Trabajo de Fin de Grado

Ingeniería Electrónica, Robótica y Mecatrónica (US-UMA)

Desarrollo de programas y experimentación de navegación de un robot móvil terrestre

Autor: Víctor Borrero López

Tutor: Carlos Bordons Alba

Tutor externo: Ramón Andrés García Rodríguez

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de programas y experimentación de navegación de un robot móvil terrestre

Autor:
Víctor Borrero López

Tutor:
Carlos Bordons Alba
Profesor catedrático

Tutor externo:
Ramón Andrés García Rodríguez

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Proyecto Fin de Carrera: Desarrollo de programas y experimentación de navegación de un robot móvil terrestre

Autor: Víctor Borrero López

Tutor: Carlos Bordons Alba

Tutor externo: Ramón Andrés García Rodríguez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mis padres por haber confiado en mí siempre y por apoyarme cuando no era capaz de levantar la cabeza.

A mis hermanos por ser pilar fundamental y a mis amigos por ser siempre esa luz que me ilumina el camino.

A mi familia y a mis compañeros. Sin vosotros nunca habría llegado hasta aquí.

A Javi, Guille y David, por enseñarme tanto en tan poco tiempo.

Agradecimientos

Cuando uno se enfrasca en la temible aventura de graduarse en una ingeniería nunca lo hace solo. Son demasiadas las personas que me han acompañado durante estos casi cinco años que he tardado en conseguirlo. Por eso, hoy más que nunca quiero acordarme de todo aquel que ha estado a mi lado en algún momento de este lustro, apoyándome para que lo consiga, y que hoy ya no está en mi vida con la misma presencia que antes.

Por supuesto, no me olvido de los que siguen hoy conmigo. Os tengo más presentes que nunca, y estas humildes palabras nada pueden compararse con todo lo que habéis hecho por mí.

A papá y mamá, nunca podré pagaros en besos y abrazos lo que significáis para mí y lo mucho que os debo. Una vida entera es poca para devolveros toda vuestra entrega. Bien sabéis que la peripecia de graduarme en esta ingeniería ha requerido de alguna que otra pirueta mortal, alguna noche sin dormir y algún llanto de frustración. A pesar de ello, no ha faltado una palmada en la espalda, un gesto de amor o una ayuda incondicional. De pocas certezas puedo hacer gala, pero una de ellas es que os quiero muchísimo y nada de esto sería mínimamente posible sin vosotros.

A mis hermanos, deciros que es mágico teneros a vosotros dos, siempre con una sonrisa y un consejo que darme, más valioso que cualquier gema preciosa que se preste.

A mi familia, sois tantos que nombraros a todos sería alargar el proyecto diez páginas más. Nunca me ha faltado un mensaje por teléfono apoyándome, animándome a seguir hacia adelante y recordándome lo finito del sufrimiento cuando uno pone empeño por acabar con él.

A mis amigos, por ser el confesionario donde espantar mis mayores miedos y por ser el mayor catalizador de alegrías posible. No importa cuán lejos os he podido tener, siempre os he sentido cerca de mí, y eso no hay dinero que pueda pagarlo.

A mi familia del Erasmus, probablemente el mayor regalo que Nápoles tuvo el placer de entregarme. Vosotros y vosotras me habéis descubierto una parte de mí de la que estoy orgulloso, y me habéis enseñado a ser feliz sin importar la hora ni el lugar. Habéis hecho de ese rincón peligroso de Italia mi segunda casa y eso no lo olvidaré jamás.

A mis compañeros de clase, en especial a Javi, Guille y David, tres personas que aparecieron mágicamente en el primer curso de carrera y de los que hago gala como amigos a día de hoy, cinco años después de que entrase (cómo no) media hora tarde por la puerta de ese aula de la ETSI. Si Sevilla tiene un color especial para mí, es en parte gracias a vosotros, y tampoco hay tesoros en el mundo con los que pagaros todo lo que os debo.

Por último, me lo dedico también a mí mismo. Por luchar hasta el final, dando siempre lo mejor, aún cuando las fuerzas brillaban por su ausencia y la meta parecía alejarse cada vez más.

Por vosotros y por mí brindo, porque al fin y al cabo estamos aquí para ser felices, ¿no?

Resumen

Este trabajo consiste en el desarrollo de programas necesarios para hacer posible la navegación autónoma de un robot móvil terrestre como lo es el modelo Rosbot de Husarion. Ha sido realizado con la plataforma ROS (Robot Operating System), ampliamente usada en labores de robótica autónoma y automatizada.

Al ser este proyecto realizado en conjunto con un equipo de estudiantes y doctorados en la Escuela Técnica Superior de Ingeniería, los resultados obtenidos en simulaciones habrían sido comprobados físicamente en los modelos previamente mencionados, para dotar a este trabajo de mayor realidad y aplicación en cuanto a resultados se refiere. Sin embargo, debido a la situación vivida durante el desarrollo de este proyecto, se ha reducido a la simulación por ordenador.

En éste, se abarca la creación de un mapa para que el robot navegue autónomamente por él, la prueba de diferentes algoritmos y la navegación con tres robots simultáneamente. Además, se elabora una solución para poder trabajar con distancias en metros usando el GPS.

Abstract

This Project consists of the development of the necessary codes to achieve autonomous navigation for mobile robots, as Husarion Rosbot model is. It has been carried out with the platform ROS (Robot Operating System), widely used in autonomous and automatic robotics.

Due to being a project made with a group of students and doctoral students in the Escuela Técnica Superior de Ingeniería, the achieved results would have been tested physically in the aforementioned model, to give this project a wider reality and application with regard to results. However, due to the situation we have all been immersed during the realization of this project, it has been reduced to PC simulation.

In this case, it covers the creation of a map so that the robot can navigate autonomously in it, to try different algorithms and the simultaneous navigation with three robots. In addition, a solution is created to be able to work with distances in meters using the GPS which would be provided by the Husarion Rosbot model.

Índice

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Para conocer el origen del término robot es necesario remontarse a principios del siglo XX, cuando fue usado por primera vez por el checo Karel Čapek, cuya razón de ser reside en el vocablo 'robota', que en su idioma significa "esclavo". Fue usado por el autor en su obra teatral de ciencia ficción "R.U.R. (Robots Universales Rossum)", escrita en 1920, representada en Praga en 1921 y posteriormente en Nueva York en 1922. En ella se cuenta la historia de unos seres artificiales creados con semejanza al ser humano, cuya finalidad es la de trabajar ejerciendo las tareas más duras. Sin embargo, dado el momento las máquinas se rebelan y comienza una insurrección contra sus creadores que acabará dando lugar al fin de la humanidad, como se cuenta en [1].

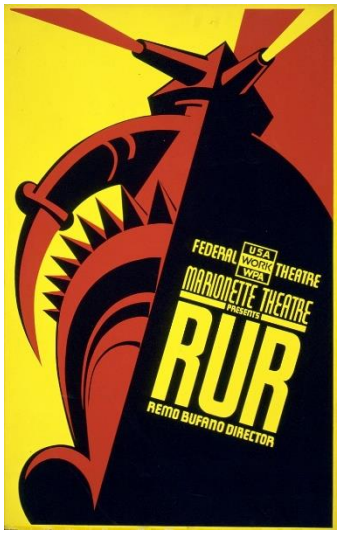

Figura 1.1 Póster de la obra de teatro R.U.R (Robots Universales Rossum). Obtenida de [1].

Esta idea de la que tanto han bebido clásicos imperecederos del cine como lo son la archiconocida Blade Runner (Ridley Scott, 1982) o la más contemporánea a su tiempo, Metrópolis (Fritz Lang, 1927), es, sin duda alguna, una temática que la ciencia ficción ha explotado casi en su totalidad, aportando cientos de visiones diferentes sobre la relación entre humanos y robots.

Sin embargo, a pesar de atribuir al autor checo el origen y acuñación del término 'robot', el verdadero padre del significado que hoy se le otorga a dicha palabra es Isaac Asimov, el cual con su obra exploró los entresijos del resurgir de las máquinas contra las personas, así como la búsqueda del ser y del comprender el objetivo de su existencia. Además, acuñó el término 'robótica', usado ahora para describir dicho campo de estudio, enumerando a su vez las conocidas Tres leyes de la Robótica [2], las cuales indican que:

- Un robot no puede dañar a un ser humano ni permitir que este lo sufra.
- Un robot debe acatar siempre las órdenes de una persona a no ser que esto entre en conflicto con la primera ley
- Un robot siempre debe asegurar su existencia a no ser que dicho acto entre en conflicto con la primera o la segunda ley.

Asimov cuenta con multitud de obras, algunas de ellas con adaptaciones al medio cinematográfico como "I, Robot" (Alex Royas, 2004) o "Bicentennial Man" (Chris Columbus, 1999). Sin embargo, su mayor legado se encuentra en la serie de relatos de la saga Fundación, en la que presenta su visión del futuro donde los robots son el eje principal.

No es, por lo tanto, descabellado afirmar que la visión que la literatura y el cine ha ofrecido sobre el futuro ha sido en una mayoría pesimista. Un futuro donde los robots se sublevan, adquieren conciencia de sí mismos y se levantan en contra de los seres humanos. Algo que, a priori, es un futuro muy lejano y poco probable dadas las tecnologías de hoy en día. De hecho, la movilidad que los robots presentan en la ficción es, en muchos casos, una utopía que suena inalcanzable a día de hoy.

Es más, desde que el humano ha teorizado con la existencia de unos seres artificiales creados para aliviar la carga de trabajo que este tiene, siempre se lo ha imaginado con forma semejante a las personas y con una movilidad prácticamente calcada a la nuestra. Nada más lejos de la realidad. Para conocer el primer robot programable y dirigido de forma digital es necesario remontarse a 1961. Creado por el estadounidense George Devol, el "Unimate" fue instalado para levantar piezas de metal caliente de una máquina de tinte y reemplazarlas.

Como se puede observar en la Figura 1.2, se encuentra muy lejos de lo que Čapek imaginó unos cuarenta años antes.

Figura 1.2. Unimate, primer robot programable (1961). Obtenida de [29].

La autonomía de la que hacen gala los robots y androides nacidos en la ciencia ficción, fruto de la imaginación de la pluma de sus creadores, era y sigue siendo algo prácticamente inalcanzable. En primer lugar, un robot autónomo es aquel capaz de trabajar con un alto grado de autonomía. Esto es principalmente deseable en campos como la exploración del espacio, limpieza, etc. En general, el mismo objetivo de siempre: aliviar al ser humano de la carga de los trabajos más duros.

Es importante destacar que existen robots industriales que son parcialmente autónomos dentro de su entorno de trabajo. Siguen requiriendo de órdenes de los trabajadores, como lo pueden ser los lugares a los que se debe desplazar y/o las acciones a realizar.

No es tarea sencilla y se le suma el hándicap de la diferencia entre los medios en los que puede trabajar el robot, ya pueda ser bajo agua, bajo tierra, en tierra, en el aire o en el espacio. Para ser autónomo, además, debe recibir información sobre el medio que le rodea, trabajar durante un tiempo sin atención de nadie, desplazarse completa o parcialmente por su entorno de acción, evitar situaciones perjudiciales para las personas y finalmente ser capaz de adquirir nuevos conocimientos, como se explica en [3].

Viendo la alta cualificación de las funcionalidades que un robot autónomo debe poseer, ¿cuánto se ha de avanzar en el tiempo para encontrar robots independientes de la acción humana? Hay quien cree que no tanto gracias a avances destacables como el aspirador robótico "Roomba" (véase Figura 1.3) de la compañía "iRobot", quienes en 2014 ya habían vendido más de 10 millones de unidades, tan sólo doce años después de su salida al mercado, capaz de auto-cargarse y desplazarse por toda la casa ejerciendo su función de robot de limpieza o los creados por la compañía Boston Dynamics [4], cuyos modelos van desde un robot con forma de perro capaz de correr imitando a uno de estos, abrir puertas y comunicarse con otros para cooperar hasta un robot con forma humana capaz de levantarse tras una caída. Además, existe la compañía Tesla, Inc. [5], liderada por el sudafricano Elon Musk, empresa líder en la creación de automóviles eléctricos (véase la Figura 1.4), cuya funcionalidad de desplazamiento autónomo es uno de sus aspectos más revolucionarios. Sin embargo, para observar un verdadero cambio respecto al desplazamiento autónomo de vehículos queda todavía mucho tiempo por delante.

Figura 1.3. Roomba modelo 604. Obtenida de [28].

Figura 1.4. Tesla Cybertruck, pick-up de más de 800km de autonomía. Obtenida de [30].

La verdadera cuestión que plantea esto es, ¿cómo es posible que un robot sea capaz de moverse de manera independiente, reconociendo el entorno, localizándose correctamente en él y llevando a cabo las tareas encargadas de manera que no ocurra nada perjudicial para aquello que le rodea ni para sí mismo?

En primer lugar, es necesario conocer los sensores de localización, aquellos que actúan como ojos del robot, ayudándole a situarse y a detectar posibles obstáculos.

1.1 Sensores de localización

Como se ha observado, los robots móviles se crean con una amplia variedad de aplicaciones posibles, con multitud de posibles objetivos y de muchas formas diversas. Además, todos presentan una característica común: deben navegar por un entorno definido, evitando los obstáculos que se le presenten, hasta alcanzar un punto de destino en el que probablemente habrá de llevar a cabo una acción, individual o conjunta junto a otros robots.

Para poder llevar esto a cabo se elabora un control que maneje adecuadamente la velocidad de las ruedas que permita al robot generar los movimientos adecuados. Esto, por supuesto, no sería posible sin la ayuda de distintos sensores que permiten al robot determinar su estado y ejercer su navegación autónoma. Como se explica en [6], lo hará respecto a un sistema de referencia global y con el mayor nivel de precisión posible. Sin embargo, como ha sido previamente mencionado, la imperfección de los sensores, siendo no lineales y con ruido, se presentan como el mayor problema para la navegación independiente.

