

ESCUELA DE INGENIERÍAS INDUSTRIALES

Ingeniería Robótica, Electrónica y Mecatrónica

Navegación Autónoma en Exteriores de un Robot
Móvil mediante Waypoints

Autonomous Outdoor Navigation of a Mobile Robot
using Waypoints

Realizado por
André Jose Lorenzo Torres

Tutorizado por
Javier Gonzalez Monroy
Cotutorizado por
Cipriano Galindo Andrades

Departamento
Ingeniería de sistemas y Automática,
UNIVERSIDAD DE MÁLAGA

MÁLAGA, 2 de Marzo de 2024



ESCUELA DE INGENIERÍAS INDUSTRIALES
GRADO EN INGENIERÍA ROBÓTICA, ELECTRÓNICA Y MECATRÓNICA

Navegación Autónoma en Exteriores de un Robot Móvil mediante
Waypoints
Autonomous Outdoor Navigation of a Mobile Robot using Waypoints

Realizado por
André Jose Lorenzo Torres
Tutorizado por
Javier Gonzalez Monroy
Cotutorizado por
Cipriano Galindo Andrades
Departamento
Ingeniería de sistemas y Automática

UNIVERSIDAD DE MÁLAGA
MÁLAGA, 2 de Marzo de 2024

Fecha defensa:
El Secretario del Tribunal

Agradecimientos

En primer lugar, quiero expresar mi más sincero agradecimiento a mi tutor, Javi, por su apoyo a lo largo de todo el proyecto, su guía ha sido fundamental para la buena finalización de este trabajo.

También agradecer a Mapir, el equipo de investigación donde he tenido el placer de trabajar para la realización de este TFG, aquí todos los integrantes me han ayudado en algún punto y por ello creo que sin su colaboración habría enfrentado mayores desafíos.

Por último, quiero agradecer a mis amigos y familiares, que me han apoyado durante este tedioso pero inspirador camino, que ha sido fundamental para mí. Sin su apoyo, este logro no habría sido posible.

Resumen: Desde el auge de la navegación autónoma en los últimos años, esta ha empezado a tener un papel crucial en diversas industrias y aplicaciones, desde la logística y la agricultura hasta la exploración espacial. El dominio de la navegación autónoma se ha convertido en un aspecto fundamental para garantizar el éxito y la eficiencia de estas tecnologías.

En el presente trabajo se aborda el desafío de permitir que un robot móvil dotado de un sistema sensorial navegue de manera autónoma en el exterior. Se han integrado y desarrollado componentes de software para conseguir una localización y una navegación precisas junto con un sistema de control y monitorización. Para ello, se han explorado diversas técnicas de localización y navegación. También se ha investigado en profundidad sobre el hardware de los sensores y la manera de interconectarlos a ROS2 mediante drivers, para el control adaptativo y robusto del trayecto.

Aunque el sistema también se ha probado en simulación, principalmente ha sido testado en los entornos de la Facultad de Telecomunicaciones de la Universidad de Málaga.

Los resultados han mostrado una buena implementación de las herramientas disponibles, junto con un sistema robusto y fiable de navegación autónoma en exteriores.

Palabras claves: Robótica Móvil, ROS2, Navegación Autónoma, LIDAR, GPS RTK, Localización, EKF, Planificadores, árboles de comportamiento.

Abstract: Since the rise of autonomous navigation in recent years, it has begun to play a crucial role in various industries and applications, from logistics and agriculture to space exploration. Mastery of autonomous navigation has become a fundamental aspect to ensure the success and efficiency of these technologies.

In this work, the challenge of enabling a mobile robot equipped with a sensory system to navigate autonomously outdoors is addressed. Software components have been integrated and developed to achieve precise localization and navigation, along with a control and monitoring system. To this end, various localization and navigation techniques have been explored. Additionally, an in-depth investigation into the sensor hardware and the way to interconnect them to ROS2 via drivers has been conducted, for the adaptive and robust control of the path.

Although the system has also been tested in simulation, it has mainly been tested in the environments of the Faculty of Telecommunications at the University of Málaga.

The results have shown a good implementation of the available tools, along with a robust and reliable outdoor autonomous navigation system.

Keywords: Mobile Robotics, ROS2, Autonomous Navigation, LIDAR, GPS RTK, Localization, EKF, Planners, behavior trees.

Índice de contenidos

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos del estudio	2
1.3. Estructura del trabajo	2
2. Estado del arte	5
2.1. Navegación Autónoma de vehículos Ackermann	5
2.2. Nav2: Stack para navegación autónoma de robots móviles	7
2.3. Localización	8
2.3.1. SLAM: Simultaneous Localization And Mapping	8
2.3.2. AMCL: Adaptive Monte Carlo Localization	9
2.3.3. Fusión de odometrías mediante filtros extendidos de Kalman	9
2.4. Sistemas sensoriales para Navegación Autónoma	10
2.4.1. Lidar	10
2.4.2. GPS	11
2.4.3. IMU	12
3. Desarrollo	13
3.1. Diseño del sistema sensorial	13
3.1.1. Lidar 3D Ouster	14
3.1.2. GPS RTK Reach RS2+	16
3.1.3. IMU Xsens MTi-3th	16
3.1.4. Implementación del sistema sensorial en ROS2	17
3.2. Interfaz de usuario	19
3.3. Procesamiento de los mensajes MQTT a ROS2	21
3.4. Localización y odometría	22
3.5. Mejoras en la localización para la orientación del sistema	27
3.6. Navegación Autónoma haciendo uso de Nav2	28
3.6.1. Conceptos básicos de Nav2	29
Árboles de comportamiento	29
Servidores de acción	30
Estructura para el control de navegación	31
3.6.2. Servidores	31
Navigator server	31
Behavior server	32
Controller server	32
Planner server	34

ÍNDICE DE CONTENIDOS

Smoother server	35
3.6.3. Mapas de coste	36
3.6.4. Plugins	37
3.6.5. Nav2 commander	38
4. Pruebas y resultados	39
4.1. Pruebas de precisión para el sistema sensorial	39
4.2. Pruebas de funcionamiento de los algoritmos de adaptación	41
4.3. Pruebas de precisión en la localización	42
4.4. Pruebas de navegación autónoma	44
5. Conclusiones	49
6. Futuras líneas de trabajo	51
Bibliografía	53

CAPÍTULO 1

Introducción

1.1. Motivación

La navegación autónoma de vehículos ha emergido como uno de los desafíos más apasionantes y complejos en el campo de la robótica. Con la creciente demanda en este ámbito, se han desarrollado múltiples empresas con proyectos que intentan ofrecer soluciones a un problema tan interesante como es la posibilidad de que un robot móvil se desplace autónomamente trazando un recorrido o ruta establecida. Según [1] existe un crecimiento anual del 5% en el volumen de ventas de robots industriales y, así como hace unos años, este crecimiento sería principalmente de manipuladores. Cada vez más, es la robótica móvil la que encuentra su camino en la industria, dando soluciones versátiles y de muy fácil implementación para tareas que antes requerían grandes espacios, donde ahora un robot móvil de pequeñas dimensiones puede soportar cargas muy grandes y ser implementado en la producción sin necesidad de modificar el entorno en lo más mínimo.

Otro punto importante es la problemática en la navegación. Esta incluye conseguir un método fiable de localización, donde la solución reside en estimar la posición final del robot compensando los errores incrementales de odometría acumulados. Además, es necesario minimizar los errores producidos por el sistema sensorial y detectar obstáculos mediante una sensorización a distancia que nos permita evitarlos y, de esta manera, planificar un camino de manera segura.

A diferencia de los espacios interiores, donde el entorno está controlado y predefinido, la navegación en exteriores presenta una serie de desafíos únicos que requieren soluciones innovadoras y adaptativas. Entre estos desafíos se encuentran la variabilidad del entorno, los cambios en las condiciones climáticas, la presencia de obstáculos dinámicos, la diversidad de superficies en el terreno y la localización en un entorno con tantas fuentes de ruido e interferencias.

Por estas razones, cada vez más equipos de investigación se centran en conseguir nuevas formas de mejorar y estandarizar las soluciones que existen, desde robots de rescate, militares o vehículos para uso personal. Aunque existen multitud de formas de afrontar este problema, hay un amplio margen de mejora. En el presente trabajo se ha tomado especial atención a la parte de localización, adaptando metodologías de estimación, como es el caso de el filtro extendido de Kalman [2], dado que la localización presenta múltiples complicaciones a la hora de ser realizada en exteriores. También se han implementado

algoritmos que intentan mejorar los errores producidos por los sensores y, finalmente, algoritmos de planificación de trayectorias que se adecuen bien con el modelo del robot (Ackermann).

1.2. Objetivos del estudio

Para considerar como finalizada la realización de este trabajo se van a establecer unos objetivos a conseguir, estos son:

- Sistema de sensorización, se dispondrá de un montaje en el chasis del robot de manera que asegure un buen uso de los dispositivos y se realizará una implementación de drivers para su uso con el SDK ROS2. Se debe buscar una manera efectiva de montar, conectar y conectar con el software.
- Interfaz de usuario, se debe buscar una manera fiable, versátil e intuitiva para que un usuario pueda probar y controlar el robot de manera rápida y sencilla, también se buscará la posibilidad de que la interfaz sea fácilmente modificable para futuras actualizaciones.
- Localización, objetivo fundamental para la navegación, se abordará la fusión de datos sensoriales heterogéneos para obtener una localización precisa y fiable.
- Evitación de obstáculos, se buscará una navegación segura, es decir, se configurarán sensores y algoritmos de navegación para conseguir una evitación de obstáculos estáticos y dinámicos.
- Navegación punto a punto y en trayectorias establecidas, se desarrollará un sistema de algoritmos que consigan una navegación basada en waypoints. El robot deberá visitar estos puntos de manera consecutiva y deberán estar basados en coordenadas GPS (latitud, longitud, altitud).

1.3. Estructura del trabajo

Este proyecto ha sido dividido en 6 capítulos bien definidos:

Primero, una introducción donde se presenta el tema y se dejan claros los objetivos del proyecto y los antecedentes en relación al tema trabajado.

Segundo, el estado del arte donde se desarrollan los temas que en el presente trabajo se exponen, sus antecedentes históricos y su base de funcionamiento práctico

Tercero, el desarrollo en si del trabajo donde se explican en detalle tanto el hardware utilizado, los métodos y herramientas, como la manera en la que se ha procedido para completar el proyecto. Esta parte ha sido escrita de manera cronológica en cuanto a los pasos que se han seguido junto con las soluciones que se han ido dando a los diversos problemas que han aparecido durante el recorrido de este proyecto.

Cuarto, las pruebas del proyecto que incluyen datos experimentales, gráficas y simulaciones donde se comparan tanto las ideas teóricas con los resultados reales como las

simuladas con las reales, se hace un estudio exhaustivo de donde hubo más complicaciones a la hora de llevar el proyecto a la realidad y como se fueron solucionando cada uno de los problemas.

Quinto, conclusiones finales del proyecto donde se reflexiona sobre las soluciones obtenidas y en cuanto son de fiables en un entorno tan variable.

Sexto, futuras líneas del trabajo donde se podría continuar este proyecto y mejorarlo.

CAPÍTULO 2

Estado del arte

En el presente capítulo se documentará un análisis detallado sobre las técnicas, dispositivos y contexto en cuanto a navegación autónoma se refiere. Se hará incapié en los antecedentes y actualidad respecto a la robótica móvil; las tecnologías software usadas, explicando así su fundamento teórico y, por último, los dispositivos sensoriales usados.

2.1. Navegación Autónoma de vehículos Ackermann

La robótica y la navegación autónoma han sido áreas de investigación fascinantes y de rápido avance en las últimas décadas. Desde los primeros experimentos pioneros hasta los desarrollos más recientes, estas tecnologías han abierto un amplio espectro de posibilidades en diversos campos, desde la exploración espacial como el rover perseverance [3] de la NASA hasta la logística industrial con los manipuladores móviles de Robotnik [4].

Uno de los puntos de partida fundamentales en la historia de la robótica autónoma fue el trabajo realizado por el neurofisiólogo británico William Grey Walter en la década de 1940 [5]. Walter diseñó y construyó los "robots tortuga", **figura 2.1**, pequeños dispositivos autónomos que demostraron comportamientos primitivos de evasión de obstáculos y búsqueda de energía, destacando su capacidad para adaptarse y responder al entorno sin control humano directo.



Figura 2.1: Interior de los robots "tortuga". Fuente: alpoma.net

Otro hito importante en este campo fue el desarrollo de la furgoneta autónoma Mercedes-Benz de Ernst Dickmanns y su equipo en la década de 1980 [6]. Este vehículo, **figura 2.2**, equipado con sensores y sistemas de control avanzados, fue capaz de navegar de forma

autónoma por carreteras y seguir a otros vehículos con un alto grado de precisión, sentando las bases para los sistemas de conducción autónoma modernos. Este vehículo fue incluso capaz de circular por una "autobahn" alemana a 90 km/h



Figura 2.2: Furgoneta Autónoma creada por Ernst Dickmanns. Fuente: [6].

En el ámbito militar, la Agencia de Proyectos de Investigación Avanzada de Defensa (DARPA) de los Estados Unidos ha desempeñado un papel crucial en el impulso de la robótica y la navegación autónoma. Uno de los proyectos emblemáticos de DARPA fue el primer vehículo que funcionaba mediante un radar, un láser, y visión computarizada. En 1987, los laboratorios HRL demostraron que se podía construir un vehículo que podía diseñar su propia ruta una vez que se salía del mapa [7]. El vehículo pudo moverse más de 600 metros a través de terreno complejo como pendientes, grandes rocas y vegetación.

Pronto grandes empresas como Google, Audi o más adelante Tesla iniciaron una nueva revolución en el mundo de la conducción autónoma, introduciendo los conceptos de *machine learning* o *deep learning* en sus vehículos como una nueva forma de toma de decisiones, de esta manera los vehículos primero aprendían por si mismos a qué decisiones tomar en cada situación en base a un entrenamiento, esto dio lugar a una gran ventaja ya que, en este campo la programación clásica era inviable por la gran cantidad de posibilidades que existen.

Audi comenzó esta travesía con su modelo RS7 autónomo [8], que recorrió a una velocidad de 240 km/h el circuito de Hockenheim en Alemania. Siendo este 5 segundos más rápido que un vehículo tripulado por un conductor profesional. Le siguió Google con una flota de 25 vehículos autónomos que sin superar los 40 km/h recorrieron las calles de Mountain View, California.

Más adelante salieron empresas como Waymo [9], que empezaron a desarrollar el concepto de taxi autónomo o Tesla con sus vehículos que incorporan el conocido **Autopilot**. Este utiliza una combinación de cámaras, radares y sensores para ofrecer características avanzadas de asistencia al conductor, como el piloto automático adaptativo y la asistencia de cambio de carril.

2.2. Nav2: Stack para navegación autónoma de robots móviles

ROS es un SDK o framework para el desarrollo de software en robots, fue desarrollado en 2007 en la universidad de Standford para dar soporte a un proyecto interno, desde el 2008 ha sido mantenido principalmente por Willow Garage, una incubadora de empresas y laboratorio de investigación robótica, aunque por su naturaleza de código abierto el crecimiento y el mantenimiento ha sido una labor común de sus usuarios [10].

ROS2 funciona en base a Nodos, programados en C++ o Python, estos comprenden los llamados paquetes y el conjunto de paquetes es un stack, uno de los stacks más conocidos, usados y mejorados es el utilizado en este trabajo, Nav2 o Navigation2 [11], que en ya su segunda versión es la opción más utilizada para la creación de algoritmos de navegación autónoma. Este stack tiene todo lo necesario para crear un sistema robusto de navegación autónoma, principalmente en interiores pero también para exteriores con pequeñas modificaciones.

El stack posee soporte para todo tipo de robots (Ackermann, Diferencial, Humanoide, Omni-direccional), aquí nos referiremos a los planificadores locales como "controladores" y a los planificadores globales como simplemente "planificadores". Nav2 cuenta con controladores muy rápidos y sencillos como el DWB (Dynamic Window Approach) que funciona exclusivamente para robots con posibilidad de giro *in situ*, TEB (Time Elastic Band) o el famoso RPP (Regulated Pure Pursuit) que aunque sencillos ya nos permiten una configuración más avanzada para el caso que aquí se presenta, como es el radio de giro mínimo, hasta otros mucho más complejos como el MPPI (Model Predictive Path Integral), un algoritmo muy robusto que usa un método de predicción en el tiempo para navegación de vehículos a grandes velocidades. Por otro lado tenemos los planificadores donde existen varias versiones de 2 algoritmos también muy conocidos, el A* y el Theta* [12].

En su primera versión usaba una máquina de estados finita como lógica de actuación, una solución muy bien estudiada por su antigüedad pero que su modificación para un problema concreto resultaba complicada. Por ello, en su nueva versión se utilizan los llamados *behavior trees* [13], unas estructuras lógicas muy versátiles y sobre todo extremadamente adaptables, gracias a ello el stack de navegación permite una configuración absoluta de todos sus componentes, desde la creación de planificadores locales, globales o plugins para añadir comportamientos hasta, capas para los mapas de coste, suavizadores de trayectoria o incluso la creación o modificación de los propios árboles de comportamiento. Su estructura funciona por bloques como se muestra en la **figura 2.3**.

