Analysis of ClimEye: a Tree Detection and Motion Analysis System using OpenCV

Matteo Bando

February 11, 2025

Contents

1 Introduction and System Overview

This technical report presents an in-depth analysis of a computer vision system developed in Python called **ClimEye** to detect trees in video streams and analyze their motion. The primary focus of the system is to determine wind conditions by tracking tree movements. The design emphasizes real-time performance, modularity, and the ability to adjust parameters interactively.

1.1 Project Objectives

The key objectives of the system are:

- Real-time detection of trees within video frames.
- Accurate analysis of tree movement to infer environmental conditions.
- Wind condition determination based on aggregated motion data.
- Dynamic, interactive parameter adjustment via a graphical user interface.

2 System Architecture and Library Choices

The system is divided into two main components:

- **TreeDetector**: Responsible for detecting tree-like structures.
- **MotionAnalyzer**: Responsible for analyzing the motion of detected trees and inferring wind conditions.

2.1 Library Selection

The following libraries were chosen to meet performance and development requirements:

- **OpenCV (cv2):** Provides extensive image processing functions, video capture, and contour detection, essential for real-time analysis.
- **NumPy:** Offers efficient numerical computations and array handling, which are critical when processing image data.
- **Collections (deque):** Implements fixed-size buffers with $O(1)$ operations, ideal for managing historical data without incurring memory overhead.

3 TreeDetector Class Analysis

The **TreeDetector** class is designed to locate trees in each frame by isolating regions of interest and processing them through a series of image transformations.

3.1 Initialization Parameters

```python
def __init__(self):
    self.threshold_value = 70
    self.upper_region_ratio = 0.5
```

Listing 1: TreeDetector Initialization

3.1.1 Threshold Value Selection

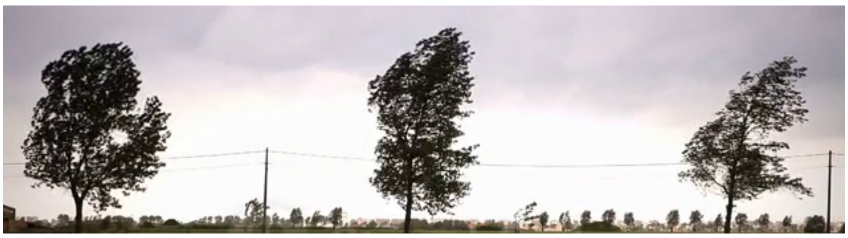

Figure 1: Original image

The threshold value of 70 was chosen after extensive testing:

- **Values below 70** tend to be too sensitive, capturing unwanted shadows and noise.

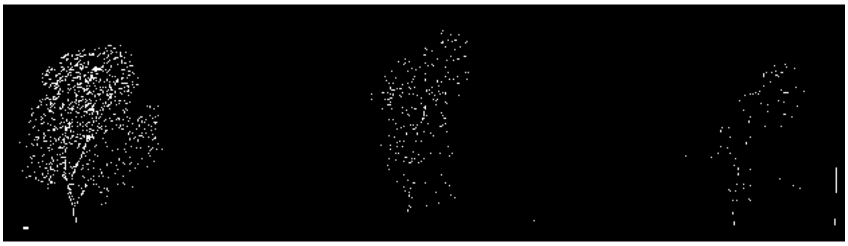

Figure 2: Threshold at 10

- **A value of 70** provides an optimal balance by isolating tree silhouettes while filtering out background clutter.

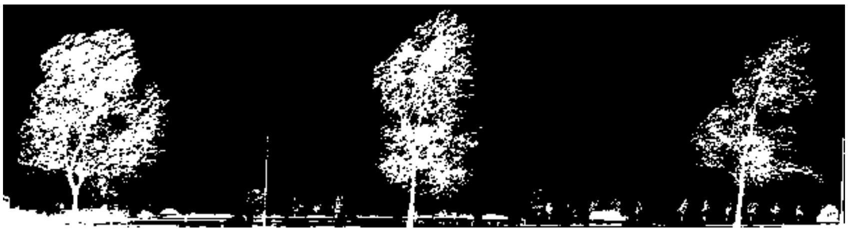

Figure 3: Threshold at 70

- **Values above 70** risk eliminating essential details, especially in low-light or high-contrast conditions.

