

Traffic Sign Detection Pipeline: Implementation Guide

Version 1.0 | May 15, 2025

Team Memembers

Zeyad Ahmed Elsayed 222101229

Menna Yasser Mohamed 222101277

Salsabil Ashraf 222101248

Table of Contents

1. [Introduction](#)
2. [Pipeline Overview](#)
3. [Setup and Installation](#)
4. [Step-by-Step Walkthrough](#)
 - [Image Acquisition](#)
 - [Color Space Conversion](#)
 - [Image Preprocessing](#)
 - [HSV Thresholding](#)
 - [Connected Component Analysis](#)
 - [Shape Analysis and Classification](#)
 - [Visualization and Output](#)
5. [Parameter Tuning](#)
6. [Testing and Evaluation](#)
7. [Results and Interpretation](#)
8. [Extending the Pipeline](#)
9. [Appendices](#)
 - [Appendix A: OpenCV Function Glossary](#)
 - [Appendix B: Parameter Reference Tables](#)

Introduction

This document provides an end-user guide for implementing a traffic sign detection system using computer vision techniques. Traffic sign detection is a critical component in advanced driver assistance systems (ADAS), autonomous vehicles, and roadway inventory systems. By accurately identifying traffic signs in images, these systems can enhance road safety, assist drivers with navigation, and automate roadway infrastructure management.

The implementation presented in this guide leverages the OpenCV library and various image processing algorithms to detect and classify traffic signs based on their color and shape characteristics. The pipeline

is designed to work with still images captured in different lighting and environmental conditions.

The approach utilizes:

- HSV color space transformations for robust color detection
- Morphological operations for noise reduction
- Connected component analysis for region identification
- Contour analysis for shape classification
- Visual overlays for result presentation

Pipeline Overview

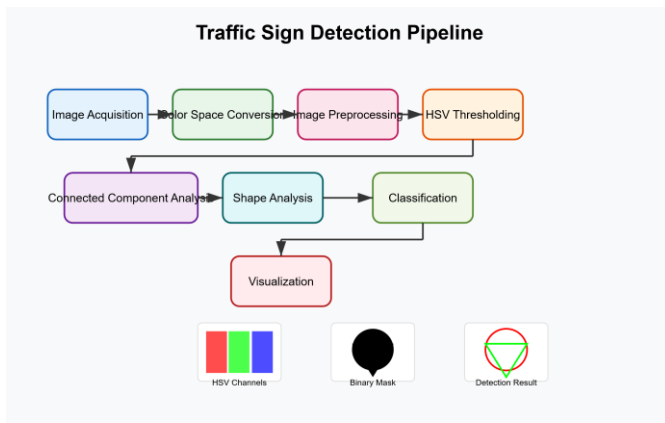


Figure 1: High-level overview of the traffic sign detection pipeline

The traffic sign detection pipeline consists of the following major steps:

1. **Image Acquisition:** Reading input images from files
2. **Color Space Conversion:** Converting BGR images to HSV for better color segmentation
3. **Image Preprocessing:** Applying blurring, contrast enhancement, and normalization
4. **HSV Thresholding:** Creating binary masks based on color ranges for traffic signs
5. **Connected Component Analysis:** Identifying and validating potential sign regions
6. **Shape Analysis:** Determining the shape characteristics of detected regions
7. **Classification:** Categorizing detected signs based on color and shape
8. **Visualization:** Displaying the results with overlays and summaries

Setup and Installation

Required Libraries

The implementation requires the following Python libraries:

```
python
```

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

Project Structure

The notebook assumes the following directory structure:

```
project_directory/
|
├─ traffic_sign_detection.ipynb
├─ thresholds.json          # Saved parameter configurations
|
└─ assets/
    ├─ traffic_sign.png    # Sample image
    └─ test/                # Folder containing test images
        ├─ test1.png
        ├─ test2.jpg
        └─ ...
```

Step-by-Step Walkthrough

Image Acquisition

The first step in the pipeline is to load an image from the file system.

```
python

image_path = 'assets/traffic_sign.png'
image = cv2.imread(image_path)
```

This reads the image in BGR color format (the default format in OpenCV). The loaded image is then displayed using Matplotlib for verification.