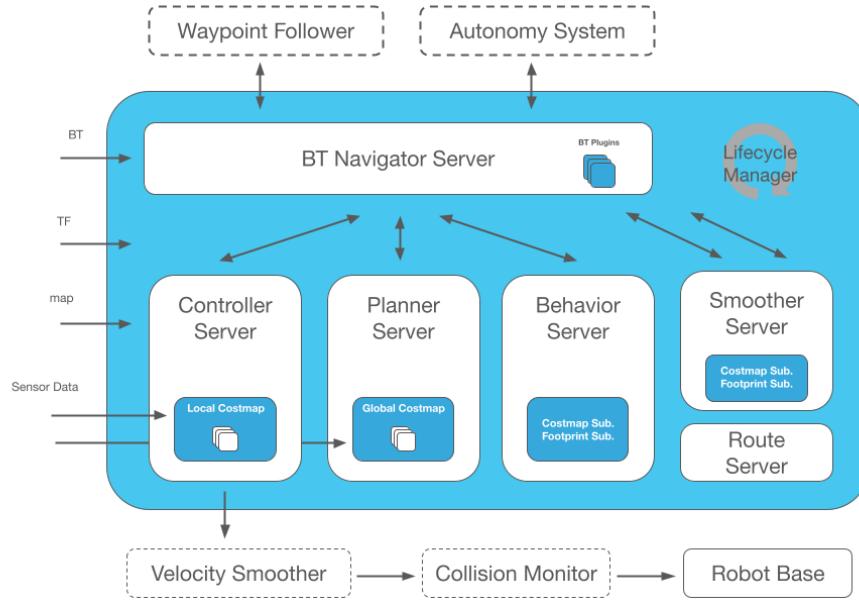


Figura 2.3: Arquitectura interna de Nav2. Fuente: [11]

2.3. Localización

La localización se conoce como uno de los principales y más complejos problemas de la robótica móvil. Se basa en un concepto muy sencillo, saber donde se encuentra el robot, una tarea que depende del entorno, de las condiciones y de los sensores disponibles, pudiendo así suponer una gran problemática. Por ello, durante los años se han desarrollado multiples técnicas de localización para intentar mitigar este gran problema, aquí se exponen los principales métodos usados a día de hoy en robótica.

2.3.1. SLAM: Simultaneous Localization And Mapping

El método SLAM, es un método utilizado en robótica y campos relacionados para que un agente móvil (como un robot) pueda construir un mapa de un entorno desconocido y al mismo tiempo determinar su propia posición respecto de ese mapa.

El proceso comienza con una estimación inicial de la posición del robot y el mapa del entorno. Esta estimación puede ser rudimentaria y se mejora a medida que el robot explora y recopila más datos, más adelante sucede una captura de datos donde el agente móvil o robot, utiliza sus sensores como cámaras, Lidar o diferentes fuentes de odometría para capturar información sobre la disposición del entorno y de su propia posición. A partir de los datos capturados se construye un modelo del entorno que puede estar en 2 o 3 dimensiones. A medida que el agente móvil se mueve y recopila más datos, el mapa y la estimación de la posición se actualizan continuamente para reflejar el conocimiento más reciente del entorno y la ubicación del agente. El algoritmo incluye técnicas avanzadas de fusión sensorial, estimación probabilística y optimización para lograr una localización y un mapeo precisos.

Este método es uno de los más usados en navegación en interiores donde hay muchos objetos y referencias para que este algoritmo pueda "localizarse". El principal problema recae al intentar usarlo en exteriores, donde suele haber pocas referencias o están mucho

más espaciadas, donde el ruido es mucho mayor y existen más fuentes, pero sobre todo, donde la idea de mapear un entorno tan extenso se vuelve inviable para la mayoría de situaciones.

2.3.2. AMCL: Adaptive Monte Carlo Localization

AMCL es un método probabilístico para la localización de robots que utiliza un enfoque de Monte Carlo (también conocido como filtro de partículas). A diferencia de otros métodos, AMCL puede adaptarse dinámicamente a la incertidumbre y las fluctuaciones en el entorno.

Primeramente se genera un conjunto de partículas que representan las posiciones probables del robot en función a una distribución uniforme, cada partícula se actualiza de acuerdo con el movimiento esperado del robot. Esto se logra aplicando las entradas de control del robot y considerando la incertidumbre del movimiento. Las partículas se ponderan de acuerdo con su probabilidad de ser correctas. Las partículas con mayor probabilidad se duplican, mientras que las partículas con menor probabilidad se eliminan.

AMCL es un método flexible y robusto, puede proporcionar una estimación precisa de la posición del robot incluso en condiciones cambiantes. Una gran desventaja es que necesita al igual que SLAM un mapa del entorno, algo poco viable en exteriores como ya se ha comentado.

2.3.3. Fusión de odometrías mediante filtros extendidos de Kalman

En la fusión de odometrías con filtros extendidos de Kalman, se utiliza un modelo del sistema y mediciones provenientes de múltiples fuentes para estimar su estado, que en este caso, sería la posición y la orientación del robot.

El filtro de Kalman es un algoritmo predictivo y recursivo desarrollado por Rudolf E. Kalman en 1960. Este algoritmo sirve para identificar el "estado oculto" o no medido de un sistema dinámico **lineal** teniendo en cuenta las varianzas de los ruidos que afectan al sistema (errores en las mediciones del sistema de sensado), este algoritmo conlleva 2 partes bien definidas, la primera es una predicción de estados donde dada una matriz que relaciona el estado anterior con el estado presente se calcula una estimación de estados y una matriz de varianzas *a priori* y posteriormente una etapa de corrección mediante una medición para calcular el llamado residuo de medición y la *ganancia de Kalman*, que a diferencia de otros métodos [14] tiene la ventaja de ser calculada dinámicamente en base a la información del error (matriz de covarianzas). Finalmente se corrige la estimación y se repite el proceso.

En el caso de sistemas dinámicos **no lineales** es posible usar una modificación del filtro conocida por "filtro extendido de Kalman o EKF" [15], donde se linealiza entorno al estado actual y donde antes teníamos una matriz que relaciona el estado anterior con el actual ahora tenemos la función f para la predicción de estados y su *Jacobiana* (\mathbf{F}) para la predicción de varianzas (2.1 y 2.2).

Para la segunda etapa también tendremos otra función h y su *Jacobiana* (\mathbf{H}) que representan la relación entre las mediciones y el estado actual que, sumado a la matriz de varianzas \mathbf{R} nos otorga la actualización de estados y de covarianza (2.3, 2.4 y 2.5).

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_k) \quad (2.1)$$

$$P_k^- = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1} \quad (2.2)$$

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \quad (2.3)$$

$$\hat{x}_k = \hat{x}_k^- + K_k (z_k - h(\hat{x}_k^-)) \quad (2.4)$$

$$P_k = (I - K_k H_k) P_k^- \quad (2.5)$$

Donde :

- \hat{x}_k^- : Predicción del estado a priori en el instante de tiempo k .
- \hat{x}_{k-1} : Estado estimado en el instante de tiempo anterior $k - 1$.
- P_k^- : Predicción a priori de la covarianza del estado en el instante de tiempo k .
- P_{k-1} : Covarianza del estado en el instante de tiempo anterior $k - 1$.
- Q_{k-1} : Covarianza del ruido del proceso en el instante de tiempo $k - 1$.
- K_k : Ganancia de Kalman en el instante de tiempo k .
- R_k : Covarianza del ruido de medición en el instante de tiempo k .
- \hat{x}_k : Estado estimado en el instante de tiempo k después de la corrección.
- $F_{k-1} = \left. \frac{\partial f}{\partial x} \right|_{x=\hat{x}_{k-1}}$: Matriz Jacobiana de la función de transición de estado.
- $H_k = \left. \frac{\partial h}{\partial x} \right|_{x=\hat{x}_k^-}$: Matriz Jacobiana de la función de observación.

2.4. Sistemas sensoriales para Navegación Autónoma

Un robot requiere siempre de un sistema que le permita interactuar con el entorno, detectarlo y analizarlo. Es por ello que la sensorización también es una parte fundamental en robótica. El esquema de sistemas sensoriales para uso en robótica móvil y más concretamente en navegación puede variar pero fundamentalmente requiere, por un lado, de una fuente de localización sin error acumulativo y otra fuente de odometría que proporcione posición y orientación de manera rápida y por otro lado, algún tipo de sistema que sea capaz de analizar el entorno. Para ello, se ha dispuesto un sistema sensorial compuesto por 3 fuentes sensoriales que recopilan estas necesidades.

2.4.1. Lidar

En los inicios de la robótica los principales sensores para la medición de distancias eran los sónares pero conforme la tecnología ha ido avanzando cada vez más se han reemplazando por los sensores Lidar gracias a su precisión y utilidad en gran cantidad de situaciones. Estos sensores son capaces de hacer un barrido horizontal de hasta 360°

y algunos un barrido vertical como el utilizado en este proyecto. Han sido utilizados tanto para detección de obstáculos, localización en el mundo o incluso para el mapeado de entornos, son muy interesantes por su buena relación entre rango y precisión, siendo algunos capaces de funcionar en exteriores de manera muy precisa.

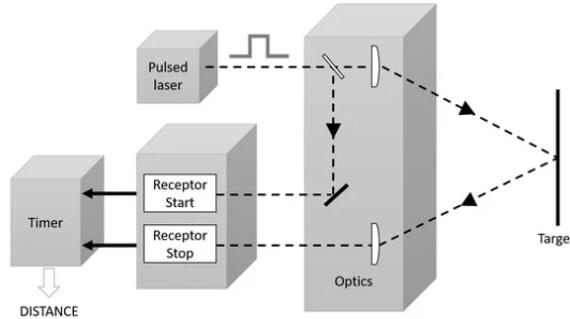


Figura 2.4: Esquema teórico de funcionamiento de un Lidar 1D, Fuente: Pyrois Tech

Estos sensores son agrupados con el nombre de "sensores de tiempo de vuelo" y es que su funcionamiento se basa en eso precisamente. Están dotados de un laser pulsado y un receptor junto con un sistema óptico de lentes y espejos 2.4, el láser emite un rayo de luz concentrada que viaja hasta el objeto a detectar, rebota en él y vuelve hasta ser detectado por el receptor del dispositivo, de esta manera conociendo que la velocidad de la luz es $c = 299,792,458\text{m/s}$ podemos definir la distancia al objeto (**D**) como:

$$D = c * \Delta t / 2$$

Si a esto le añadimos un motor eléctrico que haga rotar todo el dispositivo a gran velocidad alrededor de un eje obtenemos un barrido del entorno 2.5.

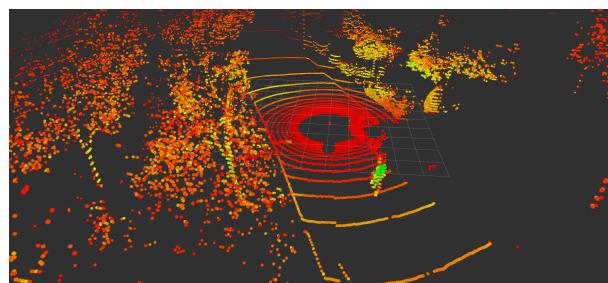


Figura 2.5: Nube de puntos de un Lidar 3D. Fuente: propia

2.4.2. GPS

El GPS es una herramienta bien conocida, estudiada y utilizada por una gran variedad de sectores, y aunque es una tecnología que funciona muy bien tiene sus limitaciones y problemas. Necesita una vista clara del cielo, lo cual no siempre es posible, las frecuencias a las que trabajan suelen ser muy bajas, del orden de unos pocos hercios, lo que dificulta conseguir una posición continua necesaria en la mayoría de casos para una correcta navegación.

Por otro lado, la precisión tampoco suele ser muy buena excepto que se utilicen tecnologías más avanzadas como son los GPS RTK (Real Time Kinematic, **figura 2.6**), que

mejora su error al añadir un protocolo de correcciones por medio de radio, modem o wifi en base a una estación fija de la que se conoce con gran exactitud sus coordenadas GPS, otorgando así hasta una precisión de unos pocos centímetros. Aún así, normalmente es muy necesario usarlo en conjunto con otros sensores, sean varios GPS, IMUs o con técnicas más avanzadas de localización como el conocido SLAM (SImultaneous Localization and Mapping), AMCL (Adaptative Montecarlo localization) o el utilizado en el presente proyecto, la fusión de fuentes de odometría basada en filtros extendidos de Kalman (EKF).

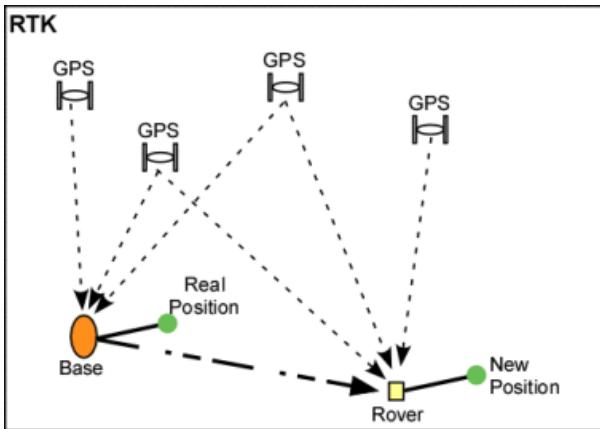


Figura 2.6: Esquema de tecnología RTK. Fuente: ResearchGate

2.4.3. IMU

Las *Inertial Motion Units* son un conjunto de sensores que normalmente existen en 2 configuraciones, 6 grados de libertad, donde encontramos un giroscopio y un acelerómetro o 9 grados de libertad, donde además se encuentra un magnetómetro.

En base a estos 2 o 3 sensores se puede calcular una orientación ya sea relativa desde el inicio del movimiento o absoluta respecto del norte magnético, estas unidades aunque son muy necesarias tienen una gran desventaja y es que son afectadas por un error acumulativo (error Abbe) [16], agregando cambios a la medición, lo que se conoce como deriva, esta deriva es muy perjudicial para conseguir una correcta localización por lo que al igual que comentamos anteriormente la fusión con otros sensores o técnicas de localización resulta muy necesaria.

CAPÍTULO 3

Desarrollo

3.1. Diseño del sistema sensorial

EL robot considerado en este trabajo fin de grado esta montado sobre una plataforma móvil adquirida a la empresa **Agilex**, figura 3.2, en concreto el modelo Hunter V2.0, un robot móvil tipo **Ackermann** muy resistente diseñado para cargas pesadas y escenarios de conducción precisos a baja velocidad, sus dimensiones son 980 x 745x 380mm, tiene una carga máxima de 150 kg y una velocidad máxima de 1.5 m/s, un radio de giro mínimo de 1.6 m y una autonomía de 22 km. El robot incluye un software para el cálculo de la cinemática inversa, la publicación de la odometría de los *encoders* de las ruedas por el topic */hunter/odom* y la suscripción de un mensaje tipo *sensor_msgs/Twist* para comandar velocidades angulares y lineales por el topic *cmd_vel*. La comunicación al ordenador de abordo se realiza mediante **Bus CAN**. En la figura 3.1 se pueden observar las especificaciones del vehículo completas. Si bien se ha usado este robot y su sistema sensorial se podría adaptar a cualquier otro de similares características.

HUNTER 2.0		AGILEX	
» SPECIFICATIONS			
Model	HUNTER 2.0	Suspension Form	Front Wheel Independent Suspension
Dimensions	980 x 745x 380mm W x H x D	Drive Form	Front-wheel Ackerman Steering Rear-Wheel Drive
Wheelbase	650mm	Working Temperature	-20-65°C
Track	605mm	Battery	24V30Ah (Standard) 24V60Ah (Optional)
Speed and Payload	6km/h, 150Payload (Standard) 10km/h, 80KG Payload (Optional) Customizable	MAX Travel (without loading)	22Km (24V30Ah Battery) 40Km (24V60Ah Battery)
Weight	65-72KG	Charger	AC 220V Charger Output 240W
Minimum Turning	1.6m	Charger Time	3.5h (24V30Ah Battery) 7h (24V60Ah Battery)
Climbing Ability	<10° With Loading	Outward Supply	24V15A Maximum total output current
Obstacle Surmounting Capacity	5cm Single-stage Right-angle Step	Code Disc Parameters	1024 Lines Electromagnetic incremental code disc
Minimum Ground Clearance	100mm	Motor	Drive 2x400W steering 400W Servo Motor
Minimum Braking Distance	0.2m 6km/h -> 0km/h (It depends on the ground conditions)	Communication Interface	Standard CAN 232 Serial Port
Steering Accuracy	0.5°	Protection Level	IP22(Customizable IP54)
Parking Function	Electromagnetic power-off parking, maximum 10° ramp parking(For parking only)		

Figura 3.1: Hoja de datos de la base del vehículo. Fuente: AgileX



Figura 3.2: Chasis del vehículo autónomo. Fuente: propia

El ordenador de abordo actualmente es un mini ordenador de la marca minis forum, un ordenador super ligero y versátil con altas prestaciones, incorpora Linux como sistema operativo, requisito fundamental para utilizar ROS2. El ordenador cuenta con una memoria RAM de 16MB y una memoria ROM SSD de 512GB, un procesador Ryzen 5 3550H de la marca AMD con una velocidad de 2.1GHz, 6 puertos USB-A, 1 puerto USB-C, conectores HDMI Y DP, 2 entradas para Ethernet y módulos WiFi y Bluetooth.

