FCEIA - UNR

Procesamiento de Imágenes y Visión por Computadora (IA52)

Trabajo Práctico N°. 3

Lunes 23 de Junio del 2025

Gianfranco Frattini (F-3771/1)

Alejandro Peralta (P-5293/1)

Matias Prado (P-5299/1)

Índice

Descripción del Script

El script se compone principalmente de dos funciones: procesar_video y detectar_carril.

func *procesar_video*

1. Carga del video de entrada: Inicializa la captura del video especificado por ruta_video usando cv2.VideoCapture.
2. Obtención de propiedades del video: Extrae el ancho (ancho), alto (alto), y la tasa de fotogramas por segundo (fps) del video de entrada.
3. Configuración del escritor de video: Prepara un objeto cv2.VideoWriter para guardar el video procesado en salida_video con el códec mp4v.
4. Creación de carpeta de salida para imágenes: Se genera una carpeta llamada «procesos» y dentro de ella, una subcarpeta con el nombre base del video de entrada (ej., «ruta_1» para «ruta_1.mp4»). Esta carpeta se utiliza para guardar imágenes intermedias del procesamiento del primer frame.
5. Bucle de procesamiento de frames:

- Itera sobre cada frame del video usando cap.read().
- Para el primer frame (primer_frame = True), la función detectar_carril es llamada con el parámetro guardar_imagenes en True y se le pasa la carpeta_salida. Esto asegura que las imágenes intermedias solo se guarden para el primer frame y no ralenticen el procesamiento del video completo.
- Para los frames subsiguientes, guardar_imagenes se establece en False.
- El resultado del procesamiento del frame (resultado) se escribe en el video de salida (out.write(resultado)).
- Existe un bloque de código comentado que permitiría mostrar el resultado en tiempo real y salir con la tecla “q”, pero se ha deshabilitado para acelerar la ejecución.

6. Liberación de recursos: Una vez que se procesan todos los frames, se liberan los recursos de captura y escritura de video (cap.release(), out.release()) y se cierran todas las ventanas de OpenCV (cv2.destroyAllWindows())

func *detectar_carril*

1. original.jpg:

- Proceso: Si guardar_imagenes es True, se guarda una copia exacta del frame de entrada.

- Propósito: Mostrar el frame original sin ninguna modificación, sirviendo como punto de partida visual del procesamiento.

