Universidad de Oviedo

Memoria del Trabajo Fin de Máster realizado por

ÁLVARO FERNÁNDEZ GARCÍA

para la obtención del título de

Máster en Ingeniería de Automatización e Informática Industrial

# NAVEGACIÓN Y CONTROL DE UN ROBOT MÓVIL OMNIDIRECCIONAL EN ROS

FEBRERO 2019

# MEMORIA

# Índice de la memoria

1. Introducción ..... 9
1.1. Objetivo general ..... 9
1.2. Estado del arte ..... 9
1.3. Resumen del método ..... 20
2. Marco teórico ..... 23
2.1. *SLAM* con filtro de partículas Rao-Blackwell ..... 23
2.2. Mapeo 3D del entorno basado en *octrees* ..... 29
2.3. Navegación basada en *Time Elastic Bands* ..... 32
3. Diseño de la coordinación de comportamientos ..... 41
4. Implementación en ROS ..... 44
4.1. Simulación en Gazebo ..... 44
4.2. Visualización en Rviz ..... 45
4.3. Selección del tipo de robot ..... 45
4.4. Sistemas de coordenadas ..... 49
4.5. Control del robot ..... 51
4.6. SLAM con GMapping ..... 54
4.7. Mapeo 3D con Octomap ..... 55
4.8. Algoritmos de navegación ..... 58
4.8.1. Mapas de coste ..... 58
4.8.2. Planificador *TEB* ..... 60
4.8.3. Envío de objetivos ..... 61
4.9. Evaluación de la posición objetivo-fijada ..... 62
4.10. Inspección de la chapa ..... 62
4.11. Coordinación de comportamientos con FlexBE ..... 64

4.11.1. Uso de la herramienta FlexBE ..... 64
4.11.2. Acciones y servidores ..... 65
4.11.3. Implementación de la máquina de estados ..... 66
4.12. Estructura en ROS ..... 72
5. Experimentos y resultados ..... 73
6. Discusión ..... 79
7. Conclusiones ..... 81
8. Referencias ..... 83

# Índice de figuras

Figura 1: Representación de una rueda convencional fija. La única variable presente es la velocidad de giro de la rueda. ..... 11

Figura 2: Representación de una rueda convencional orientable centrada. En este caso, además de la velocidad de la rueda, también es variable su ángulo de orientación. ..... 11

Figura 3: Representación de una rueda convencional orientable descentrada. Las variables presentes son la velocidad de giro y la orientación del brazo que une la rueda al robot. ..... 12

Figura 4: Distintos ejemplos de ruedas omnidireccionales. Figura extraída de [4] ..... 12

Figura 5: Representación de una rueda omnidireccional del tipo *Mecanum*. Las variables de este tipo de rueda son su velocidad de giro y la dirección en la que se induce esta velocidad. ..... 13

Figura 6: Representación gráfica del problema del *SLAM* en relación con el resto de los problemas derivados de la navegación. Figura extraída de [26]. ..... 16

Figura 7: Esquema general de la estrategia general de navegación ..... 21

Figura 8: Representación de partículas de un *SLAM* y el mapa generado a partir de ellas. Figura extraída de [73] ..... 24

Figura 9: Representación de las probabilidades extraídas de la lectura de los sensores(rojo) y de las medidas odométricas (azul). En el intervalo $L^{(i)}$ se ve cómo el producto de ambas funciones está dominado por la primera. ..... 26

Figura 10: Representación de un escenario en los que los obstáculos *A y B* no serían detectados por el sensor y podrían suponer un peligro para la base móvil. ..... 29

Figura 11: Representación volumétrica y en forma de árbol de un *octree*. Figura extraída de [76]. ..... 30

Figura 12: Ejemplo en el que pueden apreciarse nodos cuya estabilidad ha propiciado que no se subdividan hasta el tamaño mínimo. El código de colores solo indica a qué altura se encuentran los puntos. ..... 32

Figura 13: Representación de las secuencias de poses (posición y orientación del robot) e incrementos temporales que construyen las *TEBs*. ..... 33

Figura 14: Representación simplificada del hipergrafo generado por el algoritmo TEB ..... 36

Figura 15: Diagrama de control de la implementación TEB ..... 37

Figura 16: a) *Path* generado inicialmente por el algoritmo original. b) *Path* suavizado mediante el uso de *splines* y fuerzas de contracción y repulsión. ..... 37

Figura 17: Ejemplo de problema derivado de optimización únicamente local de la trayectoria. ..... 39

Figura 18: Cuatro trayectorias cuyo color indica a qué grupo homotópico pertenecen. 39

Figura 19:Representación del autómata de estados a implementar ..... 41

Figura 20: Representación de las tres poses intermedias que se alcanzan para ejecutar una inspección simple de la chapa. ..... 42

Figura 21: a) Primera fase del recorrido para cubrir la fase. b) Recorrido del último carril horizontal que completa el área. Las flechas azules indican la orientación del robot. ... 43

Figura 22: Interfaz de *Gazebo* mostrando uno de los entornos creados ..... 44

Figura 23: Interfaz de *Rviz* mostrando los mensajes generados en una simulación en el mismo entorno que la Figura 22. ..... 45

Figura 24: Representación de las ruedas *Omni-Drive* extraída de sus hojas de características [91]. ..... 47

Figura 25: A la izquierda, imagen real del modelo MPO-700, extraída de [92]. A la derecha, imagen del modelo empleado en *Gazebo*. ..... 47

Figura 26: Representación de las características del láser 2D acoplado ..... 48

Figura 27: Representación de las características del láser 3D acoplado ..... 49

Figura 28: Árbol de sistemas de coordenadas empleadas en el trabajo ..... 50

Figura 29: Representación de la posición del *inspection_frame* cuando se selecciona una zona de inspección. ..... 51

Figura 30: Ejemplo de aplicación del *inspection_frame* visualizado en *Rviz* ..... 51

Figura 31: Distintos modos de funcionamiento del método *move* de la clase *RobotController* en función de la opción *ANGLE_MODE.* Las flechas azules indican la posición inicial del robot, las malvas las intermedias y las rojas las finales. ..... 53

Figura 32: Captura del programa Rviz en la que se comprueba la mejora que supone el frame del SLAM sobre el de la odometría. ..... 55

Figura 33: Los puntos 3D no son tenidos en cuenta a partir de una cota mientras que los que sí son proyectados generan un mapa 2D seguro para un robot de una altura inferior a la cota impuesta. ..... 56

Figura 34: Imagen que muestra en escala de color los puntos que detecta el sensor y en blanco los nodos que incluyen en el *octomap* una vez se ha filtrado el suelo. ..... 57

Figura 35: A la izquierda imagen de un entorno en *Gazebo*. A la derecha modelo 3D creado tras una exploración, incluyendo una cota máxima de altura. ..... 57

Figura 36: Gráfico explicativo del cálculo de los mapas de coste. Figura extraída de [110]. ..... 59

Figura 37: Representación del mapa de coste local. El mapa de coste global se extiende más y, en la imagen, puede verse con una menor opacidad. ..... 59

Figura 38: Visualización de los principales elementos de la navegación. ..... 61

Figura 39: Representación de los distintos triángulos involucrados en la evaluación de la posición fijada. ..... 62

Figura 40: Representación de las tres poses intermedias que se alcanzan para ejecutar una inspección simple de la chapa. ..... 63

Figura 41: Parámetros relativos a la configuración del robot influyentes en la generación de una trayectoria..... 63

Figura 42: Representación gráfica del ciclo de vida de un estado *FlexBE*. Figura extraída de [120]..... 65

Figura 43: Vista general de la máquina de estados. ..... 67

Figura 44: Imagen con robot en chapa dibujada..... 68

Figura 45: Interior del superestado “Recibiendo nuevo goal” ..... 69

Figura 46: Interior del superestado “Inspección simple” ..... 70

Figura 47: Flujo de datos general en el autómata ..... 71

Figura 48: Flujo de datos dentro del superestado “Recibiendo nuevo goal” ..... 71

Figura 49: Flujo de datos dentro del superestado “Inspección simple” ..... 71

Figura 50: Representación gráfica de los principales nodos, representados por elipses, y *topics* involucrados en el trabajo ..... 72

Figura 51: Arriba trayectoria real seguida por el robot en un entorno simulado de fábrica. Abajo gráfica que compara las trayectorias registradas por los distintos sistemas de coordenadas. ..... 74

Figura 52: Gráfica que representa el error absoluto en la predicción de la coordenada Y del robot en el Caso 1. ..... 76

Figura 53: Secuencia de imágenes que muestra el error en la estimación de un *voxel* y su posterior corrección. ..... 77

Figura 54: A la izquierda, imágenes de las chapas en Gazebo. A la derecha, error en la construcción del mapa 3D. ..... 77

Figura 55: Esquema general de la estrategia general de navegación ..... 81

# Índice de tablas

Tabla 1: Conjunto de acciones empleadas en el autómata ..... 66

Tabla 2: Conjunto de servicios empleadas en el autómata..... 66

Tabla 3: Constantes y variables de la máquina en FlexBE..... 67

Tabla 4: Estadísticas de la distancia entre la pose real y la estimada por el SLAM..... 75

Tabla 5: Estadísticas de la diferencia en valor absoluto entre la orientación real del robot y la estimada por el SLAM..... 75

# 1. Introducción

## 1.1. Objetivo general

El objetivo de este trabajo es el estudio y la implementación de algoritmos de navegación y mapeo en un robot omnidireccional. Este propósito se enmarca en un proyecto en el que se diseñará un robot móvil para la inspección y subsanación de defectos en chapas, funcionando autónomamente en el entorno de trabajo. La base estará provista de sensores inerciales, cámaras que contribuirán a la localización del robot mediante la identificación de patrones, un brazo para la reparación de los defectos y sensores láser para la detección de imperfecciones y el mapeo del entorno.

Se pretende poder indicar al robot distintas localizaciones y que este sea capaz de navegar de manera autónoma por el entorno e inspeccionar las chapas indicadas elaborando un mapa de los errores presentes en ellas. Una vez hecho esto, se deberá poder seleccionar algunos de los imperfectos detectados y ordenar al robot que los repare.

Este trabajo compila los primeros meses de progreso en el módulo de navegación del proyecto. En ellos se ha conseguido que el robot sea capaz, a través de sensores, de construir un mapa 3D de su entorno sobre el que poder establecer rutas a una posición dada evitando los obstáculos presentes. Además, se ha desarrollado una estructura que coordina los distintos comportamientos del robot entre los que se encuentran la, ya citada, navegación autónoma y la inspección de la totalidad del área de una chapa.

## 1.2. Estado del arte

A lo largo de la historia los avances tecnológicos han posibilitado la automatización de procesos y la simplificación de trabajos humanos. Este progreso científico-tecnológico ha sufrido un auge significativo en el último siglo de historia. Gran parte de estos avances que han revolucionado el mundo en el que vivimos han convergido en un campo tan amplio como el de la robótica.

Existe numerosa bibliografía sobre la definición de robot y las posibles clasificaciones derivadas. Como no es el objetivo de este trabajo profundizar en debates teóricos, se tomará la siguiente definición establecida en la ISO-8373[1]:

*"actuated mechanism programmable in two or more axes with a degree of autonomy, moving within its environment, to perform intended tasks"*

Otra definición que resulta interesante es la propuesta en *"Robotics, Vision and Control*" de Peter Corke [2, p. 5]:

*"a robot is a a goal oriented machine that can sense, plan and act"*

Resulta fácil suponer que la robótica, por tanto, abarca un conjunto de máquinas extremadamente diversas. De entre ellas, este trabajo girará en torno a un robot móvil omnidireccional con ruedas, es decir, un robot capaz de moverse en cualquier dirección en todo momento.

Los robots móviles son aquellos que no están anclados a su entorno y pueden desplazarse dentro de un determinado espacio. La configuración dinámica del robot determina de qué manera se mueve el robot, es decir, partiendo desde una pose inicial, qué movimientos puede realizar y cuáles no. Estas libertades y restricciones dependen del número de ruedas empleadas y de la posición y el tipo de cada una de ellas.

Siguiendo la clasificación recogida en el artículo "*Structural Properties and Classification of Kinematic and Dynamic Models of Wheeled Mobile Robots*" de Campion *et al.*[3] podemos diferenciar dos clases principales de ruedas: convencionales y onmidireccionales.

Las ruedas convencionales son las presentes en la mayoría de los mecanismos con ruedas como coches, carritos de la compra, bicicletas etc. La velocidad imprimida por estas ruedas se basa en una rotación teóricamente pura, esto es sin deslizamiento, con el suelo. A su vez, pueden dividirse en ruedas fijas, ruedas orientables y ruedas orientables descentradas.

Un ejemplo de rueda fija puede verse en la Figura 1, donde el punto $P$ representa al centro del robot y $A$ al centro de la rueda. Se puede observar que la configuración de la rueda depende de las constantes $\alpha$, $l$, $\beta$ y $r$. El ángulo variable $\varphi(t)$ representa la rotación de la rueda.

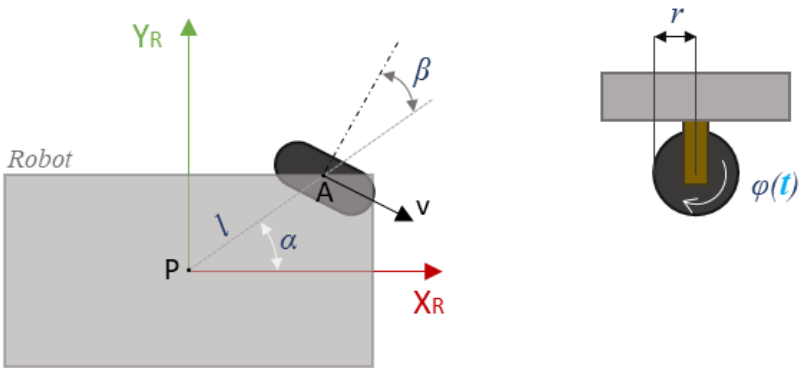


Figura 1: Representación de una rueda convencional fija. La única variable presente es la velocidad de giro de la rueda.

En la Figura 2, puede verse una representación de una rueda orientable centrada y en la Figura 3 una descentrada. Siguiendo la misma nomenclatura que en el anterior ejemplo, la configuración de las ruedas dependerá de las constantes *α, l, r* y, solo en el caso de las descentradas, *d*. Su movimiento se define por las variables *φ(t)* y *β(t)*, que hace referencia a la orientación de la misma rueda o del brazo que la conecta al robot.

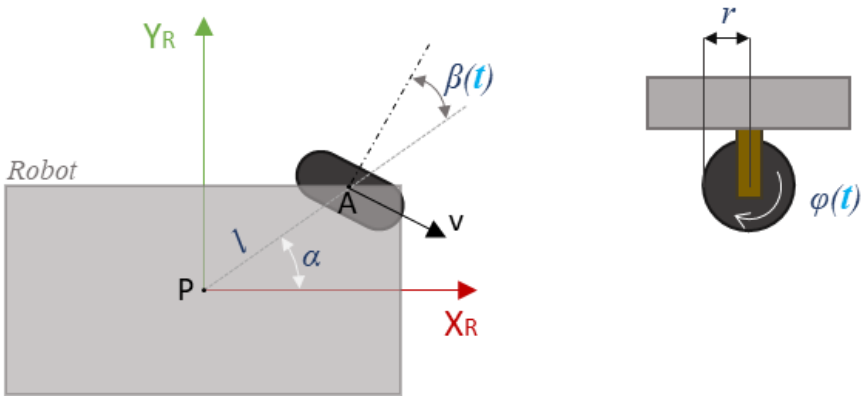


Figura 2: Representación de una rueda convencional orientable centrada. En este caso, además de la velocidad de la rueda, también es variable su ángulo de orientación.

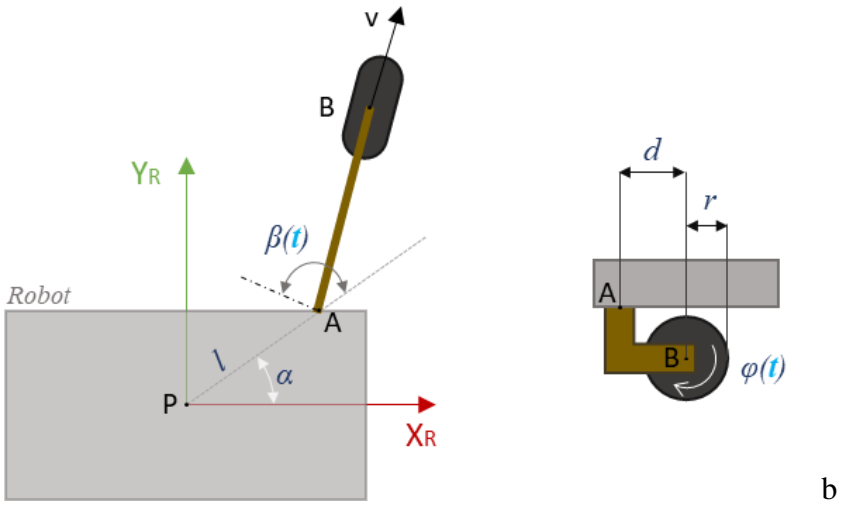


Figura 3: Representación de una rueda convencional orientable descentrada. Las variables presentes son la velocidad de giro y la orientación del brazo que une la rueda al robot.

El otro gran grupo de ruedas está compuesto por las ruedas omnidireccionales. En resumen, este tipo de ruedas, más presentes en el campo de la robótica, permiten inducir velocidades en cualquier dirección y sentido, con ingenios mecánicos similares a los que se pueden ver en la Figura 4.

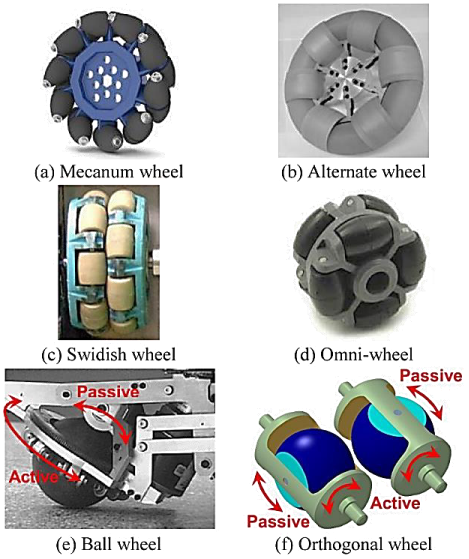


Figura 4: Distintos ejemplos de ruedas omnidireccionales. Figura extraída de [4]

Realizando una explicación más detenida del tipo *Mecanum*, uno de los más comunes, se observa que estas añaden rodillos alrededor de la circunferencia de la rueda. Los ejes de estos rodillos y el de la rueda principal difieren un ángulo γ. Como se asume que la rueda tiene un único punto de contacto con el suelo y ambos ejes pueden girar con distintas velocidades y sentidos, $\varphi(t)$ y $\varphi_{SW}(t)$, es posible generar cualquier vector de velocidad combinando los vectores de ambos ejes. En resumen, las constantes de las que depende su configuración son $\alpha$, $\beta$, $l$ y $\gamma$, Figura 5. Las variables serán las velocidades de los dos ejes de la rueda, $\varphi(t)$ y $\varphi_{SW}(t)$.

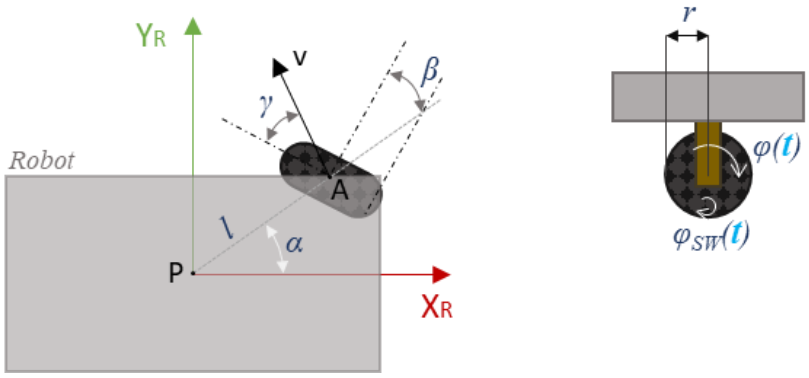


Figura 5: Representación de una rueda omnidireccional del tipo *Mecanum*. Las variables de este tipo de rueda son su velocidad de giro y la dirección en la que se induce esta velocidad.

Las variables de cada uno de los tipos de ruedas se traducen en grados de libertad de movimiento del robot. De esta manera, la combinación del número, tipo y posición de las ruedas permite otorgar distintas configuraciones cinemáticas y dinámicas al robot. Los detalles matemáticos de esta cuestión se encuentran detallados en el ya citado artículo de Campion *et al.*, [3]. En el apartado 4.3 de *Implementación en ROS* se pueden encontrar los motivos que justifican la elección de un modelo omnidireccional y algunos de los modelos comerciales que se cotejaron.

Otra bibliografía de interés sobre modelos cinemáticos de robots con ruedas puede encontrarse en el capítulo 24 del *Springer Handbook Robotics*, desarrollado por W. Chung y K. Iagnemma [5, Cap. 24], y en los primeros capítulos de *Wheeled Mobile Robotics* [6] y de *Introduction to Autonomous Mobile Robots*[7]. En ellos se pueden encontrar descripciones más detenidas de las ruedas y de los modelos matemáticos derivados.

Es común en el campo de la robótica el uso de bases móviles teleoperadas. Un repaso a diferentes estudios puede encontrarse en este artículo de R. G. Boboc *et al.*[8]. En este tipo de aplicaciones, un operador humano puede, observando el entorno, controlar el movimiento del robot para guiarlo hasta una posición final deseada. En este caso, se desarrollará un robot capaz de realizar estas tareas de manera autónoma, consistiendo la interacción humana únicamente en definir una posición objetivo. El robot deberá planificar una ruta hasta esa posición evitando los posibles obstáculos y enviar los correspondientes comandos a las ruedas para guiar a la base por el camino elegido. El conjunto de todos estos problemas complejos, intrínsecos al manejo autónomo de una base móvil, conforma el problema de la navegación.

Un gran número de tareas pueden ser desempeñadas por robots móviles que realizan una navegación básica, es decir, sin conocer el entorno que le rodea y, ni siquiera, tener una referencia de su propia localización. Este tipo de métodos de navegación se conoce como navegación reactiva, denominada así porque se basa solo en la información que toma de sus sensores.

Un buen ejemplo de estos autómatas que ejecutan una navegación simple es el vehículo de Braitenberg [2, p. 126], introducido por Valentino Braitenberg en *"Vehicles. Experiments in Synthetic Psychology"* [9]. En este concepto, la relación entre sus sensores y los motores que controlan su movimiento es casi directa. A través de dos o más sensores el robot establece, alrededor de sí mismo, un mapa de coste y el vehículo avanza hasta que alcanza un máximo. Una aplicación típica de vehículo de Braitenberg son los robots que se mueven hasta el punto de mayor iluminación, como se explora en [10].

Otro ejemplo de este tipo de robots reactivos son los conocidos como robot *bugs*, introducidos por primera vez en [11] por V. Lumelsky y A. Stepanov. Sin llegar a desarrollar una navegación compleja los robots *bug* suponen una evolución sobre los vehículos de Braitenberg, anteriormente citados. A pesar de no usar un mapa complejo, este tipo de estrategias establecen varias capas lógicas entre la lectura de los sensores y los comandos enviados a los motores. La idea principal de estas estrategias consiste en la combinación de un movimiento hacia el objetivo y la evasión de colisiones avanzando de manera paralela al borde del obstáculo. En torno a esta idea se desarrollan, entre muchos

otros, tres algoritmos básicos cuya explicación puede encontrarse en *"Wheeled Mobile Robotics"* de Klancar *et al.* [6, Cap. 4.3].

Sin embargo, las tareas más complejas requieren de un método basado en un mapa, en el que pueden definirse espacios libres y espacios ocupados u obstáculos, y en la posición del robot dentro de ese mapa.

En ocasiones no se dispondrá de uno de estos elementos y deberá obtenerse. Si no se dispone de un mapa, este deberá construirse en base a una estrategia de exploración. En ellas, el mapa se va construyendo en base a la lectura de los sensores acoplados al robot en cada pose de este. El problema continúa siendo complejo dada la ambigüedad de la información proveniente de los sensores, generalmente provocada por la naturaleza ruidosa de los mismos. En base a ello, las principales soluciones, introducidas a mediados de los ochenta por A. Elfes[12] y H. P. Moravec[13], pasan por la generación de mapas probabilísticos apoyándose en distintas versiones de filtros de Bayes [14]. Estos modelos probabilísticos, que especifican la probabilidad de que un espacio esté ocupado o no en base a la posición del robot y la lectura del sensor, dependen del tipo de sensor empleado y se denominan *modelo inverso de sensor*. Estos modelos pueden construirse de manera teórica, como en las propuestas originales, o ser inducidos directamente a partir de los datos obtenidos del propio sensor, como se propone en este artículo de S. Thrun [15].

Si se conoce el mapa del entorno pero no la posición de la base móvil, se llevarán a cabo técnicas de localización. Existe numerosa bibliografía sobre la resolución de este tipo de problema. Como ya se ha comentado, la complejidad de este ha provocado que un módulo del proyecto se dedique en gran parte a su resolución. La selección de la estrategia depende en gran medida del entorno en el que se moverá el robot y de las posibilidades de actuar sobre él. En ciertos trabajos las mejores soluciones pasan por la detección de ondas de origen conocido, como en este ejemplo de J. Biswas y M. Veloso [16] en la que se hace uso de señales *WiFi*. Una solución muy común es el uso de sensores láser de distancia cuya lectura puede asociarse con los obstáculos del mapa conocido, como por ejemplo se presenta en los artículos de Sohn *et al.* [17] y Diamantas *et al.* [18].

Otra solución muy recurrente es apoyarse en la odometría visual, es decir, el uso de la visión por computador para determinar una estimación del camino recorrido por el robot. Existen estrategias pensadas tanto para sistemas monoculares[19] como para estéreo[20].

También se puede diferenciar entre las que se aprovechan de la inclusión de elementos de referencia [21] y entre las que simplemente extraen las características del entorno sin alterarlo [22]. Finalmente, una de las soluciones más recurrentes en robótica es la fusión de sensores, generalmente mediante filtros de Kalman [23], [24], que permite combinar las estimaciones de distintos métodos, como los anteriormente mencionados, para obtener un resultado más robusto y capaz de suplir las carencias de una estrategia en solitario.

Los ejemplos incluidos en esta sección componen solo una pequeña parte de la amplia bibliografía existente sobre localización. Si se desea profundizar más en la materia pueden encontrarse capítulos dedicados enteramente al problema en textos como *Introduction to Autonomous Mobile Robots*[7, Cap. 5], *Robotics, Vision and Control* de P. Corke [2, Cap. 6], y *Probabilistic Robotics* de S. Thrun *et al.*[25].

En el caso que nos atañe se desconoce el mapa y la posición del robot en el mismo por lo que será necesario que el robot sea capaz de determinar su localización mientras construye el mapa del entorno que lo rodea. Este conjunto de estrategias se conoce como *Simultaneous Localizationd And Mapping* o, simplemente, *SLAM,* Figura 6.

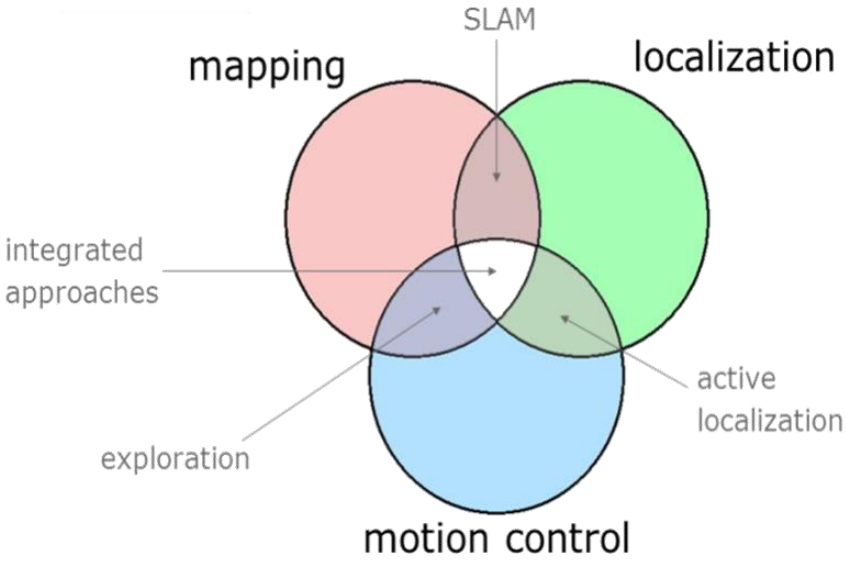


Figura 6: Representación gráfica del problema del *SLAM* en relación con el resto de los problemas derivados de la navegación. Figura extraída de [26].

Con fines explicativos, podemos dividir las estrategias de *SLAM* en base a la manera que tienen de extraer del entorno las características que permiten la ubicación y construcción del mapa.

Existen una serie de métodos que se apoyan en la visión por computador para llevar a cabo la localización y mapeo. Estas técnicas combinan la detección de puntos característicos en la imagen para el mapeo con la estimación de la trayectoria seguida. Algoritmos de este tipo son posibles incluso con el empleo de cámaras monoculares, como es el caso del algoritmo *PTAM* de Klein y Murray[27] y *ORB-SLAM* de Mur-Artal *et. al*[28].

Como resulta evidente, mejores resultados son posibles con el uso de pares de cámaras, que permiten una visión estéreo, o con sensores RGB-D, capaces de detectar la imagen con una aproximación de la profundidad de la misma. El algoritmo *ORB-SLAM*, ya mencionado, incluye en su versión mejorada[29] la capacidad de utilizar sensores de ambos tipos.

La estrategia de aprovechar la información proporcionada por sensores RGB-D también se emplea en trabajos como *RTAB-Map*[30] y *RGBDSLAM*[31]. En este caso, la naturaleza tridimensional de los puntos se utiliza para la construcción de mapas 3D.