3.1.1. Lidar 3D Ouster

En primer lugar contamos con un LIDAR 3D de la marca Ouster, **figura 3.4**, en concreto el modelo de medio alcance, con un rango que se comprende entre los 0.4 y los 120 metros, está preparado para exteriores con una visera protectora y es capaz de barrer áreas en 360º horizontalmente y 45º verticalmente, funciona a 100Hz y tiene una resolución de 2 cm de media ya que depende de la distancia, todos estos datos se pueden comprobar también en la **figura 3.3**, para la implementación con ROS2 se usó el driver oficial de la empresa.

OPTICAL PERFORMANCE	
Range (80% Lambertian reflectivity, 2048 @ 10 Hz mode)	100 m @ >90% detection probability, 100 kix sunlight 120 m @ >50% detection probability, 100 kix sunlight
Range (10% Lambertian reflectivity, 2048 @ 10 Hz mode)	45 m @ >90% detection probability, 100 kix sunlight 55 m @ >50% detection probability, 100 kix sunlight
Minimum Range	0.3 m for point cloud data
Range Accuracy	±3 cm for lambertian targets, ±10 cm for retroreflectors
Precision (10% Lambertian reflectivity, 2048 @ 10 Hz mode, 1 standard deviation)	0.3 - 1 m: ± 0.7 cm 1 - 20 m: ± 1 cm 20 - 50 m: ± 2 cm >50 m: ± 5 cm
Range Resolution	0.3 cm
Vertical Resolution	32, 64, or 128 channels
Horizontal Resolution	512, 1024, or 2048 (configurable)
Field of View	Vertical: 45° (+22.5° to -22.5°) Horizontal: 360°
Angular Sampling Accuracy	Vertical: ±0.01° / Horizontal: ±0.01°

Figura 3.3: Hoja de datos del Ouster OS1-32-U. Fuente: Ouster



Figura 3.4: Ouster montado en la torre del vehículo. Fuente: propia

A parte del sensor láser este también tiene incorporado una IMU de 6 grados de libertad, 3 para un giroscopio y 3 para un acelerómetro, decir que esta unidad de medición inercial fue la única usada durante el 90 % del proyecto lo cual, dificultó mucho la obtención de una buena localización a causa no incluir un magnetómetro y de no ser de muy alta gama, de todas formas como se explica más adelante se consiguió una buena orientación usando este dispositivo. Fue colocado en la torre del robot para máxima visibilidad del Lidar.

En las **figuras 3.5 y 3.6** se observan las pruebas que se hicieron con el vehículo en el recinto de la universidad donde para el mismo sitio se puede ver la diferencia entre las salidas de barrido láser y nube de puntos en 3 dimensiones.

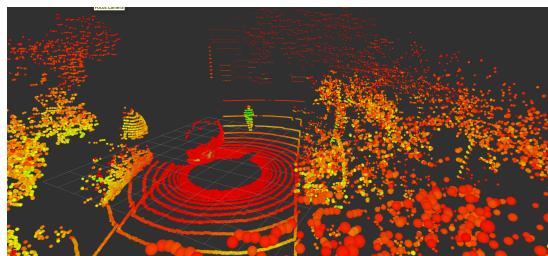


Figura 3.5: Salida de Lidar 3D en el recinto de la facultad de Telecomunicaciones. Fuente: propia

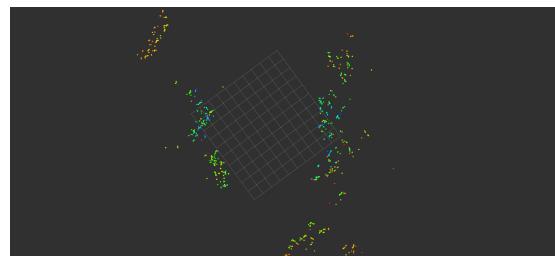


Figura 3.6: Salida de Lidar 2D en el recinto de la facultad de Telecomunicaciones. Fuente: propia

En cuanto al esquema de comunicación con el software, el driver usado publica los siguientes mensajes de datos.

- Lidar 2D, un mensaje de tipo *sensor_msgs/LaserScan* por el topic */ouster/scan*
- Lidar 3D, un mensaje de tipo *sensor_msgs/PointCloud* por el topic */ouster/points*
- IMU, un mensaje de tipo *sensor_msgs/Imu* por el topic */ouster imu*

3.1.2. GPS RTK Reach RS2+

El GPS utilizado fue un GPS Reach RS2+, **figura 3.7**, con tecnología RTK basado en el protocolo NTRIP para la corrección de posicionamiento. Este GPS funciona usando el protocolo NMEA, un protocolo extremadamente robusto utilizado en el sector marino, cuenta con una frecuencia máxima de 5Hz y una precisión de hasta 1 cm, **figura 3.8**. Para su montaje se creó un soporte con un perfil de aluminio al que se le atornilló el propio sensor, de manera que tenga una visión clara del cielo en todo momento y que a la vez no interfiera con la linea de corte del Lidar, su implementación en ROS2 también viene ampliamente documentada ya que el protocolo usado (NMEA) es muy conocido, para su conexiónado se pueden usar varios métodos, como Wifi o radio pero para este proyecto se implementó el serial por medio de USB para minimizar las interferencias y pérdidas de señal. Para la comunicación se tiene un solo mensaje de interés, este es de tipo *sensor_msgs/NavSatFix* y se publica por el topic */hunter/fix*.



Figura 3.7: GPS Reach RS2+.
Fuente: propia

REACH RS2+		
Technical specifications		
POSITIONING		
Precisione	Static	H: 4 mm + 0.5 ppm V: 8 mm + 1 ppm
	PPK	H: 5 mm + 0.5 ppm V: 10 mm + 1 ppm
	RTK	H: 7 mm + 1 ppm V: 14 mm + 1 ppm
Convergence time		
		-5 s typically
Signal tracked	GPS/QZSS L1C/A, L2C, GLONASS L1OF, L2OF, BeiDou B1I, B2I, Galileo E1-B, E5b	
Number of channels		
	184	
Update rate		
	Up to 10 Hz	
CONNECTIVITY		
UHF LoRa radio	Frequency range	868/915 MHz
Power	0.1 W	
Distance	Up to 8 km	
LTE modem		
Regions	Global	
Bands	FDD-LTE: 1, 2, 3, 4, 5, 7, 8, 12, 13, 18, 19, 20, 26, 28, 66 TD-LTE: 38, 40, 41	
UMTS (WCDMA/FDD)	1, 2, 3, 4, 5, 6, 8, 19	
Quad-Band (B50/1900, 900/1800 MHz)		
SIM card	Nano-SIM	
Wi-Fi	802.11 b/g/n	
Bluetooth	4.0/2.1 EDR	
Ports	RS-232, USB Type-C	
Data protocols	Corrections	NTRIP, RTCM3
	Position output	NMEA, LLH/XYZ
Data logging	RINEX at update rate up to 10 Hz	
Internal storage	16 GB	
MECHANICAL		
Ingress protection	IP67 water and dustproof	
Dimensions	126x126x142 mm	
Weight	950 g	
Operating temperature	-20 °C to +65 °C	
ELECTRICAL		
Autonomy	16 hrs as LTE RTK rover	
Charging	USB-C 5 V 3 A	
External power input	6–40 V	
Battery	LiFePO4 6400 mAh, 6.4 V	

Figura 3.8: Hoja de datos del GPS Reach RS2+.
Fuente: Emlid

3.1.3. IMU Xsens MTi-3th

El último sensor usado es una IMU de 9 grados de libertad de la marca Xsens (hoy en día conocida por Movella), **figura 3.9**, que cuenta con magnetómetro, **figura 3.10**. Este dispositivo es antiguo y por tanto no existía driver para ROS2, por tanto como paso preliminar se desarrolló un driver para su comunicación. El código de este driver está recopilado junto con el resto de algoritmos en el repositorio de GitHub creado para este trabajo.



Figura 3.9: Xsens 3th gen MTI.
Fuente: propia

Type name:	MTI		
Product ID:	MTI-28A53G35		
Device ID:	01301637 Dev Ver: 1.9.2.4.34		
Tested on:	30-Jun-2011		
Calibrated on:	30-Jun-2011		
Test Engineer Signature			
IMU Specifications	Accelerometer	Rate Gyro	Magnetometer
Full Scale:	50 [m/s ²]	300 [deg/s]	5 [a.u.]
Bandwidth [Hz]:	30	40	10
Default Sample Frequency [Hz]:	100		
Default Baudrate [bps]:	115200		
Basic test results	Accelerometer	Rate Gyro	Magnetometer
Noise:	0.008 [m/s ²]	0.005 [rad/s]	0.001 [a.u.]
Static accuracy residual:	0.475 [deg]	Temperature residual:	0.188 [deg]

Figura 3.10: Hoja de datos de la IMU MTi-3th.
Fuente: Xsens

En las **figuras 3.11 y 3.12** se puede observar el comportamiento de, por un lado la orientación y la velocidad angular en el eje z donde se puede comprobar una salida bastante estable y por otro lado el comportamiento de las aceleraciones, en este caso extremadamente ruidoso, también aquí se puede comprobar que la IMU está en el sistema de referencia correcto ya que \ddot{Z} debe ser positiva y tener de media un valor de aproximadamente 9.81 m/s^2 que es exactamente lo que se ve. En cuanto a la señal ruidosa de aceleraciones se verá más adelante que no es un problema dado que para utilizar esta señal en localización será integrada en el tiempo y por tanto simplemente servirá como una redundancia en la odometría de la velocidad lineal.

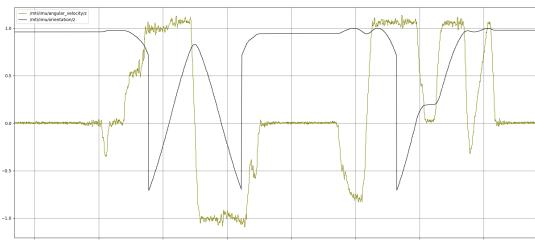


Figura 3.11: Variaciones de orientación y de velocidad angular en el eje z. Fuente: propia

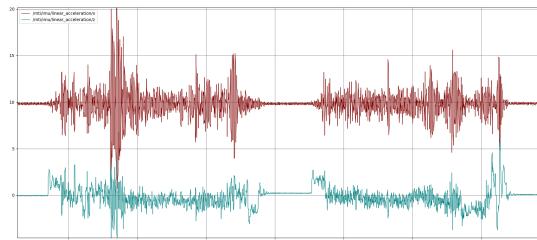


Figura 3.12: Variaciones de aceleración lineal en el eje x y el eje z. Fuente: propia

3.1.4. Implementación del sistema sensorial en ROS2

Una vez montados e instalados todos los dispositivos se hicieron pruebas para valorar su efectividad real en el terreno y la configuración de los parámetros adecuados para los drivers.

ROS2 funciona con un sistema de descripción cinemática y dinámica escrito en archivos URDF (Unified Robot Description Format), en concreto en el lenguaje **xml** y describe las transformaciones que existen entre los distintos componentes del robot, funciona como un árbol de transformaciones donde el elemento base sería el chasis, que según la **REP 105**

(ROS Enhancement Proposal) [17] el nombre para este elemento debe ser ”*base_link*”, está REP describe en detalle las convenciones respecto a nombres para todo el framework de ROS.

Las transformaciones pueden ser de 2 tipos, estáticas o dinámicas. Las estáticas las compondrán las transformaciones fijas entre los sensores y el ”*base_link*” y las dinámicas las transformaciones que pueden variar en el tiempo, como el cambio de pose desde un instante inicial al actual, el que según la **REP 105**, se debe llamar *odom*, que viene de odometría en inglés, otro ejemplo sería la transformación entre un mapa fijo definido a priori y la posición actual del robot.

Estos 2 flujos de datos son publicados por los *topics* */tf_static* y */tf*, respectivamente, para la creación del árbol estático de transformaciones, se miden las correspondientes distancias y se definen en un archivo xml como ya se ha comentado, finalmente se publican por los debidos topics. Estos árboles de transformaciones pueden contener más información como el tipo de movimiento respecto de el sistema de referencia, (fijo, de revolución acotada o de revolución continua), la geometría de los objetos y de sus colisión en el espacio o incluso de sus inercias, estas características aunque inútiles para la implementación en un robot real son muy útiles para hacer simulaciones precisas en motores de simulación como Gazebo o Coppelia (antiguo V-Rep), las transformaciones dinámicas se calcularán con uno de los paquetes usados, *robot_localization*, del cual se hablará en profundidad más adelante.

En la **figura3.13**, se puede observar el árbol completo del robot, desde las transformaciones dinámicas, (map, odom y utm) hasta las estáticas (base_link, gps_frame, mti_imu_frame, etc...)

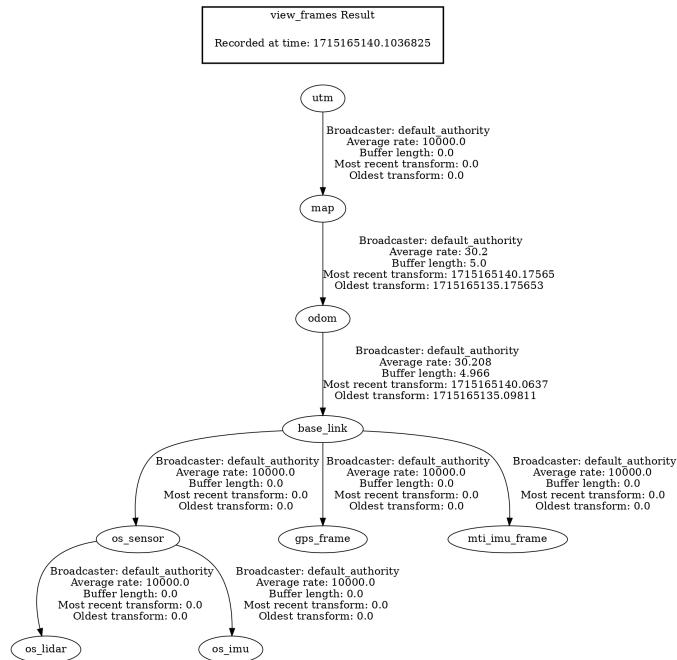


Figura 3.13: Árbol de transformaciones, Fuente: propia

3.2. Interfaz de usuario

Para la creación de la interfaz de usuario se decidió usar *React*, una biblioteca de Javascript de código abierto para la creación de interfaces de usuario, por ser una manera relativamente fácil de crear una interfaz web sin tener experiencia en este campo. Para ello y según los objetivos se hizo uso de una API de google maps para obtener un mapa junto con las herramientas (número 1 de la **figura 3.14**), que esta API incluyen, estas son, la posibilidad de mover waypoints y obtener las coordenadas GPS del mismo, establecer áreas, tanto rectángulos como polígonos o establecer polilíneas a modo de camino.

Para el protocolo de envío se escogió MQTT, por ser un protocolo rápido y sencillo de implementar, el procedimiento de uso es el que sigue, nada más conectarse a la web el protocolo de envío permanece inactivo, no es posible el envío de datos hasta que el usuario pulsa el botón de conexión (número 2 de la **figura 3.14**) con el broker MQTT, esto esta hecho para cerciorarse de que la conexión con el servidor broker se ha establecido correctamente, seguidamente se puede observar como dos botones más se activan ,**figura 3.15**, uno para publicar y otro para activar la funcionalidad del "follow me", (número 3 y 4 respectivamente en la **figura 3.15**).

El "follow me" o seguimiento dinámico de personas se describe como un protocolo por el cual el robot es capaz de seguir a una cierta distancia a un individuo, en este caso en base a la posición GPS proporcionada por un dispositivo móvil.

Finalmente, para indicar un waypoint al que ir se hace uso del marcador existente en pantalla (3 en la **figura 3.14**) y para crear un camino formado por múltiples waypoints se hace uso de la paleta de herramientas, (2 en la **figura 3.15**). Para finalizar, se hace uso del botón "publish"(3 en la **figura 3.14**), esto esta hecho para que se pueda comprobar que el envío de waypoint o waypointa es correcto antes de ser enviado.



Figura 3.14: primer estado de la interfaz.
Fuente: propia

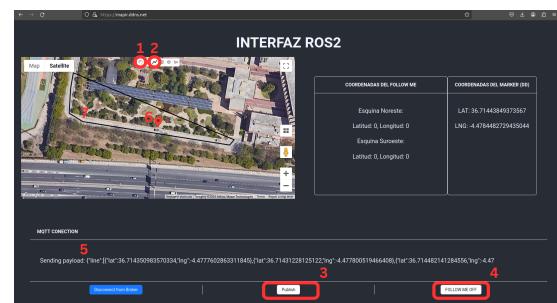


Figura 3.15: segundo estado de la interfaz.
Fuente: propia

La aplicación web está alojada en un servidor privado montado en una Raspberry pi 3B+, como software de servidor se ha hecho uso de apache2, un servicio de código abierto para implementación de servidores HTTP/1.1 y Mosquito como broker MQTT, ambos instalados y configurados en la propia Raspberry. De esta manera se abre la posibilidad de utilizar la página en cualquier lugar del mundo, también, se ha utilizado un servicio de DNS dinámico (DDNS) que permite a los usuarios asignar un nombre de dominio a una dirección IP dinámica, NO-ip en nuestro caso, de esta manera en cualquier lugar se puede abrir la aplicación y controlar el vehículo de manera visual e intuitiva. Finalmente comentar que también se desarrollado tanto el envío de areas rectangulares como polígo-

nales, ya que aunque esta funcionalidad no se ha implementado en el robot por no ser de interés para el propósito real de este robot, en los objetivos se estableció una interfaz ampliable y fácilmente actualizable.

