

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, está comenzado 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 de uso investigacional 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 precisa 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 interconectados a ROS2 mediante *drivers*, para el control adaptativo y robusto del trayecto.

El sistema aunque 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 dejado ver una buena implementación de las herramientas a disposición 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. The mastery of autonomous navigation has become a fundamental aspect in ensuring the success and efficiency of these technologies.

This paper addresses the challenge of enabling an investigational mobile robot equipped with a sensory system to navigate autonomously outdoors. Software components have been integrated and developed to achieve precise localization and navigation, along with a control and monitoring system. Various techniques for localization and navigation have been explored, as well as in-depth research on sensor hardware and their interconnection to ROS2 via drivers, for adaptive and robust path control.

Although the system has also been tested in simulation, it has primarily been tested in the environments of the Telecommunications Faculty of the University of Malaga.

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

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	3
1.3. Estructura del trabajo	4
2. Estado del arte	5
2.1. Navegación Autónoma de vehículos Ackermann	5
2.1.1. Antecedentes	5
2.1.2. Actualidad	6
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	12
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	17
3.1.4. Implementación del sistema sensorial en ROS2	19
3.2. Interfaz de usuario	21
3.3. Procesamiento de los mensajes MQTT a ROS2	23
3.4. Localización y odometría	24
3.5. Navegación Autónoma haciendo uso de Nav2	29
3.5.1. Conceptos básicos de Nav2	30
Árboles de comportamiento	30
Servidores de acción	31
Estructura para el control de navegación	31
3.5.2. Servidores	32
Navigator server	32
Behavior server	33
Controller server	33

ÍNDICE DE CONTENIDOS

Planner server	34
Smoother server	36
3.5.3. Mapas de coste	36
3.5.4. Plugins	38
3.5.5. Nav2 commander	39
4. Pruebas y resultados	41
5. Conclusiones	43
6. Futuras líneas de trabajo	45
Bibliografía	47
Apéndice A. Hojas de datos	51

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 la [1] existe un 5 % de crecimiento en el volumen anual de ventas de robots industriales y así como hace unos años este crecimiento sería principalmente de manipuladores cada vez más la robótica móvil busca su camino en la industria dando soluciones super versátiles y de muy fácil implementación para tareas que antes requerían espacios enormes, donde ahora un robot móvil de bajas dimensiones puede soportar cargas muy grandes y ser implementado en la producción sin necesidad de modificar el entorno lo más mínimo.

Otro punto importante es la problemática en la navegación que incluye conseguir un método fiable de localización donde la solución consiste en estimar la posición final del robot compensando los errores incrementales de odometría acumulados, minimizar los errores producidos por el sistema sensorial y la detección de 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 de estandarizar las soluciones que existen, desde robots de rescate, militares o vehículos para uso personal, y aunque existen multitud de formas de afrontar este problema, existe un amplio margen de mejora. Por nuestra parte, en el presente trabajo se ha tomado especial atención a la parte de localización, adaptando metodologías de estimación como el filtro extendido de Kalman,[2], dado que este presenta múltiples

CAPÍTULO 1. INTRODUCCIÓN

complicaciones a la hora de realizarlo en exteriores como ya se ha comentado, también se han implementado algoritmos que intentan mejorar los errores producidos por los sensores y finalmente algoritmos de planificación de trayectorias que se aadecuen 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 desarrollara 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 del trabajo.

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 Tercer, el desarrollo en si del trabajo donde se explican en detalle el hardware utilizado, los métodos y herramientas y la manera en la que se ha trabajado 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

2.1. Navegación Autónoma de vehículos Ackermann

2.1.1. Antecedentes

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”, pequeños dispositivos autónomos que demostraron comportamientos primitivos de evasión de obstáculos y búsqueda de energía, destacando la capacidad de los robots para adaptarse y responder al entorno sin control humano directo.

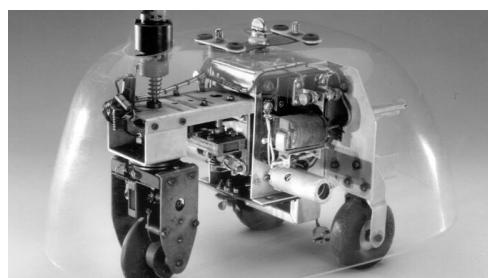


Figura 2.1: Interior de los robots ”tortuga”.

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, 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 [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.

2.1.2. Actualidad

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 ellos solos a qué hacer en cada situación en base a un entrenamiento previo, esto dio lugar a una gran ventaja ya que en este campo la programación clásica era inviable por la gran cantidad de casos 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. A este 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 un taxi autónomo [], 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 para robots, este 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 su segunda versión es la opción más utilizada para la creación de algoritmos de navegación autónoma. Este stack comprende todo lo necesario para crear un sistema robusto de navegación autónoma principalmente en interiores pero también para exteriores, comprende 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 que pueden girar *in situ*, el TEB (Time Elastic Band) o el famoso RPP (Regulated Pure Pursuit) que aunque sencillos ya nos permiten una configuración más avanzada y necesaria para nuestro caso como es el ángulo de giro mínimo, hasta otros mucho más complejos como el MPPI (Model Predictive Path Integral), un algoritmo muy complejo que usa un método de predicción en el tiempo para navegación de vehículos a grandes velocidades. Por otro lado tenemos los planificadores donde tenemos varias versiones de 2 algoritmos también muy conocidos, el A* y el Theta* [12].

En su primera versión este stack 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 de lógica muy versátiles y sobre todo extremadamente customizables, gracias a ello el stack de navegación permite una configuración absoluta de todos sus componentes, desde la creación de planificadores locales, globales, plugins para añadir comportamientos, capas para los mapas de coste, suavizadores de trayectoria o incluso la creación o modificación de los propios arboles de comportamiento.

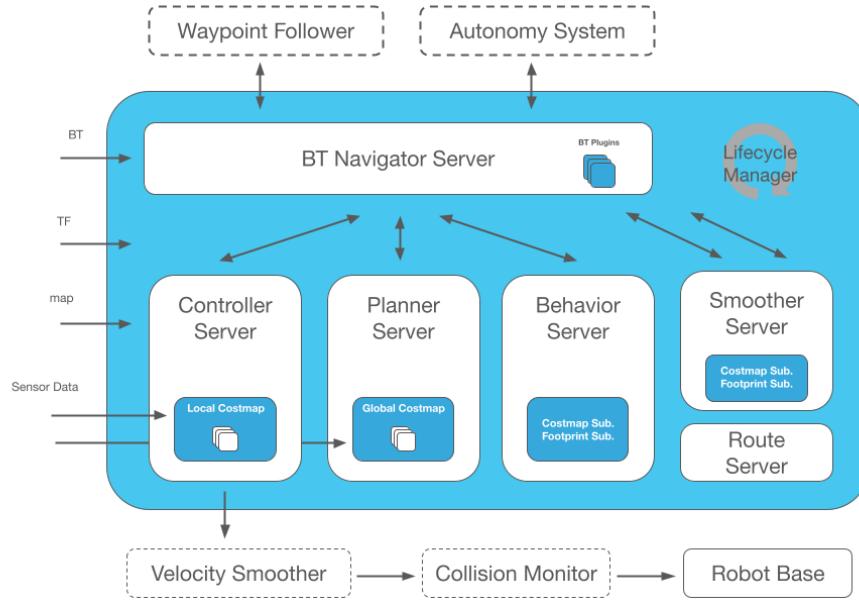


Figura 2.3: Arquitectura interna de Nav2.

