# Proyecto de Sistema ADAS (Asistente de Conducción)

## Table of Contents

# 1. Introducción

Este proyecto tiene como objetivo desarrollar un sistema avanzado de asistencia al conductor (ADAS) capaz de integrar diversas fuentes de datos (sensores CAN, protocolo 485, visión computacional) y aplicar reglas a través de un sistema experto para mejorar la seguridad en la conducción.

El sistema se compone de varios módulos clave que permiten la recepción, procesamiento y análisis de datos en tiempo real para tomar decisiones automáticas en función de los hechos observados, como alertas de colisión o correcciones automáticas en el comportamiento del vehículo.

# 2. Descripción del Sistema

El sistema ADAS está compuesto por los siguientes módulos:

## 2.1. Módulo 1: Comunicación

- **Función**: Responsable de la conexión y transmisión de datos del vehículo a través de la red.
- **Interacciones**:
    - Envío de datos del **M5** y comunicación en tiempo real con los servicios en la nube.
    - Transferencia de comandos al módulo **M5**.

## 2.2. Módulo 2: M5 (Módulo Microcontrolador)

- **Función**: Recibe los datos del GPS, CAN, etc., y gestiona la comunicación entre el vehículo y el servidor central.
- **Interacciones**:
    - Envía los datos al **Servidor ADA**.
    - Recibe datos en tiempo real (RT) y los envía al sistema para su procesamiento.
    - Ejecuta acciones en tiempo real (como alertas) basadas en la evaluación de los datos procesados.
    - Envía información de localización a través del sistema **AVL**.

## 2.3. Módulo 3: Servidor ADA

- **Función**: Es el componente principal que coordina la recepción y el envío de datos, integrando todos los sensores y módulos del sistema.
- **Interacciones**:
    - Recibe datos del **M5**.
    - Procesa los datos en tiempo real (RT) y los envía a los módulos correspondientes para ejecutar acciones.
    - Recibe los datos del sistema de **Procesamiento de Video**.
    - Envía y recibe datos del **Sistema Experto**.
    - Transmite información de retorno para activar respuestas automáticas en tiempo real.

## 2.4. Módulo 4: DVR (Grabadora de Video Digital)

- **Función**: Captura el video en tiempo real del entorno del vehículo.
- **Interacciones**:
  - Envía la transmisión de video a través de RTSP al módulo de **Procesamiento de Video**.

## 2.5. Módulo 5: Procesamiento de Video

- **Función**: Gestiona y distribuye el stream de video utilizando **GStreamer** como servidor de video.

- **Interacciones**:

  - Recibe el stream de video desde el **DVR**.
  - Acondiciona el video y lo distribuye a través de **GStreamer**, creando un **servidor de video** accesible para otros módulos.
    - **M6** (Predicción de Colisiones)
    - **M7** (Detección de Señales de Tráfico)
    - **M8** (Reconocimiento de Carriles)
    - **M9** (Detección de Fatiga del Conductor)

  se conectan al stream de video proporcionado por **GStreamer**, permitiendo que cada uno procese el video de manera simultánea y en su propio dominio.

## 2.6. Módulo 6: Predicción de Colisiones (Collision Detection)

- **Función**: Calcula la distancia y trayectoria mediante un sistema de aproximación basado en la retícula de telémetro.
- **Interacciones**:
  - Recibe datos del módulo de **Procesamiento de Video**.
  - Calcula la distancia y trayectoria entre los objetos detectados y el vehículo.
  - Envía los resultados de la detección y la distancia calculada al **Sistema Experto** para su evaluación y toma de decisiones.

## 2.7. Módulo 7: Detección de Señales de Tráfico y Límite de Velocidad

- **Función**: Detección de señales de tráfico y límites de velocidad
- **Interacciones**:
  - Recibe datos del módulo de **Procesamiento de Video**.
  - Identifica señales de tráfico y límites de velocidad
  - Envía los resultados de la detección al **Sistema Experto** para su evaluación y toma de decisiones.

## 2.8. Módulo 8: Reconocimiento de Carril (Lane Detection)

- **Función**: Detectar los carriles
- **Interacciones**:
  - Recibe datos del módulo de **Procesamiento de Video**.
  - Identifica señales de tráfico y límites de velocidad
  - Envía los resultados de la detección al **Sistema Experto** para su evaluación y toma de decisiones.

## 2.9. Módulo 9: Detección de Fatiga del Conductor

- **Función**: Detectar signos de fatiga o distracción del conductor
- **Interacciones**:
  - Recibe datos del módulo de **Procesamiento de Video**.
  - Identifica el parpadeo excesivo o el movimiento de la cabeza hacia abajo o a los lados
  - Envía los resultados de la detección al **Sistema Experto** para su evaluación y toma de decisiones.