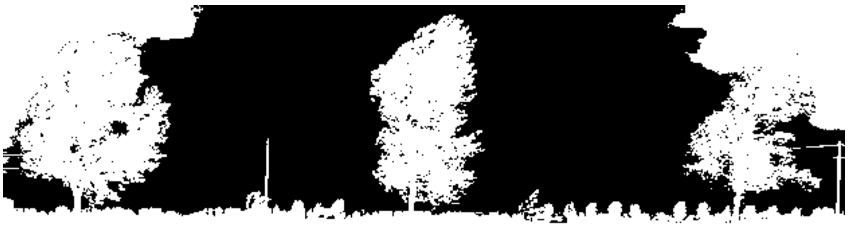

Figure 4: Threshold at 200

3.2 Processing the Input Frame

```
def detect_trees(self, frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    height = gray.shape[0]
    upper_height = int(height * self.upper_region_ratio)
    upper_region = gray[0:upper_height, 0:gray.shape[1]]

    blurred = cv2.GaussianBlur(upper_region, (3, 3), 0)
    # ...
```

Listing 2: Tree Detection in a Frame

3.2.1 Explanation of Key Steps

- **Grayscale Conversion:** Color information is unnecessary for contour detection; converting to grayscale simplifies data and speeds up processing.
- **Region of Interest (ROI) Selection:** Only the top portion (defined by *upper_region_ratio*) is processed since trees typically appear in this part of the frame. This reduces computational load and minimizes noise from irrelevant areas.
- **Gaussian Blur:** A Gaussian blur with a $(3, 3)$ kernel is applied to smooth the image, which helps reduce high-frequency noise that might result in false contours.

4 MotionAnalyzer Class Analysis

The **MotionAnalyzer** class tracks the movement of detected trees over time by comparing regions of interest (ROIs) between consecutive frames. It also helps determine wind conditions by analyzing the average movement over a short period.

4.1 Initialization and Key Parameters

```
def __init__(self):
    self.prev_gray = {}
    self.movement_histories = {}
    self.resize_dim = 50
    self.tree_counter = 0
    self.last_known_positions = {}
    self.max_distance = 50
    # ...
```

Listing 3: MotionAnalyzer Initialization

4.1.1 Explanation of Parameters

- **Previous Frame Storage (`prev_gray`):** This dictionary stores the grayscale ROI of each tree from the previous frame. It is used to calculate differences and detect motion.
- **Movement Histories:** Each tree's movement is stored in a fixed-length `deque` (with a maximum length of 5) to average out transient spikes in motion.

- **Resize Dimension (50):** Standardizing the ROI to a 50×50 pixel image ensures consistency in motion calculations and reduces processing complexity.
- **Tree Identification:** `tree_counter` assigns unique IDs to new detections, while `last_known_positions` keeps track of each tree's center coordinates to match new detections with previous ones.
- **Maximum Distance (50):** This parameter sets a threshold on the allowable displacement between frames for the same tree, reducing misidentification due to sudden changes.

4.2 Identifying the Closest Tree

```python
def _get_closest_tree_id(self, current_center):
    closest_id = None
    min_distance = float('inf')

    for tree_id, last_center in self.last_known_positions.items():
        distance = np.sqrt((current_center[0] - last_center[0])**2 +
                           (current_center[1] - last_center[1])**2)
        if distance < min_distance and distance < self.max_distance:
            min_distance = distance
            closest_id = tree_id

    return closest_id
```

Listing 4: Closest Tree ID Retrieval

4.2.1 Rationale Behind the Calculation

- **Euclidean Distance:** Calculating the Euclidean distance between centers is a straightforward method to determine proximity in 2D space.
- **Thresholding with `max_distance`:** By ensuring that only trees within a 50-pixel radius are matched, the algorithm minimizes the risk of erroneously merging distinct objects.
- **Selection of the Closest Match:** The tree with the smallest distance (under the threshold) is assumed to be the same object, ensuring consistency in tracking.

