

1 BT2 - Estimación robusta de parámetros

En esta sección tratamos la detección de rectas y otras formas geométricas características en el ámbito de las señales de tráfico, candidatas idóneas sobre el papel para ser detectadas mediante técnicas de estimación robusta de parámetros vistas en la asignatura. Comenzamos presentando el subconjunto de imágenes seleccionado para exemplificar los métodos que aquí explicamos, mostrando el resultado la aplicación del pipeline propuesto en la sección anterior para la detección de líneas de borde. Además, proponemos otro método de detección de bordes muy enfocado a este tipo de imágenes basado en la conversión del espacio de color RGB a HSV y la posterior umbralización del canal de saturación. A continuación, exponemos nuestra implementación en C para la detección de líneas y circunferencias con RANSAC y mostramos los resultados obtenidos. Finalmente, presentamos una serie de conclusiones y posibles líneas de trabajo futuras.

1.1 Dataset y preprocessamiento

Del conjunto de imágenes expuesto [1] hemos seleccionado las siguientes 11 imágenes:



Figure 1: Imágenes seleccionadas del conjunto ZOD para la detección de líneas de borde.

Como vemos, son imágenes en entornos urbanos con múltiples señales de tráfico. Incluimos una imagen con señales de obras (Imagen 4), una sin edificios (Imagen 7), una con hojas y muchas sombras (Imagen 8) y una nocturna (Imagen 10) para contar con cierta variedad.

A estas imágenes les aplicamos el siguiente preprocesamiento:

- Corrección de la distorsión de Kannala Brandt.
- Reescalado por un factor de 0'5.
- Aplicación de un filtro de medianas de 5x5 para reducir el ruido.

El escalado se ha llevado a cabo para agilizar la salidas en el notebook. Ha sido llevado a cabo mediante la función `cv2.resize()` de OpenCV con el método de interpolación `INTER_AREA`, ideal para reducciones. El resultado de este preprocesamiento se muestra en la fig. 2. Las dimensiones finales de las imágenes son de 1924x1084 píxeles.



Figure 2: Imágenes preprocesadas del conjunto ZOD para la detección de líneas de borde.

1.2 Detección de bordes

Para la detección de bordes hemos implementado dos métodos diferentes. El primero de ellos es el ya explicado en la sección anterior basado en la detección de bordes de Canny. La segunda aproximación es una segmentación por color seguida por una dilatación. En ambos casos, vamos a hacer una transformación previa de la imagen al espacio de color HSV.

1.2.1 Espacio de color HSV

El espacio de color HSV (Hue, Saturation, Value) [2] es un modelo de color que representa los colores de una manera más intuitiva que el espacio RGB. En este modelo cada canal representa:

- Hue (Matiz): es una escala circular que representa el tipo de color (rojo, verde, azul, etc.) y se mide en grados (0-360°), aunque en OpenCV se escala a un rango de 0 a 179.
- Saturation (Saturación): representa la intensidad del color, donde 0 es un tono de gris y 255 es el color completo.
- Value (Valor): representa el brillo del color. Es el clásico concepto de luminosidad, donde 0 es negro y 255 es el color más brillante.

Las señales de tráfico, al tener que ser fácilmente detectables por el ojo humano en diversas condiciones de iluminación y clima, se diseñan con colores brillantes y contrastantes. Esto las hace ideales para ser segmentadas en el espacio de color HSV, donde los cambios en la iluminación afectan menos a la percepción del color. Identificamos que todas las principales señales cuentan con el color rojo o el azul, incluso aquellas que encontramos en obras viales que sustituyen el color blanco por amarillo.