2.3. Localización

La localización se conoce como uno de los principales y más complejos problemas de la robótica móvil, este 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 puede suponer un gran problema, por ello durante los años se han creado 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 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, luego 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 su propia posición. A partir de los datos capturados se construye un modelo del entorno que puede ser 2D o 3D. 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 donde el algoritmo constantemente puede “encontrarse”, el principal problema recae cuando se intenta usar en exteriores donde suele haber pocas referencias

o están mucho más espaciadas, donde el ruido es mucho mayor y existen más fuentes y 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 hemos comentado.

2.3.3. Fusión de odometrías mediante filtros extendidos de Kalman

En la fusión de odometrías con EKF, se utiliza un modelo del sistema y mediciones provenientes de múltiples fuentes, para estimar el estado del sistema, 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. El filtro 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 de matriz de varianzas *a priori* y posteriormente una etapa de corrección mediante una medición donde en base a las estimaciones del estado y de las varianza se calcula el llamado residuo de medición y la *ganancia de Kalman*, que a diferencia de otros métodos [14], este 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" [15], donde se linealiza en torno al estado actual y donde antes teníamos una matriz que relaciona el estado anterior con el actual ahora tenemos la función f para la obtención de la predicción de estados y su *Jacobiano* (\mathbf{F}) para la predicción de varianzas (2.1 y 2.2).

Para la segunda etapa también tendremos otra función h y su *Jacobiano* (\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

2.4.1. Lidar

En los inicios de la robótica los principales sensores para la medición de distancias eran los sonares pero conforme la tecnología ha ido avanzando cada vez más se han ido reemplazando por los sensores Lidar por su precisión y utilidad en gran cantidad de situaciones, estos sensores son capaces de hacer un barrido horizontal de hasta 360º y algunos un barrido vertical, como el utilizado en este proyecto, han sido utilizados tanto para detección de obstáculos, localización en el mundo como 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.

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 laser emite un rayo de luz

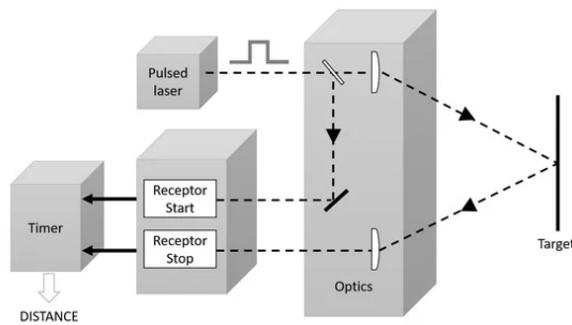


Figura 2.4: Esquema teórico de funcionamiento de un Lidar 1D

concentrada que viaja hasta el objeto a detectar, rebota en él y vuelve hasta ser detectado por el receptor del dispositivo, de esta manera sabiendo 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 al rededor de un eje obtenemos un array de valores con las distancias, es decir conseguimos un barrido del entorno 2.5.



Figura 2.5: Nube de puntos de un Lidar 3D

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 problemas, necesita una vista clara del cielo, lo cual no siempre es posible, las frecuencias a las que trabajan estos sensores suelen ser muy bajas, del orden de unos pocos hercios, lo cual dificulta mucho conseguir una odometría continua que es necesaria en la mayoría de casos para una correcta navegación, la precisión tampoco suele ser muy buena y aunque hay tecnologías más avanzadas como los GPS RTK (Real Time Kinematic), 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 cual se conoce con gran exactitud sus coordenadas GPS, dándonos hasta una precisión de unos pocos centímetros, 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) o AMCL (Adaptative Montecarlo localization), la que se usa en del presente proyecto es la fusión de fuentes de odometría basada en 2 filtros extendidos de Kalman (EKF), su implementación y características se explicará más adelante.

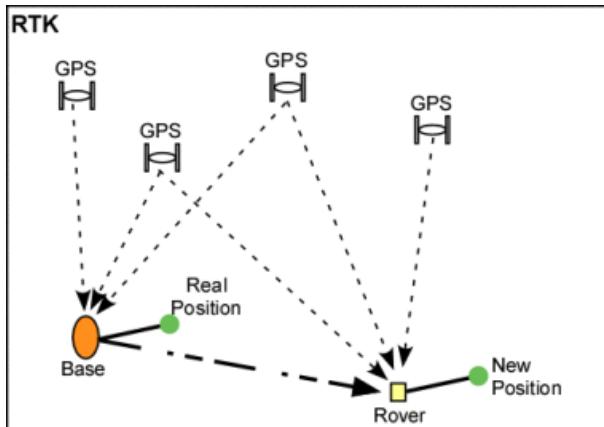


Figura 2.6: Tecnología RTK

2.4.3. IMU

Las *Inertial Motion Units* son un conjunto de sensores que normalmente puede ser de 2 tipos, de 6 ejes de libertad, donde encontramos un giroscopio y un acelerómetro o de 9 ejes de libertad donde además se encuentra un magnetómetro, en base a estos 2 o 3 sensores se puede calcular una orientación ya sea relativa al 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 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** ultra 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 con 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* y se comunica al ordenador de abordo mediante **Bus CAN**. En la **figura 3.1** se puede observar las especificaciones del vehículo completas.

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

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



Figura 3.2: Chasis del vehículo autónomo

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 0.8 y los 120 metros, esta preparado para exteriores con una visera protectora y es capaz de barrer áreas en 360º horizontalmente y 45º verticalmente, es un modelo de altas prestaciones, funciona a 100Hz y tiene una resolución de 2 cm de media ya que depende de la distancia ,todos estos datos se pueden observar también en la**figura 3.3**, para la implementación con ROS2 también se uso 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



Figura 3.4: Ouster montado en la torre

A parte del sensor láser este también tiene incorporado una IMU de 6 ejes 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 dificulto mucho la obtención de una buena localización a causa no tener 9 ejes de libertad 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** podemos 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 dos salidas del sensor.

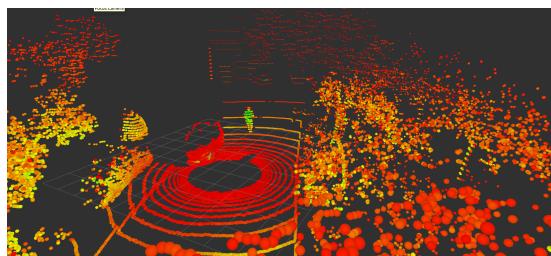


Figura 3.5: Salida de Lidar 3D en el recinto de la facultad de Telecomunicaciones

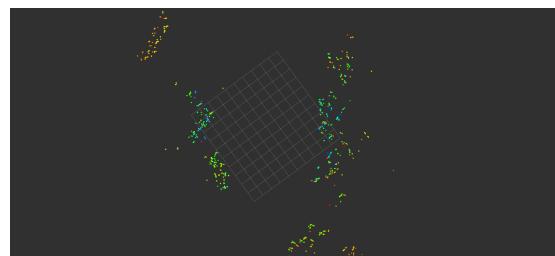


Figura 3.6: Salida de Lidar 2D en el recinto de la facultad de Telecomunicaciones

En cuanto al esquema de comunicación con el software, el driver usado publica los siguientes mensajes de datos entre varios que no interesan.

- 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+



Figura 3.7: GPS 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. Para su montaje se creo un soporte con un perfil de aluminio al cual se le atornillo en propio sensor, de manera que tenga 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 conexionado se pueden usar varios métodos, como Wifi o radio pero en este proyecto se implementó el serial por medio de USB para minimizar las interferencias y pérdidas de señal. Para la comunicación tenemos un solo mensaje de interés, este es de tipo *sensor_msgs/NavSatFix* y se publica por el topic */hunter/fix*.

