Optical Sensing of Urban Landscape and Traffic Congestion

- Ratnesh Yadav

(PSU ID- 962052529)

Abstract:

This work introduces a method combining Adaptive Background Subtraction and Optical Flow to enhance traffic congestion detection and land cover change monitoring using vehicular-mounted cameras. Our approach aims to refine data accuracy and processing efficiency. The algorithm merges these techniques to provide real-time, detailed analysis. The findings indicate a marked improvement in identifying and tracking dynamic objects, offering significant implications for urban planning and environmental management.

1. Introduction:

Urban landscapes are dynamic, ever-evolving ecosystems that pose significant challenges for monitoring and understanding their changes over time. Rapid urbanization, coupled with technological advancements, necessitates innovative approaches to keep pace with the transforming cityscapes. This project presents one such approach, leveraging the power of computer vision and remote sensing to monitor urban environments.

We focus on two key facets of urban change: vehicle movement and land cover alterations. These elements are not only indicative of a city's day-to-day activities but also reflect broader trends in urban development and planning. Tracking these changes can provide valuable insights for urban planners, policymakers, and researchers alike. To achieve this, we employ three specific computer vision techniques: Adaptive Background Subtraction, Optical Flow, and Foreground Masking. These methodologies, when combined, offer a robust framework for detecting and tracking changes in urban landscapes.

Through this project, we aim to demonstrate the potential of our system as a powerful tool for urban monitoring. The results, as we will discuss, show promising precision in detecting changes over time, underscoring the value of such an approach in the field of remote sensing.

2. Literature Review and Theory

2.1 The Landscape of Background Subtraction Techniques: Insights from Piccardi's Review

Adaptive Background Subtraction is a technique that allows for robust detection of changes in a scene. In dynamic scenes, the same pixel location often captures various background objects over time, which can be a result of permanent or temporary changes in the scene[1]. Traditional single-valued background models fail to adequately represent these changes, particularly for frequently altering scenarios such as tree movements, weather changes, or sea waves. Stauffer and Grimson proposed a multi-valued background model that uses a mixture of Gaussians to describe the probability of observing a certain pixel value, effectively capturing both the foreground and the background values.

$$P(x_t) = \sum_{i=1}^{k} \omega_{i,t}\eta(x_t - \mu_{i,t}, \varepsilon_{i,t})$$

We are using this method because it provides a robust solution to cope with the dynamic nature of the scene by accurately distinguishing between foreground and background objects, and it allows for efficient parameter updating using an expectation-maximization algorithm.

2.2 Dynamic Background Subtraction

ViBe algorithm is a novel background subtraction method that uniquely combines a minimalistic classification model, rapid single-frame initialization, and an innovative update mechanism that randomly replaces pixel samples without considering their insertion time[2]. This approach not only accelerates the adaptation to varying background dynamics but also ensures spatial consistency by allowing sample diffusion among neighboring pixels, thus enhancing the algorithm's robustness to camera movement and obviating the need for post-segmentation processing.

It is a three-step process. Step one, the Pixel Model and Classification Process aims to model each background pixel by a collection of N background sample values.

$$\mathrm{M}(x) = \{v_1, v_2, \ldots, v_n\}$$

Step Two of Background Model Initialisation makes use of a single initial frame for referencing. The third step involves updating the background model over time.

2.3. Optical Flow

Optical flow is the distribution of apparent velocities of movement of brightness patterns in an image. Optical flow can arise from the relative motion of objects and the viewer. Consequently, optical flow gives important information about the spatial arrangement of the objects viewed and the rate of change of this arrangement. Discontinuities in the optical flow help in segmenting images into regions that correspond to different objects[3].

2.4 Background modeling using a mixture of Gaussians for foreground detection

The survey on background modeling using a Mixture of Gaussians (MoG) for foreground detection serves as a crucial reference point. Our project aims to enhance traffic and environmental monitoring through advanced data analysis, and understanding the MoG approach is vital for developing our background subtraction techniques. The survey on background modeling using a Mixture of Gaussians (MoG) for foreground detection comprehensively collates advancements in the domain, showcasing the MoG method as a dynamic solution for separating foreground elements in video streams[4]. Central to this is the modeling of pixel values through multiple Gaussian distributions, allowing a versatile representation of scenes and facilitating the distinction between static backgrounds and moving objects.

Another key takeaway is the approaches suggested for the reduction in processing time. Region of interest and Hardware computing are the two approaches with which we have tried to reduce the computational time of our project.