Otros métodos prescinden del uso de imágenes y utilizan sensores de medición de distancia basados en láser o en infrarrojos. Dentro de esta familia, el *software Cartographer* [32] ofrece una solución versátil capaz de trabajar con mediciones bi y tridimensionales. Otro *software* de interés es el paquete *BLAM!* que logra realizar un mapa 3D también a partir de nubes de puntos [33].

Otra estrategia común en el campo consiste en emplear una estrategia de *SLAM* que se sustenta en un sensor 2D y, posteriormente, completar el mapa generado con la información de un sensor 3D. El equipo de robótica *HECTOR*[34] de la Universidad Técnica de Darmstad presentó su propuesta en la que combina su propio paquete de *SLAM* con un mapeo 3D a partir de una cámara *RGB-D,*[35]. Su algoritmo de mapeo y localización, descrito por Kohlbrecher *et al.* en [36] e implementado en el paquete ROS *hector_slam* [37], se basa en un proceso de *scan-matching* para estimar la posición apoyado en una implementación de Gauss-Newton[38].

La solución finalmente implementada tiene su base en esta idea pero el algoritmo de *SLAM* empleado será el presentado por G. Grisetti, C. Stachniss y W. Burgard en [39] y [40]. Las bases teóricas de este serán detalladamente explicadas en la sección 2.1, así como el uso de la librería *Gmapping*[41] que lo implementa en la sección 4.6.

Para complementar este método, se deberá seleccionar una estrategia para la creación de mapas tridimensionales. La resolución de este problema pasa por realizar análisis probabilísticos de las lecturas de los sensores en cada posición. Entre las distintas alternativas disponibles en ROS, cabe mencionar a *Libpointmatcher* [42],[43] que hace uso de un algoritmo *Iterative Closest Point* (*ICP*) [44] para gestionar las nubes de puntos. También resulta de interés la *Point Cloud Library* [45], una librería muy extendida y compatible con el sistema operativo ROS [46].

Al igual que en la propuesta del *Team HECTOR,* en este trabajo se ha optado por apoyarse en la librería *Octomap* [47],[48]. La principal característica de este método es el uso de *octrees*[49] como estructuras de datos. Los detalles sobre las bases teóricas del algoritmo y del paquete que lo implementa pueden encontrarse en el apartado 2.2 y en el apartado 4.7, respectivamente.

Una vez se tiene un mapa con los obstáculos se debe afrontar el problema de calcular la trayectoria para llegar a un punto objetivo de manera autónoma. La planificación de trayectorias suele dividirse en dos módulos. Por un lado, el planificador global, que emplea un modelo completo del entorno para generar la trayectoria general, y, por otro, el planificador local, que utiliza un mapa de dimensiones más reducidas y se emplea para esquivar obstáculos, atendiendo a los posibles cambios dinámicos en el entorno e intentando mantenerse fiel a la ruta global.

Refiriéndonos en primer lugar a los planificadores globales, existe en una serie de métodos que se apoyan en la generación de *roadmaps*[2, Sec. 5.2.3], estructuras de información que almacenan las posiciones intermedias de una trayectoria y las rutas que las unen en forma de grafos. Este tipo de estrategias son las seguidas en propuestas como las de J. Canny [50] y las de P. Bhattacharya *et al.*[51]. Algunos métodos van un paso más allá asignando pesos a cada región del *roadmap* y tratando de buscar el camino con coste menor. Es el caso de los algoritmos *Best First* [52], *A** [53] y Dijkstra [54]. Otro tipo de planificadores globales se basan en la división del espacio en pequeñas celdas. A este tipo

pertenece la exploración rápida de *random trees* [55] e implementaciones que hacen uso de una malla de neuronas [56] o campos potenciales [57]. Ambas maneras de afrontar el problema se ven comparadas en este artículo de Milos Seda[58].

Volviendo a la clasificación inicial, los planificadores locales son aquellos que generan un nuevo mapa, de tamaño más pequeño, y que planifica una nueva ruta teniendo en cuenta nuevos obstáculos y las limitaciones y dimensiones del vehículo. Cabe destacar la importancia que este planificador juega en la navegación pues será, en última instancia, el encargado de definir las posiciones que definen la ruta definitiva.

Centrándonos en aquellos que tienen una implementación en ROS, cabe destacar el *Dynamic Windows Approach* introducido en [59] por D. Fox, W. Burgard, y S. Thrun. El algoritmo *DWA* se encuentra especialmente diseñado para lidiar con las restricciones dinámicas del robot, que derivan en las velocidades y aceleraciones límites. El planificador, actuando de manera local, aproxima la trayectoria por curvaturas circulares que resulta en un espacio bidimensional de búsqueda de velocidades traslacionales y rotacionales. Esta búsqueda se reduce a aquellas velocidades que el robot puede alcanzar en el siguiente intervalo de tiempo y que permiten al robot detenerse de manera segura.

Buscando la maximización de una función objetivo se selecciona una combinación de esas velocidades. Además de las velocidades, la función también incluirá factores como una medida del avance hacia la posición objetivo y la distancia al siguiente obstáculo en la trayectoria.

Otro planificador de gran interés será el sustentado en *Elastic Bands* [60] presentado por S. Quinlan y O. Khatib. Este deforma la ruta inicial como si de una goma elástica se tratara, donde las fuerzas de atracción y repulsión son ejercidas por objetivos y obstáculos respectivamente. Será una evolución de este método, los *Time Elastic Band*, el planificador finalmente empleado en este trabajo, cuya teoría se desarrolla en la sección 2.3 y su implementación se detalla en la 4.8.2.

Repasados los distintos componentes necesarios para la navegación autónoma, cabe hacer mención a las distintas alternativas a la hora de asegurar la total inspección de un área, en este caso una chapa de metal.

Este tipo de algoritmos, denominados comúnmente como *Coverage Path Planning*, se pueden dividir en dos clases claramente diferenciadas: *online* y *offline*. Al primer grupo pertenecen aquellas estrategias en las que no se tiene conocimiento a priori del mapa y este se va descubriendo a medida que se avanza por el entorno. Las estrategias *offline*, sin embargo, parten de un mapa conocido que se debe cubrir por completo. Dado que las dimensiones de la chapa son, *a priori*, desconocidas y con el fin de alcanzar la solución más versátil se exploraron solo los métodos *online*.

Este tipo de métodos pueden dividirse, a su vez, en: algoritmos de descomposición basados en el entorno y algoritmos de división del área en una malla. Los primeros dividen la totalidad de la superficie en grandes áreas que se pueden recorrer fácilmente. Los segundos descomponen el área en pequeñas celdas idénticas que se van visitando individualmente.

Existen un número considerable de estrategias de descomposición del área. Estrategias como la descomposición trapezoidal [61] y la descomposición bustrófedon [62] se limitan a la división del área con obstáculos poligonales. La descomposición Morse [63] sí contempla objetos no poligonales, pero tiene problemas con obstáculos paralelos al avance del robot. La estrategia de descomposición en *slices* [64] ofrece la mayor versatilidad en este sentido al poder lidiar con obstáculos tanto rectilíneos como curvos. Este tipo de algoritmos resultan menos susceptibles a errores odométricos y conllevan un menor gasto computacional.

Entre los algoritmos de descomposición en mallas puede destacarse el método *Spiral-STC* [65] que cubre completamente el área realizando un recorrido espiral. Otro método de interés es propuesto por S. X. Yang y C. Luo [56]. Este establece una malla en la que cada celda se representa por una neurona. Cada neurona atrae, de manera global, al robot si está sin inspeccionar y lo repele, localmente, si se trata de un obstáculo.

El artículo de Galceran y Carreras[66] realiza un estudio general sobre el *Coverage Path Planning* y puede ser un buen punto de partida si se desea conocer más sobre estos aspectos.

## 1.3. Resumen del método

En este apartado se realizará una explicación resumida de los algoritmos implementados para llevar a cabo la estrategia general de navegación, Figura 7.

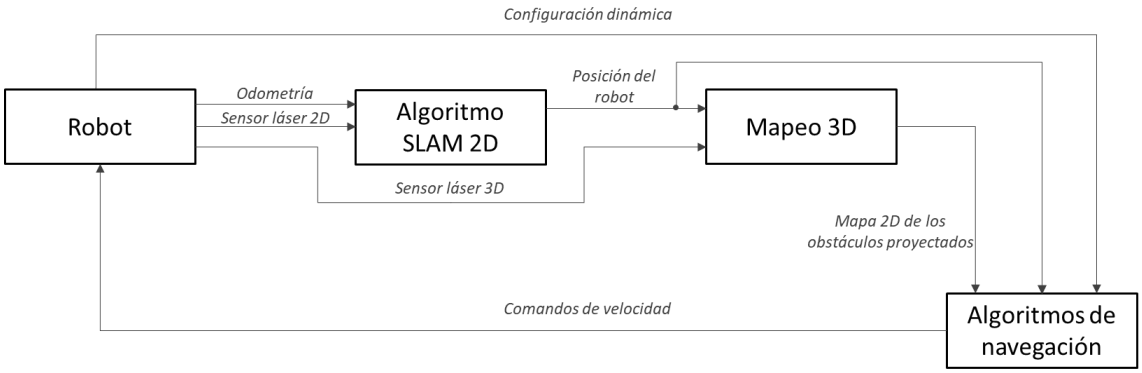


Figura 7: Esquema general de la estrategia general de navegación

Como puede verse, el algoritmo de SLAM determinará la posición del robot a partir de la odometría del robot y de la lectura de un sensor láser 2D que obtenga información del espacio de trabajo. Para ello se ha empleado una estrategia basada en filtro de partículas del tipo *Rao-Blackwellized* [39], [40], [41].

Dadas las características del ambiente será importante garantizar que el robot no colisiona con obstáculos que se encuentren en el suelo, pero tampoco con posibles obstáculos en voladizo. Este requisito obliga a la elaboración de un mapa 3D del entorno que permita dividir el espacio en un conjunto de celdas ocupadas o libres. De esta manera, la posición del robot será empleada para relacionarse con la nube de puntos provenientes de un láser 3D y así generar un modelo del entorno. En el caso que se presenta el mapa se genera siguiendo el algoritmo *Octomap* [47].

Este mapa se proyectará al plano bidimensional para, junto a la posición del robot y las características dinámicas del mismo servir como entrada al bloque de planificación de trayectorias. Con la información recibida, los algoritmos de navegación podrán enviar comandos de velocidad al robot con el fin de alcanzar de manera segura distintas posiciones objetivo. Esta labor será acometida en este trabajo por un planificador *Time Elastic Bands* [67],[68].

Para poder coordinar los distintos comportamientos se desarrollará una máquina de estados que permite reflejar la lógica de funcionamiento del robot. Esta servirá, además, como una interfaz hombre-máquina mediante la cual un operario podrá interactuar con el proceso.

Como ya se ha comentado, la implementación de todas estas estrategias se realizará bajo el sistema operativo ROS [69]. Este entorno permite comunicar fácilmente el conjunto de librerías garantizando la abstracción entre los distintos módulos y facilitando la implementación de funcionalidades de uso común y el paso de mensajes entre procesos. La explicación de los conceptos básicos de este *software* se relatará en el *Anexo I*. El uso de librerías o tipos de mensajes ya implementadas se irá referenciando a lo largo del documento.

Haciendo referencia a la estructura del presente informe, en primer lugar, en el capítulo del *Marco teórico* se explicarán los fundamentos matemáticos de los algoritmos empleados en el proyecto. En el siguiente capítulo, *Diseño de la coordinación de comportamientos*, se expondrá el funcionamiento general que se pretende obtener con el robot. Posteriormente, en *Implementación en ROS* se darán detalles sobre la implementación de las distintas librerías de código y cómo se comunican entre sí a través del entorno ROS. Las pruebas y resultados de las distintas ejecuciones se analizarán en la sección de *Experimentos y resultados*. El capítulo de *Discusión* se reservará para realizar un análisis de los resultados incluyendo alternativas futuras y trabajo a realizar. Por último, en *Conclusiones* se realizará un breve repaso al método y a los objetivos cumplidos en el trabajo.

# 2. Marco teórico

## 2.1. *SLAM* con filtro de partículas Rao-Blackwell

Los algoritmos de SLAM surgen ante las necesidades simultáneas de un modelo del entorno para estimar la localización y la de conocer la posición de robot para construir el mapa del entorno que le rodea. La principal dificultad reside en que se trata de problemas complejos intrínsecamente acoplados, lo que se conoce de manera coloquial como un problema de huevo-gallina.

Dada la inevitable incertidumbre que conllevan las aplicaciones de mapeo mediante sensores en robótica, estas se basan en métodos con bases probabilísticas. Una solución ampliamente extendida es el uso de filtros de partículas, un caso especial de filtros Bayesianos. Estos se emplean para estimar el estado de un sistema dinámico a través de lecturas de un sensor.

En cada iteración, para estimar esta variable estado se generan múltiples predicciones de la misma. Una vez generadas se ponderan en función de su veracidad con la última lectura de los sensores. En los filtros de partículas cada muestra o partícula se corresponde, por tanto, con una posible solución del problema.

Tanto las pruebas realizadas en diversas simulaciones como publicaciones académicas [70] invitan a pensar que la mejor solución sería el uso de un método que se apoye en un filtro de partículas del tipo *Rao-Blackwellized* (RBPF) introducidos por K. P. Murphy, A. Doucet *et al.,* en [71] y [72], y mejorados para el *SLAM* por G. Grisetti, C. Stachniss, y W. Burgard en [39] y [40].

El problema de SLAM puede resumirse con la ecuación 1:

$$p(x, m|z, u) \tag{1}$$

Donde $x$ representa a las poses del robot, $m$ el mapa del entorno, $z$ las observaciones realizadas y $u$ la odometría del robot en base al movimiento de sus ruedas.

Las estrategias *RBPF* suelen optar por factorizar la expresión como se ve en la ecuación 2. Esto divide el problema en dos partes como se observa en la ecuación 3. Por un lado, $p(x_{1:t} | z_{1:t}, u_{1:t-1})$ representa un problema clásico de localización en el que la trayectoria del

robot se estima a partir de las lecturas de un sensor láser y de la odometría del robot. Posteriormente, el cálculo del mapa, $p(m \mid x_{1:t}, z_{1:t})$, puede realizarse de manera más sencilla al conocerse la trayectoria y las lecturas de los sensores en cada posición.

$$p(x, m|z, u) = p(x|z, u) * p(m \mid x, z) \tag{2}$$

$$p(x_{1:t}, \quad m \mid z_{1:t}, \quad u_{1:t-1}) = p(x_{1:t} \mid z_{1:t}, u_{1:t}) * p(m \mid x_{1:t}, \quad z_{1:t}) \tag{3}$$

En el problema del SLAM cada partícula representará una trayectoria seguida y, por tanto, gracias a las lecturas de los sensores, se les podrá asociar el mapa generado durante ese recorrido, Figura 8.

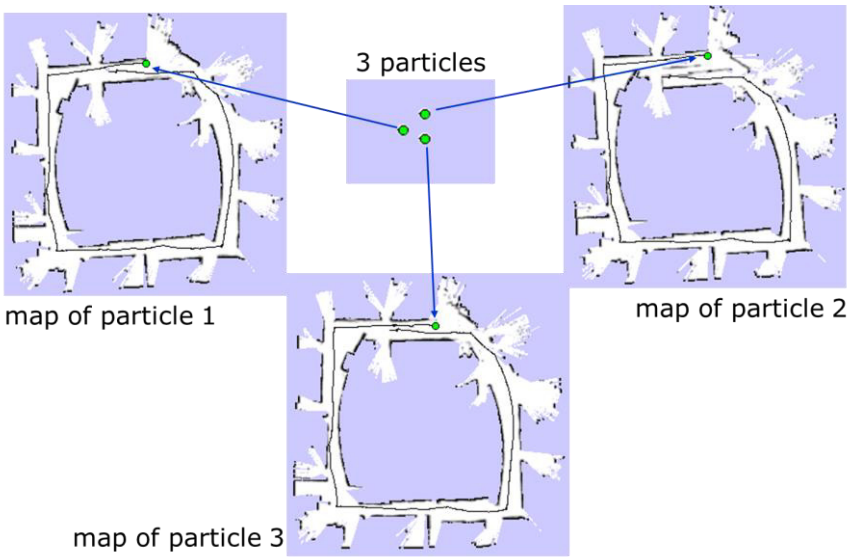


Figura 8: Representación de partículas de un *SLAM* y el mapa generado a partir de ellas. Figura extraída de [73]

Como se ha comentado en el estado del arte, para el cálculo de un mapa conocidas las poses del robot existen múltiples soluciones computacionalmente eficientes que resuelven el problema de manera analítica[13].

Para la estimación de la posición, uno de los algoritmos más extendido para este tipo de filtros de partículas es el algoritmo *Sampling Importance Resampling* (SIR), que debe su nombre a los pasos que lo componen.

En el *sampling*, el algoritmo obtiene las nuevas partículas $\{x_t^{(i)}\}$, pertenecientes a una distribución propuesta π, a partir de las empleadas en la anterior iteración $\{x_{t-1}^{(i)}\}$. Es habitual tomar como distribución un modelo basado en los movimientos realizados por el robot.

En el siguiente paso a cada partícula se le asigna un peso relacionado con su probabilidad de ser correcta. Como se ve en la ecuación 4, este cálculo se realiza como el cociente entre la distribución *target,* que se quiere aproximar, y la distribución *proposal* escogida.

$$w^{(i)} = \frac{p(x_t^{(i)} \mid z_{1:t}, u_{1:t-1})}{\pi(x_t^{(i)} \mid z_{1:t}, u_{1:t-1})} \approx \frac{target(x_t^{(i)})}{proposal(x_t^{(i)})} \tag{4}$$

El último paso antes de la estimación del mapa es el *resampling*. Durante esta etapa las partículas de menos peso son sustituidas por partículas con un peso mayor.

Tras estos pasos, para cada posición estudiada, $x_t^{(i)}$, se calcula el correspondiente mapa basándose en la trayectoria seguida y el historial de observaciones, siguiendo la ecuación 5.

$$p\left(m_t^{(i)} \middle| x_{1:t}^{(i)}, \quad z_{1:t}\right) \tag{5}$$

Las mejoras incluidas en [40] hacen referencia a dos aspectos que quedan abiertos en la propuesta original del mapeo mediante *Rao-Blackwell*: la construcción de la distribución propuesta y las condiciones del remuestreo.

### 2.1.1. Distribución propuesta mejorada

La bondad de la distribución de muestras está directamente relacionada con el desempeño del filtro de partículas. Esta distribución debe aproximar a la real, que se expresa matemáticamente como:

$$p(x_t \mid m_{t-1}^{(i)}, x_{t-1}^{(i)}, z_t, u_t) \tag{6}$$

Fuera del problema del SLAM, *s*i se pudiesen tomar muestras reales para una propuesta ideal nunca sería necesario el remuestreo pues todas las partículas recibirían los mismos pesos.

Una solución muy común es construir la distribución en base a un modelo odométrico del robot. Sin embargo, esta solución resulta subóptima al ser la información del sensor del

robot significativamente más precisa que la posición estimada del robot. Se puede dar por tanto un caso como el de la Figura 9. Muchas de las muestras proporcionadas por el *motion model* o modelo odométrico, $p(x|x',u)$, no serán significativas. En concreto todas las que caen fuera de la región $L^{(i)}$ tendrán un valor residual en contraste con la información aportada por el sensor, $p(z|x)$

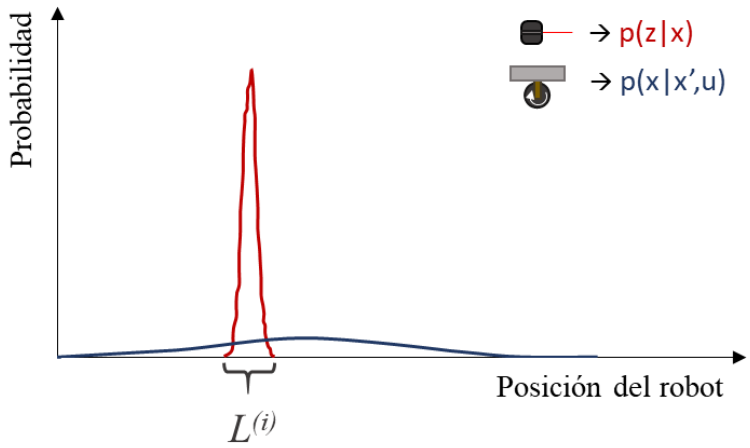


Figura 9: Representación de las probabilidades extraídas de la lectura de los sensores(rojo) y de las medidas odométricas (azul). En el intervalo $L^{(i)}$ se ve cómo el producto de ambas funciones está dominado por la primera.

Para afrontar este problema una opción es tener en cuenta las observaciones más recientes de los sensores al generar el siguiente número de muestras. De esta manera se logra centrar la generación de muestras en torno al área de máxima probabilidad, $L^{(i)}$ en la anterior figura. De acuerdo con Doucet, [71], la distribución puede expresarse por tanto como:

$$p\left(x_t \middle| m_{t-1}^{(i)}, x_{t-1}^{(i)}, z_t, u_{t-1}\right) = \frac{p\left(z_t|m_{t-1}^{(i)}, x_t\right) * p\left(x_t|x_{t-1}^{(i)}, u_{t-1}\right)}{p\left(z_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, u_{t-1}\right)} \quad (7)$$

La fórmula del peso equivalente, cuyo desarrollo puede encontrarse en [40], se expresa ahora como:

$$w_t^{(i)} = w_{t-1}^{(i)} * \int p(z_t|x') * p(x'|x_{t-1}^{(i)}, u_{t-1})dx' \quad (8)$$

Sin embargo, esta solución sigue arrastrando en parte el problema comentado en la Figura 9. En el proceso de SLAM, en primer lugar se deberá generar un conjunto de poses $x_j$

posibles a partir del movimiento del robot, $p(x_t|x^i_{t-1}, u_{t-1})$. A continuación, esas muestras recibirán un peso a partir de las observaciones de los sensores. No obstante, si las observaciones del sensor mantienen una forma tan impulsional, el número de muestras a partir del movimiento deberá ser muy alto con el fin de cubrir la zona $L^{(i)}$ con una cantidad lo suficientemente grande.

Los múltiples experimentos llevados por los autores han llevado a concluir que la distribución óptima tiene un número limitado de máximos, siendo muy común que solo cuenten con uno. A partir de esta conclusión se puede tomar la decisión de forzar las muestras a situarse cercanos a ese punto de máxima probabilidad del modelo observacional, suponiendo de manera adicional un ahorro en el gasto computacional.

Para generar de manera eficiente la siguiente generación de partículas se realiza una aproximación gaussiana, $\mathcal{N}$, definida por su media, $\mu^{(i)}_t$, y su varianza, $\Sigma^{(i)}_t$. De esta manera se establecerá que la posición resultante equivale al máximo de esta distribución La principal diferencia con el resto de los algoritmos es que, en este caso, se analizan los datos del sensor de manera previa para determinar el área de mayor probabilidad.

Para cada partícula *i*, la media y la varianza, $\mu^{(i)}_t$ y $\Sigma^{(i)}_t$, se determinan de manera individual para las *K* partículas creadas dentro del intervalo $L^{(i)}$.

$$\mu_t^{(i)} = \frac{1}{\eta^{(i)}} * \sum_{j=1}^{K} x_j * p\left(z_t|m_{t-1}^{(i)}, x_j\right) * p\left(x_j|x_{t-1}^{(i)}, u_{t-1}\right) \tag{9}$$

$$\Sigma_t^{(i)} = \frac{1}{\eta^{(i)}} * \sum_{j=1}^{K} p\left(z_t|m_{t-1}^{(i)}, x_j\right) * p\left(x_j|x_{t-1}^{(i)}, u_{t-1}\right) * \left(x_j - \mu_t^{(i)}\right)\left(x_j - \mu_t^{(i)}\right)^T \tag{10}$$

Con el factor de normalización

$$\eta^{(i)} = \sum_{j=1}^{K} p\left(z_t|m_{t-1}^{(i)}, x_j\right) * p\left(x_j|x_{t-1}^{(i)}, u_{t-1}\right) \tag{11}$$

De esta forma se obtiene una proposición cercana a la óptima que permite obtener de manera eficiente la siguiente generación de partículas. El peso podrá calcularse como:

$$w_t^{(i)} = w_{t-1}^{(i)} * \eta^{(i)} \tag{12}$$

### 2.1.2. Remuestreo adaptativo

El paso de *resampling* es crucial en una buena aplicación de un filtro de partículas. Resulta necesario, dado que el método solo es posible con un número finito de predicciones, pero un resampleo demasiado duro puede eliminar buenas muestras, llegando a provocar lo que se conoce como un agotamiento de las muestras o *sample depleiton*.

Esta propuesta sigue la formulación propuesta por A. Doucet *et al.* [74]. El número efectivo de muestras puede calcularse en base a la ecuación 13.

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} (\widetilde{w}^{(i)})^2} \tag{13}$$

Donde $N$ representa el número total de partículas y $\widetilde{w}^{(i)}$ el peso normalizado de la partícula $i$. El valor de $N_{eff}$ puede interpretarse como una medida de la dispersión de los pesos de las partículas por lo que permite medir cómo un conjunto de muestras se aproxima a la realidad. Siguiendo la propuesta de Arnaud Doucet, el *remuestreo* se producirá cuando el valor de $N_{eff}$ caiga por debajo de N/2, como se representa en la ecuación 14.

$$N_{eff} < \frac{N}{2} \rightarrow Resampling \tag{14}$$

### 2.1.3. Resumen del algoritmo

Explicados de manera general los conceptos se realizará un repaso de todo el proceso con el fin de asentar una visión global del algoritmo.

El proceso comienza cada vez que un nuevo conjunto de medidas, $(u_{t-1}, z_t)$, está disponible. Una estimación inicial de la pose del robot $x'^{(i)}_t$ se calcula a partir de la pose anterior $x^{(i)}_{t-1}$ y las medidas odométricas, $u_{t-1}$.

Basándose en esta suposición inicial $x'^{(i)}_t$ y en el mapa $m^{(i)}_{t-1}$ se desarrolla un algoritmo de *scan-matching*, [75]. Este algoritmo desencadena en la delimitación de un área de interés dentro de $x'^{(i)}_t$. En el caso de que el *scan-matching* fracase, la posición y los pesos se calculan solamente de acuerdo al *motion model.*

Alrededor del área de interés se seleccionan un conjunto de partículas. Basándose en esas muestras se calculan las medias, ecuación 9, y las varianzas, ecuación 10, de acuerdo a la

distribución objetivo. A partir de ellas puede construirse la aproximación gaussiana, $\mathcal{N}(\mu^{(i)}_t, \Sigma^{(i)}_t)$, de la que se tomará la posición estimada del robot.

Por último, se actualizan los pesos y se calcula el mapa, $m$, de acuerdo con la pose escogida y las observaciones $z_t$.

Antes de calcular la siguiente generación de muestras se llevará a cabo un resampleo en función del valor de $N_{eff}$, ecuaciones 13 y 14.

## 2.2. Mapeo 3D del entorno basado en *octrees*

Como ya se ha comentado, será importante en este proyecto la construcción de un mapa 3D. Sin él, el sensor láser 2D no bastará para poder detectar obstáculos que no cortan el plano del sensor pero que sí podrían producir una colisión con el robot, como se muestra en la Figura 10. Este problema cobra mayor importancia en un ambiente industrial en el que son habituales pequeños obstáculos en el suelo y elementos en voladizo.

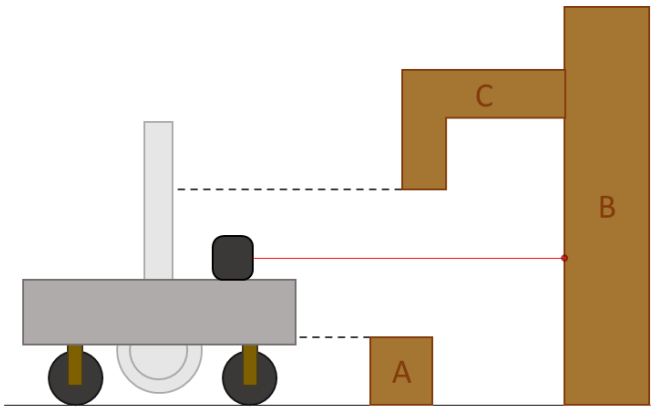


Figura 10: Representación de un escenario en los que los obstáculos *A y B* no serían detectados por el sensor y podrían suponer un peligro para la base móvil.

Como ya ha sido explicado, el algoritmo empleado para esta tarea en el proyecto será *Octomap*, introducido por Hornung *et. al* en[47]. El método nace con el fin de cumplir con tres objetivos generales: la representación probabilística, el modelado de áreas desconocidas y la eficiencia computacional. Para cumplir con ello hace uso de estructuras denominadas *octrees*[49].

Los *octrees* son estructuras en árbol de datos, enfocadas a la representación 3D, que se subdividen exactamente en 8 hijos de igual tamaño. Cada uno de los nodos del árbol representa el volumen contenido en un cubo, al que se suele referir como *voxel*. La

resolución del mapa se define como el tamaño mínimo del *voxel*, es decir, aquel a partir del cual ya no se admiten más subdivisiones.

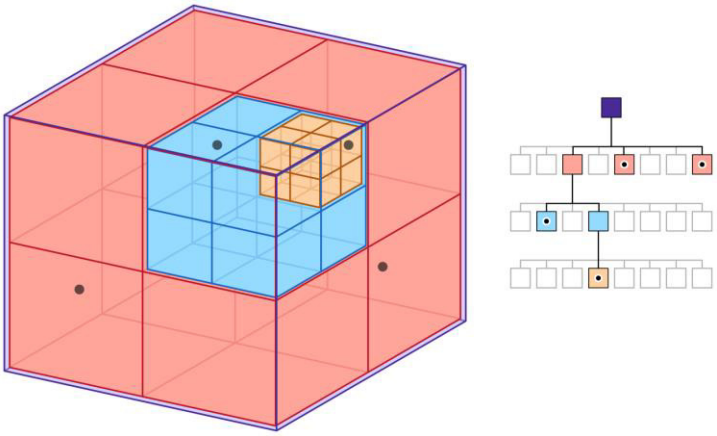

Figura 11: Representación volumétrica y en forma de árbol de un *octree*. Figura extraída de [76].

