

**Diseño e implementación de un sistema de detección y rastreo de vehículos para el modelo a escala AutoNOMOS, y simulación de una red vanet**

TESIS

PARA OBTENER EL TÍTULO DE

INGENIERO en Mecatrónica

P R E S E N T A

ISRAEL FONSECA ZÁRATE

A S E S O R: MTRO. RAFAEL GREGORIO GAMBOA HIRALES

CIUDAD DE MÉXICO 2023

Resumen

Dentro de la investigación realizada sobre vehículos autónomos, un punto importante es la detección y el seguimiento de otros participantes en el tráfico que puedan ser riesgos potenciales, no solo en la interacción con vehículos propios, pero también para otros, en casos donde se encuentren en puntos ciegos. Razón por la cual se habla de lograr un entorno colaborativo entre vehículos de esta naturaleza. Como objetivo de este trabajo se presenta una implementación sobre el AutoModelCar de un sistema de detección y rastreo de vehículos. Esto desde un enfoque que requiera poco poder de procesamiento y logre resultados cercanos en tiempo real. Para ello se eligió la combinación de filtros de cascada con máquinas de soporte vectorial para la detección, y el uso del filtro de Kalman para el rastreo. Después, pensando en un entorno colaborativo, se busca que los vehículos compartan entre sí sus detecciones mediante una red vanet basada en el modelo v2v. Para ello se simula y experimenta con el protocolo AODV y el estándar DSRC. Si bien en vehículos autónomos el método indiscutible son las redes neuronales, con los métodos usados se logró alcanzar un buen desempeño que, aunque no reemplaza las redes neuronales, sin duda es una opción que puede ser relevante para aplicaciones orientadas en tiempo real y con bajo poder de procesamiento. Mientras que en redes vanet, se experimentó sobre los estándares que se están probando hoy en día viéndolos desde un enfoque orientado a aplicaciones de seguridad. Con esto en mente se buscó tener bajos intervalos entre el envío de mensajes y un bajo RTT. Como se demuestra, el modelo v2v puede funcionar si toma en cuenta las condiciones de tráfico, para adaptar parámetros como la tasa de envíos. Aquí, los principales retos son la velocidad de los nodos, y situaciones con una alta densidad de nodos, esto último, se puede mejorar si se añade el uso de infraestructura en lugares como ciudades, pasando a un modelo v2x. Donde se espera que la infraestructura cumpla solo un rol de coordinador, y que aproveche las ventajas de una comunicación directa entre vehículos, como son la latencia y cobertura geográfica.

**Palabras clave**: Maquinas de soporte vectorial, Filtros de cascada, Filtro de Kalman, Protocolo AODV, Estándar IEEE 802.11p (DSRC).

# Tabla de contenido

[Tabla de contenido 4](#_Toc146286791)

[Índice de figuras 8](#_Toc146286792)

[Índice de tablas 17](#_Toc146286793)

[1. Introducción 19](#_Toc146286794)

[1.1. Contexto 19](#_Toc146286795)

[1.2. Identificación del problema 20](#_Toc146286796)

[1.3. Objetivos 22](#_Toc146286797)

[1.4. Metodología 22](#_Toc146286798)

[1.5. Organización del documento 23](#_Toc146286799)

[2. Análisis 26](#_Toc146286800)

[2.1. Requerimientos 26](#_Toc146286801)

[2.1.1. Requerimientos funcionales 26](#_Toc146286802)

[2.2.1. Requerimientos no funcionales 27](#_Toc146286803)

[2.3 Restricciones 28](#_Toc146286804)

[2.4 Alcance 29](#_Toc146286805)

[2.5 Trabajos relacionados 30](#_Toc146286806)

[3. Diseño 35](#_Toc146286807)

[3.1. Arquitectura 35](#_Toc146286808)

[3.2. Detección de objetos 43](#_Toc146286809)

[3.1.2. Técnicas de detección 48](#_Toc146286810)

[3.1.2 Clasificador de cascada 50](#_Toc146286811)

[3.1.3 Hog+svm 53](#_Toc146286812)

[3.1.4 Comparación con redes neuronales convolucionales 57](#_Toc146286813)

[3.2 Rastreo de objetos 60](#_Toc146286814)

[3.2.1 Técnicas 60](#_Toc146286815)

[3.2.2 Filtro de Kalman 63](#_Toc146286816)

[3.2.3 Caso multivariable 65](#_Toc146286817)

[4 Codificación 69](#_Toc146286818)

[4.1 Entrenamiento del filtro de cascada 69](#_Toc146286819)

[4.2 Entrenamiento de svm 72](#_Toc146286820)

[4.3 Detección y nodo para el vehículo 81](#_Toc146286821)

[4.4 Rastreo 90](#_Toc146286822)

[5 Resultados 100](#_Toc146286823)

[6 Diseño de la Red vanet 112](#_Toc146286824)

[6.1 Arquitectura 112](#_Toc146286825)

[6.2 Descripción 115](#_Toc146286826)

[6.3 Patrones de comunicación 117](#_Toc146286827)

[6.4 Modelos v2x, v2i y v2v 120](#_Toc146286828)

[6.5 DSRC 122](#_Toc146286829)

[6.4.1 Capa física 123](#_Toc146286830)

[6.4.2 Capa MAC 125](#_Toc146286831)

[6.4.3 Capa lógica 126](#_Toc146286832)

[6.4.4 Capas superiores 127](#_Toc146286833)

[6.5 Contenido de los mensajes 128](#_Toc146286834)

[7 Implementación de una red vanet 129](#_Toc146286835)

[7.1 Arquitectura y requisitos de la red vanet 129](#_Toc146286836)

[7.2 Implementación 130](#_Toc146286837)

[7.2.1 AODV 131](#_Toc146286838)

[7.2.2 IEEE 802.11p 135](#_Toc146286839)

[8 Resultados 144](#_Toc146286840)

[8.1 Simulación 1 144](#_Toc146286841)

[8.1.1 Diferentes velocidades e intervalos 149](#_Toc146286842)

[8.2 Simulación 2 155](#_Toc146286843)

[8.2.1 Diferentes densidades de nodos 162](#_Toc146286844)

[9 Conclusiones 176](#_Toc146286845)

[10 Apéndice 1: Documentación 181](#_Toc146286846)

[10.1 Documentación AutoModelCar 181](#_Toc146286847)

[10.2 Nodos Programados 210](#_Toc146286848)

[11 Apéndice 2: Tutorial AutoModelCar 214](#_Toc146286849)

[11.1 Guía de ROS 214](#_Toc146286850)

[11.2 Transformaciones 229](#_Toc146286851)

[11.3 Cámara 234](#_Toc146286852)

[12 Referencias 238](#_Toc146286853)

Índice de figuras

[Figura 1‑1 etapas en la programación extrema. Fuente: (Pressman, 2010) 23](#_Toc146286698)

[Figura 3‑1 En el recuadro es el nombre del nodo, en los círculos los tópicos que publican. Son los tópicos los que interconectan con otros nodos. El nodo manual\_control, a partir de la velocidad y steering se comunica con los motores. Lane\_navigation en realidad son un conjunto de nodos que se encargan del seguimiento de línea. 37](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286699)

[Figura 3‑2 Pasos para la detección 41](#_Toc146286700)

[Figura 3‑3 Pasos para el rastreo 41](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286701)

[Figura 3‑4 Pasos para el entrenamiento de máquinas de soporte vectorial 42](#_Toc146286702)

[Figura 3‑5 Ejemplo de la operación realizada en lbp, primero si el valor del pixel es menor al centro se toma 1, a continuación, partiendo de la esquina superior derecha se le asigna como 2^0, de ahí se da una vuelta en sentido de las manecillas del reloj, aumenta. Fuente: (Rosebrock 2015) 52](#_Toc146286703)

[Figura 3‑6 Características extraídas de una imagen con lbp. Fuente: (Rosebrock 2015) 52](#_Toc146286704)

[Figura 3‑7 Ejemplo de la separación de dos variables por un margen de separación, se muestra la separación óptima que logra el mayor margen entre las dos variables, fuente: (“opencv: introduction to support vector machines” s. F.) 55](#_Toc146286705)

[Figura 3‑8 Tabla de diferentes kernels para linealizar un problema. Fuente:(Kecman 2005) 57](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286706)

[Figura 3‑9 Pasos del filtro de Kalman, hay una inicialización a partir de la que se realiza la predicción, en el siguiente paso, sí se tienen las medidas se realiza la actualización, además, se puede agregar un paso de control. Fuente: (Becker s. F.) 63](#_Toc146286707)

[Figura 4‑1 Ejemplos de vehículos 69](#_Toc146286708)

[Figura 4‑2 Ejemplos de imágenes negativas para el entrenamiento 70](#_Toc146286709)

[Figura 4‑3 Imágenes de vehículos de lado 72](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286710)

[Figura 4‑4 Imágenes de la parte trasera de vehículos 72](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286711)

[Figura 4‑5 Función load\_images 73](#_Toc146286712)

[Figura 4‑6 Con blocksize (64,64), cellsize y blockstride de (4,4). 75](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286713)

[Figura 4‑7 Con blocksize (64,64), cellsize y blockstride (8,8), con solo 8x8 celdas no es suficiente para extraer las características de la imagen, por lo que el resultado es muy pobre. 75](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286714)

[Figura 4‑8 Imagen del vehículo del simulador 75](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286715)

[Figura 4‑9 Con filtro gaussiano (5,5) 75](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286716)

[Figura 4‑10 Hog features de la carretera 75](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286717)

[Figura 4‑11 Hog features de un árbol 75](#_Toc146286718)

[Figura 4‑12 Función computehogs, extrae los descriptores de todas las imágenes 76](#_Toc146286719)

[Figura 4‑13 Código de la función convert\_to\_ml, mismo propuesto en los ejemplos de opencv 77](#_Toc146286720)

[Figura 4‑14 Entrenamiento de svm con trainauto 78](#_Toc146286721)

[Figura 4‑15 Se observa el impacto de establecer pesos para el valor de c, a la izquierda la clasificación normal, a la derecha con un peso de 0.9 sobre la clase de verde, fuente: (“svm bias on weights of positives and negatives - opencv q&a forum» s. F.) 79](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286722)

[Figura 4‑16 Imagen de la parte de entrenamiento en opencv 2.4 usando train 81](#_Toc146286723)

[Figura 4‑17 Imagen de la función main, la configuración inicial, hasta el inicio del ciclo 84](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286724)

[Figura 4‑18 Ejemplos de falsos negativos 87](#_Toc146286725)

[Figura 4‑19 Fragmento del código para detectar sobre una región 90](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286726)

[Figura 4‑20 measurementnoisecov de 0.01 94](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286727)

[Figura 4‑21 mesasurementnoisecov de 0.1 94](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286728)

[Figura 4‑22 measurementnoisecov de 0.001 95](#_Toc146286729)

[Figura 4‑23 Con processnoisecov de 0.00005, los puntos azules reaccionan más lento a los cambios 95](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286730)

[Figura 4‑24 Con processnoisecov de 0.5, los puntos azules reaccionan más rápido y como consecuencia se disparan 95](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286731)

[Figura 4‑25 Código completo para crear el filtro de Kalman 96](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286732)

[Figura 4‑26 Código para hacer la predicción y corrección 99](#_Toc146286733)

[Figura 5‑1 Candidatos arrojados por el filtro de cascada 106](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286734)

[Figura 5‑2 Ejemplos de detecciones sobre frames, en la primera imagen se identifican correctamente los dos vehículos, en el segundo por el cambio de iluminación no detecta uno de los vehículos, en la tercera imagen se muestra un falso positivo. 106](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286735)

[Figura 5‑3 Predicción del filtro de Kalman, los recuadros blancos son las detecciones, y los negros la predicción (en esa misma ventana de la predicción es donde se realiza la detección parcial para buscar el vehículo y hacer la corrección. 106](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286736)

[Figura 5‑4 Imagen del simulador desde gazebo, mapa con dos coches 108](#_Toc146286737)

[Figura 5‑5 Ejemplo de detecciones en el simulador, arriba se muestra la detección, abajo la predicción 109](#_Toc146286738)

[Figura 6‑1 Fuente: (Hartenstein y Laberteaux, 2010) algunas aplicaciones y su modelo sugerido 121](#_Toc146286739)

[Figura 6‑2 Fuente: (Hartenstein y Laberteaux 2010) capas del estándar DSRC, se compone de dos partes principales, el estándar IEEE 802.11p e IEEE 1609 o wave 122](#_Toc146286740)

[Figura 6‑3 Características de la banda 5850-5925 MHz. Fuente: (“identificación de necesidades de espectro para sistemas de transporte inteligente en la banda 5850-5925 MHz”, 2021) 123](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286741)

[Figura 6‑4 OFDM en el dominio de la frecuencia. Fuente: (Cabrera y Tarrés, s. f.) 124](#_Toc146286742)

[Figura 6‑5 Fuente: (Kenney 2011) encabezados de capa 2 125](#_Toc146286743)

[Figura 6‑6 Fuente: (Kenney 2011) encabezados de la subcapa LLC y SNAP 126](#_Toc146286744)

[Figura 6‑7 Fuente: (Kenney 2011) formato de wave short message 127](#_Toc146286745)

[Figura 7‑1 Ventana de diseño para un archivo .ned 131](#_Toc146286746)

[Figura 7‑2 Código del archivo .NED, son los módulos incluidos 132](#_Toc146286747)

[Figura 7‑3 Imagen de la simulación de AODV 135](#_Toc146286748)

[Figura 7‑4 Imagen del editor de una red en sumo 136](#_Toc146286749)

[Figura 7‑5 Pestañas para crear construcciones (ala izquierda) y caminos (a la derecha) 137](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286750)

[Figura 7‑6 Ventana para crear rutas 137](#_Toc146286751)

[Figura 7‑7 Imagen final en sumo 138](#_Toc146286752)

[Figura 7‑8 Archivo .NED para la segunda simulación 139](#_Toc146286753)

[Figura 7‑9 Imagen de la simulación corriendo en omnet++, se observa un nodo retransmitiendo a todos un mensaje 143](#_Toc146286754)

[Figura 8‑1 simulación 1, se experimenta el desempeño con diferente densidad de nodos 144](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286755)

[Figura 8‑2 A la izquierda la posición inicial, a la derecha la posición final 147](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286756)

[Figura 8‑3 resultados del escenario 1 148](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286757)

[Figura 8‑4 a la izquierda los tiempos que demora "in progress frames" a la derecha "pending queue" 148](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286758)

[Figura 8‑5 Rutas formadas por el protocolo AODV 149](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286759)

[Figura 8‑6 Probabilidad de recibir 2 mensajes con AODV 152](#_Toc146286760)

[Figura 8‑7 Probabilidad de recibir 4 mensajes con AODv 152](#_Toc146286761)

[Figura 8‑8 El nodo 6 en rojo quiere transmitir 156](#_Toc146286762)

[Figura 8‑9 El nodo 6 transmite el mensaje, y los demás nodos al recibirlo retransmiten una vez, el nodo 7 y 4 no son vistos por el nodo 6, con las retransmisiones quedan informados. 156](#_Toc146286763)

[Figura 8‑10 Los nodos cercanos 5, 11 y 7 bajan su velocidad y permiten que el nodo 6 se incorpore, el nodo 4 no baja su velocidad 157](#_Toc146286764)

[Figura 8‑11 Posición final, el nodo 6 se incorpora 157](#_Toc146286765)

[Figura 8‑12 A la izquierda se observa la posición de partida de los nodos, a la derecha la posición final 158](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286766)

[Figura 8‑13 Paquetes enviados (incluye retransmisiones) y paquetes recibidos por cada nodo 159](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286767)

[Figura 8‑14 Tiempos de espera promedio de cada nodo 159](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286768)

[Figura 8‑15 Porcentaje de mensajes recibidos en promedio por los nodos 164](#_Toc146286769)

[Figura 8‑16 Tiempos de encolamiento 165](#_Toc146286770)

[Figura 8‑17 Probabilidad de recibir 2 y 4 mensajes con 10 nodos 165](#_Toc146286771)

[Figura 8‑18 Probabilidad de recibir 2 y 4 mensajes con 10 nodos, en un trayecto de 200 metros 166](#_Toc146286772)

[Figura 8‑19 Porcentaje de mensajes recibidos con 20 nodos 168](#_Toc146286773)

[Figura 8‑20 Probabilidad de recibir 2 y 4 mensajes con 20 nodos 168](#_Toc146286774)

[Figura 8‑21 Probabilidad de recibir 2 y 4 mensajes con 20 nodos, en un trayecto de 200 metros 169](#_Toc146286775)

[Figura 8‑22 Porcentaje de mensajes recibidos con 30 nodos 170](#_Toc146286776)

[Figura 8‑23 Probabilidad de recibir 2 y 4 mensajes con 30 nodos 171](#_Toc146286777)

[Figura 8‑24 Probabilidad de recibir 2 y 4 mensajes con 30 nodos, en un trayecto de 200 metros 171](#_Toc146286778)

[Figura 8‑25 Tiempos de encolamiento al aumentar el número de nodos 172](#_Toc146286779)

[Figura 8‑26 Intervalos de transmisión recomendados a una velocidad de 30 km/h 173](#_Toc146286780)

[Figura 8‑27 Intervalos de transmisión recomendados a 50 km/h 174](#_Toc146286781)

[Figura 8‑28 Intervalos de transmisión recomendados a 80 km/h 174](#_Toc146286782)

[Figura 10‑1 A la izquierda módulos y conexiones del modelo 1, a la derecha el modelo 2. Fuente: (“hardware (autonomos model v2) · automodelcar/automodelcarwiki wiki · github” s. f.) 181](file:///C:\Users\ifons\Desktop\mostrar\Rev4.docx#_Toc146286783)

[Figura 11‑1 Ejemplo de roslaunch correspondiente a manual\_control 216](#_Toc146286784)

[Figura 11‑2 Se muestran algunos tópicos relacionados a la cámara del carro, al correr manual\_control 218](#_Toc146286785)

[Figura 11‑3 Mensaje publicado en el tópico /scan relacionado al lidar 219](#_Toc146286786)

[Figura 11‑4 El tópico /app/camera/rgb/image\_raw corresponde a la imagen a color publicada por el nodo de la cámara, se lee para la detección 219](#_Toc146286787)

[Figura 11‑5 Composición del mensaje tipo laserscan para el lidar 221](#_Toc146286788)

[Figura 11‑6 Conexión con transformaciones de las coordenadas de la cámara con el vehículo. Fuente (“access the tf transformation tree in ros - matlab & simulink - mathworks américa latina” s. f.) 229](#_Toc146286789)

[Figura 11‑7 Resultado de la conexión de los frames. Fuente (“setting up transformations — navigation 2 1.0.0 documentation” s. f.) 233](#_Toc146286790)

Índice de tablas

[Tabla 5‑1 Matriz de confusión para svm con imágenes de la parte trasera de coches 100](#_Toc146286683)

[Tabla 5‑2 Matriz de confusión de svm con coches de lado 101](#_Toc146286684)

[Tabla 5‑3 Matriz de confusión resultante de juntar ambas svm 101](#_Toc146286685)

[Tabla 5‑4 Matriz de confusión de svm completa con la parte trasera de coches 102](#_Toc146286686)

[Tabla 5‑5 Matriz de confusión de svm completa con coches de lado 103](#_Toc146286687)

[Tabla 5‑6 Matriz de confusión de ambas svm completas 103](#_Toc146286688)

[Tabla 5‑7 Matriz de confusión del filtro de cascada 104](#_Toc146286689)

[Tabla 5‑8 Matriz de confusión de ambos detectores 104](#_Toc146286690)

[Tabla 5‑9 Tiempos medidos de los detectores en los tres ambientes de pruebas 109](#_Toc146286691)

[Tabla 6‑1 Comparación entre manet, vanet y fanet 115](#_Toc146286692)

[Tabla 8‑1 RTT y porcentaje de pérdidas de mensajes, al variar la velocidad e intervalo 150](#_Toc146286693)

[Tabla 8‑2 Mensajes recibidos en promedio de cada nodo, retransmisiones en promedio son el número de mensajes únicos que reciben 160](#_Toc146286694)

[Tabla 8‑3 Probabilidad de recibir cierto número de mensajes y los tiempos de espera promedio, al variar el tamaño del mensaje, velocidad e intervalo. en una red con 10 nodos 163](#_Toc146286695)

[Tabla 8‑4 Comparación de los tiempos de encolamiento y probabilidad de recibir cierto número de mensajes, en una red con 20 vehículos 166](#_Toc146286696)

[Tabla 8‑5 Comparación de los tiempos de encolamiento y probabilidad de recibir cierto número de mensajes, en una red con 30 vehículos 169](#_Toc146286697)

# Introducción

En este capítulo se expone la temática del proyecto a desarrollar como tesis, se presenta el contexto sobre coches autónomos que nos lleva a la identificación del problema. También se exponen los objetivos del trabajo, junto a la metodología seguida, y se explica la organización del documento.

## Contexto

Los vehículos autónomos han recibido un gran impulso, facilitados gracias a avances tecnológicos. Algunos de ellos son la variedad de sensores con los que se puede contar, el incremento en el poder computacional que ha permitido el uso de la inteligencia artificial, así como el desarrollo de comunicaciones a alta velocidad y con bajas latencias como es 5G. Esto ha permitido la integración de soluciones que reducen la intervención humana a la hora de conducir vehículos. El objetivo es reducir accidentes, mejorar la movilidad, comodidad y agilizar el transporte.

Este desarrollo en coches autónomos se ha visualizado desde mucho tiempo atrás. En 1939 General Motors presentaba un primer concepto de vehículo autónomo. Se trataba de un coche que se conducía a sí mismo mediante campos electromagnéticos controlados por radio y operado por púas de metal incrustadas en el camino, el cual se hizo realidad en 1958. Con la carrera espacial se dio otro gran avance. Los científicos buscaban enviar vehículos a la Luna que incorporaban cámaras para procesar imágenes del camino. Desde entonces, muchos programas de investigación gubernamentales o académicos han surgido, como “DARPA Grand Challenge” en 2004. Esta competencia de vehículos autónomos celebrada en el desierto de Mojave en Estados Unidos incorporaba temas como aprendizaje de máquina. Los avances han continuado y hoy en día hemos llegado a puntos donde compañías han sacado a la venta opciones comerciales como Volvo y Google con WAYMO desde 2017, o TESLA, Audi y BMW. (Faisal et al. 2019)

Dentro de los vehículos autónomos se tienen en cuenta diferentes niveles de autonomía, según los cuales varían las tareas que realiza el auto y la intervención que se requiere de las personas. En general, las principales funciones que se ven involucradas son la localización, percepción, planificación, control y gestión. Se incluye también una parte importante de adquisición de información. Esta es resultado tanto del uso de sensores, de la comunicación y negociación con otros vehículos o infraestructura.

## Identificación del problema

Los avances alcanzados en los coches autónomos pueden tener un efecto transformador en la vida de las personas, al mejorar la seguridad al volante y poder reducir accidentes viales. Estos accidentes, en la mayoría de los casos son propiciados por el conductor (otros se pueden deber a fallas del vehículo o condiciones del camino). Tan solo en México en 2021 se registraron 340,415 accidentes de tránsito en zonas urbanas, cerca de 40 accidentes diarios, de los cuales el 17.8% resultaba en heridas, y el 1.1% era fatal. Cifras que comparadas con el año 2020 representaron un aumento del 12.8% (“Estadísticas a propósito del día mundial en recuerdo de las víctimas de accidentes de tránsito”, 2022). Las investigaciones realizadas han analizado y determinado los siguientes problemas a resolver: tareas de percepción, análisis del entorno, planificación de trayectorias y control del vehículo.

El ITAM cuenta con un proyecto sobre coches autónomos, que inicio gracias a la donación de un modelo a escala desarrollado por el profesor Raúl Rojas en Freie Universität Berlin. Además, se tiene el simulador realizado por el profesor Marco Morales junto con el equipo de Eagle Knights del ITAM.

El modelo de coche autónomo, llamado AutoModelcar, contiene sensores y actuadores, como cámaras y un LiDAR. Cada uno de estos, es controlable de forma independiente, pero carece de nodos que los integren en su totalidad para dotarle de un funcionamiento de forma autónoma. Lo anterior no es el único problema, el vehículo no contempla módulos para la comunicación entre otros carros, y la documentación con que se cuenta es deficiente. En proyectos anteriores, alumnos abarcaron la parte de planificación de movimiento, que le permite al vehículo identificar las líneas de los carriles y moverse en ellos. El proyecto pretende continuar y trabajar sobre los problemas de percepción y comunicación. Igualmente, resolver el problema de la falta de documentación, de forma que permita que otros alumnos puedan retomarlo con mayor facilidad.

## Objetivos

El primer objetivo del trabajo es implementar sobre el modelo de carro autónomo, un conjunto de algoritmos que le permitan la detección y rastreo de otros autos. El uso de técnicas como rastreo nos puede ayudar a determinar si determinados obstáculos son o no potenciales riesgos, según la trayectoria que estén siguiendo.

Como segundo objetivo, se busca simular una red vehicular vanet. Esta debe permitirnos notificar la presencia de posibles obstáculos para poder reducir y prevenir accidentes de tránsito. Por ejemplo, mediante la oportuna detección de otros vehículos, mismos que sean comunicados a otros autos mediante una red vehicular. Pues dichos obstáculos pueden encontrarse en puntos ciegos para el otro vehículo. Esta misma red en un futuro podría llegar a permitir la negociación entre vehículos para la planificación de rutas.

## Metodología

Como enfoque elegido para la elaboración del proyecto se buscó seguir una metodología ágil. La elegida fue programación extrema (XP), que tiene un enfoque orientado al desarrollo de software, de una manera eficaz, flexible y con un buen control en el proyecto. Como se observa en la figura 1-1, la metodología pone especial énfasis en la comunicación con el cliente para recibir retroalimentación constante y adaptar el diseño a las necesidades.

La metodología de cierta forma favorece la adaptabilidad sobre la previsibilidad, mediante la división de un problema complejo en pequeñas etapas simples. Cada una de estas etapas es acompañada de fases de pruebas desde los inicios del proyecto. Estas iteraciones rápidas presentan ventajas claras al proyecto. Por ejemplo, adaptar los métodos utilizados según los resultados obtenidos, y poder identificar de manera temprana lo que no funciona o lo que requiere de mayor atención.

Aquí el cliente participa de forma activa en las pruebas y con mayor control en el proyecto. Las etapas cortas toman apenas días en construirse y probar el código, lo que reduce el riesgo de tener que realizar grandes cambios. Pasadas las pruebas, estas etapas se deben integrar con el proyecto tan pronto como sea posible, a lo que se puede añadir un paso extra de limpiar el código y hacerlo fácil de entender. En resumen, es aceptar cambios en un proceso de desarrollo con ciclos que duran pocos días.

Diagrama

Descripción generada automáticamente

Figura ‑ etapas en la programación extrema. Fuente: (Pressman, 2010)

## Organización del documento

El presente trabajo se divide en 9 capítulos, los cuales abarcan dos problemáticas principales, la implementación sobre el modelo a escala de la detección y rastreo, y la simulación de la red vanet. En el capítulo 2 se presentan los requerimientos funcionales, no funcionales y restricciones, dadas principalmente por la plataforma sobre la que se trabaja. Posteriormente, se comenta el alcance del trabajo. Además, se presentan trabajos relacionados con los sistemas de detección, junto a las investigaciones que se llevan a cabo para la creación de modelos de redes vanets.

El capítulo 3 presenta el diseño. En él, se desarrollan los temas de detección y rastreo respectivamente. Se entra en mayor detalle en qué consisten estos problemas, se describen distintas técnicas o alternativas, junto a sus fundamentos matemáticos, y se discute el porqué de su elección.

En el capítulo 4 abarca la implementación o codificación de los algoritmos para rastreo y detección. Se describe con detalle desde el entrenamiento de los modelos, los pasos de preprocesamiento de imágenes, y se presentan los algoritmos en sí, tanto para usarse fuera como dentro del vehículo con ROS. Este capítulo cubre la codificación de los primeros ciclos, junto con las pruebas (se presentan los últimos resultados alcanzados para su aceptación).

Por último, en el capítulo 5 se comentan los resultados alcanzados. Para el nodo se miden tiempos y la precisión de los detectores.

A partir del capítulo 6 se desarrolla la segunda parte de este trabajo, las redes vanet. Este es otro capítulo de diseño que cubre un nuevo ciclo iterativo del proyecto. Aquí se plantean las problemáticas principales para el uso de redes vehiculares como la falta de infraestructura y la seguridad, se repasan los modelos principales, las tecnologías facilitadoras, junto a técnicas de enrutamiento.

En el capítulo 7 se presenta la arquitectura de una red vanet, junto con la estructura y formato de los mensajes que se transmitirían, así mismo se realiza la simulación de la red. Igual que con el capítulo 4, cubre los pasos de codificación. Mientras que los resultados se presentan en el capítulo 8.

Como cierre del trabajo en el capítulo 9 o conclusiones, se repasan y comentan los resultados obtenidos, se presentan áreas de oportunidades para mejorar. Igualmente, se incluye cómo se puede avanzar en la investigación del coche autónomo, y cómo se podría implementar físicamente en el AutoModelCar la red vanet. Por último, en el apéndice 1 se agrega la documentación realizada del coche autónomo, junto con la configuración y cambios realizados para su funcionamiento. En el apéndice 2, se presenta una guía o manual para el uso de ROS y OpenCV enfocado en la programación del AutoModelcar.

# Análisis

En este capítulo se analizan los problemas presentados en el capítulo anterior y los requerimientos para nuestra solución. Se presentan, los requerimientos, restricciones y el alcance, dados principalmente por el mismo modelo de coche autónomo AutoModelcar. Se habla un poco del estado del arte o trabajos hechos en el área de percepción en vehículos autónomos. En este caso nos centramos en la detección y rastreo de otros vehículos; y en la creación de un entorno colaborativo entre vehículos, redes vanet.

## Requerimientos

En la siguiente sección, se presentan los requerimientos del proyecto. Estos se dividen en dos: en requerimientos funcionales se detalla qué se espera que realice el sistema propuesto. Del lado de los requerimientos no funcionales, se especifica las métricas y criterios con los que se evalúa la solución presentada.

### Requerimientos funcionales

Para la primera parte, se requiere la programación de un nodo de ROS. Esto es, un archivo ejecutable que utiliza librerías para comunicarse con otros, los nodos publican o se suscriben a mensajes que son identificados bajo un tópico. El nodo programado lleva a cabo tareas de detección y rastreo, en las que debe ser capaz de identificar correctamente otros vehículos particulares presentes en su campo de visión. El entrenamiento no se centra en un tipo de vehículo en particular, sin embargo, tanto la simulación, como en físico solo se tiene un tipo de diseño (y en físico solo se tiene otro carro). La distancia máxima a la que se pueden identificar los objetos variaría en función de la cámara, pero para una imagen de 640x480 píxeles, el tamaño mínimo de las detecciones debe ser de 40 píxeles. Posteriormente, debe poder rastrear su movimiento, reconociendo entre dos frames a qué vehículo se refiere, aun cuando se oculten unos detrás de otros. Además, la solución presentada debe poder integrarse junto con el programa seguidor de línea que ya se tiene.

En el punto de redes vanet, se debe comprobar el funcionamiento y utilidad de protocolos como AODV y DSRC en la comunicación entre vehículos. Se deben poder identificar correctamente los requerimientos necesarios para alcanzar una comunicación entre nodos fiable (en la parte de la red vanet, por nodos nos referimos a los vehículos). Estos requerimientos, como el intervalo entre transmisión, se deben verificar en entornos cambiantes al variar la densidad de nodos y la velocidad de estos. En AODV se verifica la comunicación entre dos nodos, con motivos para coordinarse o verificar información. En DSRC se prueba que 9 nodos se comuniquen entre sí, para tareas de seguridad o emergencias.

### Requerimientos no funcionales

Para verificar la implementación realizada sobre el carrito, la ejecución de los nodos debe cumplir con ser rápida, esto se puede medir según el tiempo que tarde en procesar un frame. Se busca obtener la mayor frecuencia posible para alcanzar resultados en tiempo real. Por el sistema usado, debe alcanzar 5 Hz. Además, los detectores deben ser eficientes, mediante un set de validación, se evalúa su precisión y exactitud. Aquí sobre todo se busca conseguir la menor tasa posible de falsos positivos. Por último, la programación realizada debe funcionar tanto para el simulador, como el carro físico.

Por el lado de la red vanet, su diseño se quiere que esté basado en estándares. Dado su objetivo para aplicaciones de seguridad, la comunicación debe ser fiable. Para ello se evalúan parámetros como RTT, tiempos de encolamiento, tasas de bit en error y % de paquetes perdidos. Se quiere que el RTT sea menor a 200 ms, pues entre mayor sea, más distancia recorren los vehículos en ese intervalo.

## Restricciones

Debido a que el principal objetivo es presentar la implementación sobre el modelo a escala AutoNOMOS, se presentan algunas limitaciones a la hora de elegir qué métodos usar. En primer lugar, el modelo viene con la versión de ROS Indigo para crear los nodos, el cual permite programarlos tanto en C++ como Python. En el lado del simulador, se tienen más opciones para la versión de ROS, como Noetic.

Adicionalmente, tanto el carro, como el simulador nos limitan en cuanto a los sensores que podemos usar, el formato e información de los mensajes que nos proveen, como la cámara de profundidad. Otra limitante se refiere a la versión de OpenCV incluida, para el caso del carro se usa la versión de OpenCV 2.4, y para el simulador la 4. Se tratan de librerías de visión por computadora, que se usarán para el procesamiento de imágenes, y para aplicar técnicas de aprendizaje de máquina.

Una última restricción en los nodos se presenta para medir los tiempos y en la elección de los métodos. Aquí se depende del poder de procesamiento del carrito, como de la computadora. Esta misma restricción nos limita tanto la cantidad de pruebas que se pueden realizar, como poder probar otros enfoques o alternativas. Del lado de las redes vanet, si bien la elección del simulador es libre, nos limita a las implementaciones de los protocolos que tienen incluidas.

## 2.4 Alcance

Si bien es un problema muy amplio, el enfoque elegido busca quedarse en un caso más general. No busca llegar a la detección en entornos con problemas de iluminación, pero si con la presencia de múltiples vehículos. Tomando en cuenta las capacidades del modelo a escala AutoModelCar, la solución debe ser un sistema de detección que funcione en tiempo real, y que sea ligero computacionalmente. Mientras el sistema de rastreo debe poder identificar que un objeto es el mismo entre dos observaciones distintas, junto con lidiar con detecciones incorrectas u observaciones faltantes. Debido a que estos van acarreando errores en la predicción.

Del lado de la red vanet, por razones de tiempo, únicamente nos limitamos a experimentar sobre uno de los estándares. Sobre éste nos enfocamos en sus tareas relacionadas con la seguridad. Además, se limita a la comunicación entre vehículos, no considera el uso de infraestructura. Esto busca sentar las bases para el modelo del carro, más no llega a implementarse.

Para las simulaciones, solamente se estudian un par de escenarios. En ellos los nodos recorren una misma distancia a diferentes velocidades (110 metros en la primera y 330 en la segunda). Otros acercamientos que no se cubren, pueden ser probar los cambios utilizando otros entornos con más obstáculos, o usar otros modelos de path loss.

## Trabajos relacionados

En la investigación sobre coches autónomos, una de las áreas de gran relevancia y que ha tenido grandes avances en los últimos años es la visión por computadora. En este campo se tienen dos aproximaciones: enfoque modular y monolítico de extremo a extremo. El primero y más apoyado consiste en partir de un modelo complejo de altas dimensiones a variables controlables de bajas dimensiones. Cada una separables en módulos que pueden entrenarse y probarse de forma independiente (por módulos se habla de fragmentos de programas que funcionan y realizan tareas de forma independiente, los cuales, mediante entradas y salidas, se interconectan con otros). Estos módulos normalmente se dividen en: percepción, análisis del entorno, planificación de trayectorias y control del vehículo. Típicamente con técnicas de aprendizaje de máquina como redes neuronales en los campos de percepción y análisis. Mientras que para planificación y control se usan enfoques tradicionales como máquinas de estado, algoritmos de búsqueda y modelos de control. Por un lado, un enfoque modular permite un resultado más fácil e interpretable para las personas. Sin embargo, no necesariamente más eficiente o rápido que el enfoque monolítico. Considerando que cada problema se ve de forma independiente. (Janai et al. 2021)

Si bien el crecimiento en aplicaciones de aprendizaje de máquina ha sido grande, se sufre de problemas como asignación de pesos, sobreajuste, o en el caso de redes neuronales de una difícil interpretación. Junto con estas técnicas, los coches autónomos van acompañados de una gran variedad de sensores, como cámaras, odometría de las ruedas y de rango (LiDAR y sonar). De estos, las cámaras se han beneficiado mucho, gracias al ser baratas y fáciles de usar, (Janai et al. 2021). Como el coche de Intel Mobileye demos equipado solo con ellas.