La funcionalidad interna tanto de MQTT como de ROS2 es en base a *topics* o temas por los que se transporta la información, estos mensajes forman una cadena desde que se envían en la web hasta que se reciben en el ROS2.

Como se puede ver en el **esquema 3.16** se han implementado 3 topics, el primero y más importante */desired_pos*, por donde se envían las coordenadas del objetivo u objetivos, este mensaje formateado en JSON contiene principalmente 2 campos, el primero es el tipo que puede ser, "marker", "line", "rectangle", "polygon" o "follow" y el segundo campo contiene o una tupla con la latitud y longitud del objetivo o un array de tuplas para los casos más complejos, el topic */follow* es una *flag*, es decir un valor booleano para saber si el "follow me" está activado o no, el tercero es un mensaje al que se subscribe el servidor web con la información de la posición actual del robot.

Se ha hablado de la implementación de la interfaz y de su envío mediante un broker hasta el robot pero no de como se recibe esta información y se transforma a un mensaje útil para el entorno de desarrollo usado (ROS2), al enviar los mensajes formateados en JSON se deberán convertir en tipo *string*. En caso contrario, nuestro receptor no los entenderá. Este es un nodo, (MQTT Bridge), que se conecta con el broker establecido, se subscribe a los topics que se requieren y los transforma a un tipo de dato que ROS2 entiende, este es también un tipo *string* pero propio de ROS2, por tanto para obtener la información necesaria se deberá *parsear* ese mensaje transformándolo de un string de vuelta a una tupla de coordenadas o a un string de tuplas de coordenadas.

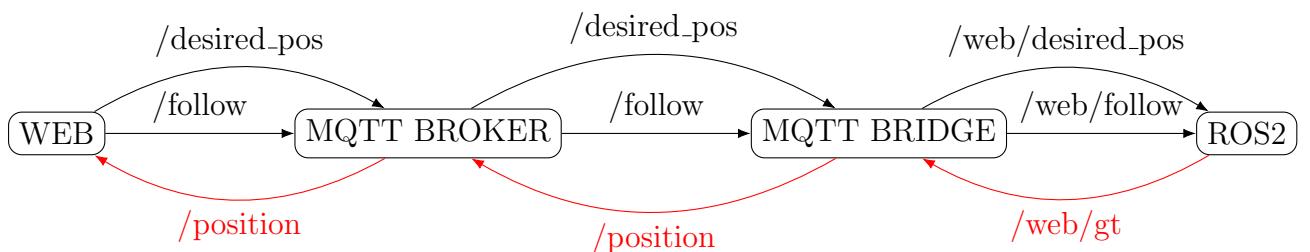


Figura 3.16: Esquema de flujo de datos desde la Web hasta ROS2

Para ello se creó un nodo que recibe la información del nodo MQTT Bridge y lo *parsea* haciendo uso de la librería *jsoncpp*, primero se lee el campo qué nos indica que tipo de mensaje es, es decir si es un marcador o es un conjunto de ellos, seguidamente lo guardamos en una variable con un tipo de dato personalizado, para los marcadores el tipo de datos contiene un campo para longitud y otro para latitud y para los caminos o conjunto de objetivos, contiene *array* de marcadores y un entero para indicar el tamaño, de esta manera se hace más sencillo el procesamiento de los mismos.

3.3. Procesamiento de los mensajes MQTT a ROS2

Por razones que se entenderán más avanzado el capítulo estos datos se deberán procesar todavía más, primero haciendo uso de un servicio de ROS2 implementado por el paquete *robot localization* que transforma las coordenadas GPS expresadas en ángulos a un sistema cartesiano centrado en el punto de inicio de movimiento del robot y después se deberá comprobar la distancia desde la posición actual del robot al punto que se desea alcanzar.

Esto es debido a la manera en la que Nav2 funciona, si la distancia es menor a un valor dictado por los parámetros de Nav2 se podrá enviar directamente al siguiente nodo, de no ser así se generarán puntos equiespaciados desde la posición del robot hasta el objetivo y posteriormente se enviará ese *array* de posiciones, cabe destacar que se deberá generar la orientación deseada, cosa que la web no proporciona.

Para ello simplemente se calculará el ángulo que existe entre la posición del robot y la de el objetivo, para el caso de tener varios objetivos contiguos, es decir una ruta o camino se hace este mismo algoritmo por cada par de puntos. El pseudo código se proporciona en el **algoritmo 1**.

Algoritmo 1 Procesamiento de objetivos

```

1: procedure PROCESAMIENTO(JSONmsg)      ▷ Cálculo final de trayectoria genérica
2:   tipo ← JSON.parse(JSONmsg)[“type”]
3:   if tipo = “marker” || tipo = “follow” then
4:     objetivo ← JSON.parse(JSONmsg)[“coordinates”]
5:     if dist(robot.position, objetivo) > rango then
6:       array_objetivos ← puntos_intermedios(robot.position, objetivo, rango)
7:       publish(array_objetivos)
8:     else
9:       publish(objetivo)
10:    end if
11:   else
12:     for p ← objetivos do      ▷ Se hace el mismo algoritmo por cada 2 puntos del
      array
13:       if dist( $p_0, p_1$ ) > rango then
14:         array_objetivos.push_back(puntos_intermedios( $p_0, p_1, rango$ ))
15:       else
16:         array_objetivos.push_back(objetivo)
17:       end if
18:     end for
19:     publish(array_objetivos)
20:   end if
21: end procedure
```

Una vez procesada la información obtenida de la Web, se publicará hacia otro nodo encargado de la lógica de control del paquete de navegación, este indicará a Nav2 donde ir y como ir, pero antes de ello se deberá conseguir una localización estable en el tiempo.

3.4. Localización y odometría

La localización en robótica es uno de los problemas más explorados dada su importancia, por ello esta parte posiblemente sea la más importante del proyecto y la que más tiempo llevo conseguir.

Primero se investigaron las mejores estrategias para conseguir una buena localización en exteriores, donde a diferencia de los espacios interiores esta tarea se vuelve un considerablemente más compleja. Existen métodos como SLAM o AMCL que funcionan muy bien en interiores pero en ambientes tan cambiantes y con tan pocas referencias estas alternativas se vuelven totalmente inservibles. Principalmente se tienen 2 fuentes de odometría (IMU, Encoders de las ruedas) y 1 fuente de localización (GPS), existen múltiples opciones para conseguir una buena localización en base a estos dispositivos pero la más usada es sin duda la utilización de filtros extendidos de **Kalman** para fusionar múltiples fuentes de odometría.

El **GNSS** (Global Navigation Satellite System) es una tecnología que funciona en base a una triangulación con satélites que estén a "la vista" en ese momento, normalmente esta posición es calculada por medio del estándar **WGS84**, siendo este el estándar más reciente creado para este propósito. A diferencia de muchos de sus antecesores este modelo tiene en consideración que la tierra es elíptica y no totalmente esférica. En la **figura 3.17** podemos ver el modelo WGS84 y sus sistema de referencia, el cual encontraríamos en el centro de la tierra con su eje **z** apuntando al Norte y su eje **x** apuntando al primer meridiano.

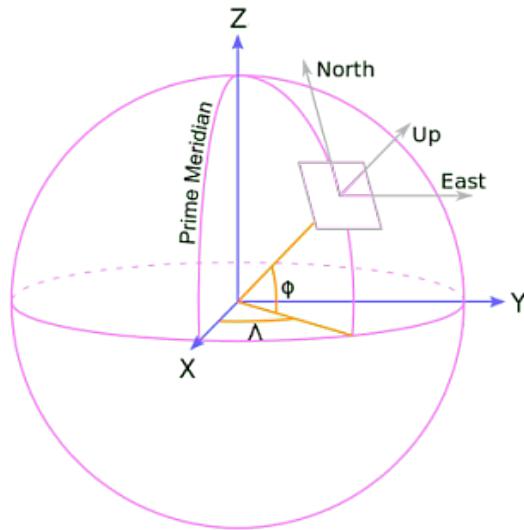


Figura 3.17: Sistema de referencia del modelo WGS84. Fuente: [11]

El principal problema de este sistema de referencia es que es bastante impráctico para la aplicación que aquí se discute, por ello, se requiere de una transformación a un sistema cartesiano y para ello se debe hablar también del sistema de coordenadas plano **UTM** (Universal Transverse Mercator), un sistema en base a la proyección de Mercator normal, pero en vez de hacerla tangente al Ecuador, se la hace secante a un meridiano y esta formado por 60 zonas que dividen el globo entero. En la **figura 3.18** se pueden observar tanto la nomenclatura de cada zona como su disposición.



Figura 3.18: Sistema de referencia del modelo UTM en Europa. Fuente: Wikipedia

A diferencia de las coordenadas geográficas este sistema se expresa en metros lo que ya da una ventaja en cuanto a esta aplicación pero sigue habiendo el problema de que estas zonas son demasiado grandes, por lo que también se necesita calcular el *offset* desde el origen de una de estas zonas a donde se inicie el robot, nos referiremos a este offset como *Datum*.

Como se puede observar en la **figura 3.19** a parte de conocer la posición, también se requiere del ángulo o *bearing* y la declinación magnética, esta última es la que menos preocupa, dado que es un dato bastante constante en el tiempo y varía por zona geográfica (varía alrededor de 0,035 radianes cada 100 años), para el caso de Málaga es igual a 0,061 rad.

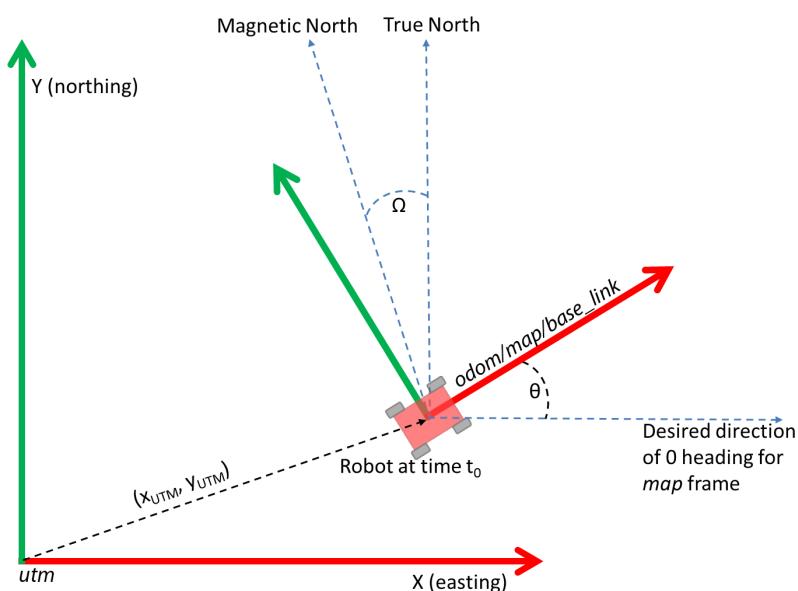


Figura 3.19: Sistema de referencia Datum. Fuente: [11]

Finalmente se cuenta con el paquete *robot localization* ya comentado, que ofrece una manera robusta de transformar los mensajes del GPS expresados en latitud y longitud (altitud se tomará igual a 0) a un sistema de referencia cartesiano, el nodo para esta tarea será *navsat_transform* y básicamente se suscribirá a 3 topics para calcular el *Datum* lo que hará una sola vez, para después en lo que se podría llamar el funcionamiento normal del nodo, publicará un mensaje de tipo *nav_msgs/Odometry* que será la transformación del GPS al sistema de coordenadas cartesianas. Para entender mejor el flujo de datos se incluye un esquema en la **figura 3.20** y una explicación de las 3 suscripciones que necesitan para trabajar.

- Un mensaje de tipo *sensor_msgs/NavSatFix* con los datos provenientes del GPS, este será */hunter/fix*
- Un mensaje de tipo *nav_msgs/Odometry* con la posición actual del robot y opcionalmente la orientación, está odometría será la salida de uno de los filtros de Kalman, esta información es necesaria en el caso de que el primer mensaje de GPS llegue después de que el robot haya empezado a moverse, este será */odometry/global*.
- Para este tercer mensaje se tiene que proporcionar la orientación global respecto al norte magnético y debe usarse la convención ENU (East, North, Up), que estipula un sistema de referencia con el Este en el eje x y el Norte en el eje y como se puede observar en la **figura 3.17**. Para ello el nodo ofrece 4 maneras de hacerlo, la primera ya ha sido comentada y es usando la parte de orientación del primer mensaje de odometría y los otros 3 se listan a continuación.
 - Una suscripción a un topic de tipo *sensor_msgs/Imu* con 9 grados de libertad, (ya que sin ellos no tenemos una orientación global respecto al norte magnético).
 - Utilizar uno de los servicios que ofrece el paquete para que de manera externa ese datum sea calculado y enviado, este servicio es */datum*.
 - Indicarlo manualmente por medio de un archivo de parámetros en formato *YAML*.

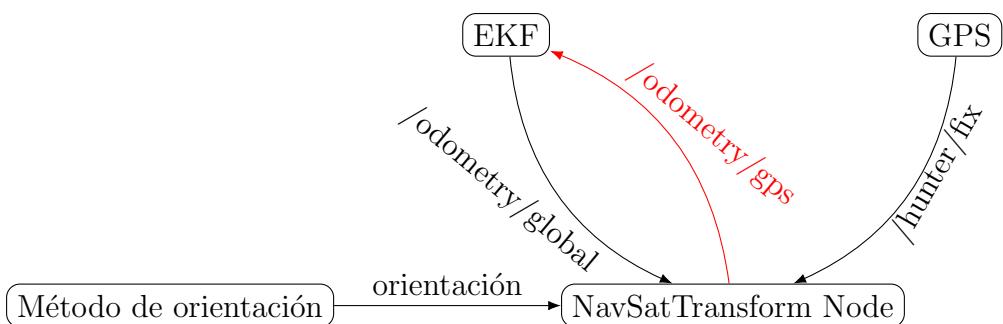


Figura 3.20: Esquema de flujo de topics en el nodo *navsat_transform*

Al comienzo del proyecto como ya se ha comentado no se disponía de una IMU con orientación global por lo que se desarrolló un nodo que entre otras cosas calcula el ángulo respecto al norte magnético y lo transforma al sistema ENU antes de enviarlo por medio del servicio que *robot_localization* proporciona. Para realizar este cálculo primero el nodo se suscribe al topic */hunter/fix* para obtener así las coordenadas GPS y al topic */hunter/odom* de tipo *nav_msgs/Odometry* para recibir la información del movimiento

del robot, también, publicará por el topic `/cmd_vel` para hacer mover al robot. El procedimiento es el que sigue.

Primeramente se permanece a la espera de la llegada de un mensaje de GPS y se guardan esas coordenadas, posteriormente se hace mover al vehículo una determinada distancia hacia delante, es decir solo en el eje **x** que será especificada por el usuario, finalmente se volverá a esperar a otro mensaje del GPS. Por tanto para la parte de posición se tomará el segundo dato de GPS y para el cálculo de la orientación global, denotada como bearing o azimut se seguirán las siguientes ecuaciones que describen el cálculo del rumbo (bearing) entre dos puntos geográficos, dados por sus coordenadas de latitud (ϕ_1, ϕ_2) y longitud (λ_1, λ_2):

$$\Delta l = (\lambda_2 - \lambda_1) * (\pi/180) \quad (3.1)$$

$$X = \cos(\phi_2) * \sin(\Delta l), \quad (3.2)$$

$$Y = \cos(\phi_1) * \sin(\phi_2) - \sin(\phi_1) * \cos(\phi_2) * \cos(\Delta l) \quad (3.3)$$

$$\text{bearing} = (\pi/2) - \text{fmod}((\text{atan2}(Y, X) + 2\pi), 2\pi) \quad (3.4)$$

Donde:

- Δl es el cambio en longitud, medida en radianes.
- X es una componente del cálculo intermedio.
- Y es otra componente del cálculo intermedio.
- bearing es el rumbo entre los dos puntos, medido en radianes.

En la **ecuación 3.4**, se puede observar el uso del módulo del ángulo respecto a 360^0 , esto es debido a que la función `atan2` devuelve un valor en el rango $[-\pi/2, \pi/2]$, y en cambio se necesita en el rango $[0, 2\pi]$, también como se ha comentado este ángulo debe estar acorde al sistema ENU y por tanto se le resta $\pi/2$ para que esto se cumpla. Finalmente se enviará al servicio y el nodo `navsat_transform` comenzará a funcionar.

Por último falta la configuración de los EKF, para esta implementación se tomará también el mismo paquete que nos ofrece la posibilidad de configurarlos de una manera sencilla.

El principal problema que se considera aquí es la diferencia entre la odometría y la localización, donde la primera es muy rápida, (más de 30 Hz) y precisa para intervalos de tiempo cortos, es decir acumula error con el tiempo. Por otro lado la localización proporcionada por el GPS es muy lenta y por tanto discreta, por esta razón no vale para usar en intervalos cortos de tiempo pero es extremadamente precisa a la larga ya que es una fuente de posición que por definición no tiene *deriva* en el tiempo, concluyendo, se necesitarán las dos para conseguir una buena localización.