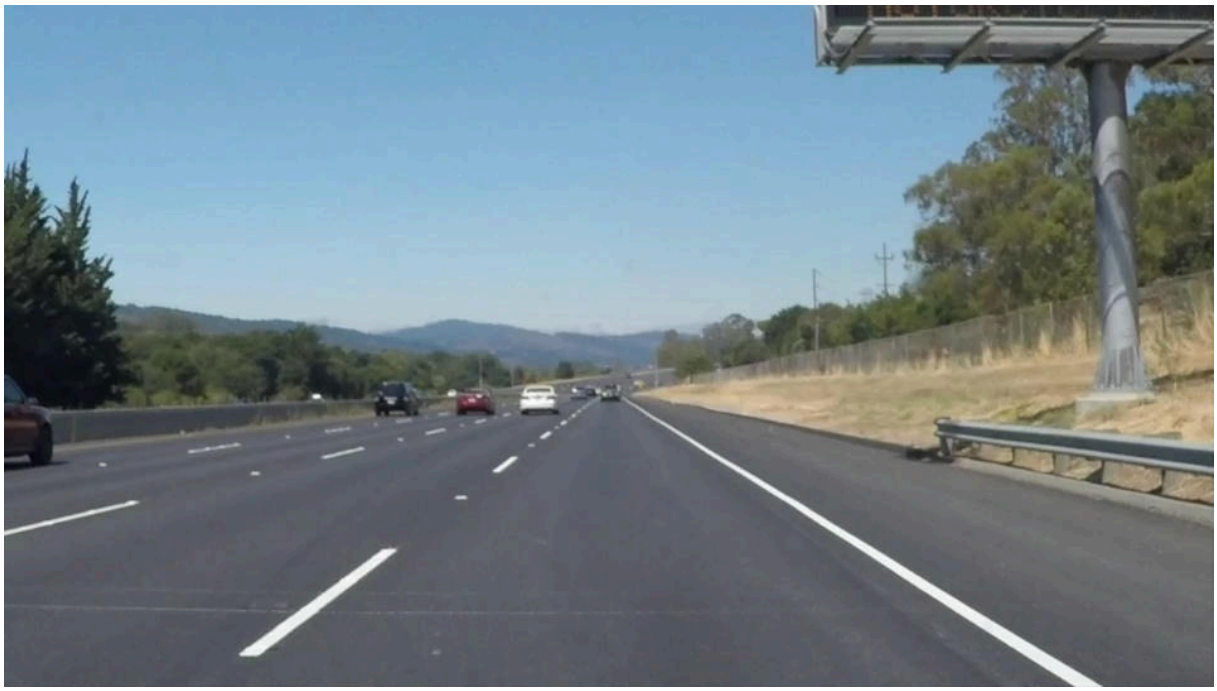

Figura 1: Imagen original del primer frame de ruta_1

2. Conversión a escala de grises y gris.jpg:

- Proceso: El frame se convierte de espacio de color BGR (formato por defecto de OpenCV para imágenes a color) a escala de grises usando cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY). Si guardar_imagenes es True, se guarda esta imagen.
- Propósito: Reducir la complejidad de los datos (de 3 canales a 1) y simplificar el procesamiento posterior, ya que la detección de bordes y líneas no requiere información de color.

Figura 2: Imagen a escala de gris del primer frame de ruta_1

3. Aplicación de filtro Gaussiano y blur.jpg:

- Proceso: Se aplica un filtro Gaussiano al frame en escala de grises (gris) con un tamaño de kernel de (5, 5) usando cv2.GaussianBlur(gris, (5, 5), 0). Si guardar_imagenes es True, se guarda esta imagen.
- Propósito: Suavizar la imagen y reducir el ruido. Esto es crucial antes de la detección de bordes para evitar que el ruido se interprete como bordes.

Figura 3: Imagen con filtro gaussiano del primer frame de ruta_1

4. Detección de bordes Canny y canny.jpg:

- Proceso: Se aplica el algoritmo de detección de bordes Canny al frame suavizado (blur) con umbrales de 70 y 150 (cv2.Canny(blur, 70, 150)). Si guardar_imagenes es True, se guarda esta imagen.
- Propósito: Identificar los bordes fuertes en la imagen, que son candidatos a ser los bordes de los carriles. El algoritmo Canny es efectivo para producir bordes finos y fuertes.

