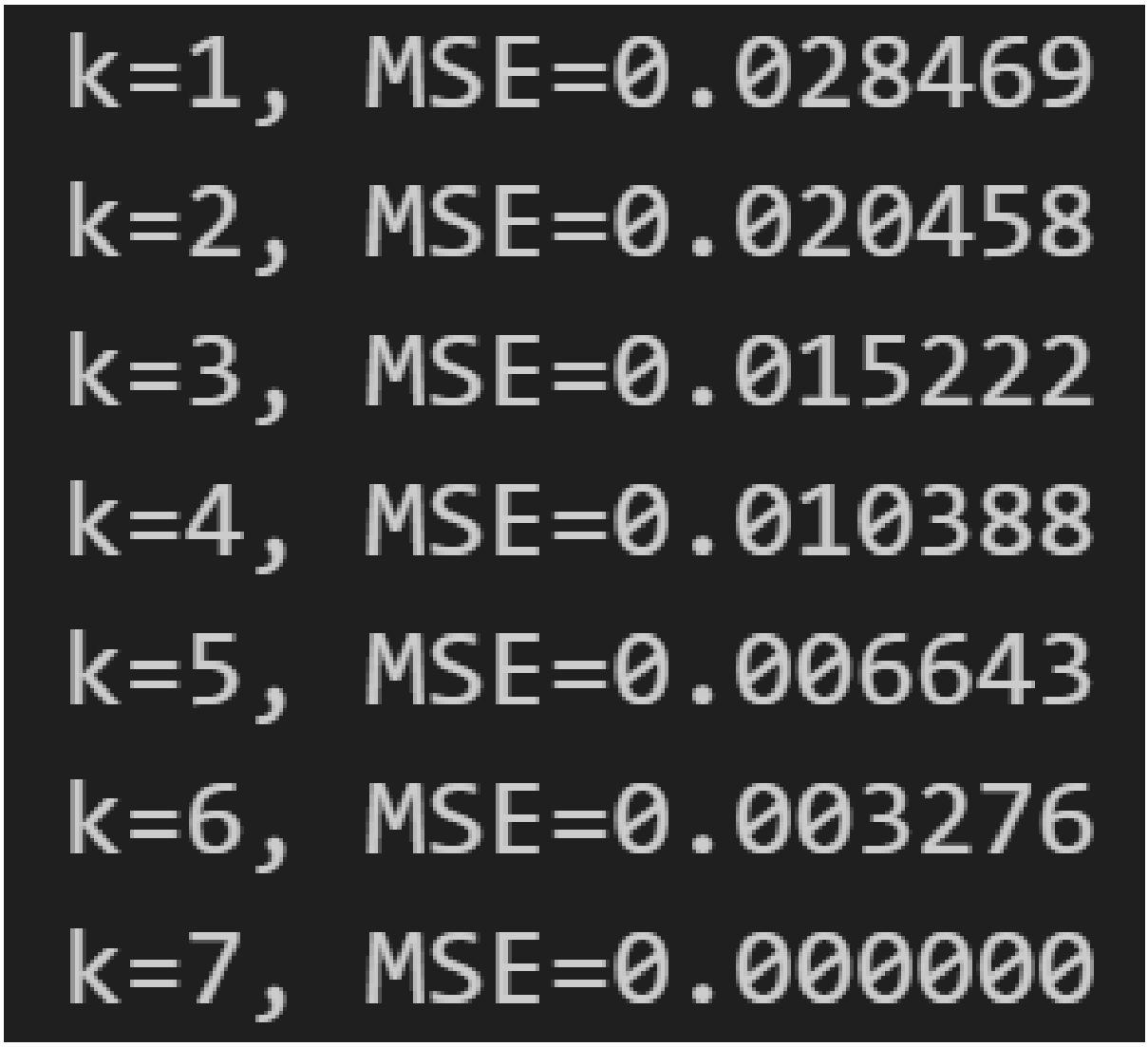


Figure 15: Orignal images comapred to reconstructed images with k = 6

b. Analyze the trade-off between compression and reconstruction error.

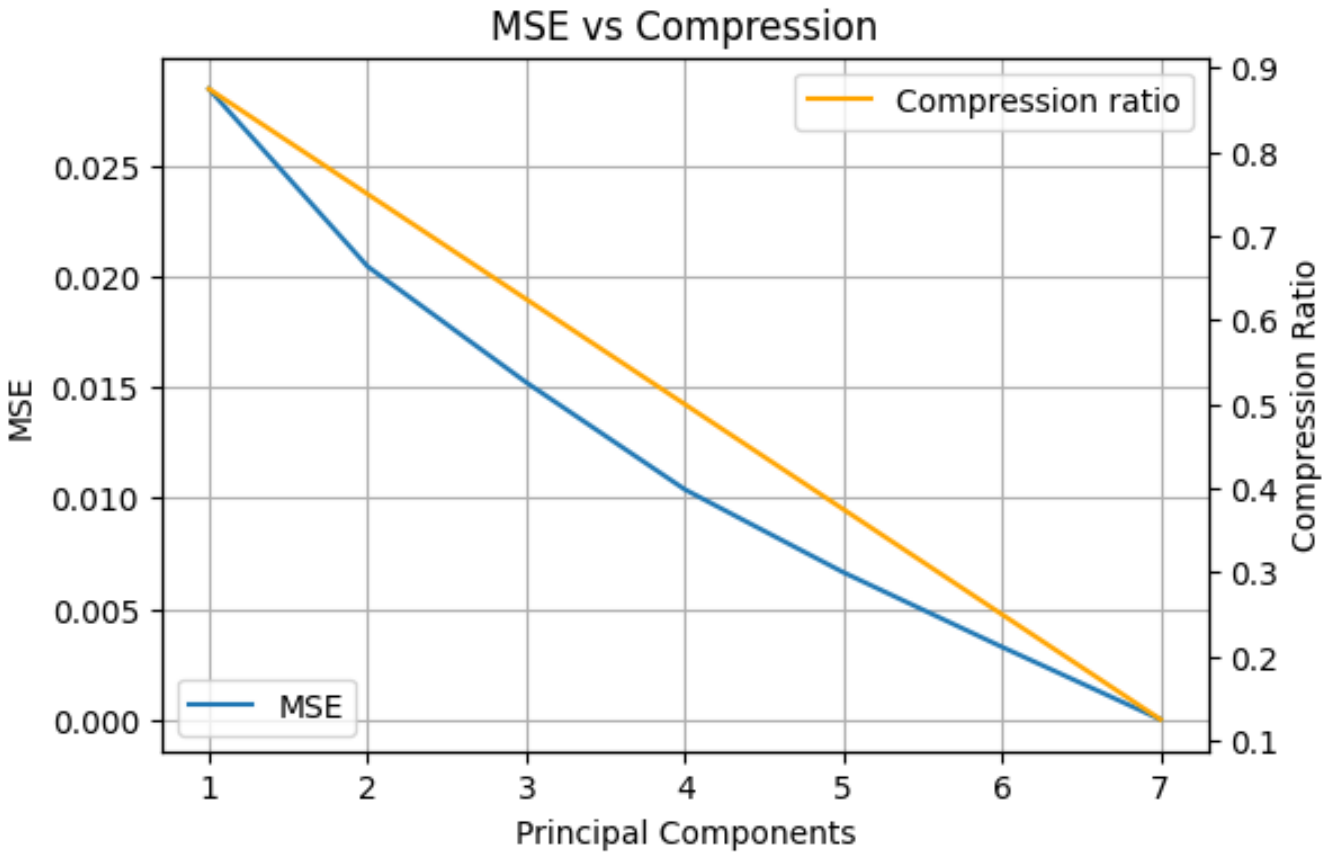


Figure 16: Orignal images comapred to reconstructed images with k = 6

In figure 16, we can see that the MSE is continually decreasing from 0.028 (k = 1) to 0.0033 (k =6), and becomes aproximally 0 at k = 7, which is in line with the dataset being rank n - 1 = 7. We chose k = 6 as a balance between compression and quality, with a PNSR of about 25 dB.

Histogram of Oriented Gradients

1. Write a Python script to compute the HOG features of a given image using a library such as OpenCV or scikit-image. Apply your implementation to at least three different images, including both simple and complex scenes.

Histogram of Oriented Gradients (HOG) features capture local shape by counting how often edges point in each direction within small regions, then normalizing and concatenating them into a feature vector. The x and y gradients are the horizontal and vertical changes in pixel intensity, revealing edge direction. The gradient magnitude is the overall edge strength at each pixel, used as the weight when voting into orientation bins. HOG is a feature descriptor used for object detection.

2. Visualize the original image, the gradient image, and the HOG feature image. Compare the HOG features extracted from different images.