3. Proposed Approach:

The above-proposed techniques individually help improve the efficiency of optical sensing. However, there is not much development in their combined application in traffic congestion and dynamic landscape sensing. As such in addressing the limitations identified in the literature, our proposed approach seeks to implement a system that leverages the strengths of both Adaptive Background Subtraction and Optical Flow methods. Adaptive Background Subtraction will be utilized to dynamically distinguish between the moving and static elements within urban and natural environments, effectively handling varying lighting and weather conditions. Optical Flow will then be employed to track the motion of distinct features in the sequence of images captured by vehicular-mounted cameras.

The integration of these two methods aims to create a more robust and adaptive tool for real-time analysis. We will develop an algorithm that effectively marries the two techniques, ensuring that changes in the visual field are captured with high precision. We anticipate that the resulting system will help identify traffic congestion and land cover changes.

4. Methodology:

The overall design of the project aims to analyze video data to detect vehicles and assess land cover changes over time. The project is part of the study on traffic analysis, urban planning, or environmental monitoring, where such information is critical.

The design involves the following steps:

- **Data Collection**: Collecting camera data from a vehicle-mounted camera. For our project, we used the vehicle-mounted camera data of a vehicle moving across Italy from the National Library of Medicine.
- **Data Processing**: Using computer vision techniques to detect vehicles and analyze land cover changes within the video frames. This step involves image processing, segmentation, and storage.
- **Data Analysis**: Quantifying the detected vehicles and land cover changes to draw insights.
- **Visualization and Reporting**: Presenting the findings in a visual format such as graphs, heatmaps, and geospatial maps for better understanding and reporting.

The methods and techniques used in the code align with the research design and include:

- **Haar Cascade Classifier for Vehicle Detection**: A machine learning object detection method capable of identifying objects in an image or video, in this case, vehicles. It's chosen for its efficiency in real-time detection. The classifier is stored in the XML format. For the project, we used the Haar Cascade Classifier made open-source by Andrews Sobral, and optimised the code to just detect the vehicles present in the driver's lane.
- **Canny Edge Detection**: An edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It is used here for segmenting images, which is a precursor to more detailed analysis.
- **Background Subtraction**: A technique that separates foreground objects (like moving vehicles) from the background. This is particularly useful for keeping track of vehicles.
- **Optical Flow**: This method estimates the motion of objects between two consecutive frames. It's crucial for understanding the dynamics within the video, such as the direction and speed of moving vehicles.
- **Foreground Masking**: This technique is used to isolate and analyze changes in the video's environment, enabling the calculation of land cover change ratios.

<u>Analysis Procedures</u>:

- Frames are extracted from the video and processed using the methods listed above.
- Vehicle detection is performed on each frame.
- Land cover changes are assessed by comparing frames over time and calculating the ratio of changes.
- The analysis results are stored and then visualized using plots and maps

<u>Code and Functionality Detail</u>:

- **Initialization Function**: The init function is used to initialize the car_cascade with a pre-trained Haar cascade file, which is necessary for car detection in images.
- **Image Processing Functions**: These functions handle different aspects of image processing. process_image converts an image to grayscale and resizes it. segment_image applies edge detection using the Canny algorithm. detect_cars uses the Haar cascade classifier to detect cars in a region of interest (ROI) or the full frame.
- **Background Subtraction and Optical Flow Functions**: apply_background_subtraction calculates the absolute difference between the current and previous frames and applies a running average to create a background model. apply_optical_flow computes the optical flow using the Farneback method, which estimates the motion between two images. draw_flow visualizes this motion on the image.
- **Overlay and Land Cover Change Functions:** The overlay_images function combines two images with specified weights to create an overlay. calculate_land_cover_change_ratio computes the ratio of non-black pixels to the total number of pixels in the foreground mask, which indicates changes in the scene.

- **Main Function**: The main function is the entry point of the script. It opens the data file, reads frames, and applies the defined functions to process the images. It uses concurrent.futures.ProcessPoolExecutor for parallel processing of tasks such as optical flow computation and car detection. The function concurrent.futures.ProcessPoolExecutor is responsible for the computing speed of the code. The system CPU used during the project had a total of 4(low) cores available for multitasking. As such four tasks were designed to run parallelly to reduce the computing speed.
- **Visualization and Output**: This section includes code that generates plots for land cover change ratios, vehicle counts over time, and a heatmap for full-frame vehicle counts. It also contains commented-out code for creating a map visualization using Folium.
- **Cleanup**: Release the video capture resource and close the output file. It also destroys all the OpenCV windows.