Cuando se utiliza el enfoque basado en aprendizaje de máquina se requieren de grandes *datasets* para el entrenamiento de los modelos. Por lo que, para avanzar en las investigaciones, se han puesto de forma libre diversos *datasets* para la detección, rastreo y estimación de movimiento. Por ejemplo, se cuenta con KITTI, uno de los *benchmarks* más famosos, u otros proyectos públicos como ApolloScape que proporcionan imágenes y anotaciones.

Para la clasificación, hay enfoques tradicionales de aprendizaje de máquina, en los que se proponen el uso de un conjunto de clasificadores de cascada, o HOG+SVM. Los cuales funcionan relativamente bien con pequeños *datasets*, son baratos computacionalmente y fáciles de interpretar. Aun así, el enfoque dominante son las redes neuronales convolucionales, usados para realizar tanto detección completa como por partes. Esto es gracias a que son más adaptables, transferibles, de mayor precisión y escalan con los datos. Un enfoque que aún parece ser insuficiente, en especial, al actuar en entornos nuevos, con diferentes condiciones de clima.

Algunos ejemplos que toman enfoques tradicionales se encuentran en los artículos de “Linear SVM classifier based HOG car detection” (Bougharriou, et al, 2017); ó “A Two-Stage Approach to People and Vehicle Detection With HOG-Based SVM” (Feng Han et al, 2006). Quienes proponen el uso de hog con svm pensado principalmente como un algoritmo ligero y rápido.

Pensando específicamente en el carro, se han utilizado distintas técnicas para las tareas de reconocimiento y evasión de obstáculos. Una de ellas realizada en el ITAM que usa en su totalidad redes neuronales, pero solo se implementó en el simulador, no el carrito físico (Zetina, 2019). Otros ejemplos suelen confiar esta tarea exclusivamente al uso del LiDAR, pero no llevan pasos para reconocer el objeto, ni rastrearlo (Ayala, 2019). Por último, orientado más al reconocimiento de señales (no de obstáculos), se suelen usar métodos como filtros de cascada (Pérez, 2017) o árboles kd (Pérez, 2023).

Con el objetivo también de mejorar la seguridad, el campo de las comunicaciones entre vehículos ha tenido un gran avance. Desde nuevos protocolos, tecnologías, y técnicas para tener mayores tasas de transmisión, a menores latencias y más confiables. Gracias en parte a 5G, *fog computing*, y antes 802.11p. Los cuales han permitido pensar en nuevas aplicaciones, como su uso para redes vanet. Aquí se tienen dos tendencias, vehículo a infraestructura y vehículo a vehículo.

En vehículo a vehículo se ha explorado desde la tercera generación de telefonía celular con la 3GPP. A manera de servicios que permiten a dispositivos comunicarse entre sí cuando están en proximidad (Tariq Islam y Cheolhyeon Kwon, 2022). El debate aquí viene desde usar bandas licenciadas (asignar nuevos espacios o reusarlas), o no licenciadas; qué técnicas de acceso y asignación de recursos usar; así como el uso de enfoques tradicionales o aquellos basados en aprendizaje de máquina.

Entre los avances se ha logrado la estandarización de algunos protocolos como DSRC. Sobre él se han reservado inicialmente 75 MHz en la banda de 5.9 GHz para usuarios con licencia en aplicaciones de sistemas de transporte inteligente. Pero debido a la falta de adopción, se reacomodaron 45 MHz de los 75 en los 5.8 GHz para usuarios sin licencia. (Jakubiak y Koucheryavy, 2008)

Las investigaciones se centran desde el tipo de tecnología. Por un lado, tecnologías celulares con amplia cobertura y seguridad, pero de alto costo y latencia. Por otro lado, el estándar de la IEEE 802.11p sobre la banda 5.8-5.9 GHz, promovido por la industria automotriz de Europa y USA. Discusiones respecto a las técnicas de *broadcasting*, tanto de banda estrecha similar a radio FM, como banda ancha. Técnicas basadas en localización y *clustering* que se limiten a solo un sitio de interés. Protocolos de enrutamiento, con métodos como reenvió oportunista, basado en trayectoria o geográfico. En este último se deben lidiar con problemas como el cambio en la topología y escalabilidad. Otros temas como la potencia de transmisión, el ajuste de la potencia y problemas de interferencias. Y, por último, pero no menos importante, la seguridad y privacidad, pues pueden suponer grandes peligros la transmisión falsa de mensajes, por lo que deben ser redes confiables y resilientes.

Se tiene múltiples programas orientados a la cooperación entre vehículos. Algunos ejemplos son “Inter-Vehicle Hazard Warning” en Europa, conducción cooperativa como CarTALK 2000, PROMOTE-Chauffeur e INVENT VLA. Igualmente, consorcios como Car2Car también en Europa o “Vehicle Safety Communication” en USA (Jakubiak y Koucheryavy, 2008).

Finalmente, ya hay vehículos en el mercado con capacidades para comunicarse de la forma V2X (vehículos a todo), como General Motors que usa DSRC, o Audi que usa una combinación entre DSRC y Celular.

# Diseño

En este capítulo se aborda el tema de la detección de objetos, junto con algunos pasos previos para el preprocesamiento de imágenes. Se presentan las técnicas de detección usadas en el trabajo y se comparan contra las redes neuronales. Además, se cubre el tema de rastreo de objetos, visto desde técnicas tradicionales. Principalmente se trata el filtro de Kalman, el cual se puede resumir como la predicción del estado siguiente, a partir de la medición de los estados anteriores, con una etapa de corrección al recibir una nueva lectura.

## Arquitectura

La primera parte del trabajo se centra en el desarrollo de un nodo de ROS encargado de las tareas de percepción para el carro. Con el objetivo de hacer reconocimiento de vehículos, el problema involucra la implementación de algoritmos para detección de movimiento. Esto se trata de un campo bien estudiado en el caso de cámaras estáticas, pero que presenta inconvenientes si la cámara está en movimiento. De manera que el problema requiere separarse en dos partes:

● Detección de objetos, en este caso vehículos

● Rastreo de los objetivos, los vehículos detectados en el paso anterior

En base a esto, se realiza entonces la programación de un nodo de ROS para el vehículo a escala AutoModelcar. Si bien podría separarse en dos nodos (archivos), esto no es conveniente por lo interconectados que están, pues la parte de rastreo se usa también en las tareas de detección. Asimismo, dadas las restricciones de alcanzar un resultado rápido en el carro, nos limitamos a programarlo en C++ (que corresponde al estándar de la ISO JTC1/SC22/WG21). Al ser Python un lenguaje interpretado, y no compilado, lleva a una pérdida importante de rendimiento.

Para el nodo se realiza un solo programa con 3 versiones diferentes, uno para las pruebas usando video o imágenes, otro para el simulador y el último para el carro físico. La diferencia de los últimos dos, se debe a que entre el carro y el simulador hay suficientes cambios en las versiones, el primero usa ROS Indigo y OpenCV 2.4, mientras en el segundo se usa ROS Noetic y OpenCV4. Esto se debe a la falta aún de estandarización de ROS, el cual está organizado por distribuciones, cada una con diferentes versiones de paquetes y requerimientos de compatibilidad. De forma similar, OpenCV son bibliotecas de visión por computadora de código abierto, manejadas por versiones. Al no estar estandarizadas, algunos puntos como las máquinas entrenadas o funciones no son compatibles entre sí.

Otros programas complementarios que se requieren son para el entrenamiento de los detectores, para probarlos, y uno más para realizar ajustes para en el rastreo. Por último, para el carro se realiza un nodo simple para la evasión de obstáculos, que se integra con el de conducción: lane\_navigation simulador\_gary.launch.

Los nodos programados para el carro se observan en la figura 3-1. En ella, la cámara comunica su imagen mediante un mensaje (con tópico: /app/camera/rgb/image\_raw), al que se suscribe el nodo programado de detección y rastreo. Este nodo, publica un arreglo con las detecciones que es leído por changeLane, que se encarga de ver si debe cambiar de carril. En general, la conducción sobre los carriles está controlado por los nodos lane\_navigation. Por lo que, en el cambio de carril se debe detener el funcionamiento del seguidor de línea. Esto se hace a través del nodo translate\_sim\_real. Tanto translate como change\_lane se comunican con manual\_control, el cual controla los motores a partir de los mensajes de velocidad y Diagrama

Descripción generada automáticamentesteering.

Figura ‑ En el recuadro es el nombre del nodo, en los círculos los tópicos que publican. Son los tópicos los que interconectan con otros nodos. El nodo manual\_control, a partir de la velocidad y steering se comunica con los motores. Lane\_navigation en realidad son un conjunto de nodos que se encargan del seguimiento de línea.

El coche autónomo cuenta con una cámara de profundidad Intel Realsense versión r200. La cámara cuenta con una resolución de 640x480 píxeles, con una frecuencia de 30 fps. El cual publica un mensaje de tipo sensor\_msgs/Image, que está formado por los campos:

[std\_msgs/msg/Header](https://docs.ros2.org/latest/api/std_msgs/msg/Header.html) header

uint32 height

uint32 width

string encoding

uint8 is\_bigendian

uint32 step

uint8[] data

Mensajes como éste, forman parte de paquetes que contienen formas estandarizadas para la representación de los datos, en el fondo todos usan std\_msgs compuesto por tipos de datos comunes, como enteros, flotantes y strings.

Los formatos (“encodings”) usados para las imágenes no son estándares, sino formas típicas para describir cómo se representan los datos en un píxel. ROS mediante los mensajes de sensor\_msgs/Image soporta varios tipos de formatos de OpenCV, Bayer y otros comunes como mono8, bgr8, bgra8 y bgr16. En los casos de imágenes a color, ROS no publica una matriz diferente para cada canal, por el contrario, los empaqueta en un solo valor. Para su uso, se debe acceder a ellos bit por bit (con ayuda del valor step), o en su lugar hacer uso de librerías.

La imagen entregada por la cámara es en formato RGB8, para usar la imagen en OpenCV nos apoyamos de cv\_bridge. Se trata de una librería de ROS que desempaqueta los datos de un mensaje imagen y los convierte a un formato de OpenCV. Por ejemplo, uno de los usados es CV\_8UC1, el cual se refiere a que cada píxel se representa con un valor de 8 bits sin signo, con un solo canal.

(“cv\_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages - ROS Wiki”, s. f.)

El nodo programado devuelve 3 mensajes. Los primeros dos también de tipo sensor\_msgs/Image, uno publica la imagen con la detección marcada, y el otro con la predicción. El tercer tipo de mensaje informa de las posiciones de los otros vehículos, y es el que se conecta con la evasión de obstáculos, se compone de:

|  |  |  |
| --- | --- | --- |
| detec.msg  uint8 x  uint8 y  uint8 width  uint8 height | detecTiempos.msg  detec posA  detec[] posSig | detecArray.msg  uint8 numDetec  detecTiempos[] array |

Son 3 mensajes los que se definen. Primero detecArray es un arreglo que indica el número de detecciones. En el mensaje detecTiempos, para cada una de las detecciones se da su posición actual y las siguientes 10 posiciones predichas. Estas posiciones corresponden al mensaje detec, y siguen el mismo formato que un objeto de tipo rectángulo en OpenCV: x, y, ancho y altura. Dentro del programa se requiere el uso de las librerías cv\_bridge para transformar el mensaje de la cámara a una matriz de OpenCV. Las librerías de OpenCV para el procesamiento de imágenes y aprendizaje de máquina. Los mensajes sensor\_msgs y la librería de ROS.

Como detectores se elige hacer una combinación de filtros de cascada con máquinas de soporte vectorial. Mediante los filtros de cascada se realiza una detección por ventana deslizante, con la que localizar candidatos. La ventaja de los filtros de cascada sobre otras técnicas se debe a su gran velocidad, a costa de precisión. Sobre este método, para la extracción de características de la imagen, por cuestión de tiempo en su entrenamiento, se elige Local Binary Patterns. El uso de máquinas de soporte vectorial con histograma orientado por gradiente (svm+hog) sirve como paso de refinamiento, para descartar los falsos positivos arrojados por el filtro de cascada. La elección de esta combinación de métodos busca conseguir maximizar la precisión y velocidad en la detección.

Del lado del rastreo, la elección fue el uso del filtro de Kalman sobre una imagen 2D. Al tener como paso previo la detección, no se requiere de técnicas de aprendizaje de máquina. Igualmente, otros métodos como filtro de Kalman extendido son innecesarios y costosos, pues el rastreo del movimiento sobre la imagen puede tomarse como un problema lineal.

El nodo entonces recibe la imagen, realiza algunos pasos de preprocesamiento, y entra a la detección. Según el número de frame, se realiza la detección del frame completo, o solo de regiones, con el uso de filtros de cascada y máquinas de soporte vectorial (figura 3-2). Hecha la detección, se pasa al rastreo, donde se crea un nuevo filtro de Kalman, o de lo contrario se actualiza (figura 3-3). Para los tres métodos, nos limitamos a los algoritmos implementados en OpenCV.

Diagrama

Descripción generada automáticamente

Figura ‑ Pasos para la detección

Diagrama

Descripción generada automáticamente

Figura ‑ Pasos para el rastreo

Dentro del filtro de Kalman se tienen 3 matrices principales: la de medición, transición y control, esta última no se usa. En el caso de transición se tienen 6 variables: la posición y velocidad en (x, y), el ancho y alto de la detección. Para la de medición se tienen 4 variables: la posición en (x, y), el ancho y alto.

El primer paso para entrenar estos detectores es preparar un dataset de carros como imágenes positivas, y uno de negativas que contiene imágenes como árboles, de la carretera o señales. Para el entrenamiento, se realiza un programa que lee las imágenes, las pasa por una fase de preprocesamiento (se escalan y se aplican filtros como blur gaussiano), se extraen sus características y se pasa al entrenamiento. Primero se prueba con un set de imágenes pequeño para probar grids de parámetros y encontrar los mejores valores, entonces se pasa al entrenamiento con el set completo. Esto se ilustra en la figura 3-4.

Diagrama

Descripción generada automáticamente

Figura ‑ Pasos para el entrenamiento de máquinas de soporte vectorial

En el entrenamiento de los detectores, aunque no tengan un estándar, normalmente se sigue una serie de pasos como buenas prácticas. Desde los que involucran la colección de datos (por ejemplo, para evitar sesgos), optimización parámetros, preprocesamiento de las imágenes (normalizar y reducir el ruido), y buenas prácticas para evaluar los resultados (como separar el set de entrenamiento del de validación). Se busca que los modelos sean replicables y con resultados interpretables.

Pasando al último nodo, para la interconexión con el seguidor de línea, se realiza un programa que cambie de carril según la detección. Este lee los mensajes con las detecciones definidos anteriormente, y en base a la predicción, realiza maniobras para esquivar obstáculos. Los mensajes que manda son de tipo sensor\_msgs/speed y sensor\_msgs/steering, ambos de tipo int16. Asimismo, se manda otro mensaje también de tipo int16 para interrumpir el programa de conducción.

En las siguientes secciones se desarrollan las dos tareas principales que abarca el nodo para la percepción: detección y rastreo. En cada una de ellas se comentan los métodos usados, las razones de su elección, y otras alternativas. Sobre cada una de ellas se desarrollan con mayor profundidad en qué consisten.

## Detección de objetos

La detección de objetos es una parte muy importante de la conducción autónoma, pues nuestro vehículo se encontrará en ambientes con otros participantes del tráfico, desde vehículos, personas, hasta animales. Para el objetivo del trabajo, nos centramos únicamente en vehículos, pero las mismas técnicas se pueden replicar a otras clases de objetos.

Dada la problemática y limitaciones del problema, donde se requiere realizar la detección desde una cámara en movimiento de objetos móviles y con formas complejas, el enfoque dominante se ha centrado en técnicas de aprendizaje de máquina. De entre ellas, la que más impulso ha recibido se trata de aprendizaje profundo con redes neuronales. Aun así, se tienen otros enfoques como el presentado, que combina máquinas de soporte vectorial y clasificadores de cascada. Estas técnicas tienen que ser lo suficientemente precisas y robustas, pues errores en la clasificación pueden presentar un gran riesgo en la seguridad de las personas. Si bien ambos tipos de errores son importantes, nuestra prioridad sería reducir los falsos positivos.

Existen otras ideas que no requieren de aprendizaje de máquina, como el uso de modelos que compensen el movimiento de las cámaras para usar técnicas tradicionales como Optical Flow. Pero resultan bastante más complejas y no suelen tener buenos resultados.

A diferencia de los humanos que somos capaces de reconocer objetos parcialmente visibles, gracias a que nuestra mente compensa las partes faltantes e invisibles. Esto se convierte en problemas complejos para la computadora, que no solo debe lidiar con aprender a identificar una clase, si no que se presentan problemas como la iluminación. A este problema se le conoce como oclusión, y ocurre cuando el objeto está oculto ya sea por otro objeto del mismo tipo “intra-class occlusion” o, por otro objeto o elemento “inter-class occlusion”.

Con la aparición de oclusión, el rendimiento de clasificadores empeora drásticamente. Una solución parecería ser crear diferentes clasificadores que lidien con la oclusión, pero aparecen nuevos problemas como la falta de datasets para poder identificar la presencia de oclusión (Saleh, Szénási, y Vámossy, s. f.). Por lo que, para este trabajo nos centramos en entrenar clasificadores para la detección de vehículos de día. Que traten de ser lo más generales para lidiar con problemas de oclusión como sombras, más no se separa el problema. Además, en casos como en la noche se pueden mejorar los resultados con el uso de otros tipos de cámaras.

El camino clásico para la detección de objetos consiste primero en una etapa de preprocesamiento de la imagen, la extracción de una región de interés, la clasificación del objeto y una etapa de verificación o refinamiento.

Para el preprocesamiento, se realizan tareas de ajuste de ganancias, rectificación de imagen o calibración de las cámaras. (Janai et al. 2021) Típicamente, se añaden filtros, se escalan las imágenes y se convierten a otros formatos, adicionalmente. Según el problema se puede agregar el uso de filtros de bordes o técnicas de compresión de imagen.

Para nuestro caso, la imagen primero se reescala para usarla en un clasificador. A continuación, se pasa a escala de grises, esto ayuda a simplificar los algoritmos y reducir la complejidad, ya que no se requiere que el modelo aprenda ciertos colores. Esto en cierta forma comprime la imagen al reducir los 3 canales a 1.

Además, se incluye el uso de filtros, los cuales ayudan a introducir imperfecciones a los datasets y nos permite que los modelos sean más robustos al mundo real. El uso de filtros de difuminado ayuda a esto, degrada la calidad de imagen y reduce también la necesidad de contar con datasets más grandes.

El difuminado en imágenes consiste en tomar píxeles vecinos y sacar un promedio de ellos, lo que reduce el detalle. (Nelson J. 2020) En OpenCV se presentan distintas técnicas para aplicar blur, algunas mejores que otros según sus aplicaciones.

El suavizado de imagen resulta de convolucionar la imagen con un filtro pasa bajas. Esto elimina las altas frecuencias que consisten en ruido y bordes de la imagen, lo que produce el suavizado. Algunos tipos son:

* Promedio

Se usa un kernel normalizado para la convolución, el cual toma el promedio de todos los píxeles bajo el área del kernel.

* Blur gaussiano

En lugar de usar una matriz, se usa un kernel gaussiano en el que se especifican el ancho y altura del kernel, junto a la desviación estándar. A diferencia del otro que promedia todo, el píxel central recibe el mayor peso, mientras los pixeles vecinos reciben pesos más pequeños al alejarse del centro.

Alternativamente se puede implementar aplicando dos veces el filtro gaussiano de una dimensión. Este método sería similar a aplicar motion blur en ambas direcciones. Pero no da el mismo resultado que la fórmula mostrada, al no considerar los bits de las esquinas de una celda. (“OpenCV: Smoothing Images” s. f.)

* Desenfoque de movimiento

Es un tipo de blur para darle un efecto a imágenes de dirección. Para crear un filtro se colocan 1’s sobre la matriz con la dirección del vector deseado, en el ejemplo se muestra un blur horizontal:

Dentro de ellas se prueba únicamente con el filtro gaussiano. Este nos ayudará a agilizar el entrenamiento de los clasificadores, al aprender de forma más rápida y general la forma de los coches. Aunque para el trabajo no se incluya, usar desenfoque de movimiento ayudaría a la detección de vehículos a altas velocidades, más que para aprender su forma. Para usarse entonces, la mejor idea sería duplicar el dataset que se tiene, y a uno de ellos aplicárselo.

Después del preprocesamiento se debe extraer las regiones de interés. El método más común es el de deslizar una ventana sobre la imagen e ir escalando el filtro o la imagen. Esto puede resultar en un proceso muy caro, según qué tan exhaustiva sea la búsqueda. Por lo que, igual que en este trabajo, se deben asumir algunas cosas, como el tamaño de la ventana, la posición de los candidatos y la proporción con la cual se incrementa el tamaño de la ventana, lo que reduce el número de comparaciones.

Otras alternativas son búsquedas selectivas, en lugar de recorrer toda la imagen, se seleccionan solo algunas localizaciones aproximadas (Janai et al. 2021). Esto se comenta en el trabajo, pues permitiría dividir la imagen en regiones y según cada una aplicar un clasificador u otro. Una forma de hacerlo puede ser combinando el uso del LiDAR, y elegir como regiones los obstáculos detectados cerca.

### Técnicas de detección

Algunos métodos populares experimentados desde el punto de vista clásico son los usados en el trabajo: clasificadores de cascada y máquinas de soporte vectorial (SVM) con histograma orientado por gradiente (HOG). En ellas, se ha observado que SVM con HOG trabaja mejor a más altas resoluciones, mientras presenta un mayor tiempo de procesamiento. Del lado de filtros de cascada funciona mejor a bajas resoluciones, alcanzando resultados prácticamente en tiempo real. Otros métodos son “Scale-Invariant Feature Transform” y “Speeded Up Robust Feature”. Todos ellos dependen de extraer manualmente la información de las imágenes, son difíciles de diseñar y limitados en sus capacidades de representación (Janai et al. 2021). También está el uso de redes neuronales convolucionales como YOLO, MobileNet, Fast R-CNN.

En las redes neuronales de preferencia trabajan con la imagen preprocesada, pero también aceptan el uso de imágenes raw. Usar las imágenes preprocesadas puede ayudar a reducir sesgos y mejorar el rendimiento. Hay algunas redes que ya están diseñadas para aplicar algunos pasos, como normalización.

Sobre las técnicas presentadas se requieren extraer características de la imagen. Para cascada se usa local binary patterns que captura la intensidad de los píxeles vecinos alrededor de cada punto de una imagen. Con él se crea una un patrón binario que permite capturar características como bordes, esquinas y texturas. Para svm se usa histograma orientado por gradiente que calcula los gradientes de intensidad de una imagen dividida en celdas, al capturar vectores que permite de forma similar identificar bordes y texturas de una imagen. La diferencia, es que LBP captura características locales para describir una imagen completa, mientras hog ve la imagen global. Hog captura formas y contornos, LBP patrones repetitivos.

Los filtros de cascada calculan las características de una imagen y la pasan sobre un conjunto de funciones que evalúan sets características locales. Si las características están presentes y cumplen cierto margen, avanzan al siguiente filtro, de lo contrario se descarta. Estos sets de características contienen un histograma de píxeles que definen patrones sobre una región de la imagen.

Las máquinas de soporte vectorial encuentran el mejor hiperplano que separa los ejemplos positivos de los negativos. Al aplicar una nueva imagen para clasificar, se extraen sus características y se aplican sobre la función del hiperplano calculado. Según la distancia y un margen, predice si pertenece a la clase. La función de los kernel es transformar un problema a más altas dimensiones donde se pueda separar linealmente. Con Radial basis function permite introducir no linealidades a la función, que le permite encontrar fronteras más complejas (en este caso formas de elipsoides), con las que aprender la forma del objeto.

De los métodos presentados, los filtros de cascada son clasificadores débiles, mientras svm es fuerte. Por débiles se refiere a clasificadores con capacidad y complejidad baja, con poca precisión. Con Adaboost, se suele usar una serie de clasificadores débiles de forma iterativa para alcanzar la precisión de uno fuerte. En cambio, un clasificador fuerte puede realizar predicciones de clases correctas por sí solo, tiene alta precisión.

### Clasificador de cascada

Se trata de un método propuesto por Paul Viola y Michael Jones en "Rapid Object Detection using a Boosted Cascade of Simple Features" del 2001. Se trata de un enfoque donde una función en cascada, en este caso un grupo de funciones que en su conjunto componen el clasificador, se revisan de forma secuencial que cumplan ciertas condiciones. En este caso evalúan algunas características (píxeles de la imagen) que determinan si el objeto puede o no pertenecer a una clase. Cada filtro evalúa solo una clase, y requiere ser entrenado por un gran set de imágenes positivas y negativas.

Primero se requiere preparar un set de imágenes positivas (ejemplos representativos de la clase u objeto que se quiere detectar) e imágenes negativas. El siguiente paso es extraer las características de la imagen, aquí se pueden tener diferentes métodos como haar o lbp para extraer la información de la imagen. De forma que, en lugar de trabajar con la imagen completa, se reduzca la complejidad del problema obteniendo solo información relevante como de los bordes.

El método elegido para extraer las características es LBP o Local Binary Patterns. Tiene la ventaja de que su entrenamiento es mucho más rápido que haar, pero menos preciso. Presentado por Ojala et al. en 1996, en LBP, para cada píxel se toma una celda alrededor de 9 píxeles (3x3). Estos píxeles se comparan con el centro, y a los vecinos se les asigna un valor de 1 si la intensidad con respecto al centro es mayor o igual. En sentido de las manecillas del reloj, cada valor se multiplica por una potencia de 2. Esto nos permite promediar los alrededores asignando un valor entre 0-255.

Xc, yc es el píxel del centro, ic es el brillo del centro, ip el brillo de un píxel vecino, s función signo. Se asigna 1 si es mayor o igual, de lo contrario es 0. Para calcular el valor del centro, como se observa en la figura 3-2, se puede iniciar desde cualquier píxel y movernos en sentido de las manecillas del reloj, a cada bloque asignándole un valor binario y calculando la suma. Aquí el problema se reduce en comparar dos medidas: el contraste en escala de grises y patrones espaciales locales. El resultado se puede observar en la imagen 3-3. (Pietikäinen 2010) (Rosebrock 2015)

Calendario

Descripción generada automáticamente

Figura ‑ Ejemplo de la operación realizada en lbp, primero si el valor del pixel es menor al centro se toma 1, a continuación, partiendo de la esquina superior derecha se le asigna como 2^0, de ahí se da una vuelta en sentido de las manecillas del reloj, aumenta. Fuente: (Rosebrock 2015)

Imagen que contiene foto, hombre, gato

Descripción generada automáticamente

Figura ‑ Características extraídas de una imagen con lbp. Fuente: (Rosebrock 2015)

Calculadas las características, la mayoría resultan ser irrelevantes. Para elegir las mejores características se usa Adaboost, mismo que le da el nombre de cascada, son un grupo de predictores sencillos que se colocan en secuencia. El procedimiento es aplicar todas las características obtenidas sobre el set de entrenamiento completo, y para cada característica encontrar el mejor umbral que las clasifique de forma correcta. Como no todas las características son relevantes, solo se escogen las que tengan la menor tasa de errores. En un principio todas las imágenes tienen el mismo peso, después de cada clasificación, se incrementa el peso de las imágenes que son erróneamente clasificadas y se calcula la nueva tasa. El proceso se repite hasta alcanzar la precisión indicada, o alcanzar cierto número de características.

Finalmente, cada característica evaluada corresponde a un clasificador débil, que por sí solos no pueden decidir sobre la imagen completa. Lo que hace es realizar la suma ponderada de los clasificadores, a diferencia de SVM que es un clasificador fuerte. Cuenta con la ventaja, de que si la imagen es negativa la puede descartar sin evaluarla por completo, de modo que funciona por etapas. (“OpenCV: Cascade Classifier” s. f.)

Como se mencionó una alternativa es usar características Haar. El cual, en lugar de crear histogramas, aproxima las características mediante rectángulos. Este método alcanza mejor precisión, pero por cuestiones de tiempo en el entrenamiento no se usa. Para este tipo de clasificador, OpenCV cuenta con un módulo para su entrenamiento. Sobre el nodo detector, a partir de la imagen ya preprocesada se ingresa al clasificar que usa la detección por ventana deslizante. La idea es que, gracias a su velocidad, sirva para detectar candidatos, y como refinamiento se usa svm.

### Hog+svm

Se trata de la combinación de dos técnicas, máquinas de soporte vectorial o svm que es un algoritmo de aprendizaje supervisado, y de histograma orientado por gradiente que permite extraer las características de la imagen.

Para empezar, el histograma orientado por gradiente es una técnica que cuenta las ocurrencias de vectores gradiente en una porción de una imagen, estos se componen de magnitud y ángulo. Para esto, primero se toman bloques de 3x3 píxeles, y se calculan los gradientes Gx y Gy para cada píxel, con la fórmula:

Lo que hace es tomar las intensidades de los píxeles anteriores y posteriores. Estos gradientes, horizontal (Gx) y vertical (Gy) se calculan multiplicando por el kernel:

Un conjunto de letras negras en un fondo blanco

Descripción generada automáticamente con confianza media

El siguiente paso es formar los histogramas, para eso se calcula la magnitud y ángulo de cada píxel:

La imagen que fue dividida en celdas, son agrupadas para formar un bloque al que se le asocia un histograma con n valores diferentes llamados bins. Cada punto del histograma toma valores entre 0 a 360 grados, con 9 bins se tiene un salto de 20 grados (toma valores de 0, 20, 40 … 360). Esta aproximación en celdas y bins permite tener una representación más compacta de la imagen. La contribución individual de las celdas que forman un bloque crea el histograma. (Tyagi 2021)

**SVM**

En el caso de máquinas de soporte vectorial son clasificadores que, a partir de un set de entrenamiento, se encuentra un hiperplano óptimo (figura 3-4) que separe los datos en dos clases representadas como -1 y 1. Este método es fácil de entrenar y permite una clasificación relativamente rápida.

Gráfico

Descripción generada automáticamente

Figura ‑ Ejemplo de la separación de dos variables por un margen de separación, se muestra la separación óptima que logra el mayor margen entre las dos variables, fuente: (“opencv: introduction to support vector machines” s. F.)

Donde a son los pesos del vector, b el sesgo, y x los datos. Además, por convención el hiperplano se escala de forma:

Los ejemplos de entrenamiento más cercanos al hiperplano son los que definen el margen de separación y son llamados vectores de soporte.

En el entrenamiento se usa una función de pérdida o hinge loss. Si la predicción es incorrecta la penalización es grande cuanto más lejos del margen está. Si la predicción es correcta y está cerca del margen, recibe una pequeña penalización. Si es lejos el hinge es de 0.

Para clasificar se aplica la función signo:

La idea es tener el mayor margen posible, de tal manera que se reduzca la probabilidad de error.

Para hallar a y b se tienen dos funciones una de error de entrenamiento y un término de regularización para evitar grandes escalas. La función de costo es:

El primer término es la distancia de los datos al margen de separación, el segundo la regularización. Función que se minimiza por el método de descenso por gradiente para hallar los valores de a y b. En este método se define aleatoriamente un tamaño de paso y una dirección, tal que minimice la función de costo:

Las máquinas de soporte vectorial usan un hiperplano lineal para separar las clases. Para casos donde los objetos no se pueden separar de forma lineal, primero se realiza un paso para poner las características en un espacio de mayor dimensión. La transformación se puede realizar por diferentes mapeos no lineales. Radial consiste en (RBF es el usado en este trabajo el cual mapea la función radialmente) (“OpenCV: Introduction to Support Vector Machines” s. f.) (Kecman 2005)

Texto

Descripción generada automáticamente

Figura ‑ Tabla de diferentes kernels para linealizar un problema. Fuente:(Kecman 2005)

El entrenamiento de este detector requiere hacerse manualmente, ya sea ingresando los parámetros o buscando los mejores a través de grids. Previo a la implementación en el nodo, requiere probarse variando tanto el kernel usado, como la configuración de HoG (por ejemplo, en cuántas celdas dividir la imagen) para buscar los mejores resultados. Completado el entrenamiento, en el nodo se incluye únicamente como predictor. Sobre los candidatos arrojados por cascada, las regiones se deben reescalar nuevamente e ingresarse para clasificar la clase. Por su lentitud, no se usa para buscar dentro de la imagen.

### Comparación con redes neuronales convolucionales

Las redes neuronales en general crean filtros que aprenden características de la imagen. Al aplicar una nueva imagen, el filtro busca la forma (características) sobre ella y recibe una calificación de qué tan parecida es. Esto avanza hasta crear la imagen completa.

Las redes neuronales convolucionales se especializan en el tratamiento de imágenes y audio. Son compuestas de 3 capas principales: capa convolucional que analiza las imágenes y detecta la presencia de características; capa de pool que reduce el tamaño y preserva solo las características importantes; opcionalmente alguna capa de activación como Sigmoid para devolver valores de 0 o 1; y finalmente la capa “Fully Connected” que conecta con otras neuronas. Con cada capa apilada permite identificar características más complejas, hasta llegar al objeto. (“What Are Convolutional Neural Networks? | IBM” s. f.)

En redes neuronales se puede usar otro método para extraer la región de interés, en lugar de ventanas deslizantes, llamado región proposal network. Este permite clasificar regiones y aplicarla dentro de alguna categoría. Region proposal toma una imagen y como salida devuelve sets de propuestas de objetos sobre los que buscar. Esto ha permitido que ganen un impulso en su velocidad, con algoritmos rápidos como YOLO, SSD y Faster RCNN. (Ren et al. 2015)

Se diferencia de los enfoques clásicos, los cuales se basan en estos fundamentos:

* Los datos se pueden modelar con un set de funciones de parámetros lineales
* En la mayoría de los problemas de la vida real, los datos siguen un comportamiento de distribución de probabilidad normal
* Debido al punto anterior, la estimación de parámetros se hace mediante el método de estimación de máxima verosimilitud, que normalmente se reduce a minimizar la suma de errores al cuadrado como función de costo.

Suposiciones que pueden ser inapropiadas. Por ejemplo, en problemas de altas dimensiones donde el mapeo a una forma lineal no es adecuado (maldición de la dimensionalidad). O cuando no sigue una distribución normal, por lo que se requiere construir de otro algoritmo más efectivo. (Kecman 2005)

Al final el enfoque de redes neuronales convolucionales recientemente ha alcanzado una eficiencia casi en tiempo real. En parte gracias a la paralelización en la GPU. Algunas muestras, por ejemplo, en Fast R-CNN que ha alcanzado una frecuencia de 0.5 Hz, Faster RCNN con región proposal alcanza 17 Hz, en cuanto a YOLO9000 alcanza 90 Hz en imágenes de 288 x 288 píxeles y 40 Hz en imágenes de 544 x 544 (Janai et al. 2021). Lograr altas frecuencias en la detección de imagen es importante, pues en un auto en movimiento, la distancia que recorre un auto entre la detección en dos imágenes de la cámara puede ser bastante amplia, y representar serios problemas de seguridad.

Por lo que, los métodos tradicionales, aunque puedan alcanzar niveles de precisión similares a redes neuronales, como SVM. Su uso va a ser limitado únicamente a casos con poco poder de cómputo, en aplicaciones cercanas a tiempo real. En general se puede decir que, en condiciones iguales, los métodos usados son más rápidos que las redes neuronales, pero menos precisos, si trabajamos con datasets pequeños o moderados. Comparando SVM contra redes neuronales, las últimas pueden ser más rápidas al incrementar el tamaño del dataset y agregar aceleración por hardware.

## Rastreo de objetos

Dentro del problema de rastreo, una vez localizado el objeto en la imagen, lo que equivale a decir que tenemos su posición estimada. El siguiente paso es predecir su movimiento y verificarlo con el siguiente frame, esto es asociar dos imágenes en el tiempo. Como dice en el libro de Joel Janai, se diferencia de la detección de objetos, donde los frames se procesan de forma independiente. Esto permite, por ejemplo, detectar posibles colisiones al poder mantener una distancia de frenado y adaptar la velocidad, o reconocer si otro vehículo intenta cambiar de carril.