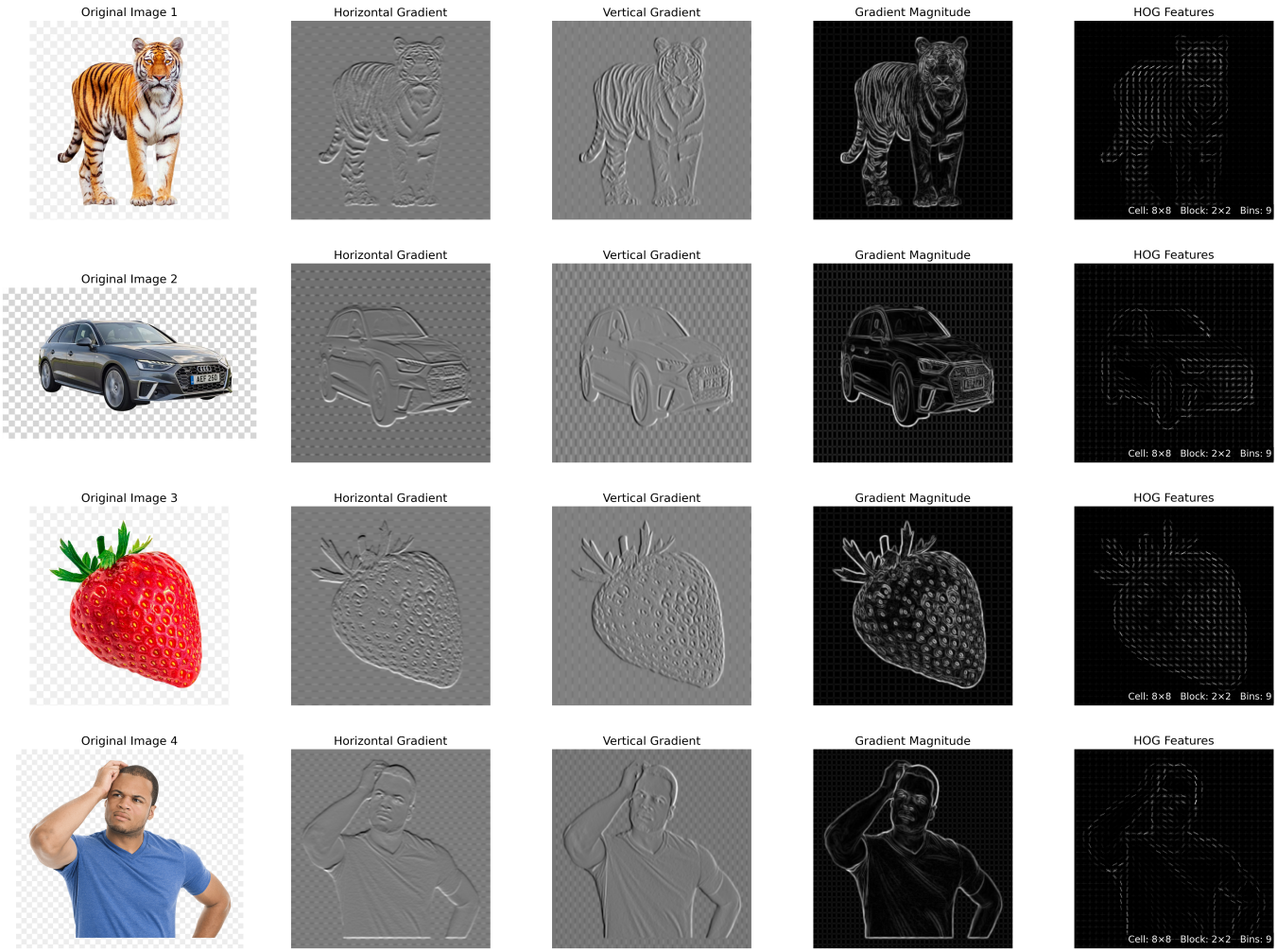
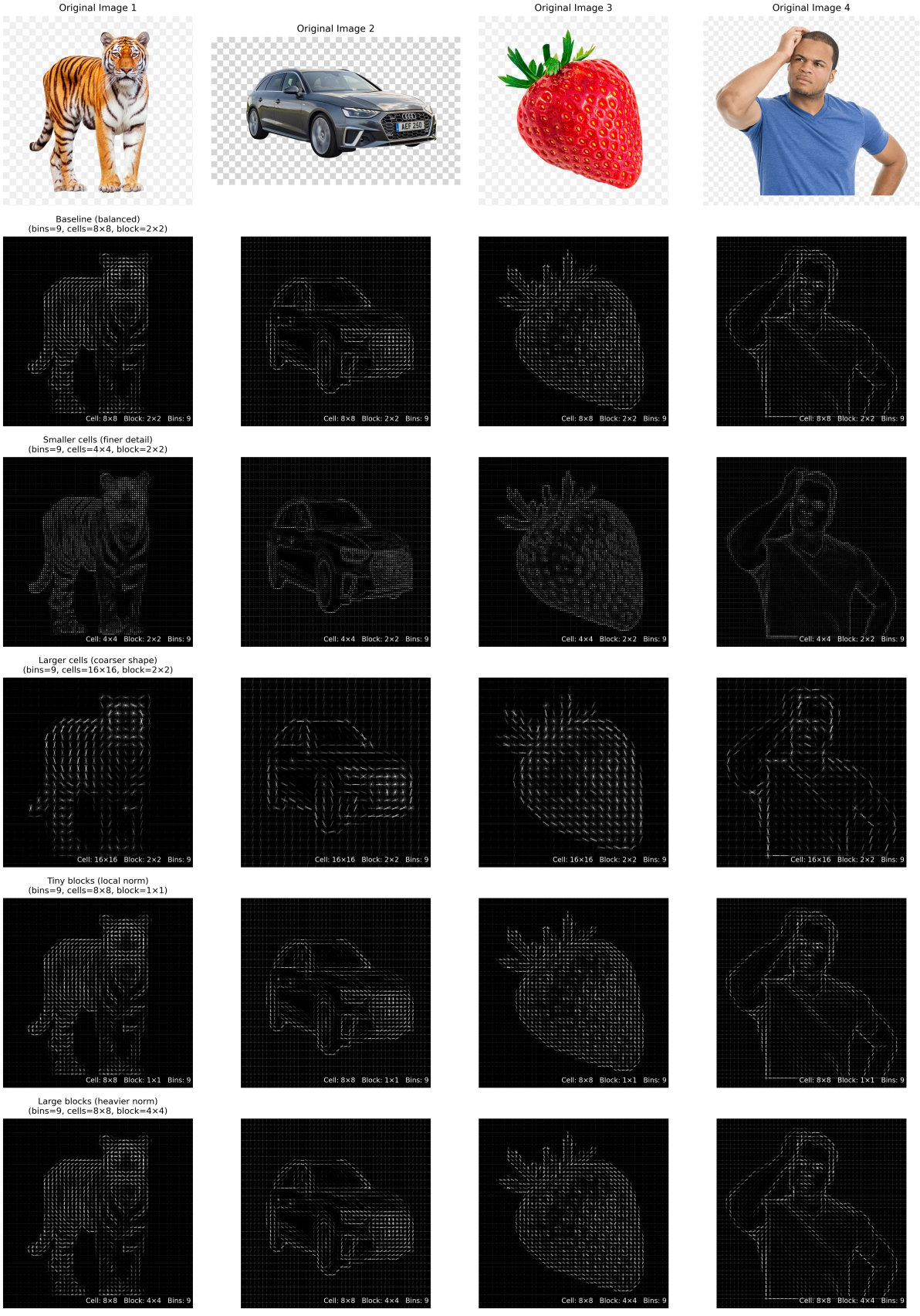


Figure 17: HOG features with baseline parameters

As shown in figure 17, HOG is better at capturing sharp edges and overall shape/contour than fine textures. Consequently, it renders the human and car contour more clearly since the original images contain fewer details and has well-defined edges. The strawberry and tiger are harder to recognize from HOG features because their appearances are dominated by fine textures as seeds and fur.

3. Discuss the impact of varying parameters like cell size, block size, and the number of bins on the resulting HOG descriptors.



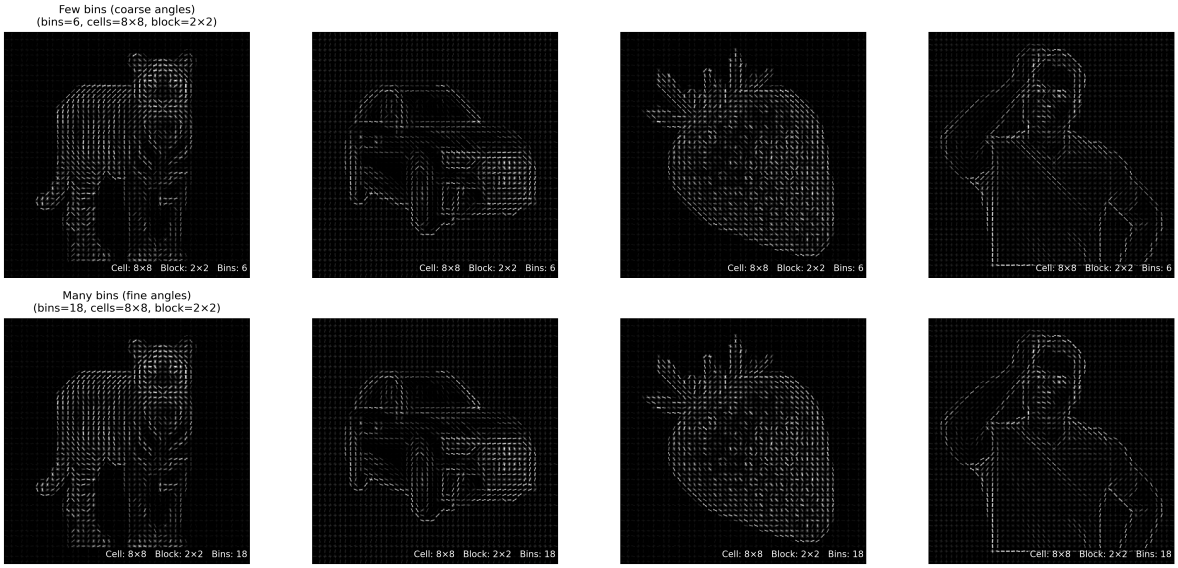


Figure 18: HOG features with different parameters

As shown in figure 18, using smaller cells (4×4) makes the HOG more sensitive to fine textures and details. This is most visible on the image of the strawberry, where individual seeds are much more recognizable compared to HOG features with other parameters. Smaller cells also improves the tiger image, revealing more fine fur detail. Larger cells (16×16) smooth local gradients and emphasize only the rough shape of the image.

