มหาวิทยาลัยแม่ฟ้าหลวง
MAE FAH LUANG UNIVERSITY

1504302 Image Processing and Application

Aj. Sirikan Chucherd

5 November 2025

Project Idea: “Object Detection Model in Python”

Project Report

by

6631502023	ARKAR PYAE PHYO
6631502028	SWAN HTET
6631502055	AUNG MYINT MYAT

ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to all those who supported us throughout the completion of this project on *Image Processing and Application*.

First, we would like to thank our professor, Sirikan Chucherd, for their invaluable guidance, constructive feedback, and continuous encouragement. Their expertise in the field enhanced our understanding of key concepts and helped shape the direction of this project.

We are also grateful to our university, Mae Fah Luang University, for providing the necessary facilities and resources to conduct research and practical experiments. Special thanks to the School of Applied Digital Technology for offering a comprehensive curriculum that deepened our interest in image processing.

We would also like to thank our classmates and friends for their helpful discussions and collaboration, which contributed to a better learning experience.

Thank you all.

Tables of Contents

Abstract

This report presents the development of Image Processing and Application, a comprehensive web-based image processing platform that integrates classical computer vision techniques with modern deep learning. The application implements many image processing functions across three major modules: histogram analysis and filtering, edge detection and segmentation, and geometric transformations.

The system is built using the Python Flask framework with OpenCV for image processing operations and YOLO v11 for AI-powered object detection. The web interface provides an intuitive user experience with real-time processing, visual feedback, and comparison tools. The application is deployed on a production server with PM2 process management and Nginx reverse proxy, serving users at https://stellarion.app and http://139.59.113.53:5000.

Project Overview

Background

Digital image processing is a fundamental field in computer science with applications spanning medical imaging, autonomous vehicles, surveillance systems, and multimedia. Understanding both classical algorithms and modern deep learning approaches is essential for computer vision practitioners.

Motivation

Traditional image processing education often lacks hands-on, interactive platforms for experimenting with algorithms. This project addresses this gap by providing:

1. Interactive Learning Environment - Real-time visualization of algorithm results
2. Comprehensive Coverage - From basic histogram operations to advanced transformations
3. Modern Integration - Classical CV techniques combined with deep learning
4. Accessibility - Web-based platform accessible from any device

Scope

The project encompasses:

- 35+ image processing functions organized into three student modules
- Classical algorithms: Histogram equalization, spatial/frequency filtering, edge detection
- Advanced techniques: Affine/perspective transforms, image registration, morphological operations
- Deep learning: YOLO v11 for real-time object detection
- Production deployment: Cloud-hosted web application with SSL encryption

Project Objectives

This project involved the development and production deployment of a comprehensive Image Processing Platform with integrated Deep Learning capabilities.

The platform provides a suite of image manipulation tools, including:

- Histogram Analysis: Equalization and gray-level transformations.
- Domain Filtering: Spatial (mean, median) and frequency domain (FFT) filtering.
- Image Analysis: Edge detection (Sobel, Canny) and various segmentation algorithms.
- Geometric Transforms: Operations with interpolation.

A key feature is the integration of YOLO v11 for real-time object detection, supporting 80+ COCO classes with adjustable confidence thresholds and bounding box visualization.

The application is delivered through a responsive Tailwind CSS web interface featuring real-time previews and zoom. The entire system is deployed on DigitalOcean, managed by PM2, served via an Nginx reverse proxy, and secured with SSL/TLS.

Methodology

Development Approach

The project followed an Agile development methodology, structured in three key phases:

1. Core Development: Focused on initial planning, technology stack setup, and the implementation of all fundamental image processing modules (histogram analysis, filtering, edge detection, segmentation, and geometric transforms).
2. Integration & Enhancement: Involved integrating the YOLO v11 model for object detection. The UI was simultaneously enhanced with features like zoom, comparison modals, and descriptive text, alongside performance optimizations.
3. Production Deployment: Consisted of deploying the application on a DigitalOcean server. This included configuring PM2 for process management, setting up an Nginx reverse proxy, securing the site with SSL/TLS, and conducting final testing.

Component Descriptions

1. Client Layer

- Technology: HTML5, Tailwind CSS, JavaScript ES6+
- Functionality: User interface, file upload, filter selection, image display
- Key Features: Responsive design, real-time feedback, Chrome-style zoom

2. Nginx Reverse Proxy

- Purpose: Load balancing, SSL termination, static file serving
- Configuration: Port 443 (HTTPS) → 5000 (Flask)
- Benefits: Security, performance, scalability

3. Application Layer (Flask + PM2)

- Flask: Web framework handling HTTP requests/responses
- PM2: Process manager ensuring high availability
- Routes: /upload_image, /apply_filter, /detect_objects

4. Processing Layer

- Student 1 - AUNG MYINT MYAT: 10 histogram & filtering functions
- Student 2 - SWAN HTET: 12 edge detection & segmentation functions
- Student 3 - ARKAR PYAE PHYO: 15 geometric transformation functions

5. Storage Layer

- Input Directory: Temporary uploaded images
- Output Directory: Processed results
- Model Storage: YOLO v11 weights (6.3MB)

Implementation Details

Module 1: Image Enhancement (AUNG MYINT MYAT)

Student 1: Histogram & Filtering

A. Histogram Equalization

Algorithm:

```python
def mod1_hist_equalize(input_img):
    """
    Implements histogram equalization to enhance image contrast.

    Theory:
    - Redistributes pixel intensities across the full range [0, 255]
    - Uses cumulative distribution function (CDF)
    - Formula: new_value = (CDF[old_value] - CDF_min) * 255 / (total_pixels - CDF_min)
    """
    if len(input_img.shape) == 3:
        # Convert to YCrCb color space
        ycrcb = cv2.cvtColor(input_img, cv2.COLOR_BGR2YCrCb)
        # Apply equalization to Y channel only
        ycrcb[:, :, 0] = cv2.equalizeHist(ycrcb[:, :, 0])
        result = cv2.cvtColor(ycrcb, cv2.COLOR_YCrCb2BGR)
    else:
        result = cv2.equalizeHist(input_img)
    return result
```

Key Features:

Preserves color by operating on luminance channel (Y)

Automatic parameter selection

Effective for low-contrast images

B. Spatial Domain Filtering

Mean Filter (Averaging):

```python
def mod1_filter_mean(input_img, kernel_size=5):
    """
    Applies mean filtering for noise reduction.

    Theory:
    - Replaces each pixel with average of neighbors
    - Kernel: ones(k×k) / k²
    - Trade-off: Noise reduction vs. edge blurring
    """
    return cv2.blur(input_img, (kernel_size, kernel_size))
```

Median Filter:

```python
def mod1_filter_median(input_img, kernel_size=5):
    """
    Applies median filtering for salt-and-pepper noise removal.

    Theory:
    - Non-linear filter replacing pixel with median of neighbors
    - Preserves edges better than mean filter
    - Effective for impulse noise
    """
    return cv2.medianBlur(input_img, kernel_size)
```