Demonstrated below is the flowchart of our current code for ease of understanding. The Python script used for the project will be added at the end in the Appendix.

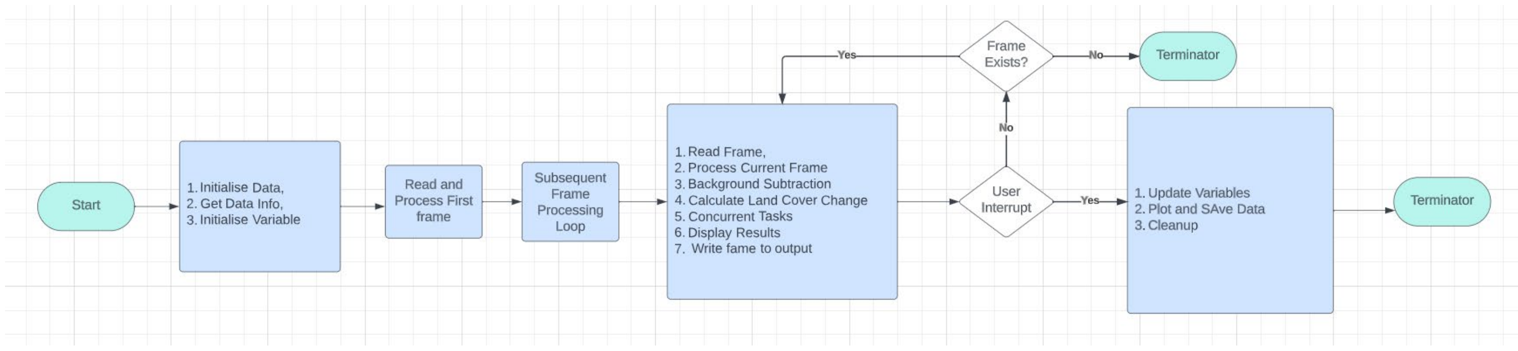

Fig. 1: Flowchart of the code implementation

5. Results:

The entire project has been run and tested on a device that has 4 cores of CPU. Increasing the number of cores would result in better processing performance. The current ratio of processing time to video length is around 1.6.

```
Video length: 3338.84 seconds
Elapsed time: 5275.878782272339 seconds
processing factor: 1.580153221559685
```

The processing factor has been reduced from the initial factor of 3 to the current 1.6 by using the parallel processing function by fully utilizing the available CPU cores.

The data used for the project has been obtained from the National Library of Medicine. It is the vehicle-mounted camera feed of fifty-five minutes.

The results obtained are in two stages:

The first stage of outputs can be seen during the real-time processing of the data. One window shows the optical flow and the vehicles detected in a frame and the other shows the foreground masking data which is later used to calculate the land cover change ratio.