Como se puede ver en la **figura 3.8** podría parecer que este dispositivo tiene una IMU pero está inaccesible para el usuario.

REACH RS2+		POSITIONING	
Technical specifications		Precision	Static: H: 4 mm+0.5 ppm; V: 8 mm+1 ppm
CONNECTIVITY		PPK	H: 5 mm+0.5 ppm; V: 10 mm+1 ppm
UHF LoRa radio	Frequency range: 868/915 MHz	RTK	H: 7 mm+1 ppm; V: 14 mm+1 ppm
Power	0.1W	Convergence time	<5s typically
Distance	Up to 8km	Signal tracked	GPS/QZSS/L1C/A, L2C, GLONASS/L1OF/L2OF, BeiDou/B1, B2I, Galileo/E1, E/C, E5b
LTE modem	Regions: Global	Number of channels	184
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/TDD: 1, 2, 3, 4, 5, 6, 8, 19 Quad-Band (850/1900, 900/1800 MHz)	Update rate	Up to 10Hz
SIM card	Nano-SIM	MECHANICAL	
Wi-Fi	802.11 b/g/n	Ingress protection	IP67 water and dust proof
Bluetooth	4.0/2.1 EDR	Dimensions	126x126x142 mm
Ports	RS-232, USB Type-C	Weight	920g
Data protocols	Corrections: NTRIP RTCM3 Position output: NMEA, LH/XYZ	Operating temperature	-20°C to +65°C
Data logging	RINEX at update rate up to 10 Hz	ELECTRICAL	
Internal storage	16 GB	Autonomy	16 hrs as LTE RTK rover
		Charging	USB-C 5V 3A
		External power input	6-40V
		Battery	LiFePO4 6400 mAh, 6.4V

Figura 3.8: Hoja de datos del GPS Reach RS2+

3.1.3. IMU Xsens MTi-3th

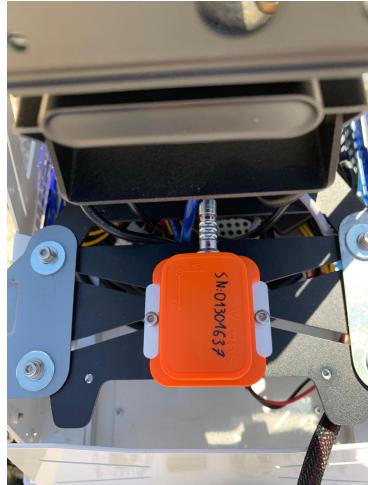


Figura 3.9: Xsens 3th gen MTI

El último sensor usado es una IMU de 9 ejes de libertad de la marca Xsens (hoy en día conocida por Movella), **figura 3.9**, este sensor no se iba a implementar ya que no se tenía conocimiento de su existencia en el departamento, es un modelo con una antigüedad de 13 años como se puede ver en la **figura 3.10**, lo que dificultó enormemente su implementación con ROS2, los drivers disponibles o no daban soporte a modelos tan antiguos o los que si lo daban estaban diseñados para la primera versión de ROS, a causa de esto el driver tuvo que ser reescrito desde cero y por tanto forma parte de este proyecto, y esta incorporado en el repositorio de GitHub ya comentado.

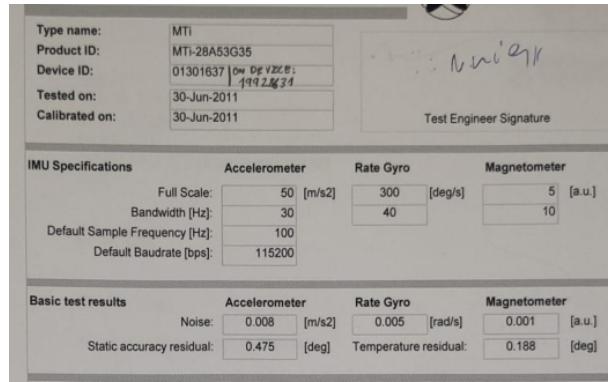


Figura 3.10: Hoja de datos de la IMU MTi-3th

En las **figuras 3.11 y 3.12** podemos observar el comportamiento de por un lado la orientación y la velocidad angular en el eje z donde podemos comprobar una salida bastante estable y por otro lado el comportamiento de las aceleraciones, en este caso extremadamente ruidoso, también aquí podemos 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 vemos. En cuanto a la señal ruidosa de aceleraciones veremos más adelante que no es un problema dado que para utilizar esta señal e localización la integraremos en el tiempo y por tanto simplemente servirá como una redundancia en la odometría de la velocidad lineal.

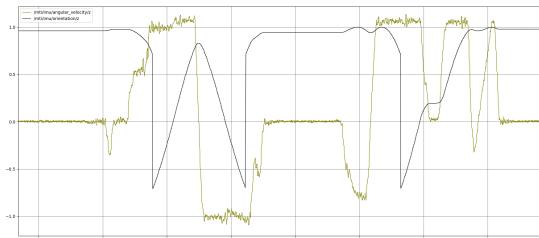


Figura 3.11: Variaciones de orientación y de velocidad angular en el eje z

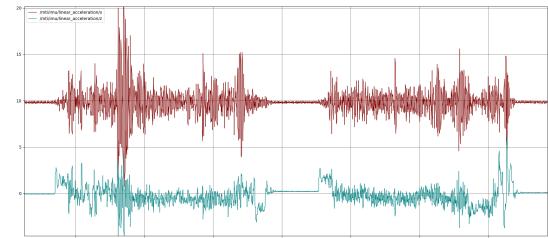


Figura 3.12: Variaciones de aceleración lineal en el eje x y el eje z

3.1.4. Implementación del sistema sensorial en ROS2

Una vez montados y 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), el nombre para este elemento debe ser "*base_link*", esta 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 compondrían 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 cual según la **REP 105**, se le llama *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 pueden ser Gazebo o Coppelia, (antiguo V-Rep), las transformaciones dinámicas las calcularemos con uno de los paquetes usados, *robot_localization* del cual hablaremos más en profundidad más adelante.

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

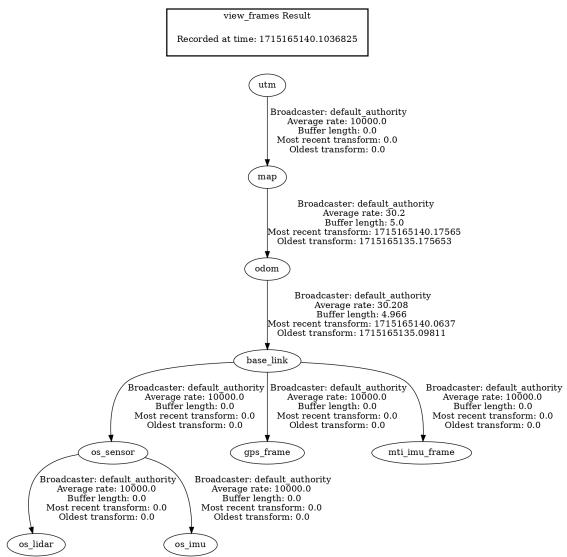


Figura 3.13: Árbol de transformaciones

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 implementados se hizo uso de una API de google para poder obtener un mapa en la web 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, como protocolo de envío nos decantamos por MQTT, por ser un protocolo rápido y sencillo de implementar, el procedimiento de uso es el que sigue, nada más conectarse al servidor la web permanece inactiva, no es posible el envío de datos hasta que el usuario ha pulsado 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, uno para publicar y otro para activar la funcionalidad del "follow_me", (número 3 y 4 respectivamente en la **figura 3.15**), 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 es correcto antes de mandarlo.



