MEMORIA 2º TRABAJO
FUNDAMENTOS DE SISTEMAS INTELIGENTES

Pablo Herrera González

Pablo Cabeza Lantigua

Universidad de Las Palmas de Gran Canaria

Fundamentos de los SIstemas Inteligentes

2nd Año Grado en Ciencia en Ingeniería de Datos

2024-2025

Índice

Introducción

Este proyecto se centra en el análisis automatizado de tráfico vehicular utilizando técnicas avanzadas de visión por computador. Implementado en Python con la biblioteca **OpenCV**, el sistema procesa un video pregrabado de tráfico y aplica técnicas de segmentación, seguimiento de objetos, y detección de eventos para contar vehículos que cruzan líneas específicas en diferentes carriles.

La solución incluye el uso de un algoritmo de sustracción de fondo (**Background Subtractor**) para detectar objetos en movimiento y determinar si cruzan líneas predefinidas en el video. El código ha sido diseñado para:

1. **Detectar y filtrar contornos:** Asegurando que solo se consideren vehículos que cumplen ciertos criterios como tamaño y posición en relación a los carriles.
2. **Rastrear objetos:** Evitando contar repetidamente el mismo vehículo al mantener un registro temporal de los centroides detectados.
3. **Contabilizar vehículos por carril:** Mostrando estadísticas en tiempo real y líneas visuales para cada carril directamente sobre el video.

El código emplea varias estrategias clave para lograr un desempeño efectivo:

- **Sustracción de fondo con cv2.createBackgroundSubtractorMOG2:** Para separar los objetos en movimiento del fondo.
- **Procesamiento de contornos:** Para obtener información como la posición y el tamaño de los vehículos detectados.
- **Detección de cruces de línea:** Utilizando una función personalizada (line_crossed) que valida si el centroide de un vehículo cruza una línea horizontal definida con coordenadas específicas.
- **Seguimiento temporal:** Para garantizar que cada vehículo sea contado solo una vez.

Contexto del Proyecto

El análisis automatizado de tráfico tiene aplicaciones prácticas en sistemas inteligentes de transporte, tales como:

- Optimización de flujos vehiculares.
- Evaluación de patrones de tráfico.
- Desarrollo de infraestructuras más seguras y eficientes.

Con este enfoque, se pretende proporcionar un marco inicial que pueda ser extendido para casos más complejos como el análisis en tiempo real o la clasificación de diferentes tipos de vehículos.

Este trabajo no sólo ejemplifica la potencia de las herramientas de visión por computador en el ámbito del transporte, sino que también sienta las bases para sistemas más robustos y escalables en la gestión de tráfico.

Librerías

La librería que se ha empleado en el desarrollo de este proyecto ha sido únicamente la librería **cv2**. **OpenCV** (Open Source Computer Vision) es una biblioteca de código abierto para procesamiento de imágenes y visión por computadora. La versión **cv2** es la interfaz para Python que permite usar esta biblioteca.

Módulos

Hemos creado un módulo para desarrollar funciones auxiliares que después hemos incorporado al algoritmo, como la indispensable función ***line_crossed(coordx, coordy, line)*** que fue desarrollada en el archivo, ***utils.py*** y que será explicada más adelante junto con **speed_calc(prev_obj, actual_obj, fps)**.

Funciones

- ## Funciones Desarrolladas

Hemos necesitado únicamente dos funciones que son **line_crossed**(coordx, coordy, line) y **speed_calc**(prev_obj, actual_obj, fps). Analizemos paso a paso en que consiste cada función:

```
def line_crossed(coordx, coordy, line):
        return line["cx1"] <= coordx <= line["cx2"] and abs(coordy - line["cy1"]) <= 10
```

La función requiere de tres parámetros, las coordenadas ***x*** y las coordenadas ***y*** de cada objeto, en este caso **coordx** será el centro horizontal del objeto , y **coordy** será el centro vertical del objeto. Sabiendo esto, line será en nuestro caso un diccionario en este formato: **{"cx1": 480, "cy1": 850, "cx2": 580, "cy2": 850}** , donde cada clave es una coordenada, que juntas delimitaran el área respectiva en el que se encuentra la línea, en este caso **cy1** y **cy2** serán siempre iguales porque nos interesa que las líneas sean totalmente horizontales.

Por lo tanto cuando se realiza: **line[“cx1”]**, se accede al valor de dicha coordenada en el ejemplo propuesto, **cx1 = 480**. Entonces lo que retorna esta función es un valor booleano (True, False), en función de que las dos condiciones se cumplan.