Existe un amplio abanico de posibilidades para conocer la localización del robot. Por ejemplo, dada una posición conocida inicial, la posición actual se puede calcular mediante la estimación del giro de sus ruedas, conociendo así el desplazamiento de este. A esta técnica de estimación se le conoce como odometría, y estima la posición y orientación actuales del robot a partir de la posición y orientación en el instante anterior. Véase la siguiente fórmula (1.1), obtenida de [7]:

$$L = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + T_s * \begin{bmatrix} v_{k-1} * \cos(\theta_{k-1}) \\ v_{k-1} * \sin(\theta_{k-1}) \\ \omega_{k-1} \end{bmatrix} \qquad (1.1),$$

siendo el subíndice k el instante actual, $k-1$ el instante previo, T_s el tiempo de muestreo, y siendo x_k y las posiciones en los ejes x e y, θ la orientación, v la velocidad lineal y ω la velocidad angular. El gran

inconveniente de este método es que su error es acumulativo. Por lo tanto, tras recorrer una determinada distancia, probablemente difieran los valores estimados de los reales.
Mientras que, por otro lado, si se desconoce la posición y orientación iniciales pero el robot dispone de un sensor (ya sea escáner, ultrasonidos, etc.) o una cámara (que no deja de ser un sensor de mayor complejidad) que permita determinar su posición relativa a otros objetos presentes en la zona, entonces podrá crear un mapa del entorno en el que se encuentra y determinar su posición en el mapa, realizando de manera simultánea la localización y el mapeado. Esta técnica, llamada SLAM (Simultaneous Localization and Mapping), usada en la Sección 2 para la elaboración del mapa donde se moverá el robot, permite fusionar los sensores de movimiento y los de entorno (escáner, ultrasonidos, cámara, entre otros). Además, en el caso de que el robot ya disponga de un mapa del entorno, esta misma técnica se puede usar para determinar la posición del robot en el mapa, de manera que éste se puede localizar a sí mismo en él. Sin embargo, la alta carga computacional que requiere el procesamiento de los sensores de entorno, así como de los sistemas de visión artificial (entre otros) hace que su implementación en tiempo real no sea tarea sencilla.

Otra de las posibles maneras para localizar al robot sería el uso de un sensor complejo como un sistema de posicionamiento global. De los más usados es importante destacar el sensor GPS, el cual permitiría que no creciera el error de posición de manera progresiva. Sin embargo, esta técnica conocida como estimación de posición global acarrea consigo el problema de tener un tiempo de respuesta elevado (debido, entre otras razones, al mayor tiempo de muestreo), además de la limitación del entorno: para entornos cerrados (cámara) y para abiertos (GPS).

Ya que la localización del robot es una parte indispensable para hacer posible la navegación autónoma y los tiempos de respuesta elevados pueden dar lugar a la desestabilización del sistema, en general, se usa como método localizador principal la estimación local de la posición mientras que el sistema de posicionamiento global se usa para corregir dicha estimación local. Esto quiere decir que se usaría, por ejemplo, los datos del sensor GPS para determinar de manera exacta una distancia recorrida respecto a un punto de referencia y así corregir el error acumulado por la odometría. De esta manera, el procedimiento se asegura una respuesta rápida gracias al bajo consumo de recursos de la estimación local y obtiene la precisión de un sistema de localización global sin perder eficacia con los largos tiempos de respuesta. Ésta y otras combinaciones se llevan a cabo mediante un sistema de fusión sensorial basado en el uso de estimadores, como pueden ser el filtro de Karman lineal (KF) o el extendido (EKF), entre otros. Este proceso toma un modelo del robot para predecir su comportamiento, tomando la precisión de cada sensor y la estimación de dicho modelo, realizando la fusión de todos estos elementos y generando una predicción lo más óptima posible para mejorar así la estimación de la posición del robot.

La importancia de una correcta estimación de la localización reside en que ésta es usada por el algoritmo de navegación o seguimiento de trayectorias, el cual controla las velocidades de las ruedas del robot. En el caso del Rosbot de Husarion, lo hace mediante un control PID que, dada una referencia de velocidad para cada rueda, regula la velocidad de los motores.

En general se hace uso del control por punto descentralizado para trayectorias sin obstáculos, cuyo diagrama de bloques puede observarse en la Figura 1.5. Su funcionamiento se basa en mover el robot siguiendo una trayectoria definida. Este control se establece partiendo de la posición y velocidad de un punto dado, situado a una distancia concreta del eje de tracción del robot. Por lo tanto, gira con antelación para que al terminar este se encuentre en línea con la trayectoria deseada.

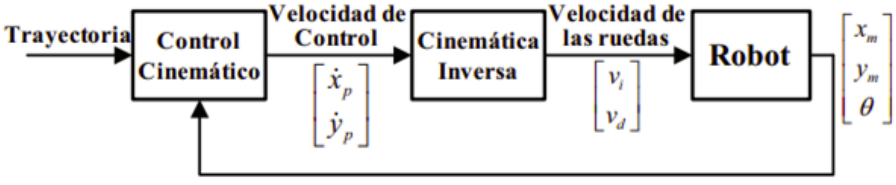

Figura 1.5. Esquema de control por punto descentralizado. Obtenida de [6].

Este sistema básico de navegación requiere una alta precisión a la hora de estimar la localización del robot. Para ello se busca un método de fusión que tenga poca incertidumbre en la estimación, que por lo general son esquemas complejos basados en filtros y modelos no lineales. Esto quiere decir que necesitan un tiempo de ejecución considerable debido a la complejidad de los cálculos que lleva a cabo. Por lo tanto, si el robot que se ha de localizar cuenta con capacidad computacional limitada, muchos de estos métodos no podrán ser implementados. No hay que olvidar, además, que la localización, a pesar de ser algo totalmente esencial para la navegación autónoma del robot, no es la única tarea que éste debe llevar a cabo, estando entre ellas las comunicaciones, la administración de los sensores y el control. Por lo tanto, teniendo estos factores en cuenta, un buen algoritmo de localización ha de ser lo más eficiente posible en cuanto a recursos utilizados y debe también incorporar la información de todos los sensores disponibles mediante el método de fusión.

1.2 Métodos de localización

En esta Sección se verán tres técnicas de localización, mediante marcas, mediante ultrasonidos y finalmente mediante filtros no lineales evolutivos.

1.2.1 Localización mediante marcas

Una de las opciones tomadas para la estimación de la localización del robot se basa en el uso de marcas. La idea es otorgar al robot la información que necesita para corregir su posición y orientación a través de las diversas marcas empleadas. Dicha información está ligada a los ángulos *pan* y *tilt* así como a la distancia a la marca. Sin embargo, para que el robot conozca las marcas y dónde se sitúan se necesita de la existencia de un mapa donde se guarde la posición de cada marca respecto al sistema de referencia absoluto que se use, para que así el robot pueda calcular su posición relativa respecto a esas marcas y su posición absoluta respecto al sistema de referencia, como se explica en [7].

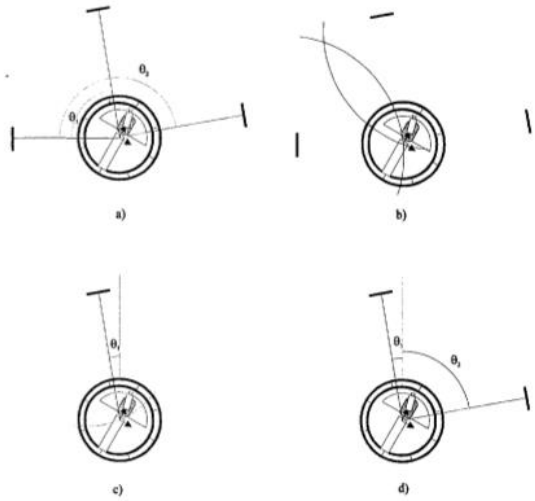

Figura 5-1. Relocalización mediante marcas.

a) ángulos respecto a dos o más marcas, b) distancia respecto a dos o más marcas, c) ángulo y distancia respecto a una marca, d) ángulos relativos entre marcas.

Figura 1.6. Localización mediante marcas. Obtenida de [7].

Las diferentes posibilidades para relocalizar el robot mediante triangulación se pueden observar en la Figura 1.6 y son la medición de: ángulos o distancia respecto a dos o más marcas (véase Figura 1.6 a) y Figura 1.6 b) respectivamente), ángulo y distancia respecto a una marca (véase Figura 1.6 c)) y ángulos relativos entre marcas (Figura 1.6 d)).

Además de conocer la ubicación de las marcas, se han de conocer también los parámetros de calibración del sistema de visión utilizado, así como los movimientos de la cámara que se necesitan para tomar la imagen.

Un ejemplo podría ser el entorno de una oficina, donde un robot se guía y se localiza mediante los carteles rectangulares con los nombres de los empleados, asociándose cada uno a una posición concreta del lugar.

1.2.2 Localización mediante ultrasonidos

Esta otra alternativa es usada en numerosos robots móviles para conocer su posición y orientación en un entorno concreto. Es, además, un componente de bajo coste que hace que sea muy utilizado por estudiantes para sus proyectos, permitiendo la detección de obstáculos, medición de distancias, etc. Además de su precio, gozan de una alta velocidad de procesamiento, así como una precisión aceptable. No obstante, como se explica en [7], localizar un robot sólo con la información de la distancia a objetos tiene un indudable inconveniente: la gran cantidad de errores que estos cometen en sus mediciones.

Como se detalla en [8], un sensor de ultrasonidos mide la distancia mediante el uso de ondas ultrasónicas. El cabezal emite una onda ultrasónica y recibe la onda que se refleja y retorna desde el objeto sobre el que se proyecta dicha onda, como se muestra en la Figura 1.7. Estos calculan la distancia al objeto haciendo uso del tiempo transcurrido entre la emisión y la recepción, conociendo la velocidad de la onda.

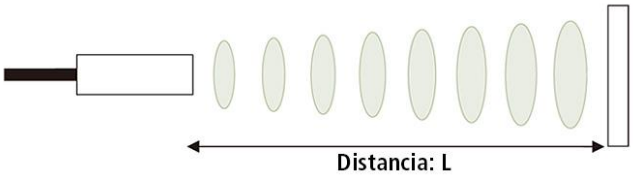

Figura 1.7. Esquema de un sensor de ultrasonidos. Obtenida de [8].

1.2.3 Localización mediante filtros no lineales evolutivos

El filtro de Kalman puede utilizarse para sistemas no lineales (aunque fuese principalmente creado para sistemas lineales), realizando una aproximación lineal del modelo de medida del sistema en el entorno de una estimación del estado, utilizando los primeros términos del desarrollo en serie de Taylor, creando así, como se profundiza en [7], el filtro de Kalman extendido.

Debido a los errores introducidos causados por el rechazo de los términos de orden superior, la aproximación lineal realizada en el filtro de Kalman extendido tiene dos principales limitaciones:

1. se trata de un estimador cuyos valores estimados obtenidos son aproximaciones de la media condicional y la covarianza del error que corresponderían a la estimación óptima de mínima varianza.
2. cuando se tienen sistemas no lineales, puede haber óptimos locales distintos del óptimo global. La convergencia al óptimo global depende de la elección de una estimación inicial adecuada como semilla para el proceso de linealización.

1.3 Navegación autónoma de robots móviles

El procesamiento de visión artificial requiere de una carga potencial enorme. A pesar de ello, el desarrollo de sistemas de navegación en tiempo real basadas en visión artificial para robots móviles se está convirtiendo en una realidad cada vez más palpable con el avance de los años.

Los otros sensores usados para navegación son los sensores infrarrojos, sensores sónar, localizadores láser, sensores sensibles a la posición (PSD) y sensores inerciales. En el caso de los infrarrojos tienen un uso limitado, un comportamiento no lineal y la reflectancia depende en la superficie del objetivo. Por otro lado, los sensores sónar presentan unos datos fáciles de leer y computacionalmente son asequibles, sin embargo, la fiabilidad de sus datos es baja debido a las perturbaciones del entorno. En cuanto a los localizadores láser en [9], se realiza una comparativa donde éstos presentan mayor fiabilidad, medida instantánea, precisión a mayor rango y mejor resolución angular que el sónar. Todo ello, como era fácil vaticinar, a un precio mucho más alto.

Por otro lado, un sistema de visión artificial es considerado como un sensor pasivo y tiene ventajas sobre los sensores activos. Los sensores pasivos no alteran el entorno emitiendo luces u ondas para obtener datos. Además, las imágenes contienen mucha más información, las cámaras son baratas y por lo tanto la visión artificial como mecanismo sensorial para robots móviles supone una solución atractiva.

1.3.1 Navegación basada en visión artificial

Se define como la técnica para guiar un robot móvil a una posición deseada a lo largo de una trayectoria en un entorno, evitando obstáculos estáticos (e incluso dinámicos) usando visión artificial. Consta de cinco elementos principales:

1. Mapas.
2. Adquisición de datos.
3. Extracción de características (como bordes, color y textura).
4. Reconocimiento de puntos de referencia.
5. Auto localización. Calcula la posición actual del robot en función de los puntos de referencia detectados, así como de su posición previa.

No obstante, no es tarea fácil, puesto que presenta cuatro principales problemas:

1. Percepción del entorno.
2. Planificación de la ruta.
3. Generación de la ruta.
4. Seguimiento de la ruta.

El objetivo principal reside en localizar obstáculos (tanto estacionarios como móviles). Conociendo esto, se dividirá la navegación en dos grupos en función del lugar: navegación en interiores y en exteriores.

1.3.1.1 Navegación en interiores basada en visión artificial

Puede ser dividida en tres subgrupos: navegación basada en un mapa predeterminado, navegación basada en la construcción de un mapa y navegación sin mapa.