Block size has less impact than cell size, but tiny blocks (1×1) preserve more local contrast and are less robust to illumination/contrast changes, while large blocks (4×4 cells) normalize gradients across a wider area, improving robustness to illumination/contrast changes, but slightly smooths local variation.

The amount of orientation also does not have the same impact as cell size, but fewer orientation bins (6) give more compact, coarse angle coding that highlights major contours, while many bins (18) capture subtle angle changes but can add redundancy/noise.

Local Binary Patterns

1. Write a Python function to compute the LBP of a given grayscale image (basic 8-neighbor). Your function should output the LBP image, where each pixel is replaced by its corresponding LBP value.

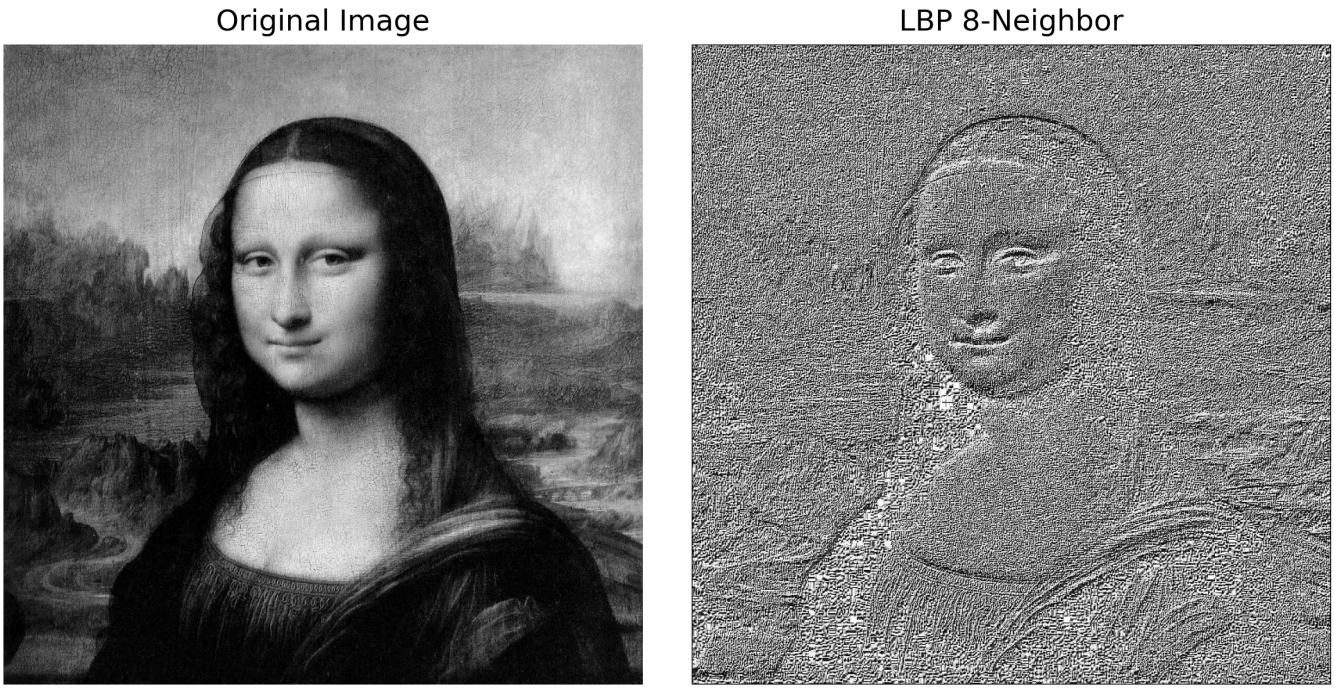


Figure 19: LBP

Local Binary Patterns (LBP) encodes local texture at each pixel by comparing the pixel's intensity to its eight immediate neighbors: 1 for each neighbor that is at least as bright as the center, otherwise 0. Reading these eight bits in a fixed order yields an 8-bit pattern that is converted to a decimal value in the range from 0 to 255, and the pixel in the LBP image is replaced by this value. Figure 19 presents the original image alongside its 8-neighbor LBP representation, looking at the pixels 1 radius (1 pixel) away from the center pixel.

2. Write a Python function to compute the histogram of the LBP image. Plot the histogram and explain what it represents in terms of the texture features of the image.

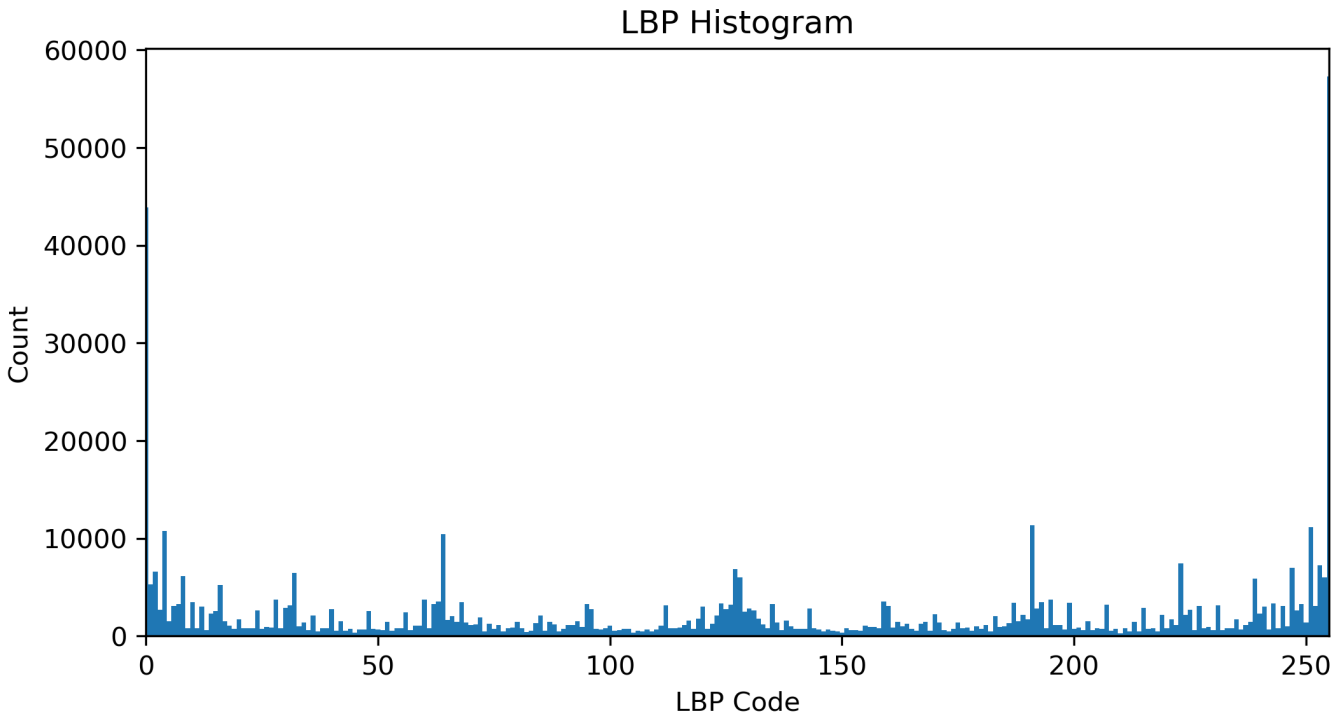


Figure 20: LBP histogram

An LBP histogram counts how many pixels in the LBP image have each code value from 0 to 255, and shows the texture distribution within the original image. The histogram is used to capture the frequency of occurrence of different texture patterns in the original image.

In figure 20, the LBP histogram shows tall peaks near the extreme codes (0 and 255), indicating many uniform patterns.

3. Apply your LBP function to at least three different grayscale images (e.g., a natural scene, a texture, and a face image). Generate and compare the histograms of the LBP images.