La segunda condición calcula la distancia absoluta entre la **coordy** y la línea, en caso de que el resultado (distancia) de esa operación sea menor que diez pixeles, cuenta como cruzado.

Pasemos con la segunda función **speed_calc():**

```
def speed_calc(obj_prev, obj_actual, fps):
        """ Calcula la velocidad de un objeto dado su posición previa y actual. Parámetros:
        obj_prev: tuple (x_prev, y_prev, frame_prev), posición previa. obj_actual: tuple
        (x_actual, y_actual, frame_actual), posición actual. fps: int, frames por segundo del
        video. Retorna: float, velocidad en píxeles por segundo. """
        x_prev, y_prev, frame_prev = obj_prev
        x_actual, y_actual, frame_actual = obj_actual
        distancia = ((x_actual - x_prev) ** 2 + (y_actual - y_prev) ** 2) ** 0.5
        tiempo = (frame_actual - frame_prev) / fps
        return distancia / tiempo if tiempo > 0 else 0
```

Esta función calcula la velocidad de los objetos para cada carril basándose en la fórmula:

$$v = \frac{distancia}{tiempo}$$

Por tanto los parámetros, **prev_obj** es una tupla compuesta por tres valores **prev_x**, **y_prev**, **prev_frame**, **x_prev**, **y_prev** son las coordenadas previas en píxeles del centroide del objeto. Luego **prev_frame** es el número de fotograma en el que se registró la posición.

Además **actual_obj**, es una tupla muy parecida a la anterior pero que contendrá, **x_actual**, **y_actual**, **actual_frame**, siendo **x_actual**, **y_actual** las coordenadas actuales del centroide del objeto, **actual_frame** es el número de fotograma actual.

Finalmente el parámetro **fps** son los frames por segundo del video, esto, permite convertir la diferencia de fotogramas a segundos.

El cálculo implementado en la función para hallar la distancia es el siguiente:

$$distance = \sqrt{(x_{actua} - x_{prev})^2 + (y_{actual} - y_{prev})^2}$$

Y para hallar el tiempo:

$$v = \frac{frame_{actual} - frame_{prev}}{fps}$$

Finalmente la velocidad es calculada si tiempo mayor que 0:

$$v = \frac{distancia}{tiempo}$$

- **<u>Funciones Desarrolladas Implementación</u>**

La función **line_crossed()** fue implementada una única vez:

```
contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
for contour in contours:
	x, y, w, h = cv2.boundingRect(contour)
	centroidx = x + w // 2
	centroidy = y + h // 2
	if cv2.contourArea(contour) < 520 or w > max_width:
		continue
	for i, line in enumerate(lines_dic):
		if line_crossed(centroidx, centroidy, line):
```

En la primera fase de este código observamos como la primera línea encuentra los contornos en la máscara binaria, además se usan los parámetros **cv2.RETR_EXTERNAL**, que extrae los contornos más externos, y **cv2.CHAIN_APPROX_SIMPLE** que simplifica los contornos para ahorrar memoria.

A continuación se realiza un bucle para cada contorno, donde se calcula un rectángulo delimitador para el contorno, donde x e y son las coordenadas de la esquina superior izquierda, el ancho y h la altura del rectángulo. Más tarde calculamos los centroides para el rectángulo. Siguiendo con la primera condición de este bucle donde se debe cumplir al menos una de las dos afirmaciones, o que el área sea menor a 520 (ruido de fondo u objetos innecesarios), o que el ancho del rectángulo sea mayor que la anchura máxima definida anteriormente, si esto ocurre se ignoran esos objetos.

Finalmente iteramos sobre las líneas del diccionario, para comprobar si ha cruzado dicha línea los centroides calculados usando la función **line_crossed().**

A continuación veremos cómo fue implementada la función **speed_calc()**:

```
fps = cap.get(cv2.CAP_PROP_FPS)...
…
if matching_object is not None:
       obj_prev = track_obj[i][matching_object]
       track_obj[i][matching_object] = (centroidx, centroidy, frames_counter)
       velocidad = speed_calc(obj_prev, track_obj[i][matching_object], fps)
       print(f"Velocidad del carril {i + 1}: {velocidad:.2f} píxeles/segundo")
```

Primeramente hay que definir la variable **fps**, que con la función get() del paquete cv2, y el parámetro CAP_PROP_FPS, se representa la cantidad de **frames por segundo (FPS)** del video. El método devuelve el FPS del video o de la cámara conectada. Por ejemplo:

- Para un video de 30 FPS, devolverá 30.0.
- Si no puede leer la propiedad, puede devolver un valor incorrecto o 0.