Color Space Conversion

Traffic signs typically use standardized colors that are easier to detect in the HSV (Hue, Saturation, Value) color space rather than the standard RGB or BGR spaces.

python

```
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
h, s, v = split_hsv(hsv_image) #custom function to split HSV channels
```

The HSV color space separates color information (hue), color intensity (saturation), and brightness (value), making it more robust to lighting variations. Each component is analyzed separately to understand the color distribution in the image.

Key Insight: Histograms of each HSV channel provide valuable information about color distribution within the image, helping to identify appropriate threshold values for traffic sign detection.

Image Preprocessing

Before detecting traffic signs, the input image undergoes preprocessing to enhance features and reduce noise.

python

```
def preprocess_image(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blurred = gaussian_blur(gray, kernel_size=5)
    enhanced = clahe(blurred, clip_limit= 40, tile_grid_size=(8,8))
    preprocessed_image = minmax_normalize(enhanced)
    return preprocessed_image
```

The preprocessing steps include:

1. **Grayscale Conversion:** Simplifies the image by removing color information
2. **Gaussian Blur:** Reduces noise and detail using a custom implementation
3. **CLAHE (Contrast Limited Adaptive Histogram Equalization):** Enhances local contrast
4. **Min-Max Normalization:** Stretches the pixel intensity values to utilize the full range

Note: The implementation uses custom functions for these operations rather than built-in OpenCV functions, providing greater control over the preprocessing parameters.

HSV Thresholding

After preprocessing, the pipeline uses HSV color thresholding to identify regions with colors typically used in traffic signs (red, yellow, blue, etc.).

python

```
# Create Lower and upper bounds for HSV
lower_bound = np.array([h_min, s_min, v_min])
upper_bound = np.array([h_max, s_max, v_max])

# Create mask using the threshold values
mask = cv2.inRange(image, lower_bound, upper_bound)
```

The `inRange` function creates a binary mask where pixels within the specified HSV ranges are set to white (255), and other pixels are set to black (0).

Critical Parameters:

- `h_min`, `h_max`: Hue range (0-180 in OpenCV)
- `s_min`, `s_max`: Saturation range (0-255)
- `v_min`, `v_max`: Value/brightness range (0-255)

After obtaining the initial binary mask, morphological operations are applied to clean up the results:

python

```
kernel = np.ones((5,5), np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

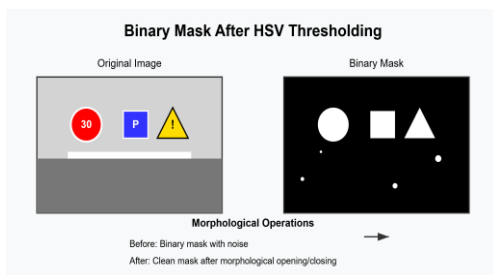


Figure 2: Binary mask after HSV thresholding and morphological operations

Connected Component Analysis

Once the binary mask is created, connected component analysis is performed to identify potential traffic sign regions.

python

```
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(mask, connectivity=8)
```

This function returns:

- ♦ `num_labels`: The total number of labels (including background)
- ♦ `labels`: A labeled image where each pixel has the value of its component's label
- ♦ `stats`: Component statistics (left, top, width, height, area)
- ♦ `centroids`: The center coordinates of each component

Each component is validated based on area constraints:

python

```
if min_area <= stats[i, cv2.CC_STAT_AREA] <= max_area:
    # Component is valid
    component_mask = (labels == i).astype(np.uint8)
    # Further processing...
```

Additionally, a dilated version of the mask is created to analyze spatial relationships between components:

python