La lectura de los sensores y de qué manera determinan si un espacio está ocupado o no se integra en este método siguiendo el trabajo de Moravec y Elfes[77]. La probabilidad de que un nodo, *n*, este ocupado se calcula en base al histórico de mediciones realizadas, $z_{1:t}$, como:

$$P(n|z_{1:t}) = \left[1 + \frac{1 - P(n|z_t)}{P(n|z_t)} * \frac{1 - P(n|z_{1:t-1})}{P(n|z_{1:t-1})} * \frac{P(n)}{1 - P(n)}\right]^{-1} \quad (15)$$

Como puede verse, la actualización depende de las observaciones realizadas, $z_{1:t}$, la estimación anterior, $P(n|z_{1:t-1})$, de la probabilidad de estar ocupado en base al sensor empleado, $P(n|z_t)$, y de, $P(n)$, una probabilidad establecida de manera previa.

Para la probabilidad del sensor, en la implementación se establece una en caso de que el rayo láser haya sido rebotado por un objeto y otra en el caso de que no. Asumiendo para la probabilidad $P(n)$ un valor del 50%, la expresión general puede expresarse en notación *log-odds*[78] como:

$$L(n|z_{1:t}) = L(n|z_{1:t-1}) + L(n|z_t) \quad (16)$$

Donde $L(n)$ es:

$$L(n) = log\left[\frac{P(n)}{1 - P(n)}\right] \quad (17)$$

Cuando estos mapas se aplican a navegación es habitual establecer un umbral en su probabilidad, ecuación 15, a partir del cual un *voxel* se considera ocupado.

Con las ecuaciones presentadas hasta ahora, para cambiar el estado de un *voxel* sería necesario registrar al menos tantas observaciones a favor del estado nuevo como se han observado a favor del antiguo. Si bien esta condición no es perjudicial si se trabaja con entornos completamente estáticos no resulta favorable si se trabaja con un entorno cambiante y se desea que el mapa se adapte rápidamente a estos cambios. Para alcanzar esta adaptabilidad se adopta la política de actualización introducida por Yguel *et al.* en [79]. Esta propuesta incluye unos límites superiores e inferiores, representados por $l_{max}$ y $l_{min}$, que dejan la fórmula para el cálculo del estado, expresada de manera logarítmica, de la siguiente manera:

$$L(n|z_{1:t}) = max(min(L(n|z_{1:t-1}) + L(n|z_t), l_{max}), l_{min}) \quad (18)$$

De esta manera se limita el número de detecciones necesarias para cambiar el estado de un nodo permitiéndose una actualización dinámica del mapa.

Estos cálculos se realizan para los nodos de menor volumen y se extrapola para los nodos jerárquicamente superiores. El cálculo de la probabilidad del nodo padre puede calcularse como la probabilidad media de sus 8 hijos o, de manera más conservadora y recomendable para aplicaciones de navegación, tomar la mayor probabilidad de ocupación entre las de los nodos hijos.

Una estrategia habitual para reducir la información redundante de *octrees* es la no división de nodos cuyos hijos tienen valores idénticos, es decir, el corte de las ramas que descienden de él, Figura 12. Si bien esto es perfectamente posible en nodos que almacenan valores discretos, resulta complicado cuando estos valores se corresponden con lecturas de sensores de naturaleza ruidosa. La solución implementada asume que aquellos nodos que alcancen las probabilidades $l_{max}$ o $l_{min}$ pasan a considerarse como estables, al haberse registrado su estado con mucha confianza. Se podrá obviar la división de un nodo si todas sus divisiones tienen el mismo estado estable. Mediciones futuras de

los sensores que contradigan este estado podrán devolverle al árbol jerárquico su rama seccionada.

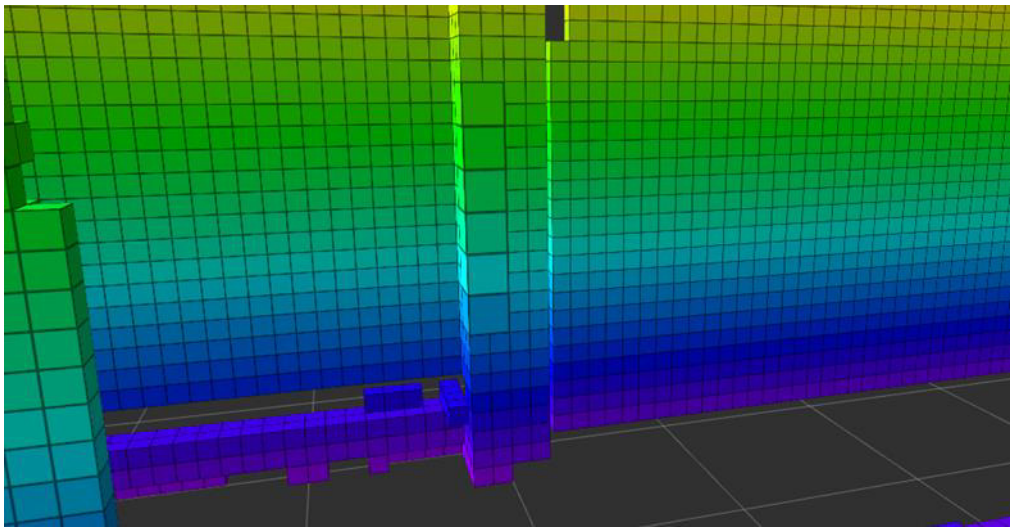

Figura 12: Ejemplo en el que pueden apreciarse nodos cuya estabilidad ha propiciado que no se subdividan hasta el tamaño mínimo. El código de colores solo indica a qué altura se encuentran los puntos.

En resumen, el método consigue cumplir con los tres objetivos establecidos de representación probabilística, eficiencia en el uso de memoria y capacidad para representar los volúmenes desconocidos.

Por último y como se verá más adelante, el paquete ROS que implementa *Octomap*[48] incluye la posibilidad de realizar una detección del suelo y, de esta manera, obviarlo en la construcción del mapa. De esta manera, la superficie plana se guardará como espacio libre mientras que las estructuras que abandonen esa planitud sí aparecerán como obstáculos. La detección de los puntos que se alinean formando aproximadamente un plano se lleva a cabo mediante el uso del método *Random Sample Consensus*, o *RANSAC*, introducido por M. A. Fischler y R. C. Bolles en [80].

## 2.3. Navegación basada en *Time Elastic Bands*

Una vez se tiene el mapa se deben poder planificar sobre él rutas seguras evitando los obstáculos. Además de evitar colisiones, los caminos deben ser lo más eficientes posibles, atendiendo a las velocidades y aceleraciones del robot, y manteniéndose coherentes con las características dinámicas de la base móvil.

Para llevar a cabo esta tarea que combina el *path planning* con el control del robot se ha optado por un planificador *Time Elastic Bands* (*TEB*)[68], [67], que nace como una mejora de los planificadores *Elastic Bands*[60]. Mientras la versión original solo atiende a la presencia de obstáculos y al cálculo del camino más corto, los planificadores *TEB* tienen en cuenta restricciones de carácter temporal como pueden ser las velocidades y aceleraciones límites del robot.

El método original de *Elastic Bands* se genera como una sucesión de, únicamente, poses u objetivos intermedios que se englobará dentro del conjunto $Q$.

$$Q = \{s_i\}_{i=0:n} \quad n \in \mathbb{N} \tag{19}$$

$$s_i = \begin{pmatrix} x_i \\ y_i \\ \theta_i \end{pmatrix} \tag{20}$$

El algoritmo *TEB* va un paso más allá añadiendo los incrementos de tiempo($\tau$) entre las distintas poses, como se representa en la Figura 13.

$$\tau = \{\Delta T_i\}_{i=0:n-1} \tag{21}$$

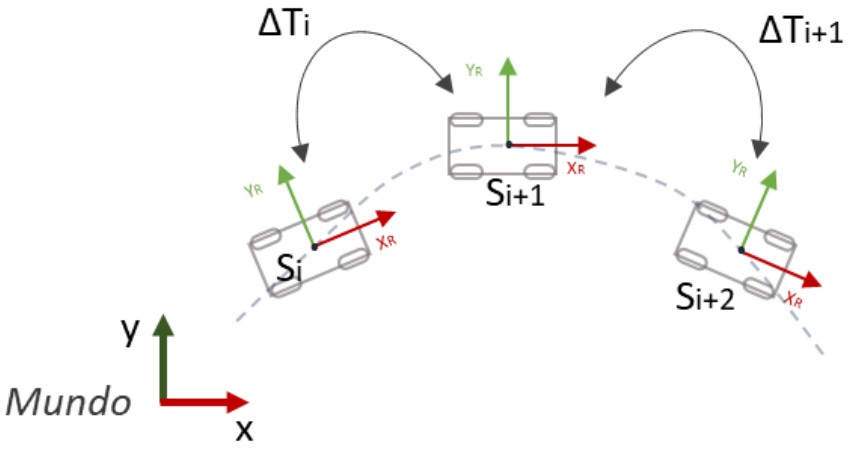


Figura 13: Representación de las secuencias de poses (posición y orientación del robot) e incrementos temporales que construyen las *TEBs*.

Toda esta información puede contenerse en el conjunto $B$.

$$B \coloneqq (Q, \tau) \tag{22}$$

Para obtener esta trayectoria que conecta la posición inicial y la final de la manera más eficiente se construye una función objetivo general como la suma ponderada de distintos componentes $f_k$. En los próximos párrafos se describirá cómo se construyen estas funciones y de qué manera representan los factores que afectan a la navegación del robot. La sucesión de poses intermedias óptima, $B^*$, se obtendrá, por tanto, optimizando esta función multiobjetivo.

$$f(B) = \sum_k \gamma_k f_k(B) \tag{23}$$

$$B^* = \underset{B}{\operatorname{argmin}}\, f(B) \tag{24}$$

Se pueden diferenciar dos tipos de funciones objetivo dentro de la TEB. Por un lado, restricciones, como los límites de aceleración y velocidad y, por otro, objetivos, que hacen referencia a la trayectoria como el camino más corto o a la evitación de obstáculos.

La implementación de este tipo de algoritmos de optimización no está muy extendida en entornos de desarrollo robóticos. En base a esto, las restricciones anteriormente mencionadas se formulan en la implementación como objetivos representados por ecuaciones definidas a trozos que penalizan la violación de esa restricción.

$$e_\Gamma(x, x_R, \epsilon, S, n) \approx \begin{cases} \left(\dfrac{x - (x_R - \epsilon)}{S}\right)^n & si\ x > (x_R - \epsilon) \\ 0 & en\ otro\ caso \end{cases} \tag{25}$$

Donde $x_r$ hace referencia a la restricción, $S$ es un factor de escala, $n$ hace referencia al orden polinómico y $\epsilon$ representa un margen de referencia.

En consecuencia, *waypoints* y obstáculos se implementan con la misma función, pero con el empleo de distintos parámetros. Como se puede ver en la ecuación 27 el parámetro a optimizar y el valor límite se introducen de manera negativa para obtener el efecto de la restricción. Desde una interpretación más visual, los *waypoints* actuarían como una fuerza de atracción sobre la banda elástica mientras que los obstáculos lo harían la fuerza de repulsión.

$$\Gamma_{path} = e_\Gamma(d_{j_{min}}, r_{p_{max}}, \epsilon, S, n) \tag{26}$$

$$\Gamma_{obstacle} = e_{\Gamma}(-d_{j_{min}}, -r_{o_{max}}, \epsilon, S, n) \tag{27}$$

Teniendo en cuenta la estructura de una TEB, Figura 13, las restricciones de velocidad y aceleración deben traducirse a restricciones geométricas. Los límites de velocidad deberán expresarse, por tanto, en función de distancias euclídeas o angulares, entre dos posiciones u orientaciones, y de intervalos de tiempo para realizar la transición entre ambas poses. La manera de establecer esta relación puede verse en las ecuaciones 28 y 29.

$$v_i = \frac{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}}{\Delta T_i} \tag{28}$$

$$\omega_i = \frac{\theta_{i+1} - \theta_i}{\Delta T_i} \tag{29}$$

De manera análoga, la relación con las limitaciones de aceleración puede expresarse como:

$$a_i = \frac{2(v_{i+1} - v_i)}{\Delta T_i + \Delta T_{i+1}} \tag{30}$$

Por último, no debe olvidarse que el objetivo de las *Time Elastic Bands* es encontrar el camino más rápido. Este fin puede representarse de manera sencilla como la suma al cuadrado de todos los intervalos temporales intermedios.

$$\Gamma_k = \left( \sum_{i=1}^{n} \Delta T_i \right)^2 \tag{31}$$

Como se explicó en la ecuación 13, cada uno de estos aspectos compone distintas funciones objetivo que combinadas con sus correspondientes pesos componen la función objetivo global.

La manera de afrontar este problema es mediante la transformación de la función multiobjetivo en un hipergrafo[81] y su posterior optimización. Un hipergrafo se diferencia de un grafo convencional en que una arista puede conectar a 2 nodos o más entre sí. Como puede verse en la Figura 14, las configuraciones del robot ($S_i$), los obstáculos($o_i$) y los incrementos de tiempo ($\Delta T_i$) componen los nodos del hipergrafo.

Las aristas, o hiper-aristas en este caso, representan las distintas funciones objetivo que afectan a distintos puntos de la trayectoria de manera local. Como se observa, las aristas conectarán aquellos vértices que influyan a su función objetivo como parámetros. De esta manera, por ejemplo, se puede apreciar como las funciones objetivo de velocidad, en azul marino, dependen del incremento de tiempo y de las dos posiciones objetivos o como la presencia de obstáculos, en granate, afecta a la posición de las poses intermedias.

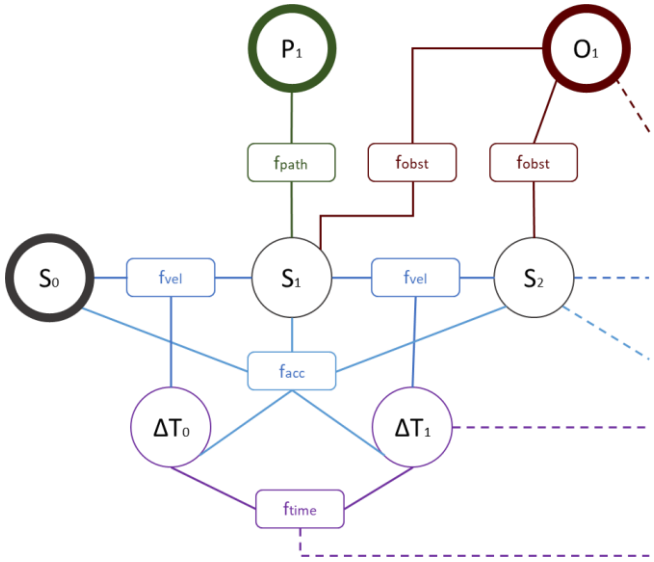


Figura 14: Representación simplificada del hipergrafo generado por el algoritmo TEB

Para obtener la trayectoria se deberá, finalmente, obtener la solución al hipergrafo que minimiza el error. En este caso, el *framework* empleado para ello es el denominado $g^2o$, desarrollado por el grupo *OpenSLAM* [82]. El desarrollo matemático completo de la estrategia se puede encontrar en el artículo de R. Kümmerle *et al.* [83] y una explicación más resumida de la implementación orientada a la resolución del hipergrafo *TEB* en el ya citado artículo de C. Rösmann *et al.* [67, Sec. III].

Explicados ya todos los conceptos de interés se pasará ahora a recapitular el funcionamiento general del algoritmo, empleando la Figura 15. Nótese que en este diagrama la palabra 'Robot' no hace referencia únicamente al elemento mecánico sino también a los sensores acoplados y al resto de algoritmos integrados.

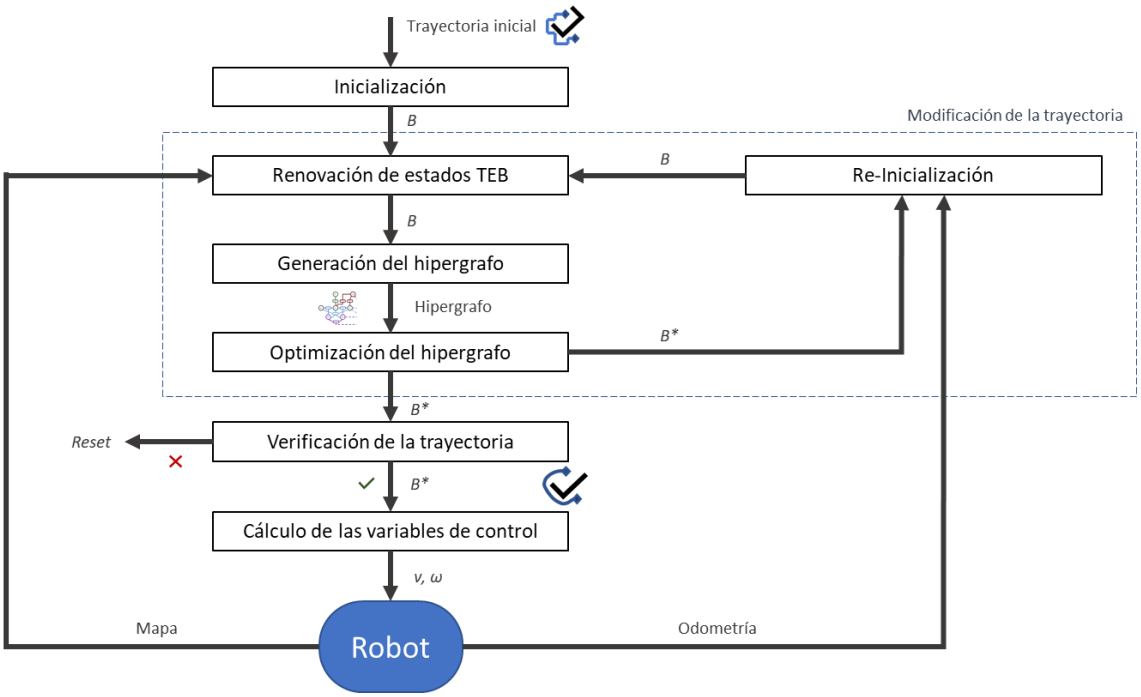


Figura 15: Diagrama de control de la implementación TEB

En la implementación original, el algoritmo partirá de la posición inicial del robot, la posición objetivo hacia la que se dirige y el mapa con los obstáculos detectados. Con esta información es posible calcular, como abordaron Kavraki et al. en [84], una ruta inicial que mantenga una distancia mínima con los obstáculos compuesta por segmentos rectos, de manera similar a como se representa en la Figura 16 a. Sobre esta trayectoria primitiva se generará la primera trayectoria que tendrá en cuenta las limitaciones temporales, *B* en la ecuación 18.

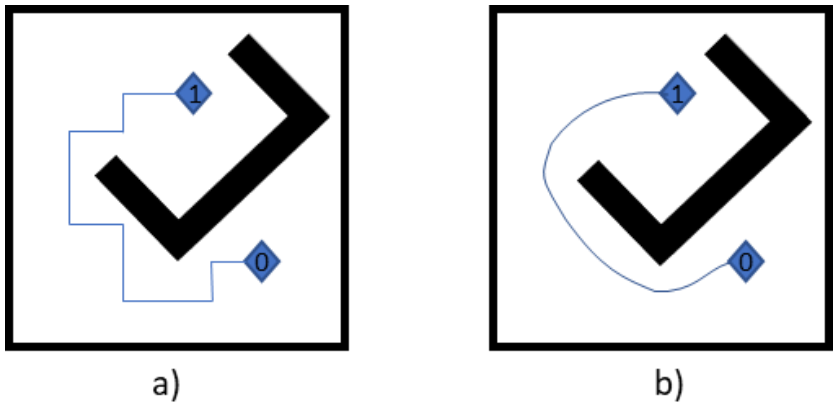


Figura 16: a) *Path* generado inicialmente por el algoritmo original. b) *Path* suavizado mediante el uso de *splines* y fuerzas de contracción y repulsión.

Sin embargo, en la mayoría de las implementaciones es habitual proporcionarle al sistema una ruta global más refinada. En nuestro caso se empleará una trayectoria generada por el planificador Dijsktra [54], el utilizado en la librería por defecto. La explicación teórica de este planificador global se limitará a las referencias aportadas al no considerarse su elección sobre otros algoritmos análogos de una importancia sustancial.

Esta trayectoria se complementa teniendo en cuenta la presencia de obstáculos y el resto de los factores anteriormente mencionados. Es en ese momento cuando se genera el hipergrafo con el fin de buscar la solución más eficiente. La secuencia de poses resultantes de la optimización del hipergrafo será la TEB, $B^*$ en la ecuación 24. En cada iteración se generará un nuevo grafo partiendo de la solución obtenida en la anterior. Factores como nuevos obstáculos o la nueva distancia al objetivo final también serán tenidos en cuenta.

Existen también una serie de restricciones que, en caso de ser vulneradas por la nueva solución, supondrían un reseteo del bucle iniciado partiéndose de nuevo de una solución simple.

Es al final de cada iteración cuando se calculan las variables de control $v$ y $w$ que deben enviarse al robot para que alcance la correspondiente configuración intermedia en la ruta planificada.

Si bien lo explicado hasta ahora sirve para generar trayectorias de navegación autónoma el planificador tendrá tendencia a verse atrapado en trayectorias óptimas a nivel local. Esto puede derivar en problemas como el que se representa en la Figura 17. Ante un obstáculo móvil el planificador continuará modificando su trayectoria para evitar la colisión cuando, en la última viñeta, una ruta por la parte derecha sería mucho más eficiente. Para solucionarlo se integra la extensión propuesta en [85] con la que ahora el algoritmo optimizará varias trayectorias generadas en paralelo y el planificador elegirá la más optima de manera global

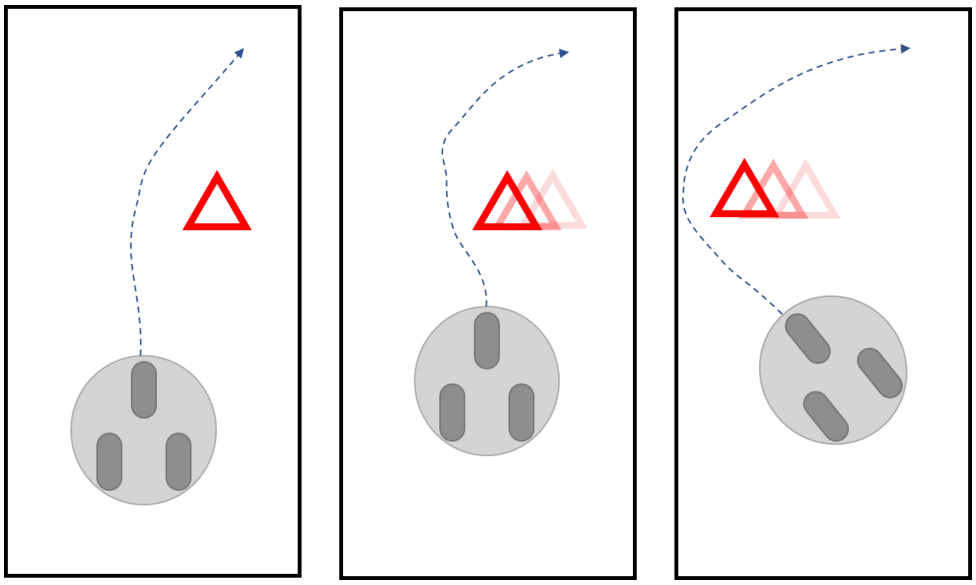

Figura 17: Ejemplo de problema derivado de optimización únicamente local de la trayectoria.

Las distintas rutas se pueden generar a partir del concepto de rutas homotópicas. Dos trayectorias, $T_1$ y $T_2$, con el mismo inicio y final, $P_o$ y $P_f$, serán homotópicas si, y solo si, pueden deformarse de manera continua hasta transformarse la una en la otra sin atravesar ningún obstáculo en ese proceso. Un conjunto de rutas en el que todas son homotópicas entre sí puede recibir el nombre de grupo homotópico.

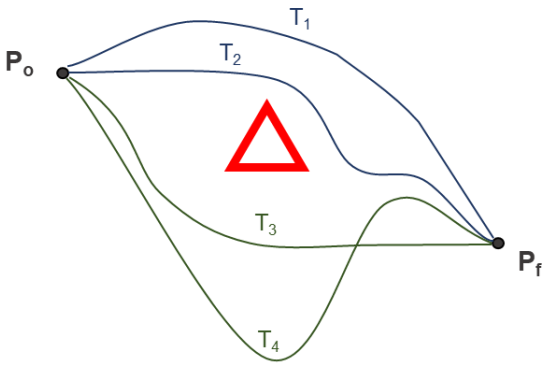


Figura 18: Cuatro trayectorias cuyo color indica a qué grupo homotópico pertenecen.

La generación de este tipo de grupos resulta, no obstante, caro computacionalmente. Por ello, los grupos homotópicos se sustituyen por grupos homólogos, que implican cálculos más sencillos. Sin entrar en detalles, se puede decir que dos trayectorias son homólogas si

resulta posible ir de $P_o$ a $P_f$ a través de $T_1$ y luego de nuevo a $P_o$ a través de $T_2$, sin haber rodeado ningún obstáculo.

En la inmensa mayoría de escenarios de navegación pueden asumirse la definición de homotopía y homología como equivalentes. Por tanto, el algoritmo generará trayectorias pertenecientes a distintos grupos de trayectorias homólogas y, entre las generadas, se escogerá aquella que sea globalmente óptima. Retrotrayéndonos a la Figura 15, la etapa de evaluación global de las trayectorias candidatas se situaría en la posición previa a la de *"Cálculo de las variables de control"*.

La generación de las distintas trayectorias y su posterior optimización se pueden encontrar teóricamente desarrollada en [85, Sec. C] y [85, Sec. D] respectivamente.

# 3. Diseño de la coordinación de comportamientos

Realizado un repaso al trasfondo teórico de los algoritmos empleados se pasará a definir la lógica de funcionamiento del robot. Para ello se utilizará un autómata de estados que incluya los diferentes comportamientos del robot. El diseño base de este autómata puede verse en la Figura 19.

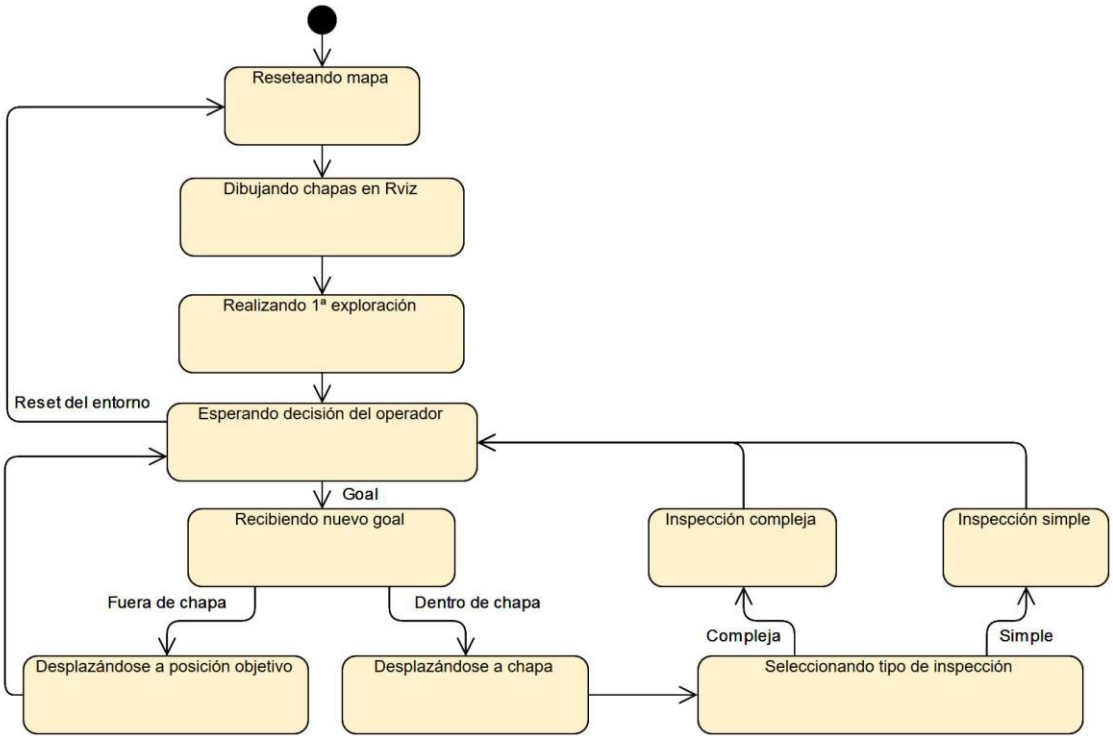


Figura 19:Representación del autómata de estados a implementar

Tras un reseteo del modelo del entorno, evitando así posibles defectos en la inicialización del algoritmo, se procederá a la representación de las chapas en la herramienta de visualización. Una vez hecho esto el robot podrá realizar unos primeros movimientos que le permitan obtener una primera aproximación del mapa que le rodea.

Tras estos estados iniciales, el operador puede seleccionar si realizar una nueva reinicialización del mapeo 3D o si enviar una posición objetivo al robot. En el caso de optarse por la segunda opción, la pose seleccionada se analizaría para comprobar si se encuentra, o no, dentro de una chapa.

Si se ha elegido una pose fuera de una chapa se deberá activar la navegación autónoma del robot hasta alcanzar la posición. Para ello se combinarán las estrategias de *SLAM*, mapeo tridimensional y planificación de trayectorias, ya comentados. Una vez finalizado el desplazamiento se volverá al estado de elección por el operario.

