# Image Edge Detection and Orientation

## *Principles of Biological Vision*

- Marri Bharadwaj (B21CS045)

## Introduction

The objective of this assignment is to work on loading and processing images, filtering using DoG (Difference of Gaussians), applying edge detection filters with various orientations, implementing Winner-Takes-All (WTA) and normalisation operations, and conducting parameter tuning against different values of sigmas, kernel sizes and thresholds.

## Key Terms

- ***Difference of Gaussians filter***: The Difference of Gaussians (DoG) filter is a technique used in image processing to detect edges by highlighting regions of the image where intensity changes rapidly. It works by subtracting one Gaussian-blurred version of the image from another, which has a slightly different standard deviation (sigma). The result emphasises areas of high spatial frequency, effectively enhancing edges and suppressing noise. DoG is often used as an approximation to the more computationally intensive Laplacian of Gaussian (LoG) filter.

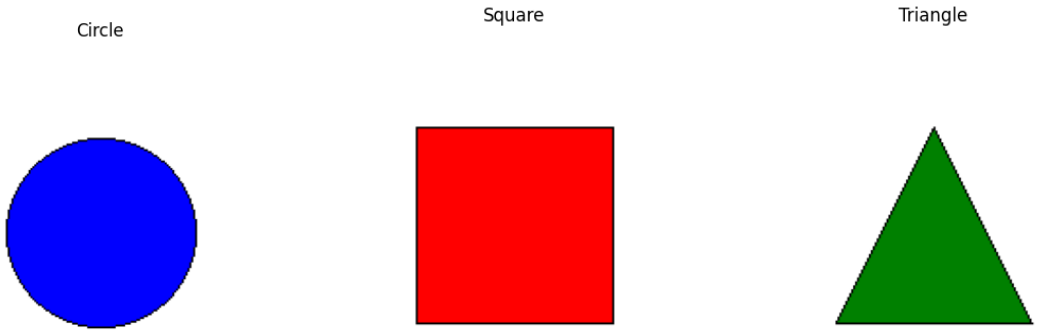
- ***Winner-Take-All Algorithm***: The Winner-Take-All (WTA) algorithm is a method used in edge detection and feature extraction where, for each pixel, the strongest response among multiple competing features (such as edge orientations) is selected. In edge detection, *WTA* is applied to determine the most prominent edge direction at each pixel by comparing the magnitudes of responses from various orientation filters. This process helps in focusing on the most significant features in the image, ignoring weaker, less relevant edges.

- ***Normalisation***: Normalisation in image processing is a technique used to adjust the intensity values of an image to a common scale, often to enhance contrast or make images more comparable. After applying operations like DoG and WTA, normalisation ensures that the resultant image values are standardised, often scaling the pixel values to a specific range. This step is crucial for improving the visibility of features and making further processing or analysis more effective, especially when combining different image processing techniques.

## *IMPLEMENTATION*

### Part- I: Generating Synthetic Images

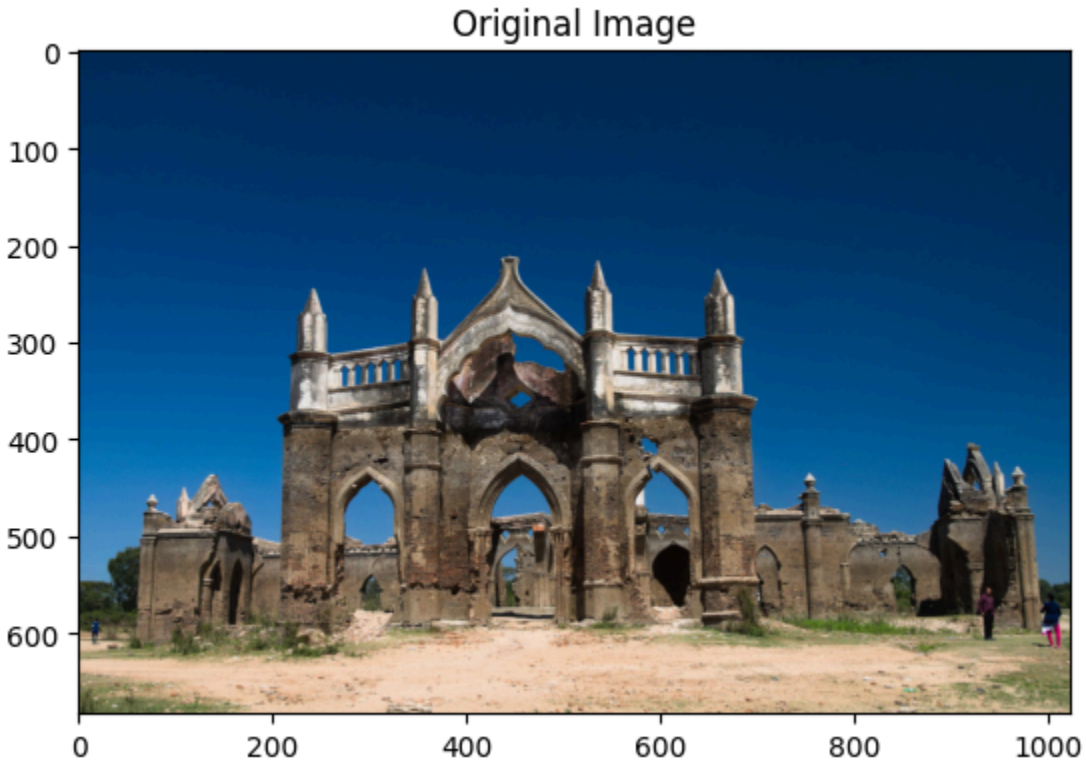
I created and displayed images of a circle, square, and triangle using Python's *PIL* and *Matplotlib* libraries. First, I created blank 200x200 pixel images and drew each shape on them, saving them as *circle.png*, *square.png*, and *triangle.png*. For the circle, I used *draw.ellipse()*, for the square, *draw.rectangle()*, and for the triangle, *draw.polygon()*. Next, I wrote a function to display these images using *Matplotlib*. The function loads each image, displays it with a title, and hides the axes for a clean view. Then, I called this function for each shape to confirm that the images were correctly generated and displayed as follows:



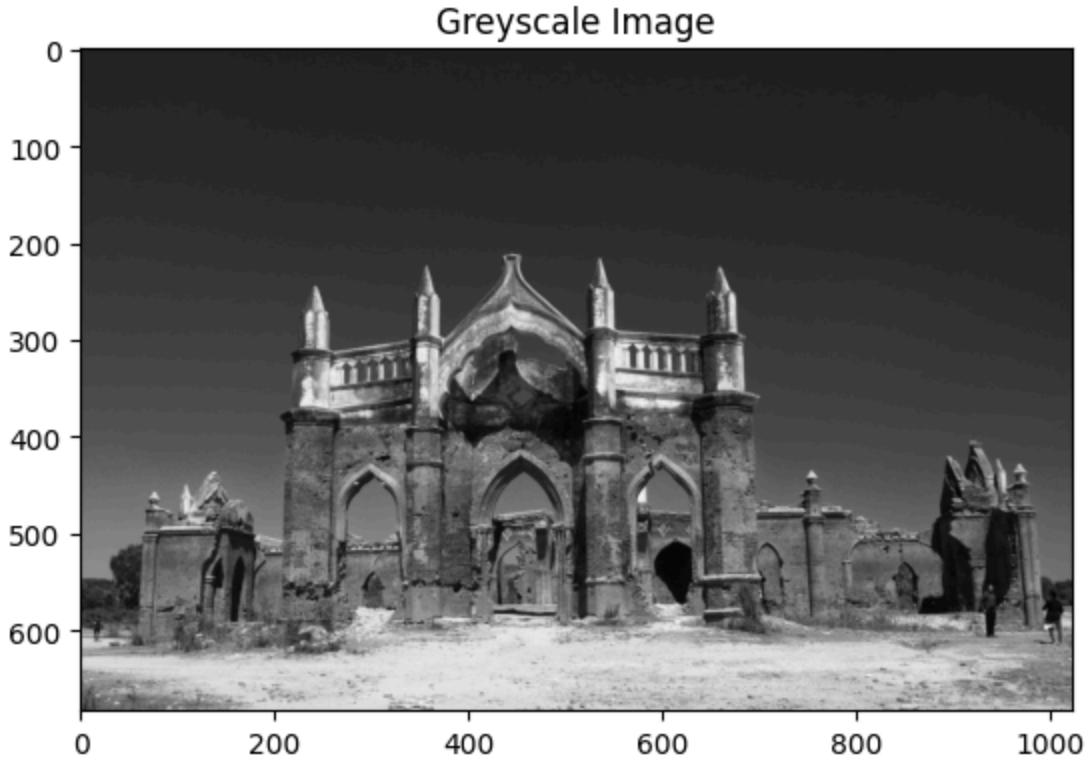

*(The outcomes of edge detection and orientation of the above shapes have been displayed post the explanation of the algorithm below.)*