Nuevamente aquí se presentan múltiples problemas y dificultades, que van desde errores en la detección y oclusión (un vehículo se puede ocultar detrás de otro). O en el caso de múltiples detecciones para poder asociar que una detección corresponde al mismo objeto entre dos frames cuando se trata de la misma clase. (Janai et al. 2021)

### Técnicas

Entre las técnicas disponibles, el enfoque principal ha sido tomarlo como problemas de inferencia bayesiana. En él se estima una función de densidad de probabilidad del estado siguiente usando el estado actual y observaciones pasadas. Posteriormente se usa un paso de corrección o actualización, el cual, mediante retropropagación ajusta los pesos. (Janai et al. 2021)

Algunos de estos métodos son el filtro de Kalman y filtro de partículas. Los cuales toman un acercamiento de espacio estado, donde se estima el estado oculto no medible de un sistema dinámico lineal. En ellos se crean sistemas clon similares al original, pero medibles. Para esto se diseña un observador y cuantas más muestras se tengan, el sistema clon convergerá al original. (Janai et al. 2021)

Para el caso del filtro de Kalman, nuestro sistema dinámico es expresado en espacio estado mediante matrices A, B, C, D, que en tiempo continuo se ve de la forma:

Donde w y v son ruido.

O para el caso discreto:

La formulación del problema es similar al observador de Luenberger, pero considera además la varianza del ruido para encontrar los valores k del controlador. En el caso de nuestro problema, el objetivo es predecir el movimiento de los coches sobre una imagen 2D, considerando la posición xy y las velocidades. Con este problema, el filtro de Kalman debería ser suficiente. Pero en casos de sistemas dinámicos no lineales se puede usar el filtro de Kalman extendido, que realiza un paso extra para linealizar el problema.

El método alternativo es el filtro de partículas, donde el rastreo se hace mediante un método secuencial de Monte Carlo. Mediante la simulación se generan modelos, a los que, al introducir una perturbación, calculan la probabilidad del siguiente estado. Mientras el filtro de Kalman parte de asumir que un problema es lineal y normal (gaussiano), el filtro extendido sirve para problemas no lineales. El filtro de partículas se aplica en problemas no lineales y no gaussianos. En lugar de derivar las ecuaciones analíticas, se usan simulaciones para generar el estado estimado, pero por esta razón es menos eficiente. (Fernández Villaverde, s. f.)

Estos esquemas, también reciben el nombre de rastreo a partir de la detección. En ellos el problema se presenta en la parte de la detección, cuando es incorrecta o tiene datos faltantes. Otros métodos incluyen el uso de redes neuronales. Como se implementa en el proyecto, también mediante la predicción se puede facilitar la detección de objetos. Al tomar regiones de interés según la posición del objeto estimado. Aun así, cada determinado tiempo se debe realizar la detección de un frame completo, no solo para detectar nuevos objetos, también porque se van acumulando errores.

### Filtro de Kalman

Como se ha mencionado el filtro de Kalman parte de un modelo dinámico, que para este problema se deduce de las ecuaciones de movimiento:

Donde x es la posición del objeto, x0 la posición inicial, v0 la velocidad inicial, a la aceleración y el intervalo de tiempo. Esta ecuación se repite para el caso de 3 dimensiones, lo que nos da un sistema de ecuaciones para x, y, z.

Los pasos se observan en la figura 3-6:

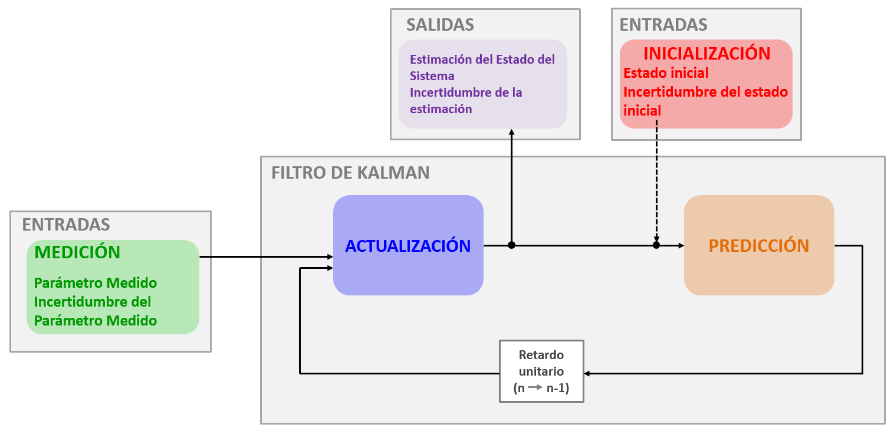


Figura ‑ Pasos del filtro de Kalman, hay una inicialización a partir de la que se realiza la predicción, en el siguiente paso, sí se tienen las medidas se realiza la actualización, además, se puede agregar un paso de control. Fuente: (Becker s. F.)

Tomando el caso en una dimensión (para 3 dimensiones se repiten las mismas ecuaciones para cada eje de coordenadas), en el caso con aceleración constante las ecuaciones son:

Caso con velocidad constante

(Becker s. f.)

El filtro de Kalman parte de 5 ecuaciones:

1. Ecuaciones dinámicas, que son la posición, velocidad y aceleración
2. Ecuaciones de actualización de estado

Donde Kn es la ganancia de Kalman.

1. Ecuación de ganancia del filtro
2. La actualización de incertidumbre estimada
3. Extrapolación de la incertidumbre estimada para sistemas estáticos

Además, se incluye para la inicialización el estado y la incertidumbre inicial del sistema. (Becker s. f.)

### Caso multivariable

Tenemos un vector de estado con las variables medidas, en el caso de 3 dimensiones con posición y velocidad se tienen 6 variables:

El cual en notación de espacio estado, el valor del estado siguiente se obtiene como:

X n+1 es el valor siguiente, x n es el valor estimado en el tiempo n, u la variable de entrada, y wn es el ruido. Nuestras ecuaciones de movimiento vienen representadas dentro de la matriz de transición F, mientras que G es la matriz de control. Como en nuestro caso solo tenemos la predicción y no hay entradas de control, no se usa la matriz G ni u. Entonces retomando las ecuaciones de movimiento para el caso de 3 dimensiones, midiendo la posición y velocidad:

Se traducen en la matriz de transición F como:

Si se incluye la aceleración queda como:

(Becker s. f.)

En nuestra implementación, a partir de las detecciones se tiene un rectángulo. Este incluye la posición (x,y) y tamaño del objeto detectado (ancho y altura). Con OpenCV se implementa el filtro de Kalman para dar seguimiento al movimiento de los vehículos en una imagen 2D. Del caso multivariable mostrado, z no se incluye, en cambio se estima también el tamaño del objeto al agregar dos dimensiones más para predecir el ancho y altura. Esto se trata de un problema bien estudiado, que se puede resolver de forma rápida y eficiente. El uso de otros métodos, como Kalman extendido o filtro de partículas es innecesario, al agregarle mayor complejidad al problema, se implicaría un mayor costo en tiempo.

Como ventaja, el filtro de Kalman es resistente a la falta de información. Incluso nos ayuda a compensar los falsos negativos del paso de la detección, al permitir darle seguimiento al vehículo ante muestras faltantes o darnos regiones sobre los que buscar.

Si bien se trata de un sistema dinámico bien conocido. Se le hacen varias simplificaciones, no se puede conseguir mucha información solo de la imagen y además acarrea errores por la imprecisión de los detectores. Sería interesante comparar contra una solución enteramente basada en redes neuronales.

La implementación del nodo entonces debe recibir la imagen de la cámara en formato sensor\_msgs/Image, es una imagen de 640x480 pixeles en RGB. A continuación, según el número de frame, se realiza el preprocesamiento y la detección de la imagen completa o solo de regiones. Las detecciones se marcan en la imagen original y se debe publicar. Posteriormente, con la ubicación de los objetos se crea un filtro de Kalman para darle seguimiento, o se realiza la corrección de uno ya existente. Del filtro de Kalman, se publica la imagen con la predicción, y para su uso en otros nodos, un mensaje personalizado. Este mensaje incluye en un arreglo todas las detecciones, y para cada una de ellas, su predicción en los 10 siguientes pasos.

# Codificación

En este capítulo se presentan con detalle los pasos para implementar los métodos de detección y rastreo en el carro. Primero se comenta el entrenamiento de los modelos, luego la programación de los nodos, y al final del capítulo se discuten los resultados alcanzados.

### Entrenamiento del filtro de cascada

El primer paso es el entrenamiento de los detectores, para ello se requiere tener *datasets* de imágenes que correspondan a vehículos y otras que no. Primero se comenzó por preparar un conjunto de imágenes que fueran representativas para la clase, un pequeño fragmento se muestra en la figura 4-1. Tomando muestras de las bases de KITTI (Andreas Geiger et al. 2013) y GTI (Arróspide et al. 2012), se partieron en dos sets de aproximadamente 7000 vehículos, y 8000 de no vehículos. Estos último correspondían a imágenes como pedazos de la carretera, sombras, árboles y señalizaciones como se ve en la figura 4-2.

Imagen de la pantalla de un celular con texto e imágenes

Descripción generada automáticamente con confianza media

Figura ‑ Ejemplos de vehículos

Captura de pantalla de un celular con texto e imágenes

Descripción generada automáticamente

Figura ‑ Ejemplos de imágenes negativas para el entrenamiento

Entrenar un clasificador de cascada usando OpenCV es bastante directo. Siguiendo ("OpenCV: Cascade Classifier Training" s. f.) se debe crear un archivo .dat que sirve como descripción de las imágenes positivas. En él se incluyen el nombre de la imagen, número de objetos presentes, coordenadas, y dimensiones del objeto.

1070.png 1 0 0 64 64

Mientras que para las imágenes negativas se crea a un archivo .txt que contiene únicamente su nombre.

El módulo traincascade se trata de un ejecutable incluido solo hasta la versión de OpenCV3.4. En ella, por medio de la consola se ejecutan los siguientes comandos:

.\opencv\_createsamples.exe -info cars.info -num 7000 -w 50 -h 50 –

vec cars.vec

Crea un archivo para las imágenes positivas .*vec*, en el comando se define el tamaño de los sets y la dimensión de los objetos.

.\opencv\_traincascade -data data3 -vec cars.vec -bg nocar.txt –

numPos 4000 -numNeg -6000 -numberStages 20 -w 50 -h 50 –

acceptanceRatioBreakValue 10e-05 -minHitRate 0.995 –

maxFalseAlarmRate 0.25 -featureType LBP

- data corresponde al folder destino

- vec archivo de imágenes positivas

- bg imágenes negativas

- numPos y numNeg tamaño del batch de imágenes que se usan en cada etapa

- numberStage número máximo de etapas

- w y h tamaño de los objetos

- acceptanceRatioBreakValue: Indica cuándo se debe dejar de entrenar el modelo, se usó el ratio sugerido de 10e-05 para evitar sobreajuste de los datos

- maxFalseAlarmRate: Máxima tasa de errores aceptada, normalmente se mantiene alta con valores de hasta 0.5, se tomó 0.25%.

- minHitRate: tasa mínima de aciertos, se eligió 0.995%

- featureType: HAAR o LBP, se usó LBP para entrenar la máquina de forma más rápida. Con HAAR a costa de un mayor tiempo de entrenamiento se podrían obtener ligeramente mejores resultados.

Al final regresa el modelo como un archivo .xml.

### Entrenamiento de svm

En el filtro de cascada se decidió solo usar un set de autos, con el fin de únicamente identificar posibles candidatos para hacer la predicción. Esto es gracias a la velocidad de cascada comparado con svm, al hacer la detección con ventana deslizante. De forma, que sobre las detecciones de cascada solo se realiza una predicción, que separa las imágenes en dos clases -1 o 1, si son o no autos.

Para descartar los falsos positivos, se usan máquinas de soporte vectorial. Aquí con el fin de lograr una mejor precisión, reducir el tiempo de entrenamiento y el número de muestras requeridas, se parte el problema en dos máquinas. Una para identificar coches desde la parte trasera y otro lateral, figuras 4-3 y 4-4 respectivamente. Es posible y recomendable separar en más clases el problema.

Una captura de pantalla de un celular con texto e imágenes

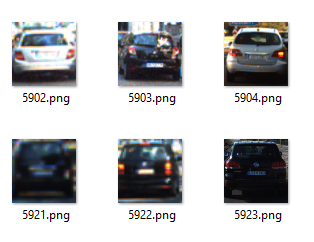
Descripción generada automáticamente con confianza baja

Figura ‑ Imágenes de vehículos de lado

Figura ‑ Imágenes de la parte trasera de vehículos

Para el entrenamiento, dentro de un programa en C++, se deben cargar las imágenes, extraer las características con HOG de cada una y preparar un vector con sus etiquetas. Es importante mencionar que entre la versión de OpenCV 2.4 (la del vehículo) y la más nueva OpenCV 4 hay bastantes diferencias, y no es posible usar el mismo modelo.

Primero se usa un método *load\_images* (figura 4-5), en el que se realiza un pequeño preprocesamiento para las imágenes. Se pasan a blanco y negro, y se les aplica un filtro gaussiano, con el fin de remover el ruido y detalles insignificantes, lo que ayuda a mejorar la precisión. Otros tipos de filtros que se pueden usar son por ejemplo el *motion* *blur*, para un efecto de velocidad, o un filtro de *sobel* para resaltar solo los bordes.

Texto

Descripción generada automáticamente

Figura ‑ Función load\_images

A continuación, con la función computeHOGs, se extraen las características de las imágenes tanto positivas como negativas.

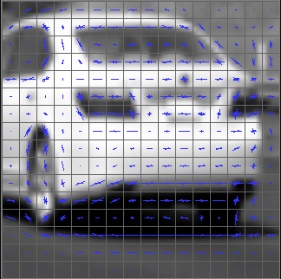
Las características que se definen son:

* winSize
* blockSize
* cellSize
* blockStride
* nbins
* derivAperture
* winSigma

De ellas, las más importantes son:

* BlockSize define el tamaño de las imágenes que se le proporcionan, normalmente se suelen usar de 64x64.
* Cellsize es el tamaño de la celda o sobre cuantos pixeles se calcula el histograma. Mientras que blockstride el paso para deslizar un bloque sobre la imagen. Cell y blockStride deben ser potencias de 2, y el residuo de blocksize entre cellsize debe ser 0.
* Nbins es el número de orientaciones o en cuántos intervalos se puede partir una celda, a mayor número mejor precisión.

Usando la implementación de HOGImage (Zhou, 2019) nos permite ver las características extraídas por HOG. Varios ejemplos se observan en las figuras de la 4-6 a 4-11, variando parámetros de HOG, como incluyendo filtros.

Diagrama

Descripción generada automáticamente

Figura ‑ Con blocksize (64,64), cellsize y blockstride de (4,4).

Figura ‑ Con blocksize (64,64), cellsize y blockstride (8,8), con solo 8x8 celdas no es suficiente para extraer las características de la imagen, por lo que el resultado es muy pobre.

Imagen que contiene interior, con baldosas, pequeño, tabla

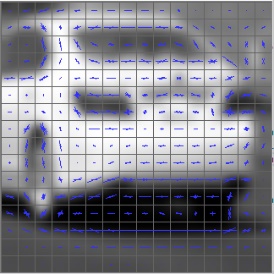
Descripción generada automáticamente

Figura ‑ Imagen del vehículo del simulador

Figura ‑ Con filtro gaussiano (5,5)

Ejemplo de imágenes negativas

Imagen que contiene Patrón de fondo

Descripción generada automáticamenteImagen que contiene interior, ventana, cama, grupo

Descripción generada automáticamente

Figura ‑ Hog features de la carretera

Figura ‑ Hog features de un árbol

Aumentar el número de características tiene un impacto muy alto en el tamaño de la máquina de soporte vectorial, junto al tiempo de procesamiento. Por ello se consideró suficiente con un tamaño de celdas y de paso de 4 pixeles, lo que equivale a dividir la imagen en 256 bloques.

Texto

Descripción generada automáticamente

Figura ‑ Función computehogs, extrae los descriptores de todas las imágenes

Para extraer HOG se usa la función computeHOGs de la figura 4-12, la instrucción principal es:

hog.compute(img\_lst[i], descriptors, Size(0, 0), Size(0, 0));

Una vez extraídas todas las características y guardadas, se crea otro vector labels donde se guardan las etiquetas. +1 si son positivas, -1 si son negativas.

size\_t positive\_countC = gradient\_lstC.size();

labelsC.assign(positive\_countC, +1)

Con el cambio de la versión a OpenCV 4, Se requiere convertir los datos extraídos por HOG a una matriz Mat de nx1, para eso se utiliza la función de la figura 4-13:

convert\_to\_ml(gradient\_lstC, train\_dataC);

Texto

Descripción generada automáticamente

Figura ‑ Código de la función convert\_to\_ml, mismo propuesto en los ejemplos de opencv

Además, con OpenCV2.4 se agrega la siguiente línea para las etiquetas:

Mat labels2 = Mat(1, labelsC.size(), CV\_32SC1,

labelsC.data()).clone();

Una vez extraídas las muestras, lo siguiente es pasar a entrenar la máquina de soporte vectorial, figura 4-14:

Texto

Descripción generada automáticamente

Figura ‑ Entrenamiento de svm con trainauto

Para crear la máquina se utiliza:

Ptr< SVM > svmC = SVM::create();

En OpenCV2.4

CvSVM \*svm = new CvSVM;

Hay dos formas de entrenar la máquina. Con train dándole los parámetros o trainAuto para buscar los mejores parámetros a partir de una malla dada. Como trainAuto es lento, es mejor reducir la búsqueda limitando el rango y definiendo qué parámetros debe optimizar.

svmC->setType(SVM::C\_SVC);

svmC->setKernel(SVM::RBF);

En primer lugar, se define el kernel y tipo, de los cuales se lograron mejores resultados con RBF y C\_SVC. A continuación, se definen los grids o valores sobre los que buscar, OpenCV ya tiene default grid. Para el tipo RBF + C\_SVC únicamente nos interesan optimizar los parámetros de gamma y c, para los que se usaron los valores por defecto. Para que no optimice los demás parámetros se utiliza:

ParamGrid CvParamGrid\_coeff(pow(2.0, -2), pow(2.0, 5), 0);

Se crea una malla en la que se da el primer valor, el valor máximo y el incremento, al darle 0 de incremento no lo optimizará.

Se define también el termCriteria que indica cuando parar.

svmC->setTermCriteria(TermCriteria(TermCriteria::MAX\_ITER,

(int)1e8, 1e-6));

Opcionalmente se pueden poner class weights, los cuales son auxiliares a C para penalizar el error como se observa en la figura 4-15.

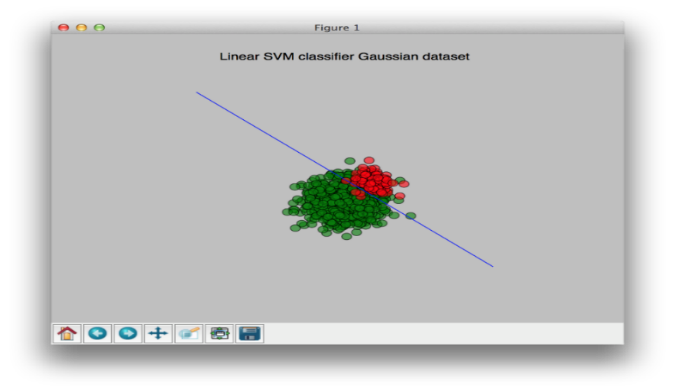
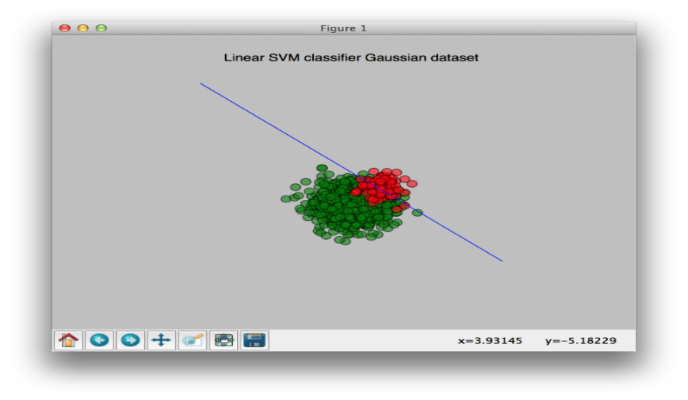


Figura ‑ Se observa el impacto de establecer pesos para el valor de c, a la izquierda la clasificación normal, a la derecha con un peso de 0.9 sobre la clase de verde, fuente: (“svm bias on weights of positives and negatives - opencv q&a forum» s. F.)

cv::Mat1f weights(1, 2);

weights(0, 0) = 0.2; //para la clase de -1

weights(0, 1) = 0.8; //para la clase de 1

En el ejemplo, se puede interpretar como que, para la solución, en la ubicación del corte tenga a lo más un 20% falsos positivos y un 80% verdaderos positivos.

Entonces la función para entrenar la máquina queda como:

svmL->trainAuto(ml::TrainData::create(train\_dataL,

ml::ROW\_SAMPLE, labelsL), 10, svmL->

getDefaultGrid(SVM::C), svmL->

getDefaultGrid(SVM::GAMMA), CvParamGrid\_p,

CvParamGrid\_nu, CvParamGrid\_coeff, CvParamGrid\_deg, false);

En primer lugar, los parámetros se optimizaron con trainAuto, fue posible reutilizarlos cuando se incrementó el tamaño de los *datasets* y para el entrenamiento en OpenCV 2.4 (figura 4-16). Sin embargo, puede que haya otros óptimos.

svmL->train(ml::TrainData::create(train\_dataL,

ml::ROW\_SAMPLE, labelsL));

Texto

Descripción generada automáticamente

Figura ‑ Imagen de la parte de entrenamiento en opencv 2.4 usando train

Para los coches de lado, los valores obtenidos fueron de 2.25x10^-3 para gamma y 12.5 para C. Para los coches desde atrás, tuvo la misma gamma, pero 62.5 en C.

En el caso de OpenCV2.4 la máquina se guarda en formato .xml, mientras que OpenCV4 en .yml. Se mencionan ambas, porque inicialmente se realizó sobre la versión 4 y es la que se usó para el simulador, posteriormente se tradujo a la versión 2.4 para el vehículo.

### Detección y nodo para el vehículo

El primer paso en ambos casos es cargar los descriptores desde la función de main:

SVM:

Ptr<SVM> svm;

svm = StatModel::load<SVM>(svmFile);

Cascada:

CascadeClassifier carC;

carC.load(obj\_det\_filename);

En 2.4, para la máquina de soporte vectorial se cambia a:

CvSVM \*svm = new CvSVM;

svm->load(svmFile.c\_str());

Después se preparan vectores que contendrán todos los objetos detectados siempre que no rebasen un determinado número de frames sin aparecer. En el caso del vehículo se usaron 15 frames, que, usando una frecuencia de 5 Hz, corresponde a 3 segundos. Se trata de un vector de rectángulos que guardan la posición, el último frame en que apareció y de filtros de Kalman. Con ello nos permite seguir múltiples objetos sin que tengan que aparecer en el frame actual.

Una vez todo inicializado, dentro de un ciclo se recibe la imagen, para las pruebas se leen imágenes extraídas de un video. Para el vehículo es necesario un paso extra para extraer la imagen de la cámara:

Al inicio del main, figura 4-17 se define:

ros::NodeHandle nh("~");

ros::Subscriber camara\_sub =

nh.subscribe("/app/camera/rgb/image\_raw", 10,

camaraRGBCallback);

Se suscribe al tópico de la cámara que nos manda un mensaje tipo sensor\_msgs::Image

Esta función de callback se entra cada que se complete un ciclo (while(nh.ok())), lee el tópico que se publica y lo guarda en una variable local. La imagen debe convertirse al formato Mat con ayuda de cv\_bridge:

cv\_bridge::CvImagePtr cv\_ptr;

cv\_ptr = cv\_bridge::toCvCopy(msg,

sensor\_msgs::image\_encodings::MONO8);

img0 = cv\_ptr->image.clone();

Con MONO8 ya recibimos la imagen en escala de grises

La imagen es de 640x480, pero podemos simplificar el problema tomando una ROI. Para este caso se recortó la altura, tomando ahora una imagen de 640x400, que corresponde a quitar una porción de la parte delantera del auto que se alcanza a ver en la cámara.

Igualmente se define una frecuencia para el ciclo con:

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteros::Rate loop\_rate(RATE\_HZ);

Figura ‑ Imagen de la función main, la configuración inicial, hasta el inicio del ciclo

Dentro del ciclo se hace primero la detección y después el rastreo. En el caso de la detección puede hacerse tomando el frame completo o solo en regiones cercanas a las detecciones. Es por ello tenemos dos funciones:

* Detección completa

detecVentana(svm, svm2,carC, img, i, &img2,

&numberD, &detections);

Función que cada 5 frames realiza la detección completa del frame. Dentro, como no se usa la detección multiescala de HOG, si no que solo la predicción svm, se deben cargar los mismos parámetros para extraer las características con que fue entrenada originalmente:

HOGDescriptor hog;

hog.winSize = Size(64, 64);

hog.blockSize = Size(16, 16);

hog.cellSize = Size(4, 4);

hog.blockStride = Size(4, 4);

hog.nbins = 9;

hog.derivAperture = 1;

hog.winSigma = 4;

//hog.histogramNormType = 0;

hog.L2HysThreshold = 2.0000000000000001e-01;

hog.gammaCorrection = 1;

hog.nlevels = 64;

El primer paso es usar el filtro de cascada para darnos una lista de candidatos en los que buscar. Usa el método de ventana deslizante y es bastante rápido, pero con muchos falsos positivos:

carC.detectMultiScale(imgGRAY, detectionsCascada, 1.1, 2, 0,

Size(40, 40), Size(200, 200));

detections2 regresa una lista de rectángulos con las detecciones,

* 1.1 es el factor por el que se va incrementando la ventana
* 2 son el número de coincidencias que deben aparecer
* Size(40,40) tamaño inicial de la ventana
* Size(200,200) tamaño máximo

Posteriormente se recorre cada candidato, tratando de tomar una pequeña área más grande del 20%.

int width = detectionsCascada[j].width \* 0.2;

int height = detectionsCascada[j].height \* 0.2;

Se cambia de tamaño a 64x64 (mismo de HOG), se aplica un filtro gaussiano y se extraen sus características:

resize(imgEsquina, imgSVM, Size(64, 64), 0, 0, cv::INTER\_AREA);

vector< float > descriptorsSVM;

imgSVM.convertTo(imgSVMF, CV\_8UC3);

hog.compute(imgSVMF, descriptorsSVM, Size(0, 0), Size(0, 0));

En compute el primer size(0,0) significan que se toman la imagen completa para hacer la predicción, el segundo corresponde al padding.

Con OpenCV2.4 se agrega para usar los descriptores:

Mat fm = Mat(1, descriptorsSVM.size(), CV\_32FC1,

descriptorsSVM.data()).clone();

Finalmente, para hacer la predicción:

float result1 = svm->predict(descriptorsSVM);

En OpenCV2.4:

float result1 = svm->CvSVM::predict(fm);

El problema se puede visualizar en la figura 4-18, donde a la hora de detectar autos que van apareciendo en las esquinas, ninguno de los dos detectores lo ven hasta que el auto no esté completo.

Imagen en blanco y negro de un carro

Descripción generada automáticamente Imagen en blanco y negro de un carro

Descripción generada automáticamente Imagen en blanco y negro de un carro

Descripción generada automáticamente

Figura ‑ Ejemplos de falsos negativos

Hecha la predicción, si el resultado es positivo (1), se guarda dos veces para poder usar groupRectangles. La función toma el promedio de los rectángulos sobrepuestos, pero solo los conserva si hay más de uno.

if (result > 0.0) {

detectionsFinal.push\_back(detectionsCascada[j]);

detectionsFinal.push\_back(detectionsCascada[j]);

}

groupRectangles(detectionsFinal, 1, 0.6);

Los rectángulos detectados se dibujan sobre la imagen y esta se publica agregando al inicio:

ros::NodeHandle nh("~");

detec\_publisher =

nh.advertise<sensor\_msgs::Image>("/detec",1);

El mensaje es bajo el tópico /detec.

Al final de la detección:

sensor\_msgs::Image img\_msg;

std\_msgs::Header header;

header.stamp = ros::Time::now();

img\_bridge = cv\_bridge::CvImage(header,

sensor\_msgs::image\_encodings::TYPE\_8UC1, img1);

img\_bridge.toImageMsg(img\_msg);

detec\_publisher.publish(img\_msg);

Se crea un nuevo mensaje imagen, cuyo formato incluye un encabezado, al que se le pone la hora. Con CV\_bridge se convierte la imagen, y se publica. Las detecciones se pueden visualizar en RVIZ.

* Detección sobre solo una región:

En este caso se llama a la función de la figura 4-19:

detecRegion(carC, img, i, predRect.x, predRect.y,

predRect.width, predRect.height, &img2, &encontro,

&detec);

Solo entra cuando hay detecciones previas, busca en los lugares indicados por la predicción de Kalman, que se realiza desde el main:

Mat state(6, 1, CV\_32F);

state = kalmans[j].predict();

Rect predRect;

predRect.width = state.at<float>(4);

predRect.height = state.at<float>(5);

predRect.x = state.at<float>(0);

predRect.y = state.at<float>(1);

En caso de no encontrarlo, busca en la posición anterior, en ambos casos se toma una ventana un 80% más grande. De forma que nuestro acercamiento es asumir que el mismo objeto debe estar cerca de donde fue detectado la primera vez. En este método únicamente se realiza la detección por cascada, si hay coincidencias se usa group rectangle para dar el promedio de la nueva posición.

carC.detectMultiScale(img2, detections2, 1.1, 2, 0,

Size(width\*0.4,height\*0.4), Size(width1,height1));

Texto

Descripción generada automáticamente

Figura ‑ Fragmento del código para detectar sobre una región

### Rastreo

El siguiente paso es el rastreo, para cada detección se revisa si es el mismo vehículo entre dos frames, basándose en que se halle en una posición cercana a la anterior. Si se cumple, se hace la actualización, en caso contrario se debe crear un nuevo filtro de Kalman. El código completo se ve en la figura 4-25.

Para crear el filtro se cuenta con la función:

void preparacionKalman(Rect detections, KalmanFilter\* kalman)

Función que recibe un rectángulo con la detección y regresa el filtro.

Para el rastreo se consideran 6 variables la posición y velocidad en x y y, el tamaño de la ventana detectada width y height. De las 6 variables, 4 son las que se miden, la posición y la ventana

int stateSize = 6; //[x, y, v\_x, v\_y, w, h]

int measSize = 4; //[z\_x,z\_y,z\_w,z\_h]

KalmanFilter kf(stateSize, measSize, contrSize);

//Matriz estados

Mat state(stateSize, 1, CV\_32F);

//Matriz mediciones

Mat meas(measSize, 1, CV\_32F);

Después de crear el filtro, se agrega la primera medición del objeto en los estados statePre y statePost.

kf.statePre.setTo(0);

kf.statePre.at<float>(0, 0) = detections.x;

kf.statePre.at<float>(1, 0) = detections.y;

kf.statePre.at<float>(2, 0) = 0;

kf.statePre.at<float>(3, 0) = 0;

kf.statePre.at<float>(4, 0) = detections.width;

kf.statePre.at<float>(5, 0) = detections.height;

kf.statePost.setTo(0);

kf.statePost.at<float>(0, 0) = detections.x;

kf.statePost.at<float>(1, 0) = detections.y;

kf.statePost.at<float>(2, 0) = 0;

kf.statePost.at<float>(3, 0) = 0;

kf.statePost.at<float>(4, 0) = detections.width;

kf.statePost.at<float>(5, 0) = detections.height;

La matriz de transición entonces queda de la siguiente forma:

1 0 1 0 0 0

0 1 0 1 0 0

0 0 1 0 0 0

0 0 0 1 0 0

0 0 0 0 1 0

0 0 0 0 0 1

Las ecuaciones son las siguientes:

x = x0 + vx0\*Δt

y = y0 + vy0\*Δt

La posición en x depende de la posición y velocidad en x en el paso anterior. No se está tomando en cuenta la aceleración, y se considera x, y independientes. Mientras que las ventanas solo dependen de su valor anterior, podría agregarse que dependieran de la posición.

kf.transitionMatrix = (Mat\_<float>(6, 6) << 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1);

La matriz de medición queda como:

1 0 0 0 0 0

0 1 0 0 0 0

0 0 0 0 1 0

0 0 0 0 0 1

kf.measurementMatrix = cv::Mat::zeros(measSize, stateSize,

CV\_32F);

kf.measurementMatrix.at<float>(0) = 1.0f;

kf.measurementMatrix.at<float>(7) = 1.0f;

kf.measurementMatrix.at<float>(16) = 1.0f;

kf.measurementMatrix.at<float>(23) = 1.0f;

Por último, se define la covarianza y el ruido.

setIdentity(kf.processNoiseCov, Scalar::all(.00005));

Para processNoiseCov se encontró adecuado un valor de 5\*10e^-5. Con un valor más grande se tiene una respuesta más rápida del filtro, pero es más propenso a errores cuando no hay detecciones.

setIdentity(kf.measurementNoiseCov, Scalar(1e-1));

setIdentity(kf.errorCovPost, Scalar::all(.1));

Imagen en blanco y negro

Descripción generada automáticamente con confianza bajaImagen en blanco y negro

Descripción generada automáticamente con confianza baja Podemos comparar como afecta modificar las covarianzas. En el caso de measurementNoiseCov se puede interpretar en qué tanto se confía en las mediciones y es un error que se propaga hacia atrás. Representa el ruido que puede haber en las mediciones o para nuestro caso el detector. En el ejemplo mostrado entre las figuras 4-20 y 4-22, los puntos azules deben seguir al verde y hacia el final no se obtienen nuevas mediciones, por lo que ya no se realiza la corrección.

Figura ‑ measurementnoisecov de 0.01

Figura ‑ mesasurementnoisecov de 0.1

Como se ve con mayor ruido “measurementNoiseCov” sigue mucho peor la señal cuando deja de tener nuevas mediciones. Además, requiere de más valores para un mejor seguimiento. En el ejemplo deja de moverse en el eje x, pero con ruido alto la predicción preserva todavía movimiento en ese eje.

Imagen en blanco y negro

Descripción generada automáticamente con confianza baja

Figura ‑ measurementnoisecov de 0.001

Imagen en blanco y negro

Descripción generada automáticamente con confianza bajaImagen en blanco y negro

Descripción generada automáticamente con confianza baja En cambio, como se ve en las figuras 4-23 y 4-24, el efecto de processNoiseCov se observa en la corrección. Con un valor más grande permite una convergencia más rápida a la señal, pero acarrea más ruido que se hace presente cuando no se tienen nuevas mediciones.