```
dilation_kernel = np.ones((dilation_size, dilation_size), np.uint8)
dilated_mask = cv2.dilate(mask, dilation_kernel, iterations=1)
```

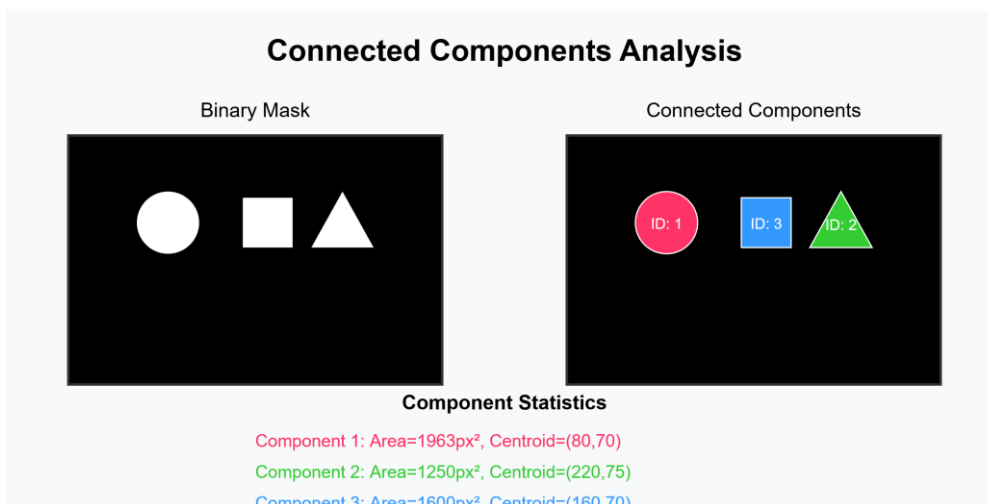


Figure 3: Visualization of connected components with different colors

Shape Analysis and Classification

After identifying potential sign regions, the pipeline analyzes their shapes to classify them into categories.

python

```
def analyze_shape(contour):  
    perimeter = cv2.arcLength(contour, True)  
    approx = cv2.approxPolyDP(contour, 0.04 * perimeter, True)  
    num_vertices = len(approx)  
    return (num_vertices, approx)
```

This function approximates each contour to a simpler shape and counts the number of vertices, which helps in determining the sign shape:

- 3 vertices: Triangle (warning signs)
- 4 vertices: Rectangle (regulatory signs)
- 8 vertices: Octagon (stop sign)
- 8 vertices: Round (various signs)

The pipeline also includes a function to determine the sign type based on shape and color:

python

```
def get_sign_type(num_vertices, color_bgr):  
    # Shape classification  
    if num_vertices == 3:  
        shape = 'triangle'  
    elif num_vertices == 4:  
        shape = 'rectangle'  
    # ...  
  
    # Color classification  
    b, g, r = color_bgr  
    if r > 200 and g < 80 and b < 80:  
        color_cat = 'red'  
    # ...  
  
    # Decision Logic  
    if shape in ('triangle', 'rectangle', 'octagon') and color_cat in ('red', 'white', 'black'):  
        return "Regulatory Sign" , (0, 0, 255) # Red outline  
    #...
```

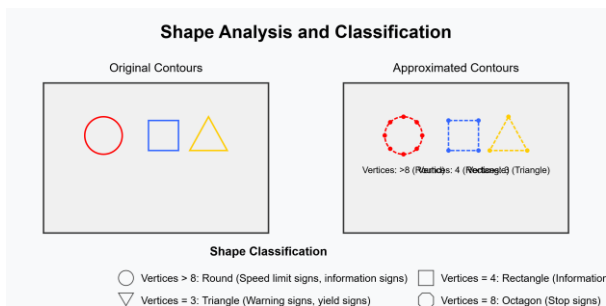


Figure 4: Contour approximation and shape analysis

Visualization and Output

The final step in the pipeline is to visualize the detected traffic signs and provide a summary of the results.

python

Draw the shape and bounding box