## Part- II: Image Preprocessing

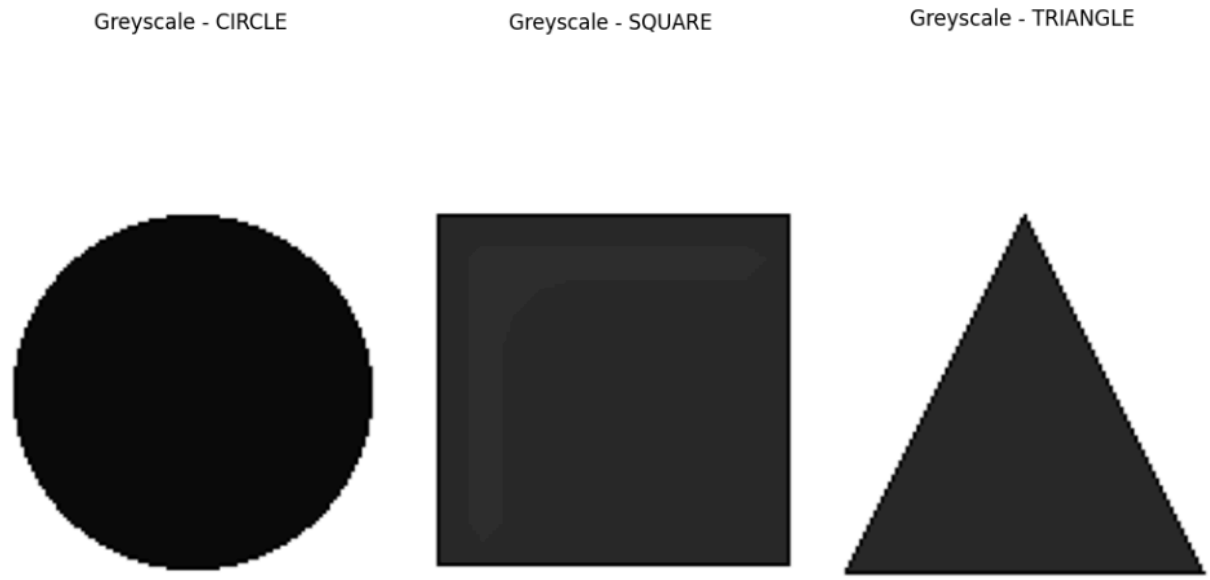
Next, I loaded, displayed, and converted my allotted image (5th image) to greyscale using *OpenCV*. First, I mounted Google Drive to access the image file. I specified the image path, stored in my Google Drive and accessible to all, and used *OpenCV* to load the image. Then, I converted the image from *BGR (Blue-Green-Red)* to *RGB* and displayed it using *Matplotlib* with the title *Original Image*. Converting it from *BGR (Blue-Green-Red)* to *RGB (Red-Green-Blue)* was significant because *OpenCV* uses *BGR* as the default colour format, while most other image processing libraries and display tools, like *Matplotlib*, expect the image data in *RGB* format. Without this conversion, the colours in the displayed image would be incorrect—blues would appear as reds, and vice versa. The conversion ensures that the colours are accurately represented when visualising or processing the image in environments that expect *RGB* format.




Also, I converted the image to grayscale using *OpenCV*'s *cv2.cvtColor()* function and displayed this greyscale version with the title *Greyscale Image*.



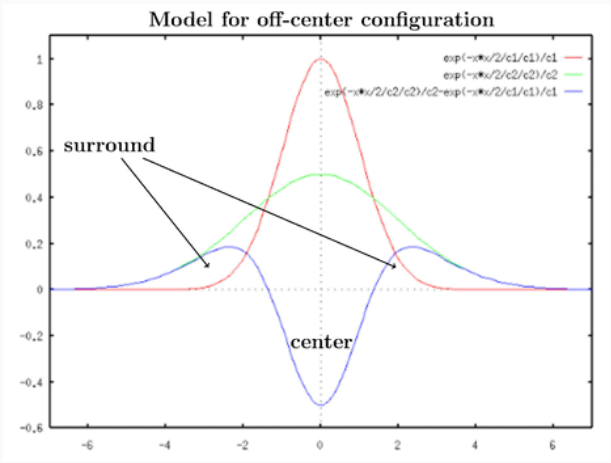

*Geometric Objects outcome-*



## Part- III: *DoG* Filtering

### Difference of Gaussian

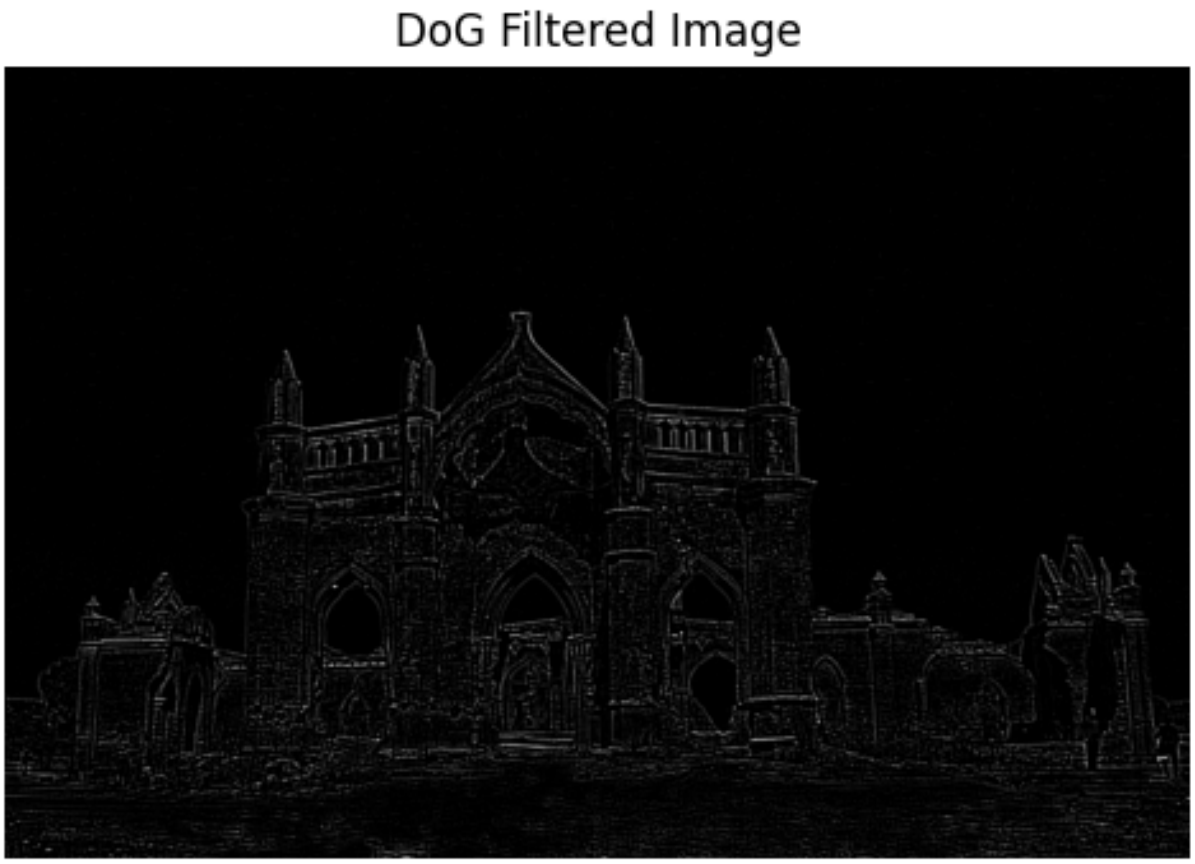
Approximates Laplacian of Gaussian (LoG)



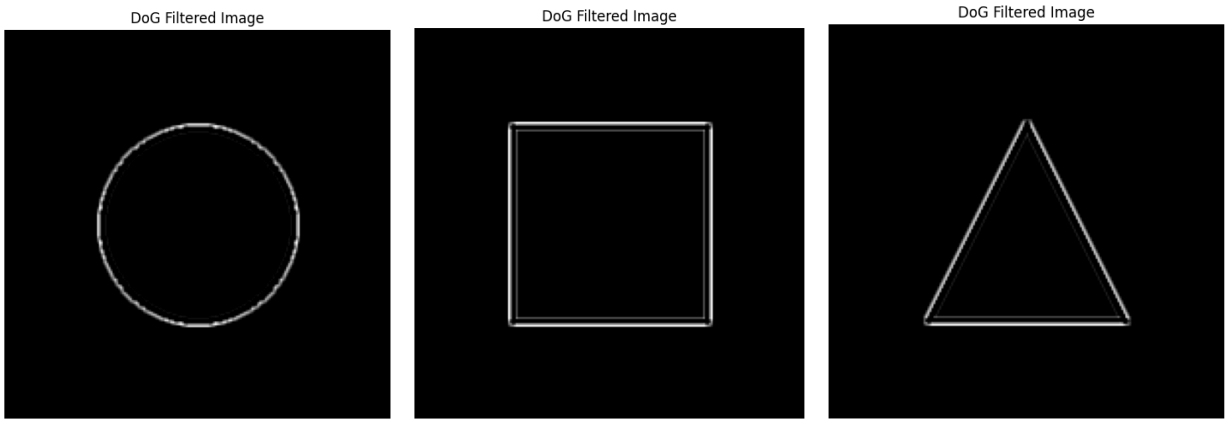

- Gaussian smoothing (noise reduction) precedes differentiation
- DoG is an approximation of LoG

