```
import sys

# Install OpenCV (only required if it's not already installed)
!{sys.executable} -m pip install opencv-contrib-python
```

Ensures that the OpenCV library is installed before execution. If running in an environment where OpenCV is already installed, this step can be skipped.

```
import cv2 # opencv library
import math
```

Imports OpenCV for handling video frames and math for mathematical operations. These are essential libraries for video processing tasks.

```
# List of video files
video_paths = ["Traffic_Laramie_1.mp4", "Traffic_Laramie_2.mp4"]

# Loop through each video and display properties
for video_path in video_paths:
    cap = cv2.VideoCapture(video_path)

    if not cap.isOpened():
        print(f"Error: Could not open video file {video_path}")
        continue  # Skip to the next video if it can't be opened

    # Get video properties
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))   # Width of the frames
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)) # Height of the frames
    frame_rate = cap.get(cv2.CAP_PROP_FPS)                 # Frames per second
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))  # Total number of frames
    duration = total_frames / frame_rate if frame_rate > 0 else 0  # Duration in seconds

    # Print video properties
    print(f"\nVideo Properties for {video_path}:")
    print(f"  - Frame Width: {frame_width} pixels")
    print(f"  - Frame Height: {frame_height} pixels")
    print(f"  - Frame Rate: {frame_rate:.2f} FPS")
    print(f"  - Total Frames: {total_frames}")
    print(f"  - Duration: {duration:.2f} seconds")

    cap.release()  # Release video capture before moving to the next
```

Loads multiple video files, checks if they open successfully, and extracts key properties such as resolution, FPS, and duration. Ensures robust error handling if a video file cannot be opened. Extracting video properties is useful for preprocessing before motion detection.

```
# Define Region of Interest (ROI) to limit detection area
detection_area = frame[260:600, 0:1040]
```

A specific region of the video frame is selected to limit detection to relevant areas, reducing noise from background activity

```
# Apply background subtraction to detect moving objects
fgmask = bg_sub.apply(detection_area)
_, fgmask = cv2.threshold(fgmask, 254, 255, cv2.THRESH_BINARY)  # Binarize the mask
contours, _ = cv2.findContours(fgmask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)  # Find object contours
```

The function then applies background subtraction to detect moving objects, followed by thresholding to obtain a binary mask. Contours are extracted from the binary mask to identify potential vehicles.

```
# Large objects, likely cars, split into two parts for better tracking
if x < 520 and area > 13500 and h > 130:
    cx1, cy1 = int(x + (x + w) / 2), int(y + (y + h/2) / 2)
    cx2, cy2 = int(x + (x + w) / 2), int(y + h/2 + (y + h/2) / 2)
    current_frame_centroids.extend([(cx1, cy1), (cx2, cy2)])

    # Draw bounding boxes around detected cars
    cv2.rectangle(detection_area, (x, y), (x + w, y + int(h/2)), (0, 255, 0), 1)
    cv2.rectangle(detection_area, (x, y + int(h/2)), (x + w, y + h), (0, 255, 0), 1)

# Smaller objects, also likely vehicles
elif area > 4700:
    cx, cy = int(x + (x + w) / 2), int(y + (y + h) / 2)
    current_frame_centroids.append((cx, cy))
```

The detected objects are then filtered based on their area, distinguishing between large vehicles (which may be split into two parts) and smaller vehicles. Bounding boxes are drawn around detected vehicles to visualize tracking

```
# Calculate Euclidean distance between tracked and current centroid
distance = math.sqrt((tracked_obj[0] - ccentroid[0])**2 + (tracked_obj[1] - ccentroid[1])**2)

# If the object is within a reasonable distance, track it
if distance < 100:
    if nearest_track_obj and nearest_track_obj[2] > distance:
        nearest_track_obj = [ccentroid[0], ccentroid[1], distance, tracked_obj[2]]
```

A dictionary, tracked_objects, is used to maintain a record of tracked vehicles based on their centroids. The function calculates the Euclidean distance between detected centroids and existing tracked objects to determine if they should be assigned a new tracking ID or linked to an existing object. If no match is found, a new object is added to tracked_objects.

```
if centroid[0] < 215:  # If the object moves past a certain point
    car_counter += 1  # Increase car count
```

Vehicles are counted when they cross a predefined boundary.

```
# Display the car count on the video frame
cv2.putText(frame, 'Cars Going to City Center: ', (10, 50),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 1, cv2.LINE_AA)
cv2.putText(frame, str(car_counter), (500, 50),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA)
```

The car count is displayed on the video frame using OpenCV

```
# Initialize background subtractor (you can reuse it)
bg_sub = cv2.createBackgroundSubtractorMOG2(history=15000, varThreshold=20, detectShadows=False)
```

A background subtractor is initialized once per video using the MOG2 algorithm. The history parameter defines how many frames the model should remember, and varThreshold controls sensitivity.

History ensures that long-term static objects are recognized as background.

varThreshold helps differentiate moving objects from the background.

detectShadows disables shadow detection to reduce false positives.

Final_Traffic_Analysis_Table

Video File	Total number of cars	Cars per minute
Traffic_Laramie_1.mp4	6	2.02
Traffic_Laramie_2.mp4	6	3.41

The application uses computer vision algorithms to process various traffic surveillance footage in order to detect and count vehicles. It counts the number of cars that pass through a predetermined area by iterating over a set of video files, subtracting the backdrop, and using centroid-based tracking. By keeping a distinct function for car counting that can be applied to other video files without requiring changes, the method guarantees modularity.

The application's automation and scalability are two of its main advantages. It becomes an effective tool for examining traffic patterns across several places or timeframes by enabling the sequential processing of many recordings without necessitating modifications to the fundamental logic. Vehicle detection is made more effective by removing immobile backdrop objects through the usage of OpenCV's cv2.createBackgroundSubtractorMOG2 for motion detection. The program also has error-handling features that keep the process from stopping altogether in the case that a video file doesn't open.

However, some restrictions might affect the tracking accuracy. The current solution may have trouble with overlapping cars, which could result in miscounts, because it depends on centroid proximity to track moving vehicles. A more advanced tracking algorithm, like Kalman Filters or Deep SORT, which are better suited for managing occlusions and object re-identification, could be used to solve this problem.

The use of predefined parameters for background removal is another possible disadvantage. Because the history and varThreshold values are preset, they might not adjust well to changes in illumination or video quality. The accuracy of detection would be increased by a more dynamic method that modifies these parameters according on the features of each video. Additionally, rather than analysing real-time traffic inputs, the application can only process recorded films at this moment. It might be more useful for traffic monitoring in the real world if its capabilities were expanded to support live video broadcasts.

To sum up, the application offers a strong basis for counting and detecting vehicles in traffic footage. It is a useful tool for batch processing because of its automated workflow and modular design. For real-world traffic monitoring applications, however, its utility and dependability would be greatly increased by advancements in tracking precision, adaptive parameter tuning and real-time processing.