Para ello se empezará creando una instancia de un filtro de Kalman para las fuentes continuas (Odometrías), este se llamará `ekf_local` ya que se usará para navegación y planificación de movimientos en el sistema de referencia `/odom`, estos filtros funcionan con matrices de 15 variables para cada fuente de odometría, estas son:

$$[X, Y, Z, \theta_x, \theta_y, \theta_z, \dot{X}, \dot{Y}, \dot{Z}, \dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z, \ddot{X}, \ddot{Y}, \ddot{Z}]$$

CAPÍTULO 3. DESARROLLO

Se asumirá que se está trabajando en 2 dimensiones (por ser un robot terrestre y Ackermann hay mucho movimientos que se pueden ignorar), de esta manera las variables se reducirían a: $[X, Y, \theta_z, \dot{X}, \dot{Y}, \ddot{\theta}_z, \ddot{X}, \ddot{Y}]$.

Pero se puede ir más allá, primero teniendo en cuenta el hardware, no todos los dispositivos proporcionan todas las variables, y eso está bien no se necesitan, sino que para que el filtro funcione correctamente y no empiece a oscilar descontroladamente, se requiere que entre todas las fuentes se disponga de las variables necesarias.

Estas, variarán mucho según cada situación, como por ejemplo los sensores de los que se dispongan, su precisión, el tipo del robot o incluso el entorno en el que el robot se mueva, esta tarea puede volverse sorprendentemente compleja y por ello se han desarrollado una serie de "normas" que indican que variables introducir y como. Estas normas han sido fruto de múltiples fuentes de información como la documentación oficial del paquete usado y la charla sobre el mismo dada por su creador en la ROSCon (ROS conference) del 2015, exhaustiva investigación por internet de múltiples usuarios y por último muchas horas de pruebas en el terreno con el propio robot. Otro dato importante a resaltar es que estas 5 normas tiene una prioridad, esto quiere decir que si alguna se contradice siempre se debe priorizar a la siguiente.

1. Siempre que se pueda se debe introducir variables directamente medidas, esto significa que si por ejemplo se posee un sensor que mide la velocidad de las ruedas y en base a eso calcula su posición, una mejor práctica es introducir la velocidad.
2. Para que el filtro sea estable, se necesita como mínimo introducir una *pose* completa, que para el presente caso de robot tipo Ackermann sería, $[X, Y, \theta_z]$, si esto no es posible entonces se deberá proporcionar sus respectivas velocidades.
3. Si se tiene una fuente de velocidad lineal y posición, una mejor práctica es proporcionar la velocidad lineal y por otro lado si esa fuente proporciona velocidad angular y orientación la mejor práctica es introducir la orientación.
4. Las matrices de covarianza que incluyen los mensajes que se proporcionan al filtro son extremadamente importantes ya que indican cuando usar una variable o cuando usar otra. Nunca se debe introducir una variable donde su varianza no haya sido calculada o esta sea 0, tampoco se deben introducir 2 fuentes de la misma variable (solo para el caso de orientación y posición) que tengan el mismo o muy parecido valor de varianza, ya que esto causará un rápido cambio entre los dos valores que puede llegar a una oscilación indeseada en la salida del filtro.
5. Se deberán analizar las restricciones de el tipo de robot usado en particular, (Ackermann, diferencial, omnidireccional, etc...). Para el presente caso de robot tipo Ackermann, no puede tener un cambio instantáneo en el eje y y por tanto \dot{Y} siempre va a ser 0, se posría pensar que no se debería introducir esta variable ya que resulta innecesaria, pero esto no es así ya que introduciendo este valor se le "indica" al filtro que este no se puede desplazar en la dirección y, el mismo planteamiento se podría hacer para para \ddot{Y} , pero dado que los valores de esta variable suelen venir de una IMU y sabiendo que estas normalmente proporcionan una cantidad de ruido demasiado grande incluso al estar en reposo, la mejor práctica en este caso en no proporcionarla.

Dadas estas normas se llegó a la siguiente configuración. Para la odometría de las ruedas

se introdujo $[\dot{X}, \dot{Y}, \dot{\theta}_z]$ y para la IMU $[\theta_z, \dot{\theta}_z, \ddot{X}, \ddot{Z}]$. De esta manera no solo cumplimos con todos los criterios mencionados sino que tenemos duplicidad en múltiples variables lo que hace al filtro mucho más robusto. Este filtro a parte de tener como salida un publicador de un mensaje tipo *nav_msgs/Odometry* con las entradas fusionadas también proporciona la transformación de */odom* a */base_link* quedando así un paso más cerca de conseguir todo el árbol de transformaciones como se puede visualizar en la **figura 3.13**.

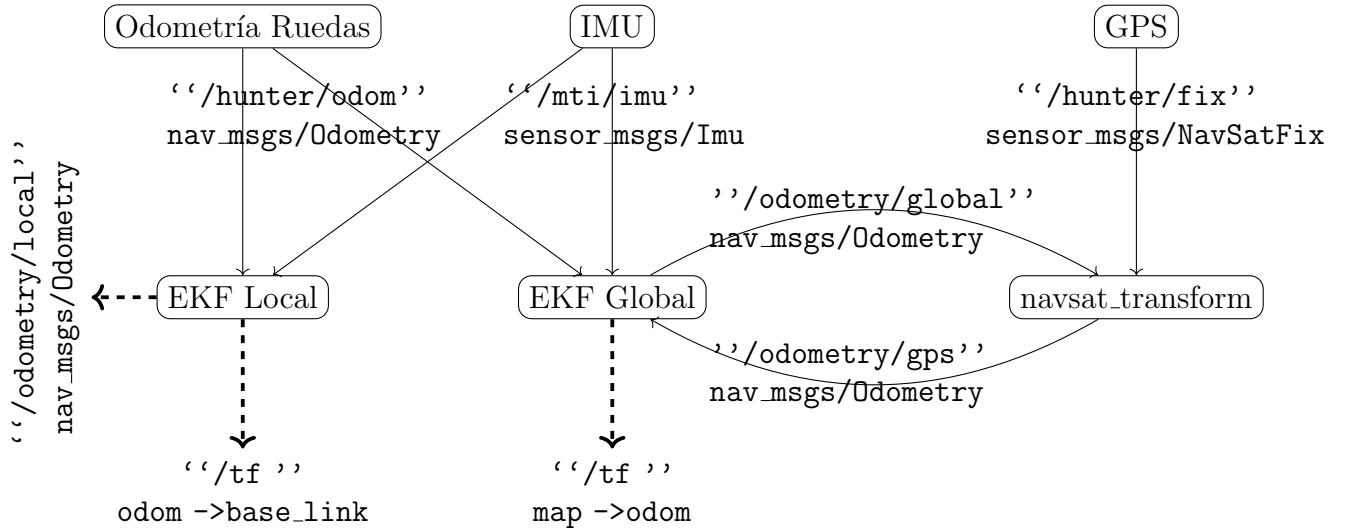


Figura 3.21: Esquema de topics completo sobre el funcionamiento de la localización

Para el caso de las entradas de fuentes de odometría de este filtro se utilizarán las mismas con la misma configuración que para el *ekf_local*, pero con la diferencia de que ahora se añadirá la salida del nodo *navsat_transform*, siendo esta */odometry/gps* como se ilustra en la **figura 3.21**, para esta fuente de datos se añadirá si o si las componentes x e y ya que son las únicas de interés, estás nos darán una localización que se mantenga en el tiempo pero con saltos discretos, por otro lado este filtro también publicará la transformación */map* a */odom* consiguiendo así un árbol de transformaciones completo.

3.5. Mejoras en la localización para la orientación del sistema

Uno de los problemas más complejos que se tiene en cuanto a localización se refiere es la orientación, así como se considera aceptable acumular un error de 1m al cabo de algún tiempo, si el mismo robot acumula 1 radian de error en θ_z resultaría inaceptable, también hay que imaginarse que un error en posición se acumulará solo en posición pero un error en orientación también afectará a la posición, todo esto sumado a que las fuentes de orientación que se manejan con las IMU no son extremadamente precisas resultan en el problema principal de la localización.

Para solucionar este problema se han tomado múltiples medidas, primero con el cálculo de las varianzas de manera que minimize los errores en el filtro de Kalman ya que este es extremadamente dependiente de la matriz de covarianzas como se explica en el apartado de **pruebas y resultados**. Segundo, haciendo uso de las dos IMU que se disponen en el

sistema sensorial, para ello la IMU de 9 grados de libertad se puede introducir de manera directa al filtro con la selección de variables que se indican en esta misma sección ya que el propio sensor posee un procesado de las señales, para el caso de la IMU integrada en Lidar de 6 grados de libertad que a parte de tener una cantidad considerable de ruido no proporciona orientación por lo que para su uso en el proyecto se decidió implementar otro filtro anterior a los dos EKF, este es el filtro "complementario" que se podría describir como una versión básica del filtro de Kalman sin el análisis estadístico que este último tiene.

Este filtro se considerá como un filtro pasa baja para el acelerómetro y un filtro pasa alta para el giroscopio [18], este filtro es capaz de suscribirse al mensaje de una IMU con giroscopio y acelerómetro y calcular en base a ellos la orientación, en el caso más básico su implementación conlleva el cálculo de la orientación siguiendo la **ecuación 3.5**, calculado en base a 2 constantes (A y B) que son ajustadas de manera manual.

$$\theta = A * (\theta_{t-1} + \theta_{gyro}) + B * \theta_{accel} \quad (3.5)$$

Para la implementación en el sistema actual se decidió usar una versión que calcula estas constantes de manera automática, a parte de calcular también una estimación del llamado 'bias', esto se refiere a una desviación constante en las lecturas del sensor, siendo este causado por diversas razones, como la temperatura del sensor, imperfecciones en su construcción o las variaciones en el suministro eléctrico, este filtro asume un periodo de reposo al inicio de su ejecución donde se promedian las lecturas y se saca una estimación inicial del 'bias'.

Finalmente se hace pasar por el nodo *generador de Datum* del que ya se ha hablando en este documento donde, se añadirán manualmente las varianzas escogidas para el sensor, para después introducirlo como entrada en los filtros de Kalman. Como se están introduciendo 2 variables iguales y siguiendo la norma número 4, no se debe introducir directamente esta entrada ya que provocaría que el filtro cambiase su salida de manera oscilatoria entre las 2 entradas de orientación (las 2 IMU). Afortunadamente el nodo de implementación de los filtros de Kalman proporciona también una configuración llamada "modo diferencial", esta lo que hará será transformar las entradas de orientación y posición para esa fuente de odometría a velocidad, de manera que no interfieran las unas con las otras. Hay que tener cuidado con este parámetro ya que, si mal utilizado, puedo provocar que el filtro acabe oscilando, por tanto la norma que se sigue es utilizarlo solo cuando hay más de una fuente que proporciona esa variable en concreto y activarlo para N-1 fuentes, donde la que se quede sin activar sea la entrada de mayor precisión (siendo ésta la IMU de Xsens de 9 grados de libertad).

Con esto se tendría solucionado el problema de la localización y así se podría pasar a la realización e implementación del "stack" de navegación.

3.6. Navegación Autónoma haciendo uso de Nav2

El stack de navegación de ROS2 está diseñado principalmente para interiores donde ha sido probado y testado, pero con algunos ajustes y modificaciones se puede usar en cualquier entorno, este funciona en base a unas estructuras lógicas llamadas *behavior trees*

[13], estas estructuras son muy versátiles por su manera de funcionar dado que resulta muy sencillo modificar y ampliar el comportamiento del stack de navegación. Los árboles de comportamiento están formados por nodos, formalmente llamados, nodos "internos" (o de control) y nodos "hoja" (o de ejecución), para entender el funcionamiento se deberá usar la terminología de nodo padre y nodo hijo. Los nodos de control tienen al menos un nodo "hijo" y gráficamente se expresan como se ilustra en la **figura 3.22**, estos aparecen por debajo de los nodos "padre".

3.6.1. Conceptos básicos de Nav2

Antes de empezar a desarrollar el sistema de navegación es conveniente familiarizarse con los conceptos clave que se utilizan en Nav2. Estos conceptos son la clave fundamental para poder hacer un uso correcto del "stack" y así sacar el máximo partido a la implementación de su código.

Árboles de comportamiento

Un Árbol de comportamiento empieza su ejecución desde el nodo "raíz" y genera una señal que se propaga por sus nodos "hijo" a una frecuencia dada, esta señal se conoce como *tick*, cuando un nodo recibe un *tick* este puede devolver *Running* si está bajo una ejecución, *Success* si la ejecución ha resultado satisfactoria o *Failure* si ha resultado en fallo, conocidos estos conceptos podemos pasar a los tipos de nodos que existen y sus características de funcionamiento. Estos se agrupan en 4 categorías que se exponen a continuación.

- Acción: Estos nodos como su nombre indica ejecutan acciones determinadas sobre el stack de navegación o sobre el robot en cuestión, estas acciones pueden ser desde borrar la memoria interna de mapas de coste hasta hacer que el robot retroceda para recalcular su ruta.
- Condición: Estos nodos se usan para decidir si ejecutar una acción u otra, se podría ver como condicionales *if else* en programación, algunos ejemplos pueden ser detectar cuando se ha llegado al objetivo o detectar si el objetivo ha cambiado.
- Decoradores: Estos nodos modifican ciertos comportamientos internos de la lógica, como puede ser cambiar la frecuencia de ejecución de una determinada sección o forzar al actualización del objetivo.
- Control, Los nodos de control son los más importantes ya que proporcionan bucles y condicionales específicos para generar algoritmos, estos son 3 para el caso de Nav2:
 - *Pipeline*, "llamará" sus nodos "hijo" uno por uno de izquierda a derecha esperando a que cada nodo devuelva *Failure* o *Success*, este nodo solo devolverá *Success* cuando todos sus nodos devuelvan *Success*, en el caso de que un nodo devuelva *Failure* este parará su comportamiento y devolverá *Failure*.
 - *Recovery*, un nodo de control propio del stack, tendrá exclusivamente 2 nodos "hijo" como se puede observar en la **figura 3.23** y solo devolverá *Success* si el primer nodo lo devuelve, en caso de que el primer nodo devuelva *Failure* "llamará" al segundo nodo y volverá a intentar "llamar" al primero en un

bucle hasta que el primer nodo devuelva *Success* o se hallan ejecutado un nº determinado de intentos estipulados en el parámetro *number_of_retries*.

- *Round Robin*, este nodo funciona en oposición a *Pipeline* donde "llamará" en bucle a todos los nodos de izquierda a derecha hasta que alguno de ellos devuelva *Success*, donde el también devolverá *Success* sin importar el resto de nodos "hijo".

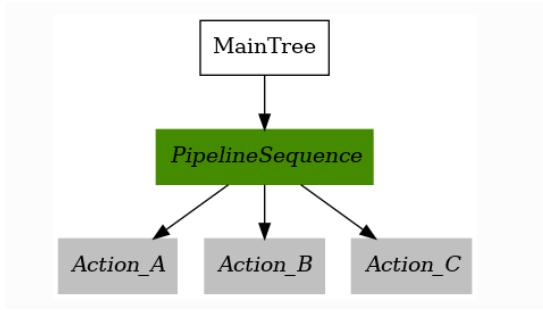


Figura 3.22: Estructura de un Árbol de comportamiento básico (Pipeline). Fuente: [11]

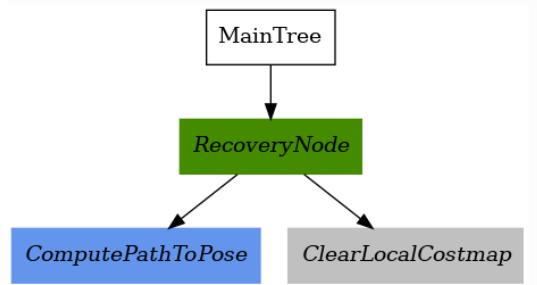


Figura 3.23: Estructura del nodo de control "Recovery". Fuente: [11]

Servidores de acción

Otro concepto clave del stack son los servidores de acción o *action servers* de ROS2, este concepto es fundamental para entender como funciona Nav2, estos "servidores de acción" cuentan con un *cliente* que hace la petición y con un *servidor* que la realiza y la **controla**, siendo esto último con lo que se diferencian de los servicios de ROS2, en los servidores de acción, el servidor proporciona un flujo constante de mensajes sobre el topic *feedback* durante el proceso de ejecución y además tienen la capacidad de ser cancelados en cualquier momento, como se puede ver en la **figura 3.24** estos servidores generan 3 flujos de datos, el primero un servicio de petición que asegura la llegada de información al servidor, un topic de feedback que devuelve la información durante el proceso y un tercero con un servicio de resultado que devuelve el resultado final de ejecución.

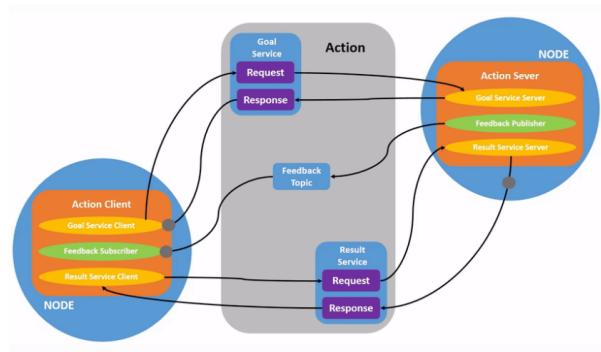


Figura 3.24: Estructura interna de un servicio de acción o *action server*. Fuente: [10]

Estructura para el control de navegación

En Nav2 todas las funcionalidades están implementadas en forma de servidores de acción y funcionan con una lógica descrita en un árbol de comportamiento totalmente modificable.