Figura ‑ Con processnoisecov de 0.00005, los puntos azules reaccionan más lento a los cambios

Figura ‑ Con processnoisecov de 0.5, los puntos azules reaccionan más rápido y como consecuencia se disparan

Texto

Descripción generada automáticamente

Imagen de la pantalla de un computador

Descripción generada automáticamente con confianza baja

Figura ‑ Código completo para crear el filtro de Kalman

**Actualización del filtro**

Para la actualización, figura 4-26, se tiene la siguiente función que recibe el filtro y lo devuelve actualizado. Recibe la imagen, si hubo una detección en el frame actual y dicha posición, así mismo como los valores del reloj para obtener el intervalo de tiempo entre las dos predicciones.

kalman(KalmanFilter kf, Rect detections, Mat img,int i,bool

found, std::chrono::monotonic\_clock::time\_point lastP,

std::chrono::monotonic\_clock::time\_point Pactual,

KalmanFilter\* act)

Igual que para la detección, se publicará la imagen con el rectángulo negro de la predicción y un mensaje con las detecciones.

ros::NodeHandle nh("~");

detec\_publisherk =

nh.advertise<sensor\_msgs::Image>("/deteck",1);

Al final de la predicción:

sensor\_msgs::Image img\_msg;

std\_msgs::Header header;

header.stamp = ros::Time::now();

img\_bridge = cv\_bridge::CvImage(header,

sensor\_msgs::image\_encodings::TYPE\_8UC1, img);

img\_bridge.toImageMsg(img\_msg);

detec\_publisherk.publish(img\_msg);

Dentro se realiza primero la predicción para actualizar al siguiente estado y dibujar sobre la imagen. Además, se actualiza la matriz de transición modificando el valor de dt que corresponde el intervalo de tiempo transcurrido entre dos frames.

kf.transitionMatrix = (Mat\_<float>(6, 6) << 1, 0, dt, 0, 0, 0, 0, 1, 0, dt, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1);

Mat meas(4, 1, CV\_32F);

Mat state(6, 1, CV\_32F);

state = kf.predict();

Rect predRect;

predRect.width = state.at<float>(4);

predRect.height = state.at<float>(5);

predRect.x = state.at<float>(0);

predRect.y = state.at<float>(1);

Si se encontró hace la corrección:

if (found) {

meas.at<float>(0) = detections.x;

meas.at<float>(1) = detections.y;

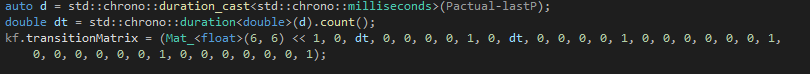
meas.at<float>(2) = detections.width;

meas.at<float>(3) = detections.height;

kf.correct(meas);

}

Como se observa, la operación que realiza es: statePre = TransitionMatrix \* statePost



Texto

Descripción generada automáticamente

Figura ‑ Código para hacer la predicción y corrección

# Resultados

Tras el entrenamiento del filtro de cascada y las máquinas de soporte vectorial, se probaron los detectores con un set de validación de unas 16000 muestras aproximadamente. Los parámetros se buscaron con un set de entrenamiento de cerca de 400 muestras, y al ir variando los parámetros de HOG, se obtuvieron resultados lo suficientemente buenos. Es importante mencionar antes, que para el set de pruebas se combinaban ambos tipos de imágenes tanto coches de lado como de la parte trasera. De modo que el set de validación está desbalanceado, pues se tienen más coches vistos desde atrás.

Usando únicamente la máquina entrenada con la parte trasera de coches se consiguieron detectar correctamente el 77.8% de los coches, con un 0.1% de falsos positivos:

Tabla ‑ Matriz de confusión para svm con imágenes de la parte trasera de coches

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.99855 (TN) | 0.00144976 (FP) |
| 1 | 0.241294 (FN) | 0.758706 (TP) |

Precisión: TP/(TP+FP) = 99.81%

Exactitud: (TP+TN)/(TP+FN+TN+FP) = 87.86%

Por otro lado, la máquina entrenada con coches de lado alcanzó a detectar correctamente solo el 45.1% de coches correctamente, con un 0.8% de falsos positivos:

Tabla ‑ Matriz de confusión de svm con coches de lado

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.991859 | 0.00814096 |
| 1 | 0.548336 | 0.451664 |

Precisión: 98.2%

Exactitud: 72.17%

Por ultimo, con la idea original de juntar ambas máquinas, se alcanzó a detectar correctamente el 87.8% de los coches, con un 0.9% de falsos negativos.

Tabla ‑ Matriz de confusión resultante de juntar ambas svm

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.985632 | 0.01436768 |
| 1 | 0.14102 | 0.85898 |

Precisión: 98.92%

Exactitud: 92.23%

Posteriormente se aumentó el tamaño del set de entrenamiento a 1500 imágenes positivas. Esto mejoró los resultados, claro que a costa de aumentar considerablemente el tiempo de entrenamiento y para cargar los detectores. El tiempo para la detección también aumento, pero en menor medida.

Máquina 1 (vista de la parte trasera de los coches):

Tabla ‑ Matriz de confusión de svm completa con la parte trasera de coches

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.994888 | 0.00511152 |
| 1 | 0.0814 | 0.9186 |

Precisión: 99.44%

Exactitud: 95.67%

Máquina 2 (vista de lado de los coches):

Tabla ‑ Matriz de confusión de svm completa con coches de lado

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.997746 | 0.002254 |
| 1 | 0.36003 | 0.63997 |

Precisión: 99.65%

Exactitud: 81.89%

Ambas máquinas:

Tabla ‑ Matriz de confusión de ambas svm completas

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.99775 | 0.00225 |
| 1 | 0.0146 | 0.9854 |

Precisión: 99.77%

Exactitud: 99.15%

Como se observa, las mejoras entre usar solo la máquina 1 y ambas son ligeras. Aun así, en las pruebas en un escenario real los resultados empeoran, siendo el mayor problema la oclusión ambiental. Debido a ella, cuando se presentan pequeños cambios en la iluminación o sombras, una detección puede no aparecer en el siguiente frame. A esto se le suma que la detección no siempre es igual. El área que detecta como vehículo o el centro puede variar ligeramente entre frames, lo que entorpece principalmente al filtro de Kalman.

En el caso del filtro de cascada, desafortunadamente es la parte más débil de la detección. Entrenada con una muestra de 6000 imágenes positivas y negativas, tiene un desempeño pobre para detectar positivos:

Tabla ‑ Matriz de confusión del filtro de cascada

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.983274 | 0.016726 |
| 1 | 0.1077 | 0.8923 |

Precisión: 98.16%

Exactitud: 93.78%

Combinadas:

Tabla ‑ Matriz de confusión de ambos detectores

|  |  |  |
| --- | --- | --- |
| Realidad/predicción | 0 | 1 |
| 0 | 0.99886 | 0.00114 |
| 1 | 0.1232 | 0.8768 |

Precisión: 99.87%

Exactitud: 93.78%

Para empezar, en la prueba de un caso real con un video. El filtro de cascada arroja por frame entre 7 y 11 detecciones o candidatos para SVM, como se ven en la figura 5-1, pese a tener una tasa muy alta de falsos positivos, la idea es que encuentre la mayoría de los vehículos. En la figura 5-2 se observa el resultado de juntar las máquinas de soporte vectorial para filtrar los candidatos. El mayor problema es la presencia de falsos positivos. Pues en el caso de falsos negativos, se tiene la ventaja de que se sigue realizando el seguimiento entre frames con el filtro de kalman. Siempre y cuando se hayan tenido suficientes muestras para hacer el seguimiento de forma correcta. El siguiente paso, en la figura 5-3 se observa la predicción Kalman.

Figura ‑ Candidatos arrojados por el filtro de cascada

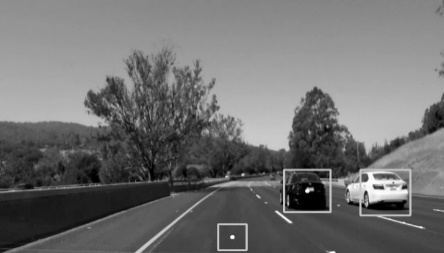


Figura ‑ Ejemplos de detecciones sobre frames, en la primera imagen se identifican correctamente los dos vehículos, en el segundo por el cambio de iluminación no detecta uno de los vehículos, en la tercera imagen se muestra un falso positivo.



Figura ‑ Predicción del filtro de Kalman, los recuadros blancos son las detecciones, y los negros la predicción (en esa misma ventana de la predicción es donde se realiza la detección parcial para buscar el vehículo y hacer la corrección.

La solución presentada no es óptima, ni se puede trasladar a cualquier problema. Según las especificaciones se requiere de optimizar algunos puntos, entre ellos:

* Tamaño mínimo y máximo para las ventanas en detección con cascada. Depende de la cámara, la distancia focal de esta, junto a su posición. Se pueden tener distintos zooms según el ángulo de visión, con los que las imágenes pueden ser más grandes o pequeñas a una misma distancia.
* Tamaño de la región de interés, el problema se realizó tomando en cuenta que la cámara del vehículo es de 640x480 pixeles, y parte del área de la imagen es ocupada por la parte frontal del vehículo.
* El número máximo de frames sin que aparezca una detección para seguir haciendo el seguimiento. Depende en parte de la potencia computacional del vehículo, con la cual se tendrá una taza u otra para hacer el procesamiento de las imágenes. Por lo que entre frames se pueden esperar mayores o menores cambios según la frecuencia.
* Mismo razonamiento, para el máximo movimiento esperado de un vehículo entre dos frames, para poder asociar una detección como correspondiente al filtro de Kalman
* Según la taza, se puede limitar el problema si se considera el entorno sobre el cual el vehículo se mueve, junto a su velocidad, se pueden ajustar mejor los parámetros del filtro, el processNoiseCov, measurementNoiseCov y errorCovPost.
* Igualmente se puede mejorar la precisión de los detectores. Limitando el problema a algún ambiente, o dividiendo la imagen y aplicar según la región distintos clasificadores.

Con el simulador se ejecuta uno de los mapas como curved\_road.launch (figura 6-30). El cual se ha modificado para que aparezcan dos vehículos. Por medio de 3 terminales se corre:

* roscore
* roslaunch autonomous\_gazebo\_simulation curved\_road.launch
* rosrun prueba detecRastreoSim

Para pruebas le damos velocidades a los coches, con los cuales se ajustaron los valores de la covarianza para el filtro de Kalman, algunos resultados se observan en la figura 5-5.

Patrón de fondo

Descripción generada automáticamente

Figura ‑ Imagen del simulador desde gazebo, mapa con dos coches

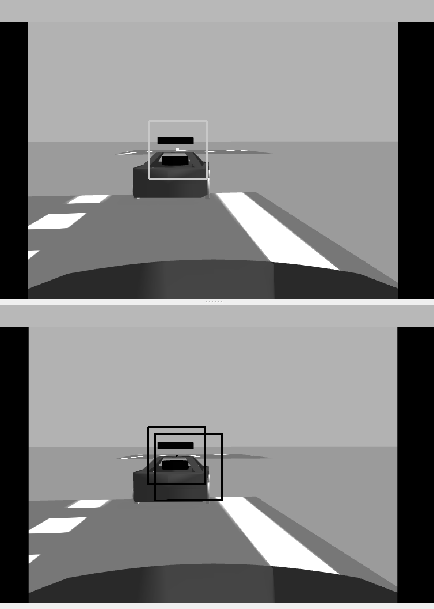
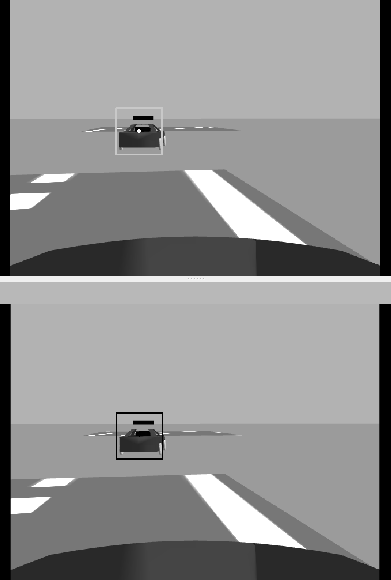


Figura ‑ Ejemplo de detecciones en el simulador, arriba se muestra la detección, abajo la predicción

El procedimiento es similar para ejecutarlo en el vehículo, se ejecutan los comandos:

* roslaunch manual\_control manual\_odroid.launch
* rosrun vision\_camara detecRastreoC

En el caso de los tiempos:

Tabla ‑ Tiempos medidos de los detectores en los tres ambientes de pruebas

|  |  |  |  |
| --- | --- | --- | --- |
|  | En Visual Studio | En el simulador | En el vehículo |
| Tiempo para cargar las dos máquinas de soporte vectorial | 967.38 segundos | 10.156 segundos | 63.829 segundos |
| Tiempo para la detección de un frame completo | 1700-2200 ms | 100-110 ms | 380-390 ms |
| Tiempo para la detección sobre una región | 60-90 ms | 50-70 ms | 60-70 ms |
| Tiempo para la predicción | 30-60 ms | 40-60 ms (predicción y corrección) | 30-40 ms |
| Tiempo total en detección de un frame completo | 1800-2300 ms | 140-180 ms | 430-450 ms |
| Tiempo total en detección solo de regiones en un frame | 150-200 ms | 110-140 ms | 90-100 ms |

Como se observa, el procedimiento en visual studio es lento, en especial a la hora de cargar las máquinas de soporte vectorial, donde demora hasta 17 minutos. Esto se debe a que únicamente aprovecha el 13% del CPU, que en nuestro caso se corrió en un procesador intel i3 6100u. Por otro lado, tanto el simulador como el vehículo pueden llevar a cabo la ejecución sin problemas, siendo el primero hasta 10 veces más rápido. De acuerdo con los tiempos que le toma procesar cada frame, en el simulador se pudo haber tomado una taza ligeramente más alta que 5 Hz. En el carro en cambio, no puede procesar más de 5 frames en un segundo.

En el caso para cargar el filtro de cascada, al ser muy ligero, prácticamente no demora tiempo, apenas milisegundos para el simulador y 103 ms en visual studio. Por un lado, en la detección, hacer el barrido con ventana deslizante con svm sería muy tardado, aun haciéndose con el filtro de cascada, es la parte de la detección que más consume tiempo. Por otro lado, en el rastreo, la mayor parte del tiempo lo consume la predicción y corrección de los filtros, crear uno nuevo es instantáneo. De lo observado en el carro físico, es que demora más en la predicción con las máquinas de soporte vectorial.

# Diseño de la Red vanet

En este capítulo se cubre el tema de redes vanet, primero se comenta cómo se realiza la implementación. Después se analizan las problemáticas que involucra la comunicación entre vehículos y se menciona ligeramente sobre el problema de localización. Además, se repasan patrones de comunicación, entre ellos AODV. Se comentan los modelos v2x, v2v y v2i, que nos permite presentar el tema del protocolo DSRC. Sobre este protocolo, se analiza sus requisitos en cada capa del modelo OSI que requiere. Por último, se comenta los contenidos requeridos en los mensajes.

## Arquitectura

Pensando en aplicaciones de seguridad. Se plantea el uso de DSRC para la comunicación de las detecciones y AODV para la coordinación entre vehículos. A partir de ahí se simula, dado un escenario, el riesgo o probabilidad de choque entre dos vehículos. En el escenario planteado, un coche busca incorporarse sobre una vialidad rápida en donde no tiene línea de visión con otros vehículos y, por lo tanto, por sí solo el nodo detector del coche no es útil. En un entorno colaborativo, los otros coches informan de sus detecciones. Con ellas, los dos vehículos deben quedar notificados de la presencia del otro, y posteriormente coordinarse.

En las dos simulaciones se calcula el riesgo de colisión entre nodos, dependiendo de cuantos mensajes logran recibir. En la primera sobre AODV, el vehículo que quiere incorporarse recorre un trayecto de 100 metros, por lo que antes de llegar (tomando en cuenta una distancia de seguridad según la velocidad que llevan) deben recibir al menos dos mensajes. Con la segunda, ahora el recorrido son 300 metros y se añade que los otros vehículos se están comunicando también, lo que traerá colisiones y retrasos.

Ambas simulaciones se realizan sobre el simulador de omnet++, y para la segunda se recurre además a veins y sumo, pues cuentan con paquetes sobre el protocolo DSRC. Una simulación de omnet++ se compone de tres archivos principales, .ned para agregar los módulos que se usan y describir la topología de la red; el archivo .ini realiza la configuración de la red; y la aplicación programada en C++ (estándar ISO JTC1/SC22/WG21), que describe el comportamiento de los nodos.

En el archivo .NED se agregan los nodos de la simulación tipo ManetRouter que representan a los vehículos. Se deben agregar otros módulos para simular el medio físico como RadioMedium y physicalEnvironment, se agrega un visualizador y un configurador para redes IP.

En el archivo .ini se manda a llamar el archivo .NED, los archivos .xml para obstáculos y otras configuraciones. Dentro se puede modificar el movimiento de los nodos y cómo se visualiza la simulación. Aquí se configura su capa física y mac al definir qué protocolos usar, la interfaz y asignarles un radio (se configura la frecuencia, sensibilidad, etc.). Los protocolos implementados forman parte de librerías que contienen los modelos, como INET y veins.

La primera simulación sobre AODV requiere los archivos antes mencionados. Para la aplicación nos basta con una que está predefinida “pingApp”, que lleva además un contador de paquetes enviados y recibidos. Se incluye también un archivo.xml que contiene los obstáculos, en este caso edificios y construcciones. Dentro del .ini se especifica el movimiento de los nodos.

En la simulación 2 sobre DSRC, se incluye el uso de veins. La topología y movimiento de los nodos ahora se describe en sumo, y este se importa sobre el archivo .ned. Para la aplicación, se toma de base “veinsInetSampleApplication” y se modifican las funciones de startApplication() para definir el envío de mensajes, aquí se programa que periódicamente informe por broadcasting las detecciones. La función processPacket(pk) modifica el comportamiento de los nodos al recibir un mensaje, aquí se hace que los nodos lo retransmitan una vez (requiere de un buffer para guardar los últimos mensajes recibidos y ver si es nuevo). Tienen además una función stopApplication(), que en este caso no se usa.

Igual que en AODV, se usa un archivo .xml para definir los obstáculos (no se pudo importar directo de sumo). El archivo .sumocfg lleva las direcciones del resto de archivos de sumo. Config.xml incluye el tipo de modelo de propagación usado, el protocolo y frecuencia. El .rou.xml para declarar la rutas, los nodos y su tiempo de partida. Por último, el .net.xml con el mapa y la velocidad máxima en cada camino de las rutas.

A continuación, se comentan sobre los retos presentes en las redes vanet, los enfoques que se han tomado y qué mensajes se transmitirían. Por último, se describen en qué consisten los dos estándares experimentados.

## Descripción

Dentro de las redes móviles ad hoc llamadas Manet, un tipo específico son las redes vehiculares ad hoc donde los nodos son coches con módulos de comunicación inalámbrica. Dentro se ha realizado una amplia investigación y experimentación, sin embargo, no solo no hay un modelo definido o preferido, si no que dependiendo de los requisitos en la comunicación se requerirían de distintas tecnologías. En la tabla 6-1 se inicia mostrando una comparativa contra redes móviles y para vehículos aéreos no tripulados (fanet), que nos sirve para presentar la problemática inicial en una red vanet.

Tabla ‑ Comparación entre manet, vanet y fanet

|  |  |  |  |
| --- | --- | --- | --- |
|  | Manet | Vanet | Fanet |
| Movilidad | Baja | Alta | Muy alta |
| Modelo de movilidad | Aleatorio | Regular | Regular para algunos caminos, principalmente con multiUAV |
| Densidad de nodos | Baja | Alta | Muy baja |
| Cambios en topología | Lento | Rápido | Rápido |
| Modelo de propagación | A nivel de suelo, por lo general no hay LOS (Line of sight) | A nivel de suelo, No hay LOS | En el aire, LOS |
| Consumo energético | Consumo eficiente | No necesita | Consumo eficiente para mini UAV’s |
| Poder computacional | Limitado | Alto | Alto |
| Localización | GPS | GPS, A-GPS, DGPS | GPS, A-GPS, DGPS |

En el caso de redes vanet se debe lidiar con una cantidad variable de nodos, que, según el tráfico, se pueden mover a gran velocidad, y que en muchos casos no hay línea de visión. Aunque como ventaja se tiene que su movimiento es conocido, y está limitado a los caminos existentes. Esto facilitaría el uso de modelos que usan infraestructura para su conexión.

Entre las tareas de seguridad que se plantean, se incluye el avisar si tanto el vehículo usuario como otro dentro de su rango de visión se detiene, da la vuelta, cambia de carril, si desacelera, o realiza alguna acción anormal. Esto con el fin de evitar colisiones.

Para estas tareas se requiere tanto de modelos que sean seguros, confiables y de baja latencia. Para nuestras tareas además se requieren de aplicaciones de localización con buena precisión, los ideales para esta tarea son los sistemas de navegación satelital. Aquí se tiene GPS y GPS asistido (A-GPS), los cuales tienen una precisión de 10-20 metros, siendo insuficientes más que para planificación de rutas. Mientras que GPS diferencial (DGPS) usa estaciones en la Tierra para calcular y realizar correcciones (son estaciones fijas con posición conocida, que reciben su ubicación del satélite, ven el error y lo comunican a usuarios) con precisión de cerca de 1 metro. Otras ideas para mejorar la precisión incluyen el uso de unidades de medición inercial (IMU) para mantener registro de la posición entre dos mediciones con el GPS. Se pone énfasis en la localización, por que, para nuestro trabajo corresponde al principal dato que se quiere comunicar, pero en el caso del modelo de coche autónomo no se cuenta con un módulo incluido. (Hartenstein y Laberteaux 2010)

## Patrones de comunicación

Para compartir información entre vehículos se consideran distintos esquemas o patrones, que dependen según qué información se quiere compartir y a qué participantes les puede interesar. Aquí la información se puede compartir por beaconing, geocasting, unicasting o diseminación de información.

Se considera broadcasting como solo un salto, donde un vehículo comparte con todos los nodos que estén dentro del rango la información. A partir de aquí, los nodos pueden reenviar la información (multihop) con el fin de abarcar una mayor área, aquí se puede apoyar de infraestructura en la carretera para darle mayor robustez. El broadcasting se puede hacer mediante distintos métodos:

* Inundación: Todo nodo que recibe la información lo reenvía, para evitar duplicación se puede implementar contadores (Time to Live ttl) de cuántas veces se debe reenviar.
* Beaconing: Se considera como el envío periódico de información, ayuda a limitar la densidad de información en la red.
* Geocasting: Los nodos reenvían la información solo dentro de un área geográfica.

Si bien todos estos métodos son considerados de inundación, presentan mecanismos de control para evitar problemas de congestión. Otras alternativas para limitar la cantidad de mensajes que son retransmitidos dentro de la red, por ejemplo, se propone realizar broadcast según la densidad de tráfico, de forma que, con mayor tráfico, menos autos deben retransmitir; o según que tanto se aleje del nodo original, los nodos retransmitan con mayor probabilidad. (Hartenstein y Laberteaux 2010)

Entre los métodos derivados de inundación, se encuentran los protocolos gossip donde un nodo se empareja con otros nodos de forma aleatoria (en lugar de retransmitir a todos dentro de su rango), estos nodos que reciben la información a su vez están emparejados con otros a los que les comparten la información. Lo que nos lleva a otra alternativa que se explora, la idea de formar grupos o clustering. Aquí hay nodos especiales que se dedican a retransmitir la información dentro de su grupo, y nodos repetidores (relay) que pertenecen a varios clusters. En general estos clusters se hacen de forma geográfica, y los repetidores son solo los nodos donde hay overlapping entre clusters.

Por último, hay propuestas basadas en unicast que pueden funcionar según algunas aplicaciones, que requieran de que dos nodos se conecten. Sin embargo, para aplicaciones de seguridad, no es ideal ya que es muy lento y presenta dificultades de escalabilidad, lo que no es útil para nuestro objetivo.

Uno de los protocolos de enrutamiento preferidos o más explorados es “Ad-hoc on demand distance vector routing” o AODV (estándar de la IETF: RFC 3561), gracias a que se trata de un protocolo ya existente usado por ejemplo en Zigbee. Es un protocolo bajo demanda que soporta tanto unicast como multicast. Cuando se requiere transmitir un mensaje, se establece una ruta a la que, como los nodos están en movimiento, periódicamente se le debe dar mantenimiento. Es reactivo porque la ruta solo se crea cuando se necesita. AODV usa mensajes de control que incluyen:

* RREQ para solicitar ruta a un nodo, cada nodo lo retransmitirá hasta llegar al objetivo o algún nodo que conozca una ruta y esté fresca.
* RREP es la respuesta de algún nodo con la ruta, ya sea el nodo objetivo o un intermediario.
* RERR mensajes de error para avisar que una ruta ya no es válida porque un nodo ya no está activo. (Aswathy 2012)

En su aplicación como multicast, que es lo que nos interesa, usa los mismos mensajes para descubrir un grupo y crear la topología con los nodos pertenecientes. Periódicamente se mandan también mensajes de actualización RREP para mantener las rutas frescas y reparar links rotos. Por último, para el enrutamiento se pueden usar diferentes parámetros como número de saltos, o probabilidad de propagación. (Janne Salmi 2000)

Pese a ser el más usado, AODV presenta algunos problemas de escalabilidad con redes grandes, donde se vuelve caro, pues los nodos requieren de almacenamiento para mantener las tablas de ruteo. Además, puede ocurrir congestión, debido a que se envían de forma constante de búsqueda, respuesta y de errores.

Una solución extra que se propone es combinarlo con lo que se llama protocolos de agregación. En lugar de retransmitir la información como se recibe, los nodos de la red reciben y procesan la información para enviar un resumen de ésta a los demás nodos conectados a la red.

## Modelos v2x, v2i y v2v

Dentro de la comunicación vehicular se presentan diferentes modelos con diferentes aplicaciones. El primero es v2i o comunicación de vehículos con infraestructura, la cual desempeña el rol de coordinador, donde la preocupación principal está en la latencia, como en la cobertura. Mientras v2v se refiere a la comunicación entre vehículos, aquí no hay un nodo central, con una funcionalidad similar a Bluetooth. Por último, el objetivo final v2x o comunicación entre vehículos contra todo, que no solo abarcaría los anteriores modelos, pero también espera comunicarse con los celulares de los peatones. En la figura 6-1 se presentan algunos ejemplos de las aplicaciones que se pueden usar y el tipo de comunicación que requerirían.

Tabla

Descripción generada automáticamente con confianza media

Figura ‑ Fuente: (Hartenstein y Laberteaux, 2010) algunas aplicaciones y su modelo sugerido

Aquí nos centraremos en el modelo v2v donde hay dos principales estándares dominantes: DSRC basado en IEEE 802.11p (contempla capas física y MAC), y c-v2x de la 3GPP que usa tecnología celular basada en 4G LTE y 5G.

El elegido para experimentar fue DSRC, debido a que es una tecnología con más tiempo de adopción, sobre la que se han creado estándares y regulaciones en países. Comparado con c-v2x, posee mayor rango (hasta 1km), soporta altas velocidades de hasta 500 km/h, presenta menor latencia (0.4 ms en promedio vs 1ms). Además, es asíncrono, no requiere un paso previo de sincronización.

Por otro lado, hay desventajas como el número de canales limitados y que el volumen de datos que puede transmitir es menor. Igualmente, c-v2x podría tener ventajas frente a obstáculos, ya que usa otra modulación (sc-dfm), y es más fácil aprovechar infraestructura celular. (Kyle Barratt, 2021)

## DSRC

Diagrama

Descripción generada automáticamente

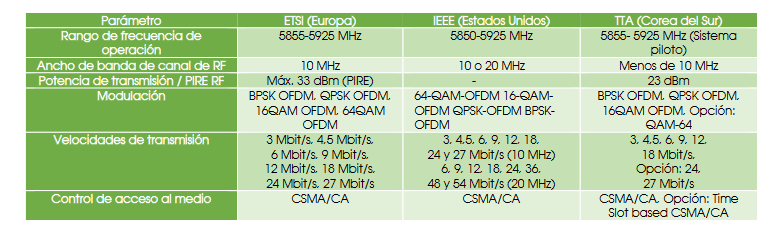
Figura ‑ Fuente: (Hartenstein y Laberteaux 2010) capas del estándar DSRC, se compone de dos partes principales, el estándar IEEE 802.11p e IEEE 1609 o wave

### 6.4.1 Capa física

DSRC o direct short range communications, es un estándar del IEEE publicado en 2002 denominado IEEE 802.11p Acceso Inalámbrico en Ambientes Vehiculares (WAVE). Parte del estándar de Wi-Fi 802.11 para aplicaciones de transporte inteligente, como la comunicación entre vehículos e infraestructura. Para el uso del estándar, el IFT en concordancia con la UIT designó la banda de frecuencias 5850-5925 MHz para servicios fijos, fijos por satélite y móvil. DSRC es un servició que incluye comunicaciones V2V y V2I, con fines de seguridad, como advertir posibles colisiones.

Como se observa en la figura 6-3, la banda considera el uso de canales de 10 MHz con motivos de robustez, con modulación QAM o OFDM, y CSMA/CA como método de control de acceso al medio. (“IDENTIFICACIÓN DE NECESIDADES DE ESPECTRO PARA SISTEMAS DE TRANSPORTE INTELIGENTE EN LA BANDA 5850-5925 MHz” 2021)

Figura ‑ Características de la banda 5850-5925 MHz. Fuente: (“identificación de necesidades de espectro para sistemas de transporte inteligente en la banda 5850-5925 MHz”, 2021)



Dentro de la capa física se trata la modulación que corresponde a la representación de datos binarios para su transmisión en un medio físico. La información es colocada sobre una señal portadora en la banda correspondiente. Existen distintos métodos, desde los más sencillos como modulación por amplitud o en frecuencia, a otros más complejos como QAM y OFDM que permiten transmitir una mayor cantidad de datos, pero son más vulnerables al ruido.

La modulación OFDM o multiplexación por división de frecuencia ortogonal se basa en la suma de N modulaciones lineales tipo QAM u otra modulación, con N portadoras diferentes. De forma que la señal queda modulada en fase y amplitud si se usa QAM, pero dispersa a lo largo de todo el ancho de banda, con las diferentes portadoras queda también modulada en frecuencia. El ancho de banda completo se divide en intervalos, al que cada uno se le asigna una modulación.

En la imagen 6-4 se observa la división del ancho de banda en subportadoras, contra una sola portadora, el número de divisiones se determina según el ancho de banda, la velocidad de los datos y la duración del símbolo, quedan separadas como: N=1/T.

Diagrama

Descripción generada automáticamente

Figura ‑ OFDM en el dominio de la frecuencia. Fuente: (Cabrera y Tarrés, s. f.)

OFDM permite un mejor aprovechamiento del espacio al permitir que se sobrelapen sus portadoras, siempre que le frecuencia central quede libre sin interferencias, cada una con la forma de una función sinc. Para su demodulación es sencillo con la transformada rápida de Fourier. (Cabrera y Tarrés, s. f.)

### Capa MAC

El estándar IEEE 802.11p abarca también la segunda capa o de control de acceso al medio (MAC), en la que se define el uso del medio de transmisión y de organización. Aquí al ser un medio inalámbrico sin un coordinador, se define el uso de CSMA/CA o acceso al medio por detección de portadora y prevención de colisiones.

DSRC hace una simplificación eliminando pasos como la sincronización e iniciación de la conexión, lo que reduce encabezados, Aun así, son pasos que serían útiles para dar seguridad a las conexiones y que se investigan en forma de lograr una configuración rápida del enlace.

El encabezado MAC como se ve en la figura 6-5 es el mismo que en Wi-Fi y consiste en encabezados de control de la trama, direcciones MAC, duración, el cuerpo del mensaje, y FCS que es la secuencia de verificación de trama. (Kenney 2011)

Tabla

Descripción generada automáticamente

Figura ‑ Fuente: (Kenney 2011) encabezados de capa 2

Con CSMA/CA en su forma básica, un nodo que quiere transmitir empieza mandando un mensaje tipo request to send (RTS) que incluye información como el total de datos a transmitir. Los nodos objetivo responden mandando un mensaje clear to send (CTS) en caso de estar libres, que incluye también el tamaño de la respuesta, de lo contrario mandan un mensaje de ocupado. El resto de los nodos, usando los mensajes RTS y CTS, calculan el network allocation vector (NAV) que es el tiempo en el que no transmitirán datos. A diferencia de solo CSMA sin CA, donde los nodos transmiten con cierta probabilidad, mediante el sensado del canal y el cálculo del NAV permite reducir las colisiones.

### Capa lógica

DSRC hace uso del estándar IEEE 802.2 control de enlace lógico, y se apoya del protocolo 802.3. Como se observa en la figura 6-6 se compone de LLC y SNAP, se trata de una subcapa que sirve de interfaz para el usuario en la capa de red, configurada para permitir conexiones desconocidas, en el campo EtherType se define 0x88DC para WAVE o 0x86DD para ipv6. (Kenney 2011)

Diagrama

Descripción generada automáticamente

Figura ‑ Fuente: (Kenney 2011) encabezados de la subcapa LLC y SNAP

### Capas superiores

El estándar “wireless access in vehicular environments” (WAVE) o IEEE 1609 son un conjunto de protocolos de acceso inalámbrico en entornos vehiculares, que sirven como recomendación para el estándar DSRC:

* IEEE 1609.4: para operación multicanal.
* IEEE 1609.3: para servicios de red en entornos WAVE, el protocolo busca reducir el overhead de UDP/IPv6, con mensajes cortos llamados WSMP que permitan transmisiones de un solo salto eficientes, el contenido se observa en la figura 6-7. Contiene la versión de WSMP usada, PSID el identificador para el tipo de servicio que se usa, en la extensión se incluye el número del canal, la tasa de transmisión y potencia de transmisión, WAVE ID indica el fin de la extensión, y la longitud del segmento de datos.

Diagrama

Descripción generada automáticamente

Figura ‑ Fuente: (Kenney 2011) formato de wave short message

* IEEE 1609.2: para servicios de seguridad, el transmisor debe tener una firma digital que garantice que tiene la autoridad para enviar mensajes y el contenido no ha sido alterado. El estándar usa el algoritmo ECDSA para firmas digitales de curva elíptica. (Kenney 2011)

## Contenido de los mensajes

Con los motivos de seguridad que se plantean en el proyecto se requiere comunicar la ubicación tanto del vehículo dada por GPS, como la localización aproximada de las detecciones, la comunicación de la velocidad calculada y el uso de bits para indicar las acciones suyas y de otros, como frenado y cambio de carril. Siguiendo un acercamiento de agregación, es decir, cada vehículo en lugar de retransmitir inmediatamente los mensajes debe modificarlos, agregar y comparar con su propia información, de forma que el área de datos sería de tamaño variable. Igualmente considerando un contexto como en geocasting no toda la información es útil, de forma que, al alejarse de cierta región, hay datos como localizaciones de otros vehículos que deben descartarse.

# Implementación de una red vanet

En este capítulo se describe a detalle cómo se implementa la simulación usando omnett para AODV, y para DSRC se añade el uso de veins y sumo. En la segunda parte se presentan los resultados alcanzados junto con gráficas.

## Arquitectura y requisitos de la red vanet

Como se repasó en el capítulo anterior, no hay un modelo bien definido de comunicación colaborativa entre vehículos, por el contrario, cada compañía lleva a cabo pruebas e investigaciones sobre sus propios modelos. Como facilitadores, tanto la IEEE y el 3GPP han sacado estándares que abarcan algunas de las necesidades que se deben de cumplir. Además, de momento la tendencia ha sido reaprovechar tecnologías y métodos ya existentes como AODV o WiFi (IEEE 802.11). En base a esto, nuestro modelo está enfocado enteramente en aplicaciones de seguridad, los mensajes que serían de tamaño variable comparten información del vehículo propio, como de las detecciones, pensando en que no todos los nodos estén conectados.

El contenido de los mensajes incluiría: en el caso del usuario, la identificación del vehículo, ubicación geográfica que requeriría del uso de sistemas como DGPS (que ofrecen mayor precisión), información referente a las rutas que tomará, como próximos movimientos y su velocidad; en el caso de las detecciones comunicar su posición actual, velocidad estimada, junto a sus próximos movimientos.

Esta información plantea 3 casos, el primero una actualización periódica de los datos, que los coches procesarían y compararían con su información, para después reenviar el mensaje en forma de un resumen actualizado. En segundo, el envío de mensajes de emergencia como que un coche frene o cambie de carril. Estos dos primeros son mensajes de tipo broadcast, para los que se considera que solo se retransmitan una vez. Por último, la posibilidad de conectarse de forma directa con otros coches (unicast) para la verificación de información.

En base a esto y como la prioridad es tener la menor latencia posible, se escoge que la red vanet está basada en comunicación vehículo a vehículo (v2v), pero se puede incluir modelos v2i o v2x, para otros servicios como planeación de rutas. Dentro del modelo v2v, se escoge como protocolo de enrutamiento para la comunicación uno a uno AODV, mientras que se realiza la simulación sobre el estándar 802.11p (primera parte de DSRC), con soporte de CSMA/CA.

## Implementación

Para el diseño de la simulación se usa OMNeT++, una biblioteca y marco de simulación en C++ orientado principalmente a modelar el tráfico en redes de telecomunicaciones. Entre sus principales bibliotecas se encuentra INET que proporciona protocolos, agentes y modelos tanto de Internet, de capa de enlace alámbrica e inalámbrica, y soporte para movilidad. Ambos nos servirán para la representación del uso de AODV en redes vanet.