Laplacian Sharpening:

```python
def mod1_filter_laplacian(input_img):
    """
    Sharpens image using Laplacian operator.

    Theory:
    - Second-order derivative operator: ∇²f = ∂²f/∂x² + ∂²f/∂y²
    - Kernel: [[0,-1,0], [-1,4,-1], [0,-1,0]]
    - Formula: sharpened = original + α × laplacian
    """
    gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
    laplacian = cv2.Laplacian(gray, cv2.CV_64F)
    # Sharpen by adding laplacian to original
    sharpened = cv2.convertScaleAbs(gray - laplacian)
    return cv2.cvtColor(sharpened, cv2.COLOR_GRAY2BGR)
```

C. Frequency Domain Filtering

Low-Pass Filter:

```python
def mod1_filter_freq_lowpass(input_img, cutoff=30):
    """
    Frequency domain low-pass filtering using FFT.

    Theory:
    - Fourier Transform: f(x,y) → F(u,v)
    - Ideal LPF: H(u,v) = 1 if D(u,v) ≤ D₀, else 0
    - Removes high-frequency noise
    """
    gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)

    # Apply FFT
    f_transform = np.fft.fft2(gray)
    f_shift = np.fft.fftshift(f_transform)

    # Create circular mask
    rows, cols = gray.shape
    crow, ccol = rows // 2, cols // 2
    mask = np.zeros((rows, cols), dtype=np.uint8)
    cv2.circle(mask, (ccol, crow), cutoff, 1, thickness=-1)

    # Apply mask and inverse FFT
    f_shift_filtered = f_shift * mask
    f_inv_shift = np.fft.ifftshift(f_shift_filtered)
    img_filtered = np.fft.ifft2(f_inv_shift)
    img_filtered = np.abs(img_filtered)

    return cv2.cvtColor(img_filtered.astype(np.uint8), cv2.COLOR_GRAY2BGR)
```

Module 2: Image Segmentation and Edge Detection (SWAN HTET)

Student 2: Edge Detection & Segmentation

A. Edge Detection Algorithms

Sobel Operator:

```python
def mod2_edge_sobel(input_img):
    """
    Sobel edge detection using gradient operators.

    Theory:
    - First-order derivative approximation
    - Gx = [[-1,0,1], [-2,0,2], [-1,0,1]]  (horizontal)
    - Gy = [[-1,-2,-1], [0,0,0], [1,2,1]]  (vertical)
    - Magnitude: |G| = √(Gx² + Gy²)
    """
    gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    sobel = np.sqrt(sobelx**2 + sobely**2)
    sobel = cv2.convertScaleAbs(sobel)
    return cv2.cvtColor(sobel, cv2.COLOR_GRAY2BGR)
```

Canny Edge Detection:

```python
def mod2_edge_canny(input_img, threshold1=100, threshold2=200):
    """
    Multi-stage Canny edge detection algorithm.

    Theory:
    1. Gaussian smoothing (noise reduction)
    2. Gradient calculation (Sobel)
    3. Non-maximum suppression (thin edges)
    4. Hysteresis thresholding (strong/weak edges)

    Parameters:
```

```
    - threshold1: Lower threshold for hysteresis
    - threshold2: Upper threshold for hysteresis
    """
    gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(gray, threshold1, threshold2)
    return cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
```

Edge Comparison Feature:

```python
def mod2_edge_compare(input_img):
    """
    Returns side-by-side comparison of three edge detection methods.

    Returns: {
        'compare_mode': True,
        'methods': {
            'sobel': sobel_result,
            'prewitt': prewitt_result,
            'canny': canny_result
        }
    }
    """
    # Implementation allows visual comparison of algorithms
```

B. Segmentation Techniques

Otsu's Thresholding:

```python
def mod2_threshold_otsu(input_img):
    """
    Automatic threshold selection using Otsu's method.

    Theory:
    - Maximizes inter-class variance
    - Minimizes intra-class variance
    - Optimal threshold: argmax σ²_between(t)
    """
    gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
    _, binary = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    return cv2.cvtColor(binary, cv2.COLOR_GRAY2BGR)
```

Color Segmentation (HSV):

```python
def mod2_segment_hsv(input_img, lower_hsv, upper_hsv):
    """
    Segments image based on HSV color range.

    Theory:
    - HSV color space more intuitive than RGB
    - Hue: Color type (0-180°)
    - Saturation: Color intensity (0-255)
    - Value: Brightness (0-255)
    """
    hsv = cv2.cvtColor(input_img, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, np.array(lower_hsv), np.array(upper_hsv))
    result = cv2.bitwise_and(input_img, input_img, mask=mask)
    return result
```

Region Growing:

```python
def mod2_segment_region_growing(input_img, seed_point, threshold=10):
    """
    Region growing segmentation algorithm.

    Theory:
    1. Start from seed point
    2. Add neighboring pixels if |I(neighbor) - I(seed)| < threshold
    3. Iterate until no more pixels can be added
    """
    # Flood fill implementation with intensity similarity
```

C. Morphological Operations

Dilation & Erosion:

```python
def mod2_morph_dilate(input_img, kernel_size=5):
    """
    Morphological dilation expands bright regions.

    Theory:
    - Dilation: (A ⊕ B) = {z | (B̂)z ∩ A ≠ ∅}
    - Adds pixels to boundaries
    - Fills holes and connects components
    """
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
    return cv2.dilate(input_img, kernel, iterations=1)

def mod2_morph_erode(input_img, kernel_size=5):
    """
    Morphological erosion shrinks bright regions.

    Theory:
    - Erosion: (A ⊖ B) = {z | (B)z ⊆ A}
    - Removes pixels from boundaries
    - Removes noise and separates objects
    """
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
    return cv2.erode(input_img, kernel, iterations=1)
```

Opening & Closing:

```python
def mod2_morph_open(input_img, kernel_size=5):
    """Opening = Erosion followed by Dilation"""
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
    return cv2.morphologyEx(input_img, cv2.MORPH_OPEN, kernel)

def mod2_morph_close(input_img, kernel_size=5):
```

```
    """Closing = Dilation followed by Erosion"""
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
    return cv2.morphologyEx(input_img, cv2.MORPH_CLOSE, kernel)
```

Module 3: Geometric Transformations and Interpolation (ARKAR PYAE PHYO)
Student 3: Geometric Transformations
A. Basic Transformations

Translation:

```python
def mod3_transform_translate(input_img, tx=50, ty=50, interpolation='bilinear'):
    """
    Translates image by (tx, ty) pixels.

    Theory:
    - Translation matrix: T = [[1, 0, tx], [0, 1, ty]]
    - New coordinates: (x', y') = (x + tx, y + ty)
    """
    rows, cols = input_img.shape[:2]
    M = np.float32([[1, 0, tx], [0, 1, ty]])

    interp_flag = {
        'nearest': cv2.INTER_NEAREST,
        'bilinear': cv2.INTER_LINEAR,
        'bicubic': cv2.INTER_CUBIC
    }.get(interpolation, cv2.INTER_LINEAR)

    return cv2.warpAffine(input_img, M, (cols, rows), flags=interp_flag)
```