### 2.10. Módulo 10: Sistema Experto

- **Función**: Aplica reglas lógicas para evaluar los datos recibidos y tomar decisiones automáticas con base en los hechos detectados.
- **Interacciones**:
  - Recibe datos de diversos módulos (**Servidor ADA**, **Procesamiento de Video**, etc.).
  - Evalúa los hechos utilizando LISA (sistema de inferencia basado en reglas).
  - Toma decisiones y las envía al módulo de **Acciones**.

### 2.11. Módulo 11: Sistema de Acciones

- **Función**: Ejecuta acciones automáticas, como alertas sonoras o frenado automático.
- **Interacciones**:
  - Recibe decisiones del **Sistema Experto**.
  - Ejecuta las acciones necesarias en tiempo real.

### 2.12. Módulo 12: HMI

### 2.13. Módulo 13: Modelo Estado del camino

## 3. Flujo de Datos

1. **Adquisición de Datos desde los Sensores**: El sistema comienza con la recopilación de datos desde varias fuentes a bordo del vehículo. Estos incluyen sensores CAN, datos del GPS, y la transmisión en tiempo real desde el **DVR** (grabadora de video digital).
2. **Transmisión al Módulo M5**: Todos los datos recolectados son enviados al módulo **M5**. Este actúa como intermediario, gestionando la transmisión de los datos hacia otros módulos del sistema. El **M5** envía esta información al **Servidor ADA** para su procesamiento.
3. **Recepción y Procesamiento en el Servidor ADA**: El **Servidor ADA** recibe los datos provenientes del **M5**. En este punto, el servidor integra toda la información y la distribuye a los módulos responsables del análisis, como el módulo de **Procesamiento de Video** y el **Sistema Experto**.

1. **Procesamiento del Video (Distribución de Fotogramas con GStreamer)**: La transmisión de video desde el **DVR** es gestionada por el módulo de **Procesamiento de Video**, que utiliza **GStreamer** para configurar un **servidor de video**. La función principal de este módulo es distribuir el stream de video en tiempo real a los módulos especializados (**M6**, **M7**, **M8**, y **M9**) de manera simultánea.

   Cada uno de estos módulos se conecta al servidor **GStreamer** y procesa el mismo stream de video en su área específica, sin necesidad de duplicar el procesamiento de fotogramas.

   - **M6** utiliza YOLOv8 especializado para la predicción de colisiones, identificando vehículos y objetos cercanos.
   - **M7** emplea YOLOv8 para la detección de señales de tráfico y límites de velocidad.
   - **M8** usa YOLOv8 para el reconocimiento de carriles, asegurando una correcta identificación de las líneas de la carretera.

- **M9** se especializa en la detección de fatiga del conductor, analizando patrones faciales y movimientos del conductor.

Este enfoque, basado en **GStreamer**, permite que el mismo stream de video sea evaluado en tiempo real desde múltiples perspectivas, lo que garantiza que cada módulo procese el mismo fotograma dentro de su propio dominio particular de análisis, optimizando la toma de decisiones y el uso de los recursos de manera eficiente.

- **Ventaja**: Al usar **GStreamer** como servidor de video, se asegura que todos los módulos puedan conectarse de manera eficiente al mismo stream sin la necesidad de duplicar el procesamiento o paralelizar manualmente los fotogramas en el módulo de video. Esto optimiza la distribución de datos y mejora la capacidad de procesamiento del sistema en tiempo real.

1. **Evaluación en el Sistema Experto**: El **Sistema Experto** recibe y centraliza la información proveniente de los módulos especializados **M6** (Predicción de Colisiones), **M7** (Detección de Señales de Tráfico), **M8** (Reconocimiento de Carriles), y **M9** (Detección de Fatiga). Cada uno de estos módulos proporciona datos procesados que son enviados en tiempo real al **Sistema Experto**.

   Este sistema aplica un conjunto de **reglas lógicas predefinidas** para evaluar la situación en función de los hechos detectados, tales como la cercanía de un objeto al vehículo, la presencia de una señal de tráfico importante, o si el conductor muestra signos de fatiga. Las reglas se implementan utilizando un sistema de inferencia basado en **LISA** (Lisp-based Intelligent Software Agents), lo que permite una evaluación precisa y transparente de cada evento detectado.

   Una vez que el **Sistema Experto** ha procesado toda la información y evaluado los hechos, los resultados de esta evaluación (alertas de colisión, advertencias de señales, alertas de desvío de carril, o alertas de fatiga) son enviados al **Sistema de Acciones** para su ejecución en tiempo real.

2. **Ciclo en Tiempo Real**: El proceso completo ocurre de manera continua y en tiempo real, lo que permite al sistema ADAS reaccionar de forma rápida y precisa ante posibles situaciones de peligro, asegurando una conducción más segura.