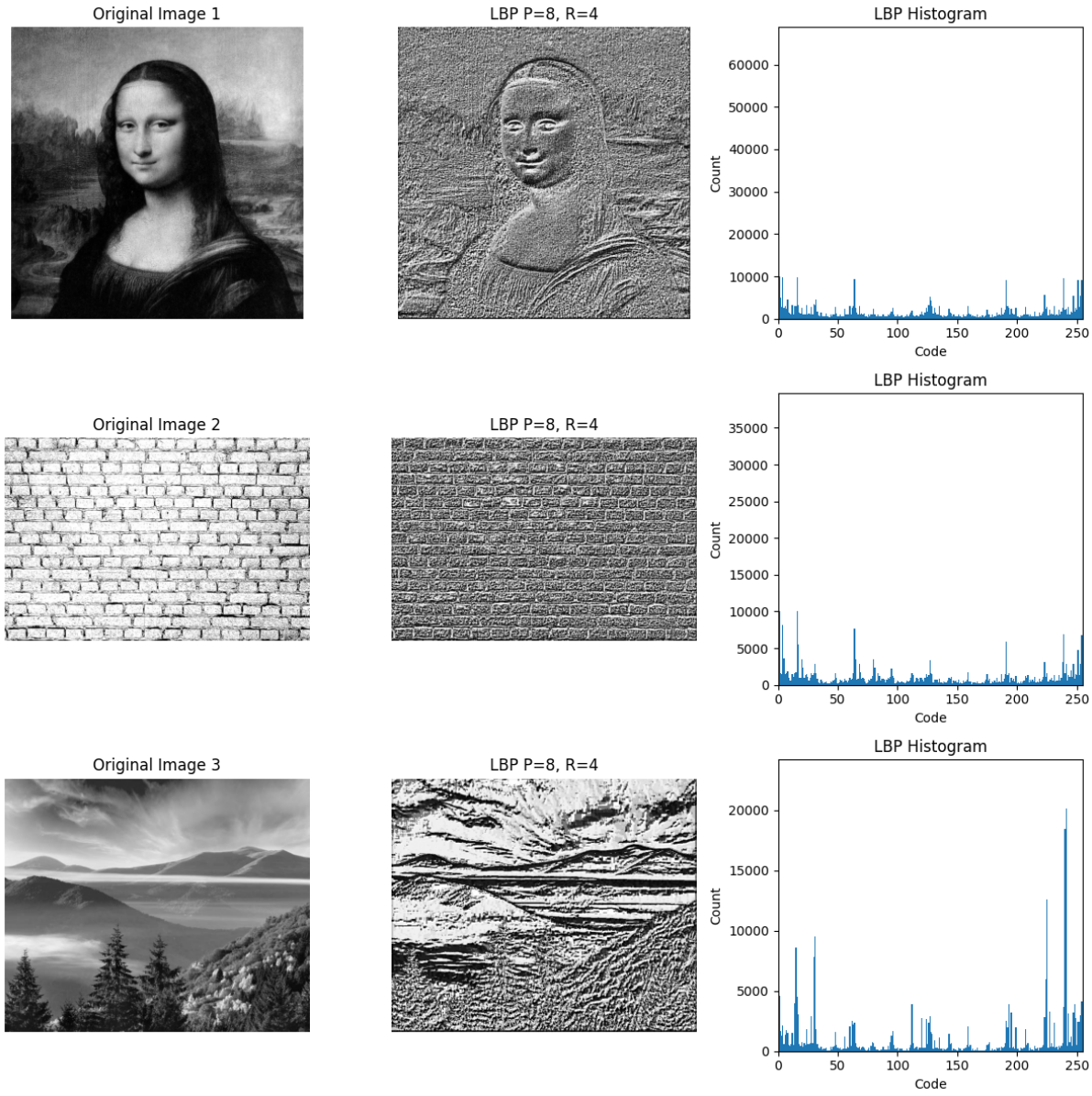


Figure 21: LBP for several images

The LBP histograms differs among the three images. The image of Mona Lisa's histogram has a more uniform distribution of pixel values, with smaller spikes and large spikes at 0 and 255. For this LBP (figure 21), we increased the radius from 1 pixel to 4, and compared to the LBP in figure 19, the edges of the subject and the countours of the background are much more preserved.

The image of the bricks wall's histogram has a slightly less uniform distribuiton, with more sparse spikes.

The image of the landscape's histogram is much less uniformly distributed, with tall, sparse spikes and not much between them.

4. Discuss the differences in the histograms and what they tell you about the textures of the different images.

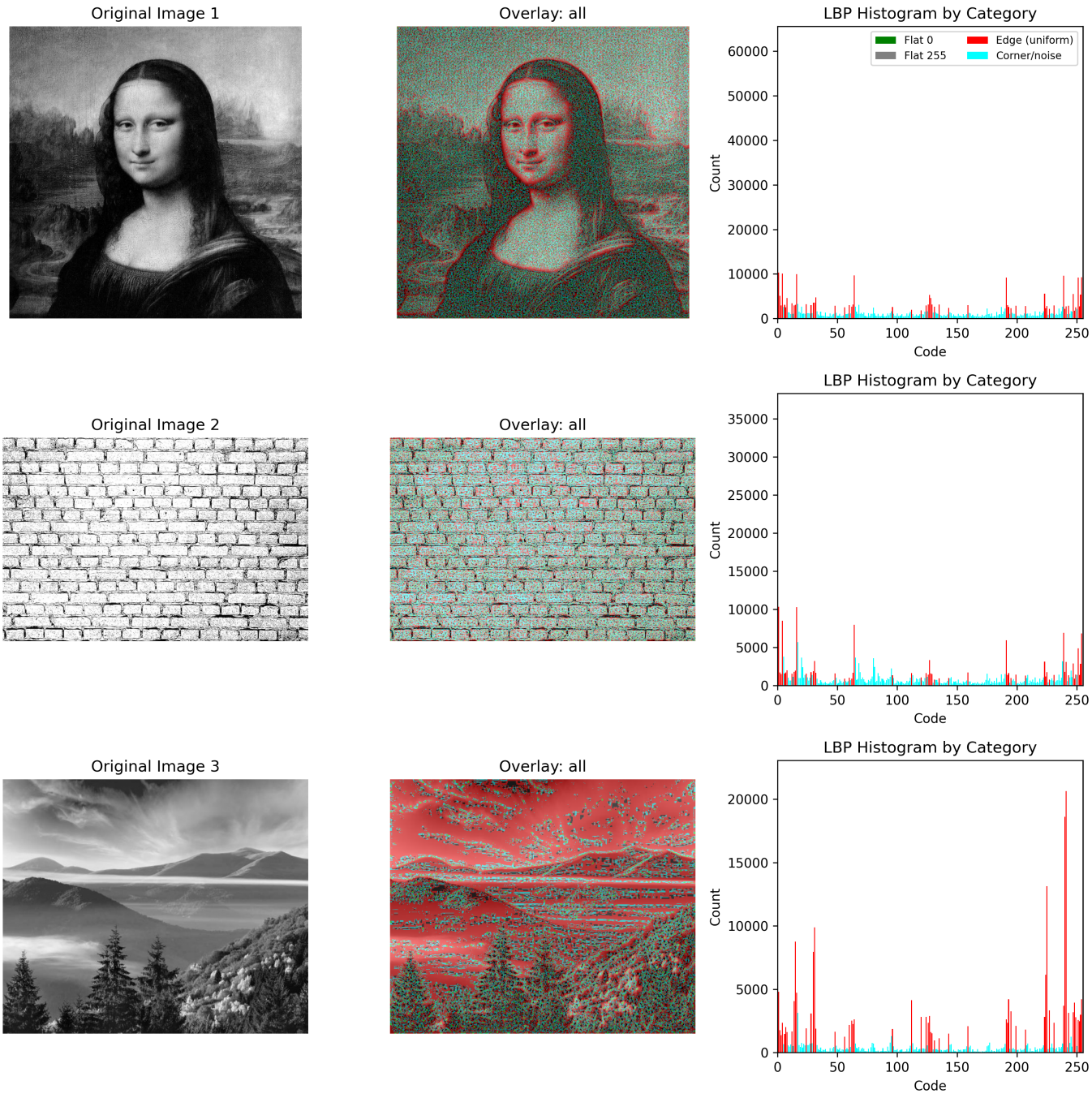


Figure 22: LBP by category for several images

We categorized LBP pixel values into 4 categories, flat 0, edge (uniform), flat 255, and corner/noise. The categories are decided based on the binary encoding of the surrounding pixel values, where the amount of transitions between 0 and 1 decide the category.

No transitions in the binary pixel value, e.g. 00000000, means it's categoriezed as flat, going into either flat 0 or flat 255, depending on whether every number is 0 or 1.

One or two transitions in the binary pixel value, e.g. 11000111, means it's classified as an edge.

More than two transitions, e.g. 10101010, are classified as a corner/noise.

After visualizing the LBP results in this way, it's much easier to interpret our results. In the image of mona lisa, you can see the edges are clearly marked in red, while the rest of the image is filled with flat 0, flat 255, and corner/noise.

The image of the brick wall has much more noise, and it also has many corners, so it's filled with way more pixels classified as corner/noise. Not many of the brick edges are correctly classified as edges, which is because of choosing a radius of 4, which isn't as good at recognizing thin edges like on the bricks. The higher radius makes the LBP recognize thich edges easier, but is worse at detecting thin edges.

The image of the forest is also not classified as well as the image of mona lisa. This time, most of the sky is classified as edges, which is caused by it having a gradient, which is interpreted as an edge when using LBP.

We selected the parameters of the LBP function to improve the categorization for the image of mona lisa, which is why it's categories seem more accurate than for the brick wall or the landscape.

To receive better results, we would perform a discrete fourier transform on the images, reducing noise. We would also vary the parameters, the amount of neighbours and the radius, based on the dataset. This is so the LBP pixel value categories fit better for all images in the dataset instead of maximising quality for one of them.

Implement a Blob Detection Algorithm.

1. Apply the blob detection algorithm to one of the provided image datasets on blackboard. Visualize the detected blobs on the original images, marking each detected blob with a circle or bounding box.