Figura 3.14: primer estado de la interfaz



Figura 3.15: segundo estado de la interfaz

La aplicación web funciona 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 HTTP/1.1 y Mosquito como broker MQTT, ambos instalados y configurados en la propia Raspberry, para la posibilidad de utilizar la pagina en cualquier sitio del mundo se ha hecho uso de 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 parte se puede abrir la aplicación aquí y controlar el vehículo de manera muy visual e intuitiva, también comentar que se ha implementado también el envío de areas tanto rectangulares como poligonales, esto ha sido ya que aunque esta funcionalidad no se ha llegado a implementar 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**, para ello hemos implementado 3 topics, el primero y más importante es /desired_pos por donde se envían las coordenadas del objetivo o objetivos es 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 estamos en modo follow_me o no, el tercero es un mensaje al que se subscribe el servidor web con la información de la posición actual del robot.

Hemos hablado de la implementación de la interfaz y de su envío mediante un broker hasta el robot pero no de como recibimos esa información y la transformamos a un mensaje útil para nosotros en nuestro entorno de desarrollo (ROS2), al enviar los mensajes formateados en JSON debemos convertirlos a un string o sino nuestro receptor no los entenderá, este es un nodo, (MQTT Bridge), que se conecta con el broker que le digamos, se subscribe a los topics que queremos y los transforma a un tipo que ROS2 entiende, este es también un string pero propio de ROS2, por lo tanto para obtener la información que necesitamos debemos parsear ese mensaje transformándolo de un string de vuelta a una tupla de coordenadas o a un string de tuplas de coordenadas.

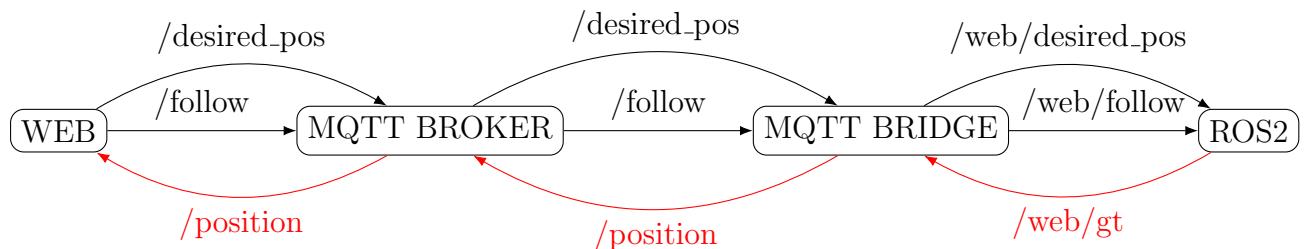


Figura 3.16: Esquema de flujo de datos desde la Web hasta ROS2

Para ello se creó un nodo que recibe la información del nodo MQTT Bridge, lo *parsea* haciendo uso de la librería *jsoncpp*, para ello primero leemos el campo que 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 las líneas 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 debemos procesar estos datos aún más, primero haciendo uso de un servicio de ROS2 implementado por el paquete de *robot_localization*, este transforma las coordenadas GPS expresadas en ángulos a un sistema cartesiano centrado en el punto de inicio y después debemos comprobar la distancia desde la posición actual del robot al punto que queremos ir, esto es debido a como funciona Nav2, si la distancia es menor a un valor dictado por los parámetros de Nav2 podemos enviarlo directamente al siguiente nodo, de no ser así generamos puntos equiespaciados desde la posición del robot hasta el objetivo y enviamos ese *array* de posiciones, también cabe destacar que debemos generar una orientación deseada, cosa que la web no nos proporciona. Para ello simplemente calculamos el ángulo que existe entre la posición del robot y la objetivo, para el caso de tener varios objetivos contiguos, es decir una ruta o camino hacemos 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)      ▷ calculo 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 ▷ hacemos lo mismo pero por cada 2 puntos del array
13:       if dist(p0, p1) > rango then
14:         array_objetivos.push.back(puntos_intermedios(p0, p1, 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 esta se publicará hacia otro nodo encargado de la lógica de control del paquete de navegación, este indicará a NAV2 donde ir y como, pero antes de ello debemos 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.

Para ello se investigaron las mejores maneras de conseguir una buena localización en exteriores, donde a diferencia de los espacios cerrados esta tarea se vuelve un poco más compleja, existen métodos como SLAM o AMCL que funcionan muy bien en interiores pero cuando estas en un ambiente tan cambiante y con tan pocas referencias a las que puedes calcular tu posición estas alternativas se vuelven totalmente inútiles, principalmente tenemos 2 fuentes de odometría (IMU, Encoders de las ruedas) y 1 fuente de localización (GPS), existen múltiples opciones para conseguir una buena localización con estos dispositivos pero la que más usan los usuarios de ROS2 son sin duda los filtros de **Kalman** para fusionar múltiples fuentes de odometría

EL **GNSS** (Global Navigation Satellite System) es una tecnología que funciona en base a hacer una triangulación con los satélites que estén a "la vista" en ese momento, normalmente esta posición es calculada por medio del estándar **WGS84**, el cual es el último estándar creado para este propósito hasta la fecha, a diferencia de mucho de sus antecesores este tiene en consideración que la tierra no es perfectamente esférica sino elíptica. En la **figura 3.17** podemos ver el modelo WGS84 y sus sistema de referencia, este estaría en el centro de la tierra con su eje **Z** apuntando al Norte y su eje **X** apuntando al primer meridiano.

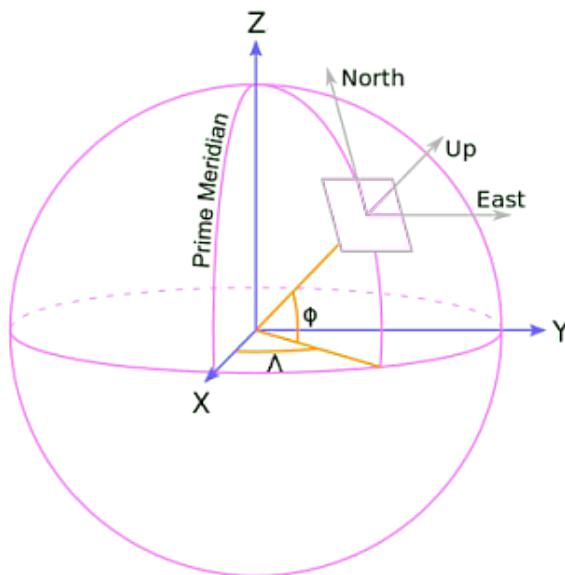


Figura 3.17: Sistema de referencia del modelo WGS84

El principal problema de este sistema de referencia es que es bastante impráctico pa-

ra la aplicación que aquí se discute, por ello necesitamos una transformación a sistema cartesiano y para ello debemos hablar también del sistema de coordenadas plano **UTM** (Universal Transverse Mercator), este es 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

A diferencia de las coordenadas geográficas este sistema se expresa en metros lo cual ya da una ventaja en cuanto a nuestra aplicación pero sigue habiendo el problema de que estas zonas son demasiado grandes por lo que también necesitamos 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 podemos observar en la **figura 3.19** a parte de conocer la posición, necesitamos también el ángulo o *bearing* y la declinación magnética, esta última es la que menos nos importa ya que es un dato que es bastante constante en el tiempo y varía por zona geográfica (varía alrededor de un 0,1 % al año), para el caso de Málaga es de 0,061 rad.