# 4. Implementación

## 4.1. Ada como Servidor de Mensajes en Tiempo Real

- **Ada** se utiliza para gestionar la comunicación de mensajes entre los módulos del sistema ADAS en tiempo real, asegurando baja latencia y alta fiabilidad. Este servidor actúa como un middleware que coordina el flujo de datos desde el microcontrolador M5 hacia otros componentes críticos, como el sistema de acciones o el sistema experto.
- **Por qué es importante**: Ada es un lenguaje altamente confiable para aplicaciones en tiempo real, garantizando que los mensajes críticos, como alertas de colisión o comandos del sistema experto, sean procesados a tiempo.

## 4.2. GStreamer como Servicio Independiente

- **GStreamer** se implementa como un servicio independiente que proporciona un canal para la distribución eficiente del stream de video desde el DVR hacia los módulos especializados de procesamiento de video.
- Este servicio permite la transmisión de video en tiempo real, asegurando que los módulos de procesamiento accedan a los flujos de video sin interferencias.
- **Por qué es importante**: La integración de **GStreamer** como un servicio independiente permite que el video en tiempo real se distribuya de manera eficiente a todos los módulos necesarios para el análisis, optimizando el uso de recursos y mejorando la latencia del sistema.

### 4.3. Python + YOLO + OpenCV: Visión Computacional y Machine Learning

- El módulo de **Procesamiento de Video** utiliza **YOLOv8**, un modelo de machine learning optimizado para la **detección de objetos en tiempo real** (vehículos, peatones, señales de tráfico), junto con **OpenCV** para tareas de detección de carriles y procesamiento de imágenes.
- Los módulos acceden al servicio **GStreamer** para recibir los fotogramas desde el DVR y realizar análisis como la predicción de colisiones, detección de señales de tráfico, reconocimiento de carriles y detección de fatiga del conductor. Esto permite un análisis simultáneo en tiempo real.
- **Por qué es importante**: Esta combinación permite que el sistema ADAS detecte rápidamente los objetos y las características del entorno, proporcionando datos clave al sistema experto para la toma de decisiones automatizadas en tiempo real.

### 4.4. Common Lisp + LISA: IA Simbólica para el Sistema Experto

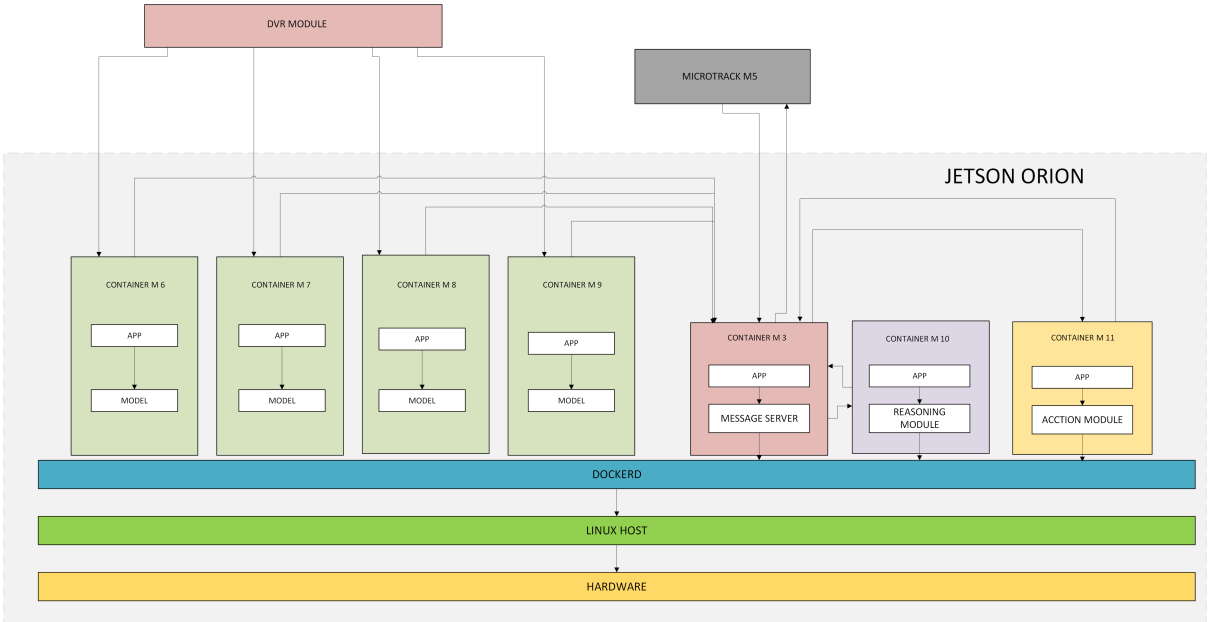
- **LISA** es un sistema basado en reglas escrito en **Common Lisp** que proporciona una **IA simbólica** para el sistema experto. Se utiliza para evaluar los hechos observados (como la distancia a un vehículo o la proximidad de un objeto) y tomar decisiones automáticas en función de reglas predefinidas.
- **Por qué es importante**: Al usar IA simbólica, el sistema experto puede ofrecer decisiones **transparentes** y fácilmente ajustables, lo que es esencial para entornos donde se necesitan reglas lógicas claras, como la conducción asistida.