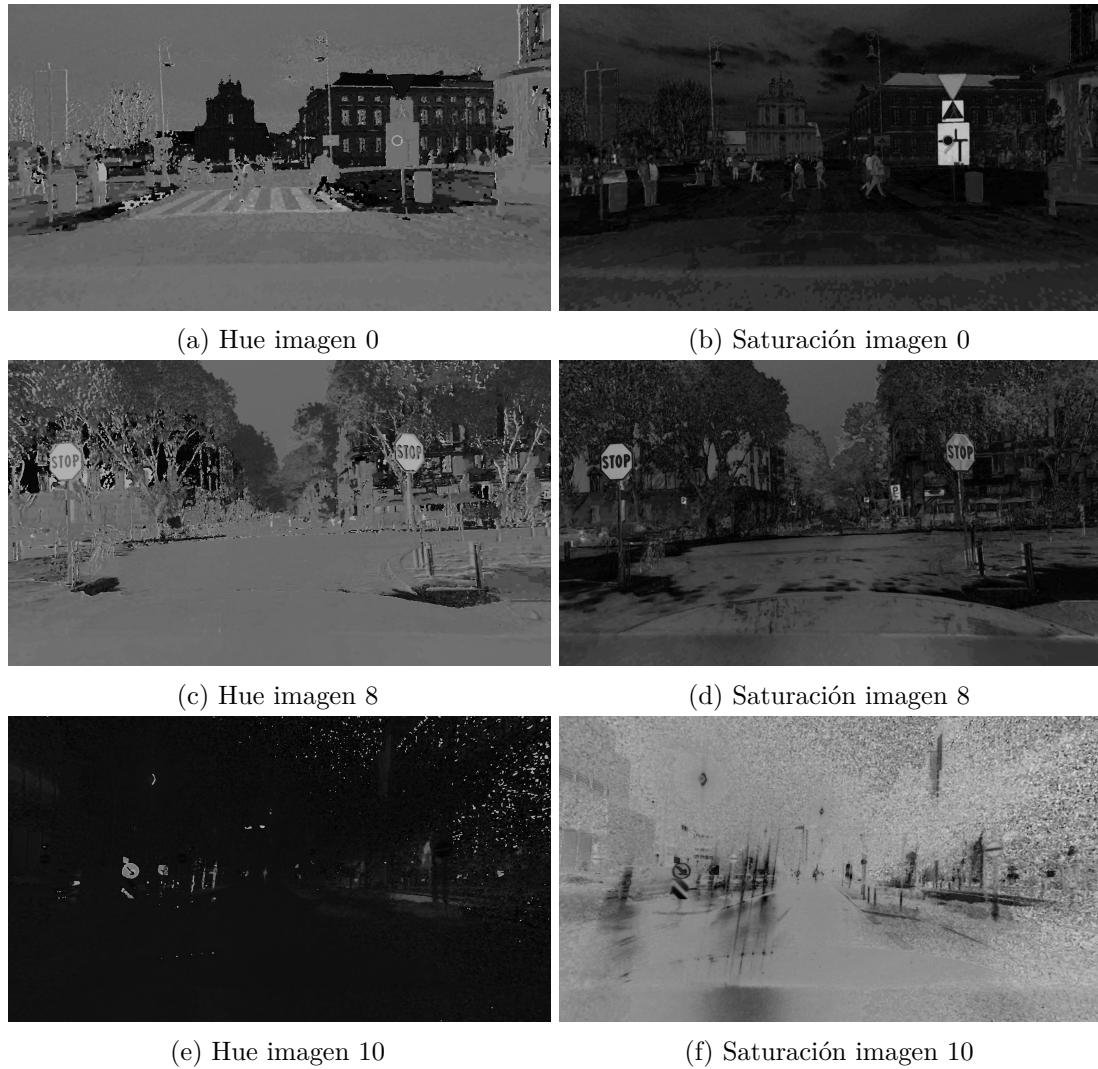


Figure 3: Canales de Hue y Saturación para las imágenes 0, 8 y 10.

En la escala de matiz, el rojo se corresponde con valores cercanos a 0 y el azul a 120. Como vemos en la fig. 3, el canal de saturación resalta mucho más las señales de tráfico que el canal de matiz. Es una excepción la imagen nocturna (Imagen 10), donde el canal de saturación apenas resalta las señales debido a la baja iluminación y a la presencia de luces artificiales que

distorsionan los colores. El canal de matiz, por otro lado, es prometedor en la imagen 10 al resaltar las señales de la zona propiamente iluminada ya que estas están fabricadas con un material que permite reflejar la luz de manera más efectiva durante la noche.

1.2.2 Operador de Canny en espacio HSV

Aplicando el pipeline de detección de bordes basado en Canny explicado en la sección anterior, pero esta vez en el espacio HSV, acabamos concluyendo que una ponderación de los canales H y S con pesos 0.25 y 0.75 respectivamente daba los mejores bordes. El proceso completo es el siguiente:

1. Convertir la imagen de RGB a HSV.
2. Calcular la imagen ponderada: $I_{weighted} = 0.25 \cdot I_H + 0.75 \cdot I_S$.
3. Suavizar $I_{weighted}$ con un filtro Gaussiano de $\sigma = 0.4$.
4. Aplicar el operador de Canny a $I_{weighted}$ con umbrales 150 y 250.

El resultado de este proceso se muestra en la fig. 4.