Para la simulación de 802.11p, aparte de los ya mencionados, se usa SUMO, un paquete de simulación de tráfico que incluye el modelado de vehículos, transporte público y peatones, lo que nos permite diseñar nuestros escenarios, que incluye la creación de rutas, agregar y controlar nodos (vehículos). Para agregar este modelo se trabaja con VEINS, un framework de código abierto para ejecutar simulaciones de redes VANET, basado en dos simuladores: OMNeT++ y SUMO, y que añade además un conjunto de protocolos para la comunicación entre vehículos.

### AODV

Para la primera simulación, el proyecto se compone de 3 archivos principales, el primero es un .NED, que corresponde a la descripción de la topología de la red, como se observa en las figuras 7-1 y 7-2:

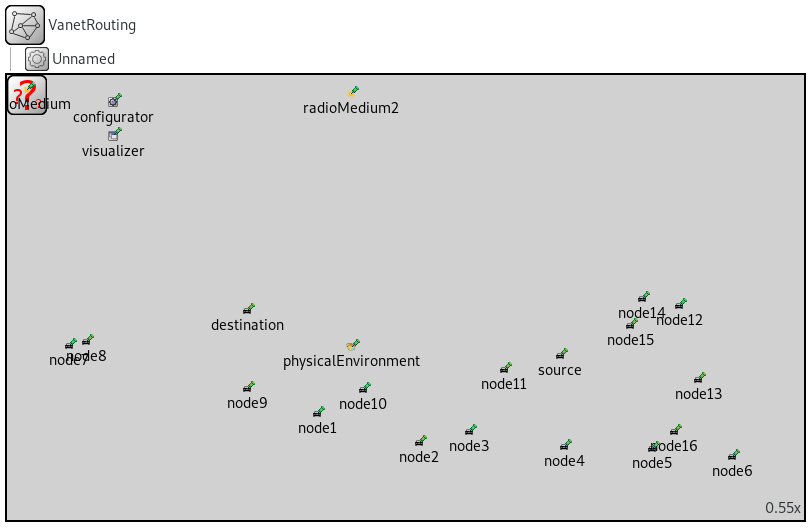


Figura ‑ Ventana de diseño para un archivo .ned



Figura ‑ Código del archivo .NED, son los módulos incluidos

En el archivo .NED se agregan los nodos tipo ManetRouter, que en nuestro caso representan los vehículos, junto a los módulos:

* radioMedium se usa el ApskScalarRadioMedium, que describe el entorno en que ocurre la comunicación, incluye modelos de propagación y calcula el ruido del medio, y cómo afecta la transmisión
* configurator se usa IPv4NetworkConfigurator para la asignación de direcciones a los nodos, y en este caso se debe definir que no se tienen rutas estáticas.
* visualizer se usa el IntegratedMultiVisualizer para generar anotaciones, como los links formados.
* physicalEnvironment para la descripción de geometrías como paredes y construcciones, que afectaran la propagación de la señal.

Una vez descrita la red, se escribe un archivo .xml que contiene los obstáculos:

<environment>

<object position="min 265 342 0" orientation="99.26 0 0" shape="cuboid 5 305 8" material="concrete" fill-color="255 255 255" opacity="0.8"/>

</environment>

El último archivo es de tipo .ini, el archivo de configuración de la red y todos los parámetros, dentro se define el tiempo de simulación y el archivo NED a usar.

sim-time-limit = 12s

network = VanetRouting

Los nodos requieren de ejecutar una aplicación para comunicarse, en este caso se usa una ya definida “PingApp”, que indica al nodo source que mande un ping al nodo destino.

\*.source.numApps = 1

\*.source.app[0].typename = "PingApp"

\*.source.app[0].destAddr = "destination"

\*.source.app[0].printPing = true

A continuación, se configura la interfaz de red para que los nodos se comuniquen entre sí, en este caso la de WiFi 802.11, se define el bitrate de 6 Mbps, la potencia de transmisión de 200 mW, el ancho de canal de 10 MHz, esto valores de acuerdo al estándar de DSRC. Además, como se hará en los resultados, se puede experimentar con la mínima potencia recibida.

\*.\*.wlan[\*].typename = "Ieee80211Interface"

\*.\*.wlan[\*].bitrate = 6Mbps

\*.\*.wlan[\*].mac.\*\*.responseAckFrameBitrate = 6Mbps

\*.\*.wlan[\*].mac.\*\*.\*Retry\* = 0

\*.\*.wlan[\*].radio.transmitter.power = 200mW

\*.\*.wlan[\*].radio.receiver.sensitivity = -85dBm

\*.\*.wlan[\*].radio.receiver.snirThreshold = 0.1dB

\*.\*.wlan[\*].radio.receiver.energyDetection = -90dBm

\*.\*.wlan[\*].radio.bandwidth = 10MHz

Para la capa física, se incluye la configuración del radio, aquí se opera sobre el módulo basado en IEEE 802.11, se define la frecuencia central de 5.9 GHz, y el uso de ODFM.

\*.radioMedium.typename = "Ieee80211ScalarRadioMedium"

\*.\*Host.wlan[\*].radio.typename = "Ieee80211Radio"

\*.\*.wlan[\*].radio.typename = "Ieee80211OfdmRadio"

\*.\*.wlan[\*].radio.centerFrequency = 5.9GHz

Para agregar el archivo .xml:

\*.physicalEnvironment.config = xmldoc("walls.xml")

\*.radioMedium.obstacleLoss.typename = "IdealObstacleLoss"

Para la movilidad de los nodos en este ejemplo se representó como:

\*.node\*.mobility.typename = "LinearMobility"

\*.node1.mobility.initialMovementHeading = uniform(6deg,10deg)

\*.node5.mobility.initialMovementHeading = uniform(186deg,190deg)

\*.node\*.mobility.speed = 8.33mps

Mientras que el área se define con:

\*\*.constraintAreaMaxX = 1462m

\*\*.constraintAreaMaxY = 816m

Por último, para agregar el uso de AODV, se incluye una aplicación para el enrutamiento:

\*.\*.routingApp.typename = "Aodv"

\*.\*.routingApp.activeRouteTimeout = 0.5s

\*.\*.routingApp.deletePeriod = 0.1s

En la figura 7-3 se observa la ventana al ejecutar la simulación:

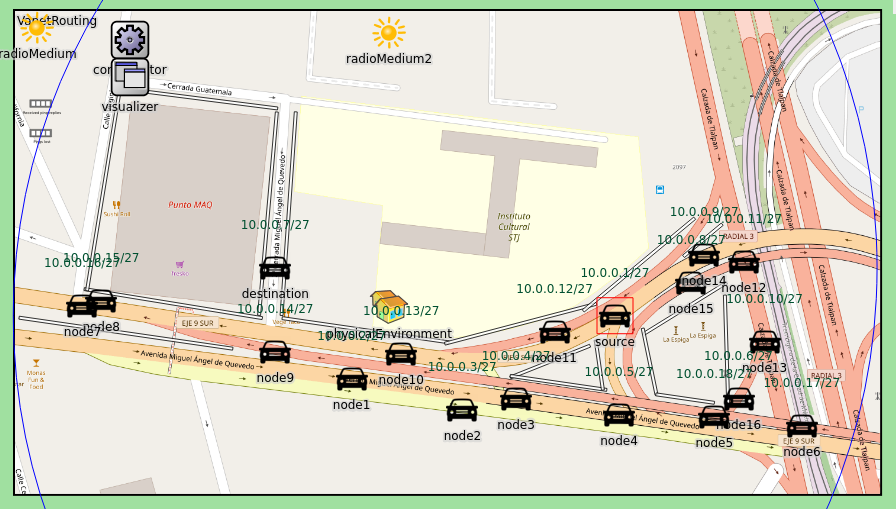


Figura ‑ Imagen de la simulación de AODV

### IEEE 802.11p

Aquí se comienza con el modelo en SUMO, se trazan de forma gráfica las rutas, junto con los edificios. Sin embargo, no se pudieron agregar en omnett los obstáculos de esta forma, por lo que se usó el archivo .xml escrito en la simulación anterior.

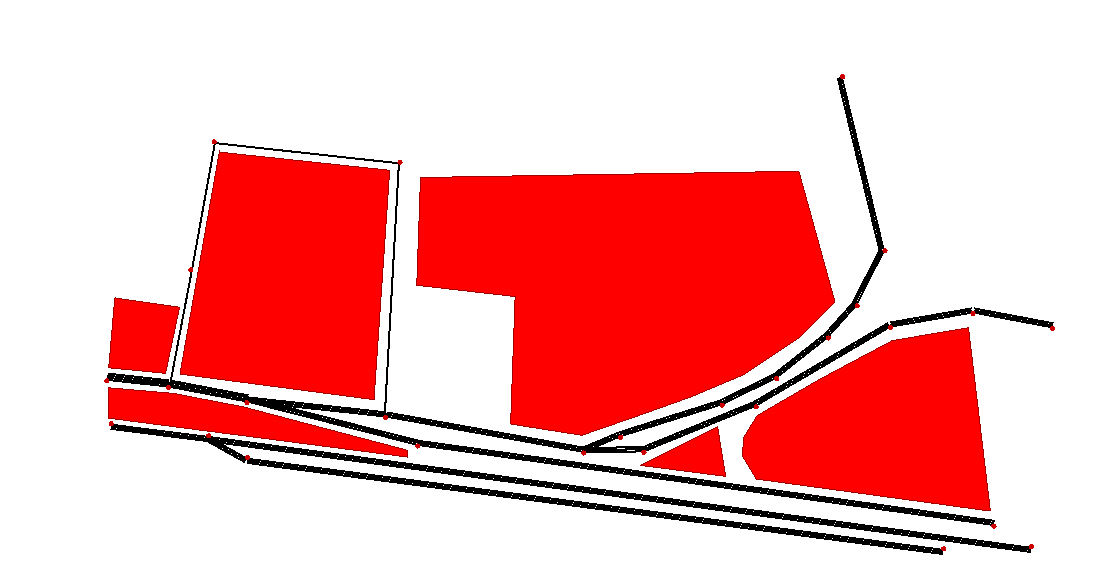


Figura ‑ Imagen del editor de una red en sumo

Se escribe un archivo .launch con los programas de SUMO, mapa2.net.xml contiene los caminos, las intersecciones y la velocidad máxima en cada uno, rutas2.rou.xml son las rutas (figura 7-6) y los vehículos (para estos se define la hora de salida y la ruta que toman), walls.poly.xml son las construcciones, para cada una se debe definir que son de tipo “building” (figura 7-5). Por último, el archivo hello2.sumocfg es creado por SUMO al ejecutarse y contiene la ubicación de los tres archivos anteriores.

<?xml version="1.0"?>

<!--debug config-->

<launch>

<copy file="mapa2.net.xml"/>

<copy file="rutas2.rou.xml"/>

<copy file="walls.poly.xml"/>

<copy file="hello2.sumocfg" type="config"/>

</launch>

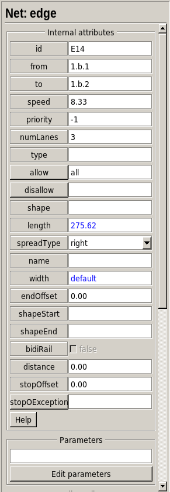
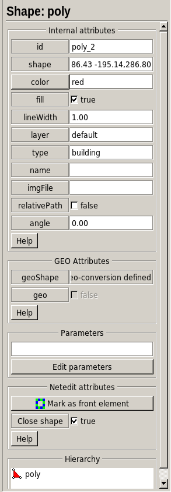


Figura ‑ Pestañas para crear construcciones (ala izquierda) y caminos (a la derecha)

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Figura ‑ Ventana para crear rutas

En el caso de los vehículos se asgregan en el archivos para las rutas:

<vehicle id="vehicle\_1" depart="2.00" route="route\_1"/>

<vehicle id="vehicle\_2" depart="4.00" route="route\_3"/>

<vehicle id="vehicle\_3" depart="5.00" route="route\_5"/>

En la figura 7-7 se observa la simulación corriendo en SUMO, los puntos amarillos son los vehículos.

Un dibujo de una cara con ojos y boca

Descripción generada automáticamente con confianza baja

Figura ‑ Imagen final en sumo

Una vez finalizada la simulación en sumo, el siguiente paso es agregarla en omnet++, se puede partir del ejemplo incluido en veins\_inet/examples/veins\_inet. Se requieren agregar los archivos correspondientes a las rutas, la red, el config y el launch. En el caso de los obstáculos se requieren agregar módulos extras en el .ned para que funcionen, o incluirlo como en la primera simulación. El archivo .NED que se muestra en la figura 7-8, al igual que el anterior incluye módulos para el radio, modelar el entorno y visualizadores, junto a un manager para controlar la comunicación.

Texto

Descripción generada automáticamente

Figura ‑ Archivo .NED para la segunda simulación

En cuanto al archivo de configuración omnetpp.ini se agrega nuevamente el archivo .ned, y el tiempo de simulación:

network = sim80211p

sim-time-limit = 120s *#120 sim 1 y 2, 260 sim3 2*

debug-on-errors = **true**

cmdenv-express-mode = **true**

image-path = ../../../../images

Igual que el anterior se requiere de una aplicación que más adelante se va a explicar los cambios realizados, con una interfaz inalámbrica.

\*.node[\*].numApps = 1

\*.node[\*].app[0].typename =

"org.car2x.veins.subprojects.veins\_inet.VeinsInetSampleApplication"

\*.node[\*].app[0].interface = "wlan0"

A continuación, para todos los nodos se agrega el estándar 802.11p con la opción de opMode, junto con el radio, la banda de frecuencia que es 5.9 GHz, potencia de transmisión 200 mW y el ancho de banda 10 MHz. Mientas el hostAutoConfigurator permite agregar direcciones ip.

\*.node[\*].wlan[0].opMode = "p"

\*.node[\*].wlan[0].radio.typename =

"Ieee80211DimensionalRadio"

\*.node[\*].wlan[0].radio.bandName = "5.9 GHz"

\*.node[\*].wlan[0].radio.channelNumber = 3

\*.node[\*].wlan[0].radio.transmitter.power = 200mW

\*.node[\*].wlan[0].radio.bandwidth = 10 MHz

\*.node[\*].wlan[\*].radio.antenna.mobility.typename =

"AttachedMobility"

\*.node[\*].wlan[\*].radio.antenna.mobility.mobilityModule =

"^.^.^.^.mobility"

\*.node[\*].wlan[\*].radio.antenna.mobility.offsetX = -2.5m

\*.node[\*].wlan[\*].radio.antenna.mobility.offsetZ = 1.5m

\*.node[\*].ipv4.configurator.typename = "HostAutoConfigurator"

\*.node[\*].ipv4.configurator.interfaces = "wlan0"

\*.node[\*].ipv4.configurator.mcastGroups = "224.0.0.1"

El módulo “VeinsInetMobility” permite rastrear la posición del nodo, aquí se usa el manager para agregar los archivos de SUMO. Para los obstáculos se agregan con el módulo physicalEnvironment junto a un modelo de propagación, que en este caso IdealObstacleLoss se refiere que la señal no pasará los obstáculos, toda la potencia se pierde.

\*.node[\*].mobility.typename = "VeinsInetMobility"

\*.manager.updateInterval = 0.1s

\*.manager.host = "localhost"

\*.manager.port = 9999

\*.manager.autoShutdown = **true**

\*.manager.launchConfig = **xmldoc**("vanetSumo2Sim2.launchd.xml")

\*.manager.moduleType =

"org.car2x.veins.subprojects.veins\_inet.VeinsInetCar"

\*.physicalEnvironment.config = **xmldoc**("walls2.xml")

\*.radioMedium.obstacleLoss.typename = "IdealObstacleLoss"

En el caso de las aplicaciones se tratan de módulos que implementan comportamientos a cada nodo, para generar patrones de tráfico, para nuestro modelo se tienen dos métodos principales: el método startApplication() con el cuál se controlan los nodos para el envío de un mensaje, para generar eventos y con veins controlar su comportamiento (velocidad, rutas, etc.); el otro método es processPacket para definir el comportamiento de los nodos cuando reciben un mensaje.

Se tienen dos casos, el primero representa el envío de mensajes de emergencia, aquí se representa un accidente o que un coche se incorpore a otro carril, el segundo caso es el envío periódico de mensajes para mantener informados a los nodos. En ambos casos, los nodos para procesar el paquete deben de retransmitirlo una vez, para el segundo caso, como se reciben múltiples mensajes, debe de tener un buffer para saber qué mensajes ya recibió.

Para controlar el nodo se usa if (getParentModule()->getIndex() == 0) para definir qué acciones realizar, con traciVehicle se puede controlar el vehículo como cambiar la velocidad (traciVehicle->setSpeed(0);) o ruta. Para programar el envío de un mensaje, se tiene un módulo aparte que representa el mensaje junto con su contenido, se define el tamaño, se rellenan sus variables, luego se crea el paquete y se le agrega el payload, con sendPacket se envía a todos.

auto payload = makeShared<VeinsInetSampleMessage>();

payload->setChunkLength(B(100));

payload->setRoadId(traciVehicle->getRoadId().c\_str());

timestampPayload(payload);

auto packet = createPacket("ChangingLane");

packet->insertAtBack(payload);

sendPacket(std::move(packet));

Por último, con una función callback se puede definir cuando ocurren los eventos:

timerManager.create(veins::TimerSpecification(callback).onesh

otAt(SimTime(70150, SIMTIME\_MS)));

En processPacket los nodos reciben un paquete enviado por broadcast, con peekAtFront se leen las cabeceras del mensaje:

auto payload = pk->peekAtFront<VeinsInetSampleMessage>();

Se puede extraer la información del mensaje y según el contenido realizar alguna acción (payload->getRoadId()), igual con traciVehicle se controla el comportamiento del nodo que recibió el mensaje.

Los nodos tienen una variable para saber si ya han retransmitido un mensaje (haveForwarded), pero solo funciona con un mensaje. Para retransmitir, se crea un nuevo paquete:

auto packet = createPacket("relay");

packet->insertAtBack(payload);

sendPacket(std::move(packet));

haveForwarded = true;

Para el caso del envío periódico de mensajes, se agregan variables donde se guardan el identificador de los últimos mensajes recibidos, para saber si ya lo había retransmitido, además pasado cierto tiempo se retiran estos mensajes por su antigüedad,

En la figura 7-9 se muestra la simulación final, para ejecutarla debe estar corriendo veins (se abre la aplicación veins\_launchd) y se lanza el archivo omnetpp.ini.

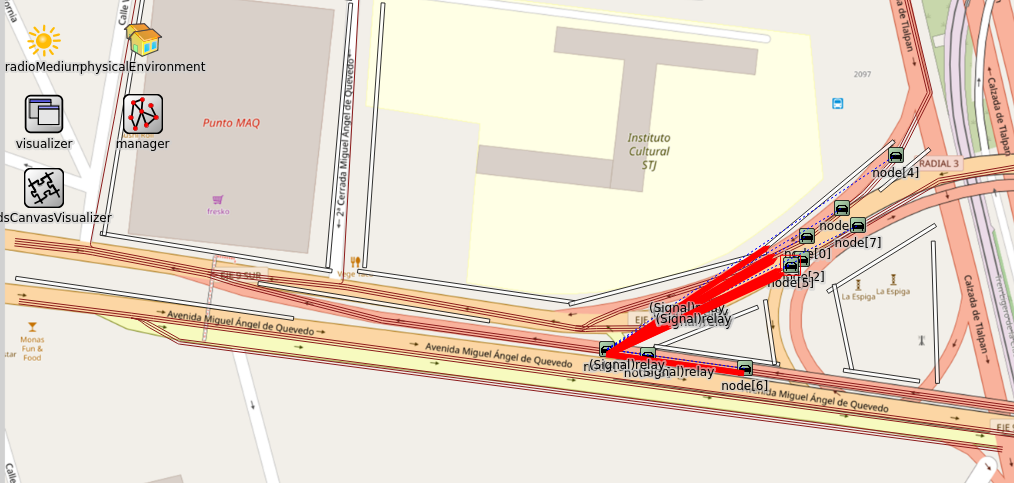


Figura ‑ Imagen de la simulación corriendo en omnet++, se observa un nodo retransmitiendo a todos un mensaje

# Resultados

## Simulación 1

Para la primera simulación se preparan un escenario como el que se observa en la figura 8-1, para los que se experimenta los resultados del protocolo AODV al intentar mandar un mensaje ping entre dos nodos, se mide el porcentaje de mensajes recibidos, tasa de errores y, el tiempo de ida y vuelta de un mensaje. Entre los parámetros principales para ver su desempeño, se varía la velocidad de los nodos, y el intervalo de transmisión entre mensajes.

Figura ‑ simulación 1, se experimenta el desempeño con diferente densidad de nodos



Los resultados varían según una serie de parámetros que configuran tanto el medio físico como la capa MAC en la simulación. En el medio físico, como se parte del estándar 802.11p se tiene que la frecuencia de transmisión es en 5.9 GHz, y la potencia máxima de 200 mW (23 dBm), además, es importante la sensibilidad del receptor, para la que se eligió un valor normal que es de -85 dBm.

Con esto y tomando como ganancia de las antenas 0 dBm, con el modelo de free space path loss, se tiene una cobertura con línea de visión de 1 km de radio, distancia que parece ser más que suficiente para medidas de seguridad. El problema en cambio se encuentra con los obstáculos, pues pueden ser puntos ciegos para los vehículos, por lo que se considera el escenario de “ideal obstacle loss”, donde la señal no atraviesa las construcciones.

Igualmente, en el protocolo de AODV es importante considerar puntos como el intervalo en que se mantienen activas las rutas, pues, un intervalo pequeño significa mayor inundación en la red de mensajes, mientras que, por otro lado, un tiempo grande tiene el problema de que la topología cambia con los nodos moviéndose, por ello en cada escenario se experimenta con diferentes velocidades: 30, 50 y 80 km/h, en las que los nodos deben cubrir la misma distancia recorrida, aproximadamente de 110 metros.

En cuanto al interés de variar el intervalo de transmisión entre mensajes ping, es que AODV se trata de un protocolo de inundación reactivo, de forma que un intervalo alto siempre significará mandar mensajes para encontrar una ruta al nodo destino, pero un intervalo bajo puede o no llevar a la inundación de la red, con la topología cambiando según la velocidad y densidad de los nodos puede llevar a retrasos en la red si se necesitan buscar constantemente nuevas rutas, o se pueden usar las ya descubiertas. De forma que en estos casos conviene que el intervalo de envío de mensajes sea adaptable, según las características de la red.

En esta simulación 1 se representa al nodo destino que se va a incorporar a la vía principal, y el nodo fuente que no tiene línea de vista busca comunicarse con él, con AODV un nodo empieza buscando una ruta en sus tablas al nodo destino, en caso de no tener una ruta fresca, manda mensaje por broadcast a los nodos alrededor que responden si conocen una ruta o por el contrario retransmiten a otros nodos para preguntar, una vez adquirida la ruta manda el mensaje y espera una respuesta.

Las características de nuestro escenario inicial son:

* Potencia de transmisión 200 mW
* Sensibilidad del receptor -85 dBm
* Velocidad de los nodos 30 km/h
* Tasa de transmisión 6 mbps
* Densidad baja
* Intervalo entre mensajes 1s
* Tiempo de vida de las rutas 0.5s

En la figura 8-2 se observa las posiciones iniciales y finales de los nodos, a propósito, se diseñó el primer escenario para que hubiera un momento en que los nodos no tuvieran un camino para comunicarse, esto será importante cuando se reduzca el intervalo entre mensajes, pues deberán mandarse mensajes del protocolo AODV para buscar nuevas rutas. En la primera simulación que dura 12 segundos, el nodo fuente intenta enviar 12 mensajes, pero solo recibe respuesta de 8, lo que es una tasa de pérdida del 33.33% con un rtt promedio de 436.4 ms. En la gráfica 8-12 se observan los mensajes del protocolo AODV enviados por cada nodo, corresponden al descubrimiento de rutas e informes de errores cuando se rompe un link. El nodo fuente intenta enviar 12 mensajes, con un intervalo entre ellos de 1 segundo, por lo que siempre se tienen que descubrir las rutas antes, mientras cada nodo debe enviar entre 7 a 15 mensajes en la duración de la simulación. En la misma figura se observan los mensajes recibidos por cada nodo, en naranja los mensajes ping que se transmitieron correctamente (8), la cantidad de mensajes recibidos depende de la posición del nodo, pero los cercanos reciben y procesan alrededor de 80 mensajes cada uno.



Figura ‑ A la izquierda la posición inicial, a la derecha la posición final

Todos estos mensajes transmitiéndose en la red tienen dos impactos importantes, hay colisiones cuando dos nodos transmiten simultáneamente, pero también significa retrasos a la hora de transmitir, pues un nodo debe esperar a que termine el anterior. En las gráficas observamos la tasa de bits de errores y la tasa de paquetes con error (son los paquetes recibidos con uno o más bits con errores), donde como se esperaría, los nodos fuente y algunos nodos que forman la trayectoria al nodo destino son los que más errores tienen. En el caso de los tiempos de procesamiento (figura 8-4), omnet lo divide en dos “pending queue” e “in progress frame”, el primero considera desde la necesidad de mandar un paquete y esperar a que el canal esté libre, el nodo fuente tarda 1.3 ms, mientras la segunda cuenta después de crear el paquete y que se transmita en el medio (junto a los tiempos de contención), que para el nodo fuente es 1 ms.

Figura ‑ resultados del escenario 1

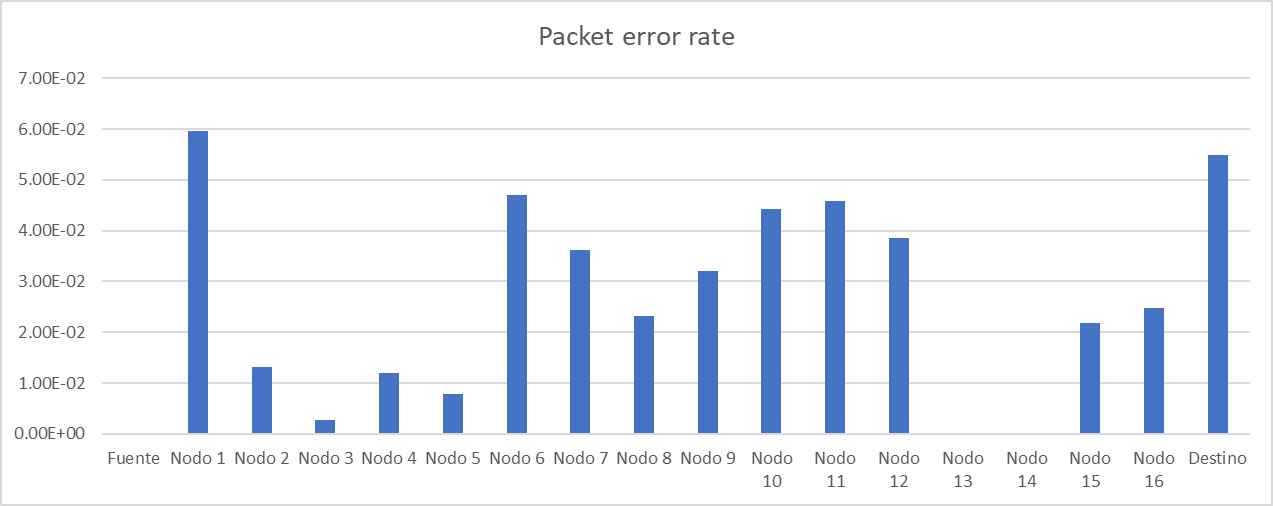
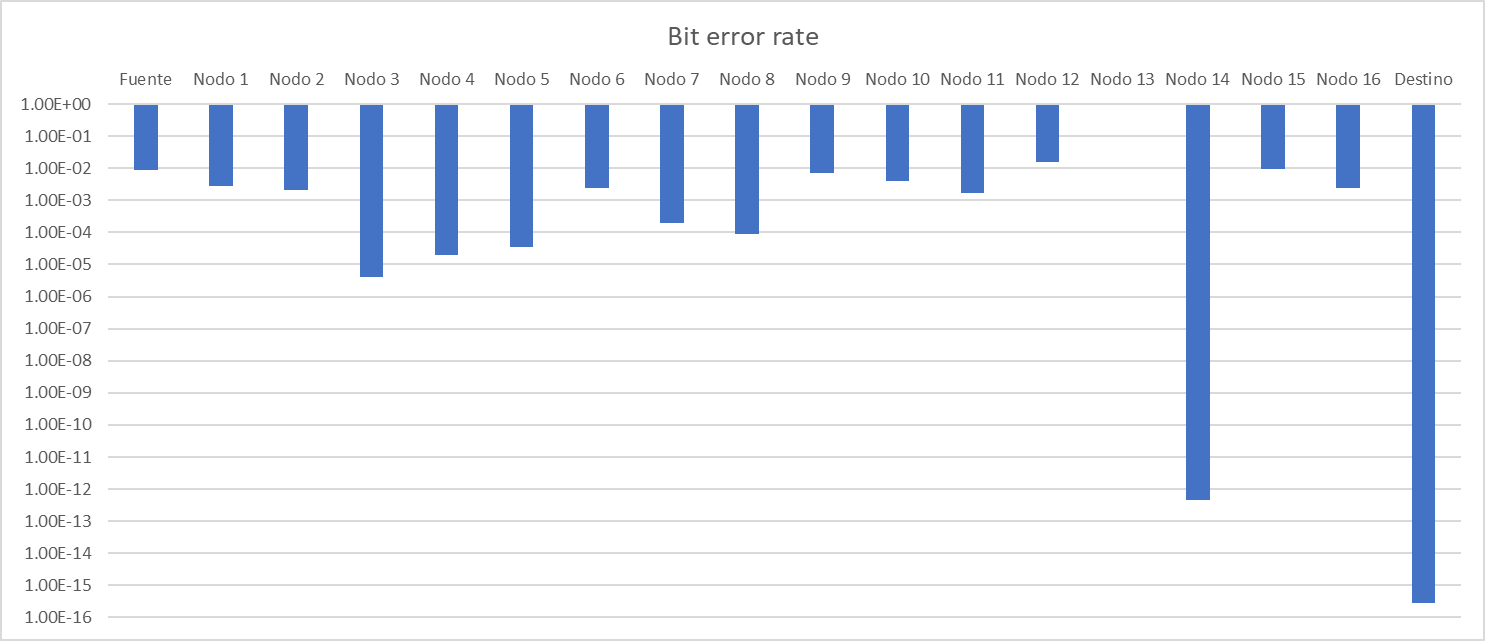
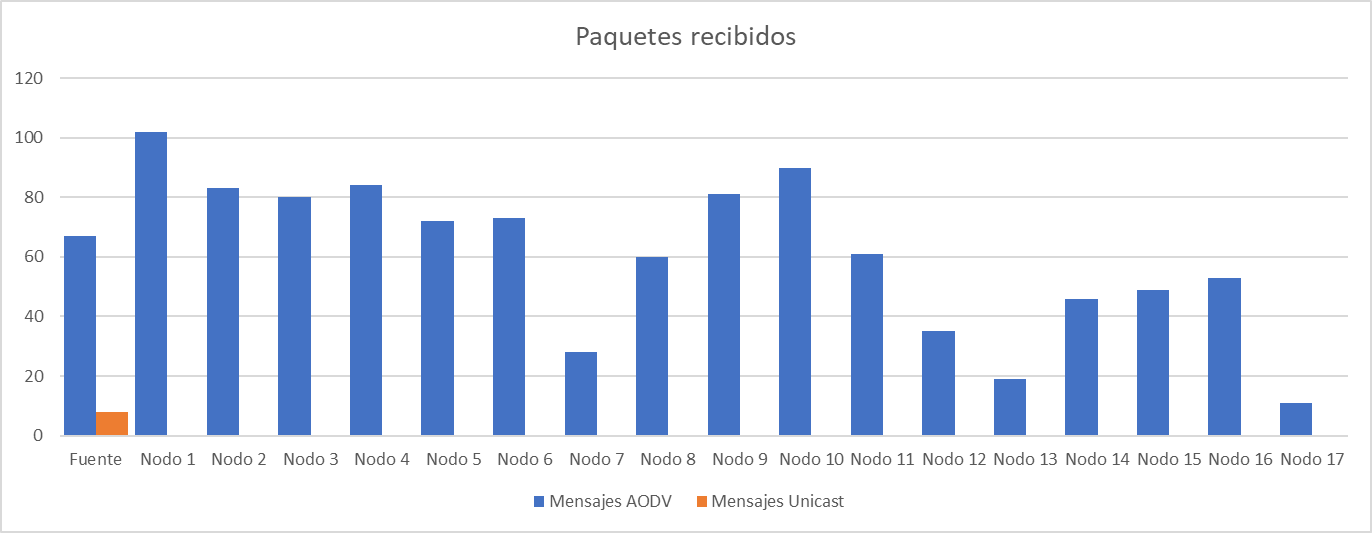
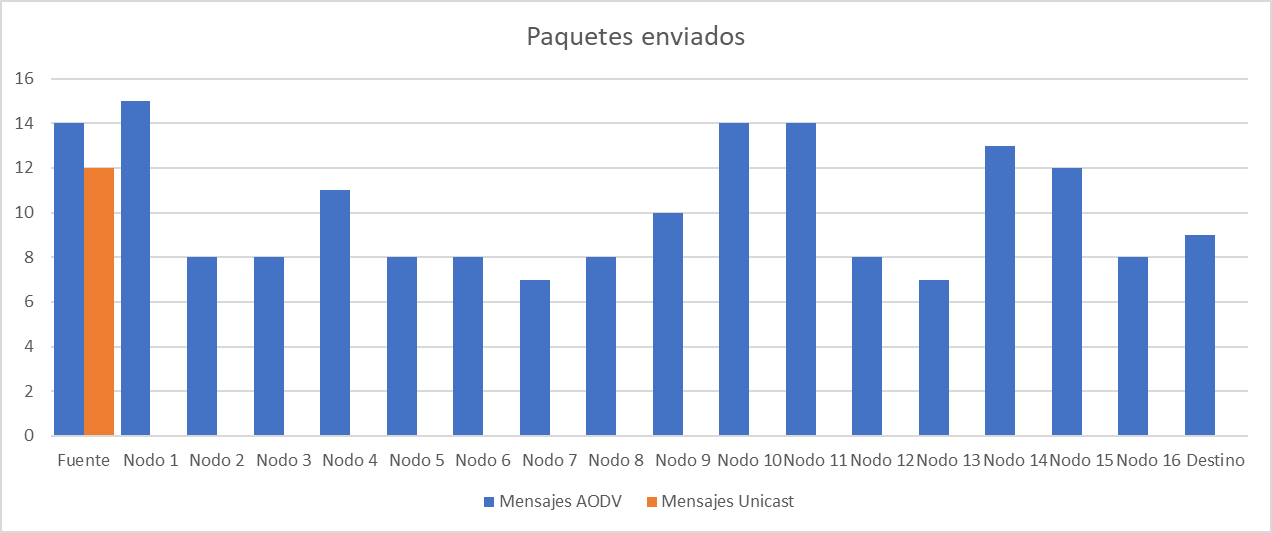
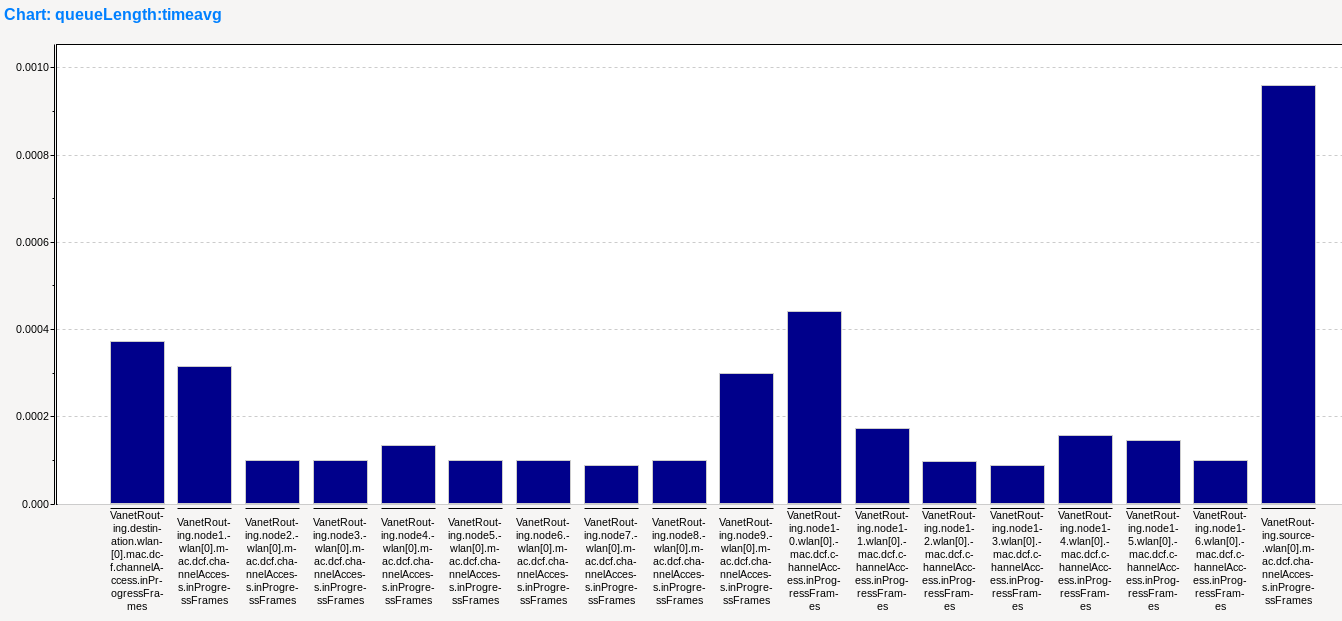
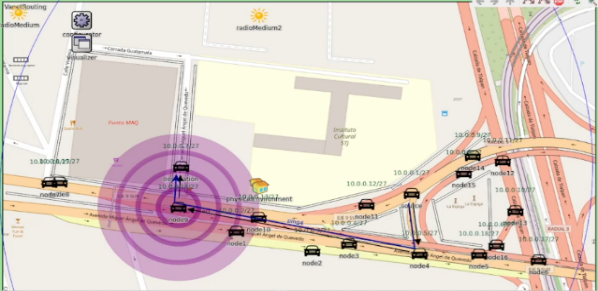


Figura ‑ a la izquierda los tiempos que demora "in progress frames" a la derecha "pending queue"



En la figura 8-5 se muestra visualmente la simulación.