### 4.5. Comunicación

- Tecnología: Sockets UNIX para la comunicación eficiente entre los módulos del sistema.
- La combinación de Sockets UNIX y el servicio independiente **GStreamer** garantiza una arquitectura de comunicaciones robusta y de baja latencia, permitiendo que los datos de video y los mensajes del sistema experto sean gestionados de manera eficiente.

## 5. Diagrama Simplificado del Sistema


# 6. Diagrama Arquitectura de Software




# 7. Anexo Anlisis de algoritmos y tecnologias

## 7.1. ANEXO Módulo 6: Mil-dot (Cálculo de Distancia)

### 7.1.1. Concepto de Optical Flow

El Optical Flow es una técnica poderosa para detectar y analizar el movimiento en una secuencia de imágenes o video. En lugar de analizar cada cuadro por separado, el flujo óptico calcula los cambios en la posición de los píxeles entre dos fotogramas consecutivos, lo que permite identificar el desplazamiento o el movimiento de los objetos en la escena.

El Optical Flow asume que la intensidad de un píxel permanece constante entre dos fotogramas consecutivos, pero puede desplazarse debido al movimiento de los objetos o de la cámara. Al calcular este desplazamiento, podemos estimar la velocidad y dirección del movimiento en cada parte de la imagen.

Los principales métodos de Optical Flow incluyen:

Método de Lucas-Kanade: Calcula el flujo en pequeños bloques (ventanas) de la imagen, asumiendo que el movimiento dentro de cada bloque es constante. Método de Horn-Schunck: Utiliza un enfoque basado en la minimización de una función de energía que combina restricciones de suavidad y coherencia de los vectores de movimiento. Farneback Optical Flow: Un método más moderno y eficiente que calcula un campo de movimiento denso (cada píxel tiene su vector de movimiento) utilizando la aproximación polinómica de los patrones de cambio. Aplicación con OpenCV En OpenCV, puedes calcular el flujo óptico denso (que calcula el movimiento de todos los píxeles) usando métodos como el de Farneback, y para el flujo óptico disperso (donde se sigue solo un conjunto seleccionado de puntos), puedes usar Lucas-Kanade.

### 7.1.2. Optical Flow denso (Farneback)

A continuación, un ejemplo de cómo aplicar el método Farneback Optical Flow para detectar movimiento en un video con OpenCV:

```
import cv2
import numpy as np

# Cargar el video
cap = cv2.VideoCapture('video.mp4')

# Leer el primer cuadro
ret, frame1 = cap.read()
prvs = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)

# Inicializar los parámetros del flujo óptico
while cap.isOpened():
    ret, frame2 = cap.read()
    if not ret:
        break

    next_frame = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)

    # Calcular el Optical Flow utilizando el método de Farneback
    flow = cv2.calcOpticalFlowFarneback(prvs, next_frame, None, 0.5, 3, 15, 3, 5, 1.2, 0)

    # Convertir el flujo en ángulos y magnitudes
    magnitude, angle = cv2.cartToPolar(flow[..., 0], flow[..., 1])

    # Crear una imagen en color basada en el ángulo y la magnitud del flujo óptico
    hsv = np.zeros_like(frame1)
    hsv[..., 1] = 255  # Saturación a 255
    hsv[..., 0] = angle * 180 / np.pi / 2  # Dirección del flujo
    hsv[..., 2] = cv2.normalize(magnitude, None, 0, 255, cv2.NORM_MINMAX)  # Magnitud del flujo

    # Convertir de HSV a BGR para visualización
    bgr_flow = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

    # Mostrar el flujo óptico y el cuadro original
    cv2.imshow('Optical Flow Farneback', bgr_flow)
    cv2.imshow('Original Frame', frame2)

    # Actualizar el cuadro anterior para la siguiente iteración
    prvs = next_frame

    # Salir si se presiona la tecla 'q'
    if cv2.waitKey(30) & 0xFF == ord('q'):
        break
```

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

Explicación del código:

1. Captura de video: Se carga un video en formato MP4 y se procesa cuadro por cuadro.
2. Conversión a escala de grises: El flujo óptico generalmente se calcula en imágenes en escala de grises para simplificar el cálculo.
3. Cálculo del flujo óptico: cv2.calcOpticalFlowFarneback se usa para calcular el flujo óptico denso entre dos cuadros consecutivos. Los parámetros de esta función controlan la precisión y la velocidad del cálculo.
4. Visualización del flujo: Se convierte la dirección (ángulo) y la magnitud del flujo en una imagen de color para visualizarlos. Los colores representan la dirección del movimiento, mientras que la intensidad (brillo) representa la magnitud del movimiento.
5. Loop sobre el video: El código se ejecuta en un bucle, procesando cada cuadro del video de manera continua.