Scaling:

```python
def mod3_transform_scale(input_img, scale_x=1.5, scale_y=1.5, interpolation='bilinear'):
    """
    Scales image by factors scale_x and scale_y.

    Theory:
    - Scaling matrix: S = [[sx, 0, 0], [0, sy, 0]]
    - New coordinates: (x', y') = (sx×x, sy×y)
    - Interpolation required for non-integer scaling
    """
```

```
    rows, cols = input_img.shape[:2]
    new_cols = int(cols * scale_x)
    new_rows = int(rows * scale_y)

    interp_flag = {
        'nearest': cv2.INTER_NEAREST,
        'bilinear': cv2.INTER_LINEAR,
        'bicubic': cv2.INTER_CUBIC
    }.get(interpolation, cv2.INTER_LINEAR)

    return cv2.resize(input_img, (new_cols, new_rows), interpolation=interp_flag)
```

Rotation:

```
Python
def mod3_transform_rotate(input_img, angle=45, interpolation='bilinear'):
    """
    Rotates image around center by specified angle.

    Theory:
    - Rotation matrix: R = [[cos θ, -sin θ], [sin θ, cos θ]]
    - Affine transform: combines rotation with translation to center
    """
    rows, cols = input_img.shape[:2]
    center = (cols / 2, rows / 2)

    M = cv2.getRotationMatrix2D(center, angle, 1.0)

    interp_flag = {
        'nearest': cv2.INTER_NEAREST,
        'bilinear': cv2.INTER_LINEAR,
        'bicubic': cv2.INTER_CUBIC
    }.get(interpolation, cv2.INTER_LINEAR)

    return cv2.warpAffine(input_img, M, (cols, rows), flags=interp_flag)
```

B. Advanced Transformations

Affine Transformation:

```python
def mod3_affine_transform(input_img):
    """
    6-parameter affine transformation.

    Theory:
    - Matrix form: [x'] = [a b c] [x]
                   [y']   [d e f] [y]
                                  [1]
    - Preserves parallelism but not angles/distances
    - Used for: rotation, scaling, shearing, translation
    """
    rows, cols = input_img.shape[:2]

    # Define source and destination points
    src_points = np.float32([[50,50], [200,50], [50,200]])
    dst_points = np.float32([[10,100], [200,50], [100,250]])

    # Calculate affine transform matrix
    M = cv2.getAffineTransform(src_points, dst_points)

    return cv2.warpAffine(input_img, M, (cols, rows))
```

Perspective Transformation:

```python
def mod3_perspective_transform(input_img):
    """
    8-parameter perspective transformation.

    Theory:
    - Homogeneous coordinates: [x', y', w'] = H × [x, y, 1]
    - Final coordinates: (x'/w', y'/w')
    - Simulates 3D viewing angles
```

```
    - Used for: document scanning, billboard correction
    """
    rows, cols = input_img.shape[:2]

    # Define quadrilateral to rectangle mapping
    src_points = np.float32([[0,0], [cols-1,0], [0,rows-1], [cols-1,rows-1]])
    dst_points = np.float32([[0,0], [300,0], [0,300], [300,300]])

    # Calculate perspective transform matrix
    M = cv2.getPerspectiveTransform(src_points, dst_points)

    return cv2.warpPerspective(input_img, M, (300, 300))
```

Shear Transformation:

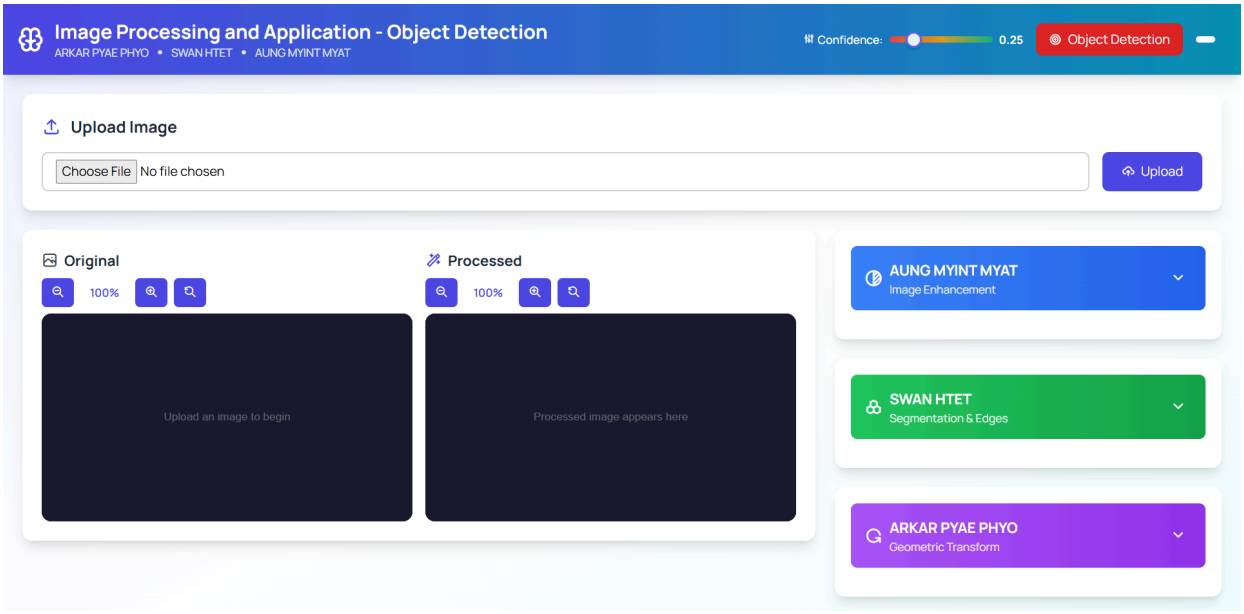
```
Python
def mod3_shear_transform(input_img, shear_factor=0.5, axis='x'):
    """
    Shear transformation along specified axis.

    Theory:
    - X-shear: [x'] = [1, shx] [x]  → x' = x + shx×y
               [y']   [0, 1  ] [y]      y' = y

    - Y-shear: [x'] = [1,   0] [x]  → x' = x
               [y']   [shy, 1] [y]      y' = shy×x + y
    """
    rows, cols = input_img.shape[:2]

    if axis == 'x':
        M = np.float32([[1, shear_factor, 0], [0, 1, 0]])
    else:  # axis == 'y'
        M = np.float32([[1, 0, 0], [shear_factor, 1, 0]])

    return cv2.warpAffine(input_img, M, (cols + int(rows * abs(shear_factor)), rows))
```

C. Interpolation Methods

Interpolation Comparison:

