Fog Removal and Image Enhancement for Traffic Signs Using Python

A report submitted for the Bachelor of Technology Odd Semester of the subject
Digital Signal Processing
of the Department **Electronics and Communication Engineering**

Submitted By:

Registration Number: 20230000508 of 2023-24
Roll No: 1181700253
Name: Kunal Rabidas

Under the guidance of:
Dr. Rinku Rabidas

Department of Electronics and Communication Engineering
Triguna Sen School of Technology
Assam University, Silchar 788011

Department of Electronics and Communication Engineering
Assam University, Silchar-788011

Certificate

This is to certify that the project titled
“Fog Removal and Image Enhancement for Traffic Signs Using Python”
submitted by
Kunal Rabidas
Registration Number: 20230000508 of 2023-24

to the Department of Electronics and Communication Engineering, Assam University, Silchar
in fulfillment of the Bachelor of Technology Odd Semester of the subject
Digital Signal Processing
is a bonafide record of the work carried out by him/her/them under my supervision.It is further certified that the candidate(s) has/have complied with all the formalities as per the requirements of Assam University

Dr. Rinku Rabidas

_____

Date and Seal

Abstract

Poor visibility due to atmospheric conditions like fog and haze is a significant factor in transportation accidents. These conditions obscure critical information, such as traffic signs and signals, posing a risk to drivers. This project presents the design and implementation of a computational model to reduce or remove fog from a single digital image. The core of this project is the **Dark Channel Prior (DCP)** algorithm, a physics-based approach to image dehazing.

The system is developed in Python, utilizing libraries such as OpenCV for image processing, NumPy for numerical operations, and Matplotlib for visualization. The model first estimates the atmospheric light and transmission map from the hazy input image. It then refines the transmission map using a **Guided Filter**, a key enhancement that significantly reduces the "halo" artifacts common in basic dehazing algorithms. Finally, it recovers a clear, dehazed image by inverting the atmospheric scattering model.

The implemented application allows a user to select a hazy image via a file dialog, after which it processes the image and displays a side-by-side comparison of the "Original" and "Dehazed" outputs. The results demonstrate a dramatic improvement in image clarity, contrast, and color fidelity, successfully making obscured traffic signs and objects visible.

Acknowledgment

We would like to extend our gratitude towards our professor, **Dr. Rinku Rabidas**, for offering their assistance as and when required.

We would also like to take this opportunity to express our sincere gratitude towards our project guide **Dr. Rinku Rabidas**, Department of **Electronics and Communication Engineering**. We profoundly appreciate and are grateful for their valuable guidance, support, and encouragement to complete this project.

We also appreciate the unwavering support of our parents, seniors, and teachers throughout our journey to complete this project. Their belief in us and feedback served as a motivating factor for us to succeed.

Table of Contents

Chapter 1: The Problem of Fog in Traffic

1.1 The High Stakes of Low Visibility

Visibility is the single most critical factor for safe driving. During winter, and in many climates year-round, atmospheric conditions like fog and heavy haze create a low-visibility environment that poses a significant threat. This atmospheric veil scatters light, reduces contrast, and washes out colors, making it incredibly difficult for drivers to perceive their surroundings accurately.

This poor visibility is a direct contributing factor to a large number of road accidents annually, ranging from minor fender-benders to catastrophic multi-car pile-ups. The inability for a driver to react in time is the root cause, and this problem is severely exacerbated by fog.

1.2 The Specific Challenge to Traffic Infrastructure

The danger is not just from other vehicles. Fog obscures the very infrastructure designed to *prevent* accidents:

- **Traffic Signals**: The distinct red, yellow, and green lights become diffuse, 'glowing' orbs, making it hard to distinguish which light is active.
- **Signboards**: Stop signs, speed limits, and warning signs lose their color and sharpness, blending into the grey background.
- **Road Markings**: Lane dividers and pedestrian crossings fade, leading to driver confusion and unsafe maneuvers.

The inability to see a red light or a stop sign until it is too late is a persistent and deadly problem. Therefore, developing a reliable method to "see through" the fog and enhance driver visibility is a vital goal for improving traffic safety.

Fig 1.0: Traffic with Heavy Fog (Delhi)

Chapter 2: Approach to the Solution

2.1 Comparative Analysis of Dehazing Techniques

Several methods exist for image dehazing, each with its own trade-offs.