### 7.1.3. Optical Flow disperso (Lucas-Kanade)

Este método es útil cuando solo se desea hacer seguimiento de un conjunto de puntos específicos (en lugar de todos los píxeles). A continuación, un ejemplo usando Lucas-Kanade para seguir puntos clave en el video:

```
import cv2
import numpy as np

# Parámetros para la función goodFeaturesToTrack
feature_params = dict(maxCorners=100, qualityLevel=0.3, minDistance=7, blockSize=7)

# Parámetros para el algoritmo Lucas-Kanade
lk_params = dict(winSize=(15, 15), maxLevel=2, criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_(

# Cargar el video
cap = cv2.VideoCapture('video.mp4')

# Leer el primer cuadro
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)

# Encontrar las características a seguir (puntos de interés)
p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)

# Crear una máscara para dibujar los trayectos
mask = np.zeros_like(old_frame)

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Calcular el flujo óptico utilizando el método Lucas-Kanade
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

    # Seleccionar los puntos buenos
    good_new = p1[st == 1]
    good_old = p0[st == 1]

    # Dibujar las trayectorias
    for i, (new, old) in enumerate(zip(good_new, good_old)):
        a, b = new.ravel()
        c, d = old.ravel()
        mask = cv2.line(mask, (a, b), (c, d), (0, 255, 0), 2)
        frame = cv2.circle(frame, (a, b), 5, (0, 0, 255), -1)

    # Mostrar el resultado
    img = cv2.add(frame, mask)
    cv2.imshow('Optical Flow Lucas-Kanade', img)
```

```
    # Actualizar el cuadro anterior y los puntos
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1, 1, 2)

    # Salir si se presiona la tecla 'q'
    if cv2.waitKey(30) & 0xFF == ord('q'):
        break

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

- Explicación:
- Selección de puntos de interés: Se utiliza cv2.goodFeaturesToTrack para encontrar puntos que son buenos candidatos para ser seguidos (esquinas o bordes fuertes).
- Cálculo del flujo óptico: Se usa cv2.calcOpticalFlowPyrLK para seguir los puntos de interés en cuadros consecutivos usando el método de Lucas-Kanade.
- Dibujar trayectorias: Los movimientos de los puntos se dibujan en la imagen para visualizar cómo se desplazan.
- Limitaciones y Consideraciones
- Condiciones de iluminación: Los cambios bruscos en la iluminación pueden afectar el cálculo del Optical Flow, ya que se basa en la consistencia de la intensidad de los píxeles.
- Ruido: El ruido en el video puede generar detecciones erróneas de movimiento.
- Eficiencia: El Optical Flow denso puede ser costoso computacionalmente, aunque puede optimizarse para su uso en tiempo real dependiendo de la resolución del video y la precisión requerida.
- Conclusión

El Optical Flow es una excelente técnica para detectar movimiento en video y podría ser más adecuado para emular ciertos aspectos del comportamiento de una cámara de video por eventos, ya que detecta cambios espaciales y temporales de manera eficiente.

### 7.1.4. Estimacion de Distancia

1. Uso de un objeto de tamaño conocido (principio de telémetro óptico)

La mayoría de las miras telescópicas utilizan una retícula diseñada para medir la distancia en función del tamaño aparente de un objeto conocido (por ejemplo, la altura promedio de una persona o vehículo). Si conoces el tamaño real de un objeto y puedes medir su tamaño aparente en la imagen, puedes calcular la distancia usando la fórmula:

$$D = \frac{(H \times F)}{h}$$

Donde:

- D es la distancia al objeto.
- H es la altura real del objeto.
- F es la distancia focal del objetivo de la cámara.
- h es la altura aparente del objeto en la imagen.
- Ejemplo de código conceptual

Usando YOLOv8 para detectar un vehículo, y sabes que la altura promedio de un vehículo es de 1.5 metros, podrías estimar la distancia basándote en el tamaño del bounding box:

```
import cv2
import numpy as np
from ultralytics import YOLO

# Constante de la altura real del objeto (en metros)
HEIGHT_REAL = 1.5  # Ejemplo: altura promedio de un vehículo
FOCAL_LENGTH = 1000  # Longitud focal de la cámara en píxeles
```

```
# Cargar el modelo YOLOv8 previamente entrenado
model = YOLO('yolov8n.pt')

# Capturar el video
cap = cv2.VideoCapture('carretera.mp4')

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Obtener predicciones de YOLOv8
    results = model(frame)

    # Iterar sobre las detecciones (por ejemplo, vehículos)
    for result in results:
        for box in result.boxes.xyxy:
            # Extraer coordenadas del bounding box
            x1, y1, x2, y2 = map(int, box)
            height_apparent = y2 - y1  # Altura aparente en píxeles

            # Calcular la distancia utilizando la fórmula del telémetro
            if height_apparent > 0:
                distance = (HEIGHT_REAL * FOCAL_LENGTH) / height_apparent
                print(f'Distance to vehicle: {distance:.2f} meters')

            # Dibujar el bounding box y mostrar la distancia calculada
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
            cv2.putText(frame, f'Distancia: {distance:.2f}m', (x1, y1 - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)

    # Mostrar el cuadro con la distancia calculada
    cv2.imshow('Frame with Distance', frame)

    # Salir si se presiona 'q'
    if cv2.waitKey(30) & 0xFF == ord('q'):
        break

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

- Explicación del código:

YOLOv8 se usa para detectar un vehículo.