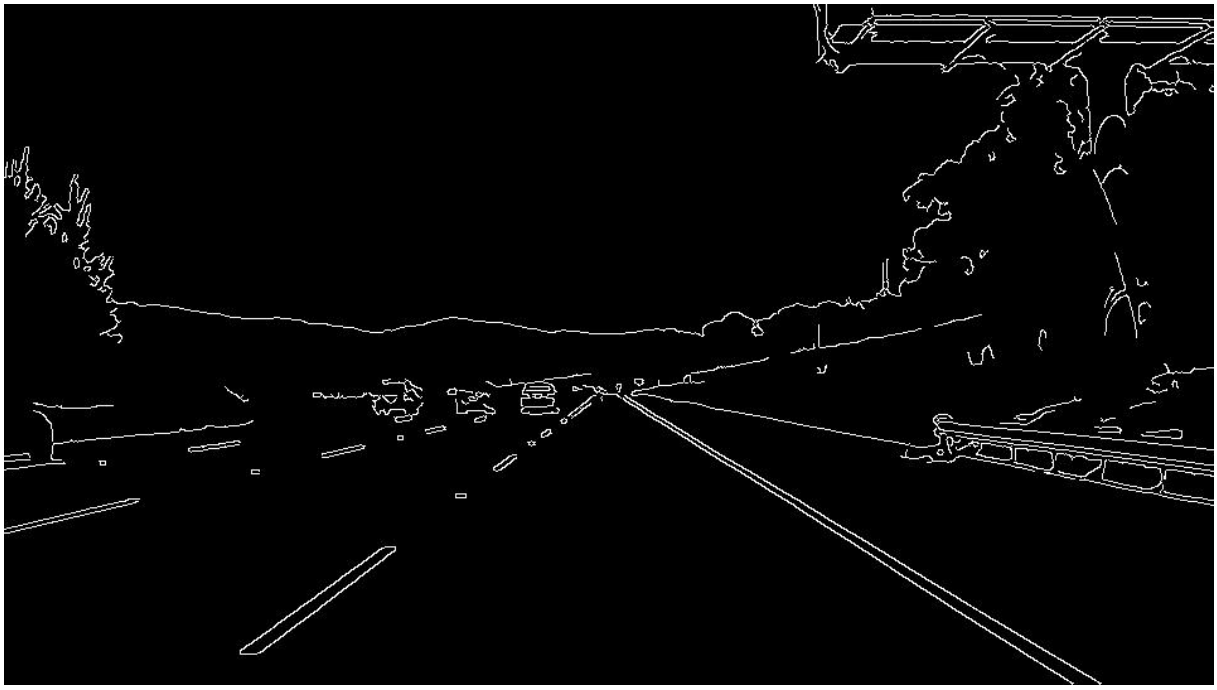

Figura 4: Imagen con detección de bordes canny del primer frame de ruta_1

5. Definición de la Región de Interés (ROI) y mascara.jpg, roi.jpg:

Proceso:

- Se define un polígono (poligono) que representa la región de la imagen donde se espera encontrar los carriles. En este caso, es un triángulo que cubre la parte inferior de la imagen y se estrecha hacia el horizonte, adaptándose a la perspectiva de la carretera.
- Se crea una máscara (mascara) del mismo tamaño que la imagen Canny, inicializada en negro (ceros).
- El polígono definido se dibuja y rellena con color blanco (255) en la máscara usando cv2.fillPoly.
- La máscara se aplica a la imagen Canny (canny) usando una operación bitwise AND (cv2.bitwise_and(canny, mascara)). Esto filtra los bordes, manteniendo solo aquellos que están dentro de la ROI.

- Si guardar_imagenes es True, se guarda la mascara binaria por separado como mascara.jpg.
- También se crea una copia del frame original (roi_visual), se dibuja el poligono sobre ella con líneas de color cian ((0, 255, 255)) para visualizar la ROI sobre el contexto original, y se guarda como roi.jpg.

Propósito:

Concentrar la detección de líneas en el área relevante de la carretera y descartar ruido o elementos irrelevantes fuera del carril (como árboles, cielo, otros vehículos), lo que mejora la precisión y eficiencia del algoritmo.

Figura 5: Mascara de la región de interés del primer frame de ruta_1

Figura 6: Región de interés (ROI) colocada sobre el primer frame de ruta_1

6. Detección de líneas Hough y clasificación (izquierda/derecha):

Proceso:

- Se utiliza la Transformada de Hough probabilística (cv2.HoughLinesP) en la imagen roi (bordes Canny filtrados por la ROI) para detectar segmentos de línea. Se especifican parámetros como la resolución de rho y theta, el umbral de detección, la longitud mínima de línea (minLineLength), y la distancia máxima entre puntos para que se consideren parte de la misma línea (maxLineGap).
- Las líneas detectadas se iteran para calcular su pendiente. Se clasifican como izquierda o derecha basándose en la pendiente (negativa para líneas a la izquierda, positiva para líneas a la derecha) y su posición horizontal en la imagen (la mitad izquierda para las líneas izquierdas, la mitad derecha para las derechas). Las líneas con pendiente muy baja (cercanas a horizontal) se descartan.

7. Promediado de líneas y lineas_detectadas.jpg:

Proceso:

- La función promediar_lineas toma un conjunto de líneas (izquierda o derecha) y utiliza np.polyfit para ajustar un polinomio de grado 1 (una línea recta) a todos los puntos (y, x) de esas líneas.