1. Navegación basada en un mapa predeterminado

En este método, el sistema tiene un conocimiento previo sobre el entorno y el sistema de navegación trabaja conociendo el mapa. Un método usado es el de “localización incremental”. En este se asume que un conocimiento aproximado sobre la localización del robot está disponible y es perfeccionado durante el proceso de navegación.

2. Navegación basdad en la construcción de un mapa

Los procedimientos de este método tratan de solucionar el reto de comenzar sin conocimiento previo sobre el entorno, mediante la exploración y construcción de una descripción interna para así proceder con la navegación.

3. Navegación sin mapa

En este caso, el procedimiento se desarrolla en su totalidad con desconocimiento de mapa alguno. La información relevante sobre el entorno es extraída mediante observación, puntos de referencia, etc., para que el algoritmo de navegación tome una decisión en base a dicha información extraída. Para el reconocimiento del entorno existen dos maneras:

- Aproximaciones basadas en modelos que utilizan modelos de objetos previamente definidos para reconocer sus características en entornos complejos y autolocalizarse en ellos.
- Aproximaciones basadas en visión artificial donde no se tiene información proveniente de imágenes previamente tomadas. La autolocalización se lleva a cabo usando algoritmos de coincidencia de imágenes.

1.3.1.2 Navegación en exteriores basada en visión artificial

La navegación en exteriores puede ser dividida en dos subgrupos:

- Navegación en entornos estructurados
- Navegación en entornos desestructurados

1.4 Evitación de obstáculos en navegación autónoma

La evitación de obstáculos es el proceso de diferenciar entre el suelo (el plano de tierra) y un objeto residiendo en él. Evitar colisiones permite a un robot navegar en un entorno sin chocar, y por eso es uno de los papeles más importantes en la navegación de robots móviles.

Para llevar esto a cabo existen multitud de posibilidades: sensores convencionales (láser, sónar, infrarrojos, etc.), así como métodos de detección basados en visión en 3D, etc.

Otra estrategia para la evitación de obstáculos es un método basado en la apariencia, para entornos estructurados. Este sistema se basa en tres suposiciones: los obstáculos difieren en aspectos respecto al suelo, la tierra es relativamente plana y todos los obstáculos deberían estar en contacto con el suelo. El sistema, además, consta de tres modos de operación: regular, adaptativo y asistivo.

Existen diferentes alternativas, explicadas en [9], donde otro método sería usando un sónar visual, donde una sola cámara está montada en el robot y cada píxel de la cámara es clasificada en píxeles de suelo, otros objetos conocidos o desconocidos, basado en los diferentes colores. Un objeto es identificado si existe un conjunto continuo de píxeles en un escaneo que corresponde al mismo color. Un objeto desconocido es detectado cuando un conjunto de colores desconocido es detectado. Además, un mapa local es creado con la distancia de los objetos al robot. Este algoritmo de visión es implementado en el robot AIBO de Sony. Otro método consiste en la combinación de una sola cámara y un sensor untrasónico. La detección de obstáculos es llevada a cabo usando un detector de bordes (canny Edge detector).

En el método basado en visión estéreo, la idea principal reside en capturar dos imágenes del entorno a la vez. La posición de un obstáculo puede ser determinada por "inverse perspective mapping". El problema de este método es que es esencialmente un procedimiento computacionalmente complejo.

1.5 Navegación autónoma de múltiples robots

El trabajo llevado a cabo en [10] presenta el esquema que se puede observar en la Figura 1.8.

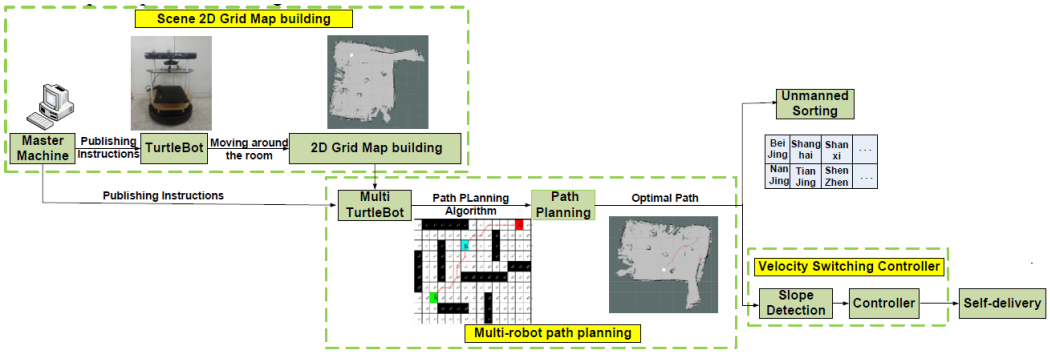

Figura 1.8. Esquema de navegación autónoma para múltiples robots. Obtenida de [10].

Como se puede observar, el mapa del entorno es una información que todos los robots comparten y es imprescindible para llevar a cabo la navegación con éxito. Se publican desde el PC las instrucciones a los diferentes robots y estos a su vez usan el algoritmo de trazamiento de trayectorias para encontrar un camino que les permita alcanzar su punto de destino. En este esquema, además, se presenta un "Velocity Switching Controller", que permite al robot modificar su velocidad si detecta que el terreno está inclinado, para poder superar dicha rampa con éxito.

1.6 Introducción a ROS

Como se ha mencionado en el resumen, el proyecto será llevado a cabo con ROS (Robot Operating System), especialmente con el modelo Rosbot de la compañía Husarion, que además aporta paquetes que permiten llevar a cabo el desarrollo de este proyecto con la simulación de su robot. Sin embargo, ¿qué es ROS y por qué no otra plataforma?

ROS es, como bien se indica en [11], a fin de cuentas, un framework que permite al usuario desarrollar software para robots. Su funcionamiento está basado en una arquitectura de grafos, de manera que el procesamiento radica en la comunicación entre nodos mediante la recepción, el envío y la multiplexación de mensajes provenientes de mútiples fuentes como lo pueden ser sensores, control, actuadores, etc.

Consta, como se detalla en [11], además, de dos partes básicas: el sistema operativo (ros) y ros-pkg, un cojunto de paquetes aportados por los usuarios que permiten implementar funcionalidades como localización, mapeo, navegación, etc. Una de sus características más rompedoras radica en su misma arquitectura: al funcionar mediante la recepción y el envío de mensajes, lo importante es el mensaje en sí y no cómo se genera. ¿Qué quiere decir esto? Que los nodos creados pueden estar en dos lenguajes diferentes de programación como lo son Python y C++. Esto permite que la comunidad se amplíe doblemente, goce de mayor alcance y por tanto crezca con mayor facilidad gracias a los desarrolladores.

Entre sus herramientas destacan Rviz (simulación y visualización 3D para robots), "catkin" (herramienta para compilar los nodos), "roslaunch" (herramienta que se usa para ejecutar múltiples nodos ROS de manera conjunta) y "rosbash", una herramienta que permite trabajar con los nombres de los paquetes de ROS, lo cual facilita al usuario el trabajo de manera muy significativa.

Además, sus elementos más importantes son los nodos y los "topics". Los nodos no son más que códigos ejecutables que generan y envían mensajes, los reciben o ambas a la vez. Por lo tanto, esos mensajes son publicados en "topics". Este tan sencillo funcionamiento es el pilar fundamental sobre el que se rige ROS. Además, existen servicios y clientes (así como servidores y clientes), que permiten generar peticiones sobre un "topic" concreto (en el caso de los servidores), así como de enviar un mensaje a un "topic" concreto, como lo haría el cliente. Conociendo esto, los comandos más usados y que mayor importancia toman en este proyecto son los explicados en [12], detallados a continuación:

- "Roslaunch": ejecución de múltiples nodos de manera local.
- "Rosrun": ejecución de un solo nodo.
- "Rostopic": permite conocer información sobre los "topics" publicados, escribir sobre ellos o leer lo que se está publicando.
- "Rosmsg": permite obtener información sobre los mensajes publicados en los diferentes "topics".

2 CONSTRUCCIÓN DE UN MAPA Y LOCALIZACIÓN DEL ROBOT

Para llevar a cabo la navegación autónoma del robot se ha de tener previamente un mapa en el cual éste se pueda desplazar. Para ello, se ha realizado en Gazebo un archivo. world, compuesto por veinte bloques cuadrados que simulan las placas de una planta termosolar (salvando las distancias). Se busca así una estructura lo más simétrica posible.

2.1 Creación de un archivo .world

En primer lugar, para elaborar con éxito un mapa se ha de crear previamente un archivo .world, que consiste en un entorno en tres dimensiones, creado en Gazebo, en el cual se pueden añadir múltiples elementos, dando lugar así a un "mundo", en el cual se han colocado los bloques previamente mencionados.

Para ello, sencillamente se han añadido dichos elementos de la manera más simétrica posible y se ha guardado el archivo en el PC. Cabe destacar la importancia de este archivo pues a partir de él se creará un mapa mediante el escaneo del láser (LIDAR) del robot.

En la Figura 2.1 se puede observar cómo luciría dicho archivo, visualizado en el simulador Gazebo.

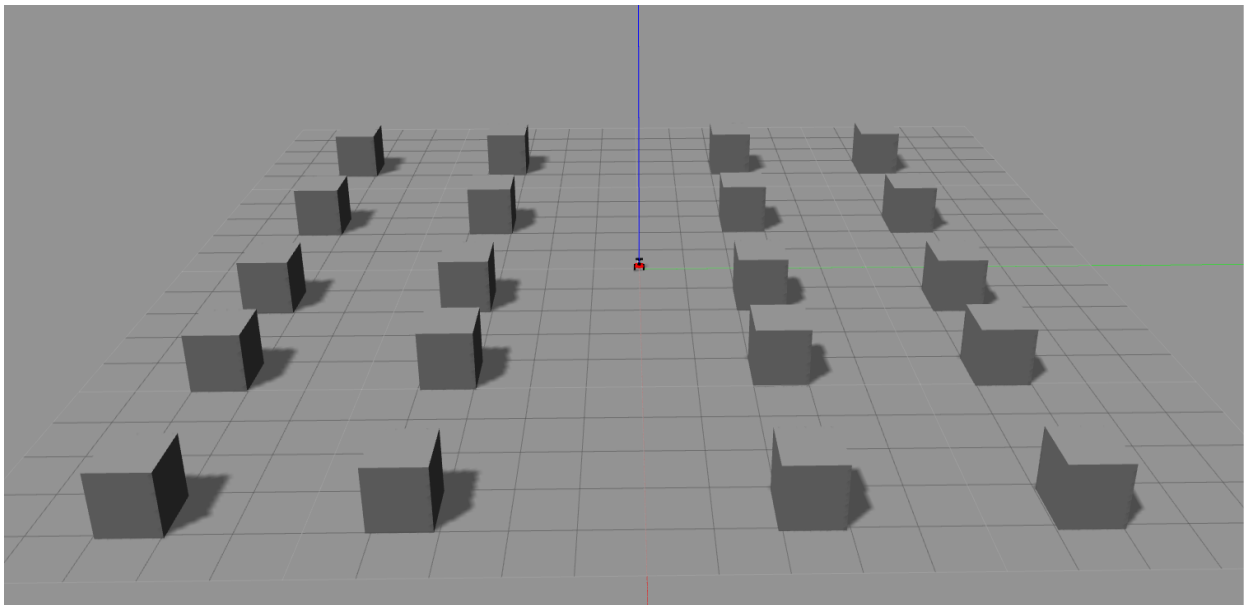

Figura 2.1. Archivo .world visualizado en Gazebo (con robot)

Para contemplar el robot junto al .world bastaría con lanzar el archivo "rosbot.launch", perteneciente al paquete "rosbot_description". Este código se encuentra en el Anexo A. Como se puede observar, ejecuta a su vez dos archivos .launch, que son los siguientes:

- **"rosbot_gazebo.launch"**. Situado en el Anexo B, en él se establece el tipo de robot que se va a usar, además de los parámetros que se le atribuyen. Para este proyecto, como no podía ser de otra manera, se elige el modelo Rosbot.

- **“rosbot_world.launch”.** Situado en el Anexo C. En éste, en cambio, se detallan los parámetros del archivo .world donde el robot se va a desplazar. Como se puede observar, uno de los argumentos es el archivo llamado “turtlebot_playground.world”, el cual es uno de los archivos de ejemplo que incluyen los tutoriales de Turtlebot para ROS.

Una vez ejecutado el archivo “rosbot.launch”, que contiene a los dos anteriores, se tiene el robot en el archivo .world mencionado previamente, listo para ser desplazado. Por lo tanto, lo único necesario para usar el archivo .world creado para el proyecto en vez del mencionado antes sería cambiar el nombre del archivo .world usado en el código “rosbot_world.launch” (Anexo C), que por defecto es ”turtlebot_playground.world”.

2.2 Creación de un mapa a partir de un archivo .world con el paquete “slam_gmapping”

El proceso llevado a cabo se corresponde con un fiel seguimiento al vídeo [13], donde para crear el mapa se usa SLAM (Simultaneous Localization and Mapping – Localización y mapeado simultáneos), que es una técnica usada en robótica para crear un mapa de un entorno concreto además de localizar al propio robot dentro de éste. Uno de los mayores problemas que debe afrontar es la inevitable incertidumbre inherente a la imperfección de los sensores encargados de analizar el entorno que rodea al propio robot, como se profundiza en [14].

Mientras que el robot se mueve, las medidas tomadas y la localización cambian. Por lo tanto, para crear el mapa es necesario incorporar las medidas tomadas en las posiciones anteriores. En este caso, ROS permite mantener en el tiempo el rastreo de los marcos de coordenadas. En [14] se explica detalladamente cómo para conseguirlo se hace uso del paquete “tf2”, el cual viene con un mensaje específico del tipo “tf/Transform”, el cual a su vez consiste en la transformación (rotación y traslación) entre dos marcos de coordenadas, los nombres de ambos marcos y la marca de tiempo.