Method Type	Example(s)	Pros	Cons
Simple Image Processing	Histogram Equalization, Contrast Stretching	Very fast, simple to implement.	Physically inaccurate. Often fails, shifts colors, and amplifies noise.
Physics-Based (Prior-Based)	**Dark Channel Prior (DCP)**, Color Attenuation Prior	**Excellent, natural-looking results.** Based on a physical model.	Slower. Relies on assumptions (priors) that can fail (e.g., on white objects).
Machine Learning (Learning-Based)	CNNs (e.g., DehazeNet), GANs	State-of-the-art results. Can be very fast *after* training.	Requires a large, paired dataset (foggy/clear) for training. Can be a "black box."

For this project, the **Dark Channel Prior (DCP)** was chosen. It represents the best balance of high-quality, physically-based results and implementation feasibility, without the complexity and data requirements of a deep learning solution.

2.2 The Core Concept: Digital Dehazing

Our project addresses this problem not by changing the physical hardware of a camera, but by algorithmically processing the image it captures. We are implementing a digital image enhancement technique known as **image dehazing**. The core idea is to take a single foggy image, model the effects of the fog, and then mathematically "subtract" those effects to restore the original, clear scene. This reveals the underlying clarity and colors that were obscured.

2.3 The Chosen Algorithm: Dark Channel Prior (DCP)

After reviewing several methods, we chose to build our solution on the **Dark Channel Prior** (DCP) algorithm. This is a highly effective and well-regarded method for single-image haze removal. The "prior" is a statistical observation: in most haze-free images, at least one color

channel (R, G, or B) has a very low intensity (is "dark") in any given patch. Fog disrupts this prior by adding a white-ish light, brightening these dark channels. By finding these "unnaturally" bright-dark channels, we can estimate the fog's density and color.

2.4 Current Status and Future Potential

Currently, we have successfully implemented and tested this solution on static images of traffic scenes, with promising results. The algorithm effectively restores clarity to foggy traffic photos.

The underlying principles of this approach are not limited to single images. Because video is simply a sequence of static images (frames), this algorithm can be extended and optimized to process video streams. This paves the way for a future implementation in real-time video dehazing, which could be a game-changing safety feature.

Fig 2.0: Concept of Dehazing the Foggy Traffic

Chapter 3: Methodology and Implementation

This section details the complete process, from setting up the development environment to the explanation of the Python code used for dehazing.

3.1 Environment Setup (Visual Studio Code)

We used Visual Studio Code as our primary code editor. The initial project setup was as follows:

1. **Open VS Code:** Launch the application.
2. **Create Project Folder:** A new folder named `dehazing` was created on the local drive.
3. **Open Folder:** In VS Code, we used **File > Open Folder...** to open the `dehazing` folder as our workspace.
4. **Create Python File:** A new file was created within the workspace named `dehaze.py`.
5. **Open Terminal:** The integrated terminal was opened using **View > New Terminal**. This opens a PowerShell terminal by default on Windows.

3.2 Virtual Environment and Dependencies

To maintain a clean and isolated project environment, we used Python's built-in `venv` module. All commands were run in the PowerShell terminal:

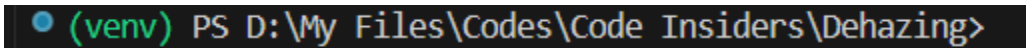
1. **Create Virtual Environment:**

```
# Creates a new folder named 'venv' in our project directory
python -m venv venv
```

2. **Activate Virtual Environment:**

```
# This command must be run to 'enter' the virtual environment
.\venv\Scripts\Activate
```

After activation, the terminal prompt changes to show **(venv)**.



3. **Install OpenCV and NumPy:**

```
# Installs the necessary libraries for image processing and numerical operations
pip install opencv-python numpy
```

4. **Install Matplotlib:**

```
# Installs the library for plotting and displaying images
pip install matplotlib
```

Note: `sys` and `tkinter` are part of the Python standard library and do not require separate installation.

3.3 Python Code and Explanation

The following code was written in the `dehaze.py` file. We will explain it block by block.