Para ello tenemos también un paquete (`robot_localization`) que nos ofrece una forma muy robusta de transformar los mensajes del GPS expresados en latitud y longitud (altitud la tomaremos igual a 0) a un sistema de referencia cartesiano, este nodo será **navsat_transform** y básicamente se subscribirá a 3 topics para calcular el *Datum* lo cual hará una sola vez y después en lo que se podría llamar el funcionamiento normal del nodo, publicará un mensaje `nav_msgs/Odometry` que será la transformación del GPS a 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 necesita para trabajar.

- un mensaje de `sensor_msgs/NavSatFix` con el mensaje proveniente del GPS, este será `/hunter/fix`
- un mensaje de tipo `nav_msgs/Odometry` con la posición actual del robot y opcional-

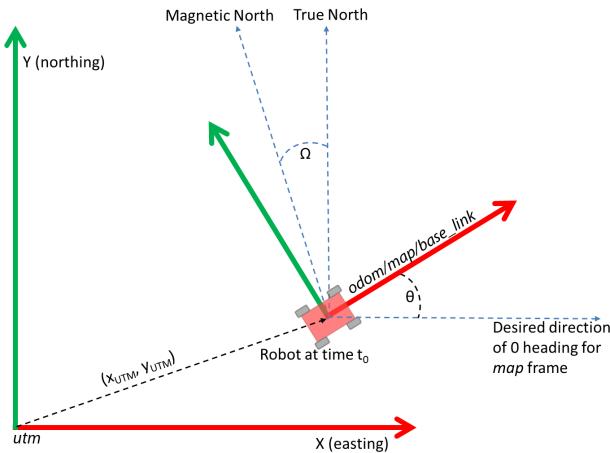


Figura 3.19: Sistema de referencia Datum

mente la orientación, está odometría será la salida de uno de los filtros de Kalman que utilizaremos para una mayor robustez en el sistema, esta información es necesaria por si el primer mensaje de GPS llega después de que el robot haya empezado a moverse, este será */odometry/global*.

- para este tercer mensaje lo que tenemos que proporcionar es 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 siendo el eje **x** y el norte el eje **y** como se puede observar en la **figura 3.17**, y para ello el nodo nos ofrece 4 maneras de hacerlo, la primera ya ha sido comentada 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 calculemos ese datum y se lo envíemos, este servicio es */datum*.
 - Indicarlo manualmente por medio de un archivo de parámetros en formato *YAML*.

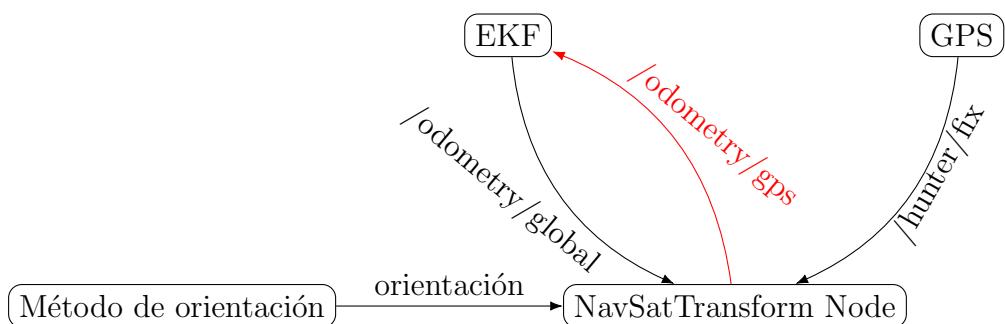


Figura 3.20: Esquema de flujo de topics en el nodo `navsat_transform`

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. Para realizar este cálculo primero el nodo se suscribe al topic `/hunter/fix` para las coordenadas GPS y al topic `/hunter/odom` de tipo `nav_msgs/Odometry` para recibir información del movimiento del robot, también publicará por `/cmd_vel` para poder hacer mover al robot. Primero permanecemos a la espera de la llegada de un mensaje de GPS y guardamos estas coordenadas, posteriormente nos movemos una determinada distancia hacia delante, es decir solo en el eje **x** que será especificada por el usuario, y finalmente volveremos a esperar a otro mensaje del GPS. Por tanto para la posición tomaremos el segundo dato de GPS y para el cálculo de la orientación global,denotada como bearing o azimut seguiremos las siguientes ecuaciones.

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

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

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

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

Las siguientes ecuaciones 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):

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.
- π es la constante pi, aproximadamente 3.14159.

Fijándonos en la **ecuación 3.4**, podemos observar que hacemos el modulo del angulo con 360° , esto es debido a que la función atan2 devuelve un valor en el rango $[\pi/2, \pi/2]$, y nosotros lo queremos en el rango $[0, 2\pi]$, también como hemos comentado el ángulo debe estar acorde al sistema ENU y por tanto le restamos $\pi/2$. Finalmente lo enviamos al servicio y el nodo a partir de aquí comienza a funcionar.

Por último falta la configuración de los EKF, para esta implementación tomaremos también el mismo paquete que no ofrece la posibilidad de configurarlos de una manera sencilla.

El principal problema que se considera aquí es la diferencia entre la localización y la localización, donde la primera es muy rápida, (más de 30 Hz) y precisa para intervalos de tiempo muy cortos, es decir *deriva* mucho con el tiempo y por otro lado la localización proporcionada por el GPS que es muy lenta, discreta, da saltos discretos en el tiempo y por tanto no vale para usar en intervalos cortos de tiempo pero es extremadamente precisa en intervalos muy largos, está es una fuente por definición no tiene *deriva* en el tiempo, por tanto necesitamos de las dos para conseguir una buena localización.

Para ello empezaremos creando una instancia de un filtro de Kalman para las fuentes continuas (Odometrías), este lo llamaremos `ekf_local` ya que lo usaremos 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 , estás son:

$[X, Y, Z, \theta_x, \theta_y, \theta_z, \dot{X}, \dot{Y}, \dot{Z}, \ddot{\theta}_x, \ddot{\theta}_y, \ddot{\theta}_z, \ddot{X}, \ddot{Y}, \ddot{Z}]$, pero ya que tomaremos que estamos trabajando en 2 dimensiones (por ser un robot terrestre y Ackermann hay mucho movimientos que podemos ignorar), de esta manera las variables que tenemos que tener en cuenta serían: $[X, Y, \theta_z, \dot{X}, \dot{Y}, \dot{\theta}_z, \ddot{X}, \ddot{Y}]$. Pero se puede ir más allá, primero con el hardware, no todos los dispositivos nos proporcionan todas estas variables y eso está bien no lo necesitamos, sino que para que el filtro funcione correctamente y no empiece a oscilar descontroladamente (comúnmente llamado "explotar"), necesitamos que entre todas las fuentes tengamos las variables necesarias, estas variarán mucho según cada situación, como 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 nos indican que variables introducir y como. Estas normas son 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 debemos priorizar a la siguiente.

1. Siempre que podamos debemos introducir variables directamente medidas, esto significa que si por ejemplo tenemos 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, mínimo necesitamos introducir una *pose* completa, que para nuestro caso de robot Ackermann sería, $[X, Y, \theta_z]$, si eso no es posible entonces debemos proporcionar sus respectivas velocidades.
3. Si tenemos 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 proporcionamos al filtro son extremadamente importantes ya que indican cuando usar una variable o cuando usar otra, nunca debemos introducir una variable donde su varianza no haya sido calculada o esta sea 0, tampoco debemos 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. Debemos analizar las restricciones de nuestro tipo de robot en particular, (Ackermann, diferencial, omnidireccional, etc...). Para nuestro caso, Ackermann, sabemos que este no puede tener un cambio instantáneo en el eje y y por tanto \dot{Y} siempre va a ser 0, podríamos pensar que no debemos introducir esta variable pero esto no es así ya que introduciendo este valor le estamos "indicando" al filtro que este no se puede mover en esa dirección, también podríamos pensar lo mismo 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 la mejor práctica en este caso es 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 una publicador de un mensaje tipo *nav_msgs/Odometry* con las entradas fusionadas también nos proporciona la transformación de */odom* a */base_link* dejándonos así un paso más cerca de conseguir todo el árbol de transformaciones como se puede visualizar en la **figura 3.13**.