Figura ‑ Rutas formadas por el protocolo AODV



A partir del primer caso analizamos y comparamos con las demás simulaciones y se presentan algunas gráficas interesantes, otros parámetros que se pueden variar y experimentar es por ejemplo con la potencia de transmisión o sensibilidad del receptor, lo que significa un menor radio de cobertura, los mensajes requieren más saltos para llegar al objetivo y las rutas son más fáciles de romperse, lo que significa que menos mensajes se reciben, puede ser interesante observar el impacto de la inundación de la red.

Por otro lado, otro experimento puede ser subir la taza de transmisión, por ejemplo, a 24 mbps, se reciben 8 paquetes de 12 (se pierden el 33%) con un mismo rtt que en el caso 1, los cambios son que lleva a una menor inundación en la red de mensajes, al mismo tiempo que sube la tasa de errores, con mayores velocidades se esperaría ver que los tiempos de encolamiento crecieran y la llegada de paquetes bajara de forma importante. Es importante mencionar que, en las gráficas presentadas de paquetes enviados y recibidos, en color azul considera solo los mensajes del protocolo AODV, no cuenta como enviados los pings.

### Diferentes velocidades e intervalos

Tabla ‑ RTT y porcentaje de pérdidas de mensajes, al variar la velocidad e intervalo

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Velocidad de los nodos | Intervalo entre mensajes | Mensajes transmitidos | Mensajes recibidos | Porcentaje de pérdidas | RTT |
| 30 km/h | 1s | 12 | 8 | 33.3% | 436.4 ms |
| 30 km/h | 0.5 s | 24 | 15 | 37.5% | 617.6ms |
| 30 km/h | 0.2 s | 60 | 20 | 66.6% | 76.65 ms |
| 30 km/h | 0.1 s | 120 | 85 | 29.16% | 2844.43 ms |
| 50 km/h | 1 s | 8 | 2 | 75% | 409.1ms |
| 50 km/h | 0.5 s | 16 | 5 | 68.75% | 135.57ms |
| 50 km/h | 0.2 s | 40 | 12 | 70% | 70.75 ms |
| 50 km/h | 0.1 s | 80 | 13 | 83.75% | 491.5 ms |
| 80 km/h | 1 s | 5 | 2 | 60% | 171.9 ms |
| 80 km/h | 0.5 s | 11 | 0 | 100% | - |
| 80 km/h | 0.2 s | 27 | 15 | 44.4% | 345.3 ms |
| 80 km/h | 0.1 s | 55 | 5 | 90.9% | 321.1 ms |

Con AODV se había configurado que las rutas se mantuvieran activas 0.5 segundos, donde este contador se reinicia cada que se recibe un nuevo mensaje. Aun así, con un intervalo entre mensajes menor a 0.5 segundos, no se garantiza que no se deban buscar nuevas rutas, debido a grandes cambios en la topología. Lo que puede significar un aumento grande de mensajes en la red, especialmente a altas velocidades. Igualmente, un mal intervalo de transmisión, en el que se requiera buscar nuevas rutas constantemente, lleva a un aumento en el número de colisiones y errores en paquetes, aumenta el tiempo de acceso al medio, lo que lleva a que el RTT lo haga también.

De los resultados, AODV resulta especialmente beneficioso a partir del intervalo de 0.2 segundos. Donde, aún con velocidades altas, los cambios en la topología no son lo suficientemente significativos para actualizar las rutas. Esto reduce la inundación de mensajes en la red, al prescindir de los mensajes de estado de AODV. Si bien, al aumentar la tasa de envío de mensajes, aumenta la tasa de paquetes perdidos, hay suficiente redundancia en la red para que los nodos queden informados. Lo que se debe controlar aquí, es al incrementar el número de nodos comunicándose simultáneamente, que la inundación no introduzca fuertes retrasos para acceder al medio, como se comprueba en la simulación 2.

En las figuras 8-6 y 8-7, se muestra el impacto que tiene la velocidad de los vehículos, sobre la probabilidad de recibir 2 y 4 mensajes respectivamente, sobre un recorrido de 100 metros. En el caso de 4 mensajes, haría que fuera totalmente insuficiente el intervalo para transmitir de 1 segundo.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Figura ‑ Probabilidad de recibir 2 mensajes con AODV

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Figura ‑ Probabilidad de recibir 4 mensajes con AODv

Aún sin otros nodos comunicándose, nos muestra las limitaciones que se tiene ya con intervalos muy bajos como de 0.1 segundos. En el cual, si bien la cantidad de mensajes recibidos es buena hasta llegar a 80 km/h, el RTT crece de forma considerable. Con 80 km/h empeora su rendimiento especialmente, por la presencia de mensajes de mantenimiento de AODV. En la simulación se debe a que, al haber una baja densidad de nodos en la red, los links se rompen y se requiere encontrar nuevas rutas. Además, se debe considerar el tiempo que toma preparar los mensajes, que en este caso superaban el intervalo de 0.1 segundos, lo que añade otros retrasos junto a encolamiento. En el caso del nodo de detección para el carro ni siquiera se alcanzan los 10 Hz de frecuencia.

Al subir la velocidad, lo observado es la drástica caída en la tasa de mensajes recibidos. Debido en parte al aumento de colisiones y los tiempos de encolamiento, pues se requiere de más mensajes de AODV para el mantenimiento de las rutas. En especial para 80 km/h, la comunicación peer to peer se vuelve costosa, cuando se requieren descubrir rutas bajo petición. Como se ve, intervalos como 0.5 segundos a 80 km/h ni siquiera funcionó y el intervalo de 0.2 observa un aumento en el RTT.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Velocidad de los nodos | Intervalo entre mensajes | Mensajes transmitidos | Mensajes recibidos | Porcentaje de pérdidas | RTT |
| 30 km/h | 0.2 s | 60 | 60 | 0% | 2.18 ms |

Los resultados nos mostrarían que el escenario óptimo sería con un intervalo de 0.2 segundos entre mensajes, para aprovechar las rutas activas, sin inundar la red. Cambiando a un escenario con mayor densidad de nodos (se probó aumentando de 16 nodos a 22), hay mayor facilidad para encontrar una ruta, que además pueda ser óptima. Esto permite reducir los mensajes de mantenimiento de AODV, que no todos los nodos deban ser partícipes en el descubrimiento de rutas, y reducir el rtt. En casos con alta densidad de nodos, parecería que un intervalo entre mensajes de 0.1 segundos puede ser posible.

En general, AODV se beneficia enormemente de que las rutas estén activas y frescas, junto a la presencia de más nodos. Esto último, no lleva a un aumento de mensajes de AODV recibidos. Como en el ejemplo mostrado, la tasa de errores y tiempos de encolamiento pueden disminuir de forma considerable.

Los resultados nos muestran que conviene usar un intervalo adaptativo entre el envío de mensajes, según la densidad y velocidad del tráfico, para casos con baja densidad de nodos, no es posible mantener intervalos muy bajos como de 100 ms entre mensajes. Mientras que, si bien puede mantener bajos intervalos a altas velocidades, se observa el aumento en el RTT y de mensajes requeridos para mantener las rutas, por lo que, al sumar más nodos transmitiendo y comunicándose, habrá un aumento en las colisiones y tiempos de encolamiento, que lleva a la necesidad de reducir la frecuencia con la que actualizan los nodos, aún sin considerar los tiempos de procesamiento de los mensajes.

La simulación nos muestra la problemática de AODV para establecer rutas a altas velocidades. El mejor resultado fue con un intervalo de 0.2 s entre mensajes, donde de 27 se entregaron y regresaron correctamente solo 15 (el 44%), el intervalo vivo que puede mantener una ruta depende de los cambios en la topología, y requiere más mensajes de mantenimiento en las rutas. Esto nos indicaría que aumentar la inundación de mensajes en la red o el uso de protocolos de enrutamiento proactivo podrían funcionar, sin embargo, como se verá en la simulación 2, no es ideal.

La idea del uso de AODV es únicamente con funciones para coordinación entre dos nodos, una vez estén notificados de la presencia del otro con DSRC. Se debe tener en cuenta que AODV es solo un protocolo que se puede implementar sobre capa 3 o 2. Para la simulación, se implementó en capa 3, y para las capas físicas y mac se usaron las mismas que en DSRC. Ambos son protocolos diferentes, por lo que para el uso de AODV se necesita de otra banda (canal), no necesariamente en 5.9 GHz. E incluso hay investigaciones para el uso de canales con AODV.

A continuación, se muestra el desempeño de DSRC con la presencia de más nodos. El mismo comportamiento debería poder trasladarse a AODV cuando más nodos se comunican entre sí. Ya que como se vio, con la presencia de nodos extras, no se incrementa la inundación con mensajes de AODV, sino que disminuye.

## Simulación 2

Para la segunda simulación consideramos el caso en el que todos los nodos intentan comunicarse entre sí (se compone de 11 nodos, 9 transmiten sus detecciones y todos retransmiten). Con el objetivo de informar de las detecciones, los nodos al recibir un mensaje lo retransmiten siempre una vez, incluido el que lo envió originalmente. Al igual que en la simulación anterior, se experimenta variando la velocidad de los nodos y el intervalo entre los mensajes, se mide la tasa de errores, tiempos de encolamiento, y el número de mensajes enviados y recibidos.

Para esta simulación, gracias al uso de veins, se implementa de forma completa el protocolo 802.11p, en el archivo omnetpp.ini únicamente se especifican los archivos de configuración y de obstáculos (junto al modelo de propagación) de la red, las modificaciones principales se hacen sobre la aplicación para definir eventos de cuando transmitir un mensaje y qué hacer al recibir uno.

Para mostrar el funcionamiento y la idea de cómo se aplicaría, primero se hizo un ejemplo práctico en el que el nodo 6 llega a una intersección y quiere incorporarse a los carriles principales, la secuencia se observa de las figuras 8-8 a la 8-11.

Imagen que contiene Esquemático

Descripción generada automáticamente

Figura ‑ El nodo 6 en rojo quiere transmitir

Imagen que contiene Diagrama

Descripción generada automáticamente

Figura ‑ El nodo 6 transmite el mensaje, y los demás nodos al recibirlo retransmiten una vez, el nodo 7 y 4 no son vistos por el nodo 6, con las retransmisiones quedan informados.

Esquemático

Descripción generada automáticamente con confianza media

Figura ‑ Los nodos cercanos 5, 11 y 7 bajan su velocidad y permiten que el nodo 6 se incorpore, el nodo 4 no baja su velocidad

Imagen que contiene Esquemático

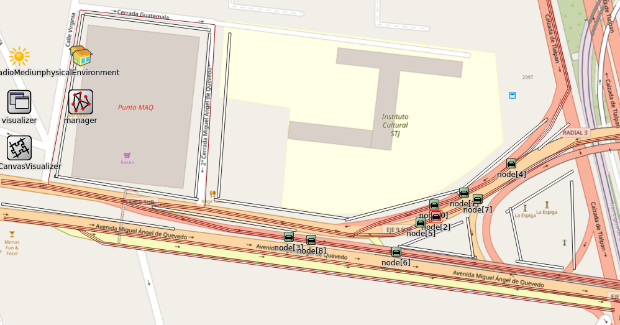
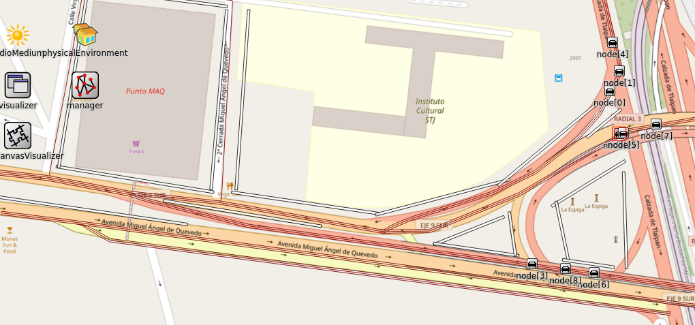
Descripción generada automáticamente

Figura ‑ Posición final, el nodo 6 se incorpora

Para experimentar con la velocidad, y ver qué tantos mensajes pueden recibir al recorrer una misma distancia, como se observa en la figura 8-12, los nodos parten de una posición inicial a una final.

Cada nodo intenta transmitir de forma aleatoria dentro de un intervalo de 100 ms, de forma que podamos observar colisiones, durante la simulación los nodos 3, 8 y 6, no son alcanzados por quien transmite el mensaje original, pero si con las retransmisiones.

Figura ‑ A la izquierda se observa la posición de partida de los nodos, a la derecha la posición final



Como en la simulación 1, partimos del primer escenario para comparar, en este, todos transmiten sus detecciones cada segundo, a excepción de los nodos 1 y 6. Los vehículos se mueven a 30 km/h, en una simulación que dura 40 segundos, lo que corresponde a que cada nodo intentará transmitir 40 veces. En las gráficas se observan los mensajes enviados y recibidos por cada nodo, estos toman en cuenta tanto sus transmisiones como retransmisiones, el caso de un intervalo entre transmisiones de 0.1 no es viable (además de presentar problemas computacionales), pues ocasiona muchas colisiones y retransmisiones de mensajes con lo que los tiempos de encolamiento aumentan bastante, y considerando que la red apenas está compuesta de 9 nodos, se esperaría que con el aumento del número de participantes, intervalos para comunicarse como de 0.2 segundos también presenten problemas, que nos muestra la falta de un nodo central que coordine a los demás nodos para transmitir.

Los nodos 1 y 6 no transmiten y no se cuentan en el promedio de mensajes enviados. En los casos presentados, todos los nodos logran transmitir todas sus detecciones. Se esperaría que al disminuir el intervalo o añadir más nodos, no siempre logre mandar todos sus mensajes, pueden ocurrir colisiones o que el medio esté ocupado, y que tengan que retrasarse en la cola.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Velocidad | Intervalo de transmisión | Duración de la simulación | Mensajes programados | Retransmisiones en promedio | Mensajes enviados por nodo en promedio | Mensajes recibidos promedio |
| 30 km/h | 1s | 40 s | 41 | 242 | 288 | 985 |

Figura ‑ Paquetes enviados (incluye retransmisiones) y paquetes recibidos por cada nodo

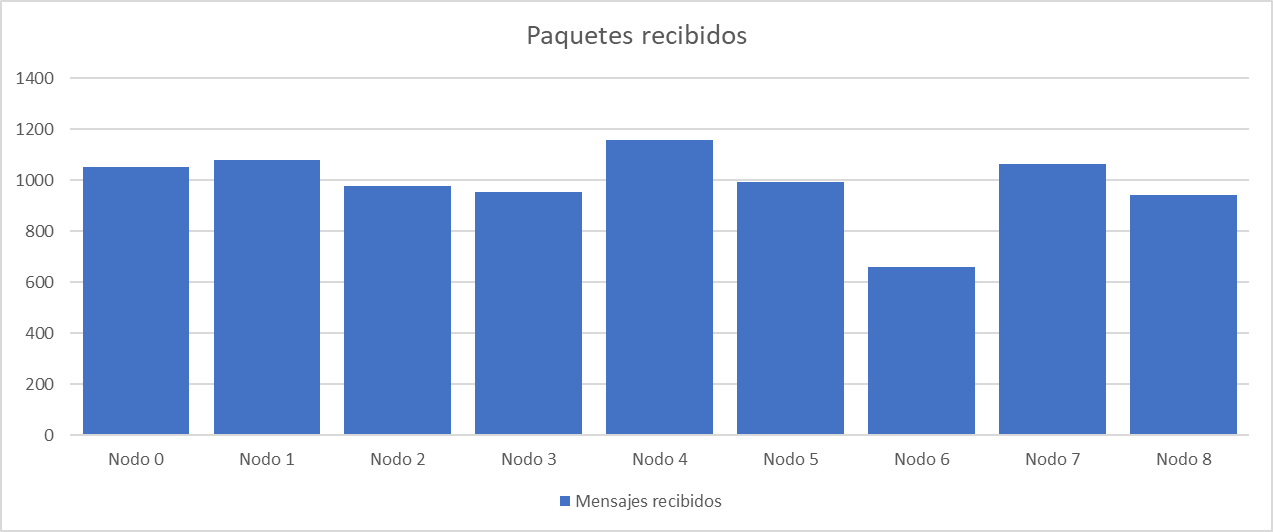
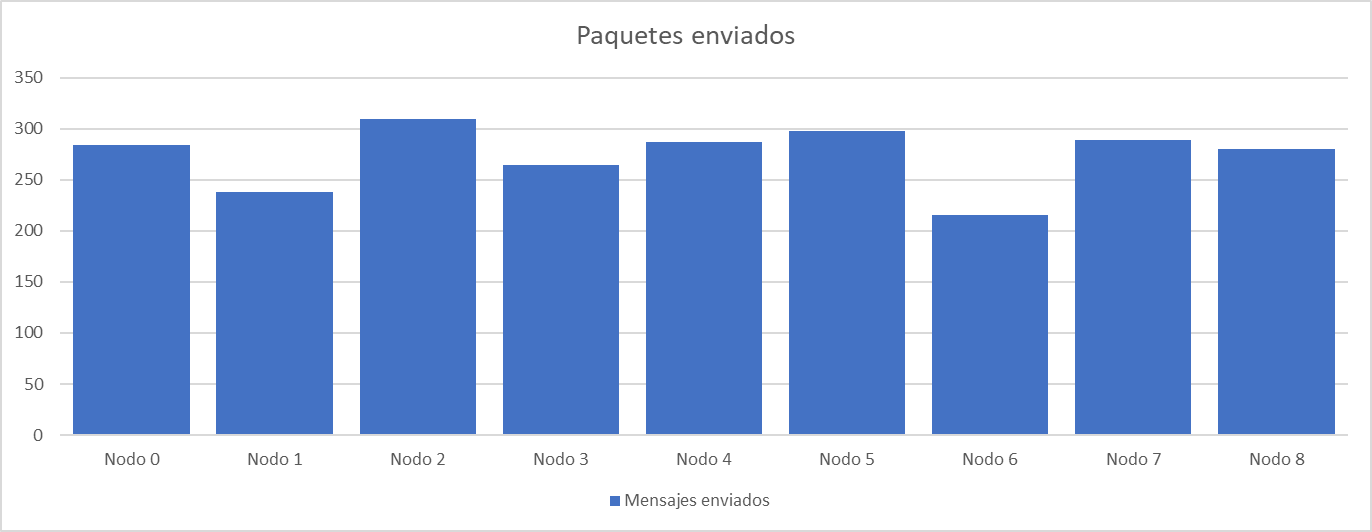
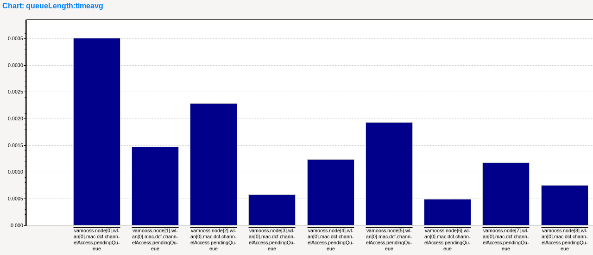
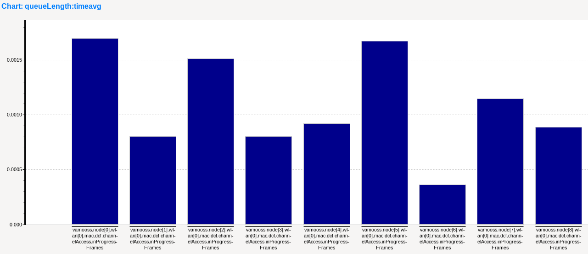


Figura ‑ Tiempos de espera promedio de cada nodo



Como en la simulación 1, los tiempos están divididos en “pending queue” e “in progress frames”, estando en promedio para acceder al medio en 15 ms, y para transmitir 30 ms. Ya desde el primer caso, que sería el mejor con una velocidad de 30 km/h y un intervalo grande entre transmisiones, los nodos dejan de recibir muchos mensajes. Si bien no todos los nodos se alcanzan entre sí durante todo el tiempo de la simulación, la mayoría se debe a colisiones a la hora de transmitir o retransmitir. Se podría decir que en la red se debían comunicar 287 mensajes en total de los 7 nodos que informan, de ellos, se recibieron en promedio 242, el 84.32%.

En la tabla se resumen los resultados, para el caso de transmitir un mensaje de 1 kB de Payload:

Tabla ‑ Mensajes recibidos en promedio de cada nodo, retransmisiones en promedio son el número de mensajes únicos que reciben

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Velocidad | Intervalo de transmisión | Duración de la simulación | Mensajes programados para el envío | Retransmisiones en promedio | Mensajes enviados por nodo en promedio | Mensajes recibidos promedio |
| 30 km/h | 1s | 40 s | 41 | 242 | 288 | 985 |
| 30 km/h | 0.5 s | 40 s | 81 | 310 | 392 | 1175 |
| 30 km/h | 0.2 s | 40 s | 201 | 770 | 978 | 3000 |
| 50 km/h | 1 s | 24 s | 25 | 102 | 128 | 413 |
| 50 km/h | 0.5 s | 24 s | 49 | 187 | 239 | 715 |
| 50 km/h | 0.2 s | 24 s | 121 | 509 | 635 | 2094 |
| 80 km/h | 1 s | 15 s | 16 | 57.44 | 74.5 | 200 |
| 80 km/h | 0.5 s | 15 s | 31 | 110.3 | 142.28 | 394 |
| 80 km/h | 0.2 s | 15 s | 76 | 281.89 | 362 | 1000.55 |

De la tabla, son solo los mensajes programados para enviar, por retrasos para obtener acceso al medio no se envían todos. Como se esperaría, al reducir el intervalo de transmisión, se tiene un pequeño aumento en la tasa de errores y en los tiempos de encolamiento, junto a un aumento en las colisiones. Esto lleva a que los mensajes recibidos aumenten en menor proporción al de nuevos mensajes transmitidos. Con el aumento de velocidad, los tiempos y tasas de error se modifican poco. Parte importante de los mensajes perdidos, no solo se dan por las colisiones, también por el aumento en la velocidad de los nodos.

Es en el intervalo de 0.2 segundos cuando se tiene grandes aumentos en el número de mensajes recibidos, en su mayoría mensajes repetidos. El acceso al medio comienza a ser problemático, lo que lleva a varios mensajes a retrasarse. Al mismo tiempo, la inundación de la red es considerable debido a que los nodos reciben muchos más mensajes que deben procesar.

Como se había adelantado en la simulación 1, tener intervalos bajos de actualización, en este caso para nosotros es informar de las detecciones de otros vehículos, dependerá como se había visto de la velocidad, pero también de la densidad de vehículos que intenten comunicarse. Con diez nodos, un intervalo de comunicación de 200 ms parece ser posible solo a altas velocidades, donde incluso presenta pequeñas mejorías. Mientras que por ejemplo a 30 km/h, el óptimo pareciera ser un intervalo entre mensajes de 1000 ms. La inundación de la red con mensajes no parece ser tan perjudicial al aumentar la velocidad en los nodos, razón de que solo haya una drástica reducción en los mensajes recibidos con 30 km/h.

### Diferentes densidades de nodos

A partir de la simulación anterior, se expande variando la cantidad de nodos, como el tamaño de mensajes. Se esperaría que, al aumentar la cantidad de nodos, el intervalo de 0.2 e incluso 0.5 segundos deje de ser viables. Igualmente, al aumentar el tamaño del mensaje, el porcentaje de recibidos caerá, y se verá especialmente afectado a mayores velocidades.

El acercamiento tomado, es que un nodo para estar notificado de la presencia de otro vehículo debe recibir cierta cantidad de mensajes. Partiendo de que cada nodo se desea que comparta sus detecciones y las posiciones predichas de ellas, en forma de coordenadas. Se toma como tamaño de mensaje 2 kB de payload. Se compara la probabilidad entonces de recibir 2 mensajes de 1 kB vs 1 mensaje de 2kB, y de recibir 4 mensajes de 1 kB vs 2 mensajes de 4kB.

El vehículo viaja a diferentes velocidades (30, 50, 80 km/h) en las que recorre un trayecto de 100 metros y 200 metros. Considerando, además, una distancia de seguridad antes de la cual deben haber recibido los mensajes y así reducir el riesgo de colisión. Para esto, se toma de base la distancia que se tiene que guardar entre dos automóviles para frenar a tiempo. Hay varias formas de calcularla, pero la usada fue la regla de 2 segundos, que consiste en multiplicar por dos la velocidad en metros por segundo. Por ejemplo: en 100 metros, un vehículo viajando a 50 km/h y con un intervalo de transmisión de 1 segundo, tendría la oportunidad de mandar 7 mensajes. Considerando una distancia de seguridad, como que el nodo debe quedar advertido aproximadamente 28 metros antes, solo tiene la oportunidad de enviar 5 mensajes.

De forma que en la simulación se obtiene el mejor intervalo para la comunicación, asumiendo que siempre hay una ruta entre los nodos. Además, en el caso de 100 metros, algunos intervalos serán insuficientes para algunas velocidades.

#### 10 nodos

Tabla ‑ Probabilidad de recibir cierto número de mensajes y los tiempos de espera promedio, al variar el tamaño del mensaje, velocidad e intervalo. en una red con 10 nodos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tamaño Payload  kB | Intervalo de transmisión (segundos) | Velocidad km/h | Porcentaje de mensajes recibidos | Probabilidad de recibir 2 mensajes | Probabilidad de recibir 4 mensajes | Tiempo promedio de encolamiento  (segundos) |
| 1 | 1 | 30 | 59.02% | 99.79% | 93.76% | 0.03 |
| 1 | 0.5 | 30 | 38.27% | 99.91% | 97.69% | 0.087 |
| 1 | 0.2 | 30 | 38.30% | 100.00% | 100.00% | 0.284 |
| 1 | 1 | 50 | 40.80% | 67.67% | 9.33% | 0.091 |
| 1 | 0.5 | 50 | 38.86% | 94.63% | 58.87% | 0.113 |
| 1 | 0.2 | 50 | 42.07% | 100.00% | 99.92% | 0.311 |
| 1 | 1 | 80 | 35.90% | 12.89% | 0% | 0.168 |
| 1 | 0.5 | 50 | 35.58% | 58.27% | 5.73% | 0.135 |
| 1 | 0.2 | 80 | 37.09% | 96.90% | 70.76% | 0.411 |
| 4 | 1 | 30 | 35.42% | 98.74% | 91.82% | 0.082 |
| 4 | 0.5 | 30 | 33.81% | 99.97% | 99.71% | 0.209 |
| 4 | 0.2 | 30 | 38.89% | 100.00% | 100.00% | 0.573 |
| 4 | 1 | 50 | 27.78% | 80.35% | 42.57% | 0.065 |
| 4 | 0.5 | 50 | 51.66% | 99.93% | 99.19% | 0.149 |
| 4 | 0.2 | 50 | 31.01% | 99.99% | 99.92% | 0.361 |
| 4 | 1 | 80 | 28.61% | 49.03% | 8.19% | 0.05 |
| 4 | 0.5 | 50 | 31.36% | 84.76% | 49.96% | 0.113 |
| 4 | 0.2 | 80 | 33.30% | 99.22% | 94.58% | 0.325 |

Figura ‑ Porcentaje de mensajes recibidos en promedio por los nodos

Los tiempos experimentan ligeros cambios al aumentar la velocidad, siendo el mayor cambio con los intervalos pequeños como 0.2 segundos. Los tiempos aumentan en los mensajes de tamaño de 1 kB, mientras que disminuyen en los de 4 kB.

Figura ‑ Tiempos de encolamiento

Con una distancia de 100 metros antes de la intersección, las siguientes gráficas muestran la probabilidad de que un nodo quede enterados de la presencia del otro:

Figura ‑ Probabilidad de recibir 2 y 4 mensajes con 10 nodos

Con 200 metros:

Figura ‑ Probabilidad de recibir 2 y 4 mensajes con 10 nodos, en un trayecto de 200 metros

#### 20 nodos

Tabla ‑ Comparación de los tiempos de encolamiento y probabilidad de recibir cierto número de mensajes, en una red con 20 vehículos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tamaño Payload  kB | Intervalo de transmisión (segundos) | Velocidad km/h | Porcentaje de mensajes recibidos | Probabilidad de recibir 2 mensajes | Probabilidad de recibir 4 mensajes | Tiempo promedio de encolamiento  (segundos) |
| 1 | 1 | 30 | 51.56% | 99.17% | 85.25% | 0.042 |
| 1 | 0.5 | 30 | 49.97% | 100.00% | 99.87% | 0.109 |
| 1 | 0.2 | 30 | ---------- | -------- | --------- | 0.345 |
| 1 | 1 | 50 | 59.96% | 91.27% | 33.63% | 0.111 |
| 1 | 0.5 | 50 | 60% | 99.83% | 94.52% | 0.243 |
| 1 | 0.2 | 50 | ---------- | -------- | ----------- | 5.98 |
| 1 | 1 | 80 | 47.28% | 22.35% | 0% | 0.076 |
| 1 | 0.5 | 80 | 47.06% | 77.36% | 15.29% | 0.258 |
| 1 | 0.2 | 80 | --------- | --------- | ----------- | 0.984 |
| 4 | 1 | 30 | 27.78% | 96.14% | 81.29% | 0.155 |
| 4 | 0.5 | 30 | 29.34% | 99.90% | 99.10% | 0.317 |
| 4 | 0.2 | 30 | ---------- | ----------- | ---------- | 4.63 |
| 4 | 1 | 50 | 22.34% | 71.75% | 31.12% | 0.133 |
| 4 | 0.5 | 50 | 27.34% | 95.90% | 80.47% | 0.375 |
| 4 | 0.2 | 50 | ---------- | ----------- | ---------- | 1.705 |
| 4 | 1 | 80 | 18.70% | 33.90% | 3.50% | 0.159 |
| 4 | 0.5 | 80 | 22.28% | 71.64% | 31.00% | 0.314 |
| 4 | 0.2 | 80 | ---------- | ---------- | ---------- | 1.486 |

Se sigue mostrando el intervalo de 0.2 segundos, únicamente para graficar los tiempos de encolamiento. Sin embargo, este intervalo ya no es viable, solo se consideran 0.5 y 1 segundo.

Figura ‑ Porcentaje de mensajes recibidos con 20 nodos

Con una distancia a la intersección de 100 metros:

Figura ‑ Probabilidad de recibir 2 y 4 mensajes con 20 nodos

Con 200 metros a la intersección:

Figura ‑ Probabilidad de recibir 2 y 4 mensajes con 20 nodos, en un trayecto de 200 metros

#### 30 nodos

Tabla ‑ Comparación de los tiempos de encolamiento y probabilidad de recibir cierto número de mensajes, en una red con 30 vehículos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tamaño Payload  kB | Intervalo de transmisión (segundos) | Velocidad km/h | Porcentaje de mensajes recibidos | Probabilidad de recibir 2 mensajes | Probabilidad de recibir 4 mensajes | Tiempo promedio de encolamiento  (segundos) |
| 1 | 1 | 30 | 43.92% | 97.28% | 71.06% | 0.271 |
| 1 | 0.5 | 30 | 47.61% | 100.00% | 99.75% | 0.495 |
| 1 | 0.2 | 30 | ----------- | ---------- | ----------- | 11.773 |
| 1 | 1 | 50 | 47.67% | 78.20% | 15.97% | 0.194 |
| 1 | 0.5 | 50 | 53.51% | 99.41% | 87.97% | 0.532 |
| 1 | 0.2 | 50 | ---------- | ----------- | ----------- | 13.23 |
| 1 | 1 | 80 | 37.63% | 14.16% | 0% | 0.369 |
| 1 | 0.5 | 80 | 51.90% | 83.53% | 21.22% | 0.734 |
| 1 | 0.2 | 80 | ---------- | ---------- | -------- | 15.4321 |
| 4 | 1 | 30 | 15.88% | 82.26% | 48.77% | 0.189 |
| 4 | 0.5 | 30 | 16.12% | 97.03% | 85.60% | 0.473 |
| 4 | 0.2 | 30 | ------------ | --------- | -------- | 13.112 |
| 4 | 1 | 50 | 22.58% | 72.19% | 31.63% | 0.221 |
| 4 | 0.5 | 50 | 17.45% | 85.30% | 54.24% | 0.509 |
| 4 | 0.2 | 50 | ----------- | -------- | -------- | 8.83 |
| 4 | 1 | 80 | 19.35% | 34.96% | 3.74% | 0.174 |
| 4 | 0.5 | 80 | 13.48% | 51.52% | 13.75% | 0.954 |
| 4 | 0.2 | 80 | ------------ | ---------- | -------- | 15.32 |

Figura ‑ Porcentaje de mensajes recibidos con 30 nodos

Con una distancia a la intersección de 100 metros:

Figura ‑ Probabilidad de recibir 2 y 4 mensajes con 30 nodos

Con 200 metros a la intersección:

Figura ‑ Probabilidad de recibir 2 y 4 mensajes con 30 nodos, en un trayecto de 200 metros

En la siguiente gráfica se muestra el incremento en el tiempo de encolamiento al agregar más vehículos a la simulación:

Figura ‑ Tiempos de encolamiento al aumentar el número de nodos

No se grafica para el intervalo de 0.2 segundos, porque a partir de 20 vehículos, los tiempos se disparan.

El vehículo tiene información suficiente para modificar su intervalo de transmisión, conoce su velocidad, y la estimada de otros nodos. Además, puede darse una idea de la cantidad de nodos en la red, mediante los mensajes recibidos. Si se agrega que conoce su ubicación, puede conocer la localización por ejemplo de cruceros, y entonces considerar su distancia a ciertos puntos ciegos.

De las mediciones observadas se puede ver que, con 10 nodos, si el mensaje es de 4 kB, el intervalo de 0.2 segundos no es viable, mientras que con mensajes de 1 kB deja de serlo a partir de 50 km/h. Con 20 nodos, el intervalo de 0.2 segundos nunca es viable. Y con 30 nodos, en ambos tipos de mensajes, el intervalo de 0.5 segundos deja de ser una opción con 80 km/h. Además, a partir de 0.5 segundos y 20 nodos, con mensajes de 4 kB se puede presentar retrasos en algunos escenarios.

A continuación, se muestran los intervalos recomendados según donde se tenga mayor probabilidad de que los nodos queden notificados, con diferentes velocidades. Se considera que deben recibir 4 mensajes de 1 kB o 2 mensajes de 2 kB.