En el caso de que, por el contrario, se seleccione una posición dentro de una chapa también se activará la navegación autónoma, pero, esta vez, para llegar al punto de acceso de la chapa. Llegado el robot a la zona de trabajo se podrá seleccionar qué tipo de inspección se desea realizar entre dos opciones: simple o compleja.

La omnidireccionalidad del vehículo y la intención de emplear un láser cuyo haz sea lo suficientemente ancho como para cubrir todo el ancho de las chapas simplificó los movimientos necesarios para llevar a cabo su inspección.

Volviendo al autómata si se selecciona una *inspección simple* se supondrá la existencia de un láser que abarca toda la chapa. Partiendo de este supuesto, bastará con realizar una única pasada por el centro de la chapa que cubra toda su longitud, como se representa en la Figura 20.

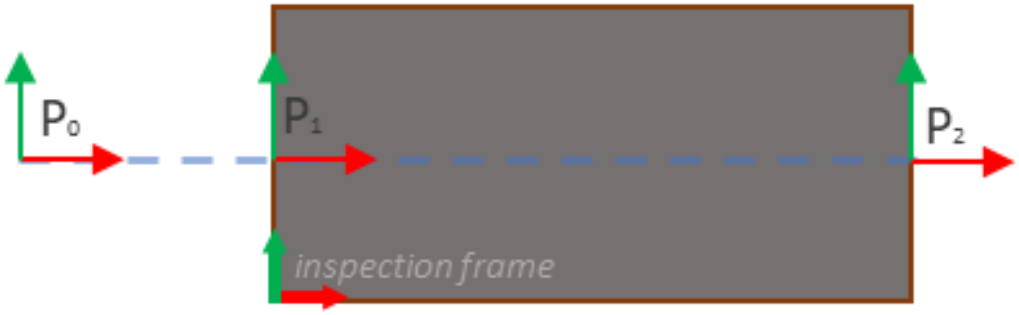


Figura 20: Representación de las tres poses intermedias que se alcanzan para ejecutar una inspección simple de la chapa.

Si se selecciona la *inspección compleja* se realizará un tipo de inspección que permitiría trabajar con sensores de menor amplitud o con chapas mayores. El recorrido de la chapa puede dividirse en dos fases, Figura 21. El robot realizará, en primer lugar, un recorrido en zigzag desplazándose hacia delante, hacia atrás y lateralmente sin rotar el robot. De esta manera, se mantiene siempre el láser hacia la dirección *x* positiva del sistema de coordenadas de la chapa. El recorrido se calcula de tal manera que se garantiza el

cubrimiento de toda el área de la chapa (las zonas sin cubrir en la figura se han dejado para una mayor comprensión del recorrido). En un último movimiento, el robot cubrirá la parte inferior de la chapa que no ha podido cubrir con anterioridad dado que esto le habría supuesto salirse de la chapa. Como esto se ha previsto se ha dejado espacio para recorrer un último carril, en este caso, orientando el láser hacia el eje *y* negativo de la chapa.

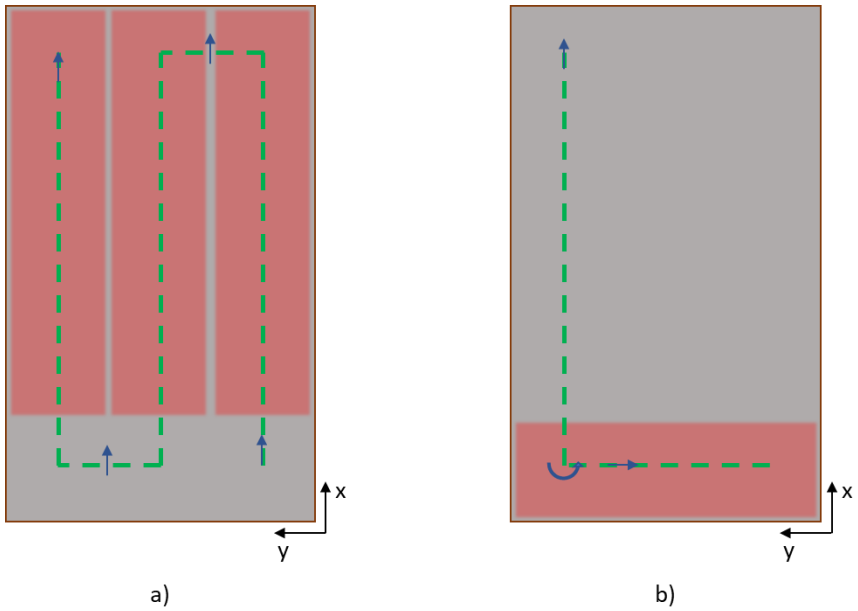


Figura 21: a) Primera fase del recorrido para cubrir la fase. b) Recorrido del último carril horizontal que completa el área. Las flechas azules indican la orientación del robot.

Finalizado el proceso de inspección se retorna al estado en el que el operador podrá seleccionar un nuevo objetivo, resetear el mapa o finalizar la ejecución.

# 4. Implementación en ROS

Esbozada la lógica de funcionamiento que se le pretende dar al robot se pasará ahora a detallar la implementación en ROS de cada uno de los nodos necesarios.

## 4.1. Simulación en Gazebo

La primera herramienta sobre la que se detendrá la explicación es el simulador *Gazebo*[86]. Este software, perfectamente integrado con ROS, permite la creación y edición de entornos pudiendo usarse modelos 3D disponibles en su propia base de datos o generados por el autor, Figura 22.

Además de estos objetos, *Gazebo* cuenta con una amplia librería de diversos sensores y modelos completos de robots creada por la comunidad. Estos modelos además de emular el aspecto y físicas del *hardware* también simulan su comportamiento por medio de *plugins*. Estos se crean de manera que imitan en gran medida la comunicación que se tendría con un dispositivo real facilitando la transición entre simulaciones y ejecuciones reales.

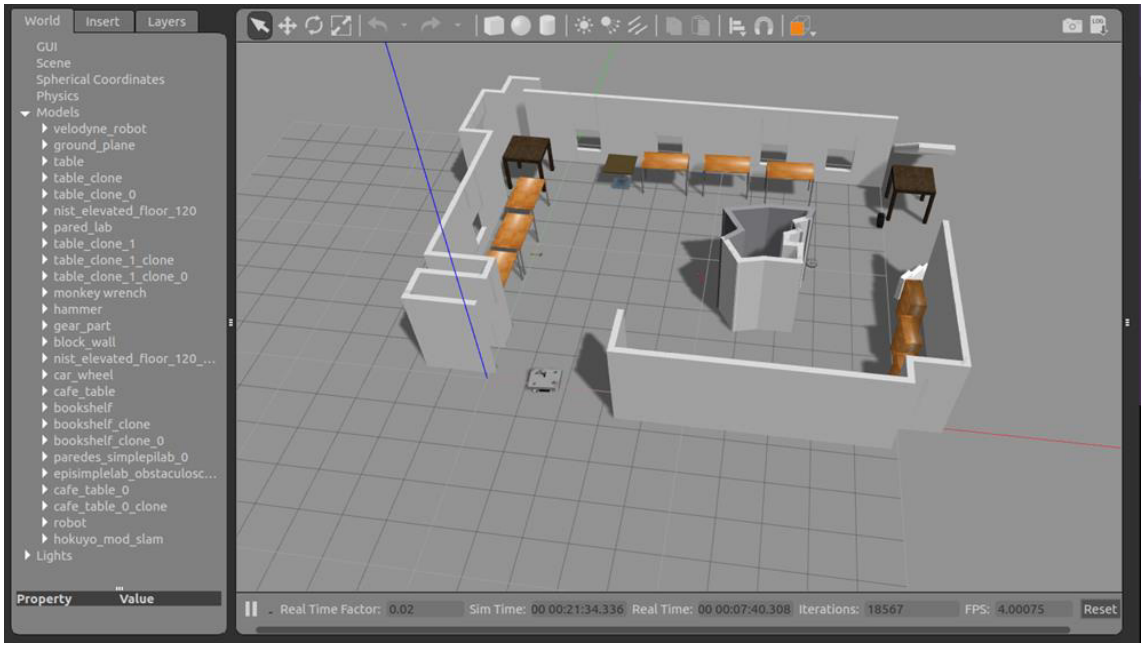


Figura 22: Interfaz de *Gazebo* mostrando uno de los entornos creados

## 4.2. Visualización en Rviz

Otra aplicación de suma importancia es *Rviz*[87]. Esta permite la visualización de los mensajes enviados dentro del sistema, Figura 23. De esta manera se podrá observar de manera mucho más intuitiva la información de, por ejemplo, mapas y sensores.

Además de permitir la visualización de diferentes *topics*, *Rviz* también dispone de diferentes herramientas que permiten interactuar con la imagen introduciendo mensajes en el sistema. Algunas de estas herramientas pueden verse en la parte superior de la interfaz en la Figura 23

Al contrario de lo que ocurre con *Gazebo* el uso de esta aplicación no está ligada exclusivamente a las simulaciones. Las visualizaciones y la interactividad seguirán siendo de gran necesidad cuando se cuente con un robot real.

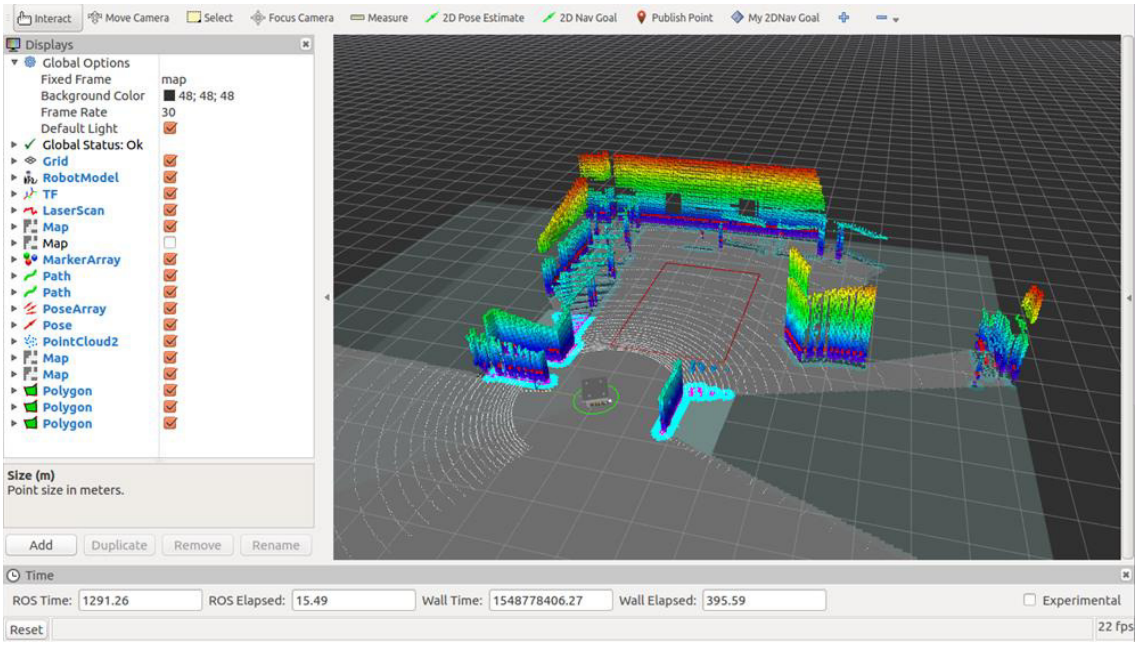


Figura 23: Interfaz de *Rviz* mostrando los mensajes generados en una simulación en el mismo entorno que la Figura 22.

## 4.3. Selección del tipo de robot

Elegir el tipo de movilidad del robot resulta una parte fundamental de este tipo de proyectos. Sin embargo, de igual o mayor importancia resulta su compatibilidad con el sistema ROS. Dado el ambiente industrial en el que operará el robot y el peso elevado con

el que deberá cargar, si se incluyen las baterías y la herramienta de reparación, la tracción y la robustez serán otros factores a tener en cuenta.

Entre los modelos del mercado que se cotejaron se encuentra el *SUMMIT-XL STEEL*[88] de *Robotnik* y el robot *Seekur*[89] de la compañía *Adept*. Ambas bases, además de ser compatibles con ROS, serían capaces de dotar de un comportamiento omnidireccional al robot mediante 4 ruedas suecas en el primer caso y 4 orientables en el segundo.

Otros modelos tenidos en cuenta han sido los modelos de la compañía *Neobotix* [90]. Estos modelos cumplían con todos los requisitos mencionados anteriormente y disponían de suficiente variedad en el que caso de que un tipo de movilidad fuese impuesta por la empresa proyectante.

La decisión final ha consistido en el desarrollo de un robot móvil omnidireccional, capaz de desplazarse en cualquier dirección, de construcción propia. Esta movilidad se conseguirá con el empleo de 3 ruedas orientables.

La selección de robot omnidireccional para el proyecto facilita el trabajo a la hora de navegar y evitar la colisión con obstáculos y será de gran ayuda a la hora de la reparación de defectos. La idea para reparar es el empleo de una muela que se colocará en un lateral del robot y ejercerá más o menos presión en función del tipo de reparación. La posibilidad de, en cualquier momento, desplazarse en cualquier dirección permitirá realizar las reparaciones de manera mucho más rápida que si fuesen necesarias reorientaciones. La selección de ruedas orientables en lugar de ruedas suecas, que también proporcionarían omnidireccionalidad, se debe a una cuestión práctica. Las ruedas suecas basan su funcionamiento en el supuesto teórico de que solo un punto de la rueda hace contacto con el suelo. Si bien son perfectamente útiles para muchas aplicaciones, se ha optado por unas ruedas orientables que proporcionen más sujeción al robot, de manera que se intenta a toda costa evitar que el robot patine mientras realiza la operación de saneado.

En concreto, el modelo de ruedas orientables que se empleará son las *Omni-Drive-Module* de Neobotix[91], Figura 24.

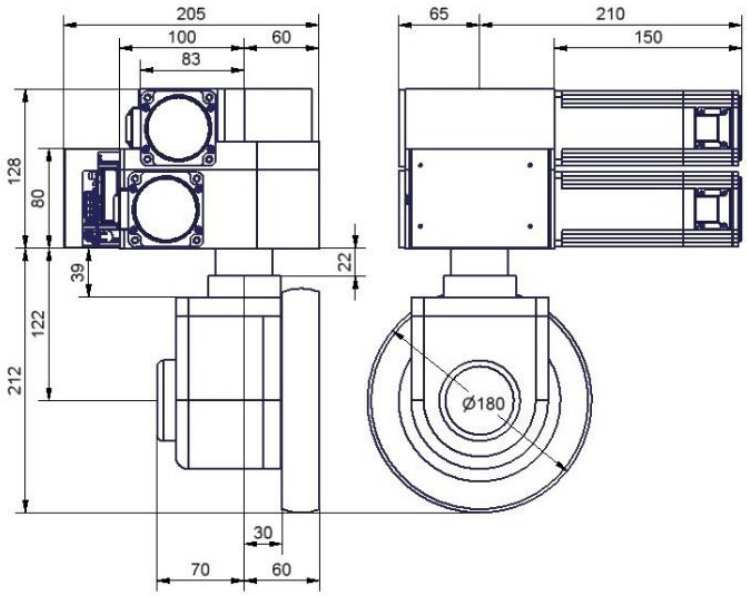


Figura 24: Representación de las ruedas *Omni-Drive* extraída de sus hojas de características [91].

Como aún no se dispone del modelo físico se empleará en *Gazebo* un modelo de robot cinemáticamente equivalente. En este caso el modelo empleado es el MPO-700 [92] de *Neobotix*, que emplea las mismas ruedas que llevará finalmente el modelo real.

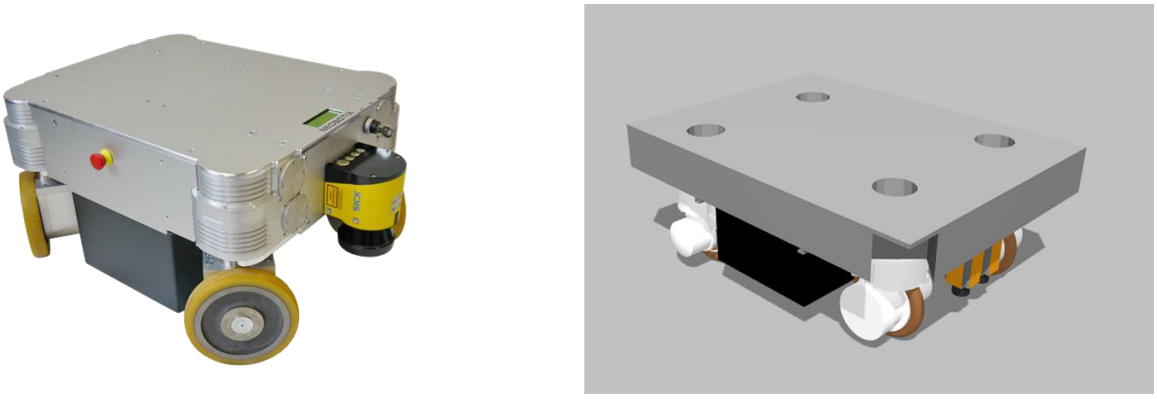

Figura 25: A la izquierda, imagen real del modelo MPO-700, extraída de [92]. A la derecha, imagen del modelo empleado en *Gazebo*.

En definitiva, tanto para este modelo como para el futuro modelo físico, se pueden enviar mensajes del tipo *geometry_msgs::Twist*[93] a través del entorno ROS. Estos mensajes se traducen en comandos a las ruedas orientables que actúan sobre sus ángulos variables,

φ(t) y β(t), y que permitirán al robot moverse en cualquier dirección. Esta comunicación se realiza mediante una modelo subscriptor-publicador, explicado en el Anexo 1.

Además de la base móvil también será necesario incluir los modelos de los sensores a través de los cuales el robot obtendrá la información de su entorno.

Para el sensor láser se tomará el modelo de láser *Hokuyo*[94] que incluye la instalación de *Gazebo*. Sobre el código de este se realizarán modificaciones para mejorar sus prestaciones de cara a nuestro proyecto. De esta manera, el modelo de láser empleado tendrá un rango máximo de 10 metros y un ángulo de apertura de 360º que se divide en 540 láseres, Figura 26. Las lecturas de este sensor se publicarán en un tópico del tipo *sensor_mgs::LaserScan* [95] y el *plugin* que implementa su funcionamiento en *Gazebo* es el *libgazebo_ros_laser*[96].

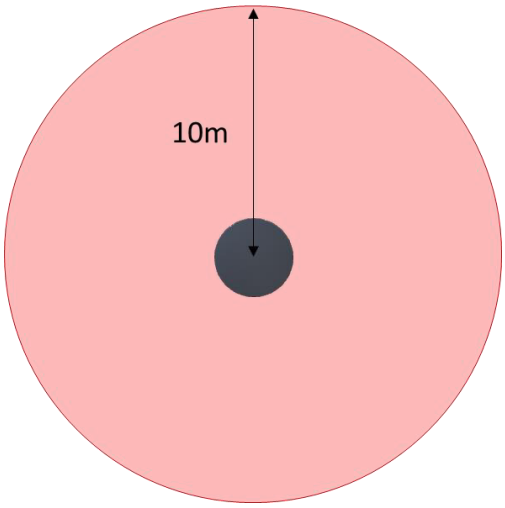


Figura 26: Representación de las características del láser 2D acoplado

Para el sensor 3D se toma el código del paquete ROS *velodyne_simulator* [97] que implementa modelos simulados de los modelos VLP-16 [98] y HDL-32E [99] de *Velodyne LiDAR*. De nuevo se realizan modificaciones de manera que se limita el rango del sensor a 10m y se le da un ángulo de apertura horizontal de 270º, conseguida por 540 rayos, y vertical de 30º, dividida en 60 láseres. En este caso, el tipo de mensajes publicándose por el sensor será *sensor_mgs::PointCloud2*[100] y el plugin que implementa su funcionamiento es el *ligbazebo_ros_velodyne_laser*, incluido en el propio paquete.

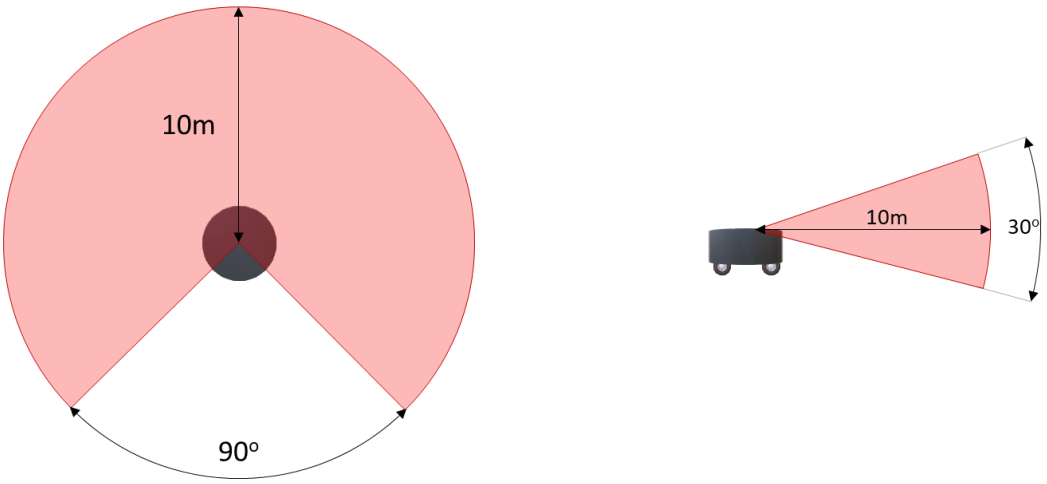


Figura 27: Representación de las características del láser 3D acoplado

El acople de los sensores no se realiza modificando directamente el código del modelo. En su lugar se ha creado una clase en C++, *coupled_robot*, que permite la adhesión de sensores u otras herramientas al robot. Para ello, se determina el la traslación y la rotación entre el sistema de coordenadas de la base del robot y el que se asocia al sensor. Esta información se utiliza para ir cambiando la posición del modelo en *Gazebo* y para actualizar la información de los sistemas de coordenadas, que se tratará en más profundidad en el siguiente apartado. La decisión de emplear estos programas permite introducir distintos sensores sin necesidad de alterar el modelo original del robot.

## 4.4. Sistemas de coordenadas

En cualquier trabajo de robótica medianamente complejo resulta fundamental el empleo de distintos sistemas de coordenadas. En ROS, el paquete más extendido para la gestión de estos ejes o *frames* y las operaciones que los involucran es *tf* [101]. Un artículo de Tully Foote, uno de los autores, presentando y desarrollando el paquete puede encontrarse en [102]. En el código propio también se ha empleado *tf2* [103], una versión mejorada de la librería original que además es computacionalmente más eficiente. A fecha de finalización de esta memoria ambos paquetes se encuentran mantenidos y son completamente compatibles.

El paquete permite gestionar la información de estos *frames* de manera rápida y eficiente, sin necesidad de concatenar operaciones introducidas en el código de manera manual por

cada creador. Su uso tan extendido es una de las claves de la modularidad de ROS, facilitando enormemente el uso combinado de distintas librerías y sensores.

Aunque aún no se ha presentado de manera completa la implementación y conexión entre los distintos paquetes, en la Figura 28 se puede ver el árbol de ejes de coordenadas presentes en el proyecto.

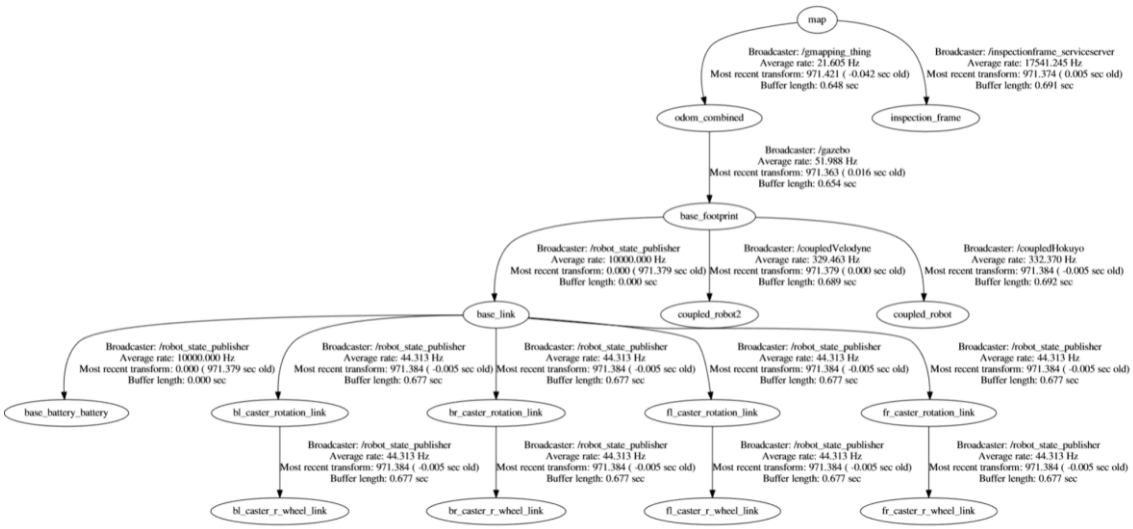


Figura 28: Árbol de sistemas de coordenadas empleadas en el trabajo

Se puede observar como el sistema de coordenadas *map* es del que dependen el resto. Este *frame* se genera, como todos, al inicio de la simulación cuando se encuentra en la misma posición y orientación que *base_footprint* y *odom_combined*. El sistema del mapa se genera a raíz de la estrategia de *SLAM* y de su relación con *base_footprint* podemos extraer la posición estimada del robot. Entre ellos aparece el *frame* que genera el cálculo de la odometría que realiza el propio modelo del robot.

También se puede ver como, inmediatamente un nivel por debajo de *map,* se encuentra el *inspection_frame*. Este sistema de coordenadas auxiliar se emplea en la inspección de las chapas. La lógica de la máquina de estados se encarga de situarlo en una esquina de la chapa, Figura 29, haciendo que los movimientos y posiciones dentro de la misma sean más intuitivos.

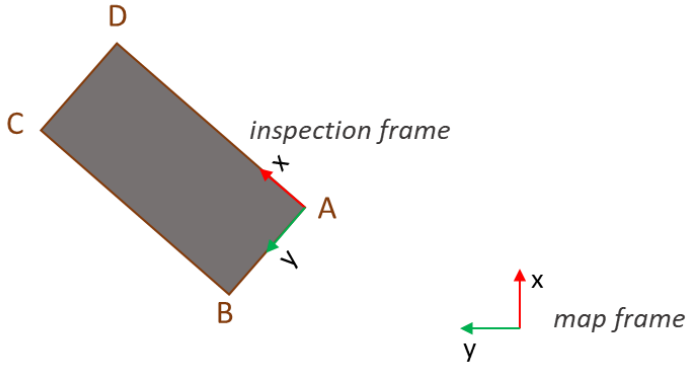


Figura 29: Representación de la posición del *inspection_frame* cuando se selecciona una zona de inspección.

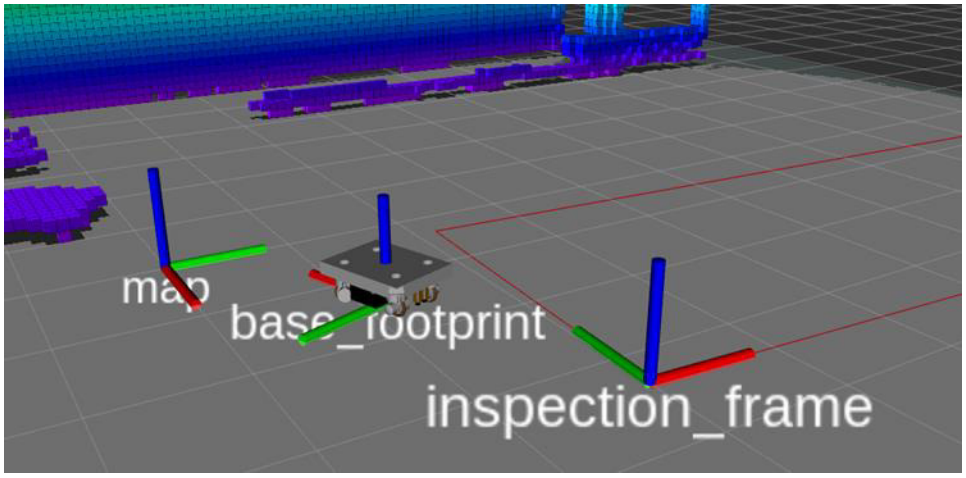


Figura 30: Ejemplo de aplicación del *inspection_frame* visualizado en *Rviz*

Bajo *base_footprint* aparecen todos los sistemas de coordenadas que dependen del robot. El de la base, las baterías y los relacionados con las cuatro ruedas están asociados directamente con el modelo del robot. Por su parte, *coupled_robot* y *coupled_robot2* son las coordenadas asociadas al sensor láser 2D y 3D, explicadas en la anterior sección.

## 4.5. Control del robot

Además de los algoritmos de navegación, resulta interesante poder enviar comandos simples de movimiento al robot. De esta forma se podrán realizar tareas como la inspección de las chapas de acero o la futura reparación de estas.

Para ello, se desarrolla una clase, *RobotController*, que permite enviar objetivos referidos a un *frame*. Una vez se solicita un objetivo se planifica una ruta simple que consiste en una línea recta, sin tener en cuenta la lectura de sensores ni ningún tipo de mapa. Para lidiar con los posibles errores derivados del envío de mensajes y del error acumulado, la trayectoria se simplifica aún más dividiéndose, de manera interna, en una serie de objetivos intermedios.