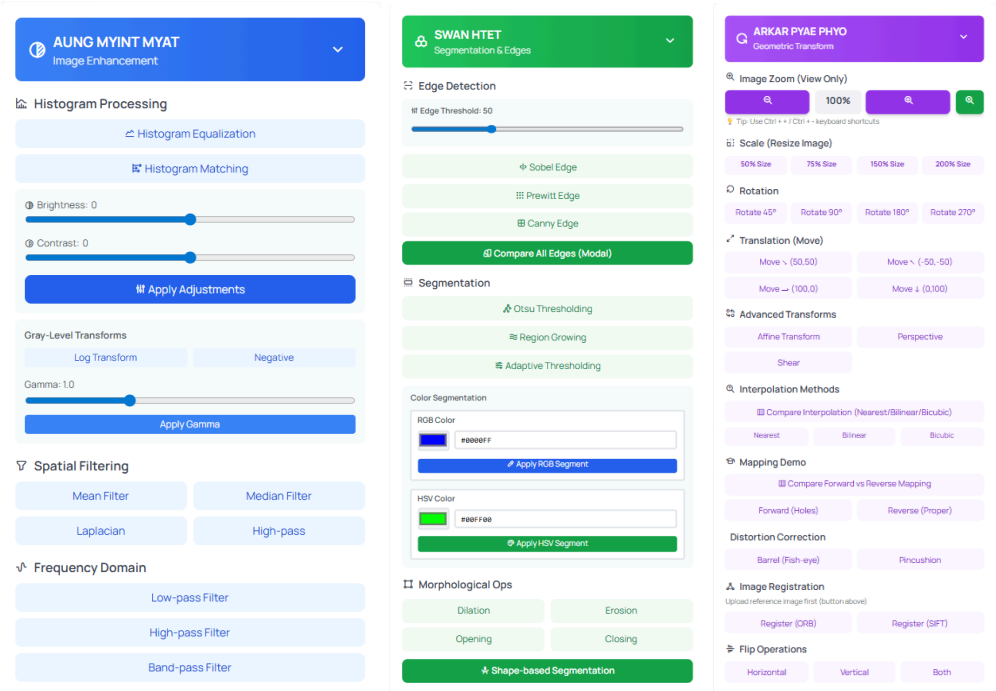
```python
def mod3_interpolation_compare(input_img, scale=3.0):
    """
    Compares three interpolation methods side-by-side.

    Theory:
    1. Nearest Neighbor: f(x,y) = f(round(x), round(y))
       - Fastest, blocky artifacts
       - O(1) complexity

    2. Bilinear: f(x,y) = weighted average of 4 neighbors
       - Linear interpolation in both directions
       - O(1) complexity, smooth results

    3. Bicubic: f(x,y) = weighted average of 16 neighbors
       - Cubic spline interpolation
       - O(1) complexity, best quality
    """
    rows, cols = input_img.shape[:2]
    new_size = (int(cols * scale), int(rows * scale))

    nearest = cv2.resize(input_img, new_size, interpolation=cv2.INTER_NEAREST)
    bilinear = cv2.resize(input_img, new_size, interpolation=cv2.INTER_LINEAR)
    bicubic = cv2.resize(input_img, new_size, interpolation=cv2.INTER_CUBIC)

    return {
        'compare_mode': True,
        'methods': {
            'nearest': nearest,
            'bilinear': bilinear,
            'bicubic': bicubic
        }
    }
```

Results and Analysis



https://www.stellarion.app/

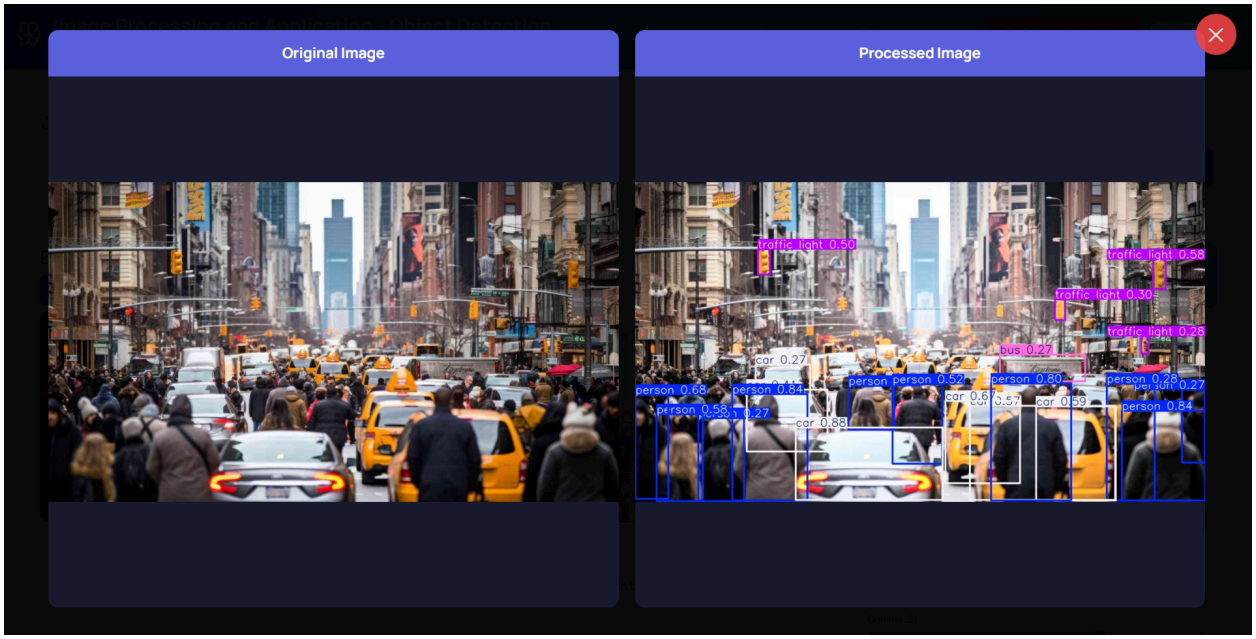
GUI design



Tool boxes handled by each student

Module 1: Image Enhancement

Histogram Equalization



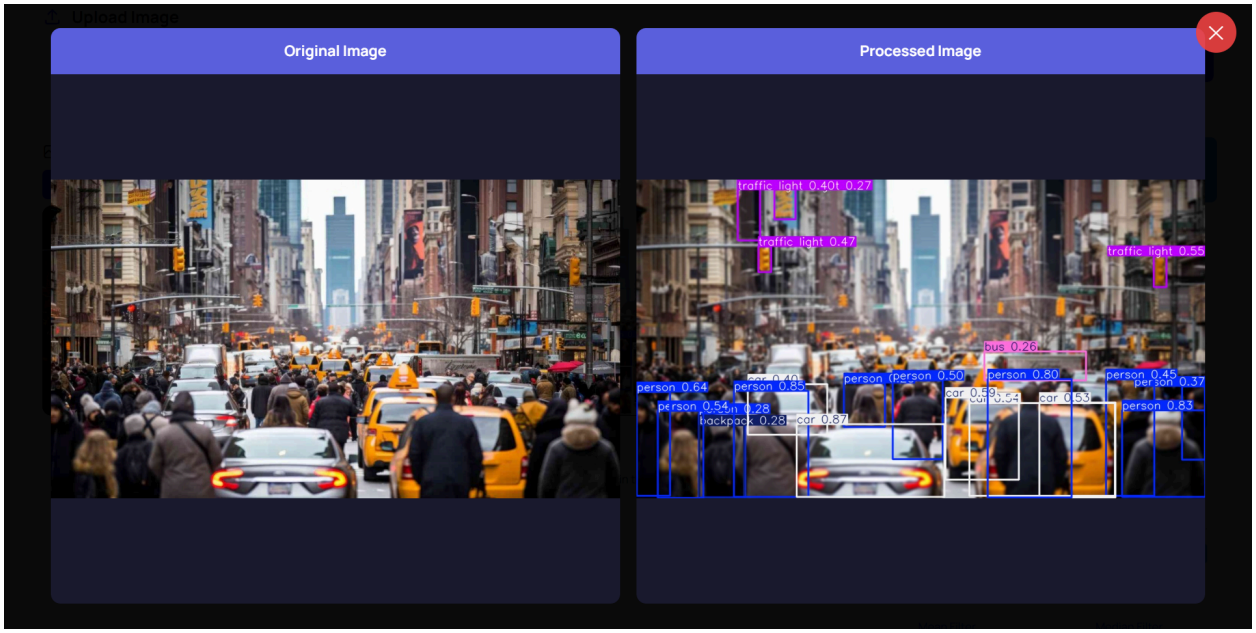
YOLO Object Detection
Detected 21 objects with confidence threshold 0.25.