1. Altura aparente: Se calcula utilizando las coordenadas del bounding box que rodea al vehículo.
2. Fórmula del telémetro: Se usa la fórmula basada en la distancia focal y la altura conocida del vehículo para calcular la distancia.
3. Resultado: La distancia se dibuja sobre el cuadro de video.

Este enfoque te permitirá estimar distancias cuando el objeto detectado tiene un tamaño conocido.

### 7.1.5. Emular el sistema de retícula

1. Retícula con marcas en miliradianes (mils)

Las retículas de los visores de francotiradores de la Segunda Guerra Mundial estaban divididas en marcas (mils), que representaban una fracción de ángulo, lo que les permitía medir el tamaño aparente de un objetivo. Un miliradian es una milésima parte de un radian, y en términos prácticos, 1 miliradian equivale a un ángulo de 1 metro a 1000 metros de distancia.

Si tienes un sistema de cámaras (como el que mencionabas con YOLO8 y OpenCV), podrías simular una retícula que divida la vista de la cámara en unidades de miliradianes o píxeles.

1. Cálculo de la distancia usando el tamaño conocido de un objeto

Para calcular la distancia, necesitas conocer el tamaño real del objeto que estás observando (por ejemplo, un coche estándar, una persona, o cualquier objeto cuya dimensión puedas estimar).

La fórmula utilizada para estimar la distancia es la siguiente:

Distancia(m) = (Tamaño real objeto(mts) / Tamaño (mils) ) x 1000

1. Proceso paso a paso:

Supongamos que quieres estimar la distancia a un coche en la carretera:

Conocer el tamaño del coche: Estimas que el coche mide aproximadamente 4 metros de largo.

Medir el tamaño aparente en la retícula: Observas el coche en tu retícula simulada y cuentas cuántas divisiones (mils o píxeles) ocupa en la vista de la cámara. Supongamos que el coche ocupa 2 miliradianes.

Aplicar la fórmula:

Distancia = (4 / 2 ) x 1000 = 2000 mts

1. Simulación con OpenCV

Podrías superponer una retícula en la imagen capturada por tu cámara, marcando las divisiones en miliradianes o píxeles. Con OpenCV, es posible detectar los bordes del objeto (por ejemplo, con YOLO8 para detectar el coche) y calcular el número de divisiones que ocupa el objeto en la retícula.

Dibujar la retícula: Puedes usar OpenCV para dibujar una cuadrícula de referencia (retícula) en la pantalla:

```
import cv2

def draw_reticle(image, divisions=10):
    height, width = image.shape[:2]
    step_x = width // divisions
    step_y = height // divisions

    # Dibujar líneas horizontales y verticales
    for i in range(1, divisions):
        cv2.line(image, (i * step_x, 0), (i * step_x, height), (255, 0, 0), 1)
        cv2.line(image, (0, i * step_y), (width, i * step_y), (255, 0, 0), 1)

    return image

# Supongamos que 'frame' es una imagen de video de la cámara
reticle_image = draw_reticle(frame)
cv2.imshow('Reticle', reticle_image)
cv2.waitKey(0)
```

Detección del objeto: Una vez que el coche es detectado (por ejemplo, con YOLO8), puedes medir cuántos píxeles o divisiones ocupa dentro de la retícula. Esto te permitirá estimar el tamaño angular del objeto y aplicar la fórmula.

Consideraciones:

1. Tamaño del objeto: Necesitas conocer o estimar el tamaño real del objeto que estás observando.
2. Calibración: Si decides usar píxeles en lugar de miliradianes, debes calibrar tu cámara para relacionar píxeles con un ángulo real. Por ejemplo, si conoces la distancia a un objeto y su tamaño real, puedes calcular cuántos píxeles de tu cámara corresponden a un miliradian.
3. Corrección por inclinación: Si la carretera o el terreno está inclinado, necesitarás aplicar la regla del tirador ("Rifleman’s Rule") para corregir la distancia dependiendo de la inclinación del objetivo.