El método *move* de la clase puede utilizarse empleando cuatro modos diferentes que hacen referencia a la definición o no del ángulo final e inicial del robot. En la Figura 31 pueden verse los distintos comportamientos ejecutados tomando *map* como un sistema de coordenadas de referencia y *base_link* como el sistema del robot. Las posiciones $P_i$ representan la primera posición del robot, en azul, la final, en rojo, y las posiciones intermedias, en morado. Siguiendo el orden de la figura, el primer modo representado, *DEFAULT_ANGLES,* permitirá desplazar al robot hasta una posición inicial sin alterar la orientación en la que se encuentra cuando se envía la orden. El siguiente modo, *DEFINED_INITIAL_ANGLE,* permite darle un ángulo inicial al robot y que este se desplace sin alterarlo. En la imagen *c,* se puede ver el comportamiento *DEFINED_FINAL_ANGLE* en el que el robot comienza a desplazarse sin alterar su orientación. Una vez alcanza la posición final rota hasta alcanzar la pose deseada. Por último, *DEFINED_ANGLES* combina los dos anteriores modos. El robot rotará hasta una orientación inicial, se desplazará hasta la posición con este ángulo y, una vez alcanzada la posición final, girará hasta alcanzar la orientación definitiva.

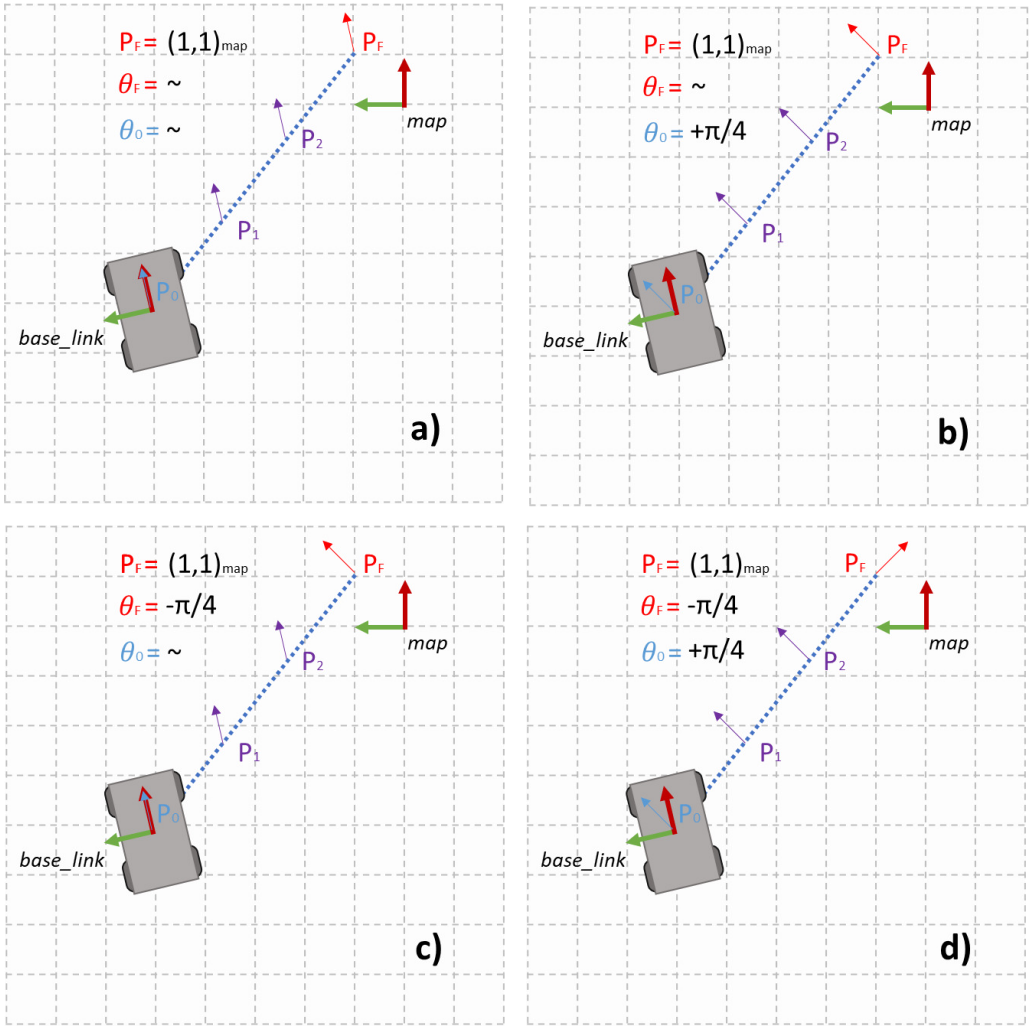


Figura 31: Distintos modos de funcionamiento del método *move* de la clase *RobotController* en función de la opción *ANGLE_MODE*. Las flechas azules indican la posición inicial del robot, las malvas las intermedias y las rojas las finales.

La clase también contempla la posibilidad de enviar una coordenada sobre el propio *frame* del robot de manera que se interpretará como una coordenada relativa. Se calculará la posición objetivo aplicada en el sistema de coordenadas del robot tal como se encuentra en el momento de envío de la orden.

Además de la función *move* también se implementan los métodos *turn, turn_rel* y *stop*. Los dos primeros permitirán al robot rotar sobre sí mismo hacia posiciones absolutas respecto a un eje y rotaciones relativas, respectivamente. La implementación *turn* incluye la posibilidad de indicar un signo de giro o que se escoja la dirección de rotación más corta hacia el objetivo.

Por último, cabe destacar que el desarrollo de la clase se ha realizado apoyándose en el paquete *tf2*[103, p. 2], explicado en la anterior sección, y usando varios de los mensajes del paquete *geometry_msgs*[104]. De esta forma, la clase no dependerá en ningún caso de elementos simulados y servirá también para enviar comandos de este tipo a la interfaz de un robot real.

## 4.6. SLAM con GMapping

Para la implementación del algoritmo de *SLAM* se utiliza el paquete *gmapping* [41]. Este software implementa una adaptación de la librería perteneciente a *OpenSLAM*[105] para hacerla compatible con el entorno ROS.

El programa recibe el sistema de coordenadas asociado a la base del robot y el sistema de coordenadas relacionado con la odometría del robot. Como resultado de aplicar el filtro de partículas se generará un nuevo sistema de coordenadas vinculado al mapa. La jerarquía de estos sistemas puede verse en la Figura 28.

También será necesario proporcionarle al nodo la información recogida en el sensor. Esto se realiza indicando el tópico que contiene los mensajes de tipo *sensor_msgs::LaserScan,* que en este trabajo vendrán proporcionadas por el sensor 2D simulado.

Respecto al mapa, la posición de la odometría indicará la posición inicial en la que ha aparecido el robot y el *frame* de la base será la estimación de la posición del robot. El sistema de coordenadas de la odometría es generado por el propio robot a partir de los comandos de velocidad que le han sido enviados. En ausencia de un mapa, su relación con la base sirve como estimación de la posición. En resumen, puede concluirse que, en consonancia con la teoría, el sistema de coordenadas del mapa supone una corrección sobre el *frame* puramente odométrico en la estimación de la pose del robot, Figura 32. Un análisis de la evolución de los errores de ambos *frames* podrá verse en la sección *Experimentos* y resultados.

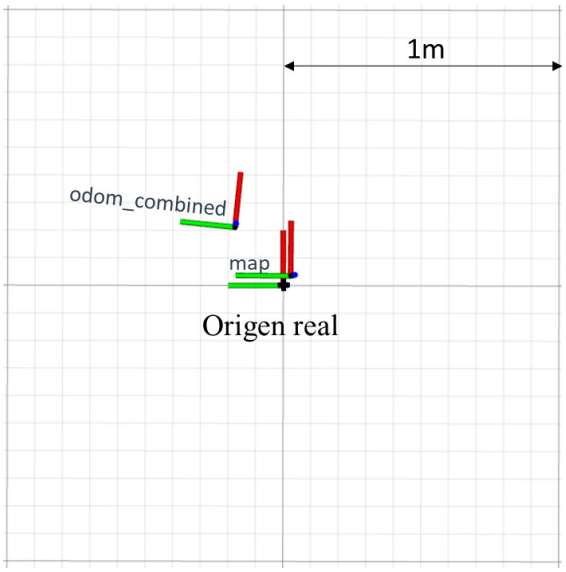


Figura 32: Captura del programa Rviz en la que se comprueba la mejora que supone el frame del SLAM sobre el de la odometría.

Además de ese sistema de coordenadas también se generan *topics* que incluyen el mapa generado que, aunque ya se ha explicado que en este caso no se emplea por no considerarse lo suficiente seguro, podría servir para otras aplicaciones.

## 4.7. Mapeo 3D con Octomap

Para realizar el mapa tridimensional del entorno se empleará el paquete *octomap_mapping* [48]. Se ejecuta el nodo *octomap_server_node* al que se le especificarán los sistemas de coordenadas tanto de la base del robot como del mapa sobre el que se pretende construir, así como el *topic*, del tipo *sensor_mgs::PointCloud2*, a partir del cual se construirá el modelo 3D. En nuestro caso, se indicará el generado por el sensor láser tridimensional que será la principal fuente de información para el paquete.

Como ya se ha explicado, el objetivo principal del mapeo 3D es añadir al mapa todos aquellos obstáculos que pueden producir una colisión en el robot. Para ello puede establecerse una cota de altura máxima para los puntos detectados, mediante el parámetro *pointcloud_max_z,* y tomar el mapa que proyecta todos los puntos del modelo en el plano, publicado en el *topic projected_map* en un mensaje de tipo *nav_msgs::OccupancyGrid* [106], Figura 33. De esta manera, se puede tomar para la primera variable un valor ligeramente superior a la altura del robot garantizándose así que se incluyen todos los

potenciales obstáculos. En el proyecto se ha decido otorgarle a la cota máxima un valor conservador de 2 metros.

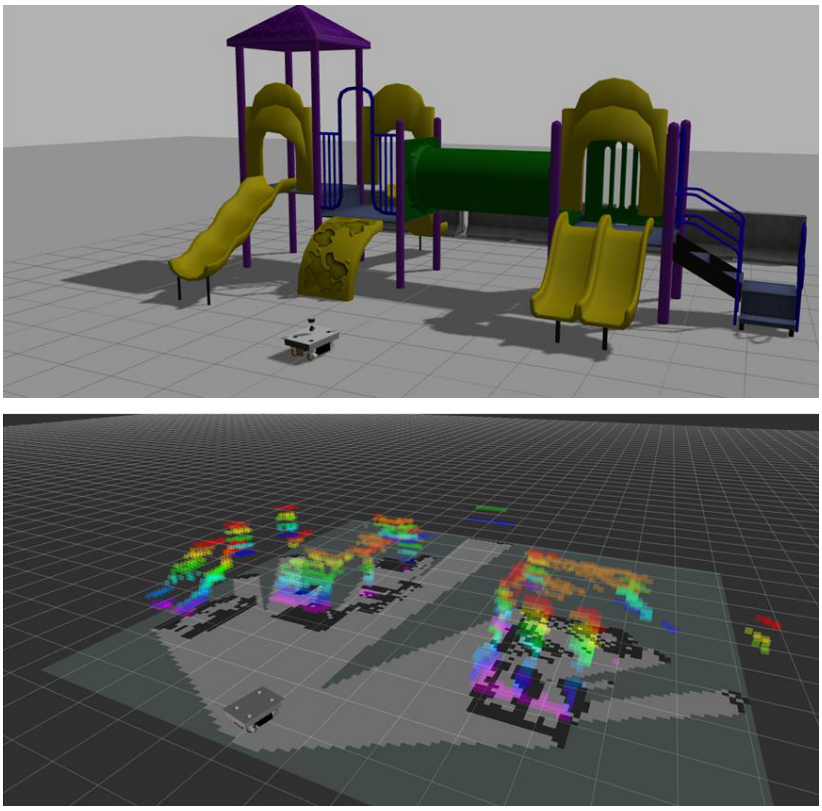

Figura 33: Los puntos 3D no son tenidos en cuenta a partir de una cota mientras que los que sí son proyectados generan un mapa 2D seguro para un robot de una altura inferior a la cota impuesta.

Como su propio nombre indica, el parámetro *resolution* permite el tamaño mínimo de cada nodo del mapa. En este trabajo, este valor se ha ido modificando en función del entorno escogido y del tipo de obstáculos presentes. Por norma general, se ha escogido una resolución de 10 centímetros cúbicos, rebajándose el valor hasta 5 en casos de mayor precisión.

Como se comentó en el marco teórico, la librería incluye la posibilidad de filtrar el suelo del mapa para que no se tenga en cuenta como un obstáculo, Figura 34. Para ello, el paquete emplea las funciones de segmentación de la librería *Point Cloud Library (PCL)* [107], de extendido uso en el manejo de nubes de puntos. Es posible configurar mediante variables la máxima altura a la que puede encontrarse el plano, así como su máxima desviación respecto a la orientación del plano XY. También es posible determinar la

distancia respecto al plano detectado que define qué puntos serán filtrados. Estos parámetros se dejan por defecto solo activándose el booleano *filter_ground,* que activa el depurado.

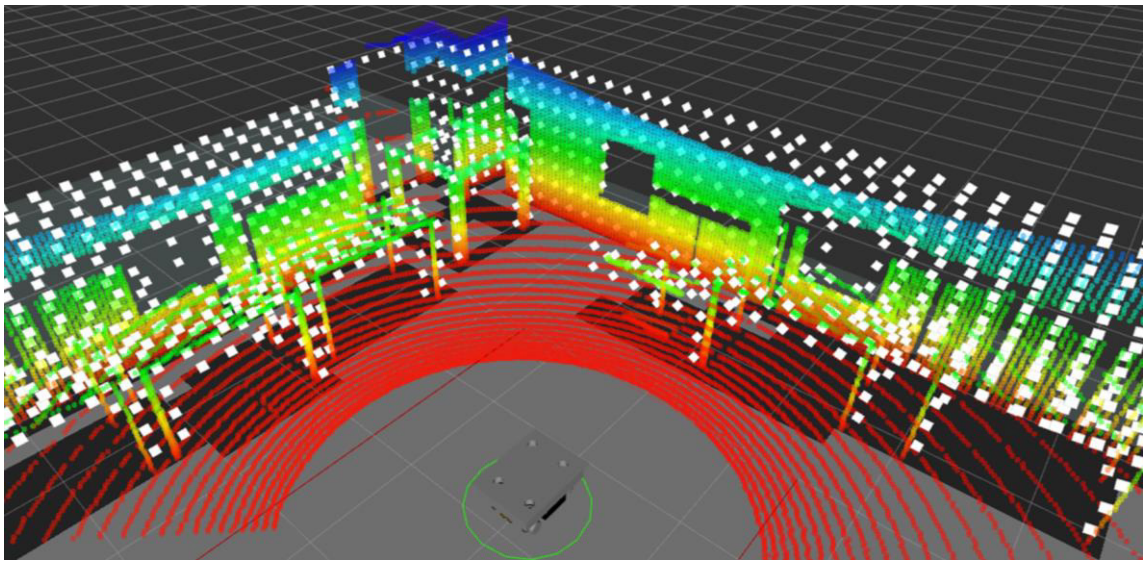

Figura 34: Imagen que muestra en escala de color los puntos que detecta el sensor y en blanco los nodos que incluyen en el *octomap* una vez se ha filtrado el suelo.

La librería también incluye la opción de modificar los valores del modelo del sensor que afectan al ámbito probabilístico del problema, detallado en la sección 2.2. Se han mantenido los valores por defecto, expuestos en [48], pues ofrecían un rendimiento óptimo.

En resumen, combinando todos estos ajustes, se consigue un mapeo del entorno satisfactorio, como puede verse en la Figura 35.

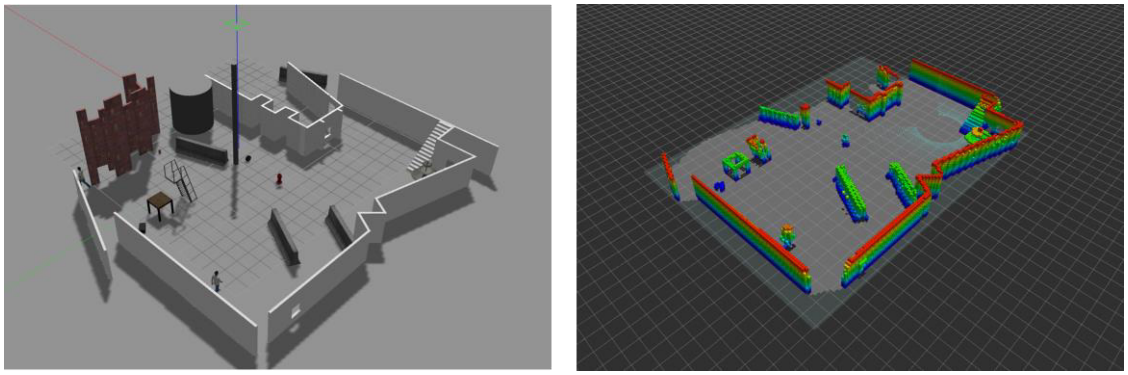

Figura 35: A la izquierda imagen de un entorno en *Gazebo*. A la derecha modelo 3D creado tras una exploración, incluyendo una cota máxima de altura.

## 4.8. Algoritmos de navegación

Para la navegación se utiliza la librería de ROS *navigation* [108]. Se trata del paquete más extendido en el control y la navegación de robots móviles. Se encargará de coordinar la información recibida del mapa y el planificador de trayectorias para enviar comandos a la base móvil. La explicación se puede dividir en las características de los mapas de coste y la configuración del planificador en sí.

### 4.8.1. Mapas de coste

El principal paquete encargado de enviar comandos y planificar las trayectorias es *move_base*[109]. Este paquete hace uso de dos mapas de coste, el global sobre el que se genera toda la trayectoria hasta el objetivo y el local, de tamaño más reducido y pensado para reaccionar a cambios sobre el mapa. Los mapas de coste son estructuras proporcionadas por la librería *costmap_2d* [110] que permiten trabajar con las zonas ocupadas, libres o desconocidas.

Los mapas de coste de este paquete pueden organizarse en distintas capas siendo las más comunes la capa del mapa estático, la capa de obstáculos y la capa de inflación. Una solución habitual es emplear un mapa estático para el mapa global y una capa de obstáculos construida de manera dinámica a partir de los sensores para el mapa local.

Gracias a *Octomap* el mapa recibido en nuestro trabajo ya tendrá un comportamiento dinámico y que garantiza al robot un entorno seguro para la navegación. De esta manera, el mapa local y global estará compuesta por los mismos tipos de capas: una capa *staticmap*[111] y, por encima, una capa *inflation*[112].

La capa estática será simplemente la lectura del mapa 3D proyectado configurada para atender a actualizaciones de este.

La capa de inflación, por su parte, permite aumentar la seguridad de la navegación dándole al mapa valores de peligro alrededor de los obstáculos detectados. En la implementación se establece un radio de inflación de 0.65 metros. El coste de este se calculará en base a la ecuación 32.

$$coste = e^{-(k \times (d-r)) \times (254-1)} \tag{32}$$

Donde $k$ es un factor de escala, que tomará el valor 10.0, $d$, es la distancia del punto al obstáculo, y $r$, es el radio de inflación de los obstáculos.

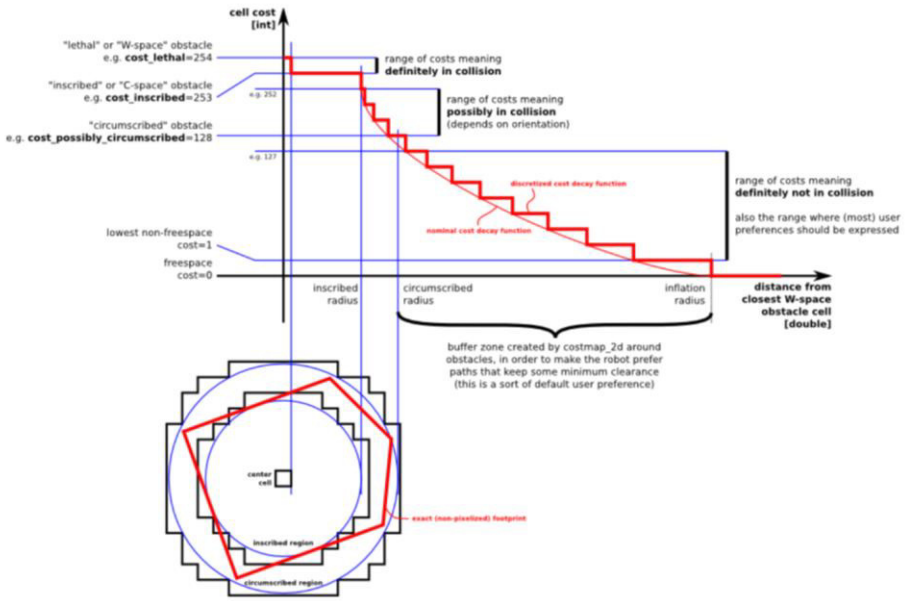


Figura 36: Gráfico explicativo del cálculo de los mapas de coste. Figura extraída de [110].

La principal diferencia entre el mapa local y el global estará en su tamaño. El mapa global recibe un tamaño máximo de 100x100 metros, en este caso, y se irá completando según se va construyendo el mapa 3D. Servirá para planificar la trayectoria inicial completa. El mapa local, sin embargo, será una ventana de 6x6 metros sobre el mapa que se irá desplazando junto al robot, Figura 37. Sobre este mapa se irán introduciendo las distintas poses intermedias de manera que respeten la trayectoria global lo máximo posible evitando la colisión con obstáculos.

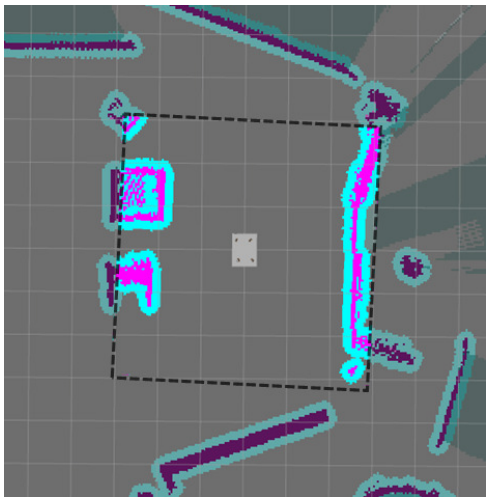

Figura 37: Representación del mapa de coste local. El mapa de coste global se extiende más y, en la imagen, puede verse con una menor opacidad.

### 4.8.2. Planificador *TEB*

Por último, una vez se ha establecido el funcionamiento del mapa será necesario configurar el planificador de trayectorias. El algoritmo *TEB* se encuentra implementado en ROS desarrollado por sus propios creadores bajo el nombre de *teb_local_planner* [113]. Para la ejecución del planificador este debe configurarse en un archivo *.yaml* y cargarse al paquete de navegación.

Los parámetros usados en el trabajo se extraen de un repositorio oficial de *Neobotix* donde se encuentran los parámetros para el MPO-700[114]. A continuación, se explicarán los parámetros más importantes.

Como se ha explicado en el apartado teórico, las limitaciones cinéticas y dinámicas del robot suponen un aspecto clave a la hora de generar la ruta más eficiente. En consonancia, se permite establecer los valores máximos de la velocidad y aceleración tanto lineales como angulares.

También es posible modificar la tolerancia a la hora de alcanzar el objetivo final y la forma externa del robot. En este último aspecto se realiza una pequeña modificación por la que se define la huella del robot como una circunferencia de 1 metro de diámetro, como podrá apreciarse en la Figura 38. De esta manera, los cálculos de colisión sobre el mapa se realizarán con una geometría que será muy similar a la presente en el robot físico.

Un grupo de parámetros hacen referencia directa a la construcción de la trayectoria. Entre los más importantes se encuentran la resolución temporal de la trayectoria, *dt_ref.* Con esta variable podrá especificarse el tiempo máximo que puede separar dos poses intermedias consecutivas que forman parte de la trayectoria generada. Puede establecerse también un número de muestras mínimo, *min_samples,* para situaciones en las que la ruta es muy corta. La distancia mínima a la que puede situarse un objetivo intermedio, *min_obstacle_dist,* también es un parámetro fundamental en la trayectoria generada.

Es posible visualizar varios aspectos de la trayectoria a seguir gracias a los *topics* que publican la trayectoria local, las poses objetivo intermedias y la trayectoria global, generada por defecto según el algoritmo de Dijsktra[115]. Estos elementos aparecen de manera respectiva en *local_plan*, *teb_poses* y *global_plan* todas dentro de *move_base/TebLocalPlannerROS*. Una visualización puede observarse en la Figura 38.

En verde puede observarse la ruta global, en azul la local, y en rojos las poses intermedias que configuran la ruta.

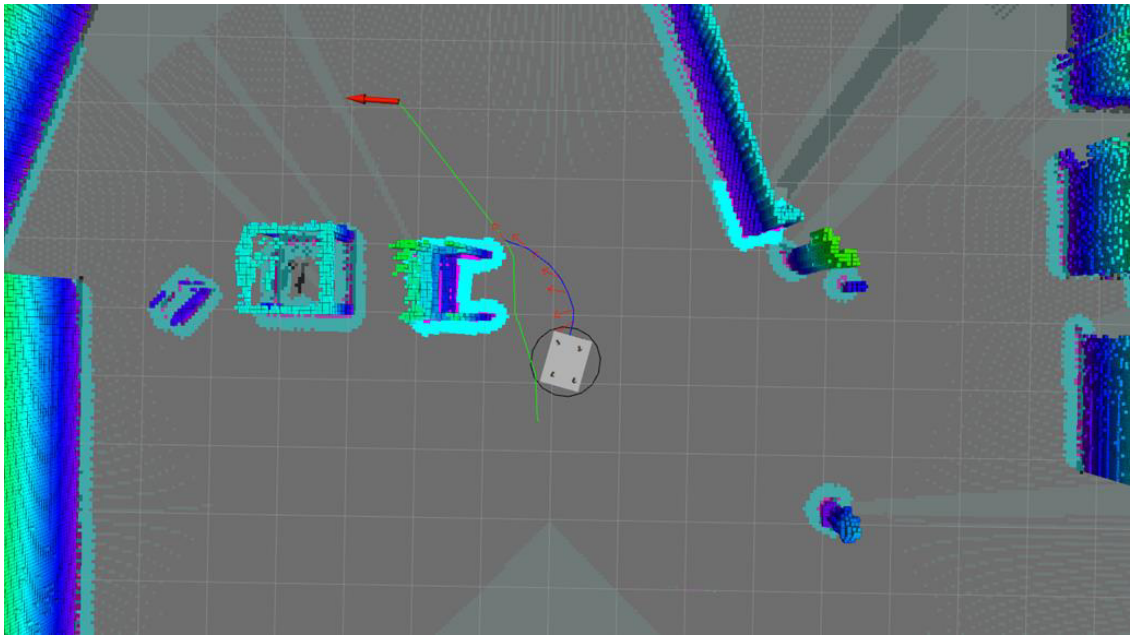

Figura 38: Visualización de los principales elementos de la navegación.

El resto de los parámetros hacen referencia a variables que afectan a la optimización del grafo y relacionadas con el peso que se le da a cada uno de los factores. De nuevo se mantienen los valores especificados en la configuración oficial del modelo.

### 4.8.3. Envío de objetivos

El paquete de navegación recibe las órdenes a través del *topic move_base_simple/goal*, del tipo *geometry_msgs::PoseStamped*[116]. Además de una publicación de ese tipo de mensaje a través de un programa estándar, también es posible enviar esta orden a través de la herramienta "*2D Nav Goal"* incluida en *Rviz*[117].

El programa incluye por defecto una herramienta de interacción que recibe la posición clicada en el entorno, genera el mensaje y lo envía al *topic* del que se lee la orden. Para este trabajo se desarrollará una herramienta análoga que sigue los mismos pasos, pero, en este caso, el mensaje se envía a un *topic* intermedio, *my_2d_goal,* en lugar de enviarse directamente al relacionado con *move_base*. De esta manera, se añade un nivel de abstracción a la herramienta y se podrá, antes de enviar la orden, analizar la posición seleccionada y decidir en función de la lógica implementada.

## 4.9. Evaluación de la posición objetivo-fijada

En la aplicación será necesario saber si el usuario ha seleccionado una posición con una chapa a inspeccionar o solo una posición hacia la que el robot debe navegar. Para ello se calcula el área del rectángulo a partir de las coordenadas de sus esquinas y también con como una suma de triángulos entre las esquinas y el punto seleccionado.

$$Área_{ABCD} = \frac{1}{2}|(y_A - y_c) \cdot (x_D - x_B) + (y_B - y_D) \cdot (x_A - x_c)| \tag{33}$$

$$Área_{ABC} = \frac{1}{2}|x_A \cdot (y_B - y_C) + x_B \cdot (y_C - y_A) + x_C \cdot (y_A - y_B)| \tag{34}$$

El punto se encontrará dentro del rectángulo si, y solo si, el área del rectángulo es igual a la suma de los triángulos formados por el punto y los vértices del rectángulo.

$$Área_{ABCD} == Área_{ABP} + Área_{BCP} + Área_{CDP} + Área_{ADP} \tag{35}$$

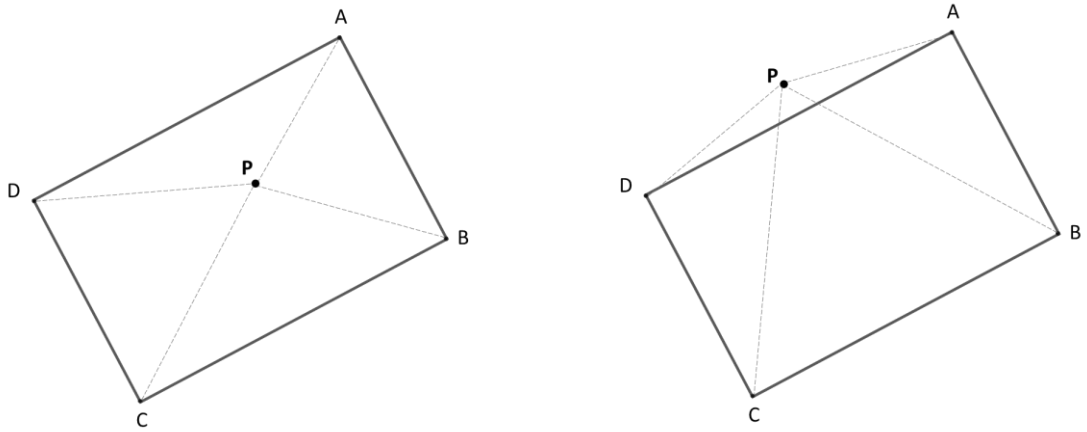


Figura 39: Representación de los distintos triángulos involucrados en la evaluación de la posición fijada.