Block 1: Core Imports

```
import cv2
import numpy as np
import sys
import tkinter as tk
from tkinter import filedialog
```

Explanation:

- `cv2`: This imports the **OpenCV** library, our primary tool for all image processing tasks (reading, writing, splitting channels, filtering).
- `numpy as np`: This imports the **NumPy** library, which is essential for efficient numerical operations. We use it to create and manipulate the arrays of pixels that make up our images.
- `sys`: Imports the **System** library. It's often included for system-level operations, though not heavily used in this specific script.
- `tkinter as tk` and `from tkinter import filedialog`: This imports the **Tkinter** library. We specifically use its `filedialog` module to create the pop-up window that allows the user to browse their computer and select an image file.

Block 2: Matplotlib Import

```
6    import matplotlib.pyplot as plt
```

Explanation: This line imports the `pyplot` module from the **Matplotlib** library. We use this exclusively at the end of the script to create a plot that displays the original foggy image and the final dehazed image side-by-side for easy comparison.

Block 3: `get_dark_channel` Function

```
8   def get_dark_channel(image, window_size):
9       b, g, r = cv2.split(image)
10      min_intensity = np.minimum(np.minimum(b, g), r)
11      kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (window_size, window_size))
12      dark_channel = cv2.erode(min_intensity, kernel)
13      return dark_channel
```

Explanation: This function implements the "Dark Channel Prior" theory.

1. It splits the image into its three color channels: Blue, Green, and Red.
2. It finds the *minimum* intensity value for each pixel across all three channels (e.g., if a pixel is `[R=100, G=50, B=200]`, its `min_intensity` is 50).
3. It creates a `kernel` (a square of `window_size` x `window_size`).
4. It performs an `erode` operation. This slides the `kernel` over the `min_intensity` image and, for each position, finds the darkest pixel within that window.
5. The result, `dark_channel`, is a grayscale image that represents the raw haze density.

Block 4: `get_atmospheric_light` Function

```
15  def get_atmospheric_light(image, dark_channel, percentile=0.001):
16      num_pixels = dark_channel.size
17      num_brightest = int(max(1, num_pixels * percentile))
18
19      flat_dark_channel = dark_channel.flatten()
20      flat_image = image.reshape(num_pixels, 3)
21
22      indices = np.argsort(flat_dark_channel)[-num_brightest:]
23      brightest_pixels = flat_image[indices]
24
25      atmospheric_light = np.mean(brightest_pixels, axis=0)
26      return atmospheric_light.astype(np.float64)
```

Explanation: This function estimates the color of the fog (known as "atmospheric light", `A`).

1. It identifies the top `0.1%` (or `percentile`) brightest pixels in the `dark_channel`. These pixels are assumed to be the most fog-occluded.
2. It gets the BGR (Blue, Green, Red) values of *these specific pixels* from the *original* foggy image.
3. It averages these BGR values to get a single color vector (e.g., `[B=210, G=220, R=225]`). This vector is our `atmospheric_light`, `A`.

Block 5: `guided_filter` Function

```
28  def guided_filter(guidance_image, input_image, radius, epsilon):
29      mean_I = cv2.boxFilter(guidance_image, -1, (radius, radius))
30      mean_p = cv2.boxFilter(input_image, -1, (radius, radius))
31
32      mean_II = cv2.boxFilter(guidance_image * guidance_image, -1, (radius, radius))
33      mean_Ip = cv2.boxFilter(guidance_image * input_image, -1, (radius, radius))
34
35      cov_Ip = mean_Ip - mean_I * mean_p
36      var_I = mean_II - mean_I * mean_I
37
38      a = cov_Ip / (var_I + epsilon)
39      b = mean_p - a * mean_I
40
41      mean_a = cv2.boxFilter(a, -1, (radius, radius))
42      mean_b = cv2.boxFilter(b, -1, (radius, radius))
43
44      output = mean_a * guidance_image + mean_b
45      return output
```

Explanation: This is an advanced edge-preserving filter. Our initial haze map is blocky; if we use it directly, we get "halo" artifacts around objects.

- This filter takes the blocky `input_image` (our transmission map) and smooths it.
- Crucially, it uses the `guidance_image` (the original image) to know where edges are.
- It smooths the map *except* at object edges, resulting in a clean transmission map that respects the boundaries of objects in the scene.