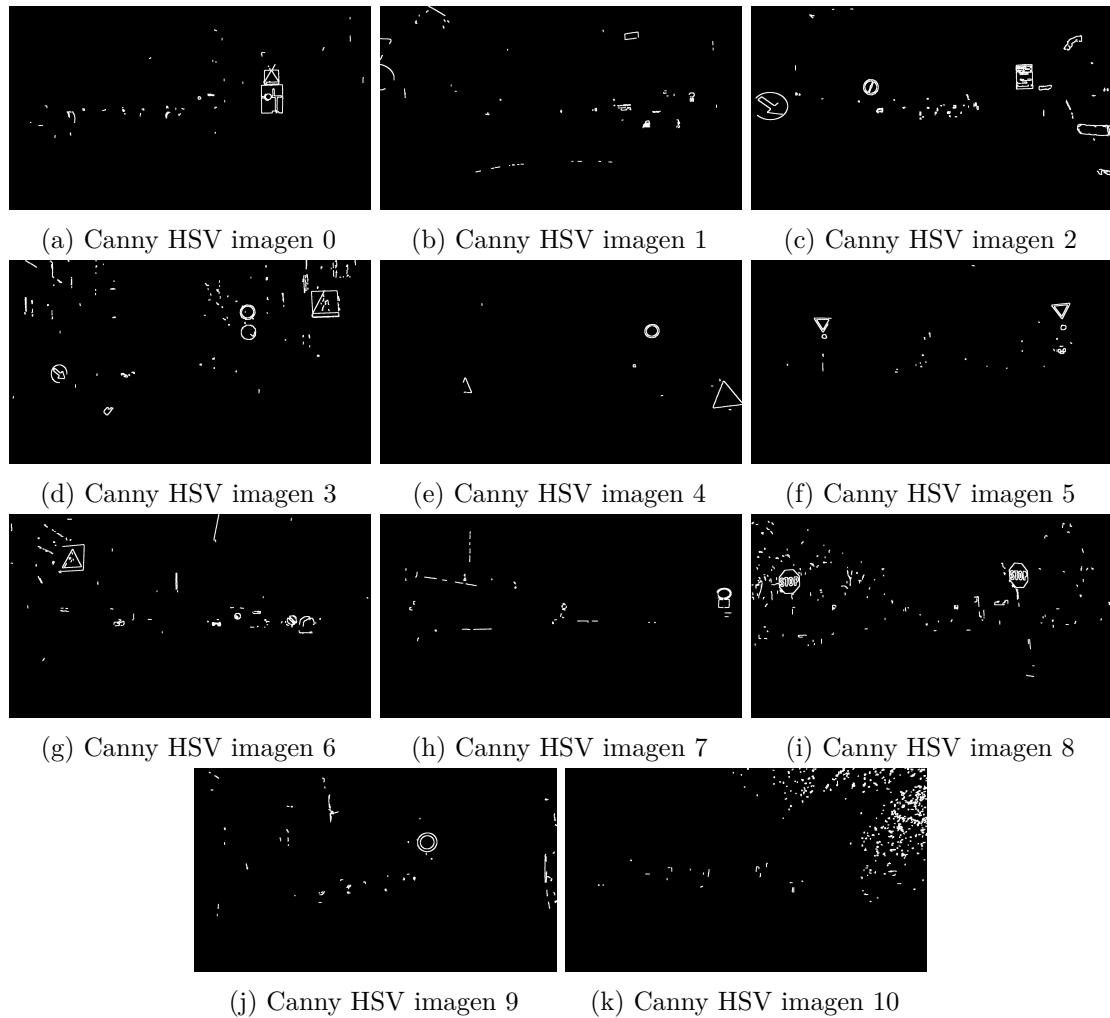


Figure 4: Resultados del operador de Canny en espacio HSV para las imágenes seleccionadas. Los bordes se muestran dilatados para mayor visibilidad.

1.2.3 Segmentación por color y dilatación

El segundo método de detección de bordes que hemos implementado se basa en un segmentación por color en el espacio HSV. Después de identificar que las señales de tráfico se caracterizan por colores azules y rojos muy saturados, hemos optado por umbralizar segmentar la imagen a partir de los rangos de matiz [165, 180] y [0, 15] para el rojo y [90, 130] para el azul. Además, aplicamos una umbralización en el canal de saturación para eliminar colores poco saturados, estableciendo un umbral en 90. El proceso completo es:

1. Convertir la imagen de RGB a HSV.
2. Crear una máscara binaria donde se cumplan las siguientes condiciones:
 - El canal de saturación sea mayor que 90.
 - El canal de matiz esté en los rangos [165, 180] o [0, 15] (rojo) o [90, 130] (azul).
3. Eliminar las regiones pequeñas mediante un filtrado de componentes conexas con un umbral de 50 píxeles.
4. Aplicar una dilatación circular de un pixel de grosor y sustraer la máscara original para obtener los bordes.