Estos servidores son 5, **figura 2.3**, *Navigator server* que es el encargado de "leer" el árbol de comportamiento proporcionado por el usuario y ejecutar las diferentes acciones, *Controller server* donde se configura el controlador o "planificador local", *Planner server* para configurar el planificador o "planificador global", *Behavior server* para la selección y configuración de los comportamientos de recuperación usados cuando el robot se queda "atascado" y por último el *Smoothen server* para procesar el camino generado y suavizarlo.

Posteriormente existen 2 mapas de coste, uno local y otro global, el local se encarga de almacenar la información en un entorno reducido y sirve para que el controlador haga uso de una buena evitación de obstáculos y el global para que el planificador genere una trayectoria evitando los obstáculos ya conocidos a más largo alcance.

Finalmente se utilizan 2 plugins externos a la estructura de Nav2, *Velocity smoother* que suaviza los comandos de velocidad para generar movimientos continuos y *Waypoint follower* que proporciona la funcionalidad de navegar un array de poses aunque estos estén a mayor distancia del alcance que tiene el mapa global, esto lo realian llamando al algoritmo de navegación cada vez que llega a un objetivo, cabe mencionar que las poses deben tener una separación máxima igual o menor al alcance del mapa global para que así Nav2 encuentre siempre un camino.

3.6.2. Servidores

Navigator server

Para el caso que aquí se expone proporcionaremos un árbol que contiene la lógica para la navegación punto a punto, en la **figura 3.25** observamos un ejemplo proporcionado por los creadores del stack para la navegación punto a punto y en la **figura 3.26** vemos el mismo ejemplo escrito y modificado en xml para ser usado en el proyecto.

La funcionalidad básica de esta lógica esta descrita en 2 sub arboles divididos por un nodo de control de tipo "recovery", la primera parte intenta navegar y la segunda intenta salir de situaciones donde la navegación se imposibilita como esquinas o la aparición de obstáculos dinámicos. En la primera parte "computa" el camino con ayuda de el "planificador" y si lo consigue calcular lo sigue con ayuda del controlador, en caso de que no pueda calcularlo o seguirlo se procede a borrar el mapa de coste global o local respectivamente.

En el caso de que esto falle se procede al sub árbol de recuperación que incorpora en nuestro caso 2 comportamientos para intentar salir de esa situación de imposibilidad de acción, primero borra los 2 mapas de coste y posteriormente con ayuda de un nodo del tipo "Round Robin" procede a realizar una parada de 5 segundos (en el caso de que sea una persona andando soluciona el problema de manera rápida) y después se mueve hacia atrás 0.15 m, este procedimiento lo intenta 6 veces hasta que se cancela por completo la acción de navegación.

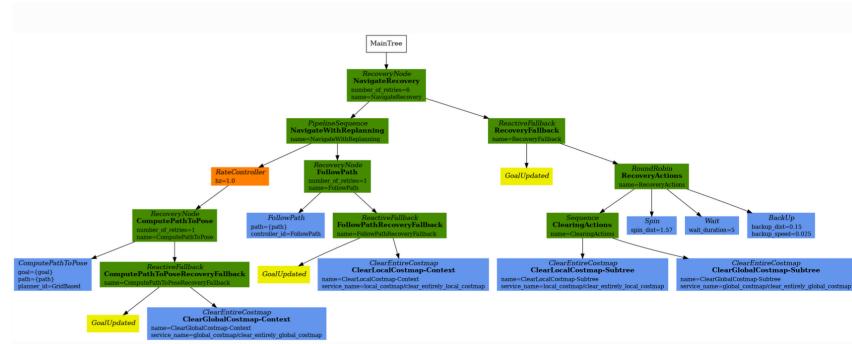


Figura 3.25: Esquema de navegación punto a punto. Fuente: [11]

```

<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <PipelineSequence name="NavigateWithReplanning">
        <RecoveryNode number_of_retries="1" name="ComputePathToPose">
          <ComputePathToPose goal="{}" path="{}" planner_id="GridBased"/>
          <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name="global_costmap/clear_entirely_global_costmap"/>
        </RecoveryNode>
        <RateController>
          <RecoveryNode number_of_retries="1" name="FollowPath">
            <FollowPath path="{}" controller_id="FollowPath"/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_costmap/clear_entirely_local_costmap"/>
          </RecoveryNode>
        </RateController>
        <ReactiveFallback name="RecoveryFallback">
          <GoalUpdated/>
          <RoundRobin name="RecoveryActions">
            <Sequence name="ClearingActions">
              <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_costmap/clear_entirely_local_costmap"/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name="global_costmap/clear_entirely_global_costmap"/>
            </Sequence>
            <Wait wait_duration="5"/>
            <Backup backup_dist="0.39" backup_speed="0.05"/>
          </RoundRobin>
        </ReactiveFallback>
      </PipelineSequence>
    </RecoveryNode>
  </BehaviorTree>
</root>
  
```

Figura 3.26: Código del árbol de comportamiento para la navegación punto a punto. Fuente: propia

Behavior server

Como se ha comentado este servidor cuenta con la posibilidad de configuración de varios comportamientos en caso de que se necesite una recuperación. Los que se han usado en el presente proyecto son *clear costmap* que borra el mapa de coste que se seleccione, este comportamiento es muy útil cuando aparecen obstáculos dinámicos como podría ser una persona que pasa andando delante del vehículo, ya que borrando el mapa de costes forzamos a recalcularlo y evitamos tener que retroceder para encontrar otro camino, el comportamiento de *Wait* hace detenerse al vehículo por un determinado tiempo, esto como el anterior proporciona una solución más segura para el caso de obstáculos en movimiento, el último comportamiento es *Back up* que simplemente hace al robot dar marcha atrás y así conseguir más espacio para recalcular la trayectoria.

Controller server

Como ya se ha comentado en esta memoria, el "planificador local" es llamado controlador y es el encargado de hacer que el robot se mueva y de que si aparecen obstáculos en el camino, sean dinámicos o no los esquive de manera segura, para la configuración de este servidor se han escogido dos plugins que controlan el progreso hacia el objetivo y que detectan la llegada al mismo y también se ha escogido un algoritmo para el propio controlador que sea compatible con un robot Ackerman, esto quiere decir que tenga en consideración en radio mínimo de giro. También se ha tenido en cuenta que sea rápido.

Nav2 nos proporciona varias implementaciones para controladores como se puede ver

en la **figura 3.27**.

Controllers			
Plugin Name	Creator	Description	Drivetrain support
DWB Controller	David Lu!!	A highly configurable DWA implementation with plugin interfaces	Differential, Omnidirectional, Legged
TEB Controller	Christoph Rösmann	A MPC-like controller suitable for ackermann, differential, and holonomic robots.	Ackermann, Legged, Omnidirectional, Differential
Regulated Pure Pursuit	Steve Macenski	A service / industrial robot variation on the pure pursuit algorithm with adaptive features.	Ackermann, Legged, Differential
MPPI Controller	Steve Macenski Aleksei Budylakov	A predictive MPC controller with modular & custom cost functions that can accomplish many tasks.	Differential, Omni, Ackermann
Rotation Shim Controller	Steve Macenski	A "shim" controller to rotate to path heading before passing to main controller for tracking.	Differential, Omni, model rotate in place
Graceful Controller	Alberto Tudela	A controller based on a pose-following control law to generate smooth trajectories.	Differential

Figura 3.27: Tabla de controladores disponibles para Nav2. Fuente: [11]

Lo primero que se hizo fue seleccionar los controladores que fuesen compatibles con el robot, estos son **TEB**, **Regulated Pure Pursuit** y **MPPI**, después de una exhaustiva investigación se llegó a la decisión de probar el MPPI (Model Predictive Path Integral) por su gran versatilidad en cuanto a configuración, este es un modelo predictivo y una variante del algoritmo **MPC**, este algoritmo es mayoritariamente usado en robots que funcionan a gran velocidad ya que predice la posición del robot en un "futuro", al probarlo se descartó por dos razones, la primera fue la gran cantidad de parámetros que necesitan ser retocados cuidadosamente para que su configuración lleve a un buen resultado, se llegarón a observar ciertas situaciones en las que no era fiable usarlo, la predicción de estados funciona muy bien para casos donde no hay obstáculos o estos se avistan a gran distancia pero en el proyecto resultaba poco fiable, la segunda razón es que este algoritmo necesita de una capacidad de cálculo muy grande y se decidió que no compensaba para las muchas funcionalidades que este posee y las pocas que se usaban en el proyecto.

Por tanto se probó la segunda opción, el algoritmo **Regulated Pure Pursuit**, una solución mucho más simple pero que daba salida al problema de manera estable, este algoritmo funciona siguiendo unos *objetivos virtuales*, **figura 3.28**, pertenecientes a el trayecto a seguir, que se van desplazando conforme el robot se acerca, manteniendo una distancia o *look ahead* que puede ser fija o recalculada dinámicamente. En la **figura 3.29** se pueden observar los parámetros escogidos para el algoritmo donde entre ellos se encuentran los parámetros típicos de configuración como el *lookahead_distance* o el radio de giro para las "incorporaciones" al trayecto.

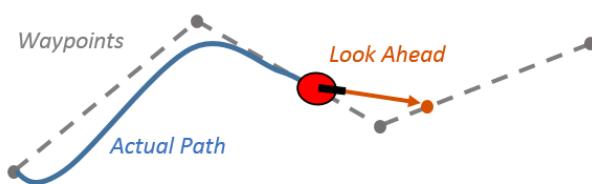


Figura 3.28: Funcionamiento del algoritmo Pure pursuit. Fuente: MathWorks

CAPÍTULO 3. DESARROLLO

```

FollowPath:
plugin: "nav2_regulated_pure_pursuit_controller::RegulatedPurePursuitController"
desired_linear_vel: 0.60
lookahead_dist: 1.0
min_lookahead_dist: 0.5
max_lookahead_dist: 1.5
lookahead_time: 1.0
transform_tolerance: 0.4
use_velocity_scaled_lookahead_dist: false
min_desired_linear_velocity: -0.35
approach_velocity_scaling_dist: 1.0
use_collision_detection: false
max_allowed_time_to_collision_up_to_carrot: 3.0
use_cost_regulated_linear_velocity_scaling: true
regulated_linear_scaling_min_radius: 2.0
regulated_linear_scaling_min_speed: 0.25
use_rotate_to_heading: false
allow_reversing: true
max_angular_accel: 1.4
max_robot_pose_search_dist: 3.0
use_interpolation: false

```

Figura 3.29: Archivo de configuración del algoritmo RPP. Fuente: [11]

Planner server

Para escoger el planificador global se realizó de manera idéntica al controlador, según la **figura 3.30** podemos ver que las únicas opciones que se tienen son **Smac Planner Hybrid** y **Smac Planner Lattice**, se procedió a investigar sobre estos algoritmos y la actual implementación de ellos en Nav2, en este caso se escogió el **Smac Planner Hybrid** por la sencilla razón que este algoritmo está especialmente diseñado para robots de tipo Ackermann y por tanto tiene la opción de habilitar la implementación de modelo Reeds-Sheep, este modelo a diferencia de su opuesto (Dubin) toma en consideración a la hora de generar un trayecto la posibilidad de ir marcha atrás, lo que parecía interesante, el Smac Hybrid es una versión modificada del conocido A*, un algoritmo de búsqueda recursiva [19] que a su vez está basado en el algoritmo Dijkstra junto con una matriz *heurística* para conseguir una solución con mayor rapidez y menor número de búsquedas o iteraciones

Planners			
Plugin Name	Creator	Description	Drivetrain support
NavFn Planner	Eitan Marder-Eppstein & Kurt Konolige	A navigation function using A* or Dijkstra's expansion, assumes 2D holonomic particle	Differential, Omnidirectional, Legged
SmacPlannerHybrid (formerly SmacPlanner)	Steve Macenski	A SE2 Hybrid-A* implementation using either Dubin or Reeds-Sheep motion models with smoother and multi-resolution query. Cars, car-like, and Ackermann vehicles. Kinematically feasible.	Ackermann, Differential, Omnidirectional, Legged
SmacPlanner2D	Steve Macenski	A 2D A* Implementation Using either 4 or 8 connected neighborhoods with smoother and multi-resolution query	Differential, Omnidirectional, Legged
SmacPlannerLattice	Steve Macenski	An implementation of State Lattice Planner using pre-generated memory structures for kinematically feasible planning with any type of vehicle imaginable. Includes generator script for Ackermann, diff, omni, and legged robots.	Differential, Omnidirectional, Ackermann, Legged, Arbitrary / Custom
ThetaStarPlanner	Anshuman Singh	An implementation of Theta* using either 4 or 8 connected neighborhoods, assumes the robot as a 2D holonomic particle	Differential, Omnidirectional

Figura 3.30: Tabla de planificadores disponibles para Nav2. Fuente: [11]

Este algoritmo cuenta con múltiples configuraciones como la ya comentada selección del modelo *Dubin o Reeds-Shepp*, radio mínimo de giro o la selección de las diferentes penalizaciones para los comportamientos del algoritmo como se puede observar en la figura 3.31.

```

GridBased:
  plugin: "nav2_smac_planner/SmacPlannerHybrid"
  downsample_costmap: false
  downsampling_factor: 3
  allow_unknown: true
  max_iterations: 1000000
  max_planning_time: 5.0
  motion_model_for_search: "REEDS_SHEPP"
  angle_quantization_bins: 64
  analytic_expansion_ratio: 3.5
  analytic_expansion_max_length: 3.0
  minimum_turning_radius: 1.6
  reverse_penalty: 1.0
  change_penalty: 0.0
  non_straight_penalty: 1.2
  cost_penalty: 2.0
  retrospective_penalty: 0.015
  lookup_table_size: 20_0
  cache_obstacle_heuristic: false
  smooth_path: True

smoother:
  max_iterations: 1000
  w_smooth: 0.3
  w_data: 0.2
  tolerance: 1.0e-10
  do_refinement: true

```

Figura 3.31: Archivo de configuración para el planificador Smac Hybrid. Fuente: propia

La selección de estas ”penalizaciones” ha sido escogida en base a pruebas realizadas con el robot y con la ayuda de las recomendaciones dadas por el creador del algoritmo, Steve Macenski [20] las cuales se reflejan en 3.1

Tabla 3.1: Parámetros de configuración para las penalizaciones del planificador

Penalizaciones	Rango
Penalización de coste	1.7 - 6.0
Penalización de giro	1.0 - 1.3
Penalización de cambio de trayectoria	0.0 - 0.3
Penalización de marcha atrás	1.3 - 5.0

Smoother server

Este servidor es el encargado de suavizar el camino que genera el planificador antes de mandarlo al controlador, así como su implementación es muy necesaria, su configuración es muy básica, para sus parámetros se escogió un nodo de suavizado con sus valores por defecto como se ve en la figura 3.32.

```

smoother_server:
  ros__parameters:
    smoother_plugins: ["simple_smoothen"]
    simple_smoothen:
      plugin: "nav2_smoothen::SimpleSmoothen"
      tolerance: 1.0e-10
      max_its: 1000
      do_refinement: True

```

Figura 3.32: Archivo de configuración para el suavizador de trayectorias. Fuente: propia

3.6.3. Mapas de coste

Como ya se ha comentado, otra parte importante de la configuración son los mapas de coste donde aquí reside uno de los problemas principales a la hora de que Nav2 funcione en exteriores. En una configuración normal uno de los requerimientos básicos para funcionar es proporcionar un mapa del entorno estático mapeado *a priori*, tarea imposible para realizar en exteriores. Para solucionar este problema existen múltiples formas de proceder, la que se decidió llevar a cabo fue eliminar la capa *estática* en la configuración de los mapas de coste. Estos mapas guardan la información sobre los obstáculos actuales y los obstáculos que se han visto a lo largo de la navegación para así poder evitarlos de manera local (evasión de obstáculos) y de manera global (generación de una trayectoria que tenga en cuenta esos obstáculos).

El mapa de costes local existe en el sistema de referencia de *odom* y se ejecuta a una frecuencia más alta (5 Hz en nuestro caso), además consta de 2 capas, la primera es una capa "persistente" en el tiempo, *voxel*, con la información del Lidar 3D, creando así zonas de obstáculos por las que el robot no puede navegar y otra capa no persistente, *inflation*, que infla estos obstáculos de manera exponencial creando zonas de peligro por donde el robot no debe pasar o al menos pasar a una velocidad reducida para así evitar choques laterales a causa de la forma del robot, **figura 3.33**, este mapa fue creado con un lado igual a 10 m y se añadió un parámetro con las medidas exteriores del robot, esto es un rectángulo, para que de esta manera el controlador tenga en consideración la forma del vehículo. En la **figura 3.33** se distingue en colores vivos el mapa local y por debajo casi transparente el global.