La corrección depende del ángulo de inclinación del terreno:

Distancia corregida = Distancia × cos(fi)

## 7.2. ANEXO Servidor GStreamer

### 7.2.1. Configuración del servidor GStreamer Socket Unix

Para capturar video a través de RTSP y emitirlo a una red, puedes usar GStreamer. Un ejemplo básico de pipeline para capturar desde una cámara RTSP sería:

```
gst-launch-1.0 rtspsrc location=rtsp://<rtsp-url> ! decodebin ! videoconvert ! jpegenc ! filesink l
```

Pasos generales para implementar esto:

1. Crear el socket Unix: Usa alguna herramienta o programa en Python o C que cree un socket Unix (con algo como socket.$AF_{UNIX}$).
2. Obtener el file descriptor del socket:

En tu código, debes obtener el file descriptor de este socket (normalmente usando socket.fileno() en Python o alguna función equivalente en C).

1. Lanzar el pipeline de GStreamer: Lanza el pipeline de GStreamer y asegúrate de pasar el descriptor del socket al elemento fdsink.
2. Leer desde el socket Unix y decodificar los frames

### 7.2.2. Cliente python + opnecv + yolo8

```
import cv2
import torch
import socket
import numpy as np

# Cargar el modelo YOLOv11 (Asegúrate de tener el modelo entrenado)
model = torch.hub.load('ultralytics/yolov11', 'custom', path='yolov11.pt')

# Crear un socket Unix para recibir el video
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
socket_address = '/tmp/unix_socket'  # Debe coincidir con el socket creado por GStreamer

# Conectar al socket Unix
sock.bind(socket_address)
sock.listen(1)
conn, addr = sock.accept()

# Bucle de procesamiento de frames
while True:
    # Leer primero 4 bytes para saber el tamaño de la imagen que llega
    length_data = conn.recv(4)
    if not length_data:
        break

    # Convertir los 4 bytes en un entero (longitud del frame)
    length = int.from_bytes(length_data, byteorder='big')

    # Leer los datos de la imagen según la longitud obtenida
    image_data = b''
    while len(image_data) < length:
        packet = conn.recv(length - len(image_data))
        if not packet:
            break
        image_data += packet

    # Convertir los datos recibidos (bytes) en una imagen
    np_data = np.frombuffer(image_data, np.uint8)
    frame = cv2.imdecode(np_data, cv2.IMREAD_COLOR)
```

```
    if frame is None:
        break

    # Detección de objetos
    results = model(frame)

    # Mostrar resultados en tiempo real
    cv2.imshow('Detección de Objetos', results.render()[0])

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Cerrar conexión
conn.close()
sock.close()
cv2.destroyAllWindows()
```

Explicación del código:

1. Socket Unix:

El socket Unix se crea con socket.$AF_{UNIX}$ y se enlaza a una dirección en el sistema de archivos (/tmp/$unix_{socket}$). El código espera a que GStreamer (u otro proceso) se conecte al socket y luego empieza a recibir datos.

1. Recepción de frames:

Primero, se leen los primeros 4 bytes del socket, los cuales indican el tamaño del frame (esto es importante porque no se envía una longitud fija de datos). Luego, se reciben los datos de la imagen de acuerdo con el tamaño recibido y se convierten en un array de NumPy. Finalmente, la imagen codificada en JPEG se decodifica usando cv2.imdecode.

1. Procesamiento con YOLO:

Una vez que el frame es decodificado, puedes pasarlo por el modelo YOLO y obtener los resultados de la detección de objetos. Mostrar el video:

1. La función cv2.imshow se usa para mostrar los resultados en tiempo real, similar a tu código original.

### 7.3. Ventajas de usar sockets Unix

# 8. Ventajas de usar sockets Unix

### 8.1. 1. Baja Latencia

Los sockets Unix permiten una comunicación extremadamente rápida, ya que no dependen de capas de red como TCP/IP o UDP. Esto reduce significativamente la **latencia**, lo que es crucial para aplicaciones en tiempo real como la transmisión de video.

### 8.2. 2. Menor Overhead

No hay encapsulación ni sobrecarga de protocolos de red, lo que hace que la transmisión de datos sea **más eficiente**. No necesitas manejar direcciones IP, puertos de red, retransmisiones o pérdida de paquetes. Esto se traduce en menos procesamiento en comparación con otros protocolos basados en red como RTSP.

## 8.3. 3. Rendimiento y Eficiencia en la Comunicación Local

Como los sockets Unix son un método de **comunicación entre procesos (IPC)** local, están optimizados para funcionar dentro del mismo sistema operativo. No involucran la red, lo que elimina posibles cuellos de botella. Esto hace que sean mucho más rápidos que los métodos que usan la red, incluso si están en la misma máquina.

## 8.4. 4. Consistencia y Fiabilidad

Al no tener que lidiar con las complejidades de la red (como la pérdida de paquetes, fluctuaciones en el ancho de banda o reintentos), los sockets Unix ofrecen una **transmisión más consistente** y confiable. No se ven afectados por condiciones externas como lo haría una transmisión basada en RTSP en una red inestable.