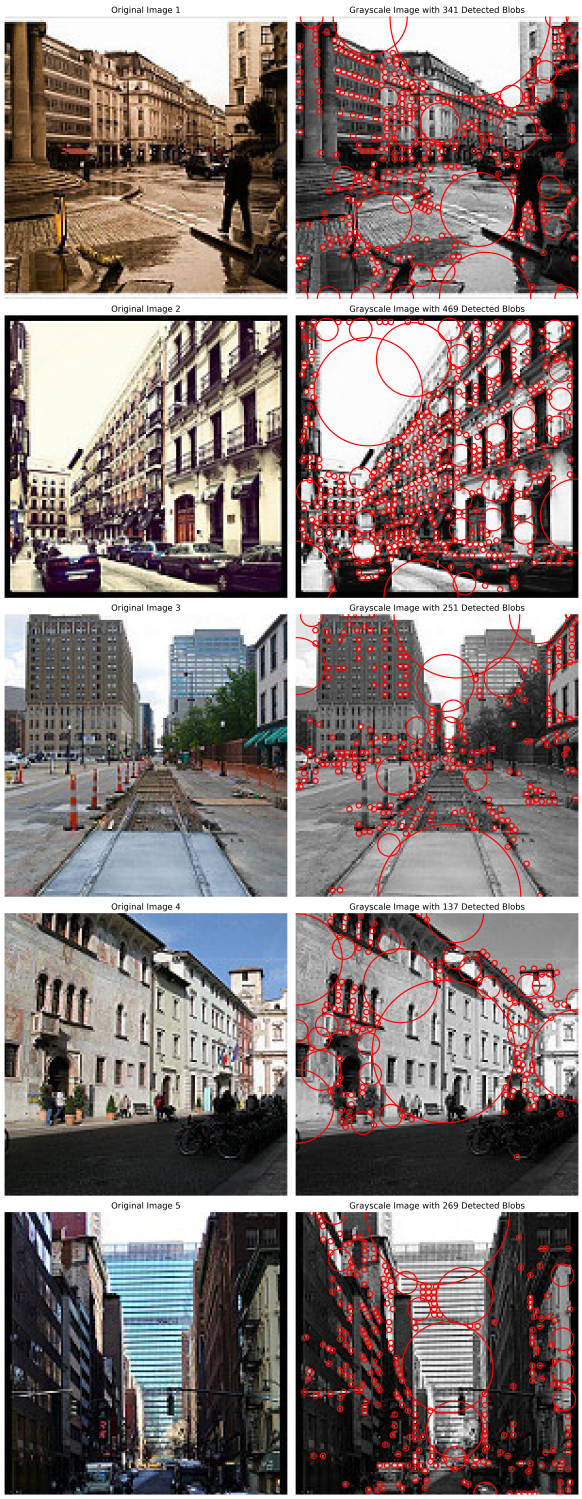


Figure 23: Blob detection applied on greyscale images

Blob detection algorithms identify regions in an image with distinct properties like brightness or color. Our implemented blob detection algorithm uses the Laplacian of Gaussian (LoG) method. The results are shown in figure 23.

2. Calculate and display relevant statistics for each image, such as the number of blobs detected, their sizes, and positions.

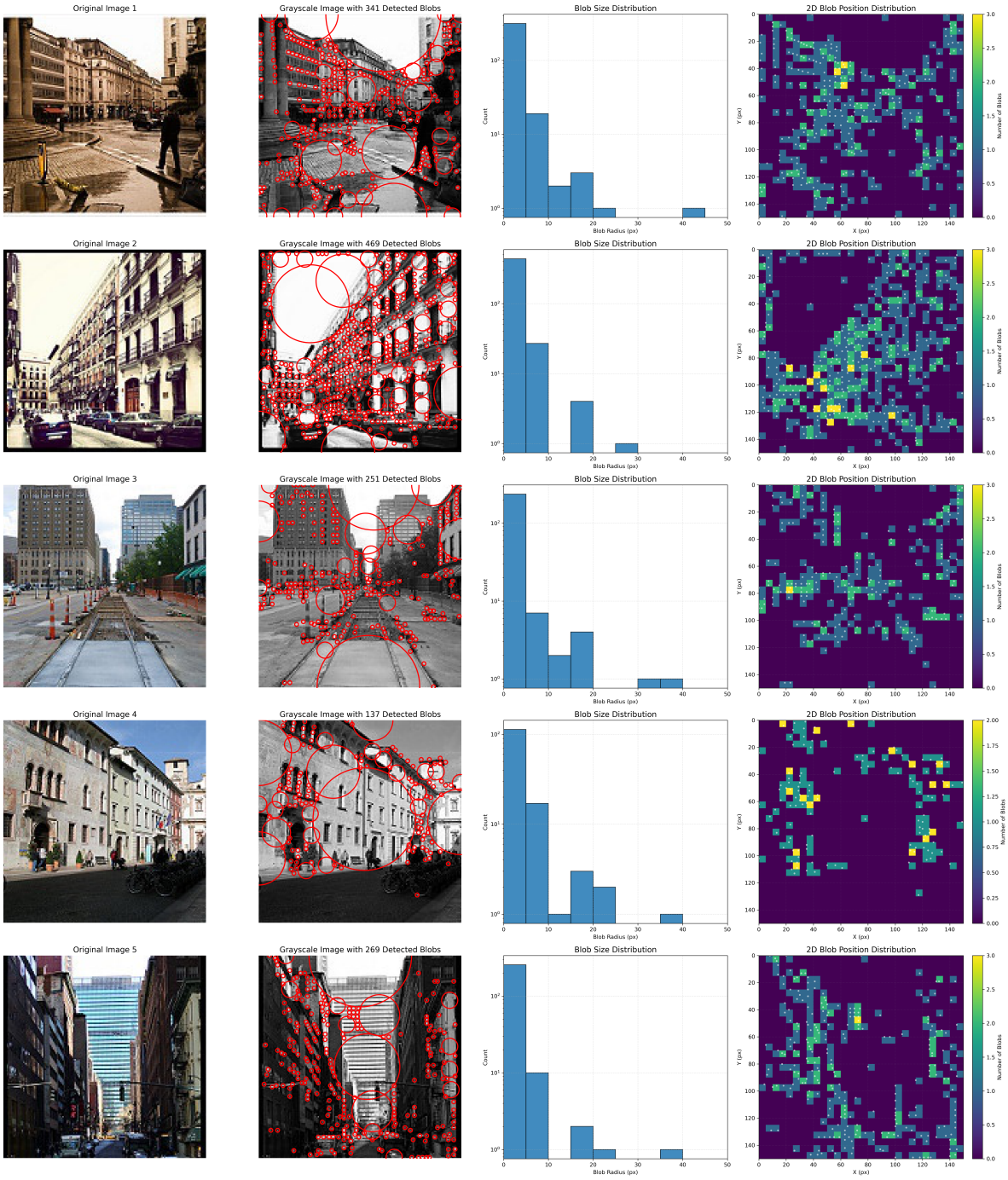


Figure 24: Blob detection statistic

The overlay of detected blobs on grayscale and RGB images helps confirm whether the blobs align with visually identifiable features or not.

Histograms of blob sizes reveal the distribution of detected radius across images and can indicate whether certain sizes are being over- or under-represented.

The 2D heatmaps of blob positions show where blobs tend to occur spatially, revealing patterns or clustering, and can also highlight issues such as biased detection in bright regions due to thresholding.

3. Evaluate and discuss the effect of different parameters in the algorithms on the detection of different blobs.

The `max_sigma` parameter defines the maximum standard deviation for the Gaussian kernel and essentially sets the upper limit for the size of blobs that can be detected. We have set this to 30, which allows detection of relatively large blobs.

If `max_sigma` is set too low, larger blobs will not be detected at all. On the other hand, a high value can lead to the detection of large, low-contrast regions that may not correspond to meaningful features.

The `num_sigma` parameter defines how many intermediate scales are tested between 0 and `max_sigma`. We set a value value of 10 so to check 10 different scales. Increasing this number can improve the precision of blob detection, especially for blobs that do not fall neatly into one of the predefined scales. However this can also greatly increases computational complexity.

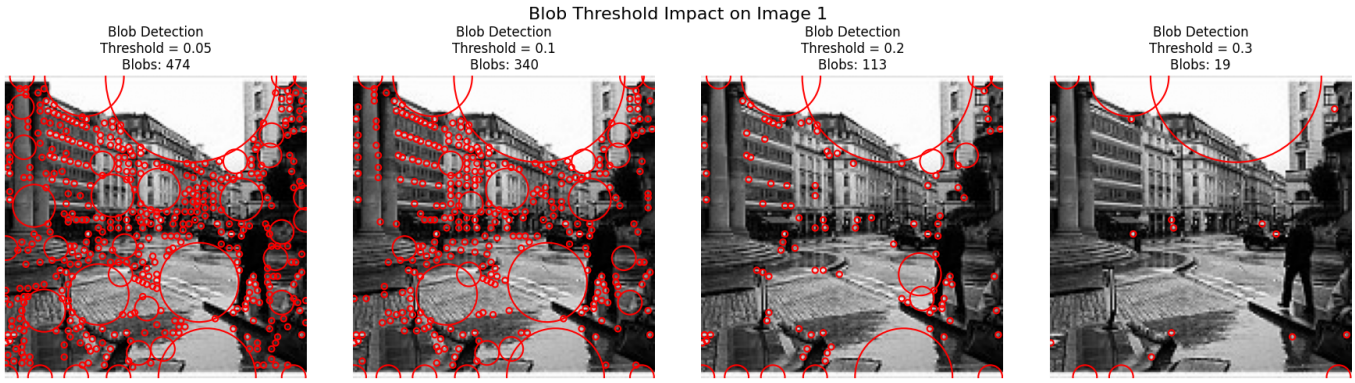


Figure 25: Blob detection with different thresholds

The `threshold` parameter determines the minimum intensity difference required for a region to be considered a blob. A low `threshold` like 0.05 makes the algorithm more sensitive, allowing it to detect faint or low-contrast blobs, but it may also detect noise, as seen in the leftmost image in figure 25. Conversely, a high `threshold` like 0.3 makes the detection stricter, potentially missing subtle features while reducing false positives, as seein in the rightmost image.

Implement a Contour Detection Algorithm

1. Apply the contour detection algorithm to the same image dataset. Visualize the detected contours on the original images, marking each contour with a different color.

Figure 26: Contour detection applied to greyscale images

Contour detection algorithms aim to identify and extract the boundaries of objects within an image, often represented as a sequence of connected points or curves. Our implemented contour detection algorithm uses the Marching Squares method. The results are shown in figure 26.