Block 6: `get_transmission_map` Function

```python
def get_transmission_map(image, atmospheric_light, window_size, omega=0.95,
                         guided_filter_radius=60, guided_filter_epsilon=0.0001):

    normalized_image = np.empty_like(image, dtype=np.float64)
    for i in range(3):
        normalized_image[:, :, i] = image[:, :, i] / atmospheric_light[i]

    transmission_estimate = get_dark_channel(normalized_image, window_size)
    transmission_map = 1 - (omega * transmission_estimate)

    image_normalized_for_guidance = (image / 255.0).astype(np.float32)
    image_gray_for_guidance = cv2.cvtColor(image_normalized_for_guidance, cv2.COLOR_BGR2GRAY)

    transmission_map_float32 = transmission_map.astype(np.float32)

    refined_transmission_map = guided_filter(image_gray_for_guidance,
                                             transmission_map_float32,
                                             guided_filter_radius,
                                             guided_filter_epsilon)

    return refined_transmission_map.astype(np.float64)
```

Explanation: This function calculates the `transmission_map`, `t(x)`, which represents how much light (from 0.0 to 1.0) gets through the fog at each pixel.

1. It first gets a *rough estimate* of the transmission using the `get_dark_channel` function on an image normalized by the atmospheric light. The `omega` parameter keeps a tiny bit of haze for realism.
2. It converts the original image to grayscale to be used as the "guide".
3. It calls the `guided_filter` to refine the rough map, producing the final `refined_transmission_map`.

Block 7: `dehaze_image` Function

```python
def dehaze_image(image, atmospheric_light, transmission_map, t0=0.1):
    clamped_transmission = np.maximum(transmission_map, t0)

    clamped_transmission_3d = cv2.merge([clamped_transmission, clamped_transmission, clamped_transmission])

    dehazed_image = np.empty_like(image, dtype=np.float64)

    for i in range(3):
        dehazed_image[:, :, i] = ((image[:, :, i] - atmospheric_light[i]) /
                                  clamped_transmission_3d[:, :, i]) + atmospheric_light[i]

    dehazed_image = np.clip(dehazed_image, 0, 255)
    return dehazed_image.astype(np.uint8)
```

Explanation: This is the final step where the magic happens. It uses the atmospheric scattering model formula, solved for the clear image `J`.

1. `I` = **Foggy Image** (the input `image`)
2. `A` = **Atmospheric Light** (from `get_atmospheric_light`)
3. `t` = **Transmission Map** (from `get_transmission_map`)
4. The formula is: `J = (I - A) / t + A`
5. `t0` is a minimum value for `t` to prevent division by zero in very foggy areas.
6. The formula is applied to all three color channels (B, G, R).
7. The final result is `clipped` to the valid 0-255 color range and converted back to a standard 8-bit image (`uint8`).

Block 8: `main` Function

```
83  def main():
84      print("Opening file dialog to select an image...")
85      root = tk.Tk()
86      root.withdraw()
87
88      input_image_path = filedialog.askopenfilename(
89          title="Select a Hazy Image",
90          filetypes=[("Image Files", "*.jpg *.jpeg *.png *.bmp"), ("All Files", "*.*")]
91      )
92
93      if not input_image_path:
94          print("No image selected. Exiting.")
95          return
96
97      output_image_path = 'dehazed_image.jpg'
98      window_size = 15
99
100     # --- TWEAKABLE VALUES ---
101     omega = 0.95
102     t0 = 0.3
103     percentile = 0.001
104     guided_filter_radius = 60
105     guided_filter_epsilon = 0.0001
106     # ----------------------
107
108     print(f"Loading image: {input_image_path}")
109     image = cv2.imread(input_image_path)
```

```
    if image is None:
        print(f"Error: Could not load image from {input_image_path}")
        print("Please make sure the file is a valid image.")
        return

    image_float = image.astype(np.float64)

    print("Calculating dark channel...")
    dark_channel = get_dark_channel(image_float, window_size)

    print("Estimating atmospheric light...")
    a_light = get_atmospheric_light(image_float, dark_channel, percentile)
    print(f"  Estimated atmospheric light (BGR): {a_light}")

    print("Estimating transmission map...")
    transmission = get_transmission_map(image_float / 255.0, a_light / 255.0, window_size, omega,
                                        guided_filter_radius, guided_filter_epsilon)

    print("Dehazing image...")
    dehazed_image = dehaze_image(image_float, a_light, transmission, t0)

    print(f"Saving dehazed image to: {output_image_path}")
    cv2.imwrite(output_image_path, dehazed_image)

    print("Displaying images...")

    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    dehazed_image_rgb = cv2.cvtColor(dehazed_image, cv2.COLOR_BGR2RGB)

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
```

```
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    ax1.imshow(image_rgb)
    ax1.set_title('Original')
    ax1.axis('off')

    ax2.imshow(dehazed_image_rgb)
    ax2.set_title('Dehazed')
    ax2.axis('off')

    plt.tight_layout()
    plt.show()

    print("Done.")

if __name__ == '__main__':
    main()
```