- Luego, calcula los puntos (x1, y1, x2, y2) que definen esta línea promedio, extendiéndola desde la parte inferior de la imagen (altura) hasta un punto superior (aproximadamente altura x 0.6).
- Las líneas promediadas para el carril izquierdo (linea_izquierda) y derecho (linea_derecha) se dibujan en una imagen en negro (imagen_lineas) con un color verde oscuro (17, 170, 0) y un grosor de 4 píxeles usando cv2.line.
- Si guardar_imagenes es True, esta imagen con solo las líneas detectadas se guarda como lineas_detectadas.jpg.

Propósito:

Obtener una única representación robusta para cada línea del carril a partir de múltiples segmentos de línea detectados, mitigando el ruido y las interrupciones en la detección de bordes. lineas_detectadas.jpg muestra claramente el resultado de la detección de las líneas del carril sobre un fondo vacío.

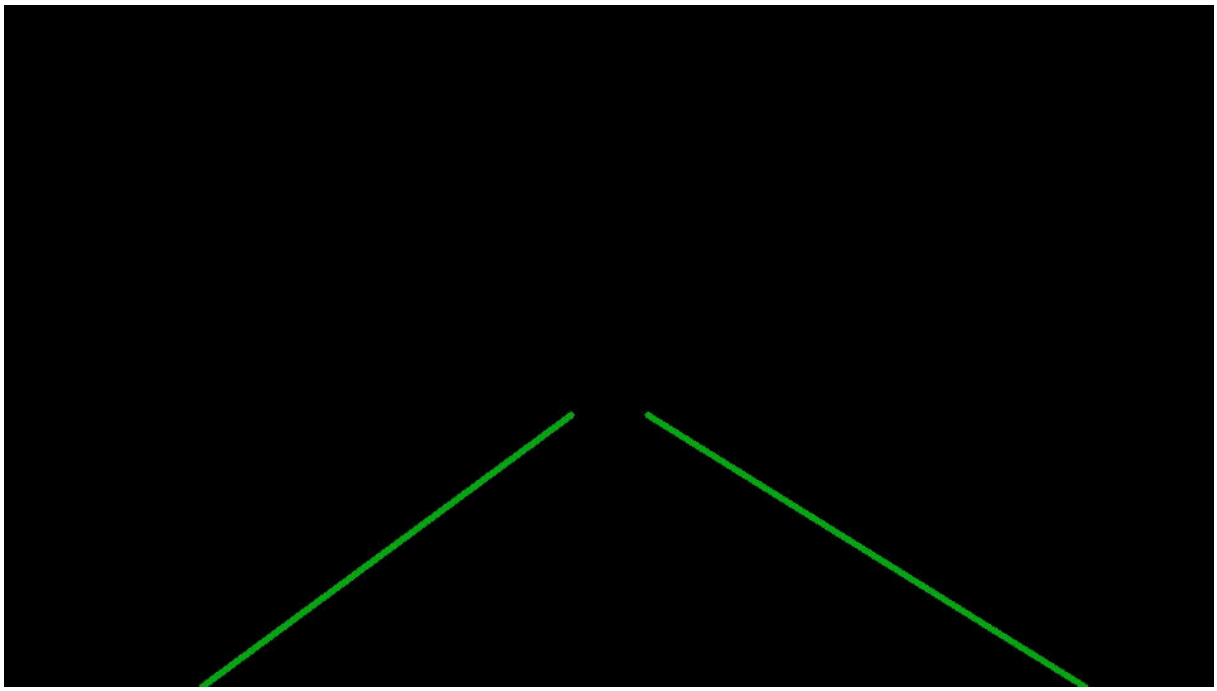

Figura 7: Lineas detectadas del primer frame de ruta_1

8. Combinación de resultados y resultado.jpg:

Proceso:

La imagen original (frame) se combina con la imagen que contiene las líneas detectadas (imagen_lineas) usando cv2.addWeighted(frame, 1, imagen_lineas, 1, 0). Esto

superpone las líneas sobre el frame original.