El paquete “slam_gmapping”, por lo tanto, hará uso del escáner del robot (LIDAR - Laser Imaging Detection and Ranging) y de odometría, transformando las medidas tomadas por el escáner al marco tf (previamente mencionado) de la odometría. En términos concretos de ROS, el paquete “slam_gmapping” toma los mensajes “sensor_msgs/LaserScan” (provenientes del escaneo del LIDAR) y crea un mapa (“nav_msgs/OccupancyGrid”). En términos generales, se podría decir que SLAM hace uso de un filtro de partículas (PF – Particle Filter), una técnica para estimación basada en un modelo. Se estiman dos elementos: el mapa y la posición del robot en él. Se podría tomar cada partícula del filtro como una posible solución al problema. La unión de todas estas partículas aproximaría la verdadera distribución del mapa y la posición del robot dadas las entradas de control (odometría) y las lecturas del sensor (LIDAR).

El entorno a partir del cual se generará el mapa está generado en el archivo .world, por el que se puede desplazar el robot y en consecuencia crear un mapa. El proceso por realizar se explica a continuación.

En primer lugar, se crea un paquete nuevo llamado “my_mapping_launcher”, replicando lo realizado en [13], con el comando “catkin_create_pkg”. En él, dentro del directorio “launch”, se crea un archivo llamado “my_mapping.launch”, que contendrá una copia del archivo “gmapping.launch.xml”, perteneciente al paquete “turtlebot_navigation”.

Sin embargo, será necesario editar dicho archivo de manera que los paramétros coincidan con los del robot con el que se trabaja, además de que la suscripción a los topics coincida con los del modelo de Rosbot. Para ello, se realiza lo siguiente:

- Se ejecuta el archivo “rosbot.launch” perteneciente al paquete “rosbot_description”. Con este comando se abre la simulación en Gazebo del robot en el entorno creado en la Sección 2.1.
- Se ejecuta el programa Rviz mediante el comando “rosrun”. Éste a su vez es también un simulador en tres dimensiones del robot, permitiendo al usuario acceder a los “topics” publicados

y observar sus valores gráficamente. Además, será el que permita elaborar un mapa a raíz del entorno previamente creado.

Como se ha de editar el archivo "my_mapping.launch" con los correctos nombres de los "topics" del robot, se hace uso del comando "rostopic list", el cual permite observar los que se están ejecutando, y comprobar cuál pertenece a la referencia del robot (en este caso, "/base_link"), cuál pertenece a la odometría ("/odom") y cuál pertenece al escáner ("/scan").

Una vez conocidos, se edita el archivo "my_mapping.launch", de manera que quedaría como se puede observar en el Anexo D. Los valores se toman iguales a los ya establecidos, pertenecientes al paquete "turtlebot_navigation". Como se puede ver, los datos introducidos son los argumentos "scan_topic", "base_frame" y "odom_frame".

Una vez se ha modificado con éxito, el robot está preparado para generar el mapa. Para ello, se realiza lo siguiente:

- Se ejecuta el archivo "rosbot_teleop.launch", perteneciente al paquete "rosbot_navigation". Este comando permite que desde una terminal de Ubuntu se pueda desplazar el robot y modificar su velocidad (tanto lineal como angular) con el teclado. Con ello se moverá al robot a lo largo del entorno para generar el mapa deseado.

Por último,

- Se ejecuta el archivo "my_mapping.launch", perteneciente al paquete "my_mapping_launcher", el cual comienza a realizar la lectura del escáner del robot, de manera que genera un mapa que posteriormente puede ser guardado.

El procedimiento es sencillo y consiste en desplazar el robot a lo largo del entorno creado para generar el mapa. Se debe realizar un movimiento exhaustivo, comprobando en Rviz mediante el "topic" del escáner ("scan") que la lectura del entorno se está realizando con éxito. Además, se puede observar con claridad el mapa que se va creando, añadiendo el "topic" "/map" en Rviz. Véase las Figura 2.2 y 2.3.

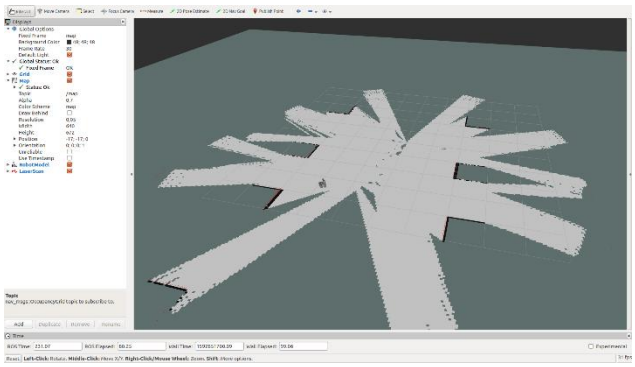

Figura 2.2. Escaneado del mapa al inicio de la simulación.

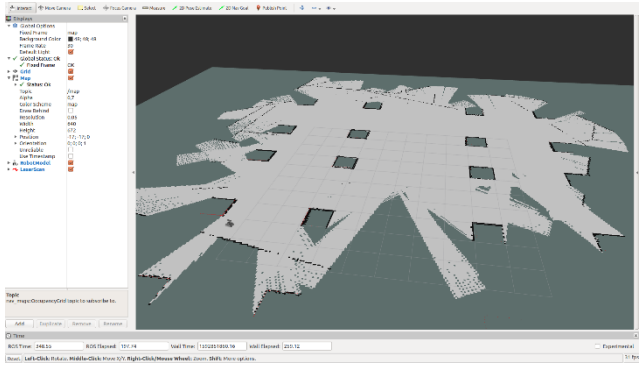

Figura 2.3. Escaneado avanzado del mapa con bloques encontrados.

Una vez se ha desplazado por él en su totalidad, se crea una carpeta en el paquete "my_mapping_launcher" de nombre "config". En ella se procede a guardar el mapa mediante la ejecución del siguiente comando:

rosrun map_server map_saver -f <map_name>

En este caso, el nombre del mapa será "cubes.map", siguiendo así la nomenclatura usada para el archivo .world.

Una vez creado, cada vez que se quiera hacer uso de éste es necesario lanzar el siguiente comando:

rosrun map_server map_server <map_name.yaml>

Cabe destacar el hecho de que, una vez creado el mapa, se emite un mensaje que indica sus medidas. En este caso, las proporciones son las siguientes: 512 x 512 píxeles, donde cada píxel representa 0.050 metros. En la Figura 2.4 puede verse una vista superior del mapa creado.

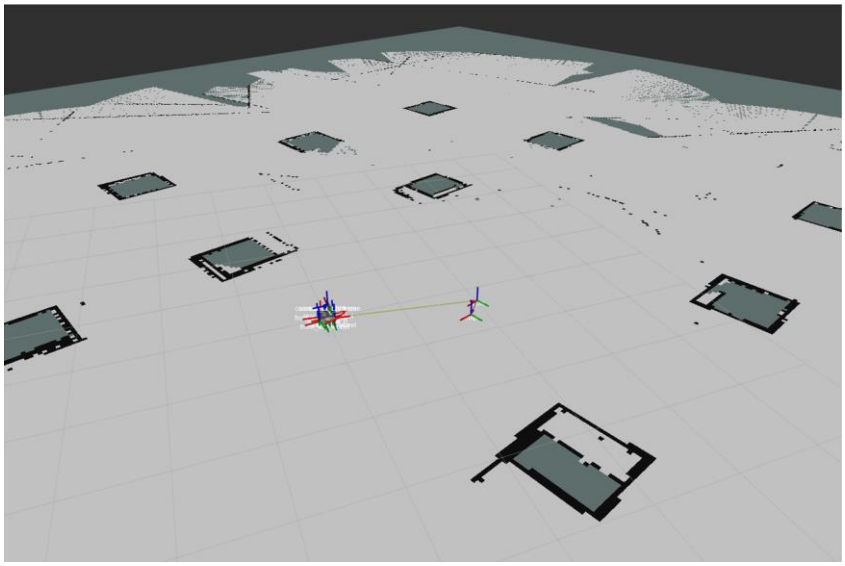

Figura 2.4. Escaneado completo del mapa.

2.3 Localización del robot en el mapa

Para localizar correctamente el robot en el mapa es necesario crear un paquete, con el mismo procedimiento con el que se ha creado el paquete "my_mapping_launcher". En este caso, el nombre será "my_localization_launcher", en el cual, dentro de su carpeta "launch", se copiará el archivo "amcl.launch", perteneciente al paquete "turtlebot_navigation". Realizando los cambios de nombres correspondientes, como se ha hecho para el archivo "my_mapping.launch", se establecen "base_link", "odom" y "scan" para "base_frame", "odom_frame" y "scan" respectivamente, como se explica en [13]. El código obtenido y modificado puede observarse en el Anexo E.

Este código, como de su nombre puede inducirse, hace uso del algoritmo AMCL, el cual, como se detalla en [15], es un sistema de localización probabilístico para un robot moviéndose en dos dimensiones. Implementa la aproximación de localización Monte Carlo, que usa un filtro de partículas para filtrar el rastreo de la posición de un robot en un mapa conocido.

Al ejecutarlo mediante el comando "roslaunch", se consigue localizar el robot en el mapa. En caso de que la estimación no sea correcta, en Rviz se puede hacer uso de la herramienta "2D Pose Estimation", añadiendo además la visualización de "PoseArray", cuyo "topic" "/particlecloud" muestra una nube de puntos concentrada en el lugar donde estima que el robot se encuentra, la cual puede verse en la Figura 2.5.

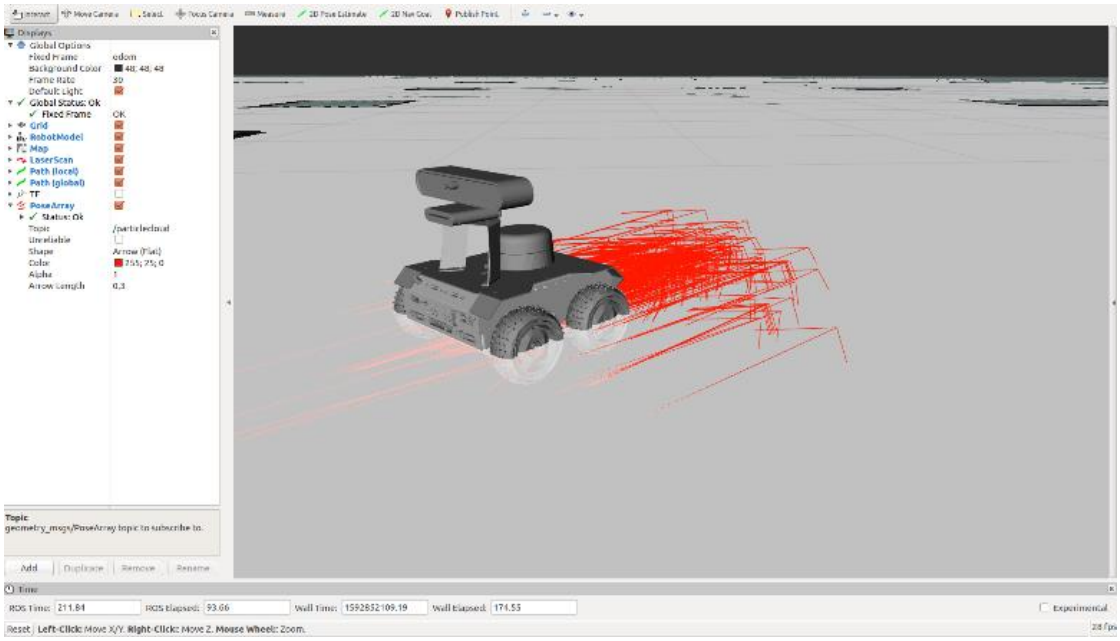

Figura 2.5. "Topic" "/particlecloud" que estima la posición del robot.

Se puede observar en el código que también ha sido añadido el "topic" "/map", de manera que el proceso tiene en cuenta el mapa en el que se desarrolla la navegación.

En la Figura 2.6 puede verse una vista superior del mapa, con el robot localizado en Rviz.

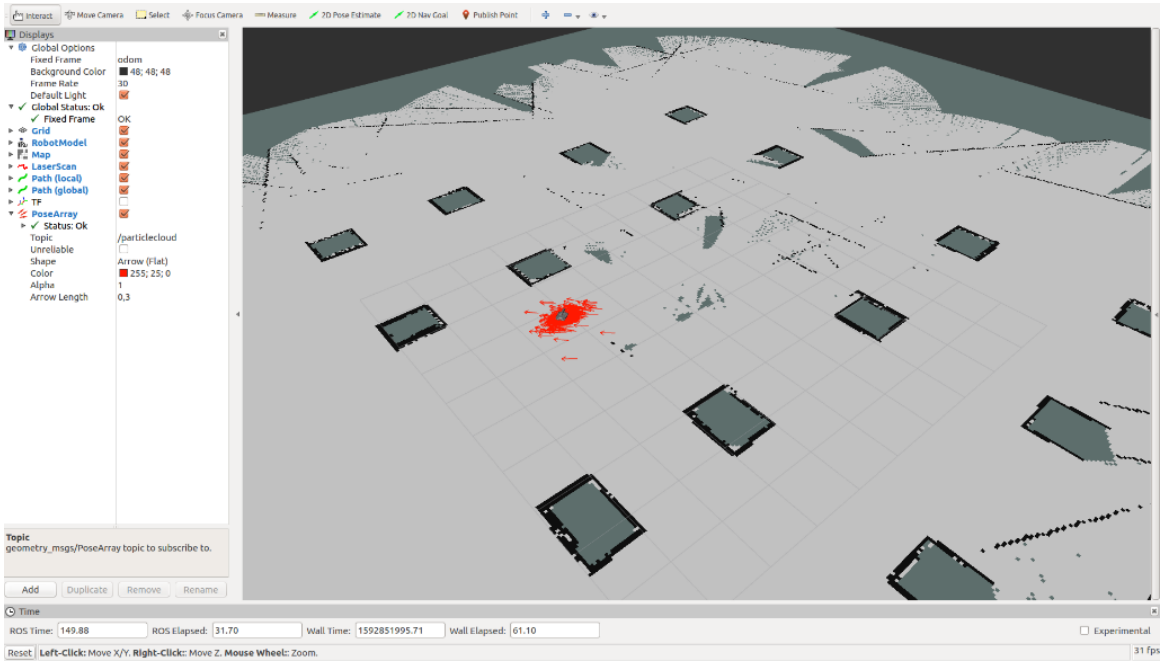

Figura 2.6. Localización del robot (Fixed Frame: odom)

3 NAVEGACIÓN AUTÓNOMA CON EL PAQUETE "MOVE_BASE"

El paquete "move_base" aporta la implementación de una acción que, dado un objetivo en el entorno, intentará alcanzarlo con una base móvil. El nodo "move_base" une un organizador global y otro local para llevar a cabo la tarea de navegación global. Además, como se explica en [16], para ello también mantiene dos mapas de coste, uno para el organizador global y otro para el local. El esquema por el que se rige su funcionamiento puede verse en la Figura 3.1.