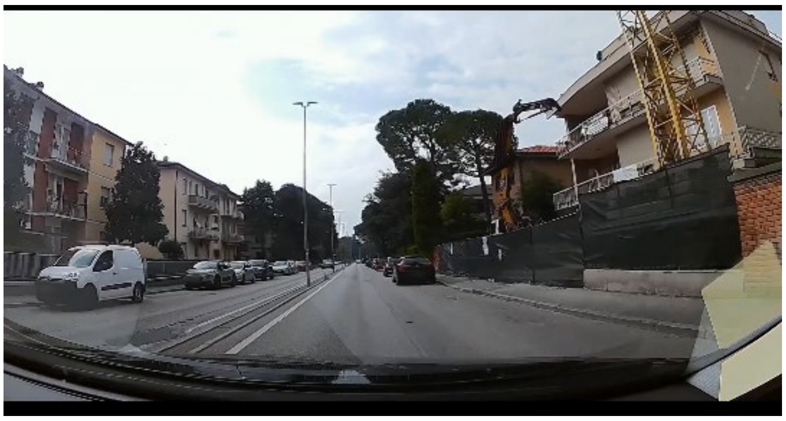
Fig. 2: RGB Data Frame

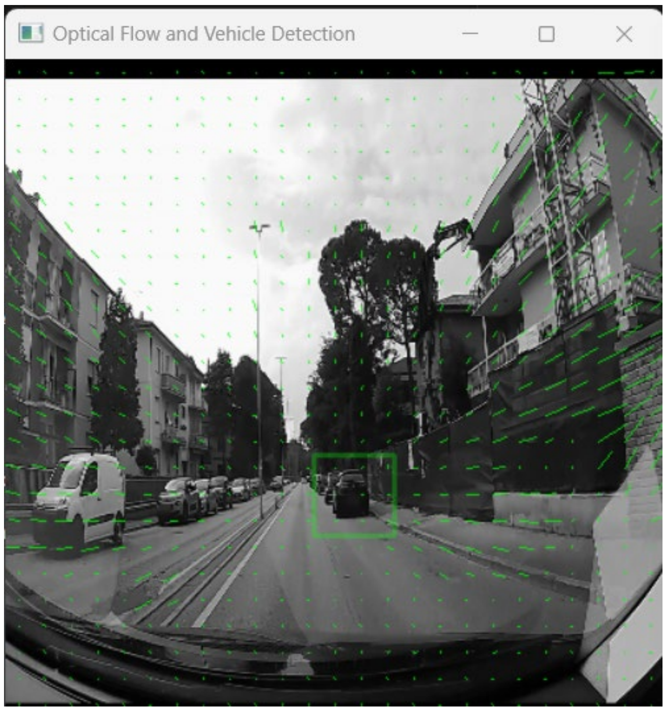

Fig. 3: Processed Frame with a vehicle detected(driver's lane) and the optical flow between two successive frames

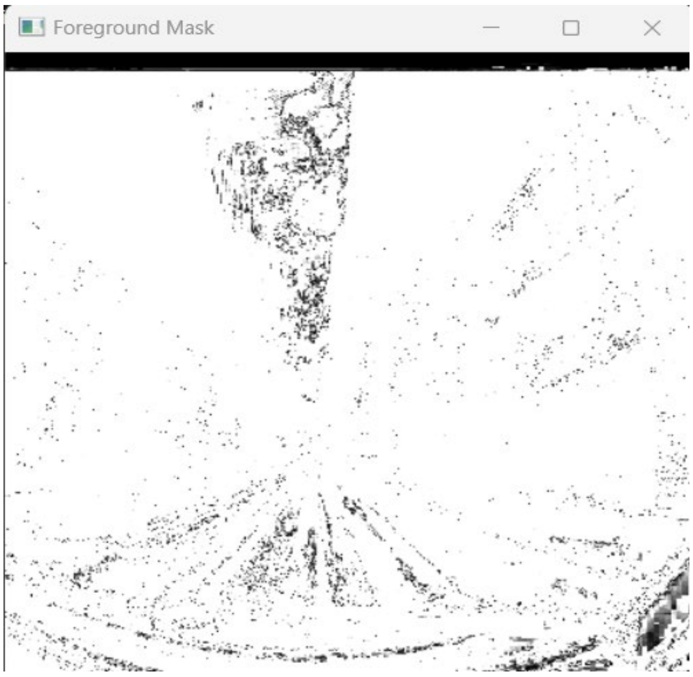

Fig. 4: Foreground masking

The second stage of outputs are multiple graphs saved as image in the output folder. It also contains another raw video of segmented frames that is used for processing during the execution. The graphs show the rate of change in landscape, vehicle count over time in a lane, and heatmap of vehicle density in the overall region of vehicle motion respectively. The land cover change ratio approaching 1 means the vehicle was in the state of continuous motion during the entire duration.