A una velocidad de 30 km/h:

Mapa

Descripción generada automáticamenteMapa

Descripción generada automáticamente

Figura ‑ Intervalos de transmisión recomendados a una velocidad de 30 km/h

A 50 km/h:

Mapa

Descripción generada automáticamenteMapa

Descripción generada automáticamente

Figura ‑ Intervalos de transmisión recomendados a 50 km/h

A 80 km/h:

Mapa

Descripción generada automáticamenteMapa

Descripción generada automáticamente

Figura ‑ Intervalos de transmisión recomendados a 80 km/h

Los mensajes con un tamaño de paquete mayor pueden funcionar en escenarios a altas velocidades, como se muestra a 80 km/h, donde incluso funciona con 20 nodos. En ellos, en lugar de tener que acceder al medio múltiples veces en un corto intervalo (por la velocidad), resulta mejor mandar menos paquetes de tamaño más grande. Esto funciona siempre que la densidad de nodos no sea muy alta.

Se observa, sobre todo, la necesidad de un nodo para coordinar y evitar la colisión entre mensajes, igualmente, puede experimentarse juntar comunicación peer to peer con el uso de canales, para dedicarlo a la comunicación de detecciones, y dejar el broadcast para otras aplicaciones de seguridad, como el caso 1 para cambiar de carril o avisar de frenado.

# Conclusiones

El tema abordado en el trabajo fue la programación de un nodo de ROS para la detección y rastreo de vehículos, junto con la simulación de una red vanet con aplicaciones en seguridad para la identificación de parámetros y verificar su fiabilidad. En ambos casos se ha buscado marcar los pasos para la continuación en la investigación en vehículos autónomos, especialmente para el modelo de coche autónomo AutoModelcar.

Dentro de los detectores se mostró con detalle la implementación y se realizó un manual, junto con la documentación del carro, que permita avanzar en la programación de nuevos nodos que se ocupen de tareas como navegación, así mismo poder expandir el problema de detección. Desde en el enfoque que se abordó, se buscó priorizar la velocidad de los métodos usados, para que fuera posible de usar en el carro físico durante la conducción. La combinación de máquinas de soporte vectorial con filtros de cascada resultó ser efectiva en este punto, alcanzando una frecuencia de 5 Hz. Con tiempos para la detección de un frame completo de 300 a 400 ms y de solo regiones de 90-100 ms.

Sin embargo, dentro de la precisión se requiere de mayor trabajo. Para ello se debe ampliar el tamaño de los datasets para el entrenamiento, y mejorar el entrenamiento del filtro de cascada, que terminó siendo la parte más débil de la detección. Igualmente se puede replicar y ampliar para la detección de otros objetos, siguiendo los mismos pasos descritos. Dentro de las ideas aplicadas con svm, fue el uso de dos detectores para el reconocimiento tanto de coches desde atrás, como de lado. Expandiendo esto, otras posibles mejoras a futuro, es dividir la imagen y en regiones, a las que a cada una aplicar detectores diferentes. De forma que se puedan suplir algunas de las fallas encontradas en los resultados, como detectar partes del coche cuando no está completo en la imagen, sin afectar gravemente el rendimiento (en lugar de aplicar todos los detectores sobre la ventana completa). Igualmente, agregar el uso del LiDAR para elegir regiones de interés, sobre las cuales hacer la detección.

La idea de usar estos métodos, frente a otras ideas más sencillas aplicadas sobre el carro en físico. Es que, mediante la identificación del objeto y su movimiento, le permite al coche generar un contexto, reconocer el ambiente donde se mueve y modificar su comportamiento de forma más efectiva y segura. Esto funciona para el modelo de AutoModelcar, pero como se ha mencionado en el trabajo, no podrá suplir las ventajas que traen las redes neuronales, apoyadas con el crecimiento en el poder computacional.

En el campo de las redes vanet, la combinación elegida contempla el uso de DSRC para que por broadcasting los nodos informen de sus detecciones. Con lo cual cubrir puntos ciegos como cruceros, donde no se puedan valer únicamente de su información. Mediante esto podrán tomar medidas evasivas, o como se propone, usar el protocolo de AODV para la negociación y coordinación entre dos vehículos. Esto se podría lograr partiendo de información que contienen los vehículos, como es su ubicación geográfica mediante gps, su velocidad, la posición y velocidad estimada de los vehículos detectados, y por los mensajes recibidos, la densidad estimada del tráfico.

En las pruebas los resultados alcanzados alcanzan a compartir mensajes con un payload de 1 kB, con intervalos por lo general de 0.5 segundos en la mayoría de los escenarios. En algunos, incluso se puede recurrir a intervalos menores de 0.2s con 10 nodos, usar paquetes con 2 kB de payload, o en situaciones no críticas compartir con un intervalo de 1 segundo. En la mayoría de los casos, el mensaje se logra compartir con una alta probabilidad, lo que ayudaría a reducir incluso riesgo de colisiones. Las limitaciones se encuentran hasta 80 km/h donde el desempeño empeora considerablemente, y con 30 vehículos (todos tratándose de comunicar simultáneamente) a partir de los 50 km/h. Esto nos muestra que pueden ser fiables para la comunicación. Y en especial si se puede considerar que conoce su entorno, y sabe que se aproxima por ejemplo a un crucero. Los resultados pueden mejorar si considera la presencia de infraestructura para coordinar (ayudaría a reducir colisiones en las transmisiones, como retraso para acceder al medio).

La investigación por cuestiones de los amplio que es el tema y por tiempo, fue muy limitada, pero se espera con los parámetros identificados, se puede profundiazar más. En especial se requiere la simulación de más escenarios, donde considerar más velocidades (y que no todos los nodos se muevan a la misma velocidad), densidad de tráfico, niveles de ruido. Mejoras que pueden ayudar a que los resultados de las simulaciones sean más realistas, se incluye el uso de otros modelos de path loss, pues el rebote de las señales puede empeorar los resultados. Así mismo, considerar temas de seguridad.

En el caso planteado para el uso de AODV, tiene algunas complejidades extra al tratarse de un protocolo reactivo, por el que conlleva mayores retrasos. Además, de que, como se menciona se requiere otra banda para transmitir y tengan tecnología mu-mimo (múltiple usuario - múltiples entradas y salidas). Algunas soluciones que se han planteado especialistas para reducir la inundación en AODV, es el uso de canales (cada nodo requerirá tener un canal por defecto, quien quiera transmitir a él debe descubrir cuál es, una forma sería enviando por todos los canales). DSRC puede ayudar en esta tarea, agregando en la información compartida, el canal correspondiente a cada vehículo, así mismo, podría servir para formar tablas de ruteo (si se extiende por medio de una aplicación para cubrir la capa 3), pese a que no están diseñados para trabajar juntos.

Por último, se planteó cómo podría implementarse en el modelo de carro físico. Para ello se requiere de hardware para la conexión a redes inalámbricas. En AODV, se necesita de un módulo transceptor inalámbrico que opere en la frecuencia deseada, como 5.9 GHz (aquí se tiene mayor flexibilidad), El protocolo se puede implementar de forma directa en Linux, y hay opciones open source como AODV-UU. Para DSRC se necesita específicamente de un módulo compatible con IEEE 802.11p. Igualmente, se requiere de un receptor gps y software, para Linux se tiene gpsd Daemon, que comparte las mediciones con todas las aplicaciones. Para su uso con Ros, a partir de sus versiones melodic y noetic, contiene la librería gpsd\_client.

Juntar DSRC con ROS de forma directa es una tarea más difícil, que requiere la implementación de una aplicación que extienda la capa 3. Pues DSRC no usa direcciones IP y ROS no considera el uso de IP dinámicas. Además, sería necesario el uso de librerías como ROS\_multimaster. En este punto, para compartir las detecciones, sería mejor no usar ros para publicar y suscribirse, ni considerar cada vehículo como un nodo de ros. En su lugar, manejar DSRC para la comunicación de los nodos y la información recibida leerla con un nodo de ROS (en lugar de intentar conectarse a otro vehículo, para suscribirse a sus mensajes).

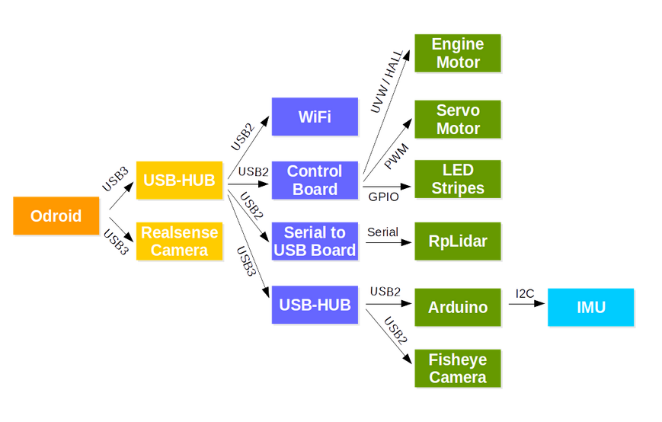
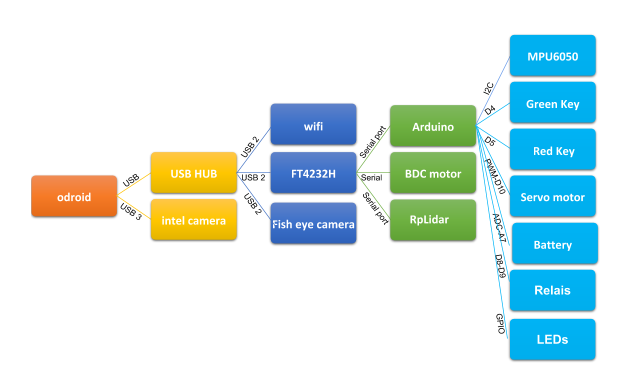
# Apéndice 1: Documentación

## Documentación AutoModelCar

AutoModelCar

Consiste en vehículos autónomos a escala 1/10 desarrollado*s* enFreie Universität Berlin para propósitos de educación, en el ITAM se cuenta con 2 vehículos uno corresponde al modelo 1 y el otro al 2. En la figura 10-1 se presenta la arquitectura de ambos.

Figura ‑ A la izquierda módulos y conexiones del modelo 1, a la derecha el modelo 2. Fuente: (“hardware (autonomos model v2) · automodelcar/automodelcarwiki wiki · github” s. f.)



Inicio

Pulse el botón verde durante 3 segundos para que encienda.

**Conexión a la odroid**

Para la conexión al robot se puede realizar conectándose a través de la red del ITAM, tanto por ethernet como wifi, mediante una terminal se debe escribir:

Carro 1

ssh [root@148.205.37.48](mailto:root@148.205.37.48)

ssh [root@10.3.79.41](mailto:root@10.3.79.41) (Wifi) se conecta a la red WIFI\_ITAM

ssh [root@10.10.0.240](mailto:root@10.10.0.240) (Wifi) se conecta a la red ITAM

Carro 2

ssh [root@10.10.0.250](mailto:root@10.10.0.250) ITAM

ssh [root@192.168.1.199](mailto:root@192.168.1.199) conexión directa

Password: elfmeter

El nombre del robot 1 es Robot8, del robot 2 es Modelcar-126.

Para ejecutar el programa principal se debe pulsar el botón verde nuevamente o escribir ./autostart.sh (se debe activar los motores para mandarle velocidades con auto\_stop)

**Instalación**

Para instalar los repositorios del carro autónomo se deben bajar de GitHub:

$ git clone https://github.com/AutoModelCar/model\_car.git

Si se requiere otra versión en concreto, se debe buscar el commit hash correspondiente a la versión deseada. Por ejemplo, para la versión 1 sobre la que se trabaja:

$ git checkout fed4cef394e412ec1f3aaa51586a6eb7689b4cc7

Los paquetes se pueden construir con dos herramientas:

* Catkin\_build

Corresponde a los pasos de instalación indicados en la wiki del AutoModel:

$ cd model\_car

$ ./copy.sh

¡Importante se debe cambiar la dirección por la del carro!

Copiará archivos como el bashrc y el autostart a la raíz, extrae los archivos set y set2.sh

Actualizar la fecha

$ date --set "yyyy-mm-dd hh:mm:ss"

$ ./set.sh

Reiniciar el carro

$ ./set2.sh

Reiniciar nuevamente el carro

* Catkin make

Otra alternativa que es la usada es mediante catkin make, para ello se debe primero crear un nuevo catkin workspace:

$ mkdir -p ~/catkin\_ws/src

$ cd ~/catkin\_ws/

Se descarga el programa en otra carpeta

$ git clone https://github.com/AutoModelCar/model\_car.git

$ git checkout fed4cef394e412ec1f3aaa51586a6eb7689b4cc7

Sobre el directorio descargado, se copia únicamente la carpeta src dentro del catkin\_ws creado:

$cp -r src /path donde se creó el nuevo workspace

$ catkin\_make

Por un lado, catkin make es una herramienta para compilar código en un workspace de catkin, equivale a la unión de los comandos cmake y make, y debe ser llamado siempre desde la raíz. En el proceso se agregan los directorios build y devel, junto con los archivos CMakeLists, los cuales preparan y ejecutan el proceso de compilación.

Por otro lado, catkin build se comporta como catkin\_make\_isolated, el cual aísla cada paquete en source y los construye de manera separada. De tal manera que no hay un archivo CMakeLists en el nivel más alto, sino que cada paquete es independiente y, por lo tanto, se puede llamar catkin build desde cualquier directorio.

Sin embargo, se debe tener cuidado con la información que se pasan entre paquetes en los archivos CMake, pues estos generan las dependencias entre todos los paquetes.

Catkin\_build y catkin\_make son incompatibles entre sí, usar uno u otro depende de cómo se construyó el repositorio originalmente. Debido a que se cuentan con dos repositorios en el vehículo, el original catkin\_ws y sobre el que se trabaja en workspaceAutonomos\catkin\_ws1\., ambos están construidos con catkin\_make. Sin embargo, hay algunas configuraciones dentro de opt/ros/indigo que catkin\_make no realiza por lo que se deben reemplazar manualmente, para poder operar la cámara de profundidad. Aun así, el uso de catkin\_build ya no es recomendado por ROS y ya no recibe mantenimiento.

Si se quiere cambiar el repositorio en el que se trabaja se puede hacer modificando el .bashrc que se encuentra en raíz, hasta abajo se debe agregar la dirección del que se desea usar. Es posible que se creen algunas dependencias entre repositorios si se usa catkin\_build, el cual se puede usar para correr los nodos base, pero no se pueden compilar nuevos programas por incompatibilidad con los otros workspaces.

**Correcciones**

Las siguientes modificaciones solo se hacen sobre la versión 1 del robot, y corresponden a cambios en algunos archivos launch relacionados con los nombres de los tópicos que utiliza:

Dentro de launch random movement

* nano random\_movement/launch/auto.launch

Reemplazar:

<include file="$(find light)/launch/light.launch"/>

<include file="$(find

send\_steering)/launch/servo\_odroid.launch"/>

Por:

<include file="$(find

send\_steering\_light)/launch/servo\_light\_odroid.launch"/>

* nano manual\_control/launch/manual\_PC.launch

Reemplazar:

<include file="$(find

send\_steering)/launch/servo\_odroid.launch"/>

Por:

<include file="$(find

send\_steering\_light)/launch/servo\_light\_odroid.launch"/>

Como se mencionó anteriormente, debido a que los paquetes fueron construidos con catkin\_make, es posible que algunas variables no se establezcan correctamente, por lo que, para poder usar la cámara de profundidad, es necesario usar el nodelet que se construye con catkin\_build. Lo que se hizo fue copiar las librerías de la versión 2 del vehículo:

Se copio librealsense del directorio raíz y el directorio /opt/ros/indigo/ completo. El problema se da en algunas configuraciones que realiza catkin\_build a la hora de construir los paquetes que catkin\_make no realiza, por lo que el programa de nodelet\_manager no puede levantar todos los nodos y entonces la cámara SR300 no logra funcionar.

Dentro de workspaceAutonomos se tienen los directorios lib y lib2, que corresponden a la librería librealsense que traía originalmente el carro y la de la versión 2 respectivamente. Se tienen también op y op2, de igual forma op corresponde al directorio original y op2 al de la versión 2 por la que se sustituye.

Para copiar las librerías se utilizan los comandos:

cp -r workspaceAutonomos/lib2/ /root/

cp -r workspaceAutonomos/op2/indigo/ /opt/ros/

Las librerías originales lir y op se guardaron dentro de workspaceAutonomos.

Cada que se utiliza catkin\_make para compilar todos los paquetes completos, hay que remplazar nuevamente la librería de opt/ros/indigo. Una solución más cómoda es usar Catkin\_make solo la primera vez que se agrega un nuevo paquete, después se puede usar:

catkin\_make --only-pkg-with-deps prueba

Lo que evita tener que copiar nuevamente las librerías. Igualmente, como se mencionó en la guía algunos paquetes fallan al compilar con cmake, lo que interrumpe su operación, por lo que se debe volver a correr hasta que no haya problemas para que agregue todos los paquetes. De lo contrario, habrá algunos nodos que no podrá encontrar al correr el programa principal.

En el caso específico del carro puede tener problemas con el reloj que impiden compilar “clock skew detected”. Para corregirlo se puede actualizar las marcas de tiempo con:

find . -exec touch {} \;

Desde las direcciones /root/workspace/catkin\_ws1 y /opt/ros/

Otro problema que puede aparecer es el error: c++: internal compiler error: Killed (program cc1plus), que normalmente ocurre porque se queda sin memoria. Como solución se agrega el parámetro -j, que limita el número de trabajos a costa de aumentar el tiempo de la compilación:

catkin\_make -j 2

**Configuración**

Se tienen 4 archivos principales que puede requerirse modificar para el uso del carrito físico, los mostrados corresponden a la versión 1 del carro, para la versión 2 es similar, con pequeños cambios como en algunas direcciones.

**Autostart.sh**

----------------------------------------------

#!/bin/bash

modprobe uvcvideo

# clear ROS log

rm -rf /root/.ros/log

# load environment settings from .bashrc

PS1='$ '

source /root/.bashrc

### Start ROS core and wait a few seconds

roscore &

sleep 5

### Start ROS launch scripts

#timeout 10 roslaunch realsense\_camera realsense\_sr300.launch

roslaunch manual\_control manual\_odroid.launch #|| exit

#roslaunch random\_movement auto.launch #|| exit

El autostart es un script que nos permite iniciar el programa elegido por defecto, lo que nos ahorra hacer algunos comandos para iniciar todos los programas. En él se hace la llamada al achivo .bashrc para realizar los exports y el source al workspace, aquí se inicia el roscore y se llama al programa principal con roslaunch que por defecto es el manual\_control. Es posible dentro del archivo mandar a llamar múltiples archivos launch simultáneamente, como son los launch para la cámara de ojo de pez, (sin embargo, el autor de los repositorios originales no recomienda usar las dos cámaras simultáneamente por incompatibilidades).

----------------------------------------------

**.bashrc**

----------------------------------------------

if [ -f /etc/bash\_completion ] && ! shopt -oq posix; then

. /etc/bash\_completion

fi

# configure bash history

HISTSIZE=100 # save only up to 100 commands

HISTCONTROL=ignoreboth # ignore duplicate commands

shopt -s histappend # append to history file instead of

overwriting it

export PROMPT\_COMMAND="history -a; $PROMPT\_COMMAND"

# Some applications read the EDITOR variable to determine your

favourite text

# editor. So uncomment the line below and enter the editor of

your choice :-)

export EDITOR=/usr/bin/joe

test -s ~/.alias && . ~/.alias

# Set shell prompt

NOCOLOR="\[\033[0;0m\]"

MAGENTA="\[\033[0;35m\]"

GREEN="\[\033[0;32m\]"

RED="\[\033[0;33m\]"

if test "$UID" = 0 ; then

LOGINCOLOR=$RED

else

LOGINCOLOR=$GREEN

fi

PS1="$RED# $NOCOLOR\$(date +%H:%M:%S)

$LOGINCOLOR\h:\w>$NOCOLOR "

export PS1

# Set path

export PATH=$PATH:/root:/sbin:/usr/sbin:/usr/local/sbin

# Look for libraries in local folder as well

export LD\_LIBRARY\_PATH=${LD\_LIBRARY\_PATH:+$LD\_LIBRARY\_PATH:}.

export ROS\_IP=148.205.37.48

export ROS\_HOSTNAME=148.205.37.48

export ROS\_MASTER\_URI=http://148.205.37.48:11311

source /opt/ros/indigo/setup.bash

#source /root/catkin\_ws/devel/setup.bash

source workspaceAutonomos/catkin\_ws1/devel/setup.bash

El bashrc es otro script que realiza una serie de configuraciones sobre las variables tanto del Shell como del entorno, como definir algunos colores y alias, así como definir el tamaño del historial. Sin embargo, lo que nos importan son los comandos:

* Ros IP y Hostname que definen la dirección de algún nodo que se quiera comunicar con el robot.
* Ros Master URI define la dirección del maestro que corresponde a la odroid, de donde se ejecutan los nodos.
* El bashrc realiza los export, realiza el source al setup.bash tanto del workspace activo como a la versión de ROS instalada. Llama, además, a exportar el path y las librerías, que definen en qué orden buscar los paquetes:

PATH=/opt/ros/indigo/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/root:/sbin:/usr/sbin:/usr/local/sbin

* Por último, carga con source el repositorio a utilizar. Source es un comando que ejecuta los archivos pasados como argumento sobre el mismo shell.

Para el caso del proyecto el repositorio que se usa es el que se encuentra ubicado en source workspaceAutonomos/catkin\_ws1/

----------------------------------------------

**interfaces**

Permite definir direcciones de red tanto cableada como inalámbrica, ubicado en /etc/network/interfaces

----------------------------------------------

# /etc/network/interfaces -- configuration file for ifup(8), ifdown(8)

# The loopback interface

auto lo

iface lo inet loopback

# Wired interfaces

auto eth0

iface eth0 inet static

address 148.205.37.48

netmask 255.255.252.0

gateway 148.205.36.2

broadcast 148.205.39.255

dns-nameservers 148.205.228.11

#

# Wireless interfaces

#

iface wlan0 inet static

address 10.10.0.240

netmask 255.255.224.0

broadcast 10.10.31.255

gateway 10.10.10.1

wpa-conf /etc/wpa\_supplicant.conf

wpa-driver wext

down killall wpa\_supplicant

El archivo interfaces permite definir los nombres lógicos para interfaces de redes y su configuración. Aquí se tienen definidas la dirección de red ethernet del carro como: 148.205.37.48, y la dirección inalámbrica: 10.10.0.240.

----------------------------------------------

**wpa\_supplicant**

Ubicado en /etc/wpa\_supplicant.conf

----------------------------------------------

#network={

# ssid="EK"

# key\_mgmt=WPA-PSK

# psk="robotica2016"

# Unencrypted

network={

ssid="ITAM"

key\_mgmt=NONE

}

WPA supplicant es un archivo de configuración para implementar WPA u otros protocolos de seguridad que usan las redes Wifi. El robot (cliente) debe realizar la negociación con el autenticador WPA, en el que indica el nombre de la red a la que debe conectarse, contraseña y tipo de clave.

**Conexión a la aplicación**

*https://github.com/AutoModelCar/AutoModelCarWiki/wiki/Android-APP*

En la wiki se encuentra una aplicación para controlar el carro a control remoto, permite mandar la velocidad, el ángulo de giro y un botón de paro de emergencia. Únicamente en Master\_URI se debe ingresar la dirección IP de la interfaz wlan0 a la que está conectado.

*http://10.10.0.240:11311/*

Posiblemente por problemas de la versión y el formato de los mensajes, solo es posible mandar la velocidad. Pues como se ve en las correcciones en los launch, se realizan modificaciones sobre el tipo de mensaje del nodo steering.

**Conexión a la computadora**

$ export ROS\_IP=148.205.37.41 %Dirección de la computadora

$ export ROS\_HOSTNAME=148.205.37.41 %Dirección de la

computadora

$ export ROS\_MASTER\_URI=http://148.205.37.48:11311/ %Dirección

del robot

Permite enviar comandos de forma remota o visualizar datos:

Ros\_ip y ros\_hostname basta con usar solo uno de los dos. Ambas se excluyen mutuamente y de usarse las dos, se tomará hostname.

No es necesario que la versión de ROS de la computadora sea la misma que la del carro.

Para pasar datos de una computadora al vehículo:

scp /home/ubuntu/myfile

username@IP\_of\_windows\_machine:/C:/Users/Name/Desktop

scp -r

ifons@192.168.1.115:/C:/Users/ifons/source/repos/detec2/dete

c2/pruebas/my\_hogE10000.xml

/root/workspace/catkin\_ws1/src/vision\_camara/src/

**Visualización**

$ rqt\_graph

Es una interfaz gráfica de ROS que nos permite visualizar en forma de grafo los nodos y tópicos que se están ejecutando.

$ rviz

Es una herramienta de visualización en 2D y 3D que despliega las lecturas de los sensores como el lidar y las cámaras.

$ gazebo

Es un simulador de robótica de código abierto en 3D, que permite la manipulación de sensores y actuadores, junto a un motor de físicas, que le permite modelar la forma en que un robot interactúa con el ambiente simulado.

**Nodos y programas**

Programas principales:

*random\_movement*

Manda algunos valores aleatorios de velocidad y giro.

*manual\_control*

Recibe la velocidad y steering para el carro, publica la información de la cámara de profundidad SR300 y el LiDAR.

**Actuadores**

Speed

rostopic -r 10 pub /manual\_control/speed std\_msgs /Int16

{“data: -1000”}

Tipo: std\_msgs/Int16

Campos:

int16 data

Valores hasta -10,000, basta con -1,000

Para avanzar dar los valores en negativo

-r 10 es la taza en Hertz a la que se publica el mensaje

Steering

rostopic pub /manual\_control/steering std\_msgs /Int16 {“data: 180”}

Tipo: std\_msgs/Int16

Campos:

int16 data

Valores entre 0-180

0 gira a la derecha

180 a la izquierda

90 centra las llantas

Stop start

rostopic pub /manual\_control/stop\_start std\_msgs /Int16 {“data: 1”}

Tipo: std\_msgs/Int16

Campos:

int16 data

Valores entre 0 y 1

1 para encender los motores

0 para apagarlos, se usa como paro de emergencia

Lights

rostopic pub /manual\_control/lights std\_msgs /String {“data: le”}

Tipo: std\_msgs/Int16

Campos:

string data

Enciende los leds con los que cuenta el vehículo.

string Luz

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

le Parpadeo de la luz amarilla izquierda

ri Parpadeo de la luz amarilla derecha

stop Enciende los 3 leds rojos de atrás (freno)

pa Enciende 2 leds rojos de atrás, y 2 blancos frontales de la izquierda

ta Enciende 2 leds rojos de atrás, y 2 blancos frontales de la derecha

re Enciende las luces blancas traseras, para cuando va de reversa

fr Enciende las luces blancas delanteras, para cuando se conduce en la

oscuridad

diL Apaga las luces

**Sensores**

LiDAR

Tópico:

/scan

Tipo: sensor\_msgs/LaserScan

Campos:

std\_msgs/msg/Header header

float angle\_min

float angle\_max

float angle\_increment

float time\_increment

float scan\_time

float range\_min

float range\_max

float[] ranges

float[] intensities

Devuelve la información del LiDAR, el cual hace un recorrido de 360° y tiene un alcance máximo de 8 metros, devuelve un arreglo con una medida para cada ángulo. Es el arreglo de ranges el que contiene las distancias en metros.

Cámara de profundidad

Tópicos:

/camera/color/image\_raw (sensor\_msgs/Image)

/camera/color/camera\_info

camera/depth/image\_raw

camera/depth/camera\_info

camera/depth/points

camera/infrared1/image\_raw

camera/infrared1/camera\_info

camera/infrared2/image\_raw

camera/infrared2/camera\_info

Los mensajes tipo raw corresponden al formato sensor\_msgs/Image, los tipos info son datos con los que están calibradas. Mientras que depth/points usa el mensaje tipo PointCloud2 e incorpora los datos de las 3 cámaras.

Tópico depth/points

Tipo sensor\_msgs/PointCloud2

Campos:

std\_msgs/msg/Header header

uint32 height

uint32 width

sensor\_msgs/msg/PointField[] fields

boolean is\_bigendian

uint32 point\_step

uint32 row\_step

uint8[] data

boolean is\_dense

El campo fields define el formato con el que se representan los datos, que en este caso es float32, mientras los datos vienen en data. Este tópico no se publica en el simulador.

Tópico: /app/camera/rgb/image\_raw

Tipo /sensor\_msgs/Image

Campos:

std\_msgs/msg/Header header

uint32 height

uint32 width

string encoding

uint8 is\_bigendian

uint32 step

uint8[] data

Cámara ojo de pez

No está incluido en el launch de manual\_control, se debe ejecutar de forma independiente con:

roslaunch fisheye\_camera\_matrix camera\_matrix\_publisher.launch

Se puede usar para navegación mediante marcas en el techo, con las cuales obtener la pose del carro.

**Otros**

Tópico /odom

Tipo: nav\_msgs

Campos:

std\_msgs/msg/Header header

string child\_frame\_id

geometry\_msgs/msg/PoseWithCovariance pose

geometry\_msgs/msg/TwistWithCovariance twist

geometry\_msgs/PoseWithCovariance:

geometry\_msgs/msg/Pose pose

double[36] covariance

geometry\_msgs/Pose:

geometry\_msgs/msg/Point position

geometry\_msgs/msg/Quaternion orientation

geometry\_msgs/Point:

double x

double y

double z

geometry\_msgs/Quaternion:

double x=0.0

double y=0.0

double z=0.0

double w=1.0

geometry\_msgs/TwistWithCovariance:

geometry\_msgs/msg/Twist twist

double[36] covariance

geometry\_msgs/Twist:

geometry\_msgs/msg/Vector3 linear

geometry\_msgs/msg/Vector3 angular

geometry\_msgs/Vector3:

double x

double y

double z

Tópico /Pose

Tipo: geometry\_msgs/Pose2D

Campos:

float64 x

float64 y

float64 theta

Solo en el simulador.

**Mensajes**

Ros funciona a través de nodos o programas independientes que corren simultáneamente y se comunican entre sí, para hacerlo emplean un mecanismo de publicadores y suscriptores. Un nodo que quiere escuchar determinado mensaje debe suscribirse al tópico correspondiente, los cuales se pueden ver con:

rostopic list

rosnode list muestra los nodos que se están ejecutando.

Igualmente se puede obtener información adicional, como la estructura y tipo de los mensajes con:

rostopic info y rosmsg info

Se pueden mostrar los mensajes publicados con:

rostopic echo

Para la comunicación con mensajes se pueden definir nuevos, o usar librerías que ya tienen definidos algunos formatos como:

Std\_msgs: Tipos primitivos de mensajes y arreglos, se usan para mandar la velocidad y el steering con tipo Int16

Sensor\_msgs: incluye mensajes tipo PointCloud2 para la cámara y LaserScan para el LiDAR.

Nav\_msgs: mensajes para mapas y odometría.

Geometry\_msgs: mensajes de tipo pose, transformadas, puntos y vectores.

Existe otro mecanismo que es el de servicios, sin embargo, para el caso del carro autónomo no se utiliza.

**Nodelets**

Son un tipo de nodo de ROS que permite correr múltiples nodos en un solo proceso, lo que permite se comuniquen entre sí de forma eficiente. En el carro autónomo se utilizan por ejemplo para la cámara de profundidad.

Es importante para la comunicación entre nodos que haya un master, el cual se ejecuta mediante:

roscore

El cual ya es llamado de forma automática con autostart.sh

Los nodos se pueden arrancar de forma individual con:

rosrun package-name executable-name

O incluirlos dentro de un launch

**Simulador**

<https://github.com/ITAM-Robotica/Eagle_Knights-Wiki/wiki>

Si gazebo truena al correr:

roslaunch autonomos\_gazebo\_simulation empty\_autonomos.launch

usar:

killall gzserver

killall gzclient

Es posible que hagan falta instalar algunas librerías como ignition fuel, msgs y transport:

sudo apt install libignition-msgs-dev

sudo apt-get install libignition-transport4-dev

Se diseño un control sencillo de posición. El simulador ya cuenta con un tópico que da la posición del robot en 2D llamado Pose2D, el cual da x, y, theta. Datos que se pueden obtener idealmente de una cámara que visualice al robot, u otro tipo de nodo como GPS, o usando navegación con la cámara ojo de pescado y marcas en el techo.

El simulador ya contiene algunos mundos para hacer pruebas con el carro. Estos se encuentran en la carpeta de autonomos\_gazebo\_simulation, donde al ejecutarse los launch, lanzan también el programa de manual\_control para el carro. Dentro de la carpeta, se encuentran tres folders. En launch son los archivos ejecutables, dentro de ellos se llama siempre a empty\_world.launch y el .world respectivo. En models contienen los archivos para simular el carro y sus módulos, y las secciones de pista. Por último, en worlds a partr de los modelos, se crean las pistas y se agrega el modelo del carro.

Por la versión de ros se requiere hacer unos cambios para que aparezca la pista completa, de lo contrario únicamente se agrega la primera ocurrencia de un modelo. Para ello, a cada pedazo de pista se debe asignar un identificador con <name>a1</name>.

<include>

<uri>model://straight\_road</uri>

<name>a13</name>

<pose>.5 -2.5 0.1 0 0 0 </pose>

</include>

El procedimiento sería similar para agregar un segundo vehículo:

<include>

<uri>model://AutoNOMOS\_mini2</uri>

<name>a1</name>

<pose>1.75 0 0.16 0 0 1.5707 </pose>

</include>

Pero, como se observa, el modelo para el segundo vehículo se llama AutoNOMOS\_mini2. Esto se debe a que los módulos de la cámara y el laser son independientes al carro, y no se duplicarían. Como solución, en models se copian los archivos de AutoNOMOS\_mini, y en model.sdf se deben asignar otros nombres a los tópicos.

Para el LiDAR, en la línea 517:

<topicName>/scan2</topicName>

Para la cámara, a partir de la línea 590:

<cameraName>app2</cameraName>

<imageTopicName>camera2/rgb/image\_raw</imageTopicName>

<cameraInfoTopicName>camera2/rgb/camera\_info</cameraInfoTopicName>

<depthImageTopicName>camera2/depth/image\_raw</depthImageTopicName>

<depthImageCameraInfoTopicName>camera2/depth/camera\_info</depthImageCameraInfoTopicName>

<pointCloudTopicName>camera2/points</pointCloudTopicName>

Esto modifica los tópicos. Para la cámara y laser se usan los nombres definidos anteriormente. Para el resto de tópicos, se añdae el identificador dado al modelo del carro en el archivo .world, por ejemplo:

/a1/manual\_control/speed

**Odometría**

Odometry está programado para resetearse cada x tiempo. Mide el ángulo con un acelerómetro, por las vibraciones del robot produce lecturas incorrectas, por lo que no es recomendable usar el control con odometría.

/odom

Mensaje tipo nav\_msgs/odometry. Publica la pose, orientación, giro linear y angular, al momento de iniciar el carro establece como punto 0,0,0. Si se desplaza comienza a contar en x, según el modelo ackerman y de bicicleta. No registra cuando se mueve de reversa

/tf

Transforma odometria al base\_link. Tanto odom como tf tienen los mismos valores, porque no hay ninguna rotación ni desplazamiento entre los frames base\_link y odom.

## Nodos Programados

Dentro de /catkin\_ws/src

catkin\_create\_pkg nombre roscpp geometry\_msgs sensor\_msgs std\_msgs

Agregar dentro de la carpeta en source los nodos

En CMakeLists buscar la parte para ejecutar nodos C++ y agregar:

add\_executable(nombre src/nombre.cpp)

target\_link\_libraries(nombre ${catkin\_LIBRARIES})

Si faltara algún paquete, en Package.xml:

<build\_depend>package</build\_depend>  
<run\_depend>package</run\_depend>

Y en CMakeLists:

|  |
| --- |
| catkin\_package( |
|  | CATKIN\_DEPENDS |
|  | nav\_msgs |
|  | std\_msgs) |

Posteriormente, sobre catkin\_ws ejecutar:

catkin\_make

Una vez ejecutado catkin\_make puede usar:

catkin\_make --only-pkg-with-deps nombre

En el caso de Python únicamente requiere hacerlo ejecutable con chmod y correr:

source devel/setup.bash

Para correr los nodos debe ejecutar autosart y con rosrun ejecutar cada uno de los nodos.

Se presentan los nodos programados que se pueden encontrar en el vehículo dentro del paquete vision\_camara y en [Github](https://github.com/ifonsecaz/Automodelcar-tests): corresponden a programas tanto del proyecto para la detección y rastreo, como ejemplos para usar la cámara y transformadas.