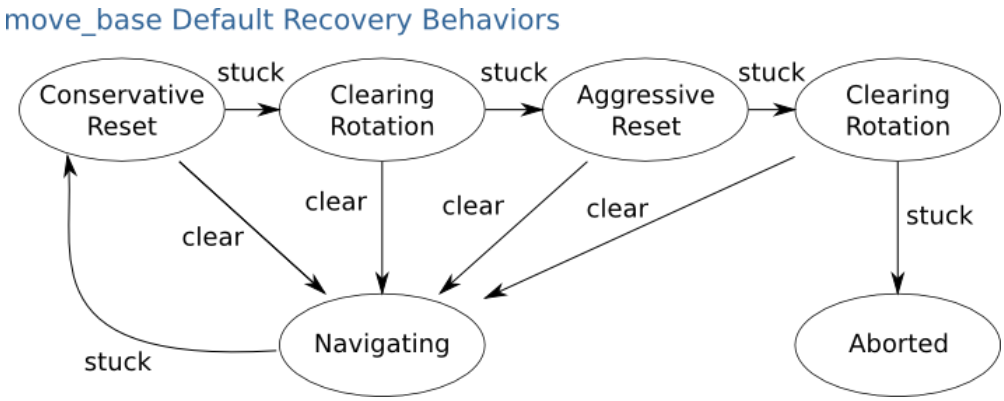

Figura 3.1. Esquema del paquete move_base. Obtenida de [16].

Ejecutar un nodo "move_base" en un robot propiamente configurado da lugar a un robot que intentará alcanzar una posición objetivo. Además, podría llevar a cabo comportamientos de recuperación si el robot percibe que se encuentra bloqueado o atascado. Por defecto, las acciones que el nodo "move_base" realizará serán las explicadas a continuación.

En primer lugar, los obstáculos fuera de la región especificada por el usuario serán eliminados del mapa del robot. Después, si es posible, el robot llevará a cabo una rotación in situ para limpiar el espacio. Si esto falla, el robot limpiará más agresivamente su mapa, eliminado obstáculos fuera de la región rectangular en la que puede rotar. A esto lo seguiría otra rotación in situ. Si todo esto falla, el robot considerará inviable su objetivo y notificará al usuario en consecuencia.

Además, el robot aporta una implementación del "SimpleActionServer", que recibe puntos objetivo, siendo "geometry_msgs/PoseStamped" el tipo de mensaje que recibe. El uso de un "ActionClient" para enviar los puntos objetivos se desarrolla en la Sección 4.3.

Una vez conocido su funcionamiento, el proceso a llevar a cabo sería similar al realizado para el mapeado del entorno y la localización del robot en él, tomando como guía fundamental lo realizado en [17].

Para empezar, se crea un paquete llamado "my_move_base", en el cual, dentro de la carpeta "launch" se creará un archivo llamado "my_move_base.launch", el cual será una copia del archivo "move_base.launch" contenido en el paquete de "turtebot_navigation". De nuevo, el procedimiento para ajustar los parámetros al robot de Husarion sería cambiar los "topics" escritos por los del modelo Rosbot. El código en cuestión puede observarse en el Anexo F.

Una vez debidamente configurado el código, existen diversas maneras de enviar posiciones objetivo al robot. Una sería mediante el comando "2D Nav Goal" de Rviz, otra mediante el comando "rostopic pub" y otra mediante un nodo que publique sobre un topic ya creado o con un "ActionClient".

3.1 Punto de destino señalado en Rviz

Una de las maneras que ROS ofrece al usuario para enviar puntos de destino al robot es mediante el comando “2D Nav Goal” de Rviz. Es un método gráfico que consiste en clicar sobre un punto en el mapa, añadiendo además la orientación final del robot en dicho punto. Se puede observar en la Figura 3.2.

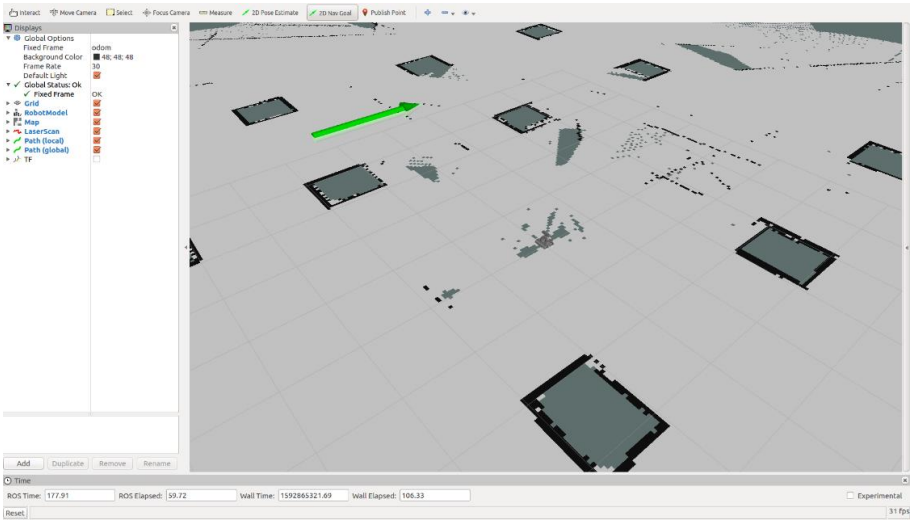

Figura 3.2. Comando “2D Nav Goal” en Rviz.

En las Figuras 4.3 a 4.8 se puede observar el trayecto realizado por el robot.

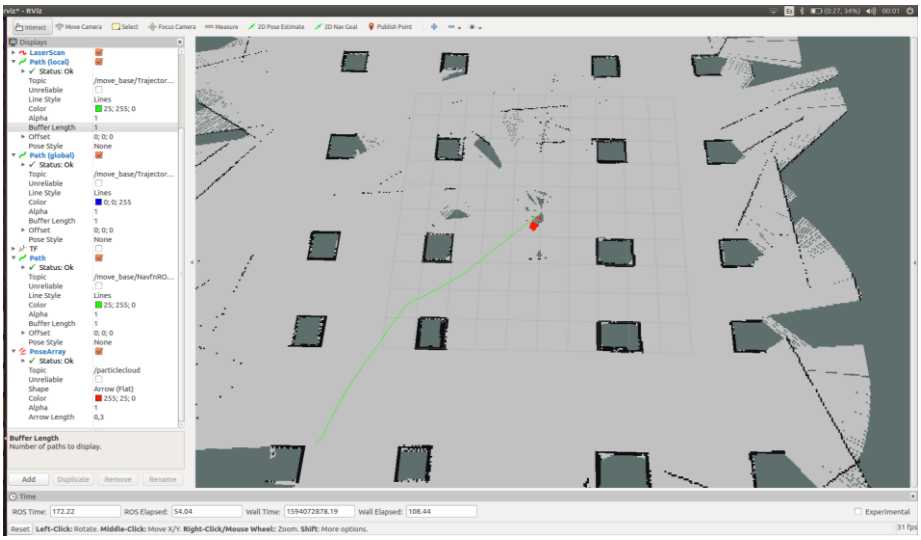

Figura 3.3. Desplazamiento del robot visualizado en Rviz (1).

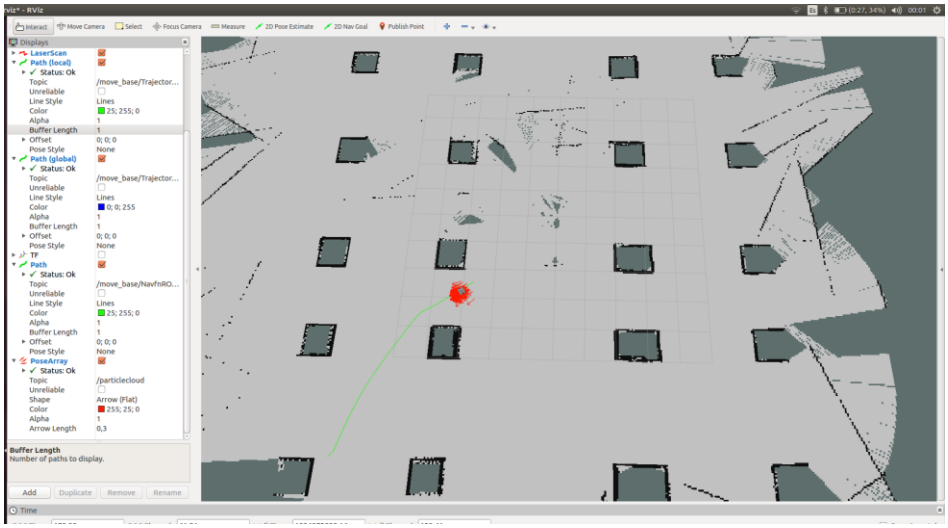

Figura 3.4. Desplazamiento del robot visualizado en Rviz (2).

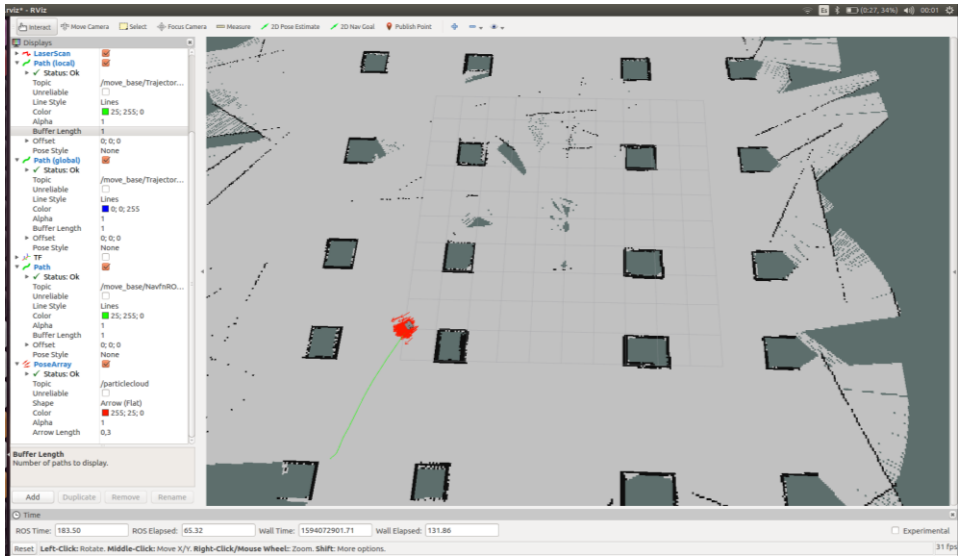

Figura 3.5. Desplazamiento del robot visualizado en Rviz (3).

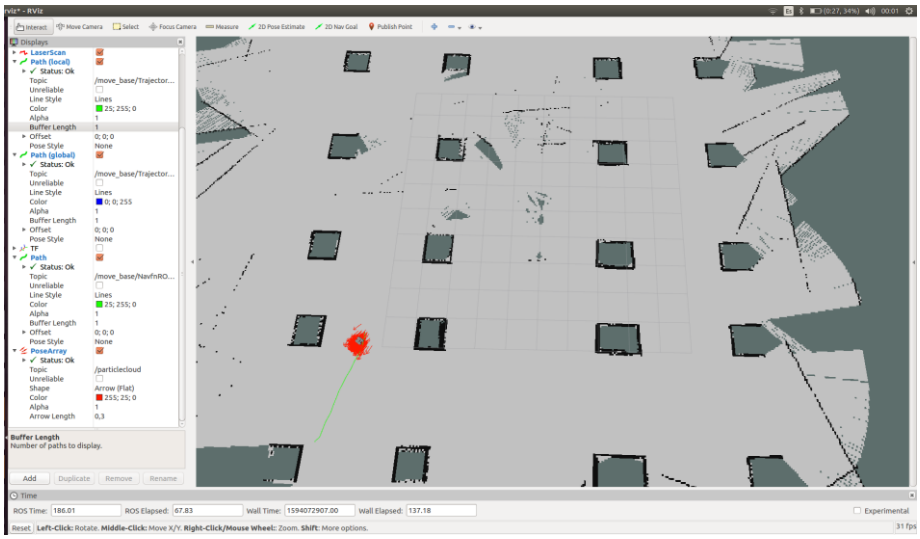

Figura 3.6. Desplazamiento del robot visualizado en Rviz (4).