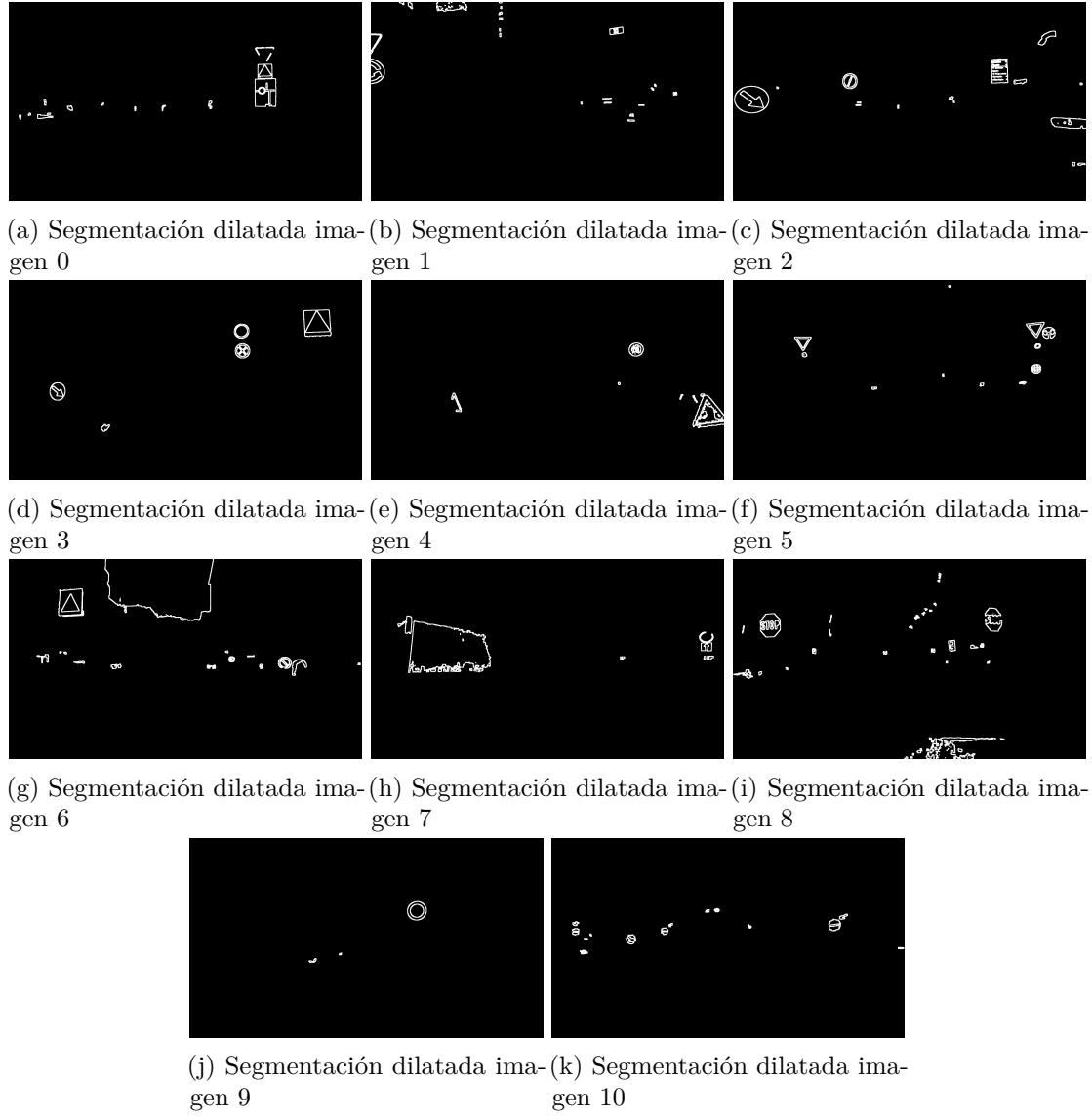


Figure 5: Resultados de la segmentación por color y dilatación para las imágenes seleccionadas. Los bordes se muestran dilatados para mayor visibilidad.

1.3 Comparación de resultados

Aunque ambos métodos ofrecen resultados aceptables, hemos observado que la segmentación por color y dilatación tiende a ser más robusta en condiciones de iluminación variables y en presencia de ruido. Lo que nos hace decantarnos por este método es el buen funcionamiento en la imagen nocturna (Imagen 10), donde el operador de Canny en espacio HSV no logra detectar los bordes de las señales de tráfico debido a la baja iluminación y al ruido introducido por las luces artificiales. Esta segunda técnica también parece generar menos falsos positivos en ciertas imágenes con fondos complejos.

1.4 Detección de features con RANSAC

Para la detección de features en imágenes hemos creado una clase en Python llamada `FeatureExtractor`. En particular, la clase sirve de interfaz para la detección de líneas, segmentos y circunferencias mediante las ideas de RANSAC.

1.4.1 Detección de rectas

Una vez contamos con una imagen de bordes, la propuesta de RANSAC es la siguiente:

1. Seleccionar aleatoriamente dos puntos de borde para definir una línea candidata.
2. Calcular la distancia perpendicular de todos los puntos de borde a esta línea.
3. Contar el número de inliers, es decir, puntos cuya distancia a la línea es menor que un umbral predefinido.
4. Repetir los pasos 1-3 un número fijo de veces o hasta que se alcance un número suficiente de inliers.
5. Seleccionar la línea con el mayor número de inliers como la mejor estimación.
6. Opcionalmente, refinar la línea utilizando todos los inliers mediante un ajuste por mínimos cuadrados.

Nosotros hemos decidido implementar nuestra versión en C, la cual puede ser consultada en `src/dgst/ffi/ransac.c` bajo el nombre de `ransac_line_fitting`. El procedimiento implementado es el recién descrito. Los parámetros que necesita son:

- input: Puntero a un array de puntos de borde.
- width, height: Dimensiones de la imagen.
- distance_threshold: Umbral de distancia para considerar un punto como inlier.
- max_iterations: Número de iteraciones de RANSAC.
- max_lsq_iterations: Número de iteraciones para el ajuste por mínimos cuadrados.
- min_inlier_count: Número mínimo de inliers para aceptar una línea.

El resultado es una recta representada en forma $ax + by + c = 0$ mediante los parámetros a, b y c (en caso de que se haya encontrado alguna línea). Adicionalmente, como funcionalidad en python incluimos la posibilidad de eliminar los inliers de la recta detectada para facilitar la detección de otras líneas en iteraciones posteriores.

Aplicando este método a nuestras imágenes de bordes exigiendo un mínimo de 200 inliers en la primera iteración y un mínimo de 50 tras aplicar RANSAC hasta 150 veces obtenemos imágenes como las mostradas en la fig. 7.

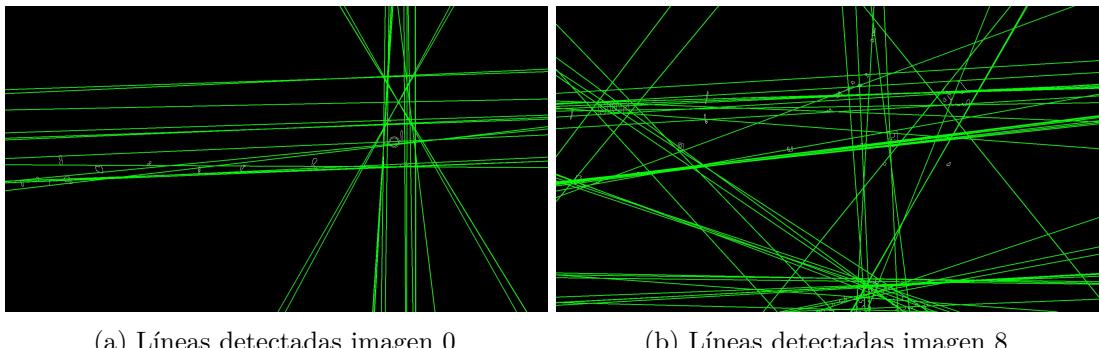


Figure 6: Resultados de la detección de líneas mediante RANSAC en las imágenes 0 y 8.