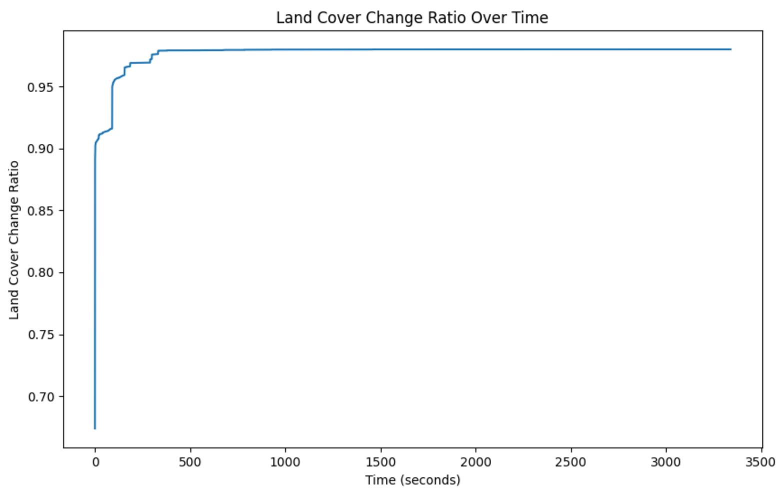

Fig. 5: Landcover Change Ratio

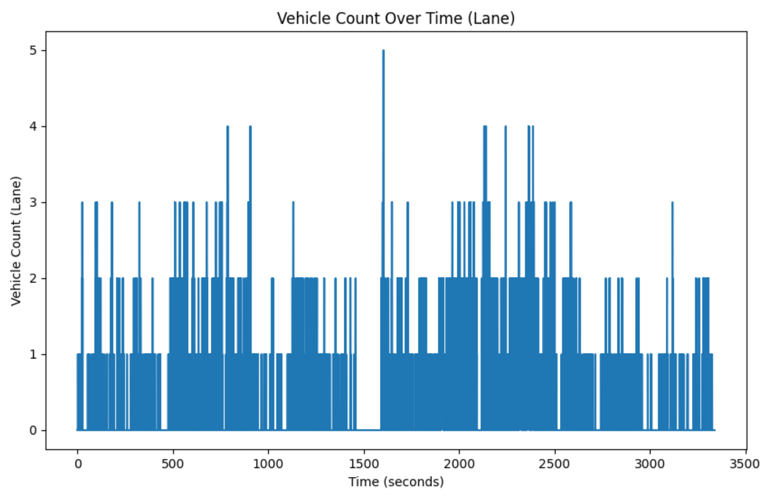

Fig. 6: Vehicle Count(Lane)

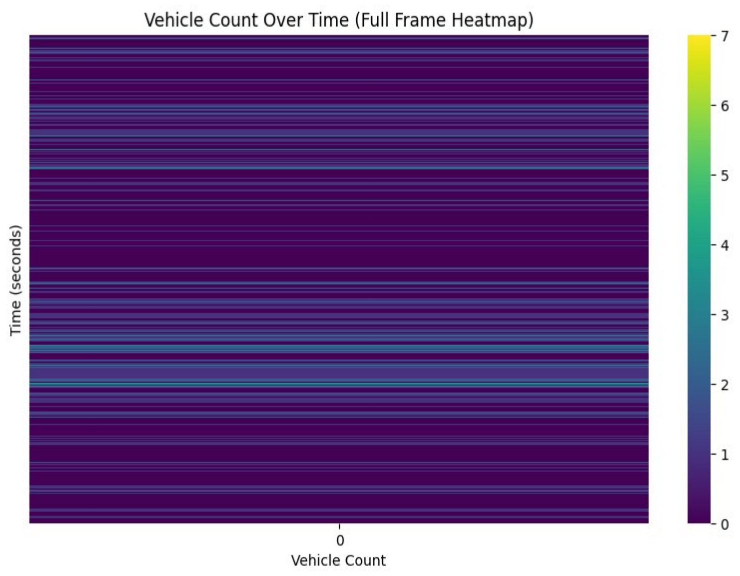

Fig. 7: Vehicle Density Heat Map

Attached below is the segmented frame obtained from converting the raw file to viewable output.

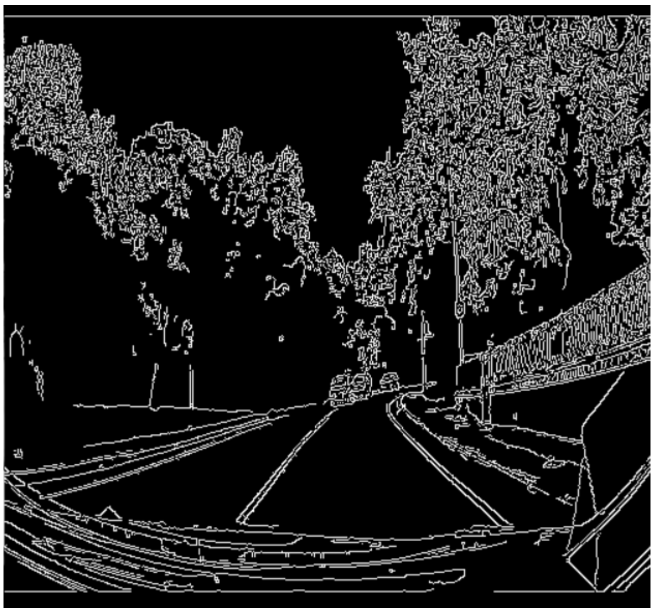

Fig. 8: Segmented Frame

We also generated a geospatial map of the starting point of the vehicular motion based on the latitudinal and longitudinal coordinates available.

```
lat, lon = 43.160133, 12.949957
```