2. Calculate and display relevant statistics for each image, such as the number of contours detected, contour area, and perimeter.

```
=== Image 1 ===
Total contours detected: 90
Contours considered (area > 10 & perimeter > 0): 12
  Contour 1: Area = 3087.0 px, Perimeter = 466.7 px
  Contour 2: Area = 20.0 px, Perimeter = 21.5 px
  Contour 3: Area = 70.0 px, Perimeter = 38.5 px
  Contour 4: Area = 482.0 px, Perimeter = 306.2 px
  Contour 5: Area = 69.0 px, Perimeter = 58.5 px
  Contour 6: Area = 321.0 px, Perimeter = 125.3 px
  Contour 7: Area = 11.0 px, Perimeter = 18.1 px
  Contour 8: Area = 90.0 px, Perimeter = 65.6 px
  Contour 9: Area = 3494.0 px, Perimeter = 1010.9 px
  Contour 10: Area = 15.0 px, Perimeter = 25.6 px
  Contour 11: Area = 24.0 px, Perimeter = 42.7 px
  Contour 12: Area = 33.0 px, Perimeter = 38.4 px
  Total Area: 7716.0 px
  Total Perimeter: 2218.0 px
  Mean Area: 643.0 px, Standard Deviation: 1194.8
  Mean Perimeter: 184.8 px, Standard Deviation: 281.8
```

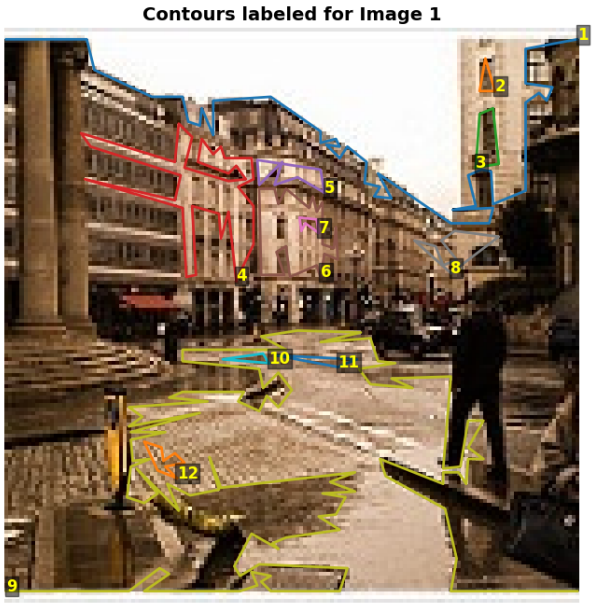


Figure 27: Statistics for contour detection on image 1

```
=== Image 2 ===
Total contours detected: 120
Contours considered (area > 10 & perimeter > 0): 16
  Contour 1: Area = 6556.0 px, Perimeter = 963.7 px
  Contour 2: Area = 628.0 px, Perimeter = 290.9 px
  Contour 3: Area = 145.0 px, Perimeter = 53.2 px
  Contour 4: Area = 2811.0 px, Perimeter = 849.6 px
  Contour 5: Area = 16.0 px, Perimeter = 26.8 px
  Contour 6: Area = 23.0 px, Perimeter = 21.6 px
  Contour 7: Area = 12.0 px, Perimeter = 17.8 px
  Contour 8: Area = 25.0 px, Perimeter = 34.1 px
  Contour 9: Area = 20.0 px, Perimeter = 27.8 px
  Contour 10: Area = 533.0 px, Perimeter = 129.0 px
  Contour 11: Area = 62.0 px, Perimeter = 38.6 px
  Contour 12: Area = 100.0 px, Perimeter = 45.2 px
  Contour 13: Area = 277.0 px, Perimeter = 68.5 px
  Contour 14: Area = 359.0 px, Perimeter = 127.6 px
  Contour 15: Area = 16.0 px, Perimeter = 14.1 px
  Contour 16: Area = 600.0 px, Perimeter = 147.4 px
  Total Area: 12183.0 px
  Total Perimeter: 2856.0 px
  Mean Area: 761.4 px, Standard Deviation: 1637.2
  Mean Perimeter: 178.5 px, Standard Deviation: 284.5
```

Figure 28: Statistics for contour detection on image 2

```
=== Image 3 ===
Total contours detected: 136
Contours considered (area > 10 & perimeter > 0): 19
  Contour 1: Area = 245.0 px, Perimeter = 119.2 px
  Contour 2: Area = 2783.0 px, Perimeter = 533.3 px
  Contour 3: Area = 421.0 px, Perimeter = 271.3 px
  Contour 4: Area = 35.0 px, Perimeter = 49.1 px
  Contour 5: Area = 28.0 px, Perimeter = 44.0 px
  Contour 6: Area = 41.0 px, Perimeter = 71.3 px
  Contour 7: Area = 98.0 px, Perimeter = 61.3 px
  Contour 8: Area = 107.0 px, Perimeter = 75.2 px
  Contour 9: Area = 441.0 px, Perimeter = 143.0 px
  Contour 10: Area = 1541.0 px, Perimeter = 349.9 px
  Contour 11: Area = 123.0 px, Perimeter = 97.2 px
  Contour 12: Area = 50.0 px, Perimeter = 69.4 px
  Contour 13: Area = 721.0 px, Perimeter = 201.3 px
  Contour 14: Area = 2943.0 px, Perimeter = 304.4 px
  Contour 15: Area = 15.0 px, Perimeter = 18.9 px
  Contour 16: Area = 62.0 px, Perimeter = 41.2 px
  Contour 17: Area = 36.0 px, Perimeter = 64.5 px
  Contour 18: Area = 36.0 px, Perimeter = 61.8 px
  Contour 19: Area = 45.0 px, Perimeter = 37.6 px
  Total Area: 9771.0 px
  Total Perimeter: 2613.9 px
  Mean Area: 514.3 px, Standard Deviation: 881.4
  Mean Perimeter: 137.6 px, Standard Deviation: 132.4
```

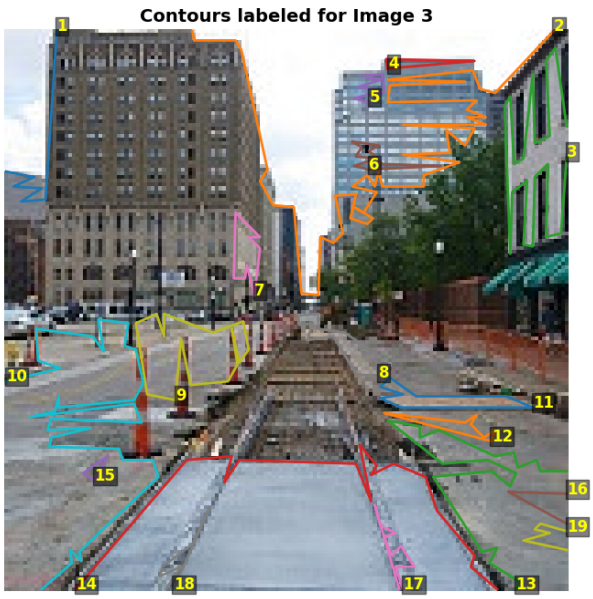


Figure 29: Statistics for contour detection on image 3

```
=== Image 4 ===
Total contours detected: 145
Contours considered (area > 10 & perimeter > 0): 22
  Contour 1: Area = 1727.0 px, Perimeter = 515.3 px
  Contour 2: Area = 37.0 px, Perimeter = 46.1 px
  Contour 3: Area = 73.0 px, Perimeter = 44.9 px
  Contour 4: Area = 1155.0 px, Perimeter = 433.0 px
  Contour 5: Area = 19.0 px, Perimeter = 30.2 px
  Contour 6: Area = 24.0 px, Perimeter = 32.1 px
  Contour 7: Area = 45.0 px, Perimeter = 60.7 px
  Contour 8: Area = 132.0 px, Perimeter = 49.9 px
  Contour 9: Area = 32.0 px, Perimeter = 42.4 px
  Contour 10: Area = 14.0 px, Perimeter = 28.0 px
  Contour 11: Area = 54.0 px, Perimeter = 34.2 px
  Contour 12: Area = 22.0 px, Perimeter = 28.3 px
  Contour 13: Area = 13.0 px, Perimeter = 24.6 px
  Contour 14: Area = 12.0 px, Perimeter = 25.9 px
  Contour 15: Area = 766.0 px, Perimeter = 414.1 px
  Contour 16: Area = 20.0 px, Perimeter = 18.8 px
  Contour 17: Area = 32.0 px, Perimeter = 48.0 px
  Contour 18: Area = 19.0 px, Perimeter = 23.6 px
  Contour 19: Area = 25.0 px, Perimeter = 22.7 px
  Contour 20: Area = 42.0 px, Perimeter = 29.7 px
  Contour 21: Area = 42.0 px, Perimeter = 25.1 px
...
  Total Area: 4316.0 px
  Total Perimeter: 1996.5 px
  Mean Area: 196.2 px, Standard Deviation: 431.4
  Mean Perimeter: 90.8 px, Standard Deviation: 145.7
```

Figure 30: Statistics for contour detection on image 4

```
=== Image 5 ===
Total contours detected: 78
Contours considered (area > 10 & perimeter > 0): 9
  Contour 1: Area = 207.0 px, Perimeter = 113.5 px
  Contour 2: Area = 4778.0 px, Perimeter = 793.6 px
  Contour 3: Area = 245.0 px, Perimeter = 114.6 px
  Contour 4: Area = 480.0 px, Perimeter = 150.3 px
  Contour 5: Area = 63.0 px, Perimeter = 85.5 px
  Contour 6: Area = 86.0 px, Perimeter = 50.6 px
  Contour 7: Area = 102.0 px, Perimeter = 89.0 px
  Contour 8: Area = 26.0 px, Perimeter = 44.2 px
  Contour 9: Area = 69.0 px, Perimeter = 52.2 px
  Total Area: 6056.0 px
  Total Perimeter: 1493.5 px
  Mean Area: 672.9 px, Standard Deviation: 1457.3
  Mean Perimeter: 165.9 px, Standard Deviation: 224.4
```