Explicando el condicional donde se implementa:

Esta línea verifica si se encontró un objeto previamente rastreado en el diccionario track_obj[i] que coincida con las coordenadas actuales del nuevo objeto dentro del margen permitido (por ejemplo, 50 píxeles en cada dirección). Si matching_object no es None, significa que se identificó un objeto existente que ya está siendo rastreado. Prosiguiendo la **Segunda línea: obj_prev = track_obj[i][matching_object],** se obtiene la información previa del objeto identificado (matching_object) desde el diccionario de objetos rastreados (track_obj[i]). La información que se guarda en obj_prev suele incluir:

- old_centroidx: Coordenada X previa del centroide.
- old_centroidy: Coordenada Y previa del centroide.
- last_seen: El último cuadro (frame) en el que se actualizó este objeto.

Pasando a la tercera línea: **track_obj[i][matching_object] = (centroidx, centroidy, frames_counter),** El diccionario track_obj[i] se actualiza con las nuevas coordenadas del centroide (centroidx, centroidy) y el número actual de cuadro (frames_counter). Esto significa que se registra la posición más reciente del objeto y el momento en que fue visto.

Por ejemplo, si las nuevas coordenadas son (320, 410) y el cuadro actual es 12, entonces: track_obj[i][matching_object] = (320, 410, 12)

En la siguiente línea **velocidad = speed_calc(obj_prev, track_obj[i][matching_object], fps).** Se calcula la velocidad del objeto rastreado. Para ello, se utiliza una función llamada speed_calc, que toma los siguientes parámetros: **obj_prev**: La posición previa y el cuadro del objeto (por ejemplo, (300, 400, 10)). **track_obj[i][matching_object]**: La posición actual y el cuadro del objeto (por ejemplo, (320, 410, 12)). **fps**: Los cuadros por segundo del video, que permiten calcular la velocidad en función del tiempo transcurrido entre cuadros.

La función speed_calc generalmente calcula la distancia recorrida por el objeto (en píxeles) dividida por el tiempo transcurrido (en segundos). Por ejemplo, si la distancia entre los dos puntos es 22.36 píxeles y el tiempo entre cuadros es 0.08 segundos (1/12.5 fps), la velocidad sería:

velocidad = 22.36 / 0.08 ≈ 279.5 píxeles/segundo. Finalmente el **print(f"Velocidad del carril {i + 1}: {velocidad:.2f} píxeles/segundo"),** imprime la velocidad calculada para el objeto en el carril correspondiente (i + 1). El valor de la velocidad se formatea con dos decimales mediante :.2f.

- ## Funciones Nativas

Las funciones nativas bien sea de **python** base o **cv2** serán enumeradas y más tarde explicadas en su implementación:

```
append()
next()
abs()
enumerate()
ord()
print()
items()
isOpened()
get()
VideoCapture()
read()
release()
createBackgroundSubtractorMOG2()
apply()
morphologyEx()
threshold()
findContours()
boundingRect()
cvtColor()
getStructuringElement()
contourArea()
line()
putText()
imshow()
waitKey()
destroyAllWindows()
```