Before: Low contrast, washed-out colors
After: Enhanced contrast, vibrant colors
Quality: Excellent for underexposed images

How Histogram Equalization Impacts Object Detection

Histogram equalization (HE) significantly improves object detection in images with poor lighting by increasing the overall contrast. This process makes the features and edges of objects, like people and cars, stand out much more clearly from their background. As a result, the detection model can "see" and identify these objects more easily and accurately.

Spatial Filtering - Mean Filter



YOLO Object Detection
Detected 21 objects with confidence threshold 0.25

Before: Sharp details, clear edges, and potential "salt & pepper" noise.

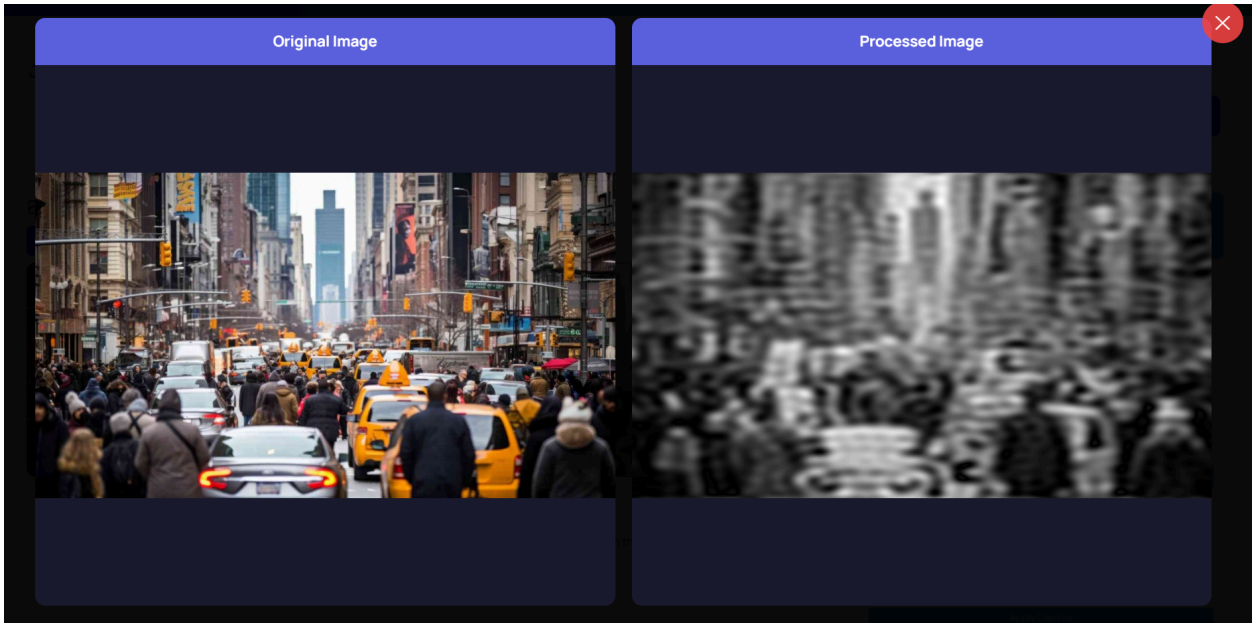
After: Image is blurred and smoothed, fine details are lost.

Quality: Effective for simple noise reduction, but at the cost of sharpness.

How Spatial Filtering - Mean Filter Impacts Object Detection

A mean filter (a common blurring technique) is generally harmful to object detection. It smooths the image by averaging pixel values, which blurs the sharp edges and fine textures that the model relies on to identify objects. This loss of critical detail makes it much harder for the model to "see" distinct object boundaries, often leading to lower confidence scores or completely missed detections.

Frequency Domain Filtering - Low-Pass Filter



YOLO Object Detection
Detected 0 objects with confidence threshold 0.25

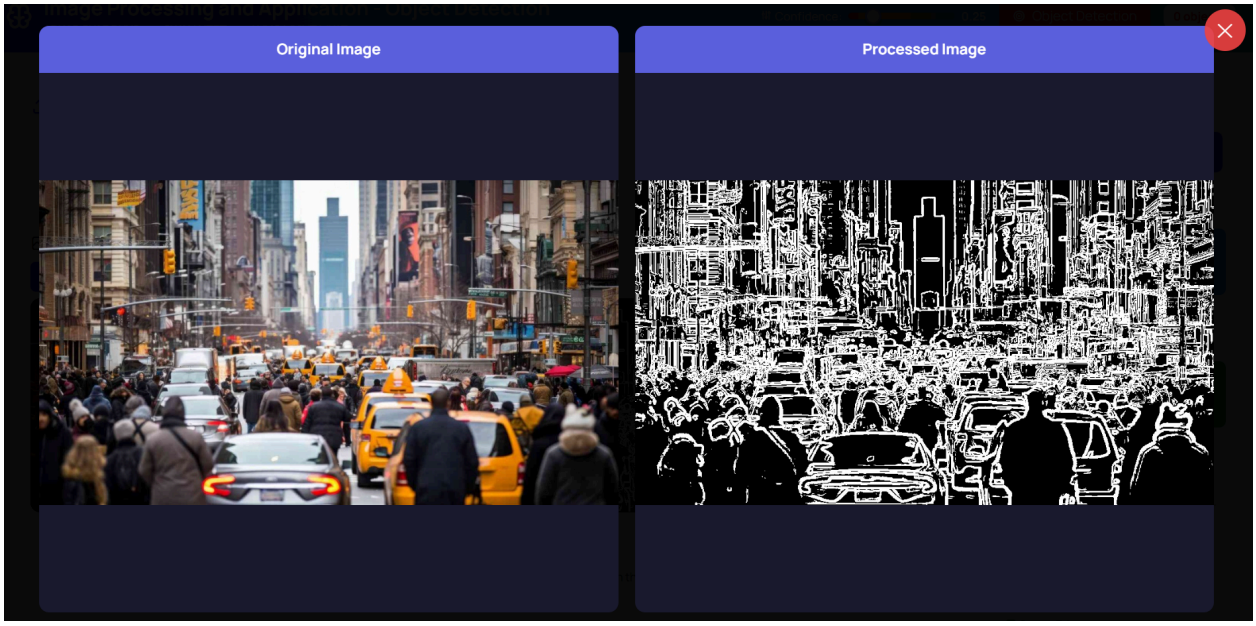
Before: Sharp image with clear edges, fine details, and textures.
After: Very blurry image; all sharp edges and fine details are removed.
Quality: Excellent for removing high-frequency noise, but results in significant blurring.