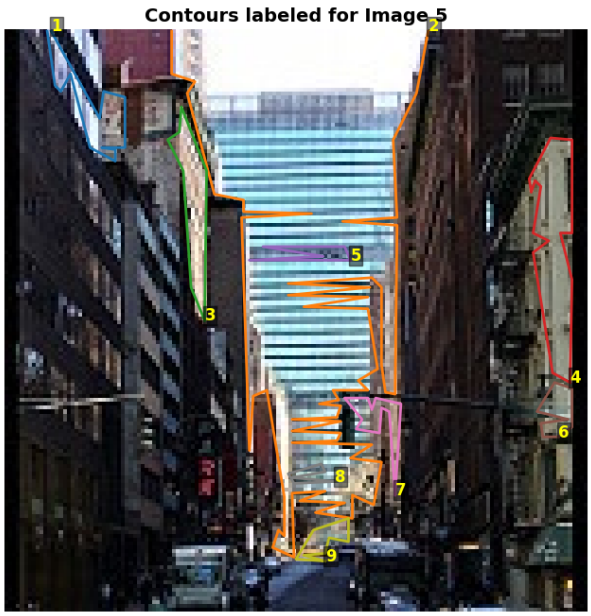


Figure 31: Statistics for contour detection on image 5

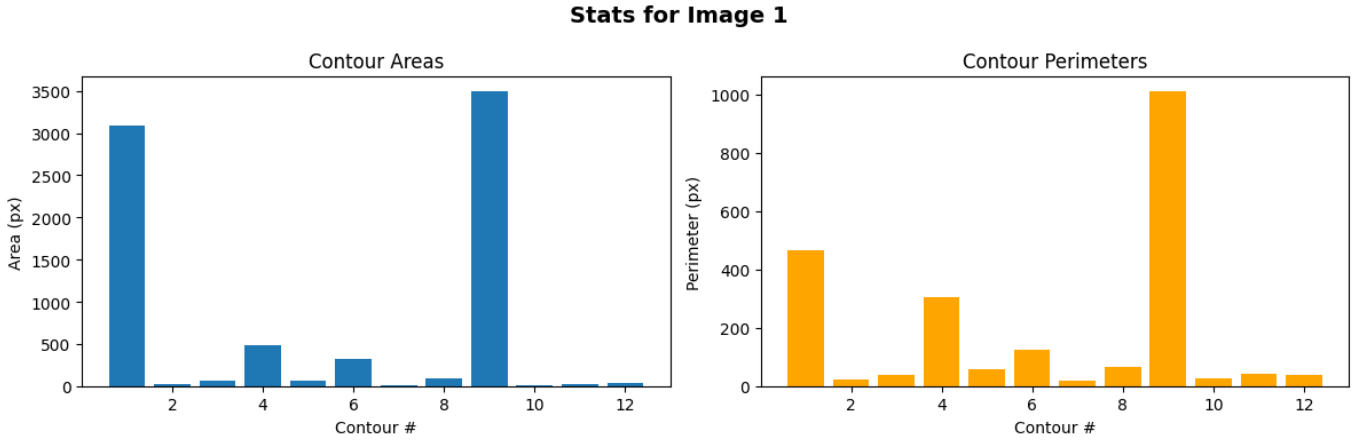


Figure 32: Bar plot of statistics for contour detection on image 1

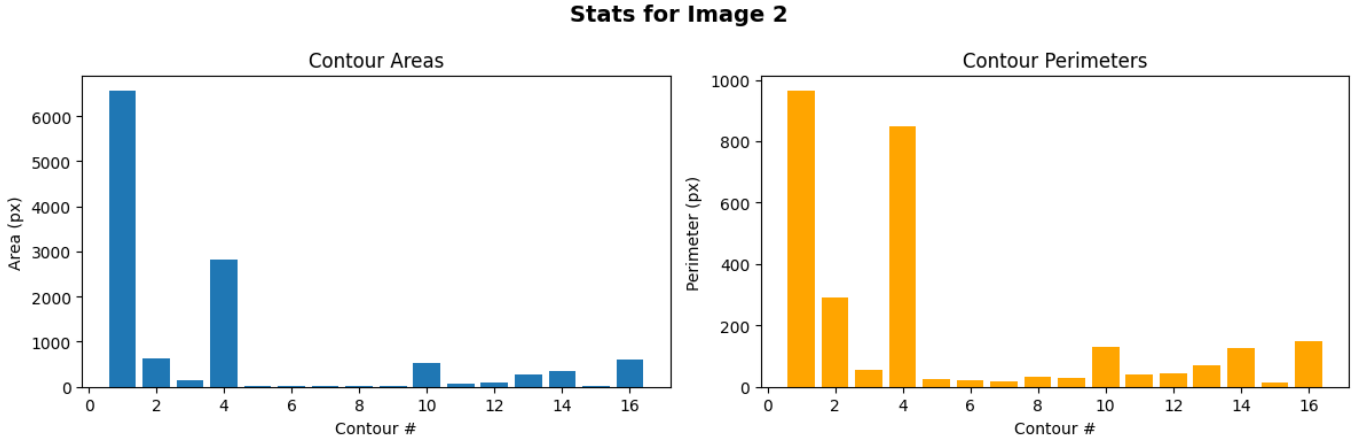


Figure 33: Histogram of statistics for contour detection on image 2

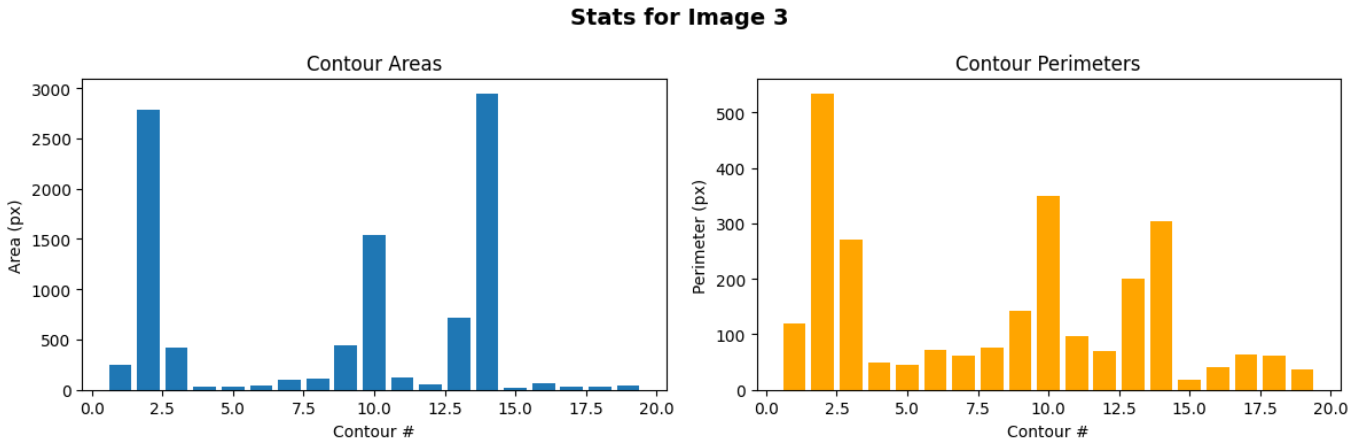


Figure 34: Bar plot of statistics for contour detection on image 3

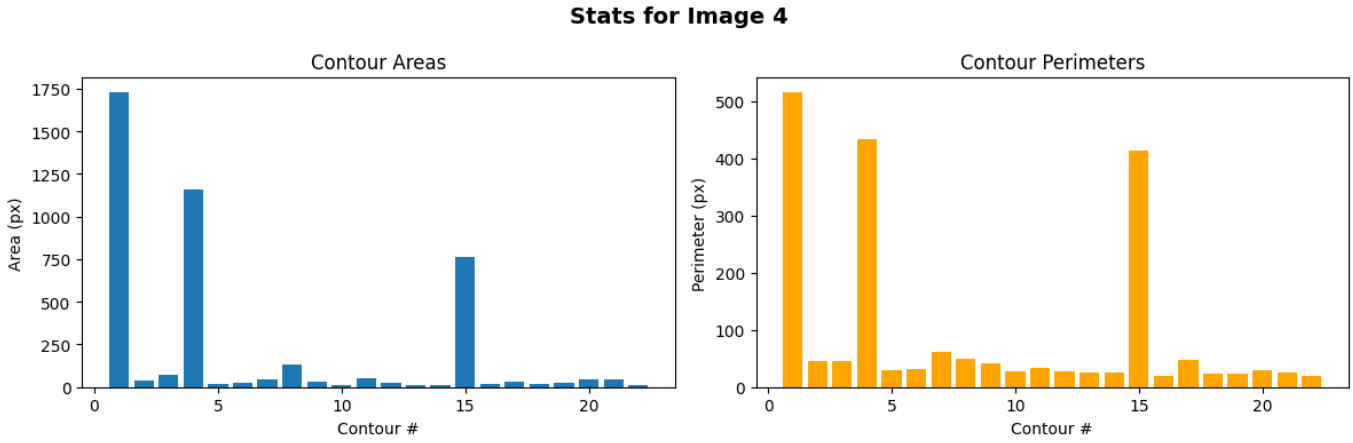


Figure 35: Bar plot of statistics for contour detection on image 4

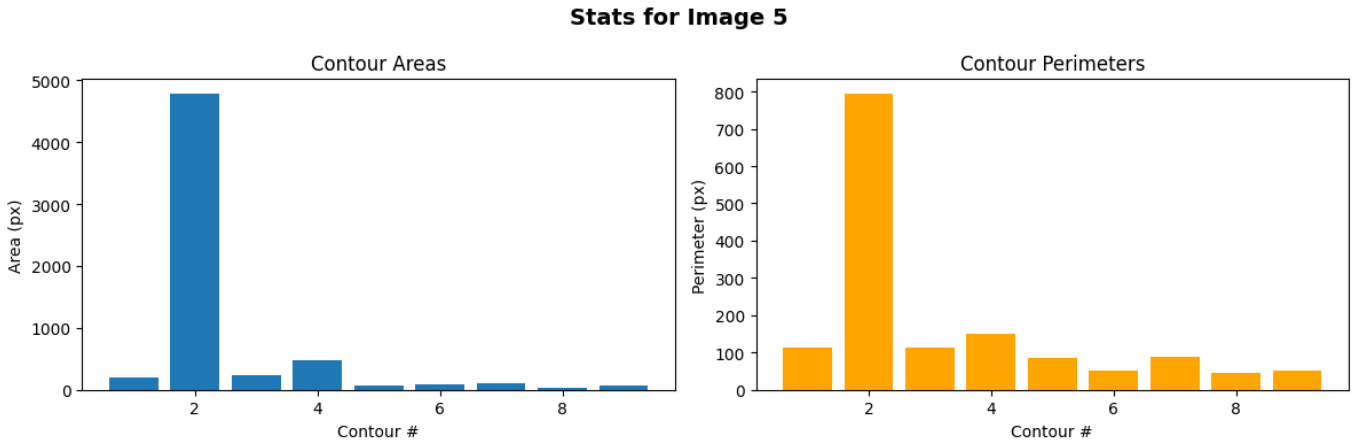


Figure 36: Bar plot of statistics for contour detection on image 5

3. Compare the results of blob detection and contour detection for the chosen dataset.

Figure 37: Blob- and contour detection applied to images (blob detection - red, countour detection - green)

Blob detection excels at highlighting regions that stand out from their surroundings. In image 3 of figure 37 (with tram tracks), blob detection better detects the windows on the left building. This is because the pixels differ in intensity from their surroundings, but there are no hard edges.

In contrast, contour detection is strongest at tracing boundaries and sharp intensity changes, so it better captures the right-hand building's windows because of its hard edges. Similar patterns can be seen in the other images.

4. Discuss the advantages and limitations of each technique.

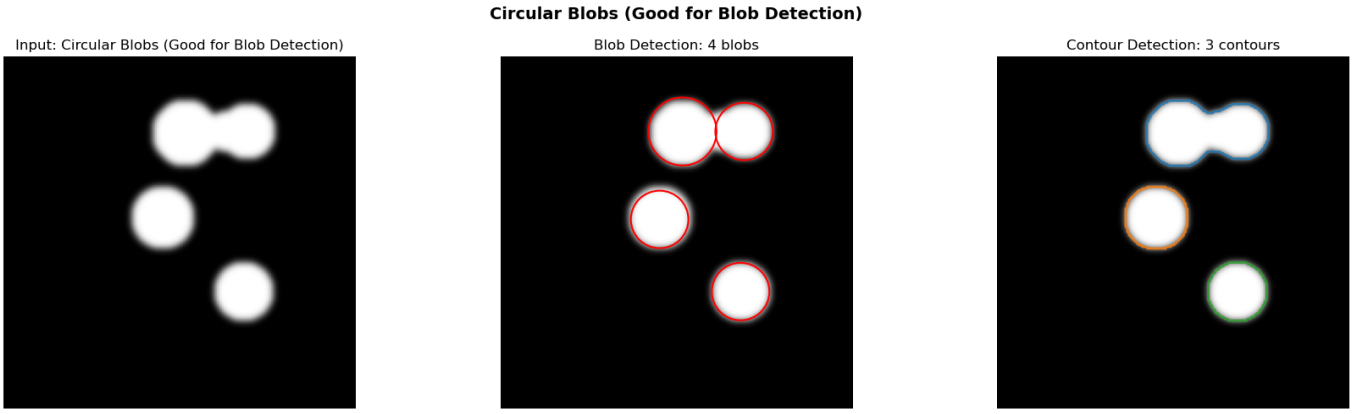


Figure 38: Blob- and contour detection applied to circular regions

Blob detection is efficient at identifying roughly circular regions and provides quick localization and size estimates, making it ideal for detecting spots or particles across multiple scales. As shown in Figure 38, blob detection correctly separates the two touching circular objects, whereas contour detection merges them into a single unit. However, the blob detection algorithm lacks detailed shape information and struggles with irregular or complex objects.

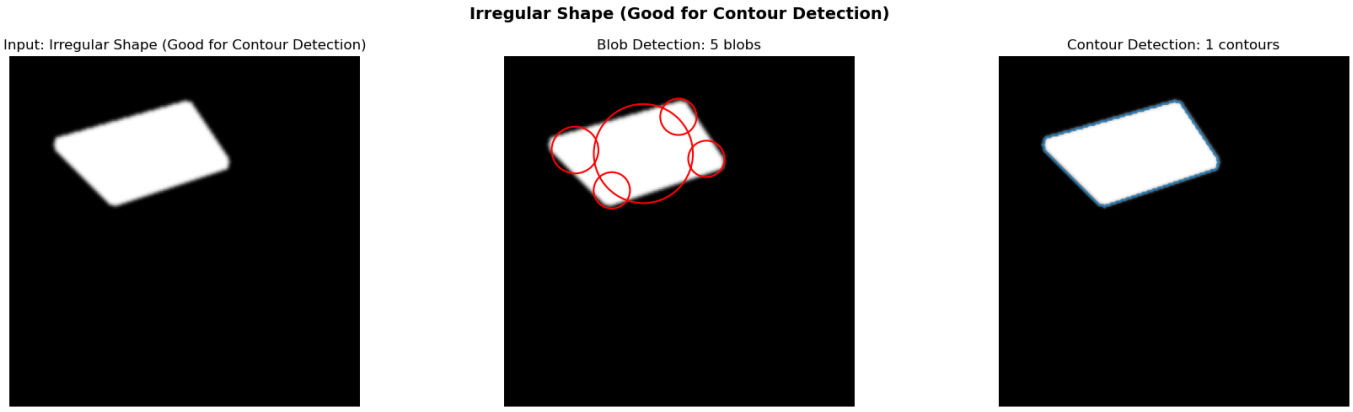


Figure 39: Blob- and contour detection applied to rectangular regions

Contour detection, on the other hand, excels at outlining precise object boundaries and capturing detailed shape features, which is valuable for morphological analysis. As shown in Figure 38, contour detection correctly identifies a the rectangular object, whereas blob detection splits the region into multiple blobs. The effectiveness of the contour detection technique depends heavily on image quality and edge definition, and it can be computationally more intensive and sensitive to noise.