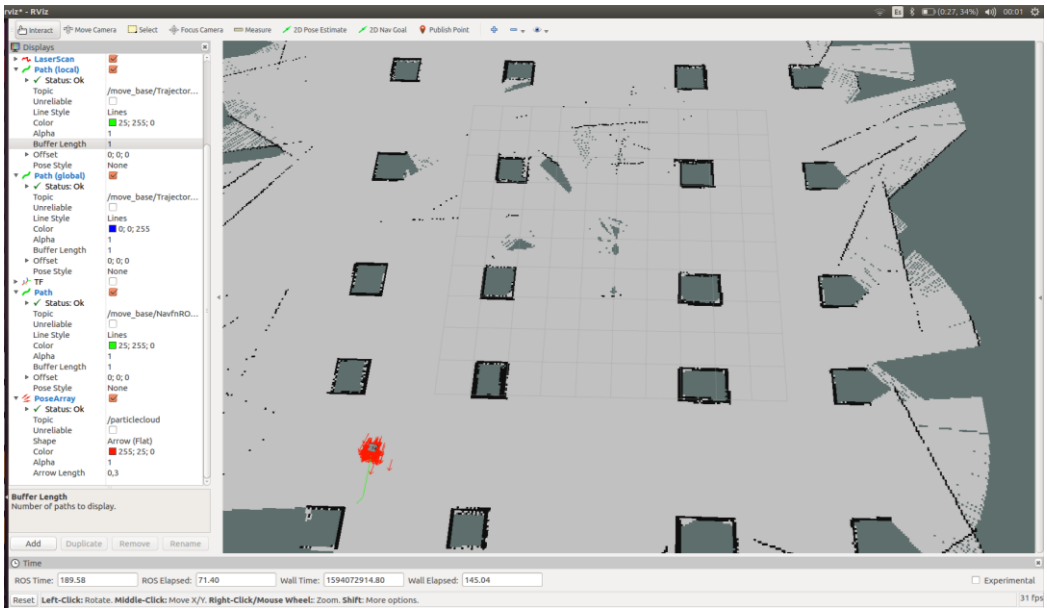

Figura 3.7. Desplazamiento del robot visualizado en Rviz (5).

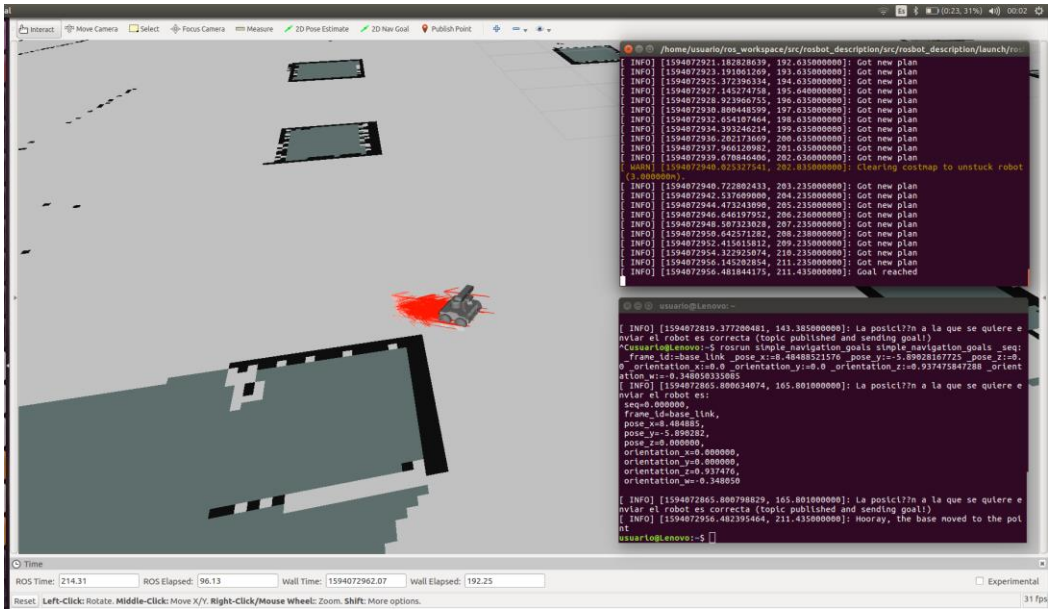

Figura 3.8. Desplazamiento del robot visualizado en Rviz (6).

3.2 Punto de destino enviado mediante “rostopic pub”

Conociendo dónde se publican los puntos enviados por Rviz, es sencillo entender que existe otra manera de enviar puntos de destino sin usar directamente el programa. Sería directamente desde la terminal, usando el comando “rostopic pub”, publicando sobre el “topic” mencionado en la Sección 3.1, como puede observarse en la Figura 3.9.

Además, en las Figuras 3.10, 3.11 y 3.12 se pueden observar diferentes trayectorias a diferentes puntos de destino.

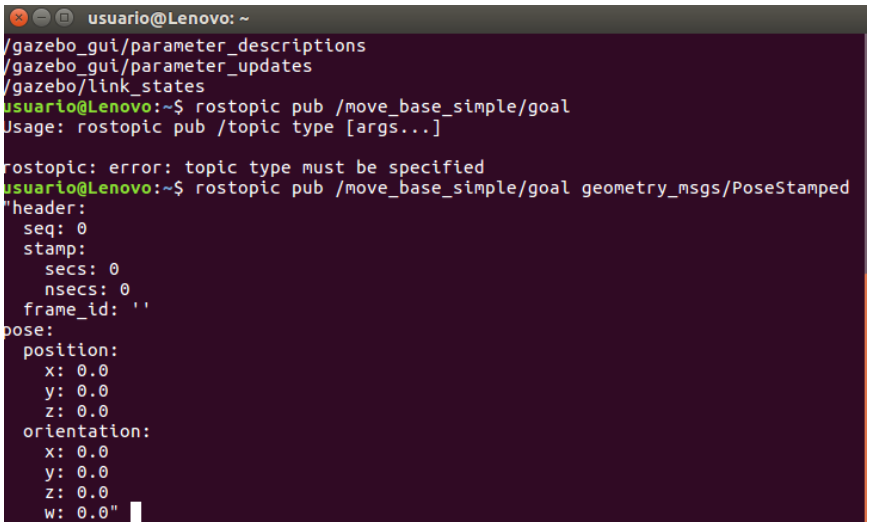

Figura 3.9. "rostopic pub" para publicar un punto de destino.

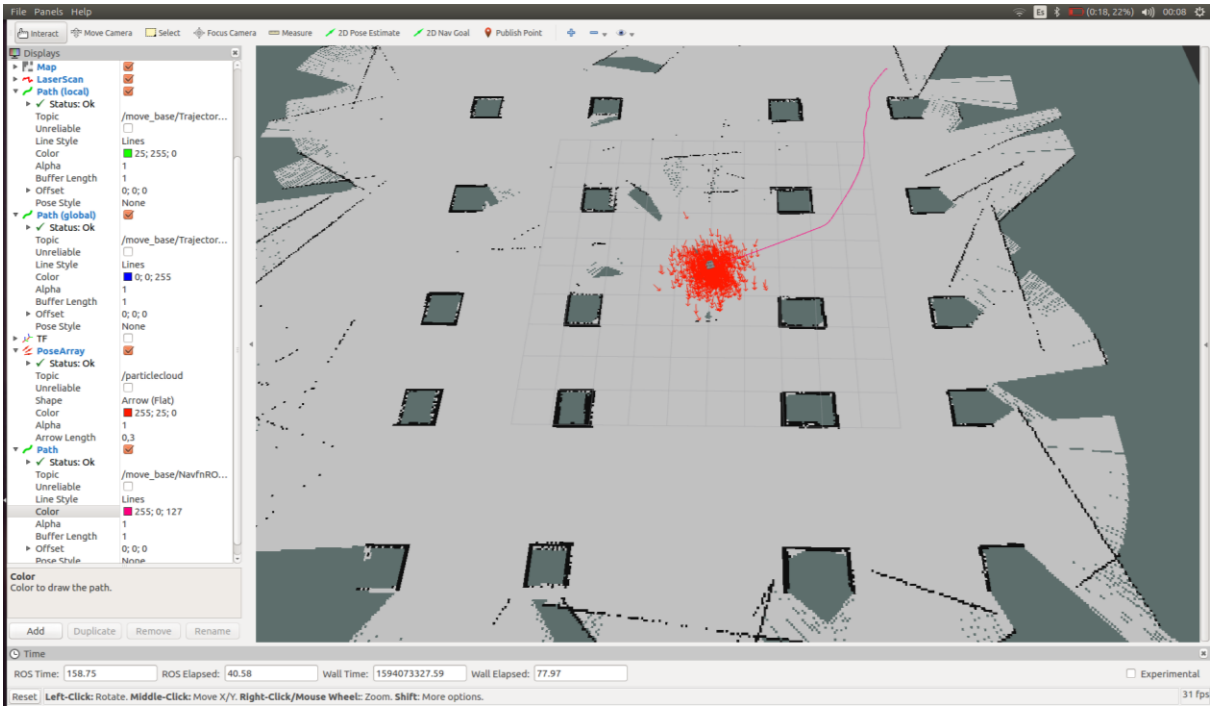

Figura 3.10. Trayectoria del robot para un punto de destino concreto.

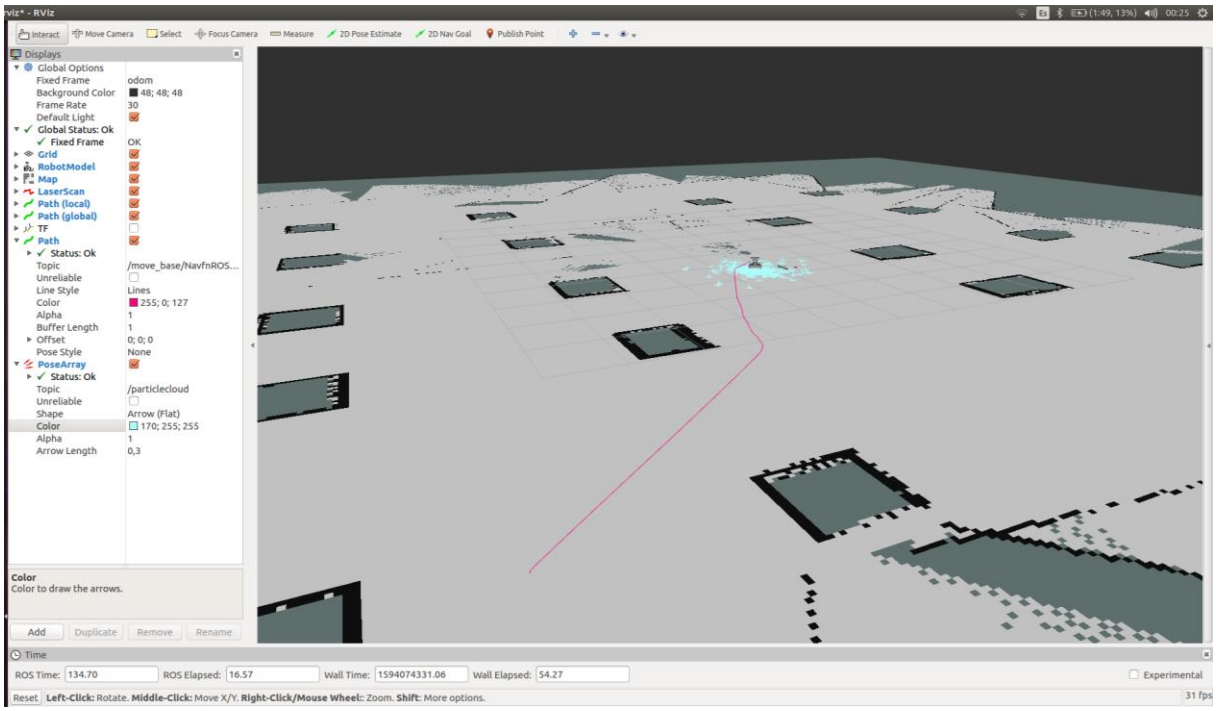

Figura 3.11. Trayectoria del robot para un punto de destino concreto. Vista lateral.

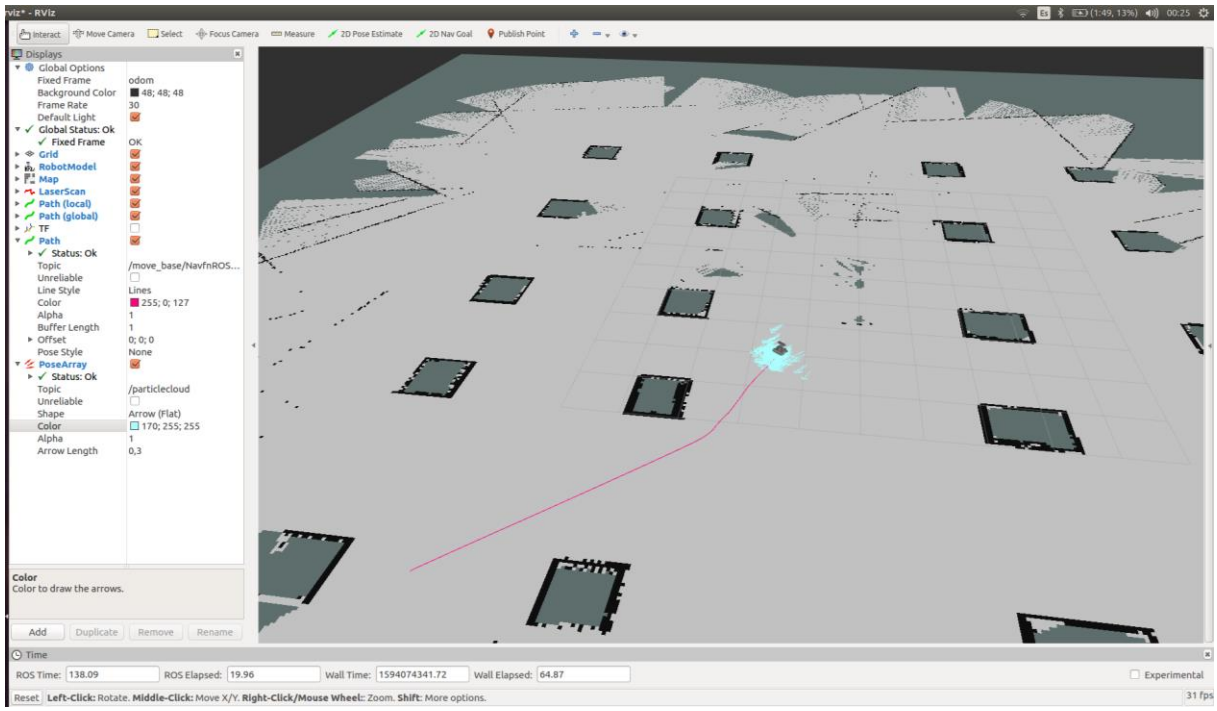

Figura 3.12. Trayectoria del robot para un punto de destino concreto. Vista superior.