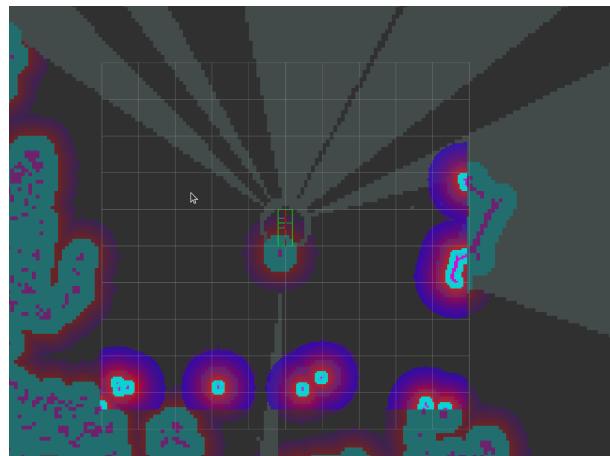


Figura 3.33: Ilustración del mapa de coste local. Fuente: propia

Para el mapa global se creó otra instancia idéntica pero con un lado igual a 50 m, el sistema de referencia de este mapa sera */map* y se cambiará la capa de *voxel* por la de *Obstacle* que será idéntica a la anterior pero esta será una capa 2D. Además, se suscribirá a el topic de *ouster/scan* ya que para el mapa global no necesitamos tanta información del Lidar así como tampoco tanta precisión para el cálculo de la trayectoria según las dimensiones del vehículo por lo que se asumirá que el robot es circular para este mapa de costes, **figura 3.34**.

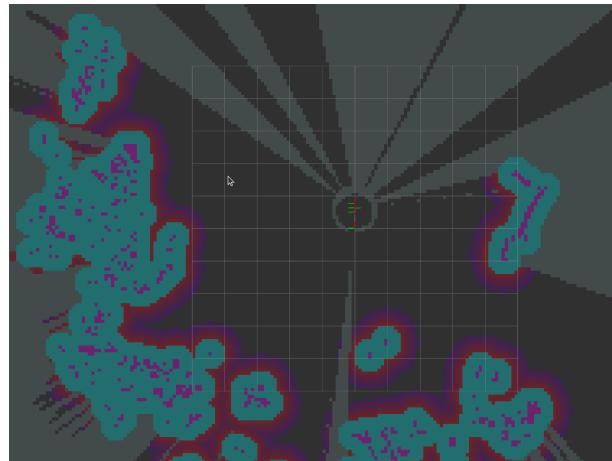


Figura 3.34: Ilustración del mapa de coste global en tonos morados transparentes. Fuente: propia

3.6.4. Plugins

Los plugins que se han usado son dos, el primero un suavizador final de velocidad donde hemos escogido controlar la velocidad en bucle cerrado y hemos seleccionado los rangos de velocidad y aceleración que queremos usar en nuestra aplicación, estos están expresados por 4 vectores como se puede apreciar en **figura 3.35** donde los valores se refieren a $[\dot{X}, \dot{Y}, \dot{\theta}_z]$ y $[\ddot{X}, \ddot{Y}, \ddot{\theta}_z]$ respectivamente.

```
velocity_smoother:
  ros_parameters:
    smoothing_frequency: 20.0
    scale_velocities: false
    feedback: "CLOSED_LOOP"
    max_velocity: [1.5, 0.0, 1.5]
    min_velocity: [-1.5, 0.0, -1.5]
    max_accel: [1.0, 0.0, 1.0]
    max_decel: [-1.0, 0.0, -1.0]
    odom_topic: "hunter/odom"
    odom_duration: 0.1
    deadband_velocity: [0.0, 0.0, 0.0]
    velocity_timeout: 1.0
```

Figura 3.35: Archivo de configuración para el plugin de suavizado de velocidades. Fuente: propia

En el caso del plugin de *waypoint follower* se tienen solo un par de parámetros, seleccionar que se quiere hacer en cada *waypoint* u objetivo (estos pueden esperar a recibir un mensaje por un topic, sacar una foto o pararse durante un tiempo), en el presente proyecto se escogió pararse y se seleccionó 0s de duración en la parada para que se cree una trayectoria continua, este plugin ofrece la posibilidad de mandar objetivos que están más lejos que el rango del mapa de costes global enviándole un array de poses que estén distanciados cada 2 objetivos una menor distancia que el rango del mapa de costes global, esto es a razón de que este plugin controla el envío del siguiente objetivo del array conforme llega al actual.

3.6.5. Nav2 commander

Como ya se ha comentado en la sección de procesamiento de mensajes, se necesita un procesamiento de las peticiones, estás se podrían clasificar en 3 situaciones, la primera donde la distancia al marcador está a menor distancia que el rango del mapa de costes global, en ese caso el procedimiento es sencillo, simplemente se publica la pose por el topic `/commander/goal` que es de tipo `/geometry_msgs/Pose`, si el objetivo esta a más distancia entonces se deberá calcular los puntos intermedios como se puede observar en la **figura 3.36**, en este otro caso publicaremos por el topic `/commander/goal_array` de tipo `/geometry_msgs/PoseArray`.



Figura 3.36: Ejemplo de cálculo de puntos intermedios. Fuente: propia

Nav2 commander es el último nodo realizado para este proyecto y es el encargado de suscribirse a los 2 topics mencionados anteriormente, además de otro topic de tipo `/std_msgs/Bool` llamado `/web/follow` que será *true* cuando el modo "follow me" este activo. En el caso de recibir un mensaje por el topic `/commander/goal_array` este se comanda al plugin de *Waypoint follower* por medio de una petición de su servidor de acción correspondiente.

En el caso de recibir un mensaje por `/commander/goal` tenemos 2 opciones o el `/web/follow` esta en *false*, donde simplemente se hará una petición al servicio propio de Nav2 para navegación punto a punto o que este a *true* donde se deberá hacer uso del mismo servidor pero indicando un árbol de comportamiento específico para esta tarea como se observa en la **figura 3.37**.