I started by applying a *Difference of Gaussians (DoG)* filter to the greyscale image to remove noise and enhance contrast, taught in *Module 02-02*. To do this, I first blurred the image using two Gaussian filters with different standard deviations—one with a smaller sigma and the other with a slightly larger sigma. By subtracting the second, more blurred image from the first, I generated the DoG image. This technique effectively highlights the edges and textures in the image while suppressing noise, making important features more prominent. I then displayed the DoG-filtered image to assess the improvements in clarity and contrast.

```python
# Apply DoG (Difference of Gaussians) filter for noise removal and contrast improvement
# First Gaussian blur with a smaller sigma
gaussian_blur1 = cv2.GaussianBlur(selected_image, (5, 5), 1)
# Second Gaussian blur with a larger sigma
gaussian_blur2 = cv2.GaussianBlur(selected_image, (5, 5), 2)
# Subtract the two blurred images to enhance edges and remove noise
dog_image = cv2.subtract(gaussian_blur1, gaussian_blur2)
```




*Geometric Objects outcome-*



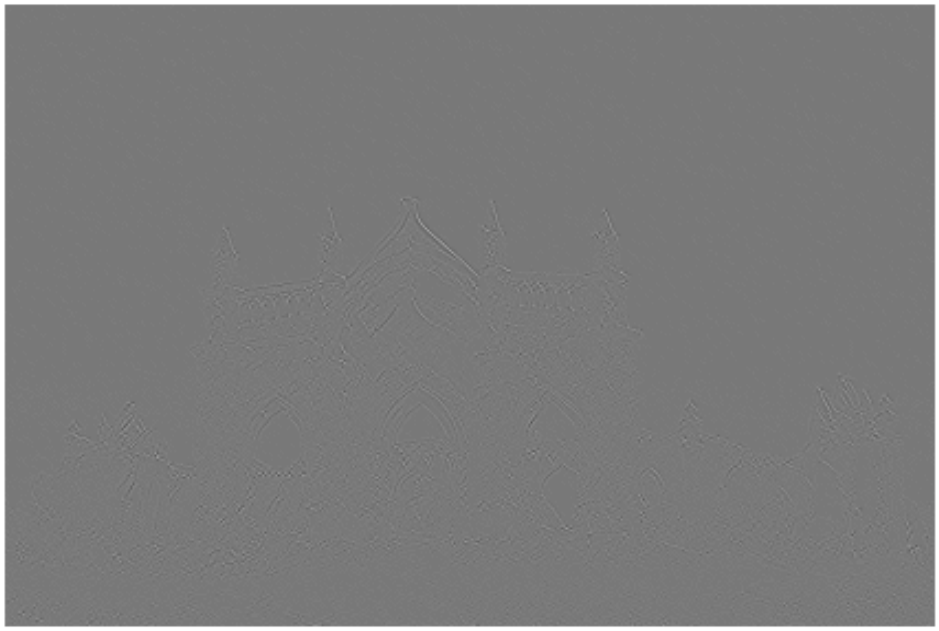

Following the *DoG* filtering, I focused on detecting edges in the image using *Sobel* filters, which are designed to identify edges at specific orientations. I created *Sobel* filters for four different orientations: 0°, 45°, 90°, and 135°. Each filter was applied to the respective *DoG* image to detect edges aligned with its respective orientation.

```
for orientation in orientations:
    if orientation == 0:
        # Sobel filter for horizontal edges
        kernel = cv2.Sobel(dog_image, cv2.CV_64F, 1, 0, ksize=3)
    elif orientation == 45:
        # Sobel filter for diagonal edges (45 degrees)
        kernel = cv2.Sobel(dog_image, cv2.CV_64F, 1, 1, ksize=3)
    elif orientation == 90:
        # Sobel filter for vertical edges
        kernel = cv2.Sobel(dog_image, cv2.CV_64F, 0, 1, ksize=3)
    elif orientation == 135:
        # Combine Sobel filters for 45-degree and 90-degree to detect edges at 135 degrees
        sobel_45 = cv2.Sobel(dog_image, cv2.CV_64F, 1, 1, ksize=3)
        sobel_90 = cv2.Sobel(dog_image, cv2.CV_64F, 0, 1, ksize=3)
        kernel = cv2.addWeighted(sobel_45, 1, sobel_90, 1, 0)
    sobel_kernels.append(kernel)
```

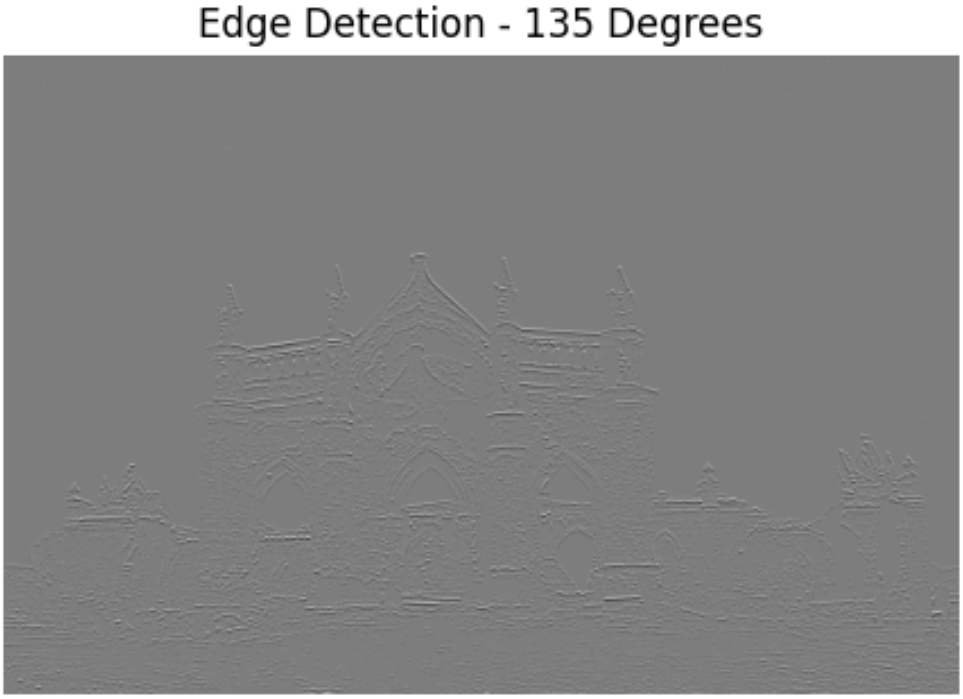
For example, the 0° filter detects vertical edges, while the 90° filter captures horizontal edges. Additionally, for the 135° orientation, I combined the effects of the 45° and 90° Sobel filters to simulate edge detection at that angle. After applying each filter, I displayed the resulting edge-detected images, allowing me to visualise how edges are emphasised differently depending on their orientation.