Una vez comprobado en el terreno que el filtro funcionaba, se introduce una segunda instancia del filtro de Kalman, esta será nombrada como *ekf_global* y su salida solo será usada para realimentar al nodo de **NavSatTransform** como se ve en la **figura 3.21**, esta instancia tendrá las mismas entradas que el otro ekf pero con el añadido de la posición del GPS proporcionada mediante el mensaje de odometría proveniente de la transformación del nodo navsat transform y por otro lado este filtro publicará también la transformación */map* a */odom* consiguiendo así un árbol de transformaciones completo.

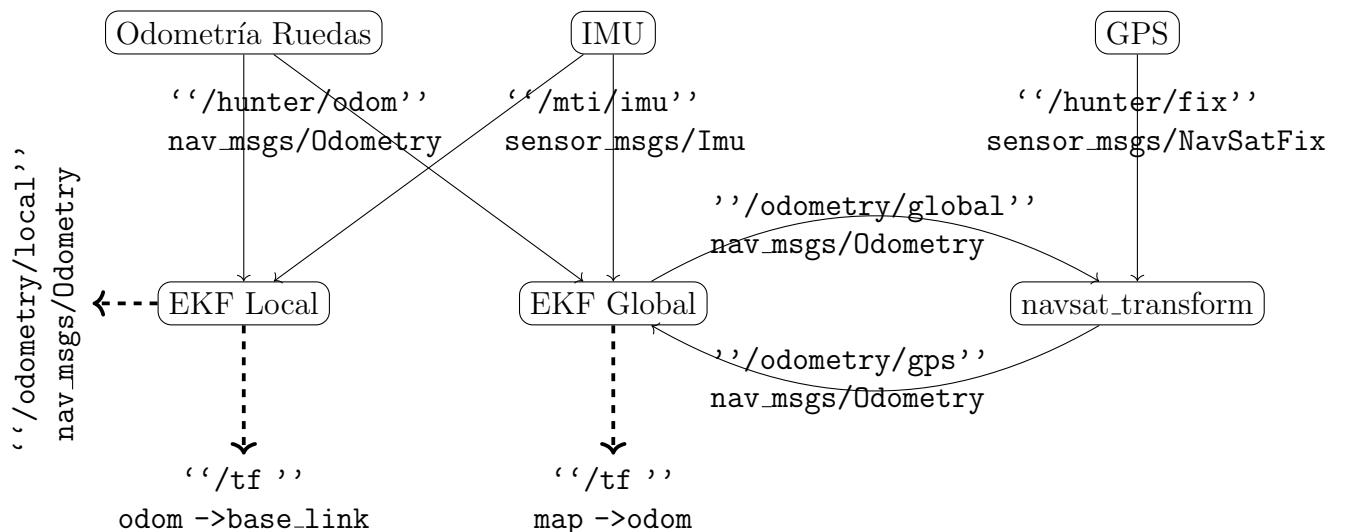


Figura 3.21: Esquema de topics completo sobre el funcionamiento de la localización

Para el caso de las entradas de fuentes de odometría de este filtro serán las mismas con la misma configuración que para el *ekf_local*, pero con la diferencia de que ahora añadiremos la salida del nodo *navsat_transform*, está es */odometry/gps* como se explica en la **figura 3.21**, para está añadiremos si o si las componentes X e Y ya que son las únicas que tiene este topic y a parte son las únicas que nos interesan, estás nos darán una localización que se mantenga en el tiempo pero con saltos discretos.

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

3.5. Navegación Autónoma haciendo uso de Nav2

El stack de navegación de ROS2 esta diseñado principalmente para interiores donde ha sido probado y testado, este funciona en base a unas estructuras lógicas llamadas *behavior trees* [13] estas estructuras son muy versátiles por su manera de funcionar 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 debemos usar

la terminología de nodo padre y nodo hijo. Los nodos de control tienen al menos un nodo "hijo" y gráficamente como se puede ver en la **figura 3.22** estos aparecen por debajo de los nodos "padre".

3.5.1. Conceptos básicos de Nav2

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* y esta 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 o otra, se podría ver como condicionales *if else* en programación, algunos ejemplos pueden ser detectar cuando se ha llegado al objetivo o si 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*, este "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*, este es 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 el primer nodo devuelva *Success* o se hallan ejecutado un nº determinado de intentos estipulados en el parámetros *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".

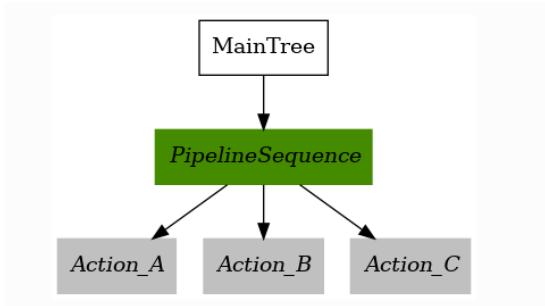


Figura 3.22: Estructura de un Árbol de comportamiento básico (Pipeline) [11]

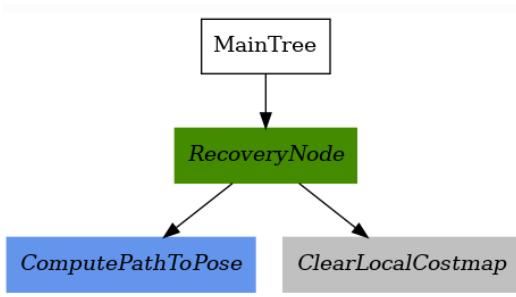


Figura 3.23: Estructura del nodo de control "Recovery"

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**, que es justo con lo que se diferencian de los servicios de ROS2, en los servicios de acción el servidor proporciona un flujo constante de mensajes sobre el *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.

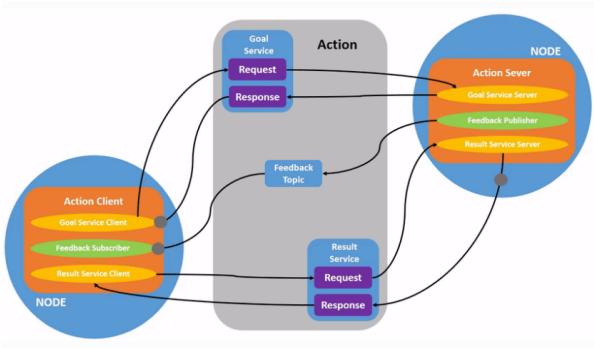


Figura 3.24: Estructura interna de un servicio de acción o *action server* [10]

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 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 *Smoother 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 estás estén a mayor distancia del alcance que tiene el mapa global, llamando al algoritmo de navegación cada vez que llega al anterior, cabe mencionar que las poses deben tener una separación máxima igual al alcance del mapa global para que así Nav2 encuentre siempre un camino.

3.5.2. Servidores

Navigator server

Para nuestro caso 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 **figura 3.26** vemos el mismo ejemplo escrito en xml para ser usado en el proyecto.