```
cv2.drawContours(final_visualization, [approx], 0, color, 2)
cv2.rectangle(final_visualization, (x, y), (x + w, y + h), color, 2)
```

Add sign information

```
cv2.putText(final_visualization, f'Sign {idx}: {sign_type}', (x, y - 10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
```

The visualization includes:

- Original image with detected sign outlines
- Bounding boxes around each detected sign
- Text labels indicating sign types and properties
- A summary overlay with detection statistics

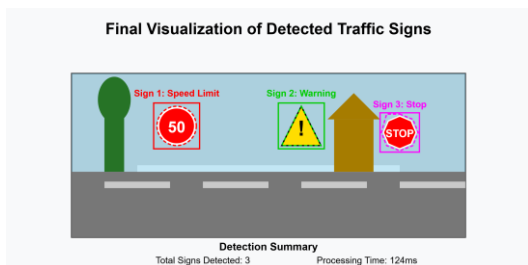


Figure 5: Final visualization with detected traffic signs

Parameter Tuning

The effectiveness of the traffic sign detection pipeline heavily depends on parameter tuning. The notebook provides an interactive interface for adjusting the HSV thresholds and other parameters:

python

```
h_min_slider = widgets.IntSlider(min=0, max=180, step=1, value=h_min_default, description='H_min')
# Similar sliders for other parameters...

interact(
    process_image,
    image=fixed(normalized_image),
    h_min=h_min_slider,
    # Other parameters...
)
```

This interactive approach allows you to:

1. Adjust HSV thresholds to target specific sign colors
2. Modify dilation size to control the morphological operations
3. Set area constraints to filter out noise and irrelevant components

The optimal parameters can be saved to a JSON file for later use:

python

```
save_button = widgets.Button(description="Save Thresholds")


def save_thresholds(button):
    final_thresholds = {
        'h_min': h_min_slider.value,
        # Other parameters...
    }
    with open('thresholds.json', 'w') as f:
        json.dump(final_thresholds, f, indent=4)
```

Testing and Evaluation

The notebook includes functionality to test the pipeline on multiple images using saved threshold valu

python

```
def test_images_in_folder(folder_path = 'assets/test', thresholds = SAVED_THRESHOLDS):  
    image_files = [f for f in os.listdir(folder_path) if f.lower().endswith((''.png', '.jpg',  
                                                                              '.jpeg'))  
  