- **Explanation:** This is the main function that runs the entire process from start to finish.
    1. It uses `tkinter` to ask the user to select an image file.
    2. It defines all the key parameters (like `window_size`, `omega`, `t0`) that control the algorithm's behavior.
    3. It loads the image using `cv2.imread`.
    4. It calls all our functions in the correct order: `get_dark_channel`, `get_atmospheric_light`, `get_transmission_map`, and `dehaze_image`.
    5. It saves the final `dehazed_image` to the disk.
    6. Finally, it uses `matplotlib` to convert the images from BGR to RGB (for correct color display) and shows the "before" and "after" images side-by-side in a new window.

3.4 Core Libraries and Tools

3.4.1 OpenCV (Open Source Computer Vision)

This is the single most important library for the project. OpenCV provides a highly optimized and comprehensive set of tools for all low-level image processing tasks. In this project, we use it for:

- `cv2.imread() / cv2.imwrite()`: Reading and writing image files.
- `cv2.split() / cv2.merge()`: Handling image color channels.
- `cv2.erode()`: The core of the `get_dark_channel` function.
- `cv2.boxFilter()`: Used inside the `guided_filter`.
- `cv2.cvtColor()`: Converting images between BGR and RGB for display.

3.4.2 NumPy

NumPy is the fundamental package for scientific computing in Python. It provides the `ndarray` object, a powerful N-dimensional array that is the foundation of our entire application. All images are stored as NumPy arrays, and all mathematical operations (division, clipping, means) are performed using fast, vectorized NumPy functions.

3.4.3 Matplotlib

While OpenCV *can* display images (`cv2.imshow`), its UI is basic. Matplotlib is a powerful plotting library that allows us to create the clean, side-by-side "Original" vs. "Dehazed" plot with titles and no axes, just as seen in the user's requirement.

3.4.4 Tkinter

Tkinter is Python's standard built-in GUI library. It is used for one simple but crucial task: calling `filedialog.askopenfilename()` to show a native "Open" file dialog. This makes the script far more user-friendly than hard-coding a filename.

Chapter 4: Output and Results

4.1 Purpose of Visual Verification

This chapter is reserved for the visual results of the project. While the code executes and produces an output, this section provides qualitative proof of its effectiveness. For a complete report, this section would contain several pairs of images demonstrating the algorithm's performance in various conditions.

4.2 Analysis of Results

When evaluating the results, we look for several key indicators of success:

- **Clarity and Sharpness:** Are details that were hidden now visible? Are the edges of signs and signals sharp?
- **Color Restoration:** Has the washed-out grey been replaced by vibrant, natural colors (e.g., a "red" stop sign, not a "pink" one)?
- **Artifact Reduction:** Are there any "halos" or blocky patches? The use of the Guided Filter is intended to minimize these.

4.3 Sample Results Table

Original Hazy Image	Dehazed Image
	
Fig 4.3.1.1: Hazy traffic scene with low visibility	**Fig 4.3.1.2:** Dehazed output showing clear signals.

Original Hazy Image	Dehazed Image
	
Fig 4.3.3.1: Foggy road with obscured traffic signs.	**Fig 4.3.3.2:** Clear signs and improved road visibility.
	
Fig 4.3.4.1: Dense fog obscuring highway traffic.	**Fig 4.3.4.2:** Clear view of vehicles and markings.
	
Fig 4.3.5.1: Fog hiding speed and weather signs.	**Fig 4.3.5.2:** Clear legible text and icons.

Original Hazy Image	Dehazed Image
	
Fig 4.3.6.1: Heavy fog hiding sign and car.	**Fig 4.3.6.2:** Sharp sign and visible vehicle details.