- **Funciones Nativas Implementación**

```
import cv2

from utils import line_crossed

video = 'trafico01.mp4'
go = True
max_width = 420
max_tracking = 14

lines_dic = [ {"cx1": 480, "cy1": 850, "cx2": 580, "cy2": 850}, {"cx1": 610, "cy1": 850, "cx2":
720, "cy2": 850}, {"cx1": 990, "cy1": 750, "cx2": 1090, "cy2": 750}, {"cx1": 1320, "cy1": 900,
"cx2": 1470, "cy2": 900}, {"cx1": 1200, "cy1": 660, "cx2": 1260, "cy2": 660}, {"cx1": 1420,
"cy1": 730, "cx2": 1500, "cy2": 730}, {"cx1": 1650, "cy1": 780, "cx2": 1770, "cy2": 780} ]

while go:
cap = cv2.VideoCapture(video)
counter = [0 for _ in lines_dic]
object_ids = [0 for _ in lines_dic]
track_obj = []
for _ in lines_dic:
        track_obj.append({})

if not cap.isOpened():
        print("Error opening the file: couldn't open the file")
        break
bg_subtractor = cv2.createBackgroundSubtractorMOG2(history=300, varThreshold=50,
detectShadows=False)
frames_counter = 0
while True:
r, frame = cap.read()
if not r:
        cap.release()
        break
frames_counter += 1
mask = bg_subtractor.apply(cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY))
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5)))

 _, mask = cv2.threshold(mask, 254, 255, cv2.THRESH_BINARY)
contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
fps = cap.get(cv2.CAP_PROP_FPS)
```

```
for contour in contours:
        x, y, w, h = cv2.boundingRect(contour)
        centroidx = x + w // 2
        centroidy = y + h // 2
        if cv2.contourArea(contour) < 500 or w > max_width:
                continue
        for i, line in enumerate(lines_dic):
                if line_crossed(centroidx, centroidy, line):
                        matching_object = next((objct_id for objct_id, (old_centroidx,
                old_centroidy, _) in track_obj[i].items() if abs(centroidx - old_centroidx) < 50
                and abs(centroidy - old_centroidy) < 50), None)

                if matching_object is not None:
                        obj_prev = track_obj[i][matching_object] track_obj[i][matching_object] =
                        (centroidx, centroidy, frames_counter)
                        velocidad = speed_calc(obj_prev, track_obj[i][matching_object], fps)
                        if speed > 0:
                                print(f"Velocidad del carril {i + 1}: {velocidad:.2f}
                        píxeles/segundo")
                else:
                        track_obj[i][object_ids[i]] = (centroidx, centroidy, frames_counter)
                        counter[i] += 1
                        object_ids[i] += 1
for i, _ in enumerate(lines_dic):
        expired_ids = [objct_id for objct_id, (_, _, last_seen) in track_obj[i].items() if
frames_counter - last_seen > max_tracking]
        for objct_id in expired_ids:
                del track_obj[i][objct_id]

for i, line in enumerate(lines_dic):
        text = f"Lane {i + 1}: {counter[i]}"
        cv2.line(frame, (line["cx1"], line["cy1"]), (line["cx2"], line["cy2"]), (0, 255, 255), 3)
        cv2.putText(frame, text, (line["cx1"], line["cy1"] - 15), cv2.FONT_HERSHEY_DUPLEX, 0.5,
        (250, 50, 255), 2)

cv2.imshow("Mask", mask)
cv2.imshow("DGT Record", frame)

if cv2.waitKey(27) & 0xFF == ord('s'):
        go = False
        break
```

```
cap.release()
cv2.destroyAllWindows()

print("========== Vehicle Counts by Lane ==========")
for i, count in enumerate(counter):
        print(f"🚗 Lane {i + 1}: {count} vehicles detected")
print("============================================")
```

- *Gestión de Listas y Estructuras de Datos*

append():
Se utiliza para agregar un diccionario vacío a la lista track_obj. Cada diccionario dentro de track_obj representa un carril y almacena información de los objetos detectados y rastreados en ese carril. Esto permite gestionar múltiples carriles simultáneamente.

items():
Recupera los pares clave-valor de los diccionarios en track_obj para iterar sobre ellos. Esto es necesario para acceder al identificador de cada objeto (objct_id) y a su información (posición y tiempo) durante el proceso de rastreo.

- *Control de Flujo y Búsqueda*

next():
Busca y retorna el primer objeto en track_obj[i] (el diccionario del carril actual) que esté lo suficientemente cerca del objeto detectado. Esto evita duplicar el rastreo de un mismo objeto. Si no encuentra un objeto coincidente, retorna None.

- *Cálculos Matemáticos y Manejo de Datos*

abs():
Calcula la distancia absoluta entre los centroides actuales y previos de los objetos. Esto determina si el objeto detectado pertenece a un objeto previamente rastreado en el carril.

enumerate():
Itera sobre lines_dic, que contiene las coordenadas de las líneas de los carriles. Proporciona tanto el índice (número del carril) como el diccionario con las coordenadas de la línea, facilitando el procesamiento organizado por carril.

- *Manejo de Teclas y Consola*

ord():
Convierte la tecla 's' a su código ASCII para compararla con el valor de retorno de cv2.waitKey(). Esto detiene el programa cuando se presiona 's'.

print():
Imprime información en la consola, como las velocidades calculadas de los vehículos y los conteos finales por carril. Ayuda a depurar y reportar resultados.