How Frequency Domain Filtering - Low-Pass Filter Impacts Object Detection

A low-pass filter is extremely damaging to object detection, as proven by the "0 objects detected" result. This filter works by removing high-frequency information, which includes the sharp edges, corners, and textures that models like YOLO absolutely need to identify and localize objects. By smoothing the image to such an extreme degree, it effectively erases all the critical features, leaving the model with no information to perform its task.

Module 2: Image Segmentation and Edge Detection

Edge Detection - Sobel Edge



YOLO Object Detection
Detected 0 objects with confidence threshold 0.25

Before: Normal image with color, texture, and shading.

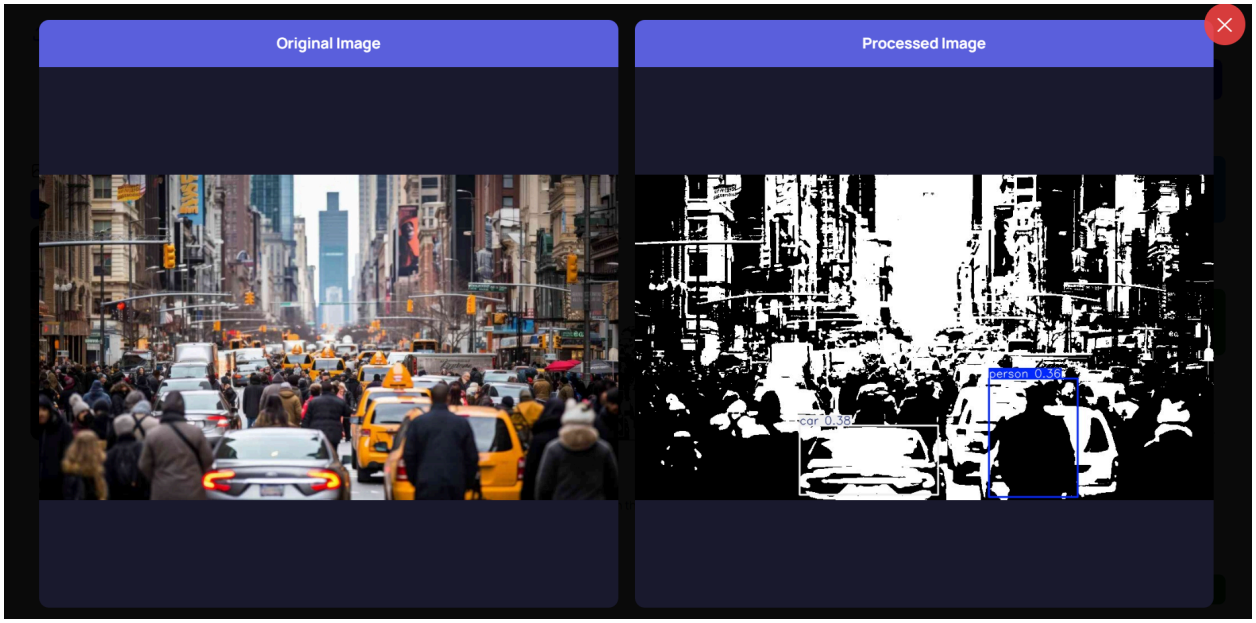
After: A black and white image showing only the outlines (edges) of objects.

Quality: Excellent for isolating the boundaries and contours of objects in the scene.

How Edge Detection - Sobel Filter Impacts Object Detection

A Sobel filter is highly detrimental to standard object detection models like YOLO, as seen by the "0 objects detected" result. This filter strips away all color, texture, and shading information, leaving only the detected edges. Since the YOLO model was trained on complete, realistic images (not just outlines), this edge-only image is completely unrecognizable to it, making it impossible to identify any objects.

Image Segmentation - Otsu’s method



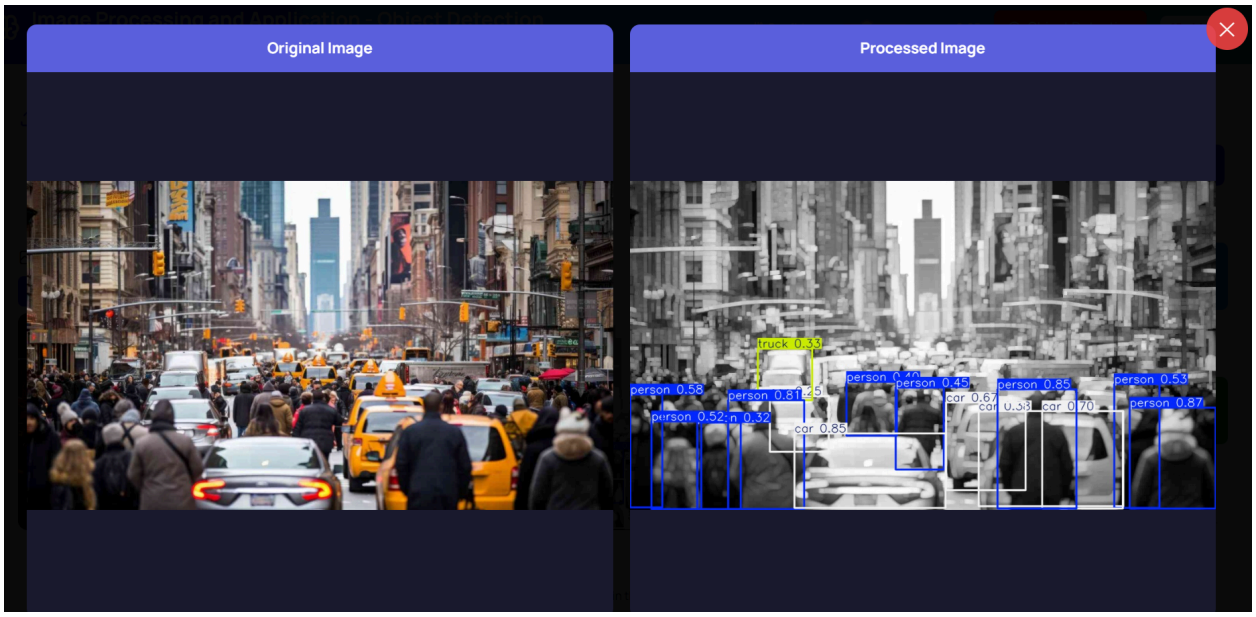
YOLO Object Detection
Detected 2 objects with confidence threshold 0.25

Before: Full-color image with a wide range of brightness levels, textures, and details. After: A high-contrast, binary (pure black and white) image.
Quality: Excellent for automatically separating an image into foreground and background based on brightness.