Edge Detection - 0 Degrees

Edge Detection - 45 Degrees

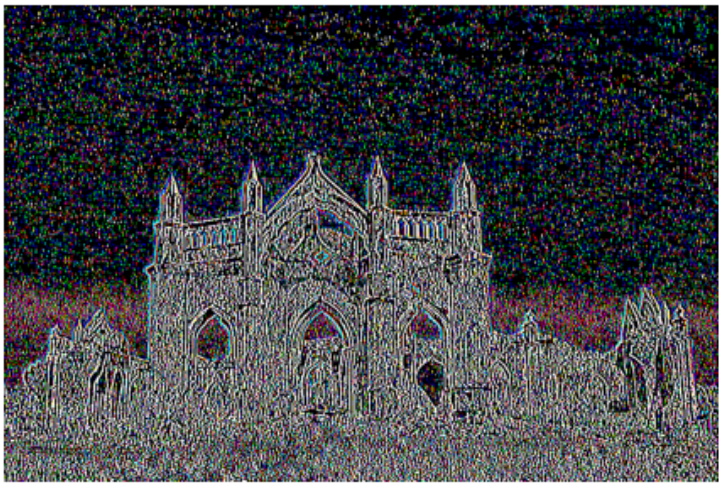


Edge Detection - 90 Degrees

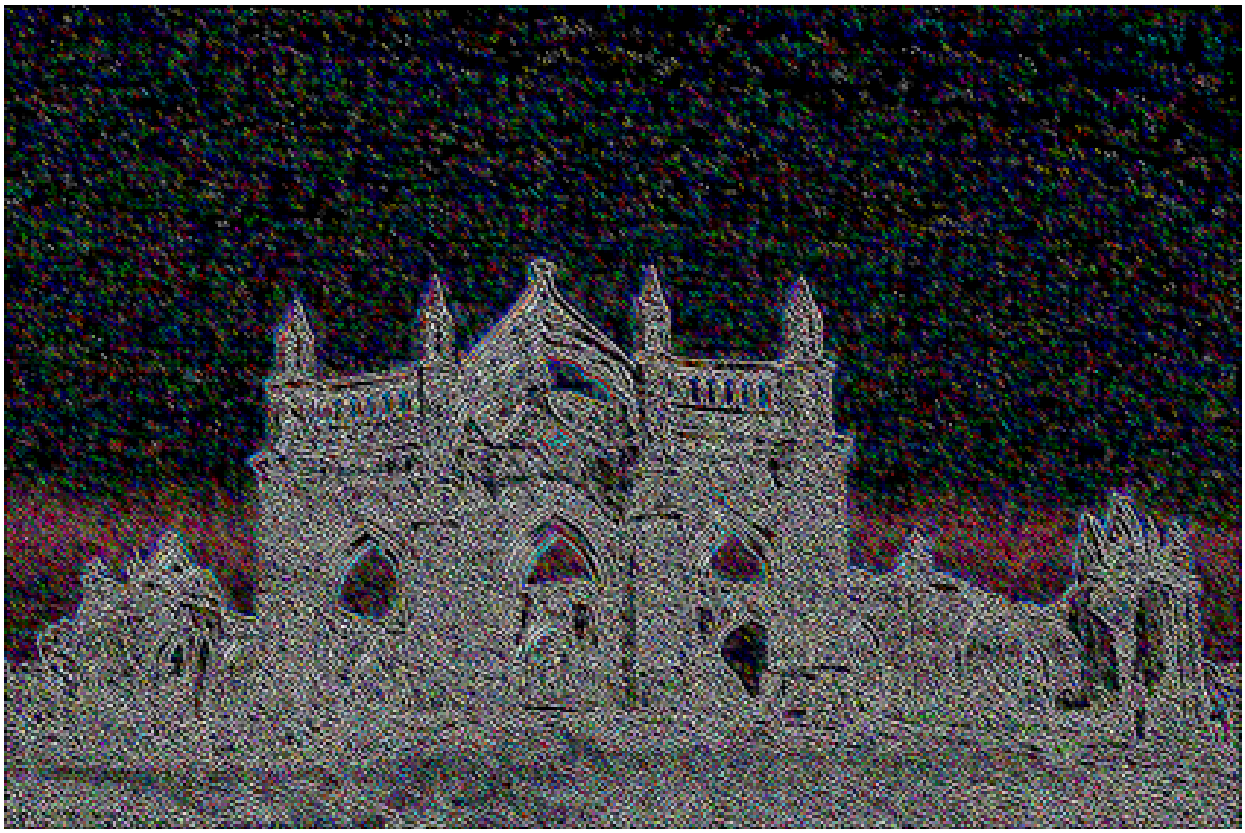



Fun testing for RGB image-

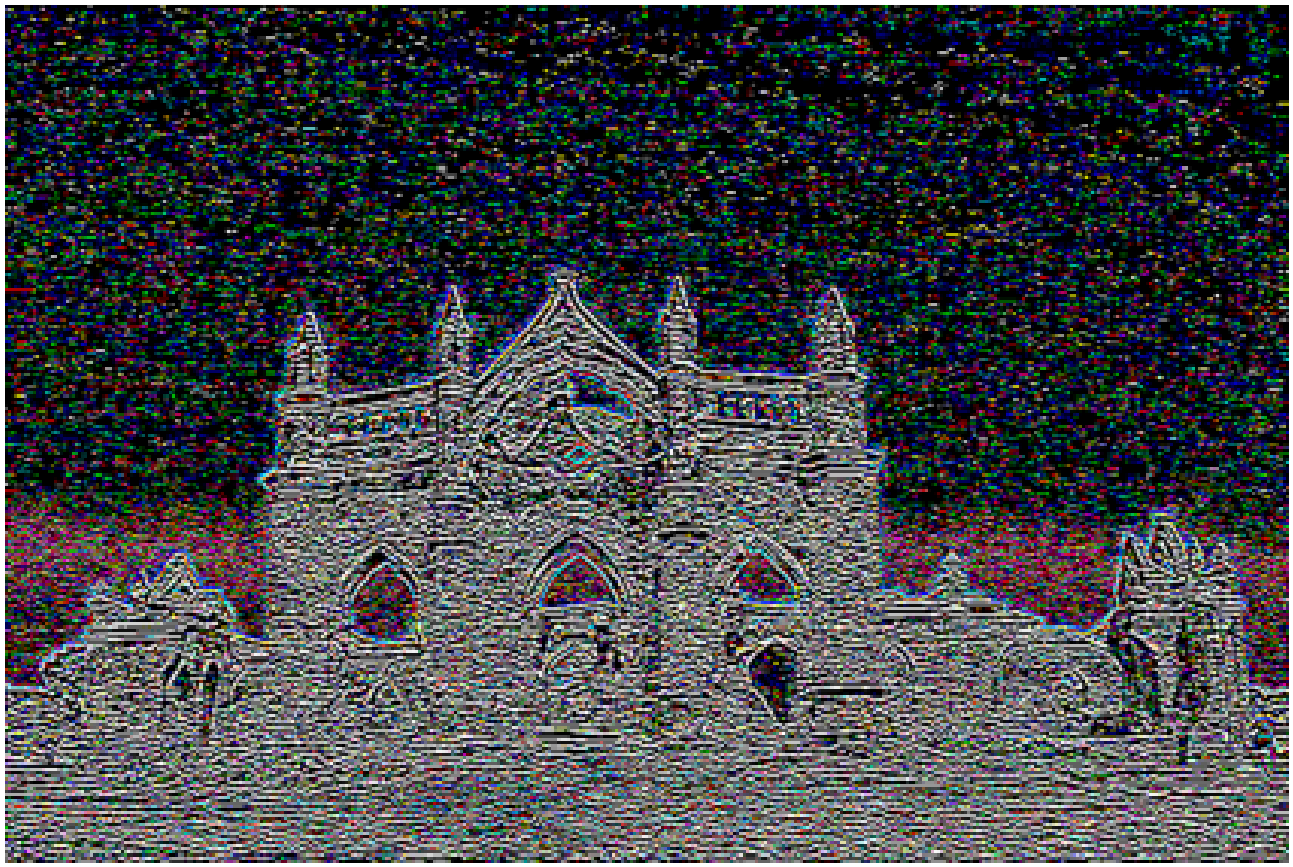
Edge Detection - 0 Degrees



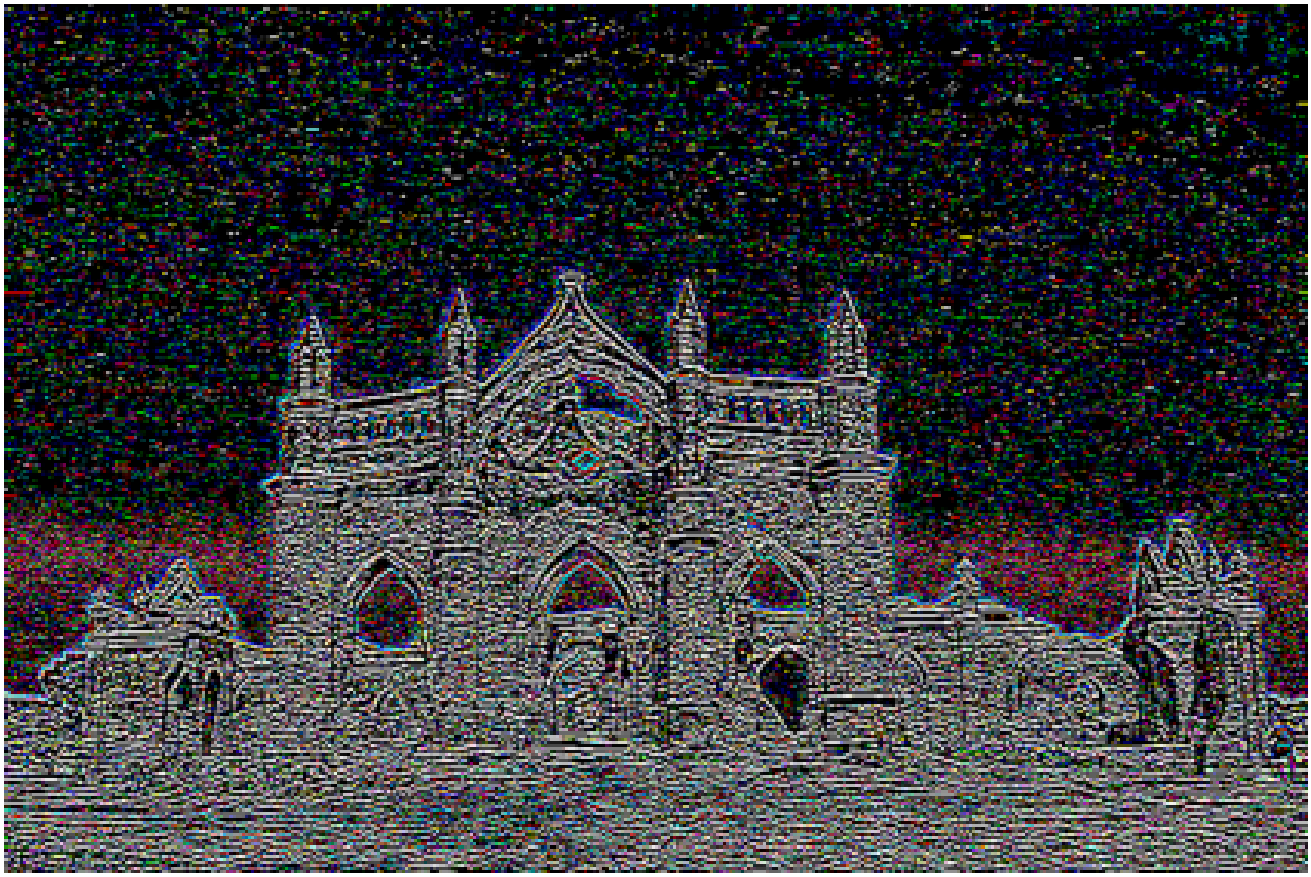
Edge Detection - 45 Degrees



Edge Detection - 90 Degrees

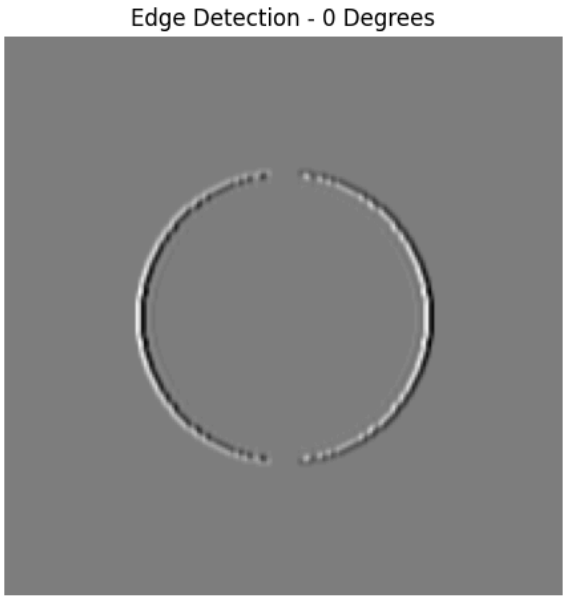


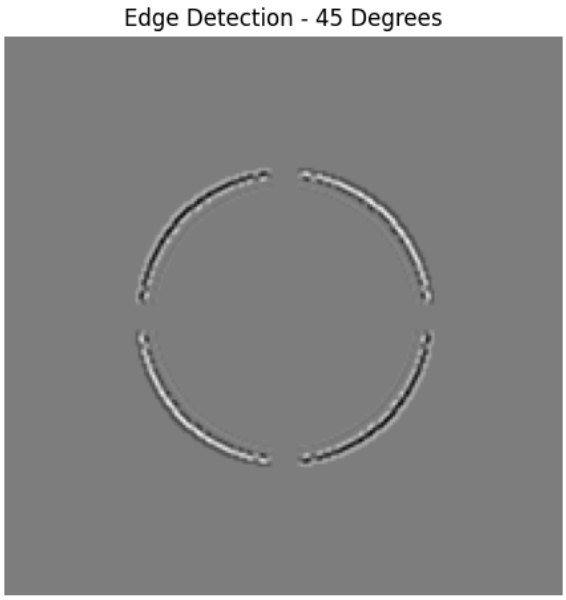
Edge Detection - 135 Degrees

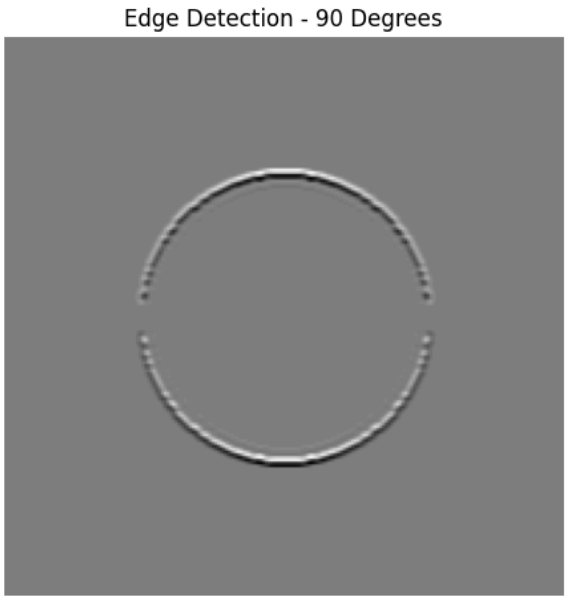