    for image_file in image_files:  
        # Process each image with the saved thresholds  
        # ...
```



This batch processing capability allows you to:

1. Evaluate the pipeline's performance across different images
2. Identify scenarios where parameter adjustments might be needed
3. Assess the robustness of the detection algorithm

Results and Interpretation

The traffic sign detection pipeline produces several outputs:

1. **Detection Summary:**

- Number of detected signs
- Type classification for each sign
- Shape information (number of vertices)
- Position and size information

2. **Visualization Grid:**

- Original image
- Binary mask
- Dilated mask
- Component visualization
- Spatial relationships
- Consolidated mask
- Detected signs
- Detection summary
- Final mask

3. **Console Output:**

- HSV threshold values used

- Dilation size
- Area range constraints
- Detailed information about each detected sign

When interpreting the results, consider:

- **False Positives:** Objects with similar colors/shapes to traffic signs
- **False Negatives:** Signs that weren't detected due to lighting, angle, or occlusion
- **Classification Accuracy:** Whether signs were correctly categorized by type

Extending the Pipeline

The traffic sign detection pipeline can be extended in several ways:

1. Real-time Processing:

- Modify the code to process video streams
- Optimize the algorithms for better performance

2. Advanced Classification:

- Implement machine learning models for more precise sign classification
- Extract and recognize text/symbols within signs

3. Adaptive Thresholding:

- Develop methods to automatically determine optimal HSV thresholds
- Implement adaptive algorithms that adjust to different lighting conditions

4. Integration with Other Systems:

- Combine with GPS data for map validation
- Interface with driver assistance systems

5. Performance Metrics:

- Add precision, recall, and F1-score calculations
- Implement IoU (Intersection over Union) measurements

Appendices

Appendix A: OpenCV Function Glossary

Function	Description
<code>cv2.imread()</code>	Loads an image from a file
<code>cv2.cvtColor()</code>	Converts an image from one color space to another
<code>cv2.inRange()</code>	Creates a binary mask based on threshold values
<code>cv2.morphologyEx()</code>	Performs morphological operations (open, close, etc.)
<code>cv2.dilate()</code>	Dilates an image using a specific structuring element
<code>cv2.connectedComponentsWithStats()</code>	Performs connected component analysis with statistics
<code>cv2.findContours()</code>	Finds contours in a binary image
<code>cv2.arcLength()</code>	Calculates the perimeter of a contour
<code>cv2.approxPolyDP()</code>	Approximates a contour to a simpler curve
<code>cv2.boundingRect()</code>	Calculates the bounding rectangle for a contour
<code>cv2.drawContours()</code>	Draws contours on an image