Chapter 5: Conclusion

5.1 Project Efficacy and Summary

The model implemented in this project, based on the Dark Channel Prior with Guided Filter refinement, is demonstrably effective in dehazing foggy images. The application of this algorithm to traffic-related images successfully enhances the visibility of critical infrastructure. By algorithmically removing the atmospheric veil, we restore color, improve contrast, and make crucial visual cues like traffic signals and signboards easily discernible. This provides a clear and significant aid to a driver, directly addressing the safety problem outlined in Chapter 1.

5.2 Future Work: Real-Time Implementation

This methodology is not limited to static images. The most logical and impactful next step is to adapt this for real-time video processing.

- **The Challenge:** The primary challenge will be optimization. Running this complex algorithm on every single frame of a 30-FPS video stream is computationally expensive.
- **The Solution:** This can be addressed through:
    1. **Algorithmic Optimization:** Simplifying calculations or estimating the atmospheric light less frequently (e.g., once per second instead of once per frame).
    2. **Hardware Acceleration:** Offloading calculations to a dedicated GPU.
    3. **Frame-to-Frame Coherence:** Using the transmission map from the previous frame as an initial guess for the current frame.

This opens the possibility for implementation in-car dashcam systems or advanced driver-assistance systems (ADAS), providing a clear, dehazed video feed to the driver.

5.3 Broader Impact and Applications

Beyond just driver-facing dashcams, this technology has other critical applications in traffic safety:

- **Autonomous Vehicles:** Self-driving cars rely on computer vision to see. This algorithm can serve as a pre-processing step, "cleaning" the input for object detection models so they can identify pedestrians, signs, and other cars more reliably in fog.
- **Security and Surveillance:** Fixed traffic and security cameras often become useless in fog. This algorithm can be run on their feeds to improve monitoring.

Overall Report Summary

This project addresses the critical safety challenge of low visibility in traffic caused by adverse weather conditions like fog. Leveraging Python and Computer Vision techniques—specifically the **Dark Channel Prior (DCP)** algorithm—we developed a robust solution to algorithmically remove haze from images.

The development was conducted in **Visual Studio Code** using a secure virtual environment (`venv`), utilizing core libraries including **OpenCV** for image processing, **NumPy** for matrix calculations, and **Matplotlib** for visualization. Our methodology involves estimating the atmospheric light and transmission map to invert the physical scattering model of the fog, effectively recovering the underlying clear scene. To ensure high-quality output, a **Guided Filter** is applied to refine the transmission map, preserving edges and preventing visual artifacts.

The successful application of this model on static traffic images demonstrates a significant improvement in the visibility of signals and signboards. This validates the algorithm's effectiveness and establishes a strong foundation for future scaling into real-time video dehazing for vehicle dashcams and autonomous driving systems.

Fig 5.0: Original Vs Dehazed Image using Open CV in Comparison View

References

1. Core Research Papers (The basis of your code)

These are the official links to the papers by Kaiming He, whose algorithms (`get_dark_channel`, `guided_filter`) you implemented.

- **Single Image Haze Removal Using Dark Channel Prior**
  - *Link:* IEEE Xplore - Single Image Haze Removal
  - *Author's PDF (Free access):* http://kaiminghe.com/publications/pami10dehaze.pdf
- **Guided Image Filtering**
  - *Link:* IEEE Xplore - Guided Image Filtering
  - *Author's PDF (Free access):* http://kaiminghe.com/publications/pami12guided.pdf

2. Traffic & Safety Applications

These links connect your specific use case (traffic signs/safety) to the technology.

- **Towards Fog-Free In-Vehicle Vision Systems** (Research on using this for cars)
  - *Link:* IEEE Xplore - Fog-Free In-Vehicle Vision
- **Vision Enhancement in Poor Visibility Conditions**
  - *Link:* ResearchGate - Vision Enhancement Review

3. Tools & Documentation

References for the software tools used in your `venv`.

- **OpenCV (Open Source Computer Vision Library)**
  - *Link:* https://opencv.org/
  - *Documentation:* https://docs.opencv.org/master/
- **NumPy (Fundamental Package for Scientific Computing)**
  - *Link:* https://numpy.org/

4. Similar Open Source Projects (GitHub)

Standard industry practice, popular repositories that use the same Python + OpenCV approach:

- **A Python Implementation of Dark Channel Prior (GitHub)**
  - *Link:* https://github.com/He-Zhang/image_dehaze
- **Single Image Haze Removal (GitHub)**
  - *Link:* https://github.com/joyeecheung/haze-removal