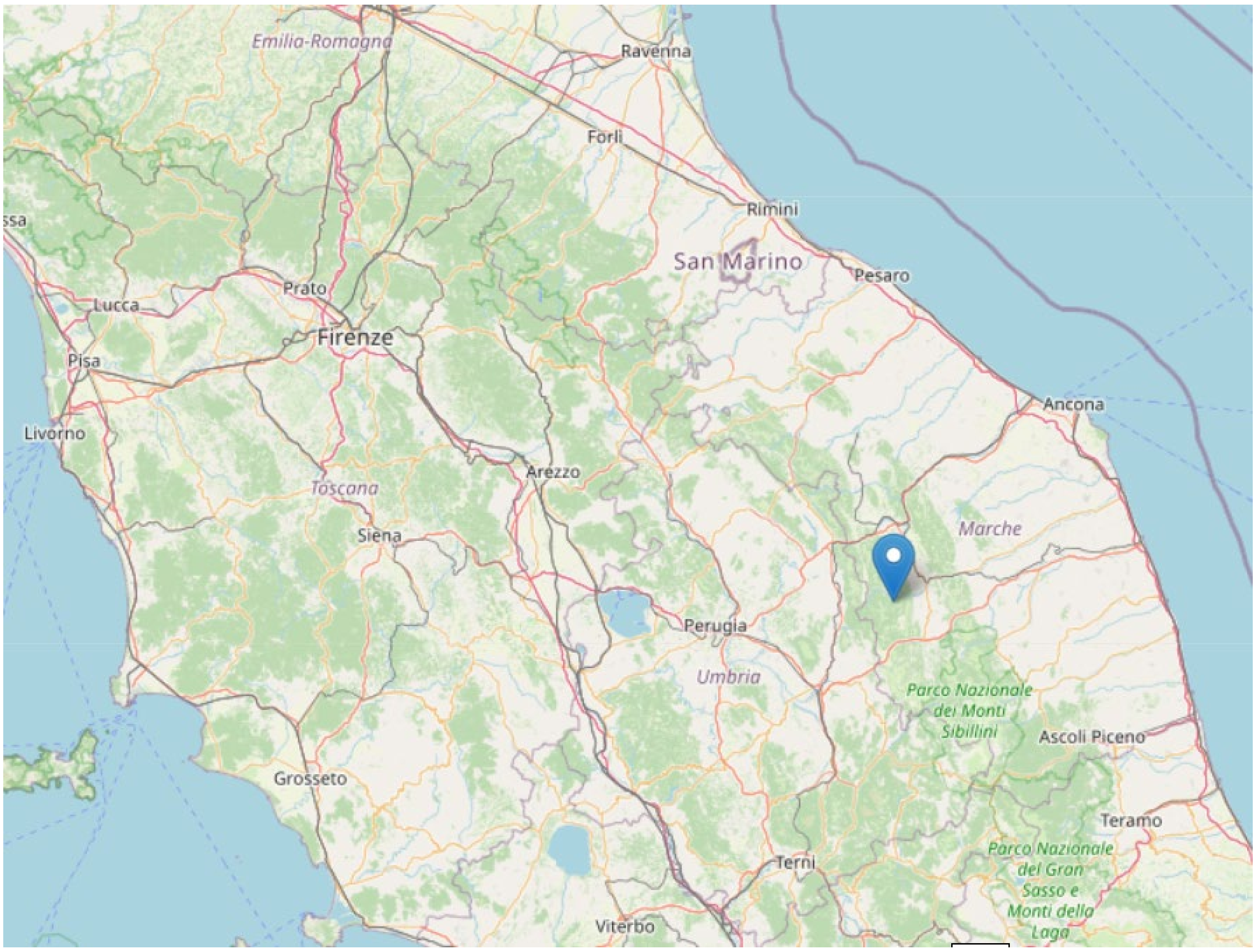

Fig. 9: Geospatial Location(Italy)

6. Conclusion

In conclusion, the work successfully integrates Adaptive Background Subtraction with Optical Flow to analyze optical remote sensing data from vehicular-mounted cameras. By synergistically combining Adaptive Background Subtraction and Optical Flow methods, a system capable of analyzing data from vehicular-mounted cameras to assess traffic patterns and land cover changes dynamically is developed. The approach has been validated by producing heat maps that effectively illustrate traffic congestion and line graphs that depict changes in land cover over time, offering insights for urban development and environmental monitoring. This method shows promise in detecting traffic congestion and monitoring land cover changes over time and provides actionable insights that could benefit urban planning and traffic management.