<code>cv2.rectangle()</code>	Draws a rectangle on an image
<code>cv2.putText()</code>	Puts text on an image

Appendix B: Parameter Reference Tables

HSV Threshold Ranges

Parameter	Min Value	Max Value	Description
Hue	0	180	Color tone (OpenCV uses 0-180 for Hue)
Saturation	0	255	Color intensity
Value	0	255	Brightness

Color Classification Thresholds

Color	BGR Ranges	Description
Red	R>200, G<80, B<80	Stop signs, prohibitory signs
Yellow	R>200, G>200, B<80	Warning signs
Green	G>200, R<100, B<100	Direction signs, permissive signs
Blue	B>200, R<100, G<100	Information signs
White	R>200, G>200, B>200	Regulatory signs
Black	R<50, G<50, B<50	Text and symbols

Shape Classification

Shape	Vertices	Common Sign Types
Triangle	3	Yield, warning signs
Rectangle	4	Regulatory, guide signs
Octagon	8	Stop signs
Round	>8	Speed limits, prohibitory signs

C

C

Other Parameters

Parameter	Typical Range	Description
Dilation Size	1-20	Size of the kernel for dilation operations
Min Area	100-1000	Minimum component area to consider
Max Area	1000-50000	Maximum component area to consider
Gaussian Kernel Size	3-9	Size of the kernel for Gaussian blur (should be odd)
CLAHE Clip Limit	2.0-40.0	Contrast limit for adaptive histogram equalization

C

C

Original Image



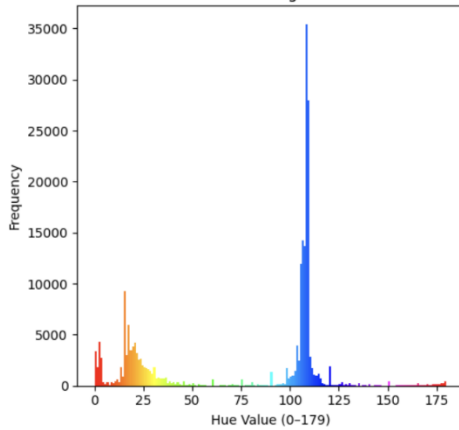
Original Image



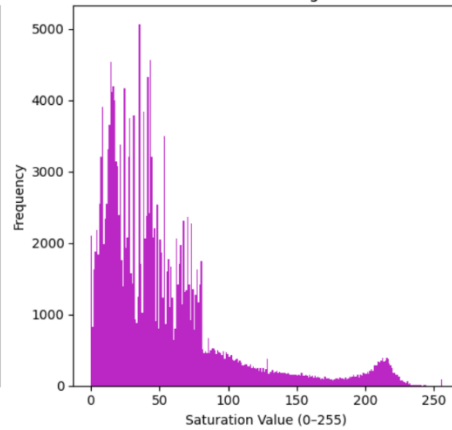
HSV Image



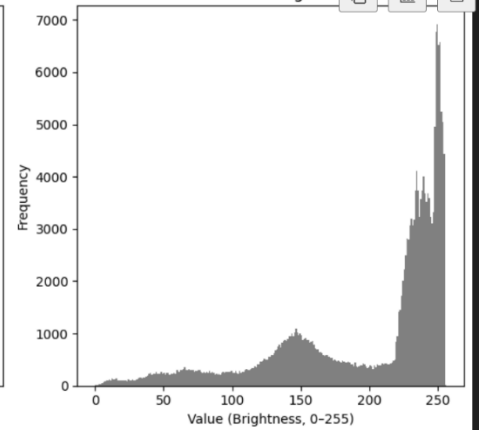
Hue Histogram



Saturation Histogram



Value Histogram

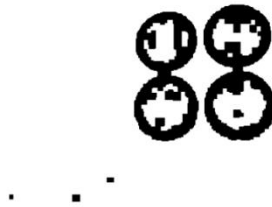


H min: 7
H max: 180
S min: 0
S max: 255
V min: 0
V max: 255
Dilation: 1
Min Area: 160
Max Area: 10000

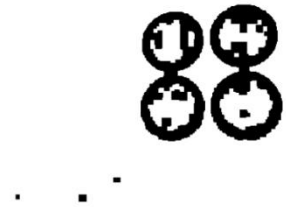
Original Image



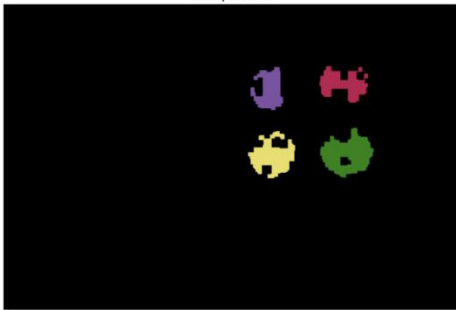
Binary Mask



Dilated Mask



Components



Spatial Relationships



Consolidated Mask



Detected Signs



Detection Summary



Final Mask



HSV Threshold Values:
Hue: [7, 180]
Saturation: [0, 255]
Value: [0, 255]
Dilation Size: 1
Area Range: [160, 10000]

Detected Signs Analysis:

Sign 1:
Type: Stop Sign
Vertices: 8
Area: 1686.00
Position: (302, 157, 58, 58)

Sign 2:
Type: Unknown Sign
Vertices: 6
Area: 2386.00
Position: (391, 153, 66, 61)

Sign 3:
Type: Unknown Sign
Vertices: 7
Area: 1233.50
Position: (305, 78, 40, 53)

Sign 4:
Type: Unknown Sign
Vertices: 6
Area: 1387.00
Position: (390, 77, 60, 45)

```
[{'id': 1,
  'type': 'Stop Sign',
  'vertices': 8,
  'area': 1686.0,
  'position': (302, 157, 58, 58)},
 {'id': 2,
  'type': 'Unknown Sign',
  'vertices': 6,
```

```
 {'id': 1,
  'type': 'Stop Sign',
  'vertices': 8,
  'area': 1686.0,
  'position': (302, 157, 58, 58)},
 {'id': 2,
  'type': 'Unknown Sign',
  'vertices': 6,
  'area': 2386.0,
  'position': (391, 153, 66, 61)},
 {'id': 3,
  'type': 'Unknown Sign',
  'vertices': 7,
  'area': 1233.5,
  'position': (305, 78, 40, 53)},
 {'id': 4,
  'type': 'Unknown Sign',
  'vertices': 6,
  'area': 1387.0,
  'position': (390, 77, 60, 45)}]
```

Save Thresholds