El algoritmo funciona bien detectando las rectas, pero resulta ser muy sensible al ruido. En cuanto hay una cantidad no demasiado alta de nubes de puntos alineados las rectas encontradas

se ajustan a este ruido. Para solucionar esto, podríamos aumentar el número mínimo de inliers requerido, pero a costa de sacrificar algunas rectas más pequeñas.

Debido a que las señales de tráfico ocupan una sección relativamente pequeña de la imagen, proponemos aplicar RANSAC mediante un algoritmo de ventana deslizante. De esta forma, permitimos eliminar esas rectas que se ajustan al ruido y detectar bordes más pequeños al poder reducir el número mínimo de inliers necesarios para determinar una recta. Para ello, utilizamos un nuevo método extractor de características llamado `windowed_ransac_line_fitting`.

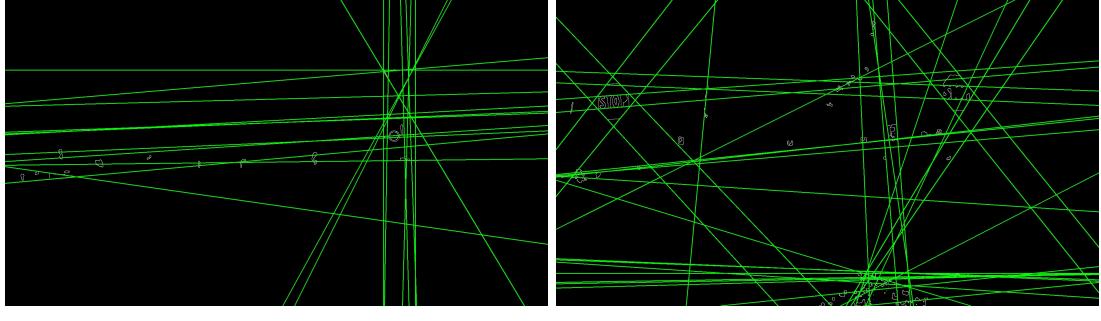


Figure 7: Resultados de la detección de líneas mediante RANSAC en ventana deslizante en las imágenes 0 y 8.

Como vemos, a pesar de utilizar en este caso un umbral de detección de inliers mayor (1.4 frente a 0.7 en el caso anterior) y un requerimiento menor de inliers por iteración, el método de ventana deslizante consigue detectar las rectas relevantes de la imagen sin verse tan afectado por el ruido.

1.4.2 Detección de segmentos

Otro problema de la baja ocupancia de las señales es que las rectas cubren innecesariamente toda la imagen. Sería ideal limitar la expresión de las mismas a los segmentos donde realmente hay señales. Es por esto que proponemos un método para inferir segmentos a partir de las rectas detectadas. A estos segmentos nosotros los llamamos 'soporte'. El procedimiento, implementado mediante el método `get_line_support` de la clase FeatureExtractor, es el siguiente:

1. Se proporciona una recta en forma $ax + by + c = 0$, desactivando la eliminación de inliers tras la detección.
2. Se reobtienen los inliers de la línea en la imagen de bordes.
3. Los inliers se proyectan ortogonalmente sobre la recta para obtener sus coordenadas 1D.
4. Se calcula la densidad de inliers a lo largo de la línea.
5. Se identifica el segmento maximal con densidad que supere un umbral dado.

Para llevar a cabo el último paso, hemos utilizado una versión adaptada del Algoritmo de Kadane [kadane], un algoritmo de programación dinámica que resuelve el problema **Maximum Subarray Problem** de encontrar un subarray de suma máxima. Los pasos de esta adaptación son, partiendo de una serie de puntos en 1D (proyecciones sobre la recta):

1. Se establece un umbral de densidad μ (número de inliers por unidad de longitud).
2. Se construye un array auxiliar A , de forma que $A[i]$ representa la contribución de densidad del i -ésimo intervalo $A_i = 1 - \mu \times (p_{i+1} - p_i)$

3. Se inicializan las variables:

- `max_sum = 0`
- `current_sum = 0`
- `start_index = 0`
- `best_start = -1`
- `best_end = -1`

4. Se recorren los elementos del array A guardando las sumas acumuladas:

- `current_sum += A[i]`
- Si `current_sum > max_sum`:
 - `max_sum = current_sum`
 - `best_start = start_index`
 - `best_end = i`
- Si `current_sum < 0`:
 - `current_sum = 0`
 - `start_index = i + 1`

5. Se devuelve el segmento `[best_start, best_end]`.

Una de las bondades de este algoritmo es que funciona en tiempo lineal con respecto al número de inliers. Podemos verlo en acción sobre una región de la imagen 0 en la fig. 8.

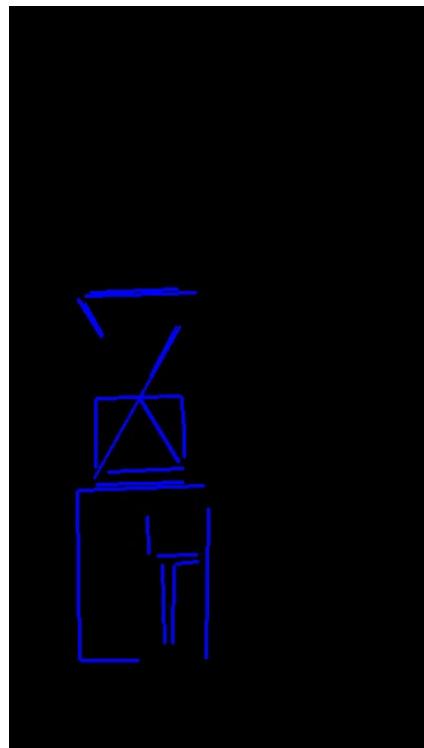


Figure 8: Resultados de la detección de segmentos mediante RANSAC en las imagen 0.