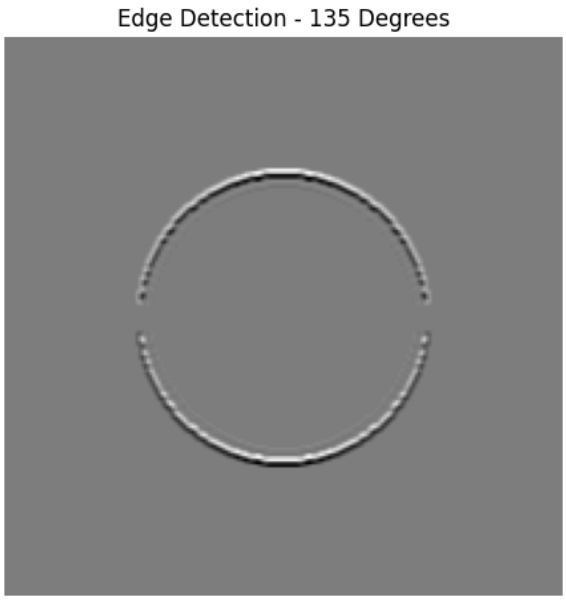
*Geometric Objects outcome-*

***Circle***



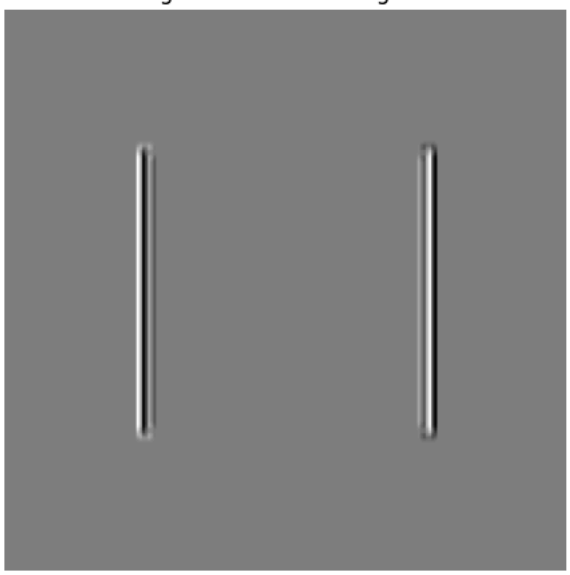



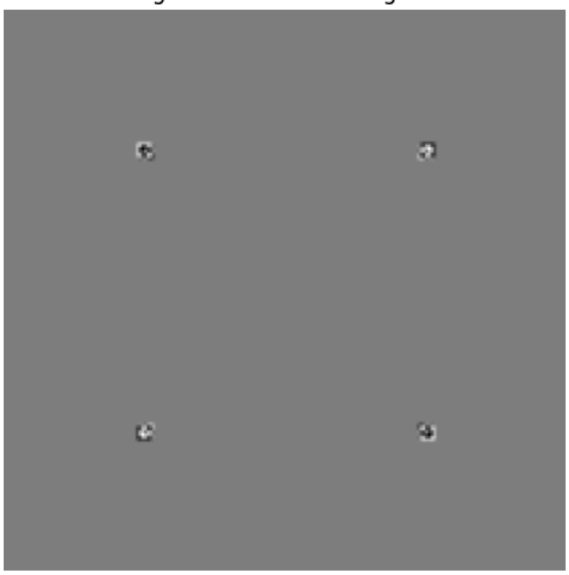



***Square***

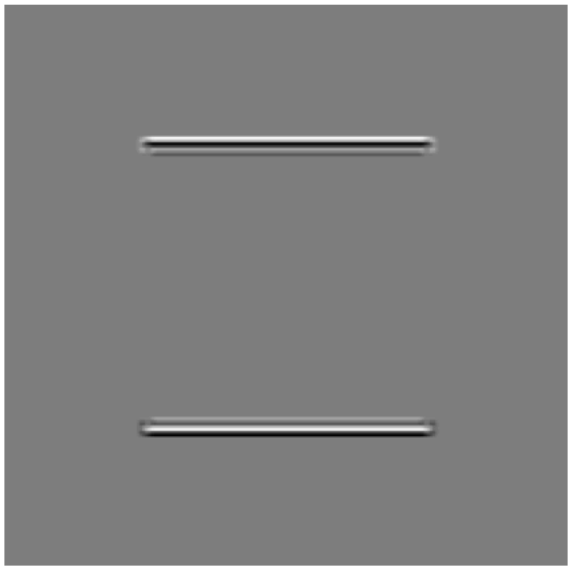
Edge Detection - 0 Degrees



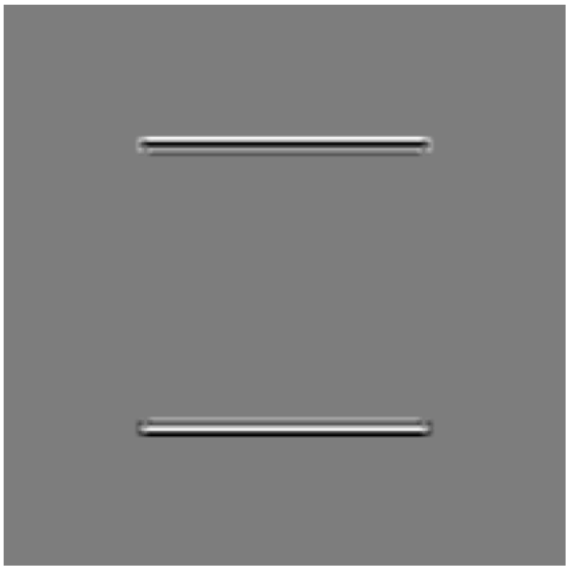
Edge Detection - 45 Degrees



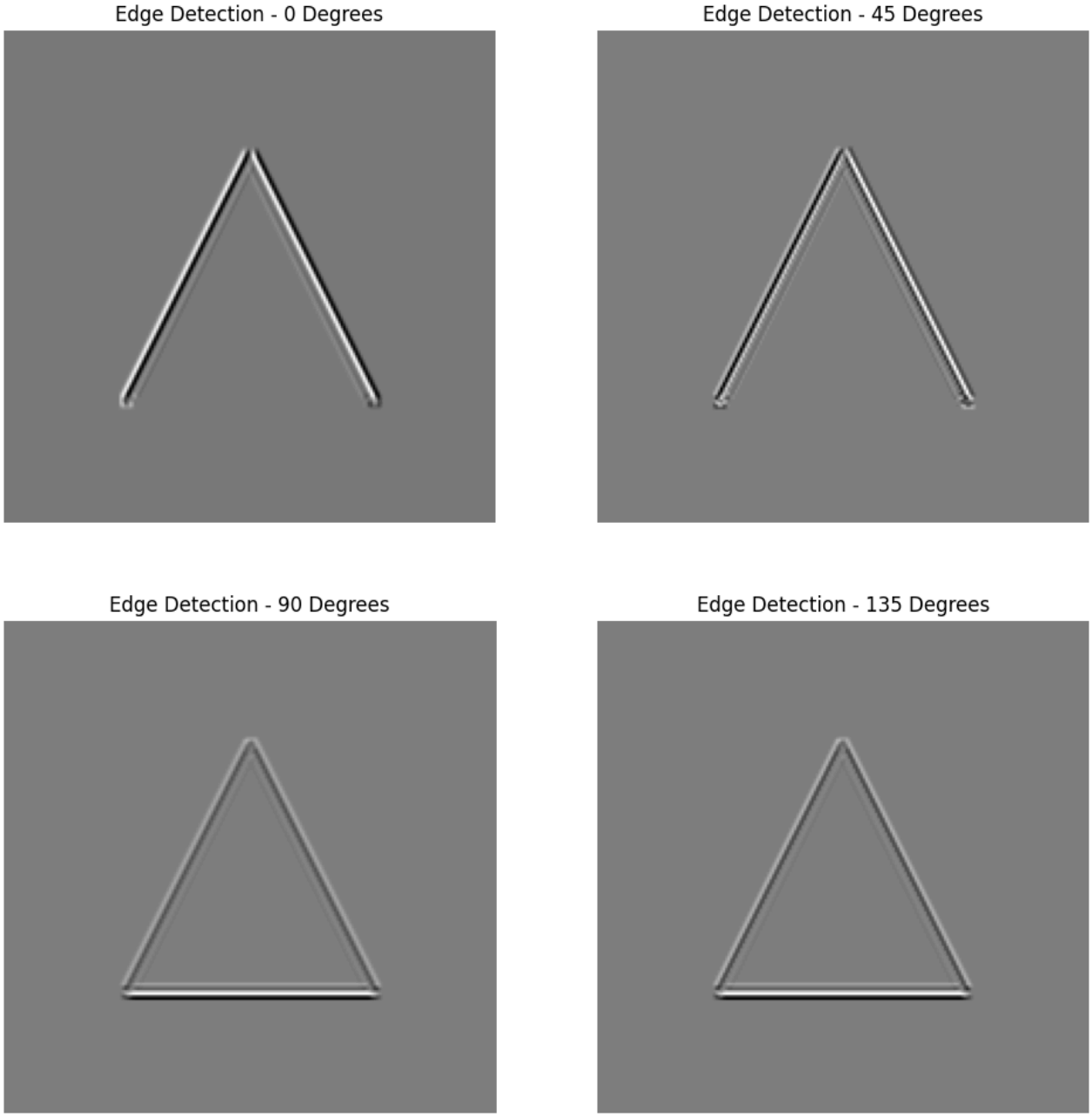
Edge Detection - 90 Degrees



Edge Detection - 135 Degrees



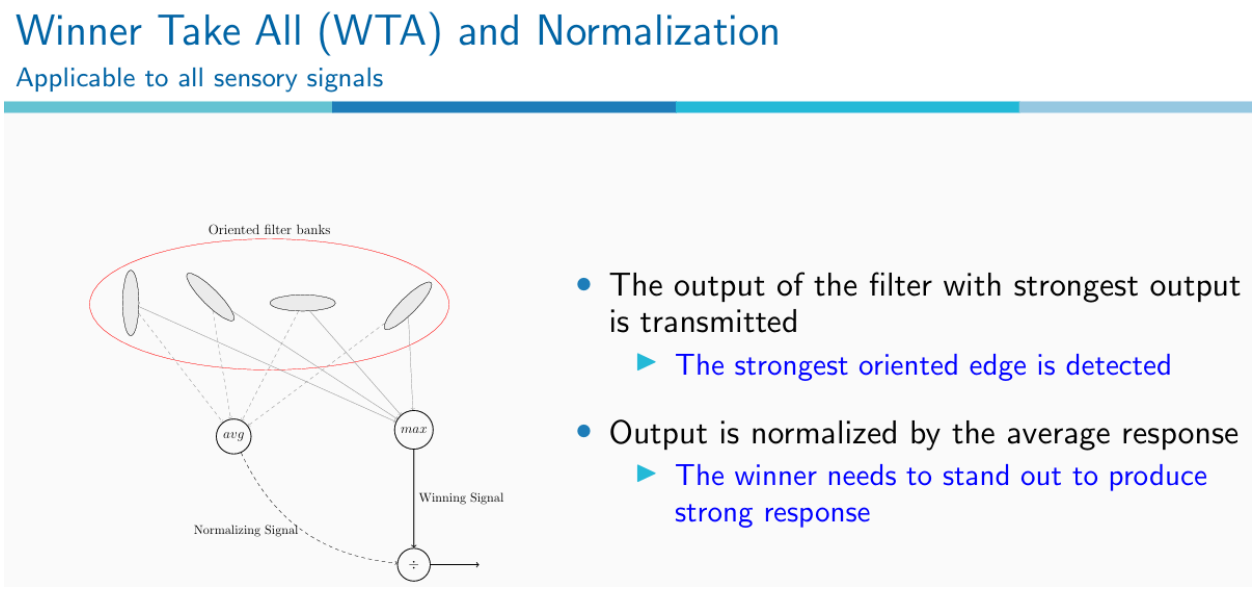
***Triangle***




## Part- IV: Winner-Take-All (WTA) and Normalisation

Then, I loaded the selected greyscale image into the variable *selected_image*. This image serves as the basis for further processing. Next, I applied the *Winner-Takes-All (WTA) algorithm* to combine the results from several *Sobel* filters, which were used to detect edges in various orientations. I initialised an empty image, *wta_image*, and updated it by retaining the maximum magnitude from each of the filtered images. This approach highlights the most significant edges detected across different orientations.