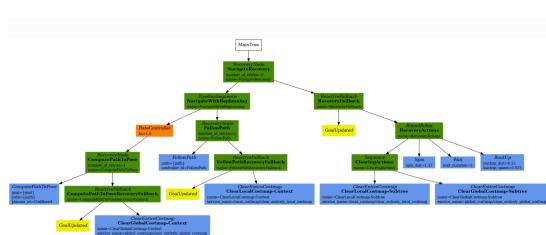


Figura 3.25: Esquema de navegación punto a punto

```
<root main_tree_to_execute="MainTree">
<!--> number of retries=6 name="NavigateRecovery">
<!--> planningSequence name="NavigateWithReplanning">
<!--> number of retries=1 name="ComputePathToPose">
<!--> ComputePathToPose goal="(goal)" path="(path)" planner_id="GridBased"/>
<!--> ClearEntireCostmap name="ClearGlobalCostmap_Context" service_name="global_costmap/clear_entirely_global_costmap"/>
<!--> RtcController>
<!--> number of retries=1 name="FollowPath">
<!--> FollowPath path="(path)" controller_id="FollowPath_h"/>
<!--> ClearEntireCostmap name="ClearLocalCostmap_Context" service_name="local_costmap/clear_entirely_local_costmap"/>
<!--> RtcListenerNode>
<!--> RtcController>
<!--> number of retries=1 name="RecoveryFallback">
<!--> RecoveryFallback name="RecoveryFallback">
<!--> Main name="RecoveryFallback">
<!--> Sequence name="ClearGlobalCostmap">
<!--> ClearGlobalCostmap name="ClearGlobalCostmap_Subtree" service_name="local_costmap/clear_entirely_local_costmap"/>
<!--> ClearEntireCostmap name="ClearGlobalCostmap_Subtree" service_name="global_costmap/clear_entirely_global_costmap"/>
<!--> Sequence name="ClearLocalCostmap">
<!--> ClearLocalCostmap name="ClearLocalCostmap_Subtree" service_name="local_costmap/clear_entirely_local_costmap"/>
<!--> ClearEntireCostmap name="ClearLocalCostmap_Subtree" service_name="global_costmap/clear_entirely_global_costmap"/>
<!--> RoundRobin>
<!--> RoundRobin duration="5" />
<!--> Backup backup_dist="0.38" backup_speed="0.05" />
<!--> RoundRobin />
<!--> RecoveryFallback>
<!--> RecoveryFallback />
<!--> BehaviorTree>
<!--> BehaviorTree />
</root>
```

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

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 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.

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 y por otro lado que sea rápido, Nav2 nos proporciona varias implementaciones para controladores como se puede ver en la **figura 3.27**. Lo

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 Budaykov	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 [11]

primero que se hizo fue seleccionar los controladores que fuesen compatibles con nuestro 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 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 lo que se descartó por dos razones, la primera fue la gran cantidad de parámetros que necesitan para su configuración dado que se observaron 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 por lo tanto, 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 robot. Por tanto se probó la segunda opción, el algoritmo **Regulated Pure**

Pursuit, una solución mucho más simple pero que daba solución al problema de manera estable, este algoritmo funciona siguiendo *objetivos* virtuales, **figura 3.28**, pertenecientes a el trayecto a seguir que se van desplazando conforme el robot se acerca, manteniendo un 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.

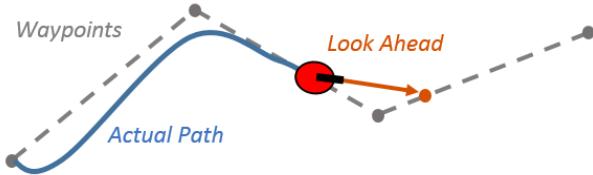


Figura 3.28: Funcionamiento del algoritmo Pure pursuit

```
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_approach_linear_velocity: 0.35
  approach_velocity_scaling_dist: 1.0
  use_collision_detection: true
  max_allowed_time_to_collision_up_to_carrot: 3.0
  use_regulated_linear_velocity_scaling: true
  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

Planner server

Para escoger el planificador global se realizó de manera idéntica al controlador, según la **figura 3.30** podemos ver las únicas opciones que tenemos 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** ya 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-Shepp, 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 cual parecía interesante, el Smac Hybrid es una versión modificada del conocido **A***, un algoritmo de búsqueda recursiva [17] que a su vez esta basado en el algoritmo Dijkstra junto con una matriz *heurística* para conseguir una solución con mayor rapidez y menor número de búsquedas o iteraciones como se puede ver en las **figuras 3.31 y 3.32**.

Planners			
Plugin Name	Creator	Description	Drivetrain support
NavFn Planner	Eitan Marder-Eppstein & Kurt Konolige	A navigation function using A* or Dijkstras expansion, assumes 2D holonomic particle.	Differential, Omnidirectional, Legged
SmacPlannerHybrid (formerly SmacPlanner)	Steve Macenski	A SE2 Dijkstra implementation using either Dubin or Reeds-Shepp 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 Dijkstra's algorithm 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 kinematic control sets 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

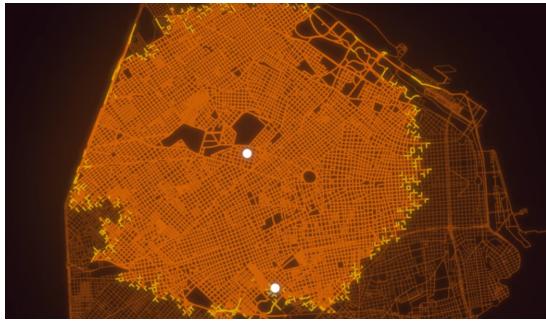


Figura 3.31: Ejemplo de búsqueda del algoritmo Dijkstra



Figura 3.32: Ejemplo de búsqueda del algoritmo A*

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 3.33.

```

planner_server:
ros_parameters:
  planner_plugins: ["GridBased"]
  expected_planner_frequency: 5.0
  use_sim_time: True
  GridBased:
    plugin: "nav2_smac_planner/$macPlannerHybrid"
    downsample_costmap: false
    downampling_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: 1.8
    reverse_penalty: 2.0
    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.e-10
      do_refinement: true
  
```

Figura 3.33: Archivo de configuración para el planificador Smac Hybrid

La selección de estos parámetros 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 [18] 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

A modo de comparación de los dos métodos, en las **figuras 3.34 y 3.35** se puede observar como generan el trayecto cada una de las configuraciones

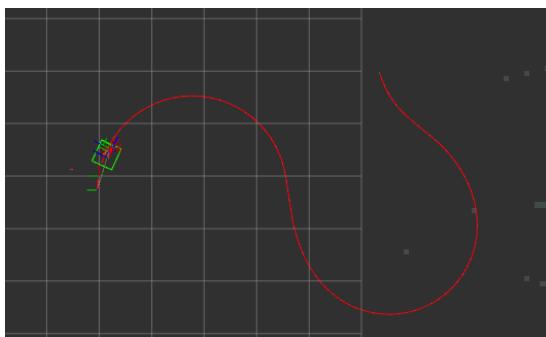


Figura 3.34: Ejemplo generación de camino para el algoritmo Smac Hybrid configurado con el modelo "Dubin" para la simulación del proyecto

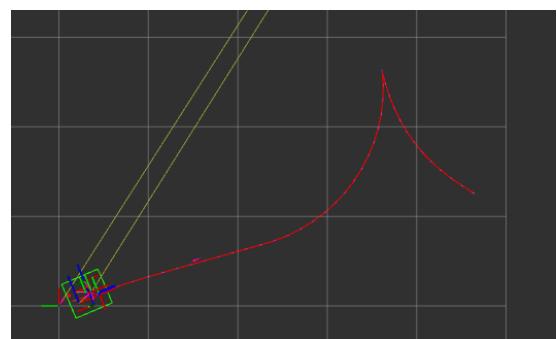


Figura 3.35: Ejemplo generación de camino para el algoritmo Smac Hybrid configurado con el modelo "Reeds-Sheep" para la simulación del proyecto

Smoother server

Este servidor esta 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 escogimos un nodo de suavizado con sus parámetros por defecto como se ve en la **figura 3.36**.