## 4.10. Inspección de la chapa

Para realizar los movimientos de inspección de la chapa se aprovecharán las funcionalidades implementadas en la clase *RobotController*. En el caso de la inspección simple, Figura 40, el cálculo de las posiciones se integrará en el diseño del propio autómata.

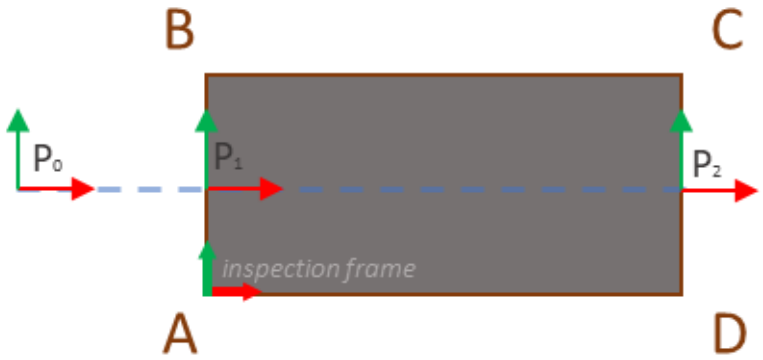


Figura 40: Representación de las tres poses intermedias que se alcanzan para ejecutar una inspección simple de la chapa.

La inspección compleja requiere, sin embargo, de un paso intermedio con el que se genere la trayectoria a seguir. Para ello se desarrolla la clase *SimpleCPP* que toma como parámetros, además del ancho y el largo de la chapa, el ancho cubierto por el haz láser y la distancia entre el láser y el centro del robot, Figura 41. Adicionalmente, en el programa se podrá especificar la distancia máxima que debe separar dos posiciones objetivo, con el fin de facilitar la labor del robot de mantener una trayectoria recta.

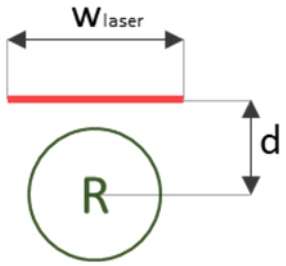


Figura 41: Parámetros relativos a la configuración del robot influyentes en la generación de una trayectoria

Una vez generada las posiciones intermedias que componen la trayectoria de inspección bastará con pasárselas al método *move* de *RobotController* para realizar el recorrido.

Antes de proseguir con las siguientes explicaciones cabe destacar que en las simulaciones realizadas las láminas de acero no se han simulado como un modelo físico. En su lugar, se han representado como zonas rectangulares del suelo que se dibujan en el visualizador *Rviz*. El simulador *Gazebo* no es el más indicado para lidiar con las colisiones entre el

robot y la chapa por lo que la problemática que supondrá el acceso del robot a la chapa deberá afrontarse con pruebas con un prototipo real.

## 4.11. Coordinación de comportamientos con FlexBE

### 4.11.1. Uso de la herramienta FlexBE

La herramienta FlexBE[118], [119] ayuda a desarrollar máquinas de estados totalmente integrada en ROS. Además de proporcionar muchos estados básicos ya implementados (imprimir mensajes por pantalla, leer un mensaje de un tópico, esperar un cierto tiempo etc.) proporciona también una interfaz de usuario que facilita la creación y monitorización del autómata.

La máquina de estados se compone de estados al uso y contenedores o *containers*. Estos últimos pueden emplearse como un superestado común, que sirve para añadir un nivel adicional de abstracción. Resulta posible la implementación de contenedores paralelos en los que cada uno de los estados dentro de ellos se ejecutan de manera simultánea y es posible no salir de él hasta que todos los estados se han finalizado. También es posible crear también superestados prioritarios que, cuando se activan, detienen la ejecución del resto de estados presentes en la máquina. En esta implementación solo ha hecho uso de estados y superestados comunes.

Los estados son, en resumidas cuentas, clases escritas en Python que heredan de la clase *EventState* y que están compuestas fundamentalmente por 6 métodos, que componen su ciclo de vida, Figura 42.

En el constructor, *_init_*, se especifican las salidas del estado, así como las variables de entrada, las variables de salida y los parámetros que se le introducen. Finalizado el proceso de construcción se procede a la ejecución del método *on_start*, que puede ser útil si se pretende realizar alguna acción en el comienzo de la máquina de estados, independientemente de si este se ha activado. En sentido opuesto, *on_stop* permitirá que cada estado lleve a cabo un proceso cuando la máquina de estados se detiene.

Finalizados los métodos iniciales el estado entra en espera hasta que es activado tras producirse una salida en otro estado. En ese momento se lleva a cabo una ejecución del método *on_enter* antes de pasar a la ejecución cíclica de la función *execute*. El bucle continúa hasta que una de las salidas, anteriormente definidas, es activada y se procede a

la ejecución de *on_exit* antes de pasar a la activación de un nuevo estado. Será en estos tres métodos en las que se programe la mayoría de las estrategias necesarias

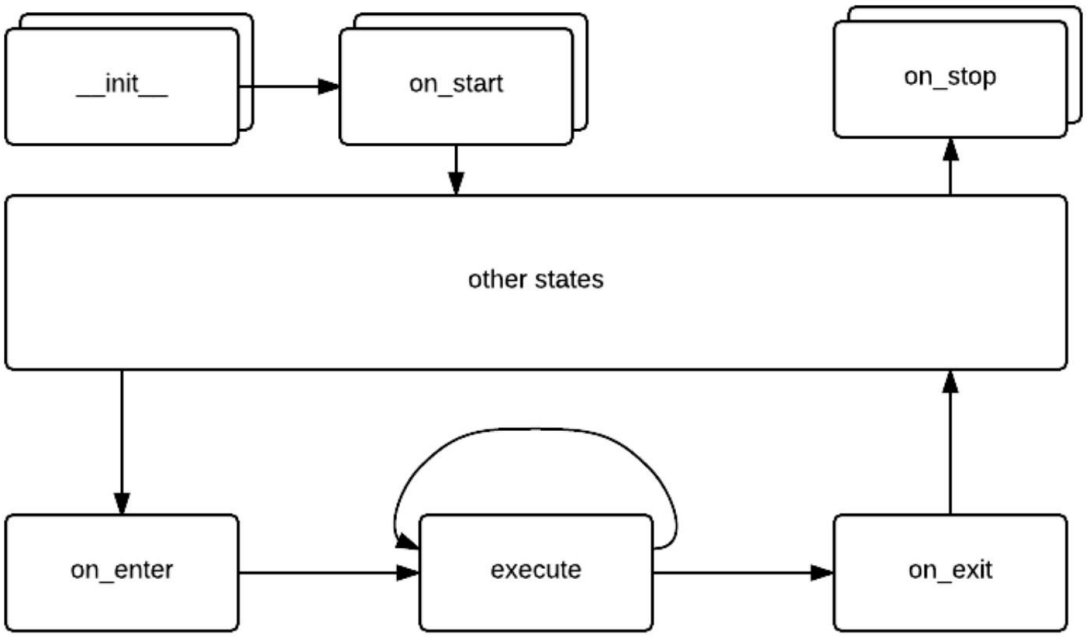


Figura 42: Representación gráfica del ciclo de vida de un estado *FlexBE*. Figura extraída de [120]

Es posible establecer, entre 4 niveles, la autonomía de cada una de las salidas de cada estado. De manera complementaria, se puede definir la cota de autonomía de nuestra máquina. Una vez establecida, todas las transiciones cuyo nivel sea inferior a ella requerirán una confirmación manual por parte del usuario para llevarse a cabo. Independientemente, cualquier transición puede forzarse en cualquier momento de la ejecución. Como se verá más adelante, estas características permiten en cierta medida establecer una interfaz de usuario.

### 4.11.2. Acciones y servidores

El código de FlexBE se encuentra escrito en lenguaje Python en contraposición con el C++ de los paquetes de desarrollo propio. Para poder aprovechar ese código se hace uso de la comunicación mediante acciones, explicada en el Anexo I. De esta manera, los estados de la máquina estarán escritos en Python y contendrán los clientes que realizan las llamadas a las acciones. El código C++ implementará los servidores que corren de

manera constante y que ejecutará la función asociada a la acción cuando el cliente lo solicite.

Los servidores que ejecutan acciones implementados específicamente para el proyecto son los que se ven en la Tabla 1. Los movimientos del robot se realizan con los métodos de la clase *RobotController.*

| Servidor - .action | Descripción |
|---|---|
| baserotation_server.cpp – BaseRotation.action | Ejecución de una rotación de la base un número especificado de grados. |
| moverobotcoord_server.cpp – MoveRobotCoord.action | Movimiento recto del robot hacia una posición referida en un *frame* especificado. |
| completerectanglecoverage_server.cpp – CompleteRectangleCoverage.action | Planificación y ejecución de la trayectoria compleja de inspección basada en los parámetros de la chapa y del robot. |

Tabla 1: Conjunto de acciones empleadas en el autómata

Otras operaciones se realizan de manera mucho más rápida por lo que se implementarán mediante servicios en lugar de mediante acciones, Tabla 2. Es el caso de la publicación de un sistema de coordenadas auxiliar y la consulta sobre la posición de las chapas.

| Servidor - .srv | Descripción |
|---|---|
| inspectionframe_server.cpp – SetFrame.srv | Publicación de un *frame* de *tf* a una distancia y orientación de un sistema de referencia. El servicio se implementa como el método de una clase por lo que se le dota de memoria respecto a unas cuantas variables. |
| chapastorage_server.cpp – ChapaStorage.srv | En función de la solicitud devuelve el número de chapas almacenadas o las dimensiones y las coordenadas de las esquinas de una lámina seleccionada. |

Tabla 2: Conjunto de servicios empleadas en el autómata

### 4.11.3. Implementación de la máquina de estados

Explicados ya los conceptos se pasará a detallar la máquina implementada. En primer lugar, es necesario comentar que los servidores de las acciones y servicios deberán estar ejecutándose en paralelo cuando se inicie la máquina de estados.

Además de estos programas de desarrollo propio también deberán estar en ejecución los nodos de los paquetes encargados del SLAM, el mapeo 3D y el de navegación, además de, obviamente, los que proporcionan la simulación del robot y los sensores.

Finalmente, antes de la ejecución de la máquina pueden ajustarse las variables que configuran su comportamiento, Tabla 3.

| Nombre | | Valor | | Descripción |
|---|---|---|---|---|
| topic_rviz_goal | = | "/my_2d_goal" | <cte.> | *Topic* del que se recibe posiciones objetivo. |
| inspection_frame | = | "inspection_frame" | <cte.> | Sistema de coordenadas auxiliar para la inspección. |
| grados | = | 180.0 | <cte.> | Número de grados para la primera exploración. |
| dist_laserobot | = | 1 | <cte.> | Distancia entre la base del robot y haz de inspección. |
| width_laser | = | 2 | <cte.> | Amplitud del haz de inspección. |
| step_path | = | 4 | <cte.> | Distancia máxima entre dos posiciones intermedias de un recorrido de chapa. |
| vel_path | = | 0.5 | <cte.> | Velocidad para el recorrido encima de la chapa. |
| rviz_goal_posestamped | = | [ ] | <vble.> | Variable auxiliar que guarda un mensaje de tipo *PoseStamped.* |
| rviz_goal_pose2d | = | [ ] | <vble.> | Variable auxiliar que guarda un mensaje de tipo *Pose2D.* |

Tabla 3: Constantes y variables de la máquina en FlexBE

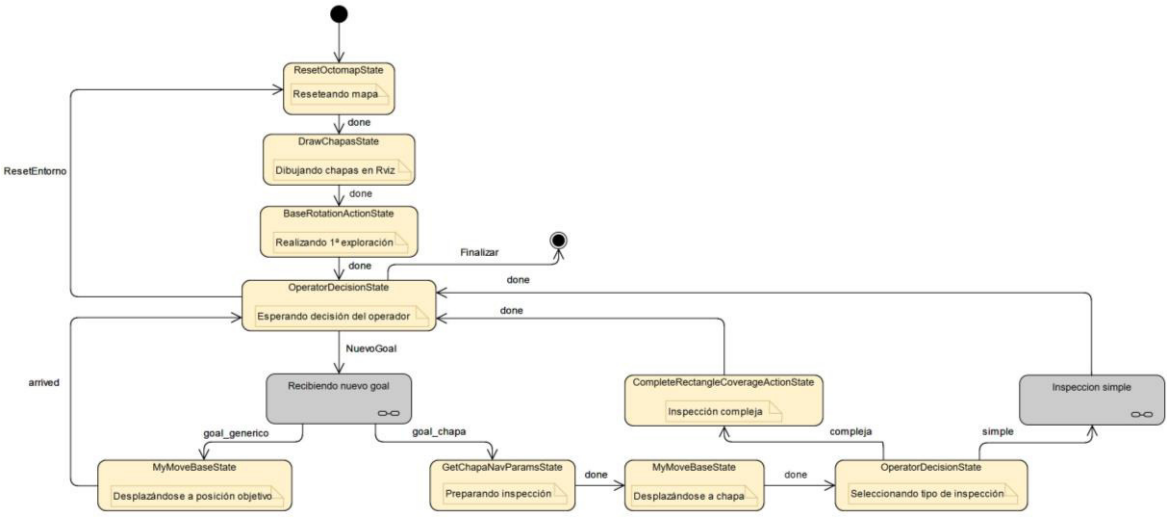


Figura 43: Vista general de la máquina de estados.

Centrándonos en la lógica implementada en el autómata, Figura 43, el primer estado en activarse será el de la re-inicialización del *octomap.* Para el borrado de todos los nodos que componen la estructura de *octrees* se invoca el servicio *reset* del nodo *octomap_server.*

El siguiente estado se empleará para representar la posición de las chapas en *Rviz*. Para ello, se accede al servidor que guarda las coordenadas de las chapas y se publica un

mensaje del tipo *geometry_msgs::PolygonStamped* [121] creando para cada una el *topic chapa_idX,* en función del identificador de la chapa dibujada.

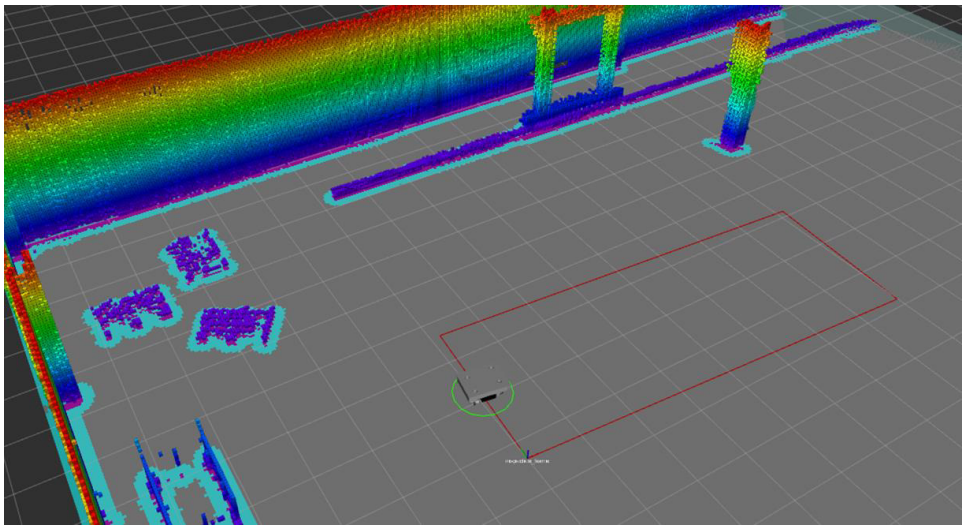

Figura 44: Imagen con robot en chapa dibujada

En la siguiente etapa se realizará una rotación de la base del robot sobre sí misma. De esta manera antes de inicializar el comportamiento el robot habrá obtenido un mapa completo de su entorno más cercano. Para realizar la rotación se hace una llamada a la acción del servidor *baserotation_server* en la que el número de grados viene definido por una variable configurada en la creación del autómata. La acción hace uso del método *turn_rel* de la clase *RobotController*, ya explicado en la sección 4.5.

Realizadas estas etapas iniciales se llega al *OperatorDecisionState,* una estancia desarrollada por *FlexBE* que permite al usuario seleccionar el siguiente paso. Se establecen tres posibles salidas: *Finalizar, ResetEntorno* y *NuevoGoal*. La primera permitirá finalizar el autómata y la segunda volver a los estados iniciales, por si se ha producido algún error en el mapa. Por último, *NuevoGoal* permitirá acceder a los estados relativos al envío de posiciones objetivo.

Esta última salida lleva al superestado *Recibiendo nuevo goal* compuesto internamente por tres estados que se ejecutan de manera secuencial, Figura 45. El primero de ellos es un *SubscriberState* de la propia librería *FlexBE* que espera a la publicación de un mensaje en un determinado *topic.* El estado se configura para leer el *topic* que publica la herramienta de selección de posiciones en el visualizador y para tener una variable de salida con la

posición clicada. El siguiente estado, *PoseStamped_Pose2D_ConversionState*, sirve para cambiar el tipo de mensaje de *PoseStamped* a *Pose2D*. Esto permite continuar al estado que evalúa si la posición seleccionada se encuentra dentro o fuera de una chapa. La comprobación se realiza cómo se indicó en la sección 4.9 y accediendo a las coordenadas almacenadas en el servicio de *chapastorage_server.* El estado tiene dos salidas que comparte con el superestado: *goal_generico*, en el caso de que la posición no esté dentro de una lámina de acero y *goal_chapa* en el caso de que sí.

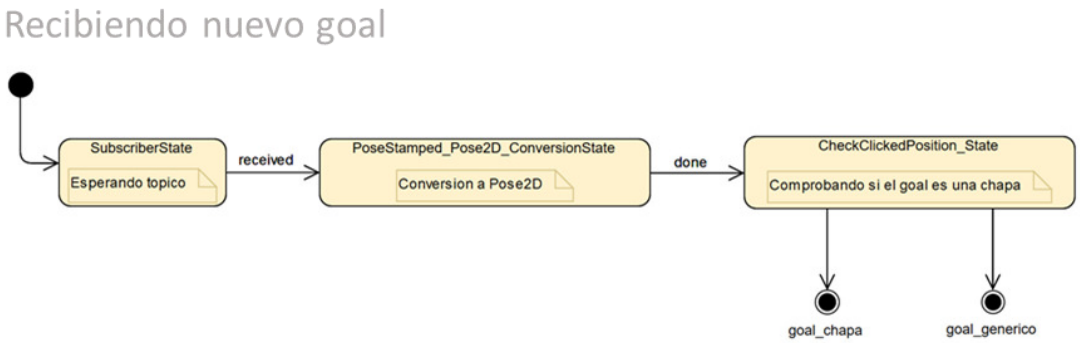


Figura 45: Interior del superestado “Recibiendo nuevo goal”

En el caso de haberse seleccionado una posición cualquiera, el autómata avanzará hacia el estado *Desplazándose a posición objetivo.* Este está implementado por *MyMoveBaseState*, que supone una pequeña variación respecto a *MoveBaseState* ya incluido en la librería. La modificación principal hace referencia al *frame* de referencia que viene por defecto. El autómata se mantendrá en este estado hasta que el robot alcance la posición deseada y se active la salida *arrived* volviendo al estado controlado por la decisión del operador.

Si la pose clicada se encuentra dentro de una chapa se accede al estado que prepara la inspección. Esta toma como entrada la *id* de la chapa que proporciona el anterior superestado. Con esa información *GetChapaNavParamsState* puede obtener del servicio las coordenadas y dimensiones de las hojas de acero. Con ello, puede variar la posición del *inspection_frame* a partir del servicio *SetFrame* que se ejecuta como un método de la clase *FrameService*. El estado también es capaz de seleccionar la pose intermedia entre las esquinas *A* y *B* de la chapa. Esta pose se envía a un nuevo estado que emplea el paquete *move_base* para que se lleve a cabo el desplazamiento.

Una vez el robot está correctamente situado en una posición inicial para la inspección se llega a un nuevo estado en el que las posibles salidas son cancelar la inspección, una inspección compleja y una inspección simple de la chapa. La opción de la cancelación hace que se vuelva, una vez más, al estado *"Esperando decisión del operador"* en el caso de que no se desee realizar ninguna inspección.

La inspección compleja se implementa en un estado que recibe las dimensiones de la chapa a inspeccionar. Con esos parámetros, junto a los que componen la configuración del robot, detallados en la sección 4.10 se puede llamar a la acción *CompleteRectangleCoverage*. Esta acción incluye la planificación de las distintas posiciones a alcanzar y el envío de comandos al robot para que los realice. Una vez finalizada la inspección se retorna al estado en el que se decide la siguiente operación.

En el caso de que se seleccione una inspección simple se accederá al contenedor *Inspección simple*, compuesto por cuatro estados secuenciales, Figura 46. El primero de ellos permite obtener las tres poses necesarias para esta inspección: una pose a cierta distancia de la chapa para cubrirla al completo, una vuelta a una posición entre las coordenadas *A* y *B* y, por último, una posición entre las coordenadas *C* y *D* que finalizará la inspección como se representa en la Figura 40. Los siguientes estados se encargan de enviar comandos simples al robot mediante la acción *MoveRobotCoord*. Para ello la acción recibe como objetivos el vector que se ejecuta por una función *move* de la clase *RobotController*. Cuando esta inspección finaliza se retorna al estado de decisión del operador.

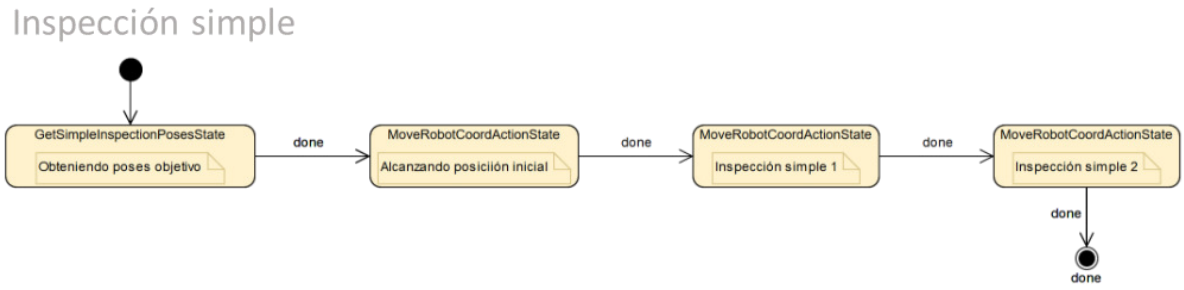


Figura 46: Interior del superestado "Inspección simple"

Finalizada la explicación cabe comentar que la inmensa mayoría de los estados cuentan con una salida *failed* que desembocan en el final homónimo del autómata y que se activa

si algún elemento en el estado desencadena en algún tipo de error. Se ha obviado en la explicación general para mayor claridad.

Por último, en las Figura 47, Figura 48, y Figura 49 se pueden ver un diagrama del envío de datos a través de la máquina de estados.

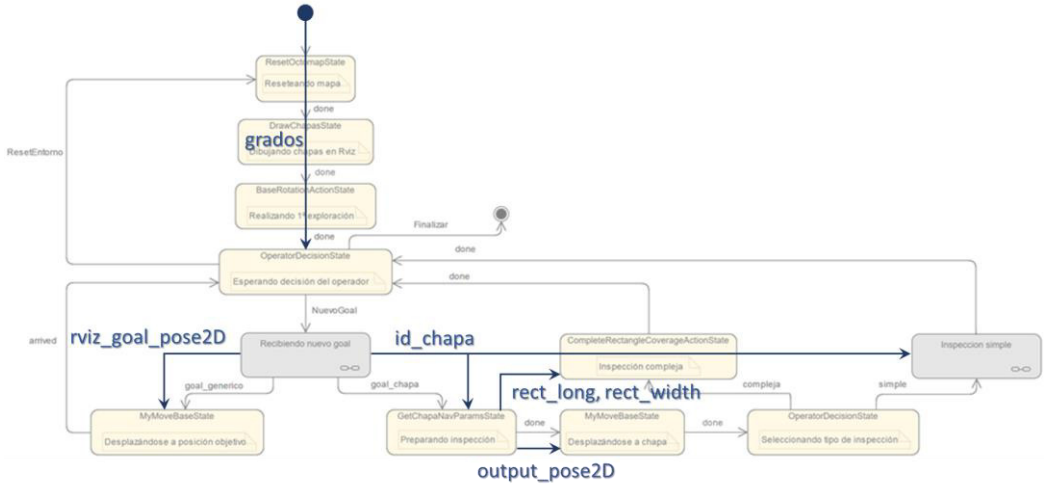


Figura 47: Flujo de datos general en el autómata

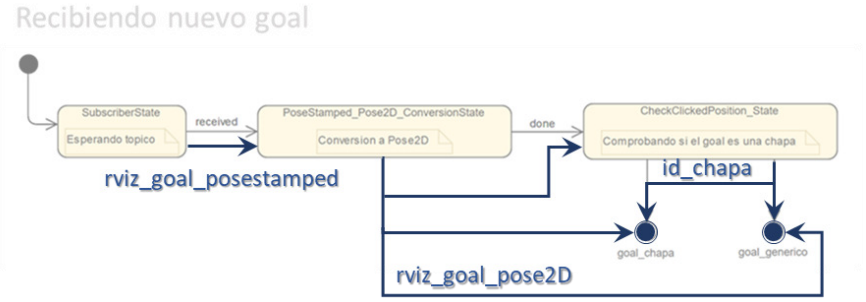


Figura 48: Flujo de datos dentro del superestado “Recibiendo nuevo goal”

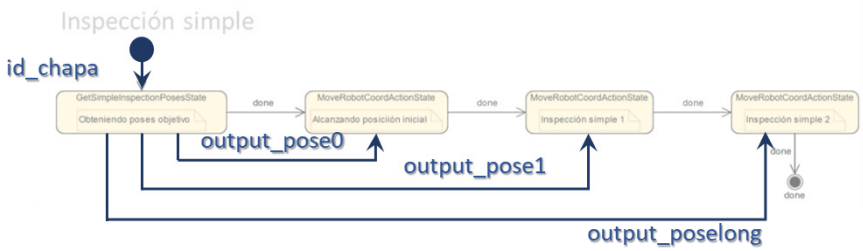


Figura 49: Flujo de datos dentro del superestado “Inspección simple”

## 4.12. Estructura en ROS

Como conclusión del capítulo y a modo de resumen general se incluye la Figura 50. En ella pueden observarse los principales nodos y *topics* involucrados en el proyecto, excluyendo por claridad de la explicación alguno de estos elementos.

En primer lugar, se pueden apreciar en la parte derecha los nodos que comunican a *tf* los eslabones del robot. En la figura también pueden verse los nodos que actúan sobre Gazebo para el acople de los sensores y como los *topics* que estos generan derivan en los algoritmos de SLAM y mapeo 3D. En la parte inferior también se observa el nodo *movebase* con sus *topics* más importantes y conectado al mapa, a los sistemas de coordenadas y al *behavior*. Con el autómata también se relaciona la herramienta de *Rviz* y los servidores de acciones.

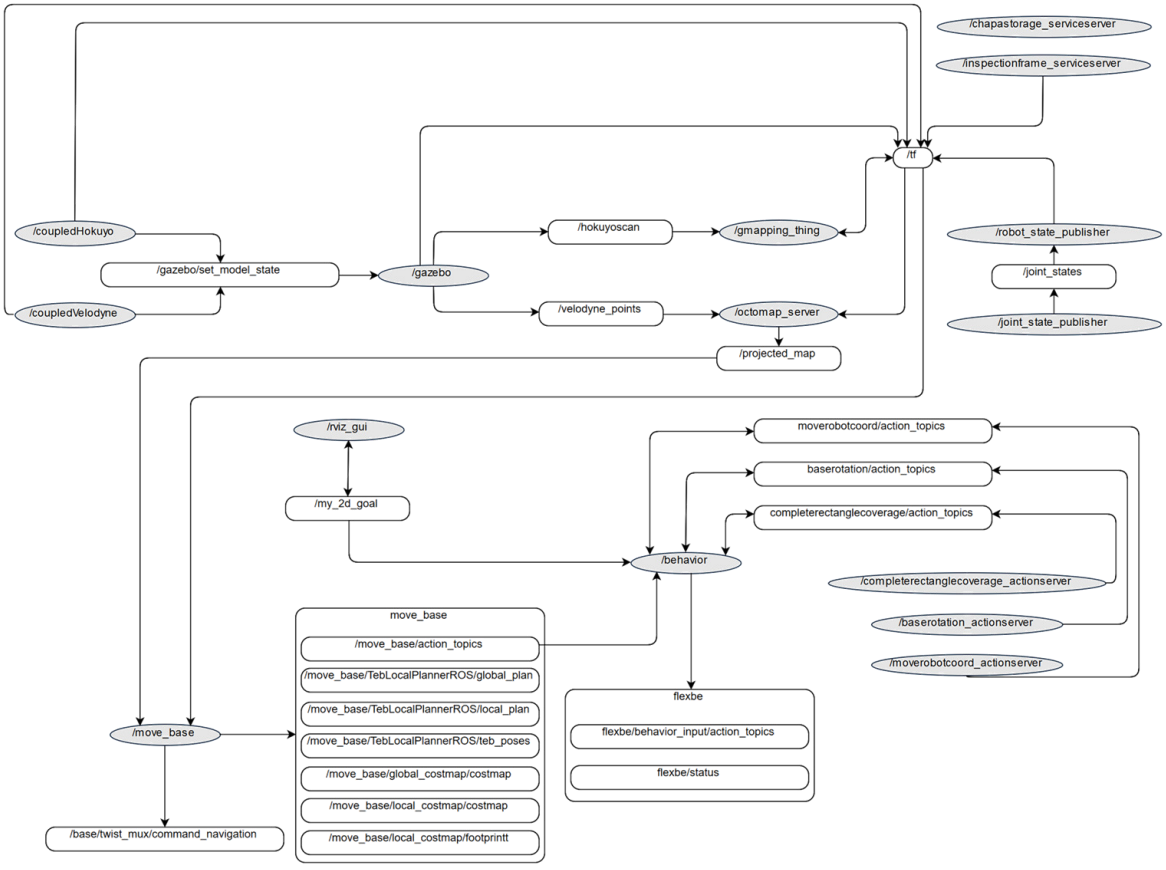


Figura 50: Representación gráfica de los principales nodos, representados por elipses, y *topics* involucrados en el trabajo

# 5. Experimentos y resultados

La valía de la implementación detallada en la anterior sección será analizada en las siguientes páginas estudiando el comportamiento conseguido en distintas simulaciones. Grabaciones de 3 de ellas y el análisis de los datos recogidos en estas pueden encontrarse junto a este informe en la carpeta de *Material adjunto*, cuyo contenido se explica en el *Anexo IV*.