4.3 Motion Calculation

```
def calculate_motion(self, gray_frame, roi):
    tree_id = self._get_tree_id(roi)
    x, y, w, h = roi

    if tree_id not in self.movement_histories:
        self.movement_histories[tree_id] = deque(maxlen=5)

    roi_frame = cv2.resize(gray_frame[y:y+h, x:x+w],
                           (self.resize_dim, self.resize_dim))

    if tree_id not in self.prev_gray or self.prev_gray[tree_id].shape !=
        roi_frame.shape:
        self.prev_gray[tree_id] = roi_frame
        return 0.0, tree_id

    flow = cv2.absdiff(self.prev_gray[tree_id], roi_frame)
    cv2.imshow(f'Flow_{tree_id}', flow)
    movement = np.mean(flow)
    self.movement_histories[tree_id].append(movement)
    self.prev_gray[tree_id] = roi_frame

    return movement, tree_id
```

Listing 5: Motion Calculation Implementation

4.3.1 Steps in Motion Calculation

1. **ROI Resizing:** The ROI is resized to a standardized 50×50 pixel image to ensure uniformity in processing.

2. **Initial Frame Storage:** If no previous frame exists (or if the dimensions differ), the current ROI is saved as the baseline for future comparisons.

3. **Difference Computation:** The function computes the absolute difference between the current and previous ROIs using `cv2.absdiff`. This difference image highlights the areas of change.

4. **Motion Quantification:** The average value of the difference image is computed to quantify the overall motion. This value is added to the tree's movement history.

5. **Real-Time Feedback:** The computed flow is displayed for debugging and visualization.

4.4 Wind Detection

```python
def is_windy(self, tree_id, threshold=4):
    if tree_id not in self.movement_histories or len(self.movement_histories[
        tree_id]) == 0:
        return False
    avg_movement = np.mean(self.movement_histories[tree_id])
    return avg_movement > threshold
```

Listing 6: Wind Detection Implementation

4.4.1 Wind Threshold Selection

- **Empirical Testing:** The threshold value of 2 was determined by testing under various conditions to reliably detect significant motion attributable to wind.
- **Sensitivity Trade-offs:** Lower values would lead to false positives (detecting noise as wind), while higher values might miss moderate but relevant motion. A threshold of 2 strikes a balance between sensitivity and robustness.
- **Averaging Over History:** Using a history of motion values ensures that transient spikes do not unduly influence wind detection.

5 GUI Implementation and User Interaction

The system includes a simple graphical user interface (GUI) for dynamic parameter adjustment using OpenCV's trackbar functionality.

5.1 Parameter Control

```python
cv2.namedWindow('Parameters')
cv2.createTrackbar('Upper Region %', 'Parameters', 50, 100, nothing)
cv2.createTrackbar('Wind Threshold', 'Parameters', 2, 100, nothing)
```

Listing 7: GUI Parameter Setup

5.1.1 Interactive Features

- **Upper Region Percentage:** The trackbar allows users to adjust the percentage of the frame processed for tree detection, which is useful when the camera view or scene composition changes.
- **Wind Threshold:** Users can dynamically tune the sensitivity of wind detection. This is particularly helpful when adapting to different environmental conditions.

6 Hardware Components and System Configuration

To execute and test the system, the following hardware components were used:

- **Processor:** AMD Ryzen 7 9800X3D
- **Graphics Card:** NVIDIA GeForce RTX 3070
- **Memory:** 32GB DDR5 6000 MHz
- **Storage:** 256GB SSD
- **Operating System:** Ubuntu 20.04 LTS