Este árbol indicará a Nav2 que debe hacer el procedimiento estándar para ir a un punto pero recortando el trayecto 1m para así quedarse siempre a una distancia de la persona, también para hacer más dinámica la búsqueda del objetivo las actualizaciones se realizarán por el topic `/goal_update` del tipo `/geometry_msgs/PoseStamped`, por tanto cuando el "follow me" esté activo, los nuevos objetivos enviados por la aplicación web se publicarán por este topic.

```
<root main_tree_to_execute="MainTree">
  <BehaviorTree Id="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <RateController hz="1.0">
        <Sequence>
          <GoalUpdater input_goal="{goal}" output_goal="{updated_goal}">
            <ComputePathToPose goal="{updated_goal}" path="{path}" planner_id="GridBased"/>
          </GoalUpdater>
          <TruncatePath distance="1.0" input_path="{path}" output_path="{truncated_path}" />
        </Sequence>
      </RateController>
      <KeepRunningUntilFailure>
        <FollowPath path="{truncated_path}" controller_id="FollowPath"/>
      </KeepRunningUntilFailure>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

Figura 3.37: Programación en xml del árbol de comportamiento para seguimiento dinámico de una persona ("follow me")

Hay que resaltar que aunque el algoritmo del "follow me" ha sido probado y funciona, este es altamente dependiente de la precisión del GPS empleado y ya que los dispositivos móviles tienen una precisión bastante baja, el resultado obtenido no es bueno.

CAPÍTULO 4

Pruebas y resultados

En el presente capítulo se trata de corroborar que la teoría funciona, poniendo a prueba los algoritmos desarrollados. Es decir, que todos los sistemas de abordo son efectivos en su trabajo cumpliendo con los requisitos establecidos, para así sacar conclusiones finales sobre las soluciones dadas para este proyecto.

4.1. Pruebas de precisión para el sistema sensorial

Para esta prueba se condujeron varios experimentos con la esperanza de encontrar las varianzas de las IMU y así obtener una buena localización.

El primer sensor que se puso a prueba fue la IMU Xsens MTi ya que como se puede ver en la **figura 3.10**, el fabricante proporciona unos valores de ruido que se pueden usar para inducir unos valores en la matriz de covarianza. Por lo que de manera conservadora se tomarán estos mismos valores en las diagonales de las matrices.

■ Orientación	■ Velocidad Angular	■ Aceleración Lineal
$\begin{bmatrix} 0,475 & 0 & 0 \\ 0 & 0,475 & 0 \\ 0 & 0 & 0,475 \end{bmatrix}$	$\begin{bmatrix} 0,005 & 0 & 0 \\ 0 & 0,005 & 0 \\ 0 & 0 & 0,005 \end{bmatrix}$	$\begin{bmatrix} 0,008 & 0 & 0 \\ 0 & 0,008 & 0 \\ 0 & 0 & 0,008 \end{bmatrix}$

Después de varias iteraciones se ajustaron estos valores. Para la orientación se decidió dejar el valor de 0,475, para velocidad angular se bajó a 0.004 y para la aceleración lineal se subió hasta el valor de 0.01. Como se pueden observar en la **figura 4.1** la línea azul corresponde a la salida del sensor y la roja con menor ruido a la salida del filtro. Concluyendo así, la existencia de una mejora considerable en el error al tener en cuenta estos valores de varianza.

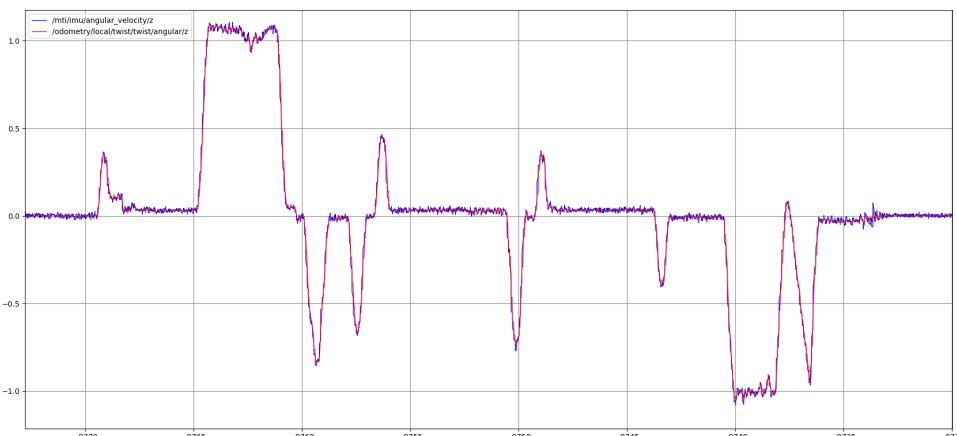


Figura 4.1: Comparación entre la salida directa del sensor y la salida después de procesarse por el filtro de Kalman con las matrices de covarianza.

Para la IMU integrada del Ouster dado los buenos resultados de los valores de la IMU de Xsens se introdujeron los mismos valores de varianza y se implemento el filtro complementario para mejorar el error, eliminando el "bias" del dispositivo y calculando la orientación en base a su giroscopio y acelerómetro. En la **figura 4.2** se puede observar la salida directa del sensor (gris) y la procesada por el filtro complementario (rojo) donde se ve claramente un suavizado en la salida junto con la eliminación de gran parte del *offset* o 'bias' que posee la señal sin procesar.

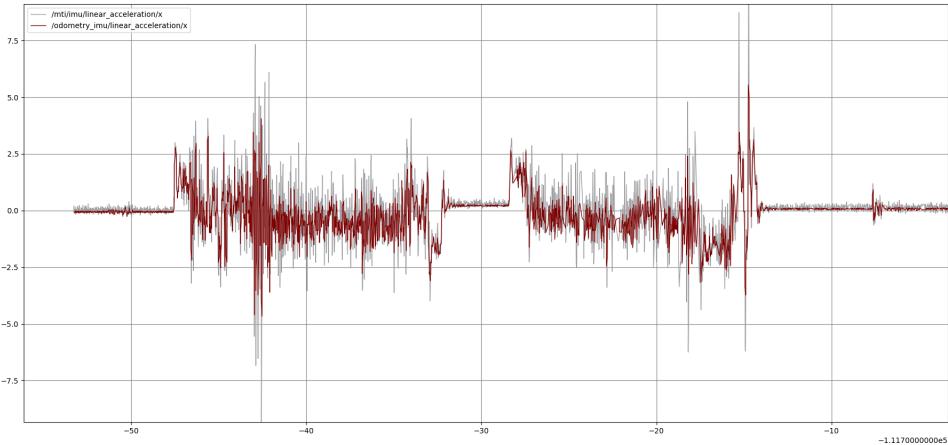


Figura 4.2: Comparación entre la salida directa del sensor y la salida del filtro complementario

Para terminar se condujo una prueba de movimiento con el GPS y el RTK conectado para comprobar tanto la configuración del nodo de transformación como para corroborar la precisión que puede dar este dispositivo. La información vendrá dada por un mensaje de tipo *NavSatFix*, en concreto el campo *Status* de tipo entero, pudiendo ser -1 (cuando no se encuentra solución), 0 (cuando la solución es de tipo "sencilla", del orden de 4-5 m de error), 1 (cuando la solución es de tipo "doble", del orden de 10-50 cm de error) o 2 (cuando la solución es de tipo "fija", con el esperado 1 cm de error), por tanto nos interesa que el GPS se mantenga con un *status* igual a 2.

En las siguientes figuras se puede observar el experimento donde tanto la salida del filtro, **figura 4.4**, como la directa del sensor, **figura 4.3** permanecen estables, comentar que el offset que existe en la **figura 4.4** entre la señal de odometría del robot (rosa) y la salida del filtro de Kalman global(verde) es a causa de el procedimiento para el cálculo del datum, donde el robot se mueve una determinada distancia (en este caso se puede apreciar aproximadamente 1m), esto no afecta al filtro ya que esta variable se introduce como relativa (se le resta el valor inicial a todos los posteriores) de manera que desde el punto de vista del filtro comienza en 0.

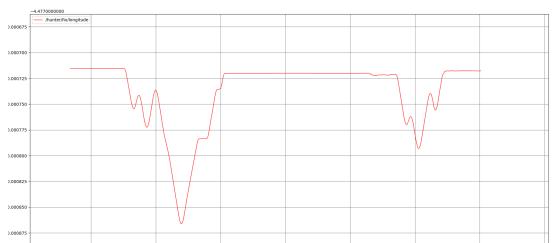


Figura 4.3: Salida del GPS con RTK.

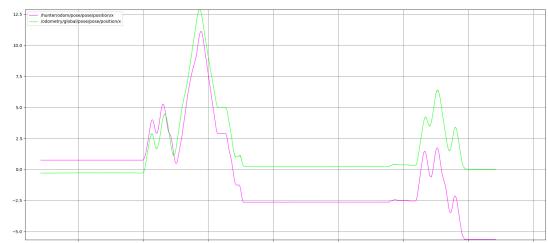


Figura 4.4: Salida del filtro.

4.2. Pruebas de funcionamiento de los algoritmos de adaptación

El primer algoritmo de adaptación es el generador de Datum o *DatumGen*, para esta prueba se quiso comprobar la repetibilidad del algoritmo y la precisión a la hora de calcular el ángulo. Para ello se condujeron 2 experimentos, el primero con el robot mirando perfectamente al Oeste donde se probó la repetibilidad del sistema y un segundo experimento donde se probó la precisión calculando el ángulo para varios valores de orientación. Para las 2 pruebas el robot se moverá a una velocidad constante de 0.8 m/s hacia delante 2 m.

En el primer experimento se realizaron 6 pruebas desde la misma posición inicial.

Tabla 4.1: Resultados de repetibilidad del algoritmo

Número de prueba	ángulo del algoritmo (rad)	ángulo de la IMU (rad)
1	2.9631345	-2.9277734
2	2.9295575	-2.9964118
3	2.9349424	-3.0181004
4	2.9248358	-3.0091396

Como se puede ver en la **tabla 4.1** se consigue una repetibilidad buena con una desviación típica de 0.01718 rad, que equivale a aproximadamente 0.98º.

Para el segundo caso se realizó el experimento para los 360º en intervalos de 90º con los mismos parámetros que en el anterior (0.8 m/s y 2 m de movimiento lineal).

Tabla 4.2: Resultados de precisión del algoritmo

Dirección	ángulo del algoritmo (rad)	ángulo de la IMU (rad)
Este	0.1175064	0.2226914
Norte	1.4017609	1.8223669
Oeste	2.9154884	3.0262419
Sur	4.541442707	4.292687807

Como también se ve en la **tabla 4.2** se consiguió una precisión de 0.17089 radianes, siendo estos resultados aceptables para el proyecto.

El segundo algoritmo a testar será el encargado de generar los arrays de poses en base a las peticiones de objetivos solicitadas por el usuario por medio de la web. Para comprobar el correcto funcionamiento sencillamente se harán 2 peticiones, la primera a un punto más alejada que el rango del mapa de coste y la segunda una ruta de una distancia considerable, el algoritmo deberá pintar en Rviz todos los objetivos generados con sus orientaciones colocadas correctamente. Como se observa en las **figuras 4.5 y 4.6** el resultado es correcto. Además, en el número 1 de la **figura 4.5** observamos el inicio del trayecto y en el 2 el final donde se crea el último objetivo, entremedias se aprecia un marcador adicional corroborando el buen funcionamiento del algoritmo.

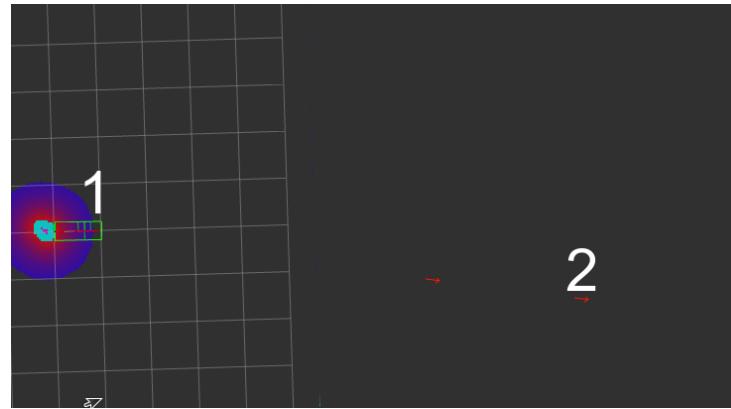


Figura 4.5: Array de poses objetivos por el algoritmo para una petición de objetivo a más distancia del rango del mapa de coste.



Figura 4.6: Array de poses objetivos generadas por el algoritmo para una petición de camino.

Como comparación en las siguientes figuras, **figuras 4.7 y 4.8**, se observa la petición enviada por la web donde podemos ver el objetivo único a una distancia considerable y el camino creado con la herramienta de linea de la API de google maps ya comentada anteriormente.

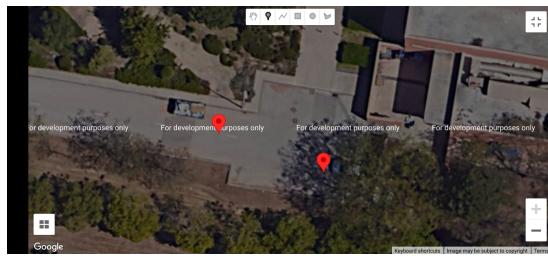


Figura 4.7: Petición de navegación a un objetivo más alejado que el rango del mapa de coste.



Figura 4.8: Petición de navegación para un trayecto establecido.

4.3. Pruebas de precisión en la localización

Lo que en este apartado se quiere demostrar es la capacidad que tiene el sistema de localización para mantener una precisión aceptable a lo largo del tiempo. Para ello se condujeron 2 experimentos donde para el primero se implementó una localización con 1 sola IMU y se trazó un recorrido para comprobar la precisión de estimación, posteriormente se realizó otro experimento donde se introdujeron las 2 IMU en los filtros de Kalman, para así comprobar si existe mejora en la orientación.

Para llegar a una conclusión se utilizó la herramienta de ROS2 *Rviz*, que ayuda a ilustrar todo tipo de mensajes de ROS, para esta prueba es necesario centrar la atención en 3 objetos. Como se ve en las **figuras 4.9 y 4.10** existen 2 flechas, la verde indica la odometría local y la roja la odometría global, estás deben estar siempre alineadas, una tendencia a separarse indicaría una diferencia entre el GPS y la odometría, por otro lado también se han ilustrado los sistemas de referencia de *odom*, *map* y *base_link*, una buena localización implicará que al final del trayecto la distancia entre el sistema de referencia fijo *map* y el sistema *odom* sea pequeña. Por último, se observará la varianza de los filtro de Kalman representada por un "óvalo" morado alrededor del camino, si crece en tamaño significará que el sistema esta perdiendo noción de donde se encuentra. Por lo que el comportamiento ideal que se busca es que el "óvalo" permanezca constante en tamaño o al menos que crezca de manera paulatina.

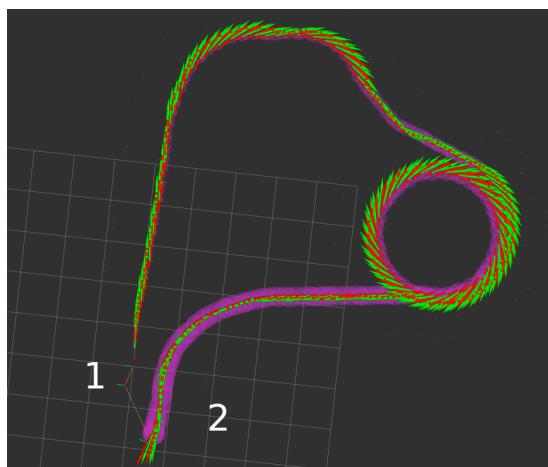


Figura 4.9: Localización mediante el uso de 1 IMU.



Figura 4.10: Localización mediante el uso de 2 IMU.

Como se puede apreciar en las **figuras 4.9 y 4.10** tanto el error de estimación como la diferencia entre las 2 odometrías es prácticamente igual para los dos casos. Sin embargo, se puede apreciar una menor deriva en los sistemas de referencia al final del trayecto realizado con 2 IMU, esto se observa en el 1 de ambas figuras.

4.4. Pruebas de navegación autónoma

Para comprobar finalmente la totalidad del trabajo, en concreto, una navegación cumpliendo con los objetivos propuestos, se desarrollaron 4 escenarios en los que se cubren todas las posibilidades de navegación.

- Navegación punto a punto.

A continuación en las **figuras 4.13 y 4.14** se pueden apreciar varios frames del movimiento real del robot para una navegación punto a punto, como referencia este se movió alrededor de 10 metros. En las **figuras 4.11 y 4.12** se comprueba el mismo movimiento ilustrado en Rviz al lado del comando ofrecido por medio de la página web.



Figura 4.11: Comando de navegación para un objetivo.

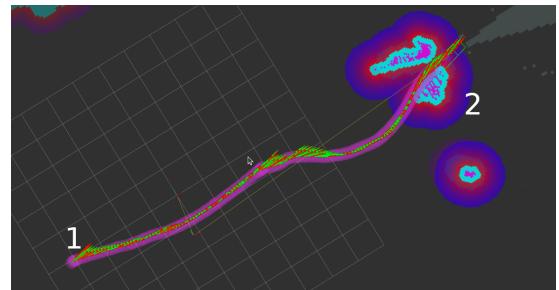


Figura 4.12: Representación de navegación punto a punto en Rviz.

Por tanto se puede comprobar que es capaz de realizar una navegación punto a punto de manera efectiva (el 1 de la **figura 4.12** es el inicio y el 2 es el final).



Figura 4.13: frame de video con el robot en la posición de inicio.



Figura 4.14: frame de video con el robot en la posición final.

- Navegación de trayecto complejo.

Para probar una navegación más compleja se comanda un trayecto con múltiples objetivos consecutivos, igualmente en las **figuras 4.17 y 4.18** se observan distintos frames del video grabado durante las pruebas realizando este trayecto y en las **figuras 4.15 y 4.16** se observa de nuevo la odometría en Rviz junto con el comando de navegación enviado por la aplicación web.



Figura 4.15: Comando de navegación para un triángulo.

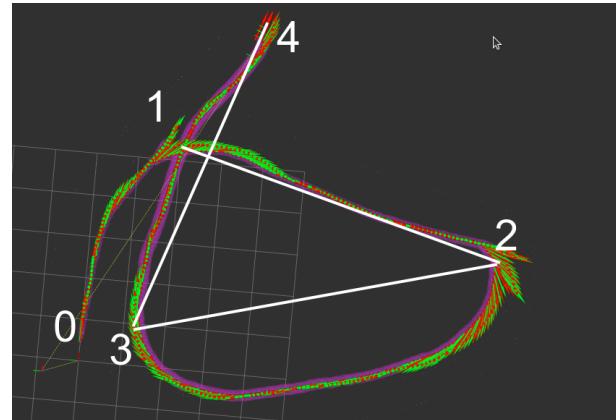


Figura 4.16: Representación de navegación compleja en Rviz.

Como se puede observar en la **figura 4.16** el 0 corresponde al inicio de movimiento desplazándose hacia cada uno de los objetivos consecutivos. Además, se aprecia un desvió entre 2 y 3 de la misma figura, esto es a causa de que el algoritmo de planificación encontró un camino más óptimo a la hora de acabar con la orientación deseada en el objetivo, (es decir sin tener que maniobrar como se puede apreciar en 1 o en 2 de la misma figura).



Figura 4.17: frame de video con el robot en la posición 2



Figura 4.18: frame de video con el robot moviéndose de la posición 2 a la 3.

- Navegación de largo trayecto.

En esta prueba el concepto que se quiere demostrar es la capacidad que tiene el sistema de localización en conjunto con el de navegación para mantener un buen conocimiento de la posición y orientación del robot. Es decir, que al cabo de un trayecto largo con múltiples objetivos consecutivos la odometría no haya "derivado" en exceso consiguiendo llegar al objetivo con precisión. Igualmente que para las otras pruebas observamos en las **figuras 4.21 y 4.22** partes del video y en las **figuras 4.19 y 4.20** se observa la representación de la odometría en Rviz junto con el camino propuesto por el usuario mediante la página web.



Figura 4.19: Comando de navegación para un largo trayecto.

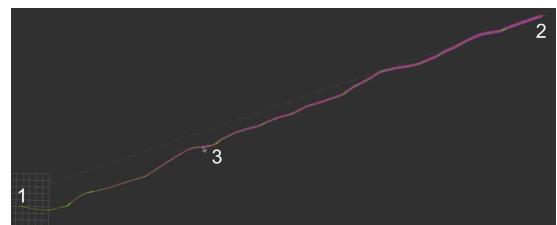


Figura 4.20: Representación de navegación de largo trayecto en Rviz.

Al igual que el resto de pruebas, esta también demostró unos buenos resultados a la hora de maniobrar largas distancias. En 3 de la **figura 4.20** se aprecia un pequeño desvío del trayecto al igual que en otras partes del recorrido, esto es debido a que el robot en estos casos debe maniobrar para llegar a los objetivos intermedios con la orientación correcta.



Figura 4.21: frame de video con el robot en la posición de inicio.



Figura 4.22: frame de video con el robot en la posición final.

- Navegación con evitación dinámica de obstáculos.

Por último se quiere demostrar que el robot es capaz de navegar de manera autónoma evitando obstáculos dinámicos en su camino. Para ello se procedió a comandar al robot moverse a un objetivo arbitrario y para simular un obstáculo se paso por delante del robot primero andando sin detenerse muy lentamente y después parándose en su camino. Este comportamiento se puede observar en los frames del video (**figuras 4.24 y 4.25**) y también en la representación de Rviz en la **figura 4.23**. Aclarar que los puntos rojos en esta misma figura muestran donde se paso por delante del robot, siendo en 1 donde se interrumpio su camino de manera rápida (andando sin detenerse), esperando ver como el robot simplemente espera 5 segundos para después seguir su camino y en 2 y 3 donde se pasó delante del robot simulando un obstáculo fijo para así comprobar como este después de esperar y dar marcha atrás, recalcula su trayecto esquivándolo de manera efectiva.

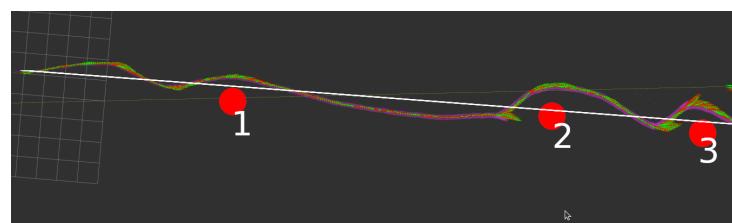


Figura 4.23: Representación de navegación con evitación dinámica de obstáculos en Rviz.

En la **figura 4.23** se puede apreciar muy claramente el funcionamiento de los comportamientos comentados en el capítulo anterior (wait, Back up) donde en 1 no se aprecia casi movimiento ante el obstáculo, simplemente se espero y desvió un poco su trayectoria. Por otro lado, el 2 y en 3 a causa de la persistencia del obstáculo decidió maniobrar hacia atrás para posteriormente evitarlo.



Figura 4.24: frame de video con el robot parado a causa de un nuevo obstáculo.



Figura 4.25: frame de video con el robot esquivando el obstáculo.

Como se ha podido comprobar el vehículo autónomo consigue llegar a todos los objetivos de manera precisa y segura completando así todas las pruebas con éxito y por tanto cumpliendo todos los objetivos propuestos en este trabajo.

CAPÍTULO 5

Conclusiones

Durante el presente proyecto se ha desarrollado una interfaz de control que cumple con los objetivos propuestos; es funcional y fácil de usar, se ha implementado el envío y procesamiento de todas las posibles funcionalidades propuestas.

Se han desarrollado algoritmos de adaptación tanto de sensores como para la mejora de la localización con buenos resultados en la fiabilidad del sistema, donde el cálculo de la orientación por medio del movimiento del robot a resultado de tener una precisión más que aceptable.

En cuanto a la localización se ha conseguido mantener una buena precisión a lo largo de un tiempo y una distancia considerable, habiéndola probado durante más de 1 hora y recorriendo algo más de 1 km continuado, habiendo comprobado también que es fiable incluso bajo pérdidas momentáneas de la señal RTK del GPS o incluso de la propia señal GPS durante varios segundos.

La navegación ha resultado también bastante fiable consiguiendo que realice cambios de dirección con la aparición de obstáculos dinámicos y sea capaz de llegar a su objetivo con un error menor a unas decenas de centímetros.

Este sistema ha sido desarrollado de manera ampliable y sobre todo para que sea utilizado en los futuros proyectos del departamento, siendo este proyecto una solución completa y robusta para ello.

CAPÍTULO 6

Futuras líneas de trabajo

Este proyecto presenta aunque aceptable para el alcance de el trabajo, una carencia en localización para ser un vehículo de exteriores, una vía de trabajo que considero mejoraría enormemente este tema sería la caracterización de los sensores IMU, consiguiendo así unos valores precisos de varianza para cada estado del sistema, dado que estas han sido impuestas de manera fija y escogidas con poco razonamiento teórico.

Otro campo que consideró mejoraría este proyecto sería el desarrollo de los comentados *behavior* o comportamientos específicos para el modelo Ackermann ya que actualmente los únicos que existen son *Wait* y *Back up* y aunque suficientes para conseguir una navegación buena, la creación de un comportamiento que retroceda de manera más "inteligente" ayudaría en gran medida a la fluidez del sistema.

Para el caso del "follow me" definitivamente se debería buscar otra manera de realizarlo sea con un gps externo o con ayuda de algún tipo de detección de personas por medio de cámaras o por medio de la reflectancia de el Lidar.

Por último, sería interesante ampliar las funcionalidades de navegación, como podría ser un barrido de areas, para ello sería necesario la creación de algoritmos para la generación de trayectorias dadas las areas.

Bibliografía

- [1] IFR. «World Robotics 2023 Report». En: *IFR Press Room 2.2* (2023).
- [2] et al. Rolan Bacilio Anota. «Localización de un robot móvil mediante el filtro de Kalman extendido y el simulador Gazebo». En: 1.1 (2020).
- [3] NASA. *Mars 2020: Perseverance Rover*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 2020. URL: <https://science.nasa.gov/mission/mars-2020-perseverance/>.
- [4] Robotnik S.L. *Manipuladores Móviles*. Página web. Fecha de Acceso: 12 de marzo de 2024. 2024. URL: <https://robotnik.eu/es/productos/manipuladores-moviles/>.
- [5] Owen Holland. «The first biologically inspired robots». En: *Robotica* 21.4 (2003), págs. 351-363.
- [6] Cristina Sánchez. *Ernst Dickmanns, el desconocido padre alemán de los coches inteligentes*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 1980. URL: https://www.eldiario.es/hojaderouter/tecnologia/ernst-dickmanns-vehiculo-autonomo-inteligente_1_5858992.html.
- [7] unknown. *A Brief History of Automated Driving — Part One: The Driverless Car Era Began 100 Years Ago*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 2020. URL: <https://www.apex.ai/post/a-brief-history-of-automated-driving-part-two-research-and-development>.
- [8] Quadis. *El Audi RS7 autónomo*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 2015. URL: <https://www.quadis.es/articulos/el-audi-rs7-autonomo/131746>.
- [9] Waymo LLC. *Taxis autónomos*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 2019. URL: <https://waymo.com/intl/es/>.
- [10] ROS2:HUMBLE. *ROS2: Documentation*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 2007. URL: <https://docs.ros.org/en/humble/index.html>.
- [11] Nav2. *Nav2 Stack documentation*. Página Web. Fecha de Acceso: 12 de marzo de 2024. 2013. URL: https://docs.nav2.org/getting_started/index.html.
- [12] Junkai Sun y col. «Path Planning Algorithm for a Wheel-Legged Robot Based on the Theta* and Timed Elastic Band Algorithms». En: *Symmetry* 15.5 (2023), pág. 1091.
- [13] Michele Colledanchise y Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.
- [14] Wikipedia. *Observador de Luenberger*. Pagina Web. Fecha de Acceso: 12 de marzo de 2024. 2019. URL: https://es.wikipedia.org/wiki/Observador_de_Luenberger.
- [15] Gerasimos Rigatos y Spyros Tzafestas. «Extended Kalman filtering for fuzzy modelling and multi-sensor fusion». En: *Mathematical and computer modelling of dynamical systems* 13.3 (2007), págs. 251-266.
- [16] Dr. Emilio Prieto. ... es el error de Abbe? Página Web. Fecha de Acceso: 12 de marzo de 2024. 2024. URL: <https://www.e-medida.es/numero-8/es-el-error-de-abbe/>.

BIBLIOGRAFÍA

- [17] Wim Meeussen. *Coordinate Frames for Mobile Platforms*. Página Web. 2010. URL: <https://www.ros.org/reps/rep-0105.html>.
- [18] Ignacio JR Sánchez y Alejandro C Limache. «Un Algoritmo de Estimación de Orientación Espacial y su Implementación Embebida Utilizando Sensores Microelectromecánicos». En: *Mecánica Computacional* 36.46 (2018), págs. 2121-2134.
- [19] Dmitri Dolgov y col. «Practical search techniques in path planning for autonomous driving». En: *Ann Arbor* 1001.48105 (2008), págs. 18-80.
- [20] Steve Macenski. *Smac Planner*. Git hub. 2020. URL: https://github.com/ros-navigation/navigation2/tree/main/nav2_smac_planner.