3.3 Punto de destino enviado por el usuario mediante la ejecución de un "ActionClient" (con comprobación de las posiciones de los cubos)

Viendo los anteriores puntos uno puede llegar de manera sencilla a la conclusión de que éstos no son los métodos más ergonómicos para enviar puntos de destino al robot. Por ello, se ha desarrollado un código que permite al usuario, mediante la ejecución de una acción de ROS, enviar puntos de destino al robot.

Antes de empezar es necesario conocer qué es una acción de ROS y porqué se ha elegido esta vía para llevar resolver el problema. En ROS ya existen los servicios para enviar una petición a un nodo para llevar a cabo una tarea, además de recibir una respuesta a la petición. Sin embargo, como se profundiza en [18], en casos en los que los servicios tomen mucho tiempo de ejecución, el usuario necesita tener la posibilidad de cancelar la petición durante la ejecución o tener información periódica acerca del estado de la petición. Para ello, el paquete "actionlib" ofrece herramientas para crear servidores que ejecuten objetivos durante un largo tiempo y que puedan ser interrumpidos. Además, dicho paquete también ofrece la posibilidad de crear clientes que envíen peticiones al servidor. El esquema de funcionamiento de los "ActionClient" y los "ActionServer" puede observarse en la Figura 3.13.

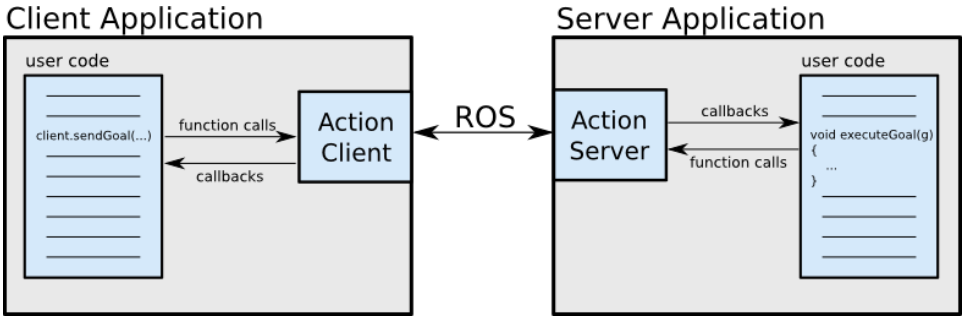

Figura 3.13. Esquema del funcionamiento de un "ActionClient" y un "ActionServer". Obtenida de [18].

Este esquema representa sencillamente cómo funcionan los servidores y clientes del paquete "actionlib". Como es fácil imaginar, el objetivo es crear un cliente que envíe peticiones al servidor. La duda es, ¿quién actúa como servidor? La respuesta es también simple: el paquete "move_base". Éste, con su ejecución, crea un servidor que a su vez genera un topic llamado "move_base/goal", al que se le enviarán los puntos de destino del robot.

Para ello se ha creado un paquete llamado "simple_navigation_goals", en el cual se incluye un código escrito en lenguaje C++, el cual será el cliente que envíe las peticiones al servidor. El código permite, a su vez, realizar las siguientes acciones:

- Recibir los parámetros del punto de destino como argumentos.
- Almacenar la posición de los cubos del mapa generado en la Sección 2.1.
- Detectar si la posición enviada coincide con el cubo.
- Determinar qué cubo ocupa la posición a la que se quiere enviar el robot.
- Publicar la posición (si es correcta) en el topic "/move_base/goal".

Para determinar si la posición a la que se quiere enviar el robot es correcta o no se ha creado una función en C++, que dependiendo de los valores de x e y, se analicen unos cubos u otros, para que el análisis de las posiciones sea lo más óptimo posible y requiera de la menor carga computacional que se pueda. En este caso, la función "CheckPosition" recibe como parámetros los valores de x e y de la posición de los cubos. La posición de los cubos se almacena en una matriz de valores llamada "cubes", donde los elementos son el nombre del cubo ("CubeX", siendo X el número establecido para cada uno), y sus posiciones x e y. Para conocer la distribución de los cubos se puede observar la Figura 3.14.

Como se puede observar, los cubos se han dividido en cuatro zonas diferenciadas, que serán determinadas por el valor de las posiciones de x e y. Finalmente, el cliente, al ser ejecutado con unos parámetros, determinará la validez de la posición, y en el caso de que sea correcta, será enviada al "topic"

correspondiente, provocando el movimiento del robot. Además, éste esperará a que el movimiento se realice con éxito, enviando un mensaje en consecuencia tanto si lo consigue como si no. Esto permite, además, no enviar múltiples puntos de manera simultánea, pudiendo provocar el bloqueo del robot.

El código, siendo una adaptación del código que puede observarse en [19], puede verse en el Anexo G.

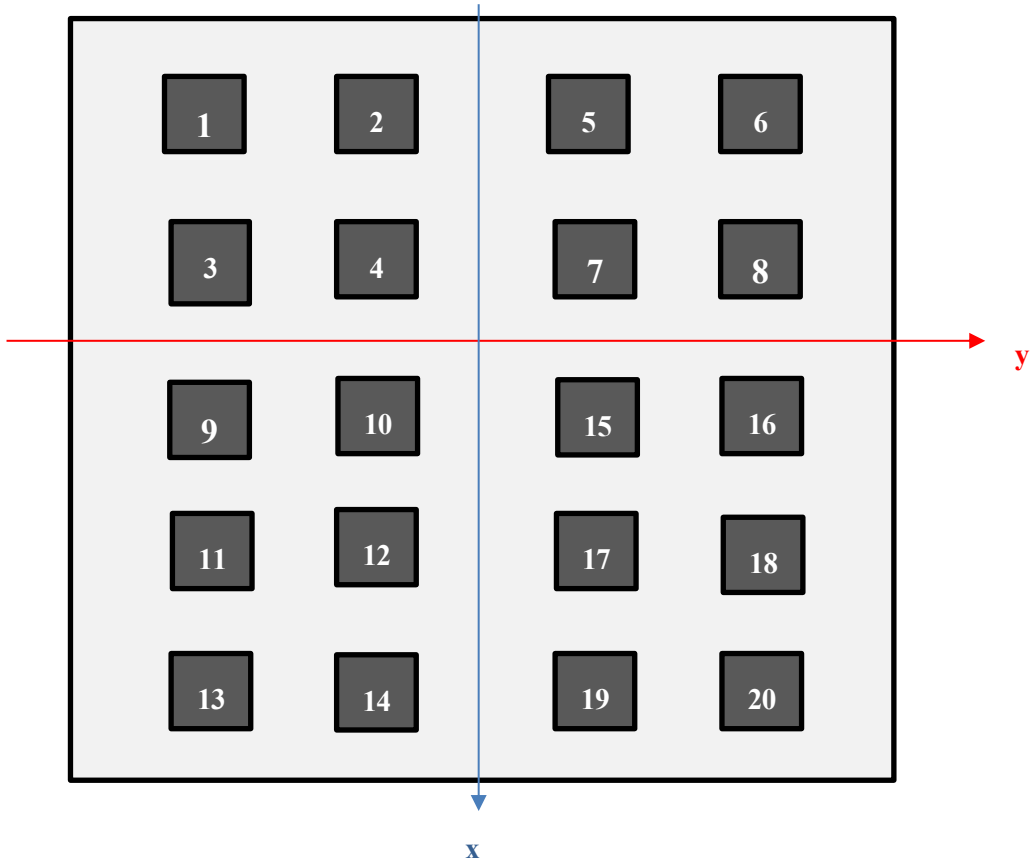

Figura 3.14. Disposición esquemática de los cubos en el mapa

Una vez creado el cliente, bastaría con ejecutarlo. Existen dos maneras, mediante un "roslaunch", necesitando de crear un archivo .launch previamente que contenga la llamada al cliente con los parámetros correspondientes o mediante una simple llamada al cliente, introduciendo los parámetros como argumentos directamente desde el terminal. En el segundo caso, la ejecución sería la mostrada a continuación, pudiendo verse en la Figura 3.15:

rosrun simple_navigation_goals simple_navigation_goals _seq:=0 _frame_id:=base_link _pose_x:=8.48488521576 _pose_y:=-5.89028167725 _pose_z:=0.0 _orientation_x:=0.0 _orientation_y:=0.0 _orientation_z:=0.937475847288 _orientation_w:=-0.348050335085

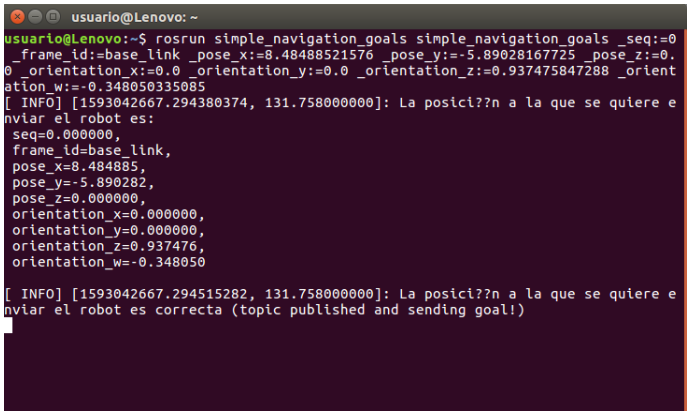

Figura 3.15. "rosrun" del "ActionClient" simple_navigation_goals.

En las Figuras 3.16 a 3.22 puede observarse el desplazamiento de un robot a su punto de destino, pudiendo contemplarse la trayectoria trazada y seguida, así como la localización.

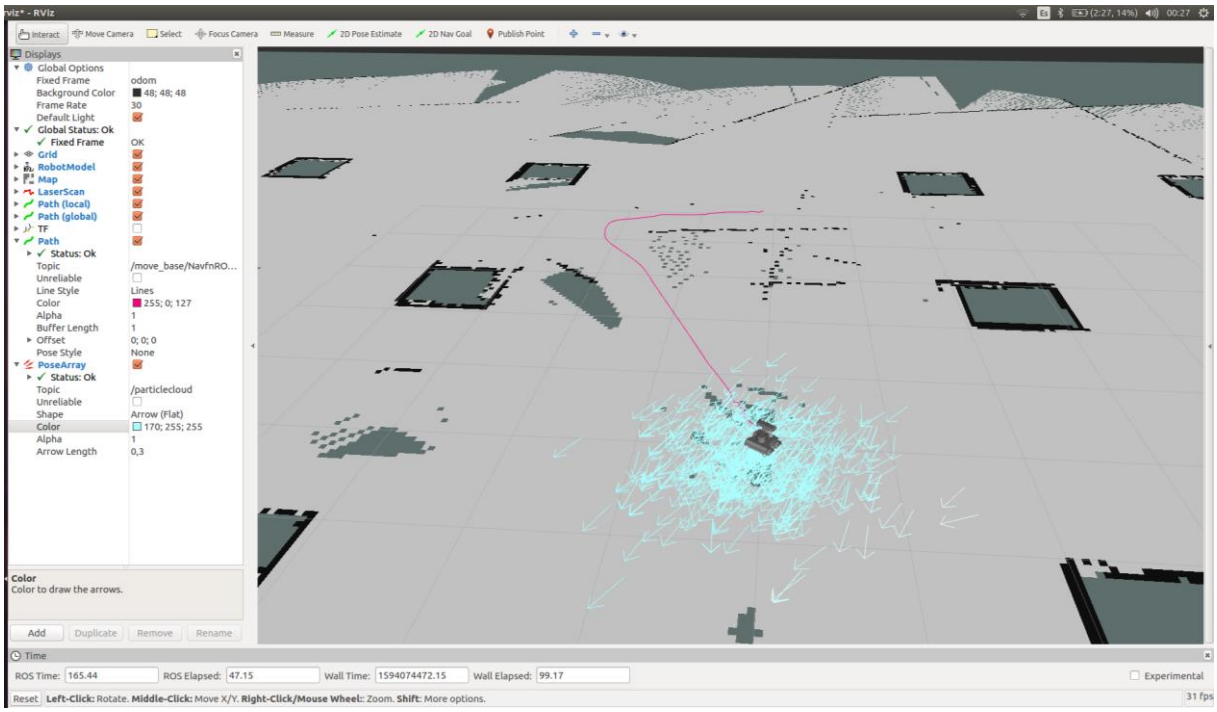

Figura 3.16. Desplazamiento del robot mediante “ActionClient”. Inicio del movimiento.