Propósito:

Generar el resultado final visual, donde las líneas detectadas se muestran claramente sobre la carretera en el video. Si guardar_imagenes es True, este resultado final se guarda como resultado.jpg.

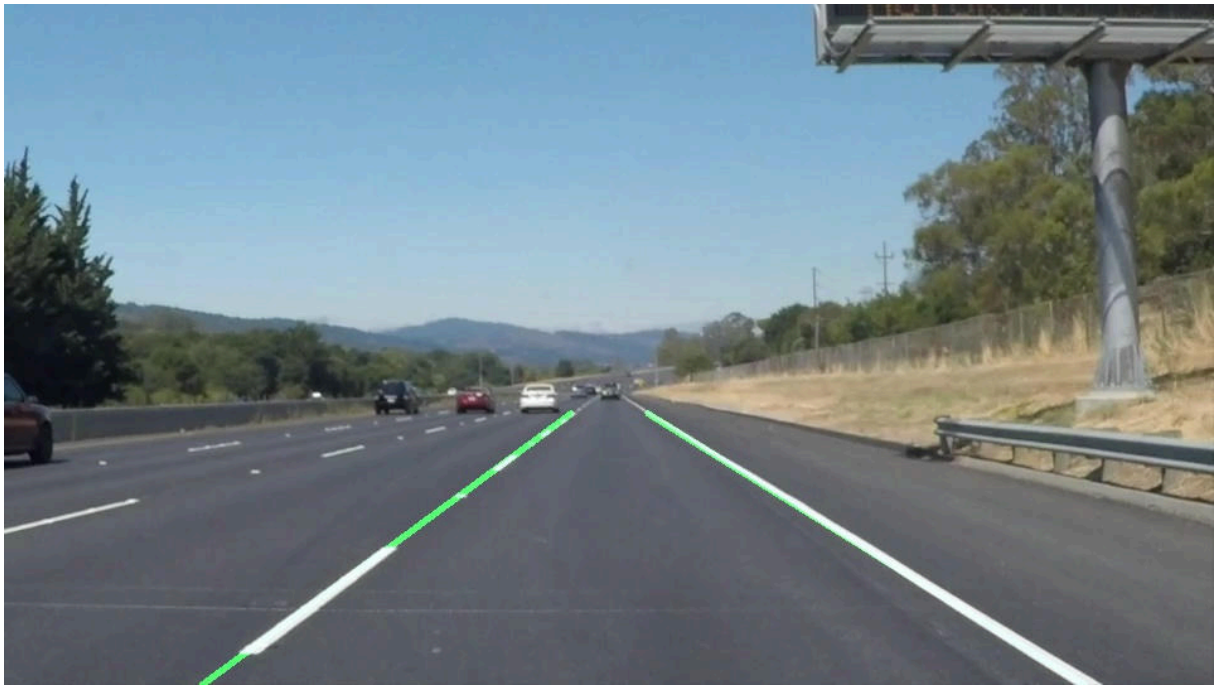

Figura 8: Resultado de la detección sobre el primer frame de ruta_1

Conclusiones

Para evaluar la capacidad de generalización del script de detección de carriles, se utilizaron videos adicionales, ruta_3.mp4 y ruta_4.mp4, que fueron grabados por nosotros.

En particular, el video ruta_3.mp4 presentaba un desafío adicional debido a su formato vertical y la inclusión de un cambio de carril durante la grabación. A pesar de estas condiciones que difieren de los videos iniciales (ruta_1.mp4 y ruta_2.mp4, los cuales fueron grabados con una cámara fija apuntando al frente), el desempeño del script fue muy satisfactorio, logrando detectar los carriles de manera adecuada incluso en un escenario más complejo.

Por otro lado, el video ruta_4.mp4 mantenía características similares a los videos originales ruta_1.mp4 y ruta_2.mp4. En este caso, el script demostró un rendimiento excelente, confirmando su robustez en condiciones de grabación esperadas.