Los segmentos azules representan los soportes detectados. Al igual que hacíamos con las rectas, permitimos eliminar los inliers de los segmentos detectados para facilitar la detección de otros segmentos en iteraciones posteriores. El método permite incluso borrar los inliers de

una recta para la que no se haya detectado soporte, lo que resulta muy útil para eliminar rectas que se ajustan al ruido. Incluso podemos especificar una longitud mínima para el soporte y una umbral de detección de inliers diferente a la de la recta para mayor flexibilidad.

1.4.3 Detección de circunferencias

La última característica de bordes que exploramos es la detección de circunferencias, ya que algunas señales de tráfico son circulares. El procedimiento es similar al de las rectas, pero en este caso seleccionamos tres puntos aleatoriamente para definir una circunferencia candidata. El resto de pasos son análogos a los descritos para las rectas. Hemos implementado este método en C bajo el nombre `ransac_circle_fitting` en el módulo `src/dgst/filters/ffi/ransac.c`. Los parámetros que necesita son:

- input: Puntero a un array de puntos de borde.
- width, height: Dimensiones de la imagen.
- distance_threshold: Umbral de distancia para considerar un punto como inlier.
- max_iterations: Número de iteraciones de RANSAC.
- min_inlier_ratio: Proporción mínima de inliers respecto a la longitud del radio para aceptar una circunferencia.
- min_radius, max_radius: Radios mínimo y máximo permitidos para las circunferencias.

Sobre este procedimiento, también resulta necesario acotar el radio máximo y mínimo de las circunferencias a detectar para así evitar encontrar circunferencias extremadamente pequeñas o tratar rectas como circunferencias de radio muy grande. Si bien en el caso de las rectas establecíamos un número máximo de inliers, en el caso de las circunferencias resulta más apropiado establecer una proporción mínima de inliers con respecto al radio de la circunferencia, pues un match de radio muy grande puede tener muchos inliers sin que realmente represente una circunferencia en la imagen. Como orientación para este valor, el "número pi" para circunferencias discretas es aproximadamente $\sqrt{8}$. La prueba de ello puede ser un ejercicio sencillo para el lector. Por tanto, un valor razonable para la detección de circunferencias con este método debería ser aproximadamente $2\pi_d r / r = 2\pi_d \approx 5.6$. Esto suponiendo que el umbral de distancia para considerar un inlier es pequeño.

Aplicando este método a nuestras imágenes de bordes con un umbral de distancia de 1 píxel, 10000 iteraciones y una proporción mínima de inliers de 4 obtenemos imágenes como las mostradas en la fig. 9.

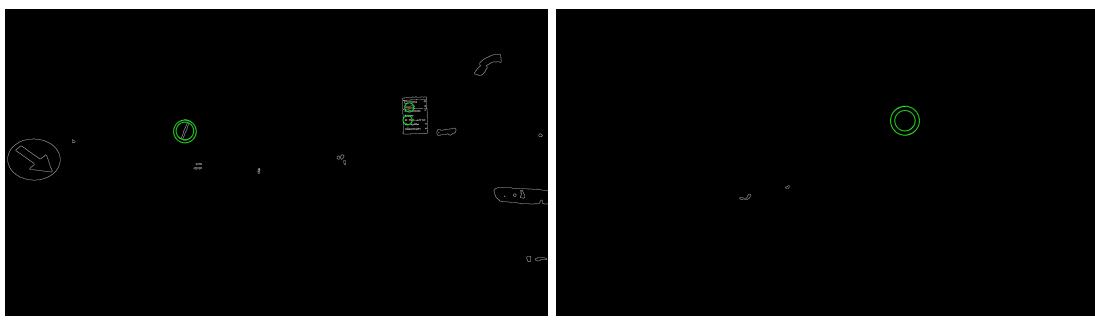


Figure 9: Resultados de la detección de circunferencias mediante RANSAC en las imágenes 2 y 9.

Muchas de las circunferencias de nuestro dataset son detectadas. No obstante, queda en evidencia que este método es muy sensible a la perspectiva ya que no se logra captar las señales cuya normal se encuentra inclinada con respecto a la cámara (aparecen como elipses). Sería una adición interesante la detección de este tipo de objetos y se plantea como una ampliación futura de este trabajo.

1.4.4 Puesta en común de los métodos

Finalmente, proponemos un pipeline que combina todos los métodos anteriores para este tipo de imágenes:

1. Preprocesamiento de la imagen (corrección de distorsión, reescalado, filtro de medianas).
2. Detección de bordes mediante segmentación por color y dilatación.
3. Detección de líneas mediante RANSAC en ventana deslizante sin eliminación de inliers.
4. Detección de segmentos para cada línea detectada.
5. Detección de circunferencias.
6. Repetir los pasos 3-5 relajando los criterios de detección para encontrar más características.

Podemos ver los resultados de este pipeline en la fig. 10.