La versión de ROS utilizada ha sido ROS Kinetic Kame instalada en un sistema operativo Ubuntu 16.04 (Xenial), el recomendado por la librería de robótica. En cuanto a hardware, las simulaciones han sido ejecutadas en un ordenador portátil Lenovo B50-80 con 11,6GB de memoria RAM y procesador *Intel* i5-5200U de 4 núcleos (2,2GHz).

El gasto computacional de la simulación, sobre todo el relativo al sensor láser 3D, resultó bastante elevado para el hardware disponible. El hecho de alcanzar ratios de 0.05 entre el tiempo simulado y el tiempo real hacen imposible realizar un análisis serio sobre la velocidad de procesamiento del método empleado. Sin embargo, tanto pequeñas pruebas realizadas en ordenadores más potentes, así como la múltiple documentación sobre los paquetes, que están ampliamente testeados en el mundo real, invitan a pensar que el método podrá trasladarse y funcionar sin problemas en un robot real, donde no será necesaria la simulación de sensores y el entorno.

El algoritmo de SLAM de *Gmapping* ha cumplido de manera satisfactoria con su propósito, proporcionando al sistema una localización aceptable del robot. Si bien estos resultados se dan en un medio simulado en el que los sensores tienen un comportamiento más ideal que en la realidad, en la Figura 51 puede verse como la trayectoria del robot se replica casi a la perfección. También puede observarse como la trayectoria generada por el SLAM mejora notablemente a la referida únicamente a la odometría. Para lograr este estudio se ha creado un sistema de coordenadas adicional que se mantiene fijo en el origen real de la trayectoria del robot.

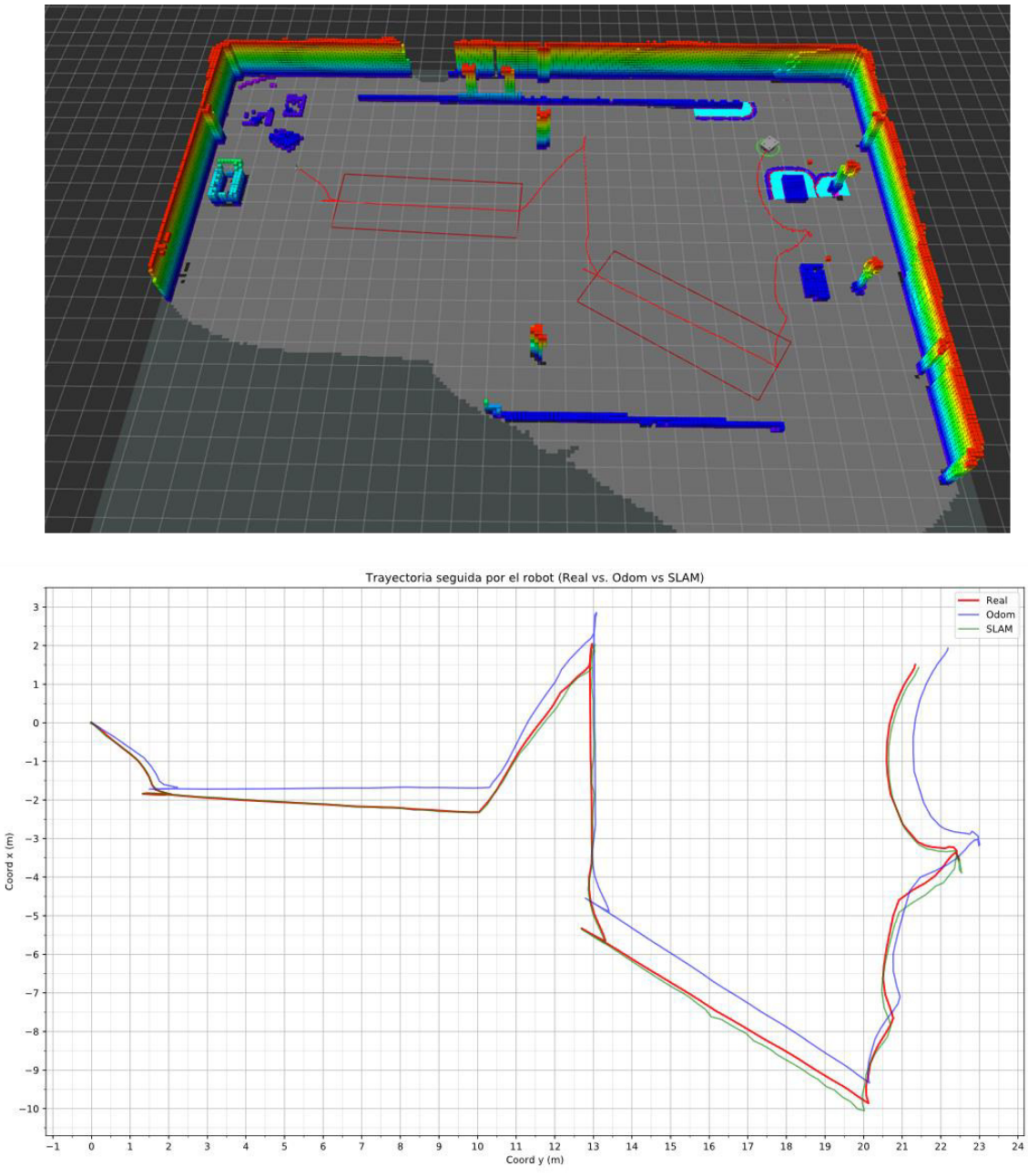


Figura 51: Arriba trayectoria real seguida por el robot en un entorno simulado de fábrica. Abajo gráfica que compara las trayectorias registradas por los distintos sistemas de coordenadas.

Realizando un análisis más preciso de los datos, en la Tabla 4 puede observarse como la distancia máxima entre la pose estimada por *Gmapping* y la pose real del robot en ninguno de los tres casos estudiados supera los 40 cm. Se observa que en todos los casos la media no supera los 10 cm.

| | Caso 1 (test) | Caso 2 (laboratorio) | Caso 3 (fábrica) |
|---|---|---|---|
| **Nº de muestras** | 191 | 284 | 230 |
| **mean** | 0.0574 | 0.0721 | 0.0979 |
| **std** | 0.0476 | 0.0335 | 0.0921 |
| **min** | 0.0014 | 0.0028 | 0.0010 |
| **max** | 0.2427 | 0.1979 | 0.3867 |

Tabla 4: Estadísticas de la distancia entre la pose real y la estimada por el SLAM

Esta precisión también se da en la estimación de la orientación del robot. En la Tabla 5 se observa como el error medio en la orientación del robot apenas supera los 2º.

| | Caso 1 (test) | Caso 2 (laboratorio) | Caso 3 (fábrica) |
|---|---|---|---|
| **Nº de muestras** | 191 | 284 | 230 |
| **mean** | 2.02082 | 1.8011 | 0.9235 |
| **std** | 2.0998 | 2.3266 | 0.9130 |
| **min** | 0.0870 | 0.0120 | 0.0010 |
| **max** | 10.3830 | 15.5710 | 6.3430 |

Tabla 5: Estadísticas de la diferencia en valor absoluto entre la orientación real del robot y la estimada por el SLAM

La mejora que supone el algoritmo de localización frente a la simple odometría puede visualizarse en la Figura 52. Aunque en este caso solo se observa el error absoluto de la coordenada *y* el comportamiento con la coordenada *x* y en el resto de los casos es equivalente. Se puede ver como el valor de la odometría va aumentando siendo incapaz de corregir el error. El SLAM, sin embargo, es capaz de corregir sus malas posiciones y mantiene un error mucho más admisible.

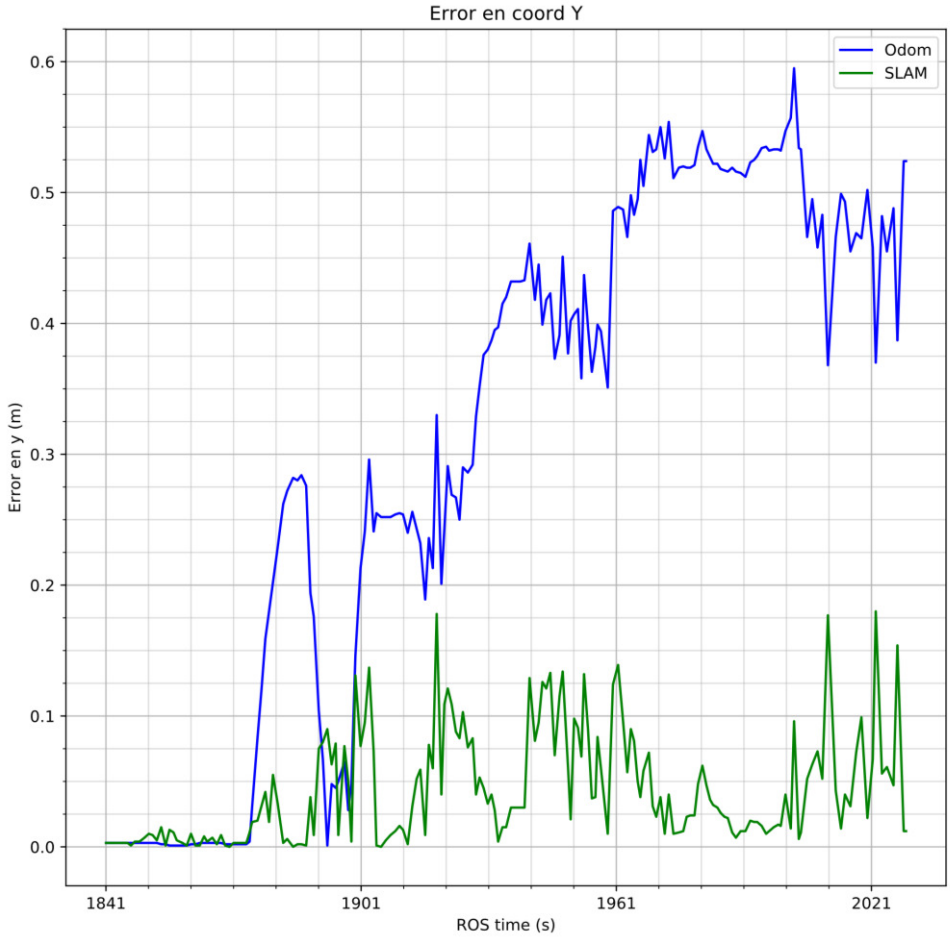


Figura 52: Gráfica que representa el error absoluto en la predicción de la coordenada Y del robot en el Caso 1.

Se concluye por tanto que el SLAM proporciona una precisión y veracidad admisibles para el propósito de este trabajo. Los errores cometidos, incluso en sus máximas expresiones son compatibles con la navegación autónoma al afrontarse esta desde un punto de vista conservador, con medidas como, por ejemplo, la inflación de obstáculos.

El análisis cuantitativo del modelado del entorno no es posible de manera tan directa. Sin embargo, en todas las simulaciones se ha logrado un mapeo satisfactorio detectándose la gran mayoría de posibles obstáculos para el robot.

Aunque no de manera habitual se han dado algunos casos en los que se detectaban como obstáculos puntos libres en la realidad. Si bien este tipo de error resulta menos perjudicial que el error opuesto supone que la trayectoria diseñada por el planificador esté muy lejos

de ser la óptima. En cualquier caso, pudo comprobarse que el *voxel* erróneo se corrige con el paso del tiempo al realizarse sobre su volumen nuevas mediciones.

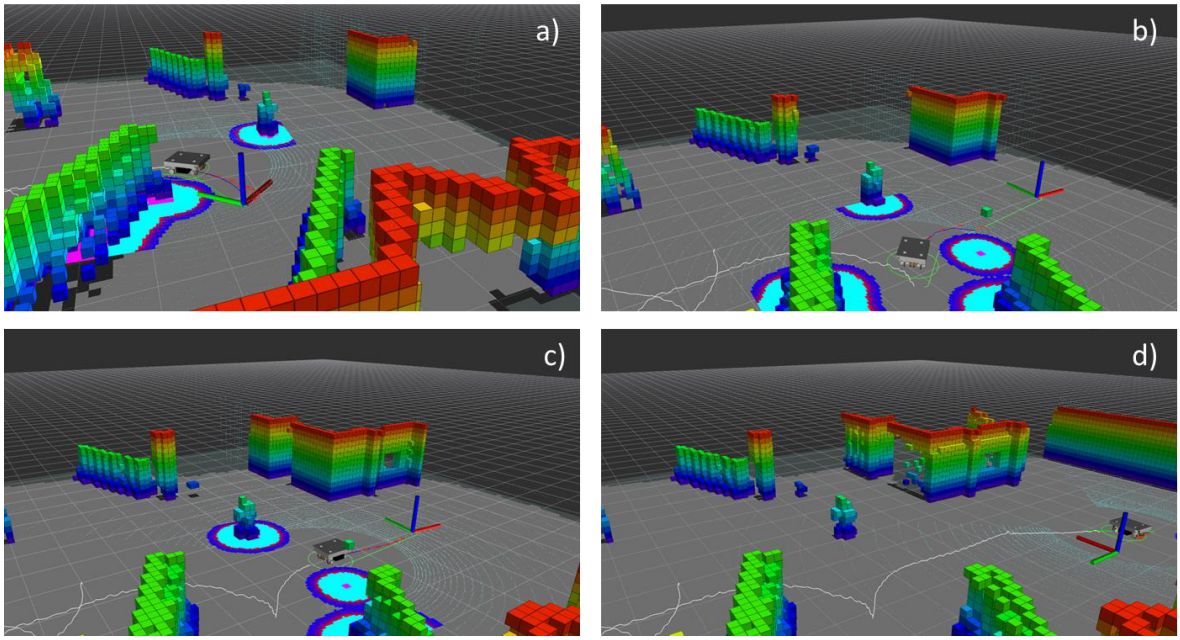


Figura 53: Secuencia de imágenes que muestra el error en la estimación de un *voxel* y su posterior corrección.

Un error que deberá atenderse en la realidad es la detección de las chapas en el suelo. A pesar de que se aprecian las perturbaciones en la nube de puntos del sensor, el filtrado del suelo parece eliminar esos obstáculos, Figura 54.

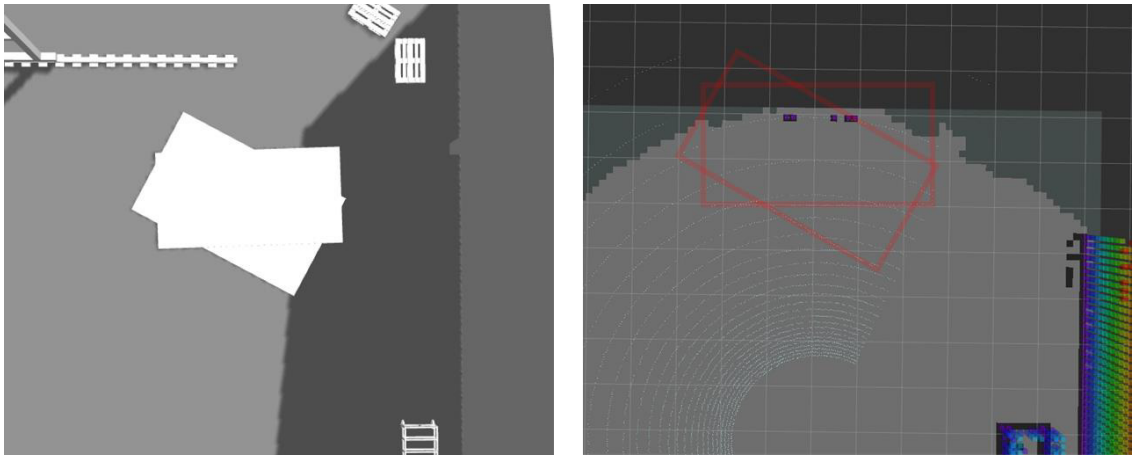

Figura 54: A la izquierda, imágenes de las chapas en Gazebo. A la derecha, error en la construcción del mapa 3D.

Al tratarse de elementos planos de un grosor muy fino se ha considerado que la detección de estos tipos de elementos escapa a los objetivos planteados para este trabajo. Sin embargo, en el futuro se deberá acometer sobre un prototipo real un reajuste más preciso de los parámetros o la inclusión de determinadas excepciones al mismo.

Por último, la planificación *TEB* ha funcionado satisfactoriamente en los casos presentados. Se ha observado como no se aprovecha al 100% la omnidireccionalidad del vehículo dado que es común que el robot priorice avanzar en la dirección *x* de su base. Si bien esto provoca en ocasiones que la trayectoria no sea la más rápida posible, los parámetros se han mantenido de este modo al ser los proporcionados por el fabricante y al no tratarse de una perdida sustancial de eficiencia. Por lo demás el planificador ha sido capaz de alcanzar todas las posiciones objetivo y de esquivar todos los obstáculos, incluso apariciones imprevistas como la de la Figura 53.

# 6. Discusión

A tenor de los resultados comentados en la anterior sección se puede concluir que, mediante la implementación, se ha logrado alcanzar el comportamiento deseado en el diseño inicial. También se ha comprobado la posibilidad de combinar los distintos algoritmos en una solución integral y que estos ofrecen el comportamiento descrito en sus respectivas documentaciones. La implementación del autómata de estados supone un gran paso en el proyecto general, pues a partir de esta base se podrán integrar con menor esfuerzo nuevos comportamientos.

Al estar el proyecto orientado, de manera ideal, hacia un funcionamiento de robótica colaborativa, un importante aspecto a abordar como trabajo futuro, será la consecución de una navegación y un mapeo 3D capaz de garantizar la seguridad en un entorno dinámico. Aunque las librerías escogidas están pensadas para satisfacer estos requisitos, se deberá llevar a cabo un proceso de pruebas con los distintos parámetros y configuraciones para definir de manera precisa las limitaciones del funcionamiento en estas condiciones.

Otra cuestión que afrontar será una mejora en la gestión de los datos de la chapa. Actualmente, la posición, orientación y tamaño de las chapas son valores fijos que se almacenan en un servicio. En un futuro, esta información deberá ser coordinada de manera dinámica con el módulo de detección de la chapa. Las chapas pueden introducirse o retirarse del entorno e, incluso, las dimensiones pueden variar si se produce un error en la primera detección.

Existen otras posibles mejoras en la inspección de las chapas. El primero de ellos es que hasta ahora no se contempla de qué manera el robot se incorporará a la chapa. Si bien el grosor de estas y las ruedas escogidas para el robot deberían hacer que no suponga un problema subirse a la hoja de acero, quizás este proceso podría acarrear cierros problemas para los algoritmos de mapeo y localización. Se deberán explorar esas situaciones y decidir que posibles soluciones o excepciones debe implementarse.

En el estado actual del proyecto no se contempla tampoco la aparición de posibles obstáculos durante la inspección de la plancha de acero. Resultará necesario implementar de manera paralela la detección de estos obstáculos de manera que, si se encuentra alguno, el proceso de inspeccionado se detenga. La librería utilizada para implementar las

máquinas de estados contempla la creación de estados paralelos por lo que esta mejora no debería suponer un gran problema.

Aunque se han logrado alcanzar resultados satisfactorios la conclusión de este trabajo no deja de conformar una versión prototípica de un proyecto en la que tienen cabida distintas mejoras. La mayoría de los aspectos a mejorar giran en torno al acercamiento al problema real. En este sentido, un paso importante será el uso de un robot y unos sensores físicos reales.

Un aspecto muy positivo es que, gracias al trabajo hasta ahora realizado, se dispone un simulador fácilmente modificable en el que podrán probarse modelos de sensores reales con el fin de probar si son válidos para nuestros propósitos. Adicionalmente, como se ha explicado a lo largo del trabajo, el funcionamiento es independiente del entorno de simulación por lo que las modificaciones para lograr el funcionamiento del código con un robot real no deberían suponer un gran trabajo.

Una tarea cuya ejecución es inevitable es la integración del módulo de navegación, tratado en este trabajo, con el módulo de localización e inspección de la chapa y el módulo de control del brazo reparador. La concepción modular del proyecto y del propio ROS deberían facilitar este cometido. La capacidad de la máquina de estados para trabajar con estados paralelos también podrá ser de gran utilidad.

Por último, y con una vista más a largo plazo, se deberá afrontar la creación de una interfaz de usuario que combine la interacción que se realiza en el entorno de la máquina de estados, *FlexBEApp* [122], y en la herramienta de visualización, *Rviz* [87]. Será necesario hacer que esta interfaz sea más intuitiva y funcione a más alto nivel que el actual, para el cual es necesario un conocimiento avanzado del conocimiento del proyecto.

# 7. Conclusiones

Finalizadas las explicaciones y el análisis de los resultados obtenidos se puede concluir que los objetivos marcados al comienzo del proyecto se han cumplido. Tras el trabajo realizado se dispone de un robot simulado, cinemáticamente equivalente al futuro robot físico, y que es capaz de navegar por el entorno y realizar los movimientos necesarios para una inspección.

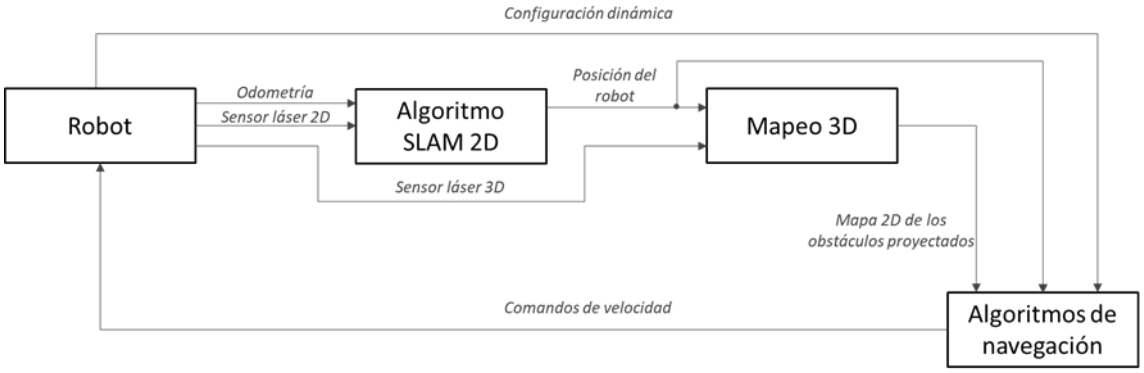


Figura 55: Esquema general de la estrategia general de navegación

Recuperando la figura utilizada en la introducción del informe se puede comprobar que todos los módulos se han resuelto de manera satisfactoria. El algoritmo de SLAM utilizado ha arrojado unos buenos resultados proporcionando un sistema de ubicación al robot. La futura combinación de este sistema con una localización más precisa en momentos clave, como la inspección de las chapas, parece una solución viable. El mapeo 3D por su parte ha demostrado la capacidad de *Octomap* para generar entornos dinámicos y fiables. Por último, la navegación del robot ha respondido de manera correcta y ha podido ser integrada en la lógica de una máquina de estados que coordina los comportamientos del robot.

El proyecto, integrado en un entorno de investigación y desarrollo, ha servido también para abordar el estudio de múltiples alternativas, algunas de ellas recogidas en el estado del arte de este informe. De esta manera, no se ha realizado simplemente un desarrollo de códigos e implementación de librerías, sino que se ha logrado obtener una visión general del problema de la navegación autónoma en robótica.

Entre las competencias adquiridas desde el comienzo del proyecto, se encuentra la introducción al entorno de robótica ROS desde cero hasta la capacidad de desarrollar un sistema complejo como el que aquí se presenta. Los conceptos y problemas de la robótica orientada a bases móviles con ruedas también han resultado novedosos y el trabajo ha servido como una introducción de ese campo.

El desarrollo del proyecto ha servido también para asentar algunos de los conocimientos adquiridos tanto en el máster cursado como, previamente, en el grado. Los conocimientos sobre robótica, autómatas de estados, programación C++ y Python, análisis y visualización de datos, y manejo de sistemas operativos Linux han resultado indispensables para el desarrollo del presente trabajo.

# 8. Referencias

[1] «ISO 8373:2012(en), Robots and robotic devices — Vocabulary». [En línea]. Disponible en: https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-2:v1:en. [Accedido: 16-dic-2018].
[2] P. Corke, *Robotics, vision and control*. New York, NY: Springer Berlin Heidelberg, 2017.
[3] G. Campion, G. Bastin, y B. D'AndrCa-Novel, «Structural Properties and Classification of Kinematic and Dynamic Models of Wheeled Mobile Robots», p. 16.
[4] S. Yu, C. Ye, H. Liu, y J. Chen, «Development of an omnidirectional Automated Guided Vehicle with MY3 wheels», *Perspect. Sci.*, vol. 7, pp. 364-368, mar. 2016.
[5] *Springer handbook of robotics*, 2nd edition. New York, NY: Springer Berlin Heidelberg, 2016.
[6] G. Klančar, A. Zdešar, S. Blažič, y I. Škrjanc, Eds., *Wheeled mobile robotics: from fundamentals towards autonomous systems*. Oxford: Butterworth-Heinemann, an imprint of Elsevier, 2017.
[7] R. Siegwart, I. R. Nourbakhsh, y D. Scaramuzza, *Introduction to autonomous mobile robots*, 2nd ed. Cambridge, Mass: MIT Press, 2011.
[8] R. G. Boboc, H. Moga, y D. Talabă, «A REVIEW OF CURRENT APPLICATIONS IN TELEOPERATION OF MOBILE ROBOTS», vol. 5, n.º 2, p. 8, 2012.
[9] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*, vol. 95. 1986.
[10] I. Rañó, «A systematic analysis of the Braitenberg vehicle 2b for point-like stimulus sources», *Bioinspir. Biomim.*, vol. 7, n.º 3, p. 036015, sep. 2012.
[11] V. Lumelsky y A. Stepanov, «Dynamic path planning for a mobile automaton with limited information on the environment», *IEEE Trans. Autom. Control*, vol. 31, n.º 11, pp. 1058-1063, nov. 1986.
[12] A. Elfes, «Using occupancy grids for mobile robot perception and navigation», *Computer*, vol. 22, n.º 6, pp. 46-57, jun. 1989.
[13] H. P. Moravec, «Sensor Fusion in Certainty Grids for Mobile Robots», en *Sensor Devices and Systems for Robotics*, A. Casals, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 253-276.
[14] Z. Chen, «Bayesian filtering: From Kalman filters to particle filters, and beyond», p. 69.
[15] S. Thrun, «Map Learning and High-Speed Navigation in RHINO», p. 24.
[16] J. Biswas y M. Veloso, «WiFi localization and navigation for autonomous indoor mobile robots», en *2010 IEEE International Conference on Robotics and Automation*, Anchorage, AK, 2010, pp. 4379-4384.
[17] Hee Jin Sohn y Byung Kook Kim, «A Robust Localization Algorithm for Mobile Robots with Laser Range Finders», en *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Barcelona, Spain, 2005, pp. 3545-3550.
[18] S. C. Diamantas y R. M. Crowder, «Localisation and Mapping Using a Laser Range Finder: A Goal-Seeking Approach», en *2009 Fifth International Conference on Autonomic and Autonomous Systems*, Valencia, Spain, 2009, pp. 270-276.
[19] E. Royer, M. Lhuillier, M. Dhome, y J.-M. Lavest, «Monocular Vision for Mobile Robot Localization and Autonomous Navigation», *Int. J. Comput. Vis.*, vol. 74, n.º 3, pp. 237-260, jul. 2007.
[20] M. Agrawal y K. Konolige, «Real-time Localization in Outdoor Environments using Stereo Vision and Inexpensive GPS», en *18th International Conference on Pattern Recognition (ICPR'06)*, Hong Kong, China, 2006, pp. 1063-1068.
[21] P. Lightbody, T. Krajník, y M. Hanheide, «An efficient visual fiducial localisation system», *ACM SIGAPP Appl. Comput. Rev.*, vol. 17, n.º 3, pp. 28-37, nov. 2017.
[22] A. C. Murillo, J. J. Guerrero, y C. Sagues, «SURF features for efficient robot localization with omnidirectional images», en *Proceedings 2007 IEEE International Conference on Robotics and Automation*, Rome, Italy, 2007, pp. 3901-3907.