The *Winner-Takes-All (WTA) algorithm* is used to combine results from multiple edge-detection filters. It processes the images filtered by *Sobel* kernels at different orientations to create a single edge-detected image. The *WTA approach* works by taking the maximum magnitude of the edges detected by any of the filters at each pixel location. This means that the most prominent edge feature, as detected by any filter, is preserved. The significance of WTA is that it consolidates multiple edge-detection results into one comprehensive image, highlighting the strongest and most significant edges, which can improve the detection of important features in the image, as taught in *Module 02-02*.




```
# Winner-Takes-All (WTA) algorithm: Combine Sobel filter results
wta_image = np.zeros_like(selected_image, dtype=np.float32)  # Initialize an empty image to store WTA results
for filtered_image in sobel_kernels:
    magnitude = np.abs(filtered_image)  # Compute the magnitude of the filtered image
    wta_image = np.maximum(wta_image, magnitude)  # Apply WTA by keeping the maximum magnitude
```

To enhance the contrast of the image, I **further used** *Advanced Contrast Normalisation (ACN)*. The *acn_normalising* function was defined to apply this technique by adjusting the contrast based on the relationship between the *WTA* image and the original greyscale image. The function uses parameters *alpha* and *beta* to control the normalisation process, improving the visibility of features.

*Advanced Contrast Normalisation (ACN)* is used to enhance the visibility and contrast of features in an image. *ACN* normalises the *WTA* image by adjusting its contrast relative to the original grayscale image. The normalisation process involves scaling the *WTA* image with a parameter (*alpha*) and adding an offset (*beta*). The purpose of *ACN* is to enhance the contrast of the important features highlighted by the *WTA* algorithm, making them more prominent and easier to distinguish. By normalising the image, ACN helps in improving the visual clarity and quality of the final processed image, which is particularly useful in image analysis and feature extraction tasks.

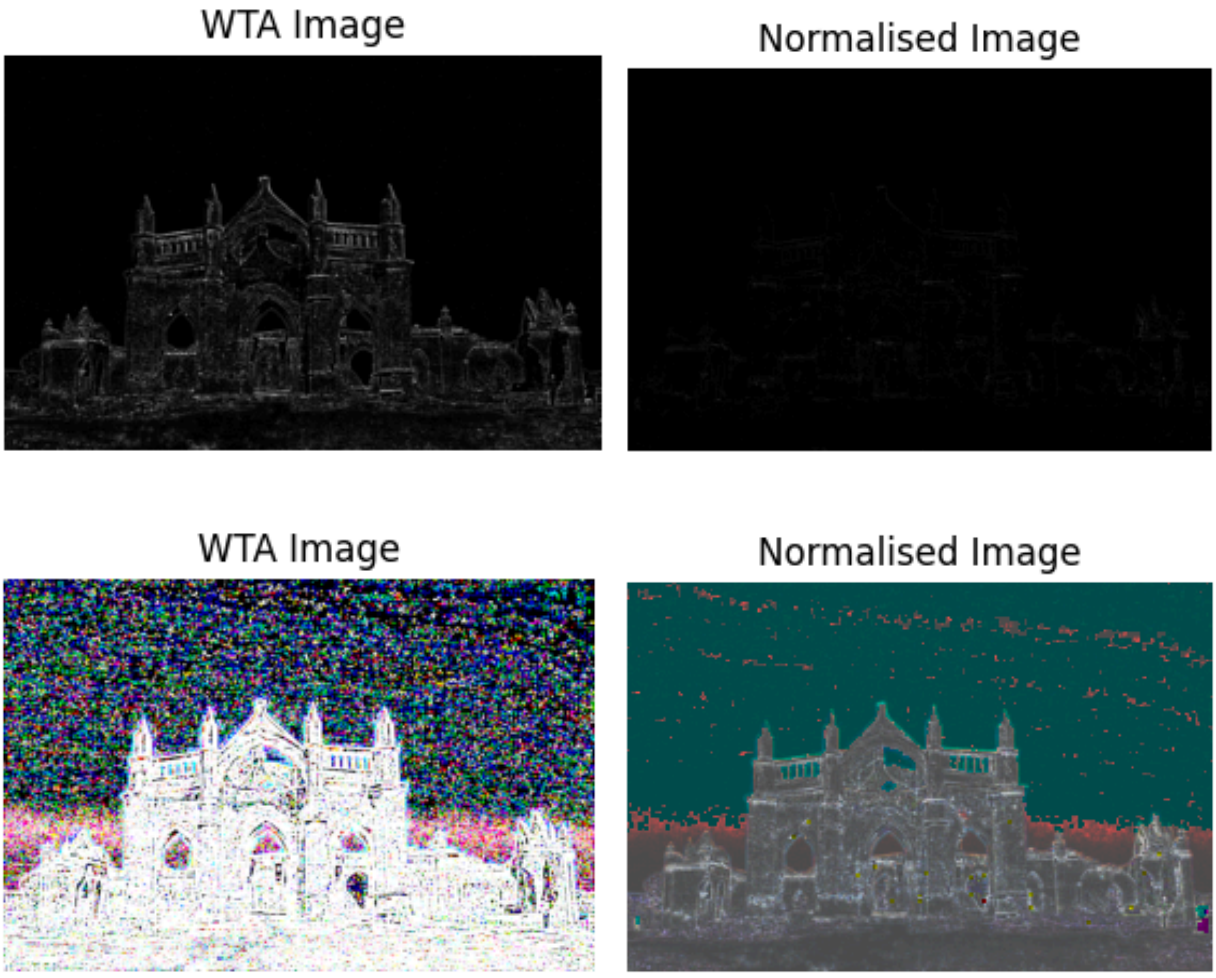
```
# Apply Advanced Contrast normalisation (ACN) for normalisation
def acn_normalising(image, wta_image, alpha=0.7, beta=0.3):
    """
    Apply Advanced Contrast normalisation (ACN) to normalise the image.

    Parameters:
    - image: The original image (grayscale).
    - wta_image: The WTA processed image.
    - alpha: Weight for normalisation.
    - beta: Offset for normalisation.

    Returns:
    - normalised_image: The normalised image after applying ACN.
    """
    normalised_image = alpha * (wta_image / image) + beta
    return normalised_image

# normalise the selected image using the ACN function
normalised_image = acn_normalising(selected_image, wta_image, alpha=0.7, beta=0.3)
```

Then, I visualised the results using *Matplotlib*. I created a figure with three subplots: the original greyscale image, the WTA image showing the edge detection results, and the normalised image after applying ACN. Each image was displayed with titles and without axes to provide a clear and comparative view of the different stages of image processing.