References:

[1] M. Piccardi, "Background subtraction techniques: a review," 2004 IEEE International Conference on Systems, Man, and Cybernetics (IEEE Cat. No.04CH37583), The Hague, Netherlands, 2004, pp. 3099-3104 vol.4, doi: 10.1109/ICSMC.2004.1400815.

[2] O. Barnich and M. Van Droogenbroeck, "ViBe: A Universal Background Subtraction Algorithm for Video Sequences," in IEEE Transactions on Image Processing, vol. 20, no. 6, pp. 1709-1724, June 2011, doi: 10.1109/TIP.2010.2101613.

[3] Horn, B. K., & Schunck, B. G. (1981). Determining optical flow. Artificial intelligence, 17(1-3), 185-203.

[4] Bouwmans, T., El Baf, F., & Vachon, B. (2008). Background modeling using mixture of Gaussians for foreground detection - a survey. Recent Patents on Computer Science, 1(3), 219-237.

Dataset:

[1] Vehicle-mounted camera data from the National Library of Medicine

[2] Haar Cascade Classifier for cars by Andrews Sobral

Appendix:

Script.py

```
import seaborn as sns
import cv2
import numpy as np
import time
import concurrent.futures
import matplotlib.pyplot as plt
import folium

# Global variable for the classifier
car_cascade = None

def init():
    global car_cascade
    car_cascade = cv2.CascadeClassifier('cars2.xml')

def process_image(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return cv2.resize(gray, (400, 400))

def segment_image(img):
    return cv2.Canny(img, 50, 150)

def detect_cars(img):
    global car_cascade
    height, width = img.shape
    roi = img[int(height/2):, int(width*0.375):]
    cars = car_cascade.detectMultiScale(roi, 1.1, 3)
    img_bgr = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
    for (x, y, w, h) in cars:
        x += int(width*0.375)
        y += int(height/2)
        cv2.rectangle(img_bgr, (x,y), (x+w,y+h), (0,255,0), 2)
    return img_bgr, len(cars)

def detect_cars_full_frame(img):
    global car_cascade
    cars = car_cascade.detectMultiScale(img, 1.1, 3)
    return len(cars)

def apply_background_subtraction(previous_frame, current_frame, alpha):
    return cv2.absdiff(previous_frame, current_frame), cv2.addWeighted(previous_frame, 1-alpha,
current_frame, alpha, 0)

def apply_optical_flow(previous_frame_gray, current_frame_gray):
    return cv2.calcOpticalFlowFarneback(previous_frame_gray, current_frame_gray, None, 0.5, 3, 15, 3, 5,
1.2, 0)

def draw_flow(img, flow, step=16):
    if len(img.shape) == 2:
        vis = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
    else:
        vis = img.copy()
    h, w = vis.shape[:2]
    y, x = np.mgrid[step/2:h:step, step/2:w:step].reshape(2,-1).astype(int)
    fx, fy = flow[y,x].T
    lines = np.vstack([x, y, x+fx, y+fy]).T.reshape(-1, 2, 2)
    lines = np.int32(lines)
    cv2.polylines(vis, lines, 0, (0, 255, 0))
    return vis

def overlay_images(image1, image2):
    return cv2.addWeighted(image1, 0.7, image2, 0.3, 0)

def overlay_images(image1, image2):
    return cv2.addWeighted(image1, 0.7, image2, 0.3, 0)

def calculate_land_cover_change_ratio(foreground_mask):
    non_black_pixels = np.sum(foreground_mask != 0)
    total_pixels = foreground_mask.size
    return non_black_pixels / total_pixels

def main():
    video_file = 'dataset/video-C1-20221011-1.mp4'
    cap = cv2.VideoCapture(video_file)
    fps = cap.get(cv2.CAP_PROP_FPS)
    frame_rate = 1
    frame_count = 0
    counts = []
    time_stamps = []

    video_length_in_frames = cap.get(cv2.CAP_PROP_FRAME_COUNT)
    video_length_in_seconds = video_length_in_frames / fps
    print(f"Video length: {video_length_in_seconds} seconds")

    out = open('output/segmented_output.raw', 'wb')

    ret, frame = cap.read()
    processed_frame = process_image(frame)
    avg = np.float32(processed_frame)
    previous_frame_gray = np.copy(processed_frame)

    start_time = time.time()

    counts_full_frame = []
    time_stamps_full_frame = []
    land_cover_change_ratios = []  # list to store the land cover change ratio for each frame

    with concurrent.futures.ProcessPoolExecutor(max_workers=4, initializer=init) as executor:
        while True:
            ret, frame = cap.read()
            if not ret:
                break

            frame_count += 1
            if frame_count % frame_rate == 0:
```

```
            processed_frame = process_image(cv2.resize(frame, (400, 400)))
            foreground_mask, avg = apply_background_subtraction(avg, np.float32(processed_frame),
alpha=0.02)

            land_cover_change_ratio = calculate_land_cover_change_ratio(foreground_mask)
            land_cover_change_ratios.append(land_cover_change_ratio)  # store the ratio

            future_optical_flow = executor.submit(apply_optical_flow, previous_frame_gray,
processed_frame) if frame_count > 1 else None
            future_detect_cars = executor.submit(detect_cars, processed_frame)  # ROI for vehicle detection
            future_detect_cars_full_frame = executor.submit(detect_cars_full_frame, processed_frame)  # Full
frame for vehicle detection
            future_segment_image = executor.submit(segment_image, processed_frame)

            optical_flow_frame = None
            if future_optical_flow:
                flow = future_optical_flow.result()
                optical_flow_frame = draw_flow(processed_frame, flow)

            detected_frame, count = future_detect_cars.result()
            counts.append(count)
            time_stamps.append(frame_count / fps)

            count_full_frame = future_detect_cars_full_frame.result()
            counts_full_frame.append(count_full_frame)
            time_stamps_full_frame.append(frame_count / fps)

            if optical_flow_frame is not None:
                optical_flow_and_detection = overlay_images(optical_flow_frame, detected_frame)
                cv2.imshow('Optical Flow and Vehicle Detection', optical_flow_and_detection)

            segmented_frame = future_segment_image.result()
            out.write(segmented_frame.tobytes())

            cv2.imshow('Foreground Mask', foreground_mask)

            k = cv2.waitKey(1) & 0xff
            if k == 27:
                break

            previous_frame_gray = np.copy(processed_frame)

    elapsed_time = time.time() - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
    print(f"processing factor: {elapsed_time/video_length_in_seconds}")

    # After processing all frames, plot the land cover change ratios
    plt.figure(figsize=(10, 6))
    plt.plot(time_stamps, land_cover_change_ratios)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Land Cover Change Ratio')
    plt.title('Land Cover Change Ratio Over Time')
    plt.savefig('output/land_cover_change_ratio.png')

    heatmap_data = np.array(counts_full_frame).reshape(-1, 1)
    plt.figure(figsize=(10, 6))
    ax = sns.heatmap(heatmap_data, cmap='viridis', yticklabels=False)  # Turn off default y-tick labels
    n = len(time_stamps_full_frame)
    ticks = ax.get_yticks()
    ax.set_yticks([n-1 if t == max(ticks) else t for t in ticks])  # Re-adjust y-ticks to match timestamps
    ax.set_yticklabels([str(time_stamps_full_frame[int(t)]) for t in ax.get_yticks()])  # Set y-tick labels as
timestamps
    plt.ylabel('Time (seconds)')
    plt.xlabel('Vehicle Count')
    plt.title('Vehicle Count Over Time (Full Frame Heatmap)')
    plt.savefig('output/vehicle_count_full_frame_heatmap.png')

    # Existing plot for ROI vehicle count
    plt.figure(figsize=(10, 6))
    plt.plot(time_stamps, counts)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Vehicle Count (Lane)')
    plt.title('Vehicle Count Over Time (Lane)')
    plt.savefig('output/vehicle_count_lane.png')

    # lat, lon = 43.160133, 12.949957

    # # Create a map centered at the location
    # m = folium.Map(location=[lat, lon], zoom_start=13)

    # # Add a marker to the map for the location
    # folium.Marker([lat, lon], popup='Traffic Congestion').add_to(m)

    # # Save it as html
    # m.save('map.html')

    cap.release()
    out.close()
    cv2.destroyAllWindows()

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