* validacion.cpp

Programa para entrenar las máquinas de soporte vectorial, hacer la validación con un set de imágenes y poder visualizar los elementos extraídos por HOG.

Paquetes: Hace uso de HOGImage y OpenCV

* detecRastreoSim.cpp

Programa que funciona sobre el simulador del vehículo autónomo, realiza la detección y rastreo de vehículos en imágenes proporcionadas por la cámara realsense y hace la publicación de las detecciones. El simulador corre en cualquier versión de ROS, en este caso se usó Noetic, y la versión de OpenCV es la 4.0.

Suscribe: /app/camera/rgb/image\_raw

Publica: /detec, /pred y /detecciones

Los primeros publican imágenes con recuadros para marcar las detecciones y la siguiente predicción de Kalman, respectivamente. El último, usa un mensaje personalizado, para comunicar todas las detecciones y sus 10 siguientes predicciones de cada una.

Paquetes: Se usa OpenCV4, cv\_bridge, sensor\_msgs,

Se requieren importar los paquetes al programa, e incluirlos como se menciona en el tutorial de ROS a los archivos CMakeLists y package.xml, todos los nodos requieren importar también roscpp.

* detecRastreoC.cpp

Programa que funciona en el vehículo, igualmente hace la detección y rastreo, en este caso el carro tiene ROS Indigo, y la versión de OpenCV es la 2.4.

Suscribe: /app/camera/rgb/image\_raw

Publica: /detec, /pred y /detecciones

Paquetes: OpenCV2.4, sensor\_msgs, cv\_bridge

* detecRastreo.cpp

Este programa no es de ROS, se usa para las pruebas pasándole una carpeta con imágenes tomadas por una cámara y devuelve otras dos carpetas de imágenes con las detecciones y las predicciones.

Paquetes: OpenCV4

* validKalman.cpp

Permite realizar pruebas para ajustar el filtro de Kalman.

# Apéndice 2: Tutorial AutoModelCar

En este capítulo se presenta una guía para el uso de ROS y OpenCV enfocado a la programación del vehículo. Para comenzar hablemos de ROS o “Robotic Operating System”, el cual se trata de un *middleware* que *“*provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots. ROS provee abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes, entre otras cosas, siendo *open source”* (“es - ROS Wiki” s. f.). La plataforma principal para usar ROS es en distribuciones de Linux como Ubuntu, y permite la programación en C++ y Python.

## Guía de ROS

Tomando de base el libro “A gentle introduction to ROS” de Jason M. O’Kane (O’Kane 2014). A continuación, se realiza una pequeña guía introductoria de ROS con los comandos principales, así como la descripción y estructura para lograr una mejor comprensión del programa.

**Librerías**

Para empezar, es importante mencionar que los paquetes instalados por *apt-get* para ros y otros paquetes externos como los *nodelet* de la cámara, tienen su raíz en */opt/ros/indigo*. Los ejecutables se almacenan en el subdirectorio *lib*, al igual que los archivos *include*. Cuando los necesita, ROS encuentra estos archivos buscándolos en los directorios listados en la variable de entorno *CMAKE\_PREFIX\_PATH*, que se establece en el *setup.bash*. Para ver los archivos en un directorio se puede usar:

Rosls nombre\_del\_paquete

**Catkin workspace**

Dentro de nuestro ambiente se tiene un *workspace* o un folder que permite modificar, construir e instalar paquetes de *catkin* (“catkin/workspaces - ROS Wiki” s. f.). Si bien no es requerido, facilita la organización. Dentro del folder normalmente se tienen las carpetas de *build*, *devel* y *src*. En esta última se agregan nuestros proyectos, estos paquetes *catkin* son los que podemos ejecutar como nodos.

**Nodos**

Uno de los objetivos básicos de ROS es permitir el diseño de software como una colección de pequeños programas, en su mayoría independientes, llamados nodos, que se ejecutan todos al mismo tiempo. Para que esto funcione, esos nodos deben poder comunicarse entre sí. La parte de ROS que facilita esta comunicación se llama ROS master y, para iniciar el nodo maestro, se utiliza:

roscore

Mientras que los nodos se pueden arrancar de forma individual con:

rosrun nombre\_del\_paquete nombre\_del\_ejecutable

Como para el programa hecho:

rosrun vision\_camara kalman

Los nombres de los nodos no son necesariamente los mismos que los nombres de los ejecutables de esos nodos. Se puede establecer explícitamente el nombre de un nodo como parte del *rosrun command*:

rosrun nombre\_del\_paquete nombre\_del\_ejecutable

\_\_name:=nombre\_del\_nodo

Esto es útil si se quieren correr múltiples nodos del mismo tipo. Igualmente se pueden arrancar múltiples nodos de forma simultánea mediante un *launch, como en la figura 11-1*. En el vehículo se tiene el programa *autostart* que se encarga ya de correr el *roscore*, llamar a las variables de entorno y correr un *roslaunch* con todos los nodos necesarios para manejar el vehículo, llamado *manual\_control*:

roslaunch manual\_control manual\_odroid.launch

Texto

Descripción generada automáticamente

Figura ‑ Ejemplo de roslaunch correspondiente a manual\_control

Dicho *launch* corre los nodos que se encargan de controlar los motores, la comunicación con el Arduino, los nodos para mandar la velocidad y el *steering*, así como el LiDAR, las cámaras, entre otros. Como se observa en la imagen, llama a buscar los paquetes dentro de la variable de *path*, y arranca otros archivos .*launch*, para llamar a los nodos.

Para llamar a algún nodo, dentro de un archivo *launch* se escribe:

<node pkg=”nombre\_del\_paquete” type=”nombre\_del\_nodo”

name=”nombre\_al\_ejecutarse”> </node>

Por ejemplo, para el nodo auto\_stop:

<node pkg=”auto\_stop” type=”auto\_stop\_node" name="auto\_stop"

output="screen">

<param name="angle\_front" type="int" value="35" />

<param name="angle\_back" type="int" value="35"/>

<param name="break\_distance" type="double" value="0.6"/>

</node>

**Tópicos y mensajes**

Al tener ROS un enfoque distribuido, los diferentes nodos deben comunicarse con otros compartiendo mensajes bajo un mismo contexto llamado tópico. Para hacerlo cada nodo debe suscribirse y publicar los mensajes que le interesan, esta información se puede ver con el comando:

rosnode info nombre\_del\_nodo

Para ver los nodos ejecutándose, como en la figura 11-2, se usa el comando:

rostopic list

Texto

Descripción generada automáticamente

Figura ‑ Se muestran algunos tópicos relacionados a la cámara del carro, al correr manual\_control

O se puede ver información específica de los mensajes, como en la figura 11-3, con *rostopic*. El comando *echo* que permite ver los mensajes reales que se están publicando sobre un solo tópico:

rostopic echo nombre\_del\_tópico

Texto

Descripción generada automáticamente

Figura ‑ Mensaje publicado en el tópico /scan relacionado al lidar

Este comando descargará todos los mensajes publicados sobre la terminal.

Para desplegar más información sobre un tópico se puede usar el comando *info*. El cual devuelve el tipo de mensaje, quiénes son sus publicadores y quiénes son los suscriptores, así como la frecuencia a la que se publican, un ejemplo se muestra en la figura 11-4:

rostopic info nombre\_del\_tópico

Texto

Descripción generada automáticamente

Figura ‑ El tópico /app/camera/rgb/image\_raw corresponde a la imagen a color publicada por el nodo de la cámara, se lee para la detección

Este comando, además, nos permite publicar de forma directa a algún tópico, aunque lo ideal sería realizarlo dentro de un nodo.

rostopic pub frecuencia\_en\_hz nombre\_del\_topico tipo\_de\_mensaje

{contenido: }

El cual repite el mensaje a una frecuencia dada. Por ejemplo, se puede mandar la velocidad al carro con el mensaje:

rostopic -r 10 pub /manual\_control/speed std\_msgs /Int16

{“data: -1000”}

En ROS los mensajes publicados cuentan con un formato con el que se deben escribir, lo que permite una mayor estandarización. Uno puede definir nuevos tipos de mensajes o aprovechar de las librerías que tienen tipos de mensajes ya definidos. Se puede visualizar el formato del mensaje con el comando:

rosmsg show nombre\_del\_mensaje

Los mensajes típicamente usan campos con datos compuestos de otros en una forma de anidamiento, por ejemplo: geometric\_msgs/Vector3, cuyos mensajes constan a su vez de tres campos que representan a: “x”, “y” y “z”.

Con *rosmsg* *show* se visualiza como están compuestos los campos, junto a los tipos y su identación. Además, pueden contar con arreglos de longitud fija o variable, y constantes. Por ejemplo, en la figura 11-5 se muestra el mensaje LaserScan.

Texto

Descripción generada automáticamente

Figura ‑ Composición del mensaje tipo laserscan para el lidar

**Creación de mensajes**

Se presenta un ejemplo de mensaje personalizados. Para la publicación de las detecciones del vehículo, se realiza en forma de arreglos, donde su contenido es similar a la estructura rectángulo de OpenCV. En él se incluyen x, y, ancho y altura. Para esto, en el paquete que los utiliza, se debe crear un folder para los mensajes “msg”, dentro se colocan los archivos tipo .msg. Para las detecciones usamos detec.msg, detecArray.msg y detecTiempos.msg.

Tenemos tres mensajes. En el primero se encuentra la estructura de una solo detección. En el segundo es el que se usa para publicar diferentes predicciones a lo largo del tiempo. Y el último para mandar un arreglo con las detecciones de diferentes vehículos:

|  |  |  |
| --- | --- | --- |
| detec.msg  uint8 x  uint8 y  uint8 width  uint8 height | detecTiempos.msg  detec posA  detec[] posSig | detecArray.msg  uint8 numDetec  detecTiempos[] array |

Para compilarlo en package.xml se agrega:

<build\_depend>message\_generation</build\_depend>

<run\_depend>message\_runtime</run\_depend>

En CMakeLists:

find\_package(catkin REQUIRED COMPONENTS

message\_generation

)

add\_message\_files(

FILES

detec.msg

detecTiempos.msg

detecArray.msg

)

En generate messages se agregan los tipos de menajes que se usan:

generate\_messages(

DEPENDENCIES

std\_msgs

)

catkin\_package(

CATKIN\_DEPENDS geometry\_msgs roscpp std\_msgs

message\_runtime

)

Una vez compilado, para usarlo en un programa se debe llamar con include:

#include "detecMSG/detec.h"

#include "detecMSG/detecArray.h"

**Workspace y paquetes**

Para crear un *workspace* primero se debe crear el directorio con *mkdir*. Lo siguiente es crear una carpeta *src* para los paquetes, desde ella se usa el comando:

catkin\_create\_pkg nombre\_del\_paquete

El cuál únicamente crea el directorio que llevara los nodos, junto con las carpetas de *build* y *devel*. Además se agregan dos archivos de configuración.

catkin\_create\_pkg nombre\_del\_paquete rospy roscpp sensor\_msgs

Permite durante la creación, incluir las librerías que se utilizarán, como *rospy* que es para Python y roscpp para C++.

Los archivos que se crean son package.xml, el cual es el archivo de configuración donde se agregan las dependencias. Y CMakeLists.txt que contiene una lista de instrucciones, como que ejecutables deben crearse, qué archivos fuente usar, dónde encontrar los archivos y bibliotecas que se usarán. Es el archivo package.xml el que lo identifica como un paquete de ROS.

Si se quiere agregar nuevas dependencias se debe modificar:

* Package.xml

Por ejemplo, para agregar *cv\_bridge* se incluye:

<build\_depend>cv\_bridge</build\_depende>

<build\_export>cv\_brdige</build\_export>

<exec\_depend>cv\_bridge</exec\_depend>

El primero para compilar, el segundo para exportarlo y el último para ejecutarlo.

También se debe modificar el archivo *CMakeLists* en:

find\_package(

cv\_bridge

)

Se recomienda usar *catkin\_make* para compilar antes de empezar a escribir. Con ello generará cabeceras, dependencias y bibliotecas.

**Programar un nodo**

ROS permite programar nodos tanto en C++ como en Python. En el caso de programar en C++, se debe incluir lo siguiente:

* El encabezado *ros/ros.h*
* La función *ros::init*
* Y *ros::NodeHandle* que registra el programa como un nodo.

En el caso de Python:

* import rospy
* rospy.init\_node('talker', anonymous=True)
* rospy.resolve\_name(name, caller\_id=None)

Este último, equivale a *NodeHandle*, pero para Python no es necesario.

**Suscripciones**

Para leer un mensaje se requiere declarar un suscriptor, el cual llama a una función a la que le envía el mensaje como parámetro, se llaman funciones de *Callback*:

ros::Subscriber camara\_sub =

nh.subscribe("/app/camera/rgb/image\_raw", 10,

camaraRGBCallback);

Dicha instrucción se escribe en el *main*, recibe el mensaje publicado en el tópico */app/camera/rgb/image\_raw* que corresponde a la cámara, y llama a la función *camaraRGBCallback*, el 10 corresponde al tamaño del buffer de los mensajes.

void camaraRGBCallback(const sensor\_msgs::Image& msg){}

Como se ve, en la función de *callback* recibe el mensaje, aquí se debe declarar el tipo, en este caso es un *sensor\_msgs::Image* (previamente se debía haber incluido *#include <sensor\_msgs/Image.h>*).

**Publicar**

Para publicar se requiere agregar el tipo de mensaje con *#include*. Aquí se requiere declarar un *Publisher* y necesita del *ros::NodeHandle*. Por lo que, si se quiere publicar desde dentro de una función es mejor declarar el *Publisher* como variable global:

ros::Publisher detec\_publisher;

E incluir dentro de la función:

ros::NodeHandle nh("~");

detec\_publisher =

nh.advertise<sensor\_msgs::Image>("/detec",1);

Se inicializa el *Publisher* con el tipo de mensaje, dentro el tópico o nombre con que se puede leer: */detec*.

Para publicarlo bastaría con crear una variable del mismo tipo como *sensor\_msgs::Image nombre*, y usar *detec\_publisher.publish(nombre).*

sensor\_msgs::Image img\_msg;

detec\_publisher.publish(img\_msg);

Para usar OpenCV se requiere importar los módulos correspondientes a las funciones que se usarán y cv\_bridge:

#include "opencv2/opencv.hpp"

#include <cv\_bridge/cv\_bridge.h>

#include "opencv2/imgproc/imgproc.hpp"

#include "opencv2/ml/ml.hpp"

Por un lado, Cv\_bridge permite traducir una imagen de ROS a tipo math para OpenCV:

Mat img;

cv\_bridge::CvImagePtr cv\_ptr;

cv\_ptr = cv\_bridge::toCvCopy(msg,

sensor\_msgs::image\_encodings::MONO8);

img = cv\_ptr->image.clone();

En *encodings* se puede convertir al formato deseado. En este caso es una imagen a blanco y negro, y cada pixel con un valor de 8 bits (0 a 255).

Por otro lado, para convertir una imagen tipo math a ROS, se requiere crear una variable del tipo del mensaje y rellenar sus campos, como el encabezado.

sensor\_msgs::Image img\_msg;

std\_msgs::Header header;

//header.seq = counter;

header.stamp = ros::Time::now();

img\_bridge = cv\_bridge::CvImage(header,sensor\_msgs::image\_encodings::TYP

E\_8UC1, img1);

img\_bridge.toImageMsg(img\_msg);

detec\_publisher.publish(img\_msg);

En el código anterior, se muestra cómo se convierte y se publica una imagen.

**Compilar un nodo**

Por un lado, con Python hay que asegurarse que el archivo es ejecutable, desde la consola de comandos se aplica:

chmod u+x

Una vez finalizado se ejecuta:

source devel/setup.bash

Que debería establecer variables que le permiten a ros encontrar el paquete y sus ejecutables.

Por otro lado, si el nodo está escrito en C++, primero debe agregarse dentro de CMakeLists como un ejecutable:

add\_executable(nombre\_del\_nodo fuente\_del\_nodo)

target\_link\_libraries(nombre\_del\_nodo ${catkin\_LIBRARIES})

${OpenCV\_LIBRARIES}

La primera instrucción declara el nombre del ejecutable y una lista de archivos fuente que lo componen. En caso de ser más de uno se separan con espacios. La segunda instrucción le indica a *CMake* que el nodo está ligado a las librerías definidas en *find\_package*. Además, para el use de OpenCV, en build include directories se agrega:

${OpenCV\_INCLUDE\_DIRS}

Posteriormente se debe usar:

catkin\_make

catkin\_make volverá a compilar todo el proyecto. Se puede usar para compilar un solo paquete cuando se quiera reflejar pequeños cambios en un programa (solo cuando el paquete ya fue construido anteriormente con catkin\_make):

catkin\_make --only-pkg-with-deps <paquete>

Aunque cmake falle, salvo que se deba a un error de código, se debe correr cuantas veces sea necesario hasta que finalice sin errores. Esto es para que agregue todos los paquetes.

Al finalizar se debe usar también:

source devel/setup.bash

## Transformaciones

Forma

Descripción generada automáticamente

Figura ‑ Conexión con transformaciones de las coordenadas de la cámara con el vehículo. Fuente (“access the tf transformation tree in ros - matlab & simulink - mathworks américa latina” s. f.)

Para cada objeto tanto del vehículo, como un punto detectado por una cámara o el LiDAR se puede representar su posición dentro de un *frame* o sistemas de coordenadas en 3D. Estos *frames*, como se observa en la figura 11-6, se pueden relacionar entre sí mediante transformaciones que se componen de traslaciones y rotaciones.

Entonces, con el objetivo de poder visualizar los datos obtenidos por los sensores todos juntos y en la posición correcta, se requiere asignarles *frames* de coordenadas a cada sensor respecto al carro, y mediante transformaciones unirlos al sistema de coordenadas del vehículo, llamado *base\_link*.

Los sensores al publicar mensajes de posición ya cuentan con sus propios *frames*:

Odometría: */odom*

Realsense camera: */SR300\_depth\_optical\_frame*

LiDAR: */laserscan*

Estos *frames* pueden visualizarse por separado, y partiendo de que están centrados en las coordenadas (0,0). Para ello en rviz se debe cambiar el *frame* global a cada uno y agregar el respectivo tópico que quiere visualizarse.

Por defecto en los paquetes del vehículo, el nodo Odometry ya cuenta con un nodo de transformadas tf, que para visualizar en rviz se debe hacer referencia al *frame base\_link*

Global options

Fixed frame

Cambiar a base\_link o odom

Cambiar a laser para ver laserscan

SR300\_depth\_optical\_frame

Para ver la cámara en el espacio 3D

(En el caso de la cámara mediante la opción de *add by topic* se pueden visualizar los mensajes tipo raw de color, IR y depth.)

*base\_link frame*: Está unido al robot, tal que es fijo, la x apunta al frente del carro. Se encuentra centrado y se mueve con él.

Frames de los sensores:

* *Base\_camara*: para la cámara de profundidad, se debe medir la distancia x, y, z que tiene la cámara respecto al centro del robot. Además, se deben hacer unas rotaciones: 1.57 (90°) en roll y 1.57 en pitch.
* *Base\_laser*: Para el LiDAR, no requiere rotaciones, solo traslación. Por lo que se debe medir x, y, z respecto al centro del robot.
* *Odom*: frame que debe ser continuo y evolucionar junto a la pose, lo ideal es que se publique a altas frecuencias para evitar saltos discretos. Representa el inicio del movimiento.

**Estructura del programa de las transformadas**

Para las transformaciones se requiere de un *broadcaster*, el cual usa 5 argumentos:

* La rotación especificada en cuaterniones
* Un vector de tres dimensiones que representa la traslación (x,y,z)
* Una marca de tiempo, normalmente se usa la del sistema: ros::Time::now()
* Nombre del frame padre, si es con respecto al carro: “base\_link”
* Nombre del frame hijo que se crea, como “base\_laser” para el LiDAR

Dentro de un programa:

#include <tf/transform\_broadcaster.h>

Se crea el mensaje

tf::TransformBroadcaster laser\_broadcaster;

ros::Time current\_time, last\_time;

current\_time = ros::Time::now();

last\_time = ros::Time::now();

geometry\_msgs::Quaternion

Para dar la rotación en cuaterniones, se puede simplemente llamar a una rutina de *transform* y dar el ángulo:

laser\_quat = tf::createQuaternionMsgFromYaw(th);

Se rellenan los campos del mensaje, el cual es de tipo *TransformStamped* y se encuentra en *geometry*:

geometry\_msgs::TransformStamped laser\_transf;

laser\_transf.header.stamp = current\_time;

laser\_transf.header.frame\_id = "base\_laser";

laser\_transf.child\_frame\_id = "base\_link";

laser\_transf.transform.translation.x = x;

laser\_transf.transform.translation.y = y;

laser\_transf.transform.translation.z = 0.0;

laser\_transf.transform.rotation = laser\_quat;

Imagen que contiene Gráfico

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Figura ‑ Resultado de la conexión de los frames. Fuente (“setting up transformations — navigation 2 1.0.0 documentation” s. f.)

Como en la figura 11-7, se le da los valores en xyz de la traslación, la rotación, el nombre del frame y el frame padre.

laser\_broadcaster.sendTransform(laser\_transf);

Esta es la transformada de un frame a otro. Falta entonces publicar el mensaje con la posición, esto se hace asociando el mensaje al frame. Como ventaja, los mensajes de ROS cuentan con un campo header en el cual se puede asociar. Por ejemplo, un mensaje de tipo laser, con frame\_id al frame creado, el cual ya tiene una transformada para conectarlo al vehículo:

sensor\_msgs::LaserScan tflaser;

tflaser.header.frame\_id = "base\_laser";

## Cámara

A continuación, se describe el uso de la cámara de profundidad realsense. Dentro de los primeros pasos realizados fue programar un par de nodos con la tarea de localizar una marca de algún color y el control para moverse a dicha marca. El objetivo fue comprender el funcionamiento de la cámara y cómo extraer los datos de la nube de puntos. Para comenzar, la cámara realsense publica los siguientes tópicos:

* Cámara de color en formato RGB

camera/color/image\_raw

* Cámara de profundidad con distancias en mm.

camera/depth/image\_raw

* Nube de puntos en formato XYZARGB, junta la imagen de profundidad con la de color.

camera/depth/points (sensor\_msgs/PointCloud2)

* Para las cámaras de infrarrojo:

camera/infrared1/image\_raw

camera/infrared2/image\_raw

Los tipos image\_raw publican mensajes del tipo sensor\_msgs/Image. Además, están los tipos Info que publican datos para la calibración.

El tópico Depth/points incorpora la información de las cámaras infrarroja y de color. Se debe tomar en cuenta que la información publicada es incompleta. Mezcla en un mensaje de tipo xyzargb la información de la cámara de profundidad con la argb. En caso de que falte algún dato de profundidad, su correspondiente color de ese píxel tampoco se incluye. Estos valores están rellenos de 0 y en r,g,b tienen un valor de 96.

Para extraer la información de la nube de puntos hay varias alternativas. Con Python se puede convertirlo con numpy asarray para separar en x, y, z, argb. Para separar los colores el problema es que se trata de un proceso muy lento usando unpack y pack de bytes. Se puede hacer la separación en C++ con reinterpret\_cast que accede al lugar de memoria y lo maneja como si se tratara de otro tipo de variable.

Desafortunadamente no hay una mejor opción para realizar este proceso en python. En cambio, en C++ se pueden usar unas librerías:

#include <pcl\_ros/point\_cloud.h>

#include <pcl/point\_types.h>

#include <pcl\_conversions/pcl\_conversions.h>

#include <pcl/conversions.h>

Lo que se hace es cambiar del tipo de mensaje sensor\_msgs::PointCloud2 a pcl::Point cloud<XYZRGBA> ("Point Cloud Library (PCL): pcl::PointXYZRGBA Struct Reference" s. f.) que permite acceder más fácilmente a los valores:

pcl::PCLPointCloud2 pcl\_pc2;

pcl\_conversions::toPCL(msg,pcl\_pc2);

pcl::PointCloud<pcl::PointXYZRGBA>::Ptr pt\_cloud(new

pcl::PointCloud<pcl::PointXYZRGBA>);

pcl::fromPCLPointCloud2(pcl\_pc2,\*pt\_cloud);

Entonces por medio de un ciclo se puede acceder para cada punto de la imagen a su valor correspondiente:

for(int i = 0; i < height\*width; ++i){

xA.push\_back(pt\_cloud->points[i].x);

yA.push\_back(pt\_cloud->points[i].y);

zA.push\_back(pt\_cloud->points[i].z);

r.push\_back(pt\_cloud->points[i].r);

g.push\_back(pt\_cloud->points[i].g);

b.push\_back(pt\_cloud->points[i].b);

}

En cambio, para usar las imágenes de tipo sensor\_msgs/Image se puede hacer uso de cv\_bridge para convertirlas a una matriz de OpenCV:

cv\_bridge::CvImagePtr cv\_ptr;

cv\_ptr = cv\_bridge::toCvCopy(msg,

sensor\_msgs::image\_encodings::MONO8);

Mat img = cv\_ptr->image.clone();

msg es la lectura recibida del tópico, y se define el formato al que se convierte la imagen, en este caso MONO8 es una imagen en blanco y negro, donde cada píxel vale 8 bits.

Para publicar una imagen se usa el camino inverso:

sensor\_msgs::Image img\_msg;

std\_msgs::Header header;

header.stamp = ros::Time::now();

img\_bridge = cv\_bridge::CvImage(header,

sensor\_msgs::image\_encodings::TYPE\_8UC1, img);

img\_bridge.toImageMsg(img\_msg);

detec\_publisherk.publish(img\_msg);

Se crea el tópico y se le agregan encabezados, con cv\_bridge se elige un encoding que transforme la imagen, y al final se puede publicar.

# Referencias

“Access the tf Transformation Tree in ROS - MATLAB & Simulink - MathWorks América Latina”. s. f. Accedido 15 de abril de 2023. <https://la.mathworks.com/help/ros/ug/access-the-tf-transformation-tree-in-ros.html>.

Andreas Geiger, Philip Lenz, Christoph Stiller, y Raquel Urtasun. 2013. “Vision meets Robotics: The KITTI Dataset”. *International Journal of Robotics Research (IJRR)*, International Journal of Robotics Research (IJRR), . <https://www.cvlibs.net/publications/Geiger2013IJRR.pdf>.

Arróspide, Salgado, y Nieto. “Video analysis based vehicle detection and tracking using an MCMC sampling framework” EURASIP Journal on Advances in Signal Processing. 2012. doi: 10.1186/1687-6180-2012-2

Aswathy, M C. 2012. “A Cluster Based Enhancement to AODV for Inter-Vehicular Communication in VANET”. *International Journal of Grid Computing & Applications* 3 (3): 41-50. <https://doi.org/10.5121/ijgca.2012.3304>.

Ayala-Alfaro, Victor. (2019) 2021. “AutoModelCar packages v3”. Python. <https://github.com/Victor-ayala/autoModelCar>.

Becker, Alex. s. f. “Online Kalman Filter Tutorial”. Accedido 5 de junio de 2023. <https://www.kalmanfilter.net/>.

Bougharriou, Hamdaoui, y Mtibaa, 2017, “Linear SVM classifier based HOG car detection”. 2017 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA) 241-245. doi: 10.1109/STA.2017.8314922.

Cabrera, Margarita, y Francesc Tarrés. s. f. “Multiplexación por división en frecuencias ortogonales (OFDM)”, Universidad Oberta de Cataluña, . <https://openaccess.uoc.edu/bitstream/10609/63345/2/Teor%C3%ADa%20de%20la%20codificaci%C3%B3n%20y%20modulaciones%20avanzadas_M%C3%B3dulo%205_Multiplexaci%C3%B3n%20por%20divisi%C3%B3n%20en%20frecuencias%20ortogonales%28OFDM%29.pdf>.

“catkin/workspaces - ROS Wiki”. s. f. Accedido 12 de abril de 2023. <http://wiki.ros.org/catkin/workspaces>.

“cv\_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages - ROS Wiki”. s. f. Accedido 4 de septiembre de 2023. <http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages>.

“es - ROS Wiki”. s. f. Accedido 12 de abril de 2023. <http://wiki.ros.org/es>.

“Estadísticas a propósito del día mundial en recuerdo de las víctimas de accidentes de tránsito”. 2022, INEGI Comunicado de prensa, n.o 662/22 (noviembre): p. 6

Faisal, Asif, Tan Yigitcanlar, Md. Kamruzzaman, y Graham Currie. 2019. “Understanding Autonomous Vehicles: A Systematic Literature Review on Capability, Impact, Planning and Policy”. *Journal of Transport and Land Use* 12 (1). <https://doi.org/10.5198/jtlu.2019.1405>.

Felipe Pérez. (2017) 2017. “AP-automodelcar”. C++. <https://github.com/Flperez/AP-automodelcar>.

Feng Han, Ying Shan, Ryan Cekander, Harpreet S. Sawhney, y Rakesh Kumar. 2006. “A Two-Stage Approach to People and Vehicle Detection With HOG-Based SVM”. *PERFORMANCE METRICS FOR INTELLIGENT SYSTEMS WORKSHOP*, PERFORMANCE METRICS FOR INTELLIGENT SYSTEMS WORKSHOP, , 133-40.

Fernández Villaverde, Jesús. s. f. “Kalman and Particle Filtering”. University of Pennsylvania. <https://www.sas.upenn.edu/~jesusfv/filters_format.pdf>.

Hartenstein, Hannes, y Kenneth P Laberteaux, eds. 2010. *VANET: Vehicular Applications and Inter‐Networking Technologies*. 1.a ed. Wiley. <https://doi.org/10.1002/9780470740637>.

“Hardware (AutoNOMOS Model v2) · AutoModelCar/AutoModelCarWiki Wiki · GitHub”. s. f. Accedido 11 de julio de 2023. <https://github.com/AutoModelCar/AutoModelCarWiki/wiki/Hardware-(AutoNOMOS-Model-v2)>.

“IDENTIFICACIÓN DE NECESIDADES DE ESPECTRO PARA SISTEMAS DE TRANSPORTE INTELIGENTE EN LA BANDA 5850-5925 MHz”. 2021. Instituto Federal de Telecomunicaciones. <https://www.ift.org.mx/sites/default/files/industria/temasrelevantes/17437/documentos/documentodereferenciaidentificaciondenecesidadesstien59ghz_0.pdf>.

Jakubiak, Jakub, y Yevgeni Koucheryavy. 2008. “State of the Art and Research Challenges for VANETs”. En *2008 5th IEEE Consumer Communications and Networking Conference*, 912-16. Las Vegas, Nevada, USA: IEEE. <https://doi.org/10.1109/ccnc08.2007.212>.

Janai, Joel, Fatma Güney, Aseem Behl, y Andreas Geiger. 2021. “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art”. arXiv. [http://arxiv.org/abs/1704.05519.](http://arxiv.org/abs/1704.05519.%20)

Janne Salmi. 2000. “AODV Multicast Features”. Department of Computer Science and Engineering Helsinki University of Technology. <https://www.niksula.hut.fi/~janski/iwork/>.

Kecman, Vojislav. 2005. “Support Vector Machines – An Introduction”. En *Support Vector Machines: Theory and Applications*, 177:605-605. <https://doi.org/10.1007/10984697_1>.

Kenney, John. 2011. “Dedicated Short-Range Communications (DSRC) Standards in the United States”. *Proceedings of the IEEE* 99 (agosto): 1162-82. <https://doi.org/10.1109/JPROC.2011.2132790>.

Kevin Gay. 2022. “WG: VT/ITS/AV Decision Making”. IEEE. 2022. <https://sagroups.ieee.org/2846/>.

Kyle Barratt. 2021. “DSRC vs C-V2X: Comparing the Connected Vehicles Technologies - GTT Wireless”. 18 de noviembre de 2021. <https://www.gttwireless.com/dsrc-vs-c-v2x-comparing-the-connected-vehicles-technologies/>.

Martínez-Díaz, Margarita, y Francesc Soriguera. 2018. “Autonomous Vehicles: Theoretical and Practical Challenges”. *Transportation Research Procedia* 33: 275-82. <https://doi.org/10.1016/j.trpro.2018.10.103>.

Nelson, Joseph. 2020. “The Importance of Blur as an Image Augmentation Technique”. Roboflow Blog. 13 de marzo de 2020. <https://blog.roboflow.com/using-blur-in-computer-vision-preprocessing/>.

O’Kane, Jason M. 2014. *A Gentle Introduction to ROS*. Version 2.1.1 (3e3d9c5), Generated in November 20, 2014. Columbia, SC: Jason M. O’Kane.

“OpenCV: Cascade Classifier”. s. f. Accedido 29 de mayo de 2023. <https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html>.

“OpenCV: Cascade Classifier Training”. s. f. Accedido 21 de abril de 2023. <https://docs.opencv.org/3.4/dc/d88/tutorial_traincascade.html>.

“OpenCV: Introduction to Support Vector Machines”. s. f. Accedido 31 de mayo de 2023. <https://docs.opencv.org/4.x/d1/d73/tutorial_introduction_to_svm.html>.

“OpenCV: Smoothing Images”. s. f. Accedido 31 de mayo de 2023. <https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html>.

Pérez Reyes. 2023. “AutoModelCar\_2023\_UPV”. C++. <https://github.com/OmarPerezReyes/AutoModelCar_2023_UPV>.

Pietikäinen, Matti. 2010. “Local Binary Patterns”. *Scholarpedia* 5 (3): 9775. <https://doi.org/10.4249/scholarpedia.9775>.

“Point Cloud Library (PCL): pcl::PointXYZRGBA Struct Reference”. s. f. Accedido 12 de abril de 2023. [https://pointclouds.org/documentation/structpcl\_1\_1\_point\_x\_y\_z\_r\_g\_b\_a.html.](https://pointclouds.org/documentation/structpcl_1_1_point_x_y_z_r_g_b_a.html.%20)

Pressman, R. 2010. *Ingeniería del Software un Enfoque Practico*. México, D.F: McGraw-Hill

Ren, Shaoqing, Kaiming He, Ross B. Girshick, y Jian Sun. 2015. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. *CoRR* abs/1506.01497. <http://arxiv.org/abs/1506.01497>.

“Road vehicles — Functional safety ISO 26262-1:2011”. ISO. 2011. <https://www.iso.org/standard/43464.html>.

Rosebrock, Adrian. 2015. *Local Binary Patterns with Python & OpenCV.* PyImageSearch (blog). 7 de diciembre de 2015. <https://pyimagesearch.com/2015/12/07/local-binary-patterns-with-python-opencv/>.

Saleh, Kaziwa, Sándor Szénási, y Zoltán Vámossy. s. f. *Occlusion Handling in Generic Object Detection*

*Setting Up Transformations — Navigation 2 1.0.0 documentation*. s. f. Accedido 15 de abril de 2023. <https://navigation.ros.org/setup_guides/transformation/setup_transforms.html>.

Shetty, Anirudha B., Bhoomika, Deeksha, Jeevan Rebeiro, y Ramyashree. 2021. *Facial recognition using Haar cascade and LBP classifiers.* Global Transitions Proceedings 2 (2): 330-35. <https://doi.org/10.1016/j.gltp.2021.08.044>.

Shuttleworth, Jennifer. 2019. “SAE J3016 Automated-Driving Graphic”. 2019. <https://www.sae.org/site/news/2019/01/sae-updates-j3016-automated-driving-graphic>.

*SVM bias on weights of positives and negatives - OpenCV Q&A Forum*. s. f. Accedido 18 de abril de 2023. <https://answers.opencv.org/question/26818/svm-bias-on-weights-of-positives-and-negatives/>.

Tariq Islam y Cheolhyeon Kwon. 2022. *Survey on the State-of-the-Art in Device-to-Device Communication: A Resource Allocation Perspective | Elsevier Enhanced Reader.* 2022. <https://doi.org/10.1016/j.adhoc.2022.102978>.

Tyagi, Mrinal. 2021. *HOG(Histogram of Oriented Gradients)*. Medium. 24 de julio de 2021. <https://towardsdatascience.com/hog-histogram-of-oriented-gradients-67ecd887675f>.

*What Are Convolutional Neural Networks? | IBM.* s. f. Accedido 5 de junio de 2023. <https://www.ibm.com/topics/convolutional-neural-networks>.

Zetina, Erick. (2018) 2023. “ai\_controlled\_car”. Python C++. <https://github.com/Zetinator/ai_controlled_car>.

Zhou, Zhi-Qiang. 2019. *HOG Visualization Using OpenCV(C++)*. Github. <https://github.com/zhouzq-thu/HOGImage>.