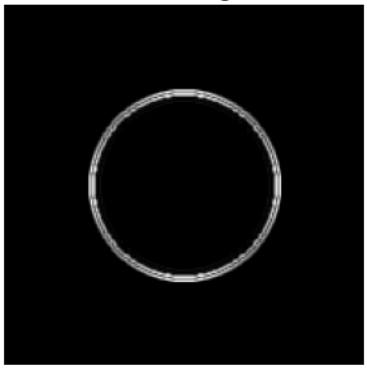



*Geometric Objects outcome-*

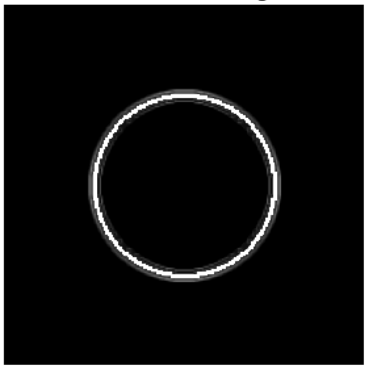
***Circle***

Greyscale Image

WTA Image



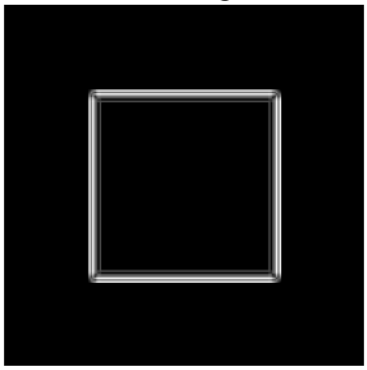
Normalised Image



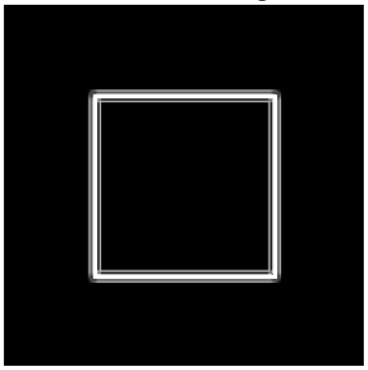
***Square***

Greyscale Image

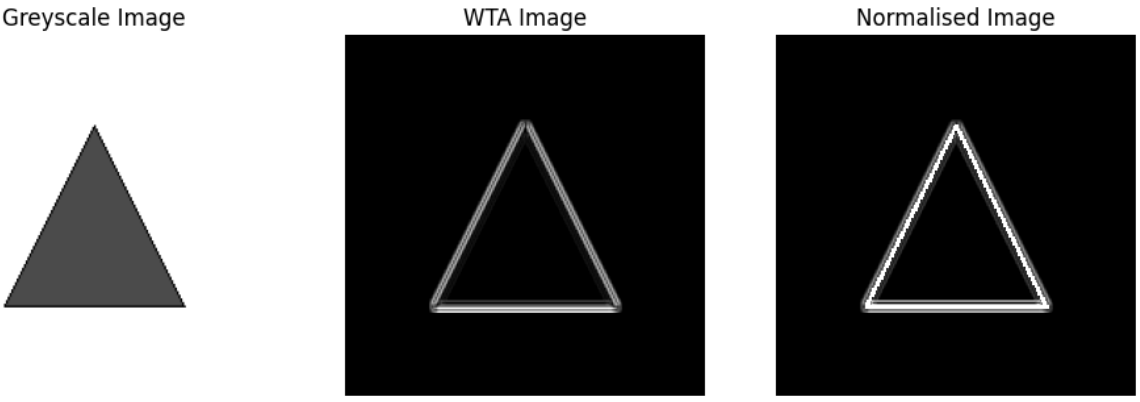
WTA Image



Normalised Image



***Triangle***

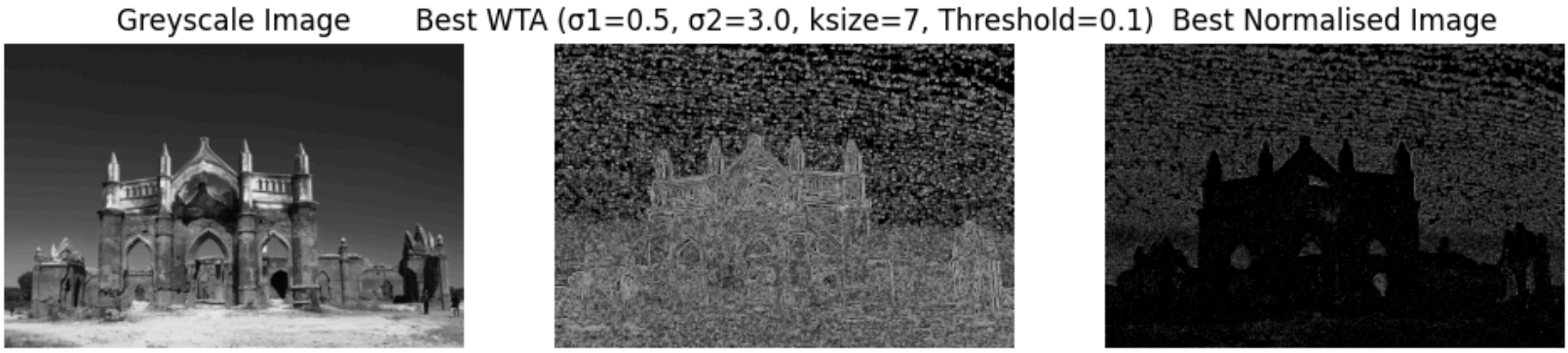



## Part- V: Experimentation (aka Parameter Tuning)

I began by defining several parameters for Gaussian blurring (using different *sigma* values and kernel sizes) and thresholding. These parameters are used in different functions to process the greyscaled image. The first function, *apply_dog_filter*, applies a Difference of Gaussians (DoG) filter to the image. This filter enhances edges by subtracting one blurred version of the image from another, using different Gaussian blur strengths (defined by sigma values and kernel size). The next function, *apply_edge_detection*, used *Sobel* filters to detect edges in the image at different orientations (0°, 45°, 90°, and 135°). The edges were then processed using the *Winner-Takes-All (WTA)* approach, which retains only the strongest responses, followed by *Advanced Contrast Normalisation (ACN)* to enhance the image contrast.

Then, I iterated through combinations of the defined parameters (sigma values, kernel sizes, and thresholds) to find the best combination that produces the highest quality edge-detected image. I evaluated each combination by applying the *DoG* filter, followed by edge detection and then *WTA* and normalisation. The results for each combination were displayed using *matplotlib*, showing the original grayscale image, the DoG filtered image, the WTA image, and the normalised image for each combination of parameter- essentially **performing parameter tuning**.

Finally, I identified the best result based on the sum of the normalised image's pixel values. This best result, along with its corresponding parameters, was displayed, allowing to display the most effective filtering and edge detection configuration for the image.

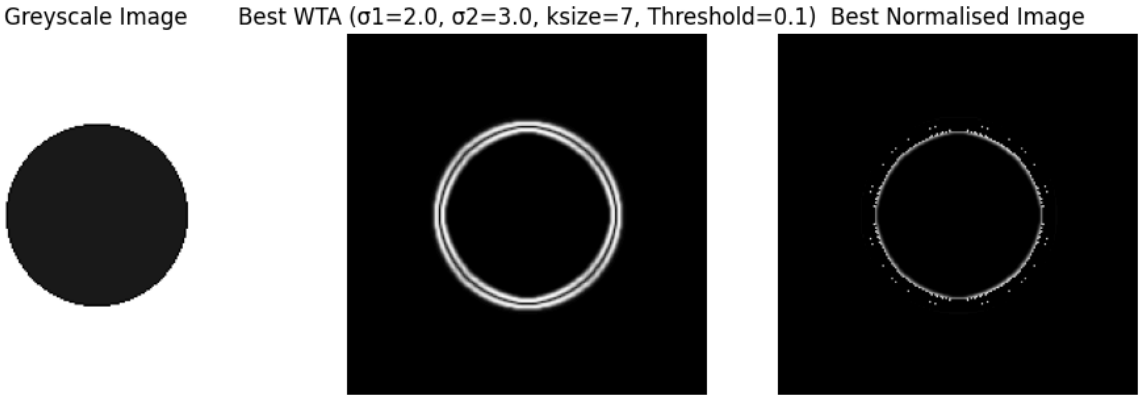



Thus, I got the best result of edge detection for the parameter of-

- $\sigma_1 = 0.5$
- $\sigma_2 = 3.0$
- *kernel size* = 7
- threshold = 0.1

*Geometric Objects outcome-*

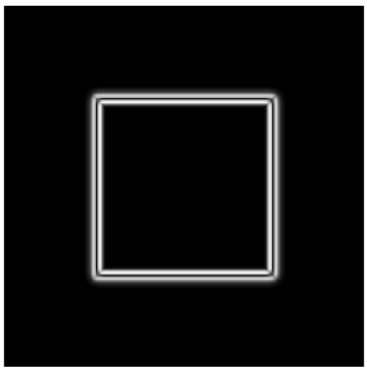
***Circle***



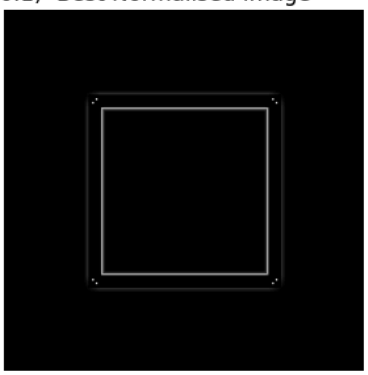
## *Square*

Greyscale Image

Best WTA (σ1=1.0, σ2=3.0, ksize=7, Threshold=0.1)



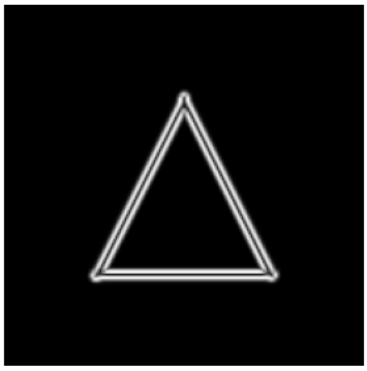
Best Normalised Image



## *Triangle*

Greyscale Image

Best WTA (σ1=2.0, σ2=3.0, ksize=7, Threshold=0.1)



Best Normalised Image