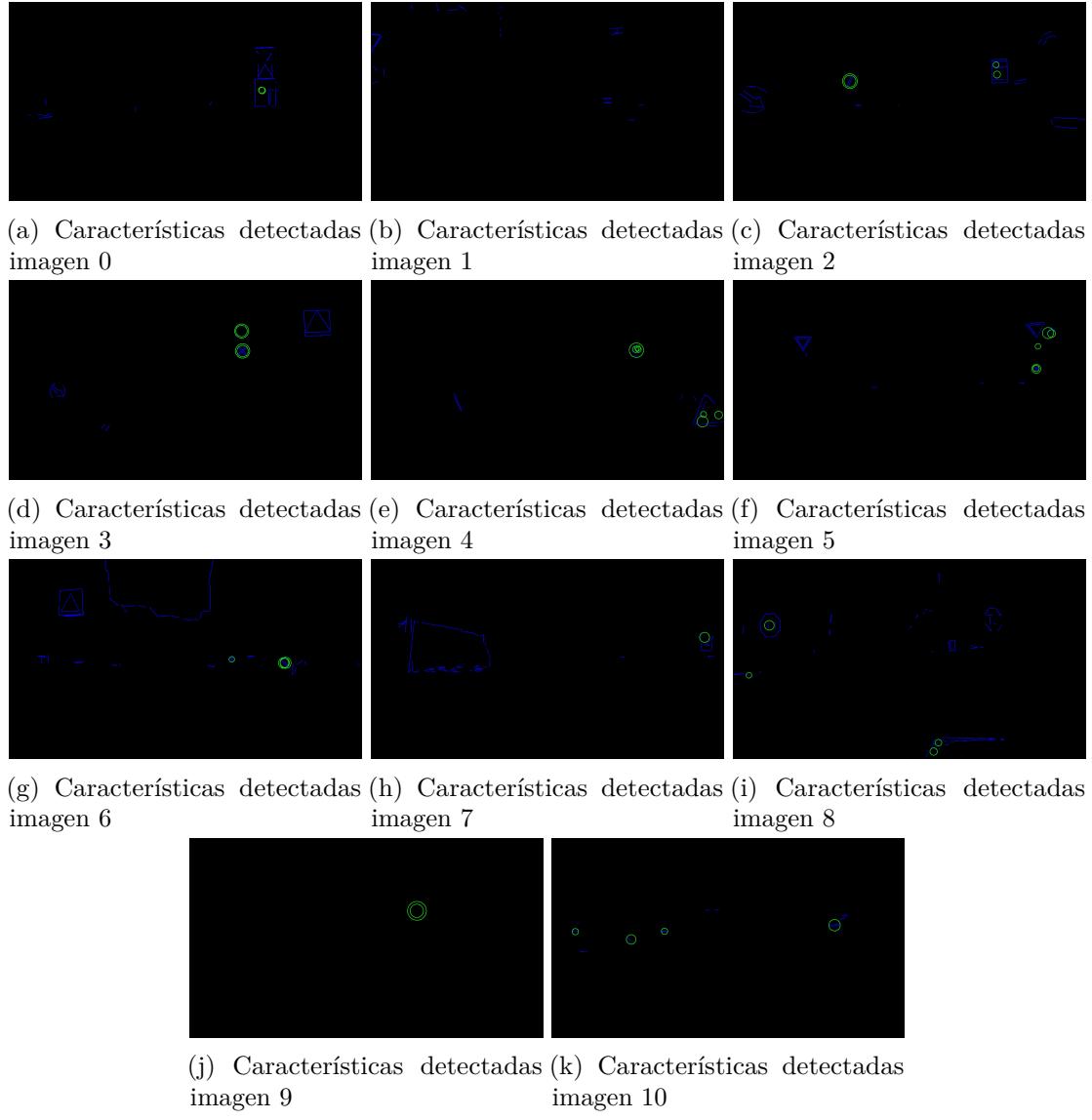


Figure 10: Resultados del pipeline completo de detección de características.

1.5 Análisis de RANSAC para la detección de rectas

En los apartados anteriores estableciamos los parámetros para RANSAC de forma manual para conseguir resultados aceptables. Esta sección, sin afán de ser un análisis exhaustivo ni completo pues la convergencia de estos métodos depende en gran medida en los elementos que se puedan encontrar en cada imagen, puede orientarnos a la hora de calibrar los parámetros en un trabajo futuro. En concreto, vamos a centrarnos en el algoritmo de detección de rectas.

1.5.1 Análisis de rendimiento

En primer lugar vamos a analizar el tiempo de ejecución del algoritmo. Sabemos que su complejidad temporal teórica es $O(it \times k)$ donde it es el número de iteraciones y k el número de píxeles de borde. Para comprobarlo, hemos medido el tiempo de ejecución en imágenes 1000x1000 con diferentes cantidades de píxeles de borde y hemos obtenido las siguientes gráficas:

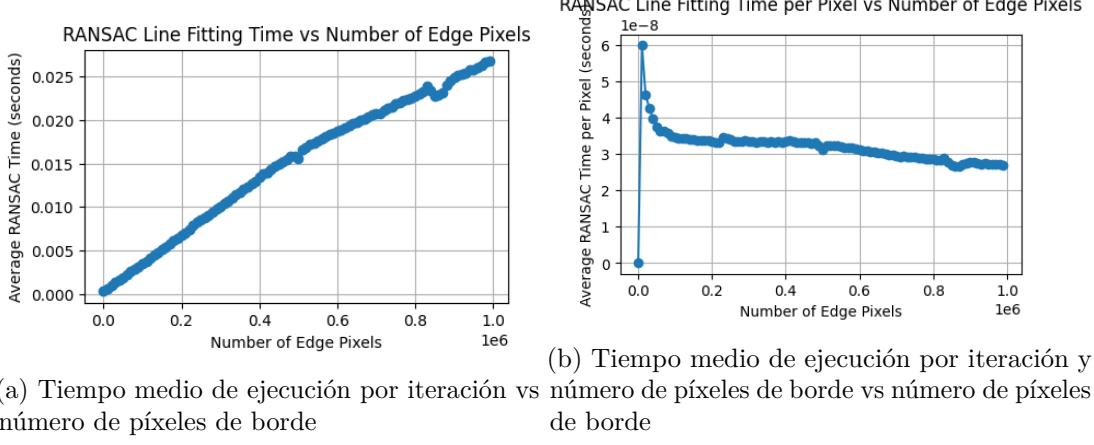


Figure 11: Análisis del tiempo de ejecución del algoritmo RANSAC para la detección de rectas.

Resulta evidente la relación lineal entre el número de píxeles de borde y el tiempo de ejecución por iteración, confirmando la complejidad teórica del algoritmo. En una ejecución monohilo en un equipo con procesador AMD Ryzen 7 5800H, el tiempo medio por iteración y píxel de borde es de aproximadamente 0.03 microsegundos.

1.5.2 Análisis de convergencia

Estudiar cuánto tarda en converger el algoritmo es algo más complicado ya que depende en mayor medida de los elementos que puedan estar presentes en la imagen. Para poder concluir cuántas iteraciones son necesarias para detectar una recta vamos a hacer lo siguiente:

1. Establecer un valor inicial de max_iterations.
2. Ejecutar RANSAC 100 veces.
3. Almacenar la tasa de éxito (número de veces que se detecta una recta con al menos 150 inliers).
4. Aumentar max_iterations en un 10%.
5. Volver al paso 2.

Podemos repetir este procedimiento hasta obtener una tasa de éxito superior al 90% o hasta superar las 1000 iteraciones.

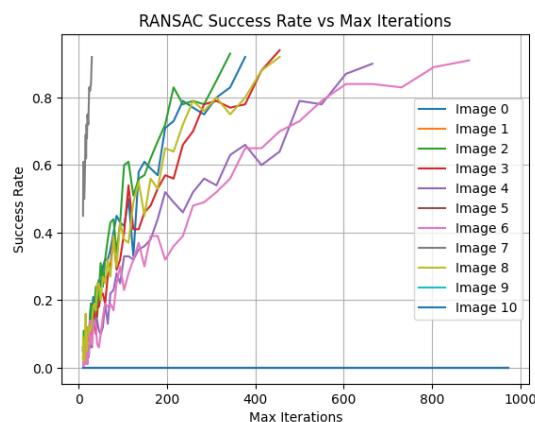


Figure 12: Convergencia del algoritmo RANSAC para la detección de rectas.

Buscando una recta con 150 inliers, que para los rangos en los que estamos trabajando se considera un número alto, vemos como en las imágenes 7, 2, 8, 0 y 3 ransac encuentra dicha recta el 80% de las veces en menos de 400 iteraciones. Para los casos en los que no existe dicha recta, como en la imagen 1 o la imagen 10, la gráfica se muestra como se esperaba.

En cualquier caso, en todas las imágenes en las que sí existe dicha recta parece encontrarse un 20% de las ocasiones en las 100 primeras iteraciones. Este hecho nos va a ayudar a establecer un número mínimo de iteraciones necesario para detectar una recta.

Sea X la variable aleatoria que modela la cantidad de iteraciones necesarias para detectar una recta que cumpla las condiciones en el caso de que exista. X sigue entonces una distribución binomial con probabilidad de éxito de $p = 20\%/100 = 2 \times 10^{-3}$. Entonces: $P(X \leq n) = 0.95 \Leftrightarrow 1 - (1 - p)^n > 0.95 \Leftrightarrow (1 - 2 \times 10^{-3})^n > 0.05 \Leftrightarrow n \ln(1 - 2 \times 10^{-3}) > \ln 0.05 \Leftrightarrow n > 1497$.

En conclusión, estableciendo el máximo de número de ejecuciones a 1500, en caso de que exista una recta que cumpla las condiciones que buscamos, será encontrada el 95% de las veces. Para imágenes como las vistas en este documento, que cuentan con no más de 10000 píxeles de borde, y con el análisis previo podemos establecer un tiempo estimado para encontrar una recta de:

$$3 \times 10^{-8} \frac{s}{it} \times 1500 \frac{it}{recta} = 4.5 \times 10^{-5} \frac{s}{recta}$$

Por tanto, el tiempo de cómputo estimado por recta es de 0.045 milisegundos, un tiempo más que asumible teniendo en cuenta las condiciones planteadas. Para usos posteriores se propone:

1. Comenzar con un número de inliers excesivo.
2. Aplicar RANSAC con 1500 iteraciones.
3. Si no se encuentra una recta, decrementar el número de inliers.
4. Repetir 2 hasta encontrar una recta o llegar a un número mínimo de inliers.

Aun comenzando con 300 inliers (lo que correspondería a más o menos un tercio de la longitud de la imagen con la que trabajamos), estableciendo un decrecimiento de 2 inliers por iteración y suponiendo que encontramos de media una recta por iteración, hablamos un tiempo de cómputo de unos 6 milisegundos. Un tiempo corto aún procesando video en directo.