[23] R. E. Kalman, «A New Approach to Linear Filtering and Prediction Problems», *Trans. ASME--J. Basic Eng.*, vol. 82, pp. 35-45, 1960.
[24] G. G. Rigatos, «Extended Kalman and Particle Filtering for sensor fusion in mobile robot localization», p. 6.
[25] S. Thrun, W. Burgard, y D. Fox, *Probabilistic Robotics*. The MIT Press, 2000.
[26] C. Stachniss, «Exploration and Mapping with Mobile Robots», p. 245.
[27] G. Klein y D. Murray, «Parallel Tracking and Mapping for Small AR Workspaces», en *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, Nara, Japan, 2007, pp. 1-10.
[28] R. Mur-Artal, J. M. M. Montiel, y J. D. Tardos, «ORB-SLAM: A Versatile and Accurate Monocular SLAM System», *IEEE Trans. Robot.*, vol. 31, n.º 5, pp. 1147-1163, oct. 2015.
[29] R. Mur-Artal y J. D. Tardos, «ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras», *IEEE Trans. Robot.*, vol. 33, n.º 5, pp. 1255-1262, oct. 2017.
[30] M. Labbe y F. Michaud, «RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation», p. 40.
[31] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, y W. Burgard, «An evaluation of the RGB-D SLAM system», en *2012 IEEE International Conference on Robotics and Automation*, St Paul, MN, USA, 2012, pp. 1691-1696.
[32] W. Hess, D. Kohler, H. Rapp, y D. Andor, «Real-time loop closure in 2D LIDAR SLAM», en *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden, 2016, pp. 1271-1278.
[33] E. Nelson, *B(erkeley) L(ocalization) A(nd) M(apping)!-GitHub*. 2019.
[34] «Team Hector - Team Hector». [En línea]. Disponible en: http://www.teamhector.de/. [Accedido: 15-ene-2019].
[35] T. Darmstadt y S. Kohlbrecher, «Leveraging 3D Perception for RoboCup Rescue and the DARPA Robotics Challenge», p. 51.
[36] S. Kohlbrecher, O. von Stryk, J. Meyer, y U. Klingauf, «A flexible and scalable SLAM system with full 3D motion estimation», en *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, Kyoto, Japan, 2011, pp. 155-160.
[37] «hector_slam - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/hector_slam. [Accedido: 15-ene-2019].
[38] B. D. Lucas y T. Kanade, «An Iterative Image Registration Technique with an Application to Stereo Vision», 1981, pp. 121–130.
[39] G. Grisettiyz, C. Stachniss, y W. Burgard, «Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling», en *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Barcelona, Spain, 2005, pp. 2432-2437.
[40] G. Grisetti, C. Stachniss, y W. Burgard, «Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters», *IEEE Trans. Robot.*, vol. 23, n.º 1, pp. 34-46, feb. 2007.
[41] «gmapping - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/gmapping. [Accedido: 22-nov-2018].
[42] F. Pomerleau, F. Colas, R. Siegwart, y S. Magnenat, «Comparing ICP variants on real-world data sets: Open-source library and experimental protocol», *Auton. Robots*, vol. 34, n.º 3, pp. 133-148, abr. 2013.
[43] «libpointmatcher - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/libpointmatcher?distro=hydro. [Accedido: 16-ene-2019].
[44] P. J. Besl y N. D. McKay, «A method for registration of 3-D shapes», *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, n.º 2, pp. 239-256, feb. 1992.
[45] R. B. Rusu y S. Cousins, «3D is here: Point Cloud Library (PCL)», en *2011 IEEE International Conference on Robotics and Automation*, Shanghai, China, 2011, pp. 1-4.

[46] «pcl_ros - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/pcl_ros. [Accedido: 16-ene-2019].
[47] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, y W. Burgard, «OctoMap: an efficient probabilistic 3D mapping framework based on octrees», *Auton. Robots*, vol. 34, n.º 3, pp. 189-206, abr. 2013.
[48] «octomap_mapping - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/octomap_mapping?distro=kinetic. [Accedido: 21-nov-2018].
[49] R. Meagher, «The Octree Encoding Method for Efficient Solid Modeling.», p. 171.
[50] J. Canny, «A new algebraic method for robot motion planning and real geometry», en *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, Los Angeles, CA, USA, 1987, pp. 39-48.
[51] P. Bhattacharya y M. L. Gavrilova, «Voronoi diagram in optimal path planning», en *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, 2007, pp. 38-47.
[52] R. Dechter y J. Pearl, «Generalized Best-First Search Strategies and the Optimality of A», p. 32.
[53] Xiang Liu y Daoxiong Gong, «A comparative study of A-star algorithms for search and rescue in perfect maze», en *2011 International Conference on Electric Information and Control Engineering*, Wuhan, China, 2011, pp. 24-27.
[54] A. Felner, «Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm», p. 5.
[55] A. Abbadi, R. Matousek, S. Jancik, y J. Roupec, «RAPIDLY-EXPLORING RANDOM TREES: 3D PLANNING», *Unpublished*, 2012.
[56] S. X. Yang y C. Luo, «A Neural Network Approach to Complete Coverage Path Planning», *IEEE Trans. Syst. Man Cybern. Part B Cybern.*, vol. 34, n.º 1, pp. 718-724, feb. 2004.
[57] P. Vadakkepat, Kay Chen Tan, y Wang Ming-Liang, «Evolutionary artificial potential fields and their application in real time robot path planning», en *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, La Jolla, CA, USA, 2000, vol. 1, pp. 256-263.
[58] M. Seda, «Roadmap methods vs. cell decomposition in robot motion planning», 2007.
[59] D. Fox, W. Burgard, y S. Thrun, «The dynamic window approach to collision avoidance», *IEEE Robot. Autom. Mag.*, vol. 4, n.º 1, pp. 23-33, mar. 1997.
[60] S. Quinlan y O. Khatib, «Elastic bands: connecting path planning and control», en *[1993] Proceedings IEEE International Conference on Robotics and Automation*, Atlanta, GA, USA, 1993, pp. 802-807.
[61] B. Chazelle, «Approximation and decomposition of shapes», en *Advances in Robotics 1: Algorithmic and Geometric Aspects of Robotics*, 1987, pp. 145-185.
[62] H. Choset y P. Pignon, «Coverage Path Planning: The Boustrophedon Cellular Decomposition», en *Field and Service Robotics*, A. Zelinsky, Ed. London: Springer London, 1998, pp. 203-209.
[63] E. U. Acar, H. Choset, A. A. Rizzi, P. N. Atkar, y D. Hull, «Morse Decompositions for Coverage Tasks», *Int. J. Robot. Res.*, vol. 21, n.º 4, pp. 331-344, abr. 2002.
[64] S. C. Wong y B. A. MacDonald, «Complete Coverage by Mobile Robots Using Slice Decomposition Based on Natural Landmarks», en *PRICAI 2004: Trends in Artificial Intelligence*, vol. 3157, C. Zhang, H. W. Guesgen, y W.-K. Yeap, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 683-692.
[65] Y. Gabriely y E. Rimon, «Spiral-STC: an on-line coverage algorithm of grid environments by a mobile robot», en *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, 2002, vol. 1, pp. 954-960 vol.1.
[66] E. Galceran y M. Carreras, «A survey on coverage path planning for robotics», *Robot. Auton. Syst.*, vol. 61, n.º 12, pp. 1258-1276, dic. 2013.

[67] C. Rösmann, W. Feiten, T. Wosch, F. Hoffmann, y T. Bertram, «Efficient trajectory optimization using a sparse model», en *2013 European Conference on Mobile Robots*, Barcelona, Catalonia, Spain, 2013, pp. 138-143.
[68] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, y T. Bertram, «Trajectory modification considering dynamic constraints of autonomous robots», p. 6.
[69] «ROS.org | Powering the world's robots». [En línea]. Disponible en: http://www.ros.org/. [Accedido: 19-ene-2019].
[70] J. M. Santos, D. Portugal, y R. P. Rocha, «An evaluation of 2D SLAM techniques available in Robot Operating System», en *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Linkoping, Sweden, 2013, pp. 1-6.
[71] K. Murphy y S. Russell, «Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks», en *Sequential Monte Carlo Methods in Practice*, A. Doucet, N. Freitas, y N. Gordon, Eds. New York, NY: Springer New York, 2001, pp. 499-515.
[72] K. P. Murphy, «Bayesian Map Learning in Dynamic Environments», p. 7.
[73] G. Grisetti y C. Stachniss, «Learning Grid Maps with Rao-Blackwellized Particle Filters», p. 57.
[74] A. Doucet, N. de Freitas, y N. Gordon, «Sequential Monte Carlo Methods in Practice», p. 12.
[75] P. Abbeel, «Scan Matching Overview».
[76] «GKOctree - GameplayKit | Apple Developer Documentation». [En línea]. Disponible en: https://developer.apple.com/documentation/gameplaykit/gkoctree. [Accedido: 19-ene-2019].
[77] H. Moravec y A. Elfes, «High resolution maps from wide angle sonar», en *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, St. Louis, MO, USA, 1985, vol. 2, pp. 116-121.
[78] Stephanie, «Log Odds: Definiton and Worked Statistics Problems», *Statistics How To*, 14-dic-2017. [En línea]. Disponible en: https://www.statisticshowto.datasciencecentral.com/log-odds/. [Accedido: 28-ene-2019].
[79] M. Yguel, O. Aycard, y C. Laugier, «Update Policy of Dense Maps: Efficient Algorithms and Sparse Representation», en *Field and Service Robotics*, vol. 42, C. Laugier y R. Siegwart, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 23-33.
[80] M. A. Fischler y R. C. Bolles, «Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography», *ACM*, 1981, p. 42.
[81] «Hypergraph - Encyclopedia of Mathematics». [En línea]. Disponible en: https://www.encyclopediaofmath.org/index.php/Hypergraph. [Accedido: 28-ene-2019].
[82] «g2o: A General Framework for Graph Optimization». [En línea]. Disponible en: https://openslam-org.github.io/g2o.html. [Accedido: 27-ene-2019].
[83] R. Kummerle, G. Grisetti, H. Strasdat, K. Konolige, y W. Burgard, «g2o: A general framework for graph optimization», en *2011 IEEE International Conference on Robotics and Automation*, Shanghai, China, 2011, pp. 3607-3613.
[84] L. E. Kavraki, P. Svestka, J.-C. Latombe, y M. H. Overmars, «Probabilistic roadmaps for path planning in high-dimensional configuration spaces», *IEEE Trans. Robot. Autom.*, vol. 12, n.º 4, pp. 566-580, ago. 1996.
[85] C. Rösmann, F. Hoffmann, y T. Bertram, «Planning of multiple robot trajectories in distinctive topologies», en *2015 European Conference on Mobile Robots (ECMR)*, 2015, pp. 1-6.
[86] «Gazebo : Tutorial : ROS overview». [En línea]. Disponible en: http://gazebosim.org/tutorials?tut=ros_overview. [Accedido: 08-oct-2018].
[87] «rviz - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/rviz. [Accedido: 15-ene-2019].
[88] «SUMMIT-XL STEEL», *Robotnik*. [En línea]. Disponible en: https://www.robotnik.eu/mobile-robots/summit-xl-steel/. [Accedido: 19-sep-2018].

[89] «Seekur», *robots.ros.org*. [En línea]. Disponible en: https://robots.ros.org/seekur/. [Accedido: 08-oct-2018].
[90] «Neobotix- Technical-Data-Platforms». [En línea]. Disponible en: https://www.neobotix-robots.com/fileadmin/files/downloads/Allgemeines/Technical-Data-Platforms.pdf. [Accedido: 18-sep-2018].
[91] «Omni-Drive-Module Datasheet». Neobotix. [En línea]. Disponible en: https://www.neobotix-robots.com/fileadmin/files/downloads/Anleitungen/Fahr-Dreh-Modul-AssemblyInstructions.pdf. [Accedido: 08-oct-2018]
[92] «MPO-700 - Neobotix». [En línea]. Disponible en: https://www.neobotix-robots.com/omnidirectional-robot-mpo-700.html. [Accedido: 16-ene-2019].
[93] «geometry_msgs/Twist Documentation». [En línea]. Disponible en: http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/Twist.html. [Accedido: 01-feb-2019].
[94] «osrf / gazebo_models / source / hokuyo / model.sdf — Bitbucket». [En línea]. Disponible en: https://bitbucket.org/osrf/gazebo_models/src/6cd587c0a30e/hokuyo/model.sdf?at=default&fileviewer=file-view-default. [Accedido: 20-ene-2019].
[95] «sensor_msgs/LaserScan Documentation». [En línea]. Disponible en: http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html. [Accedido: 20-ene-2019].
[96] «gazebo_plugins - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/gazebo_plugins. [Accedido: 24-ene-2019].
[97] «DataspeedInc / velodyne_simulator». [En línea]. Disponible en: https://bitbucket.org/DataspeedInc/velodyne_simulator. [Accedido: 20-ene-2019].
[98] «VLP-16». [En línea]. Disponible en: https://velodynelidar.com/vlp-16.html. [Accedido: 05-dic-2018].
[99] «HDL-32E». [En línea]. Disponible en: https://velodynelidar.com/hdl-32e.html. [Accedido: 20-ene-2019].
[100] «sensor_msgs/PointCloud2 Documentation». [En línea]. Disponible en: http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/PointCloud2.html. [Accedido: 20-ene-2019].
[101] «tf - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/tf. [Accedido: 14-ene-2019].
[102] T. Foote, «tf: The transform library», en *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, Woburn, MA, USA, 2013, pp. 1-6.
[103] «tf2 - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/tf2. [Accedido: 14-ene-2019].
[104] «geometry_msgs - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/geometry_msgs. [Accedido: 22-ene-2019].
[105] «OpenSLAM.org». [En línea]. Disponible en: https://openslam-org.github.io/g2o.html. [Accedido: 21-nov-2018].
[106] «nav_msgs/OccupancyGrid Documentation». [En línea]. Disponible en: http://docs.ros.org/kinetic/api/nav_msgs/html/msg/OccupancyGrid.html. [Accedido: 25-ene-2019].
[107] «PCL - Point Cloud Library (PCL)». [En línea]. Disponible en: http://pointclouds.org/. [Accedido: 21-ene-2019].
[108] «navigation - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/navigation. [Accedido: 21-nov-2018].
[109] «move_base - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/move_base?distro=melodic. [Accedido: 25-ene-2019].
[110] «costmap_2d - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/costmap_2d. [Accedido: 25-ene-2019].

[111] «costmap_2d/hydro/staticmap - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/costmap_2d/hydro/staticmap. [Accedido: 25-ene-2019].
[112] «costmap_2d/hydro/inflation - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/costmap_2d/hydro/inflation. [Accedido: 25-ene-2019].
[113] «teb_local_planner - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/teb_local_planner. [Accedido: 21-nov-2018].
[114] *configuration and launch files for Neobotix MPO-700: neobotix/neo_mpo_700*. neobotix gmbh, 2018.
[115] «global_planner - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/global_planner. [Accedido: 01-feb-2019].
[116] «geometry_msgs/PoseStamped Documentation». [En línea]. Disponible en: http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html. [Accedido: 24-ene-2019].
[117] «navigation/Tutorials/Using rviz with the Navigation Stack - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack#A2D_Nav_Goal. [Accedido: 28-ene-2019].
[118] «FlexBE Behavior Engine». [En línea]. Disponible en: http://philserver.bplaced.net/fbe/. [Accedido: 05-dic-2018].
[119] I. Maurtua, A. Ibarguren, J. Kildal, L. Susperregi, y B. Sierra, «Human–robot collaboration in industrial applications: Safety, interaction and trust», *Int. J. Adv. Robot. Syst.*, vol. 14, n.º 4, p. 172988141771601, jul. 2017.
[120] «flexbe/Tutorials/The State Lifecycle - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/flexbe/Tutorials/The%20State%20Lifecycle. [Accedido: 15-ene-2019].
[121] «geometry_msgs/PolygonStamped Documentation». [En línea]. Disponible en: http://docs.ros.org/lunar/api/geometry_msgs/html/msg/PolygonStamped.html. [Accedido: 26-ene-2019].
[122] «User interface (editor + runtime control) for the FlexBE behavior engine: FlexBE/flexbe_app». [En línea]. Disponible en: https://github.com/FlexBE/flexbe_app. [Accedido: 24-ene-2019].

# ANEXO I: TERMINOLOGÍA BÁSICA DE ROS

# 1. Terminología básica de ROS

El conjunto de las estrategias que engloba el proyecto se enmarca en ROS (Robot Operating System) [1], un entorno de trabajo de software libre que permite la integración de distintas librerías. Este sistema operativo, impulsado por el *Willow Garage* [2] se encuentra ampliamente extendido en el campo de la robótica lo que ha provocado el sucesivo aumento de implementaciones *software* para la misma.

De esta manera, ROS permite comunicar las distintas partes del proyecto desde las lecturas de los distintos sensores a estrategias tan diversas como la localización del robot mediante visión artificial, el control de las ruedas de la base móvil o el manejo del efector que se encargará de la reparación en la chapa.

En este anexo se cubrirá la explicación de la terminología básica asociada al entorno de manera que sea posible para un usuario no experto seguir sin dificultades el resto del informe. Para ello se atenderá a la explicación de los conceptos básicos del entorno ROS desde dos enfoques, siguiendo el ejemplo de la documentación de ROS [3].

## 1.1. ROS como un sistema de archivos

**Paquetes**

Cada una de las librerías de código que componen ROS se denominan paquetes, o metapaquetes si, a su vez, contienen librerías diferenciadas en su interior. Dentro de ellos se almacenan los programas implementados en C++ o Python así como los archivos de configuración en los correspondientes formatos (*.yaml*, *.cfg*, *.csv* …)

**Mensajes, servicios y acciones**

Estos archivos del tipo .msg, .srv y .action. permiten generar a través de archivos muy simples la generación de clases C++ destinadas a la comunicación en ROS. Ejemplos de la estructura de estos archivos se puede ver en la siguiente Figura 1.

| .msg | .srv | .action |
|---|---|---|
| int32 x<br>int32 y | *#Request*<br>float32 a<br>float32 b<br>---<br>*#Reply*<br>float32 sum | *# Goal*<br>uint32 id<br>---<br>*# Result*<br>uint32 total_done<br>---<br>*#Feedback*<br>float32 percent_done |

Figura 1: Ejemplo de la estructura de los archivos .msg, .srv, .action

**Launch**

Este tipo de archivos permiten ejecutar en el mismo terminal varios nodos e incluye. cuando es necesario, un nodo maestro, [4].

## 1.2. ROS como un sistema gráfico de computación

**Nodos**

Los nodos son los distintos procesos que se ejecutan en el entorno. De manera general un proyecto de robótica comprenderá varios nodos: lectura de un sensor, control de un accionamiento, ejecución de un algoritmo complejo, etc. Los nodos se escriben en lenguaje, generalmente C++ o Python que se conectan con el entorno por medio de las librerías *roscpp*[5] y *rospy*[6].

**Maestro**

Este nodo es el encargado de coordinar el resto ejecutables. Sin el maestro, el resto de los nodos no podrán localizarse ni comunicarse mediante mensajes.

**Servidor de parámetros**

El servidor de parámetros es un espacio generado por el maestro y común a todos los nodos para almacenar variables, [7].

***Topics***

Los *topics* constituyen el canal de transmisión a partir del cual los distintos nodos pueden comunicarse estableciendo comunicaciones de publicador-subscriptor. Cada *topic* se crea

otorgándole un nombre que le identifica y asociado con un tipo de dato concreto que se corresponde con el tipo de mensajes que puede contener.

Respecto a un *topic* un nodo puede actuar como un publicador, si envía mensajes al *topic*, o como un subscriptor, si lee los mensajes publicados en él. No es necesario que exista ninguna relación adicional al *topic* entre los publicadores y subscriptores, de manera que esta comunicación provee una cierta abstracción entre la producción y consumición de información. Cabe destacar que es posible cualquier número concurrente de publicadores y subscriptores e incluso nodos que ejercen ambas funciones. Un esquema simple del uso de un *topic* puede encontrarse en la Figura 2.

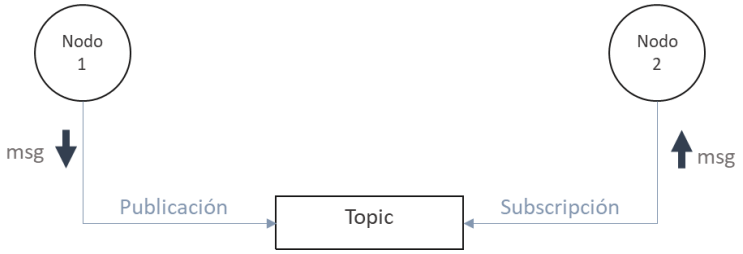


Figura 2: Representación de una comunicación publicador-subscriptor básica en ROS.

La creación de un programa subscriptor y uno publicador, así como la estructura de los archivos *.msg* a partir de los cuales se generan los mensajes pueden encontrarse en [8],[9] y [10].

**Servicios**

Otro modelo de comunicación esencial en la comunicación de nodos es el solicitud-respuesta. En ROS este sistema se logra a través de los servicios. Los mensajes involucrados se dividen en dos partes diferenciadas: una para la solicitud y otra para la contestación.

Será necesario, por tanto, un nodo que ofrezca un servicio, servidor, y otro que lo solicite, cliente. Cuando esto suceda, se ejecutará en el primero lo cálculos pertinentes mientras el cliente está a la espera. Cuando el servidor devuelva la respuesta el cliente continuará su ejecución. Un esquema simple del uso de un servicio puede encontrarse en la Figura 3.

Figura 3: Representación de una comunicación servidor-cliente básica en ROS.

La creación de un programa cliente y uno servidor, así como la estructura de los archivos *.srv* a partir de los cuales se generan los mensajes pueden encontrarse en [11] ,[12] y [13].

**Acciones**

Las acciones, incluidas en el paquete *action_lib* [14], suponen una pequeña variación respecto a los servicios. Estas al estar pensadas para procesos más largos no detienen la ejecución del cliente mientras espera por la respuesta.

El mensaje se dividirá en este caso en tres en lugar de en dos partes: *goal*, *result* y *feedback*. Los dos primeros mensajes resultan respectivamente equivalentes a *request* y *reply*. Por su parte, *feedback* hace referencia a los mensajes de realimentación que puede enviar el servidor durante la ejecución de la opción y permiten al cliente obtener información de la evolución del proceso. Un esquema simple del uso de una acción puede encontrarse en la Figura 4

Figura 4: Representación de la ejecución de acciones en ROS.

La creación de un programa cliente y uno servidor, así como la estructura de los archivos *.action* a partir de los cuales se generan los mensajes pueden encontrarse en [14].

**Bags**

Permiten guardar la información publicada en los *topics* y reproducirla cuando sea necesario respetando la coherencia temporal de los mismos. [15].

## 1.3. Bibliografía

[1] «ROS.org | Powering the world's robots». .
[2] «Willow Garage». [En línea]. Disponible en: http://www.willowgarage.com/. [Accedido: 29-ene-2019].
[3] «ROS/Concepts - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/ROS/Concepts. [Accedido: 29-ene-2019].
[4] «roslaunch - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/roslaunch. [Accedido: 29-ene-2019].
[5] «roscpp - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/roscpp. [Accedido: 29-ene-2019].
[6] «rospy - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/rospy. [Accedido: 29-ene-2019].
[7] «Parameter Server - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/Parameter%20Server. [Accedido: 29-ene-2019].
[8] «msg - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/msg. [Accedido: 29-ene-2019].
[9] «ROS/Tutorials/WritingPublisherSubscriber(c++) - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29. [Accedido: 29-ene-2019].
[10] «ROS/Tutorials/WritingPublisherSubscriber(python) - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29. [Accedido: 29-ene-2019].
[11] «srv - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/srv. [Accedido: 29-ene-2019].
[12] «ROS/Tutorials/WritingServiceClient(c++) - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29. [Accedido: 29-ene-2019].
[13] «ROS/Tutorials/WritingServiceClient(python) - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29. [Accedido: 29-ene-2019].
[14] «actionlib - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/actionlib. [Accedido: 11-ene-2019].
[15] «ROS/Tutorials/Recording and playing back data - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data. [Accedido: 29-ene-2019].

# ANEXO II: PRESUPUESTO

# 1. Costes de ejecución material

El coste de ejecución material incluye tres categorías, coste de equipos, coste de software y coste de mano de obra por el tiempo empleado en el proyecto. Para los equipos se considera una amortización de 3 años.

## 1.1. Costes de equipos

| CONCEPTO | PRECIO UNITARIO | CANTIDAD | SUBTOTAL |
|---|---|---|---|
| Ordenador portátil Lenovo B50-80 | 527 € | 0,178 | 93,85 € |

**Subtotal: …… 93,85 €**

## 1.2. Costes de software

| CONCEPTO | PRECIO UNITARIO | CANTIDAD | SUBTOTAL |
|---|---|---|---|
| Sistema operativo Linux: Ubuntu 16.04 | 0 € | 1 | 0 € |
| ROS Kinetic Kame y otras librerías *software* | 0 € | 1 | 0 € |
| Microsoft Office 365 | 10,70 €/mes | 1,5 | 16.05 € |

**Subtotal: …… 16,05 €**

## 1.3. Costes de mano de obra

El coste unitario de la mano de obra será de 30.00 €/h. La dedicación será de 25 horas semanales.

| CONCEPTO | CANTIDAD | SUBTOTAL |
|---|---|---|
| Estudio del problema | 5 días | 750,00 € |
| Estudio del entorno ROS | 15 días | 2250,00 € |
| Selección del modelo de robot | 15 días | 2250,00 € |
| Estudio y elección del algoritmo de SLAM | 25 días | 3750,00 € |
| Estudio y elección del algoritmo de mapeo 3D | 5 días | 750,00 € |
| Estudio y elección del algoritmo de navegación | 20 días | 3000,00 € |
| Diseño del autómata de estados | 5 días | 750,00 € |
| Implementación del autómata de estados | 30 días | 4500,00 € |
| Pruebas de funcionamiento | 20 días | 3000,00 € |
| Extracción de resultados | 10 días | 1500,00 € |
| Realización de la documentación | 45 días | 6750,00 € |

**Subtotal: ……. 29250.00 €**

## 1.4. Coste total del presupuesto de ejecución material

| CONCEPTO | SUBTOTAL |
|---|---|
| Coste de equipos | 93,85 € |
| Coste de software | 16,05 € |
| Coste de mano de obra | 29250,00 € |

**Subtotal: ……. 29359,90€**

# 2. Gastos generales y beneficio industrial

Los gastos generales y beneficio industrial son los gastos obligados que se derivan de la utilización de las instalaciones de trabajo más el beneficio industrial. Se estima un porcentaje del 10 % para los gastos generales y un 6% de beneficio.

| CONCEPTO | SUBTOTAL |
|---|---|
| **Gastos generales y beneficio industrial** | 54697,58 € |

# 3. Importe total

| CONCEPTO | SUBTOTAL |
|---|---|
| **Coste total del presupuesto de ejecución material** | 29359,90 € |
| **Gastos generales y beneficio industrial** | 4697,58 € |

**TOTAL:** ……. **34057,48 €**

**IVA 21%:** ……. **7152,07 €**

**TOTAL, IVA INCLUÍDO:** ……. **41209,55 €**

El Importe Total del proyecto suma la cantidad de:

**Cuarenta y Un Mil Doscientos Nueve, con Cincuenta y Cinco Céntimos**

# ANEXO III: CRONOGRAMA

| Tareas | 2018 | | | | | | | | | | | | | | | | 2019 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Septiembre | | | | Octubre | | | | Noviembre | | | | Diciembre | | | | Enero | | | | Febrero | | | |
| | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 | S1 | S2 | S3 | S4 |
| Estudio del problema | ■ | | | | | | | | | | | | | | | | | | | | | | | |
| Estudio del entorno ROS | ■ | ■ | ■ | | | | | | | | | | | | | | | | | | | | | |
| Selección del modelo de robot | | | ■ | ■ | ■ | | | | | | | | | | | | | | | | | | | |
| Estudio y elección del algoritmo de *SLAM* | | | | | ■ | ■ | ■ | ■ | ■ | | | | | | | | | | | | | | | |
| Estudio y elección del algoritmo de mapeo 3D | | | | | | | | ■ | | | | | | | | | | | | | | | | |
| Estudio y elección del algoritmo de navegación | | | | | | | | | ■ | ■ | ■ | ■ | | | | | | | | | | | | |
| Diseño del autómata de estados | | | | | | | | | | | | | ■ | | | | | | | | | | | |
| Implementación del autómata de estados | | | | | | | | | | | | | ■ | ■ | ■ | ■ | ■ | ■ | | | | | | |
| Pruebas de funcionamiento | | | | | | | | | | | | | | | | ■ | ■ | ■ | ■ | | | | | |
| Extracción de resultados | | | | | | | | | | | | | | | | | | | ■ | ■ | | | | |
| Realización de la documentación | | | | | | | | | | | | | | ■ | ■ | | ■ | ■ | ■ | ■ | ■ | ■ | ■ | |

Tabla 1: Cronograma estimado de la realización del trabajo

# ANEXO IV: MATERIAL ADJUNTO

# 1. Material adjunto

En este anexo se detallan el contenido de la carpeta que se incluye junto a la entrega de este informe. Este se divide en 3 carpetas: *Código*, *Análisis de resultados* y *Vídeos*.

En la primera pueden encontrar los paquetes, de creación propia o modificados, desarrollados para el proyecto. En total el espacio de trabajo se compone de 8 paquetes:

*spawn_package:* Código encargado de la carga del mundo, el robot y los sensores en el simulador.

*irobot_behaviors:* Conjunto de archivos que incluyen los estados implementados y el código principal de la máquina de estados.

*action_server_package:* Código que lanza los servidores de las acciones empleadas.

*aux_controllers:* Paquete que contiene las clases para el control simple del robot.

*my_rviz_tools:* Carpeta que contiene los archivos necesarios para la creación de una herramienta en el programa de visualización.

*irobot_move_base:* Paquete encargado de la configuración de la navegación autónoma.

*slam_launchers:* Paquetes encargados de la configuración del SLAM

*velodyne_simulator:* Paquete que incluye el modelo simulado del sensor 3D.

Cabe señalar que, al tratarse de un proyecto en desarrollo, ciertos elementos del código no se encuentran en su versión definitiva.

En la carpeta *Análisis de los resultados* se encuentran las gráficas y tablas obtenidas de en 3 simulaciones y que han sido explicados en la memoria.

Grabaciones de esas simulaciones se encuentran, por último, en la carpeta *Videos.* Las grabaciones se han realizado sobre archivos *bag,* explicados en el anexo de ROS. Un defecto en uno de ellos ha obligado a acelerar ligeramente el vídeo *"epilab.mov"* con el fin de no prolongar su duración de manera excesiva.