```
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 3.36: Archivo de configuración para el suavizador de trayectorias

3.5.3. 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 y es que 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 maneras de proceder, la que se decidió llevar a cabo de 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) y 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 debe pasar muy despacio para así evitar choques laterales a causa de la forma del robot, **figura 3.37**, este mapa lo crearemos con un lado igual a 10 m y añadiremos un parámetro con las medidas exteriores del robot, esto es un rectángulo para que el controlador tenga en consideración la forma del vehículo.

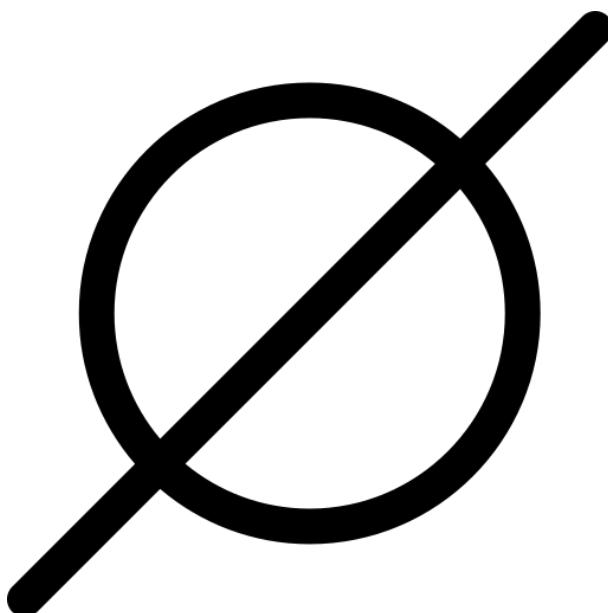


Figura 3.37: TODAVIA NO SE SI PONER EL ARCHIVO DE CONFIGURACIÓN O UNA IMAGEN DE RVIZ

Para el mapa global crearemos otra instancia idéntica pero con un lado igual a 50 m, el sistema de referencia de este mapa sera *map* y cambiaremos la capa de *voxel* por *Obstacle* que será idéntica a la anterior pero esta será una capa 2D, 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 asumiremos que el robot es circular para este mapa de costes, **figura 3.38**.

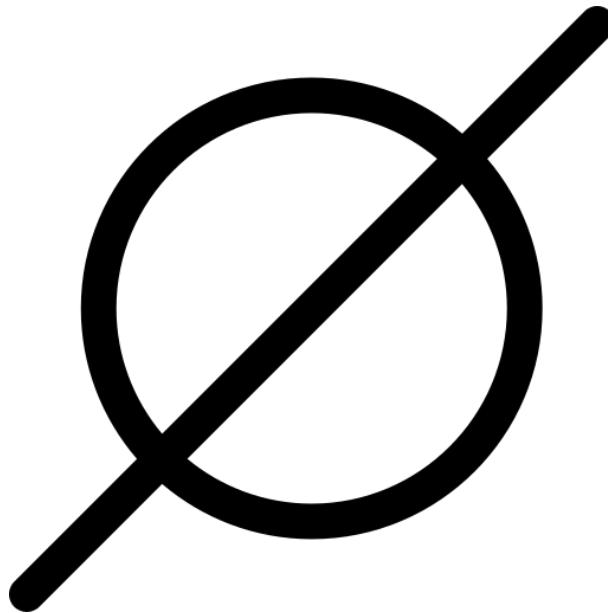


Figura 3.38: TODAVIA NO SE SI PONER EL ARCHIVO DE CONFIGURACIÓN O UNA IMAGEN DE RVIZ

3.5.4. Plugins

Los plugins que hemos usado son dos, el primero un suavizador final de velocidad donde hemos escogido controlar la velocidad en bucle cerrado de control 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 3.39** donde los valores se refieren a $[\dot{X}, \dot{Y}, \dot{\theta}_z]$ y $[\ddot{X}, \ddot{Y}, \ddot{\theta}_z]$ respectivamente.

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

Figura 3.39: Archivo de configuración para el plugin de suavizado de velocidades

En el caso de el plugin de *waypoint follower* tenemos solo un par de parámetros, seleccionar que queremos hacer en cada *waypoint* o objetivo (estos pueden esperar a recibir un mensaje por un topic, sacar una foto o pararse durante un tiempo), nosotros hemos escogido pararse y hemos seleccionado 0s de duración para que se cree una trayectoria continua, este plugin nos 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 distanciadas en el espacio menos que el rango del mapa de costes global, esto es ya que este plugin controla el envío del siguiente objetivo del array conforme llega al anterior.

3.5.5. Nav2 commander

Como ya hemos comentado en la sección de procesamiento de mensajes necesitamos un procesamiento de las peticiones, estas se podrían clasificar en 3 situaciones, la primera donde la distancia al marcador está a menor distancia de el rango de el mapa de costes global, en ese caso el procedimiento era sencillo, simplemente publicamos las pose por el topic `/commander/goal` que es de tipo `/geometry-msgs/Pose`, si el objetivo esta a más distancia entonces debemos calcular los puntos intermedios como podemos observar en la **figura 3.40**, en este otro caso publicaremos por el topic `/commander/goal_array` de tipo `/geometry-msgs/PoseArray`.

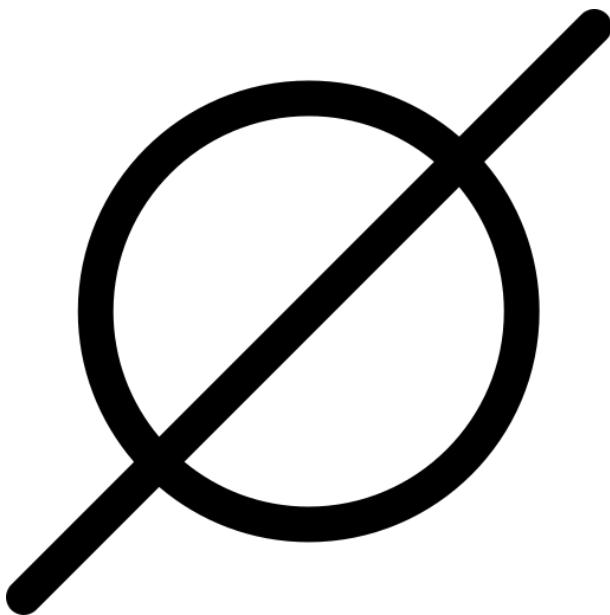


Figura 3.40: Ejemplo de cálculo de puntos intermedios

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 3.41**.

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 de el objetivo las actualizaciones de el objetivo se actualizarán por el topic `/goal_update` del tipo `/geometry_msgs/PoseStamped`, por tanto para el caso en el que esto suceda 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 3.41: 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

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, habiendo comprobado también que esta 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.

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.

BIBLIOGRAFÍA

- [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/>.
- [17] Dmitri Dolgov y col. «Practical search techniques in path planning for autonomous driving». En: *Ann Arbor* 1001.48105 (2008), págs. 18-80.
- [18] Steve Macenski. *Smac Planner*. Git hub. 2020. URL: https://github.com/ros-navigation/navigation2/tree/main/nav2_smac_planner.

Apéndices

APÉNDICE A

Hojas de datos