So blob detection is best for fast, approximate feature localization, while contour detection is preferred when detailed shape and boundary information is required.

5. Analyze the impact of different parameters (e.g., threshold values, filter sizes) on the detection results.

The performance of both blob and contour detection methods is sensitive to parameters like threshold values and filter sizes.

In blob detection, adjusting the `threshold` controls the sensitivity. Lower thresholds detect more blobs but increase false positives, while higher thresholds reduce noise but may miss subtle features. Similarly, the choice of `max_sigma` and `num_sigma` affects the scale range and granularity of detected blobs. The variations of threshold values in blob detection are shown in figure 25.

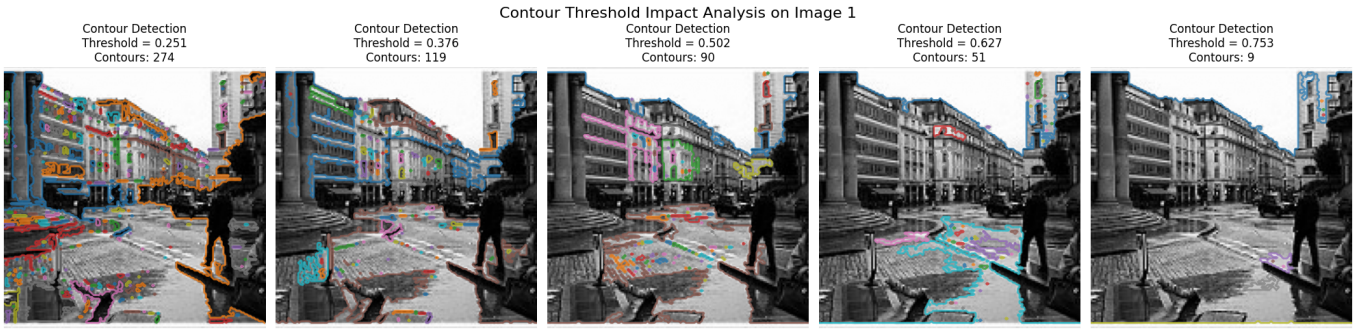


Figure 40: Contour detection with different thresholds

For contour detection, the threshold used in binarization critically impacts which features are segmented, too low of a threshold may merge objects or include noise, while too high may fragment or miss contours. Additionally, morphological operations like removing small objects depend on filter sizes that balance noise reduction against losing small meaningful contours. Contour detection with different thresholds are shown in figure 40.

6. Provide examples where one technique might be more suitable than the other.

Blob detection is more suitable in applications where the target objects are roughly circular and uniformly bright or dark against the background. For example, detecting cells in microscopy images, stars in astronomical images, or bubbles in fluid simulations. Its strength lies in quick localization and size estimation of round features across different scales.

In contrast, contour detection is ideal when precise object boundaries and shape details are essential, such as in medical image analysis, character recognition, or analyzing irregularly shaped objects like leaves or cracks. It enables detailed morphological analysis, making it preferable when shape complexity and boundary accuracy matter more than speed or rough position.