- *Manejo de Vídeo y Fotograma*

VideoCapture():
Abre el archivo de vídeo especificado en video para su procesamiento fotograma a fotograma.

read():
Lee el siguiente fotograma del video. Si no hay más fotogramas disponibles (por ejemplo, el video termina), devuelve False.

release():
Cierra el archivo de video y libera los recursos asociados para evitar fugas de memoria.

isOpened():
Comprueba si el video se pudo abrir correctamente. Si no, muestra un mensaje de error y detiene la ejecución.

get():
Obtiene información del video, como el número de fotogramas por segundo (FPS), que es fundamental para calcular velocidades.

- *Detección y Segmentación*

createBackgroundSubtractorMOG2():
Crea un modelo de sustracción de fondo para identificar objetos en movimiento. Esto elimina el fondo estático del video, dejando solo los objetos dinámicos.

apply():
Aplica el modelo de sustracción al fotograma actual, generando una máscara que resalta los objetos en movimiento.

morphologyEx():
Realiza operaciones de cierre morfológico en la máscara para reducir el ruido (pequeños puntos o huecos) y unir áreas desconectadas.

threshold():
Convierte la máscara en una imagen binaria (blanco y negro) para simplificar la detección de objetos.

findContours():
Encuentra los contornos de los objetos en la máscara binaria. Estos contornos se usan para identificar los límites de los vehículos.

- *Medición y Validación*

boundingRect():
Calcula un rectángulo delimitador alrededor de un contorno. Esto define la posición y tamaño del objeto detectado.

contourArea():
Calcula el área de un objeto detectado. Los objetos con un área demasiado pequeña o demasiado grande (ruido o vehículos irrelevantes) son descartados.

- *Procesamiento y Visualización*

cvtColor():
Convierte el fotograma de color a escala de grises antes de aplicar la sustracción de fondo. Esto simplifica el análisis al eliminar la información de color.

getStructuringElement():
Crea un elemento estructurante utilizado en las operaciones morfológicas para procesar la máscara (como suavizar bordes).

line():
Dibuja líneas de referencia en la imagen del video para mostrar los carriles y las líneas de detección.

putText():
Agrega texto en el fotograma del video, como el número de vehículos detectados en cada carril.

imshow():
Muestra dos ventanas: una con la máscara procesada (Mask) y otra con el video en tiempo real (DGT Record).

waitKey():
Pausa el procesamiento por un corto tiempo para mostrar los fotogramas. Detecta si se presiona la tecla 's' para detener el programa.

destroyAllWindows():
Cierra todas las ventanas de OpenCV al final del procesamiento.

Problemas encontrados y soluciones

En el primer bucle **while** hubo que inicializar una variable ***go***, a la que asignaríamos el valor de True porque realizar un bucle **while True** no permitía cerrar el programa de la forma que se requiería. En la función **cv2.createBackgroundSubtractorMOG2()** el parámetro detectShadows por defecto está en el valor True, esto resultaba en errores de conteo para el carril central sobre todo por los vehículos grandes, por tanto hubo que manualmente inicializarlo a False, para que no detectara las sombras.

Output Final

```
========== Vehicle Counts by Lane ==========
🚗 Lane 1: 5 vehicles detected
🚗 Lane 2: 4 vehicles detected
🚗 Lane 3: 11 vehicles detected
🚗 Lane 4: 14 vehicles detected
🚗 Lane 5: 4 vehicles detected
🚗 Lane 6: 10 vehicles detected
🚗 Lane 7: 6 vehicles detected
============================================
```

Conclusión

El desarrollo de este sistema de análisis de tráfico vehicular mediante visión por computador ha demostrado ser una herramienta poderosa para automatizar la contabilización de vehículos en diferentes carriles. A través de técnicas como la sustracción de fondo, el procesamiento de contornos y el seguimiento de objetos, se logró construir una solución funcional capaz de analizar videos y generar estadísticas útiles en tiempo real. Este trabajo destaca el potencial de las tecnologías basadas en OpenCV para abordar problemas prácticos en sistemas de transporte.

Sin embargo, durante el desarrollo surgieron varios contratiempos que reflejan los desafíos inherentes a proyectos de esta naturaleza. El proyecto ha servido como un punto de partida valioso para explorar sistemas de análisis de tráfico automatizados. Como posibles mejoras futuras, se propone:

- Implementar técnicas de aprendizaje profundo para una detección más robusta y menos dependiente de parámetros manuales.
- Incorporar un módulo para clasificar vehículos por tipo (autos, camiones, motocicletas).
- Optimizar el sistema para su implementación en tiempo real con videos en streaming.