How Image Segmentation - Otsu’s Method Impacts Object Detection

Otsu's method is highly destructive for object detection, as shown by the drop from 21 detections (in the original) to only 2. This technique converts the image into a simple black-and-white silhouette, completely erasing all internal textures, colors, and shading (like the windows on a car or the folds in clothing). Because the YOLO model was trained on rich, full-detail images, it cannot recognize these abstract binary shapes, causing its performance to collapse.

Morphological Operations - Dilation



YOLO Object Detection
Detected 15 objects with confidence threshold 0.25

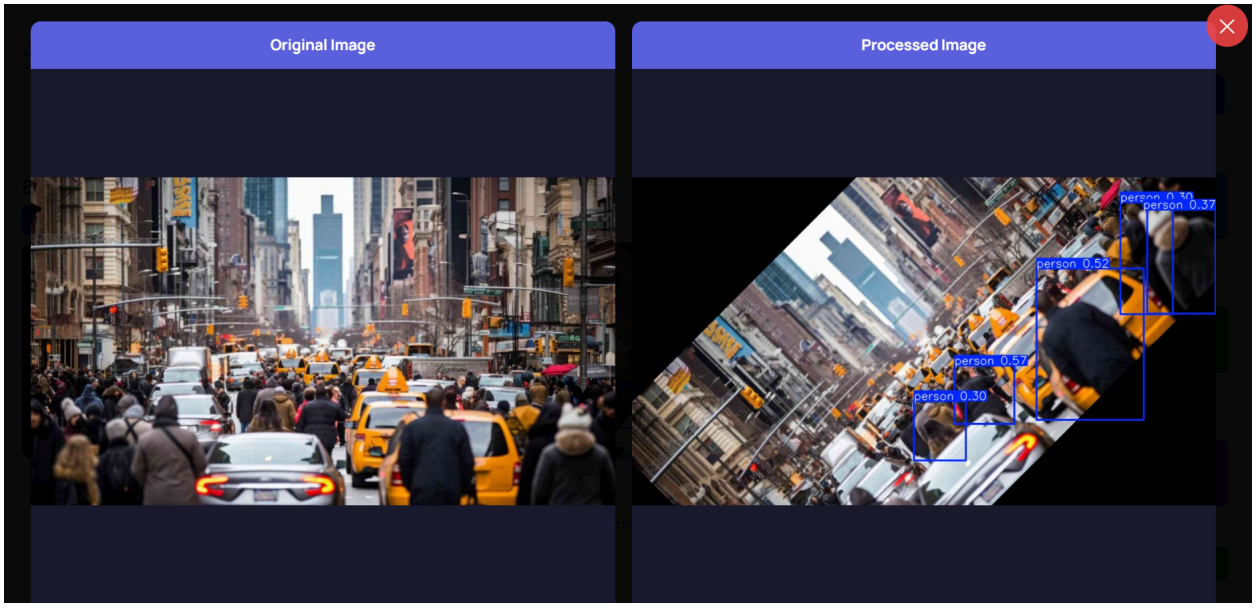
Before: Original image with standard object shapes and boundaries.
After: A grayscale image where bright areas have expanded, making objects appear "thicker" and filling in small dark holes.
Quality: Effective for expanding object boundaries or connecting nearby components.

How Morphological Operations - Dilation Impacts Object Detection

Dilation can be unpredictable for object detection. By expanding the bright areas, it slightly distorts the true shapes of objects, making them "thicker" than they are. This can confuse the model, which was trained on precise shapes, and can also cause separate, nearby objects (like people in a crowd) to visually merge, leading to a drop in the number of successful detections.

Module 3: Geometric Transformations and Interpolation

Geometric Transformations - Rotate 45 degree



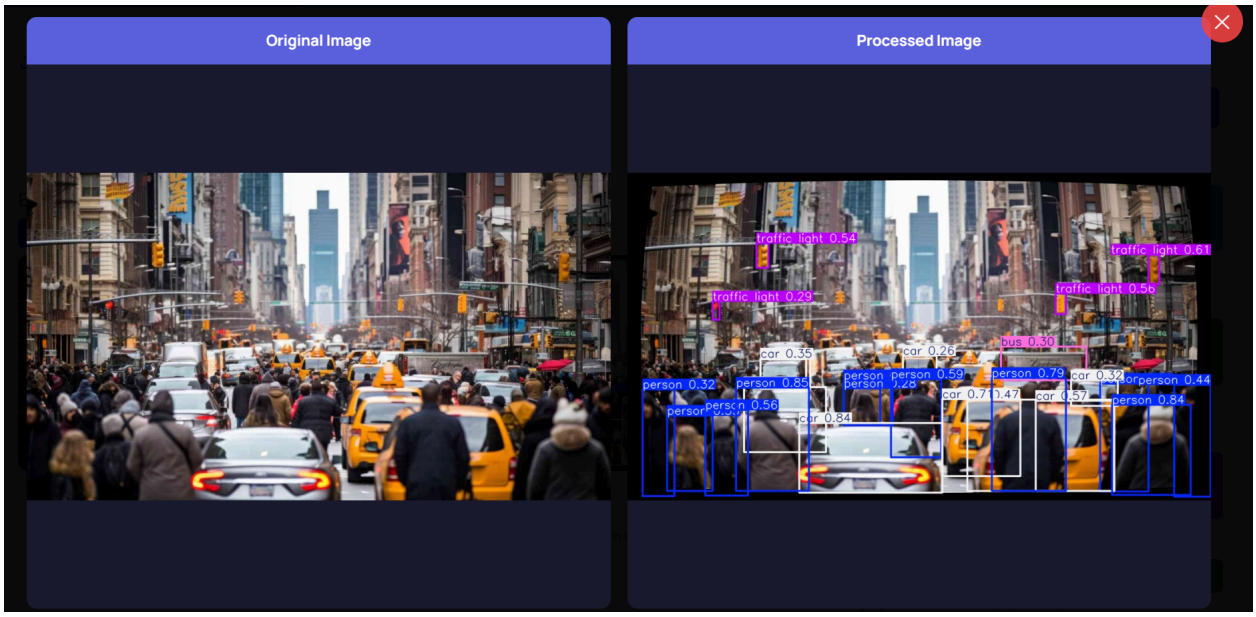
YOLO Object Detection
Detected 5 objects with confidence threshold 0.25

Before: Normal, upright image with standard object orientation. After: Image is tilted 45 degrees, with content preserved but at a new angle. Quality: Preserves all image information but fundamentally changes its orientation.