## 8.5. 5. Uso Eficiente de Recursos

Como no es necesario gestionar protocolos de red, la **carga sobre la CPU y la memoria** es menor en comparación con los protocolos de transmisión a través de redes. Esto puede resultar en una mejor **eficiencia energética** y menor uso de recursos del sistema. La transmisión local es mucho más eficiente porque se elimina el overhead de procesamiento que requiere el manejo de protocolos de red.

## 8.6. 6. Ancho de Banda Ilimitado

Los sockets Unix no están limitados por los **anchos de banda de la red**. Puedes transmitir grandes cantidades de datos a gran velocidad sin preocuparte por congestión o latencias adicionales impuestas por la red. Esto es especialmente útil para la transmisión de video de alta calidad, donde cada milisegundo cuenta.

## 8.7. 7. Simplicidad de Implementación para Entornos Locales

Los sockets Unix son más **simples de implementar y mantener** en un entorno local, ya que no requieren configuración de red, apertura de puertos o manejo de firewalls. Esto reduce la complejidad y hace que la configuración sea más directa cuando los procesos que se comunican están en la misma máquina o en una red local confiable.

## 8.8. 8. Seguridad Mejorada

Al ser locales, los sockets Unix no están expuestos a la red externa, lo que **reduce significativamente los riesgos de seguridad**. No es necesario preocuparse por ataques de red, como la interceptación de datos (man-in-the-middle), que podrían ocurrir en una transmisión RTSP expuesta a la red. El acceso a los sockets Unix puede ser controlado a nivel del sistema de archivos (mediante permisos), lo que permite una **mayor granularidad de control** en quién puede acceder a ellos.

## 8.9. 9. Sin Necesidad de Manejar Protocolo RTSP

Con RTSP, es necesario manejar el protocolo de control de transmisión (configuración, inicio de sesión, cierre, etc.), lo que puede añadir complejidad y latencia. En cambio, con los sockets Unix, simplemente transmites los datos directamente, sin necesidad de gestionar la lógica asociada al protocolo de control.

## 8.10. 10. Soporte Nativo en Sistemas Unix/Linux

Los sockets Unix están bien soportados de forma **nativa en sistemas operativos Unix/Linux**. Esto hace que sean una opción natural y eficiente en estas plataformas, con amplia documentación y herramientas disponibles para su gestión.

### 8.11. 11. Menos Dependencia de la Red

Los sockets Unix eliminan cualquier dependencia de la infraestructura de red. Si la red tiene problemas o está congestionada, una transmisión RTSP puede fallar o experimentar interrupciones. Esto no sucede con los sockets Unix, que son locales y no dependen de la conectividad externa.

### 8.12. 12. Escalabilidad en Sistemas Locales

En aplicaciones donde múltiples procesos necesitan comunicarse en la misma máquina (como varios servicios o componentes que comparten datos), los sockets Unix son muy escalables y eficientes, permitiendo **comunicaciones simultáneas** de manera efectiva.

### 8.13. Casos de Uso Típicos

- **Transmisión de video en tiempo real localmente** (como en tu caso, donde GStreamer envía el stream de video a un socket para ser procesado en Python).
- **Comunicación entre procesos en la misma máquina** (IPC).
- **Sistemas de alta disponibilidad** donde la latencia es crítica y se deben minimizar las dependencias externas.

### 8.14. Conclusión

Usar **sockets Unix** es una opción extremadamente eficiente para **comunicación local**, donde las necesidades de velocidad, baja latencia y simplicidad son fundamentales. Supera a RTSP en estos aspectos cuando se trata de transmisión dentro de la misma máquina o en entornos cerrados.

## 9. Recursos de software

### 9.1. Videos para testear

- CABA https://youtu.be/VZKuJcpgCA8?si=e_b-FRSzdABe2Y_R
- CABA NOCHE https://youtu.be/75X9vSFCh14?si=G1nb0jK7gxl0I-6l
- CABA to San Isidro https://youtu.be/Y-M8nAQ0gYo?si=x4gxleG-02DiWzXi

### 9.2. Detecion de objetos (Módulo 7)

- Dataset https://universe.roboflow.com/selfdriving-car-qtywx/self-driving-cars-lfjou/dataset/6
- train https://colab.research.google.com/drive/1RPv4PCoEJqiKbAHwpwiIcR30qLKZqtEi#scrollTo=cVv-rRzuh9d4
- Preprocessing Resize: Stretch to 416x416

### 9.3. Ultralytics YOLOv8 - Detección de marcas en la carretera

- Dataset https://www.kaggle.com/code/prakharsinghchouhan/ultralytics-yolov8-road-mark-detection/input

### 9.4. Trafic signals

- Dataset https://universe.roboflow.com/selfdriving-car-qtywx/self-driving-cars-lfjou

Created: 2024-11-28 Thu 13:10

Validate