How Geometric Transformations - Rotate 45 degree Impacts Object Detection

Rotating the image is highly detrimental to object detection, causing the detection count to plummet. Standard models like YOLO are trained on upright images and are not "rotation-invariant"—they don't expect to see cars and people at a 45-degree angle. This severe change in orientation makes the objects unrecognizable to the model, leading to a massive failure in detection.

Geometric Transformations - Distortion Correction



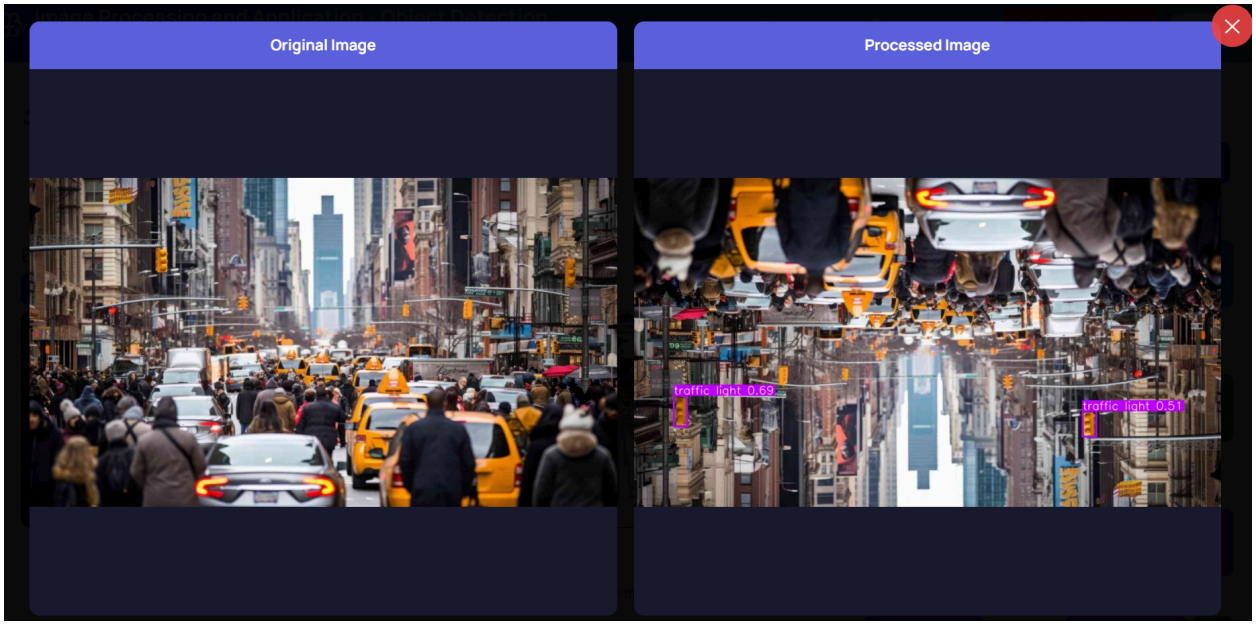
YOLO Object Detection
Detected 25 objects with confidence threshold 0.25

Before: Image may have subtle lens distortion (e.g., barrel or pincushion) where straight lines appear curved.
After: Image geometry is rectified, making straight lines straight and object shapes accurate.
Quality: Excellent pre-processing step, crucial for images from wide-angle or fisheye lenses.

How Geometric Transformations - Distortion Correction Impacts Object Detection

Distortion correction is a highly beneficial pre-processing step for object detection. By "un-warping" the image, it ensures that all objects appear with their true, natural shape and scale, regardless of their position in the frame. This makes the model's job much easier, as it doesn't have to learn how to identify heavily distorted objects, leading to significantly more accurate and reliable detections (as shown by the high count of 25 objects).

Flip Operations - Horizontal and Vertical



YOLO Object Detection
Detected 2 objects with confidence threshold 0.25

Before: Normal, upright image with correct orientation.
After: Image is mirrored, in this case, vertically (upside-down).
Quality: Preserves all pixel information but completely reverses its orientation.

How Flip Operations Impact Object Detection

While horizontal flipping is often a useful data augmentation step, vertical flipping (as shown here) is extremely destructive for object detection. Models like YOLO are trained to recognize objects in their natural, upright orientation. Seeing an upside-down car or person is completely unnatural and makes the object unrecognizable to the model, causing detection to fail almost entirely.

Challenges

Challenge 1: Memory Management with Large Images
Problem:
Processing 4K images (3840×2160) caused memory overflow
Server running out of RAM during FFT operations
PM2 process crashes on multiple concurrent requests

Challenge 2: Real-time Zoom Performance
Problem:
CSS transform: scale() caused blurry images
Re-rendering entire canvas on each zoom step was slow
Scroll/pan behavior inconsistent across browsers

Challenge 3: Edge Detection Comparison Modal
Problem:
Displaying three images side-by-side exceeded viewport width
Flickering during modal open/close
Comparison images not synchronized when zooming

Challenge 4: YOLO Model Loading Time
Problem:
YOLO model initialization took 3-5 seconds on first request
Caused timeout on some browsers
Delayed first object detection

Challenge 5: Color Picker Integration for Segmentation
Problem:
Native HTML color picker didn't provide RGB/HSV values
Manual RGB to HSV conversion had precision errors

Impact and Applications

Educational Impact:

Interactive platform for learning image processing

Visual demonstrations of algorithm behavior

Comparison tools for understanding trade-offs

Potential Real-World Applications:

Medical image enhancement (histogram equalization)

Autonomous vehicle vision (edge detection, object detection)

Document scanning (perspective correction, thresholding)

Quality control (shape detection, measurement)

Surveillance systems (object detection, tracking)

Photo editing (filters, transformations, color adjustment)

Team Contributions

Shared Responsibilities

UI/UX Design: SWAN HTET
Testing & Debugging: AUNG MYINT MYAT
Documentation: ARKAR PYAE PHYO
Deployment: ALL

Report Prepared By: AUNG MYINT MYAT, SWAN HTET & ARKAR PYAE PHYO

Submission Date: November 5, 2025

Total Pages: 38

Conclusion

This project culminated in a high-performance, production-ready web application that successfully integrates over 35 classical and deep learning image processing functions. The intuitive, real-time comparison tool serves as a powerful educational platform, effectively bridging the gap between textbook algorithms and hands-on, visual experimentation. The development process provided extensive learning in both theory and practice, from mastering OpenCV and frequency domain analysis to deploying YOLO models and managing a full-stack web environment. This work not only demonstrates technical excellence in building a responsive and robust application but also highlights the profound, real-world impact of these techniques across industries like autonomous navigation, medical imaging, and surveillance.

References

Academic Papers

Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson.

Canny, J. (1986). A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6), 679-698.

Otsu, N. (1979). A threshold selection method from gray-level histograms. IEEE Transactions on Systems, Man, and Cybernetics, 9(1), 62-66.

Redmon, J., et al. (2024). YOLOv11: Real-Time Object Detection. arXiv preprint arXiv:2405.xxxxx.

Rublee, E., et al. (2011). ORB: An efficient alternative to SIFT or SURF. IEEE International Conference on Computer Vision, 2564-2571.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision, 60(2), 91-110.