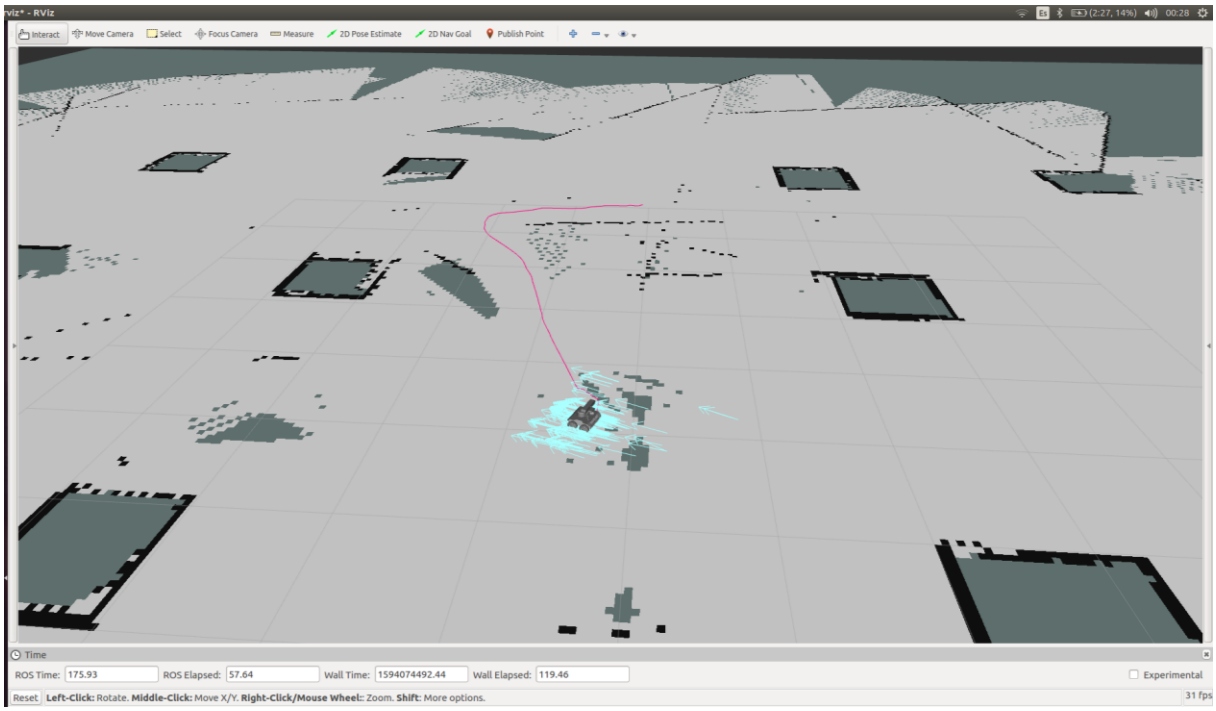

Figura 3.17. Desplazamiento del robot mediante “ActionClient”. Tras un pequeño desplazamiento.

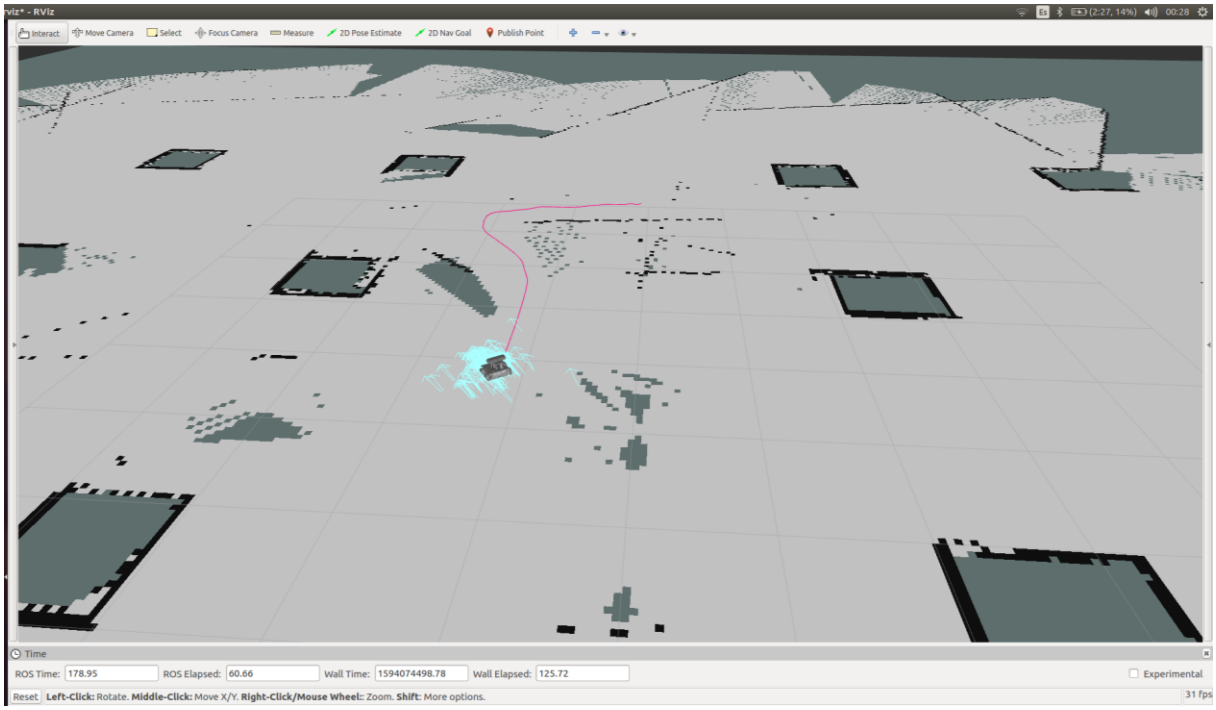

Figura 3.18. Desplazamiento del robot mediante “ActionClient”. Modificando la trayectoria.

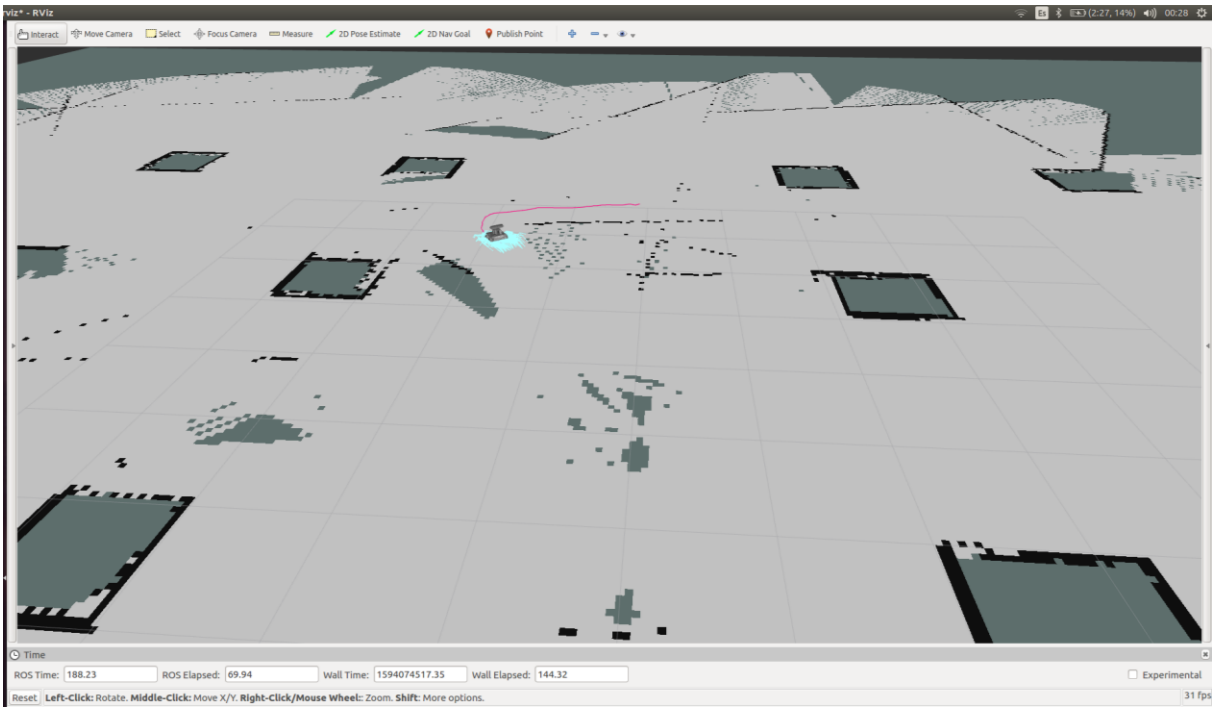

Figura 3.19. Desplazamiento del robot mediante “ActionClient”. Evitando los obstáculos.

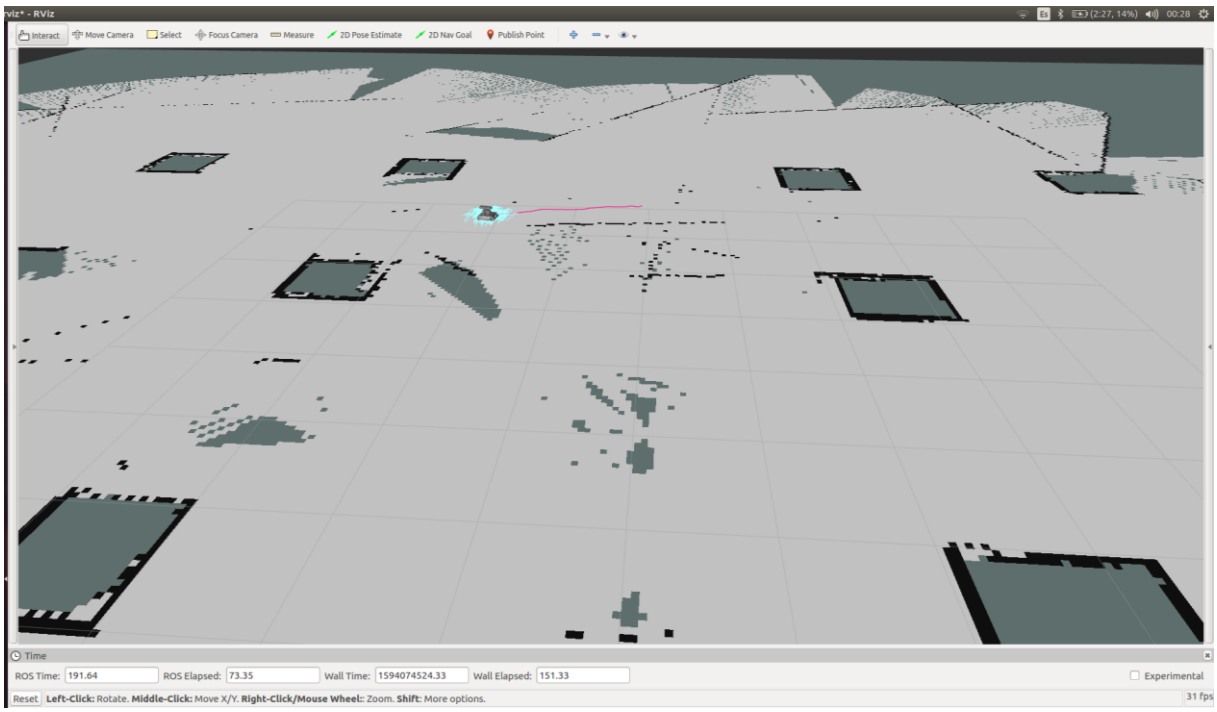

Figura 3.20. Desplazamiento del robot mediante “ActionClient”. Finalizando el desplazamiento.

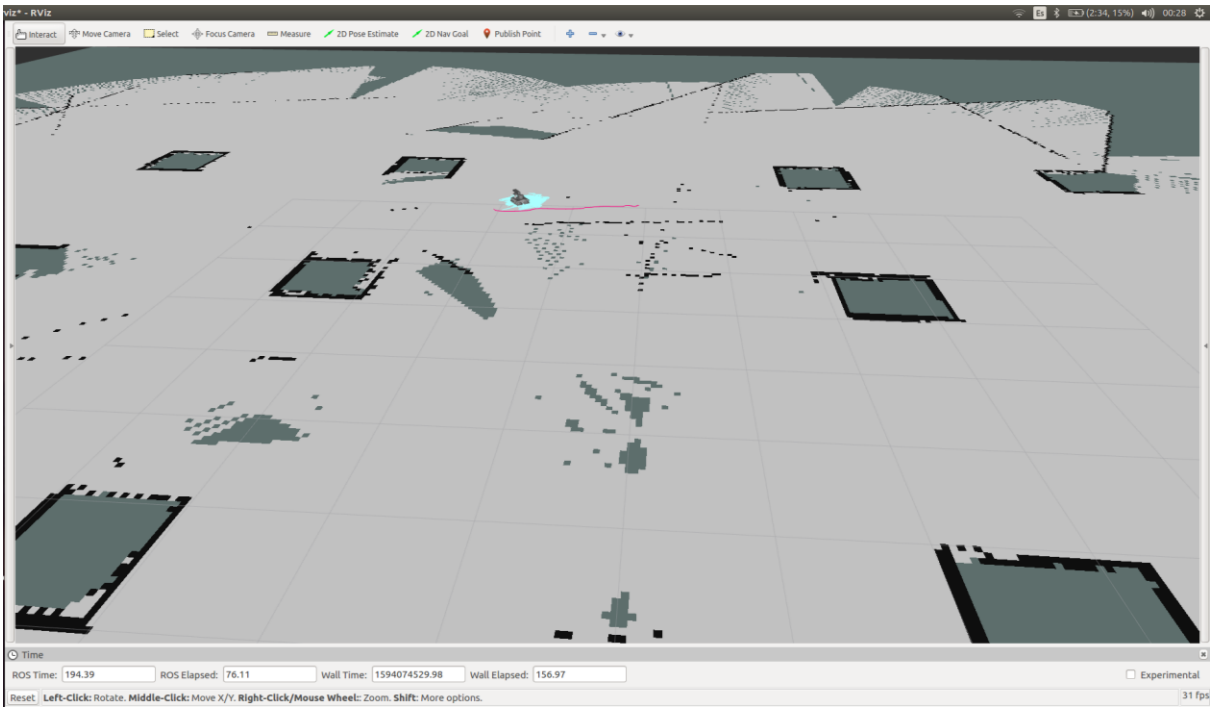

Figura 3.21. Desplazamiento del robot mediante “ActionClient”. Últimos instantes.

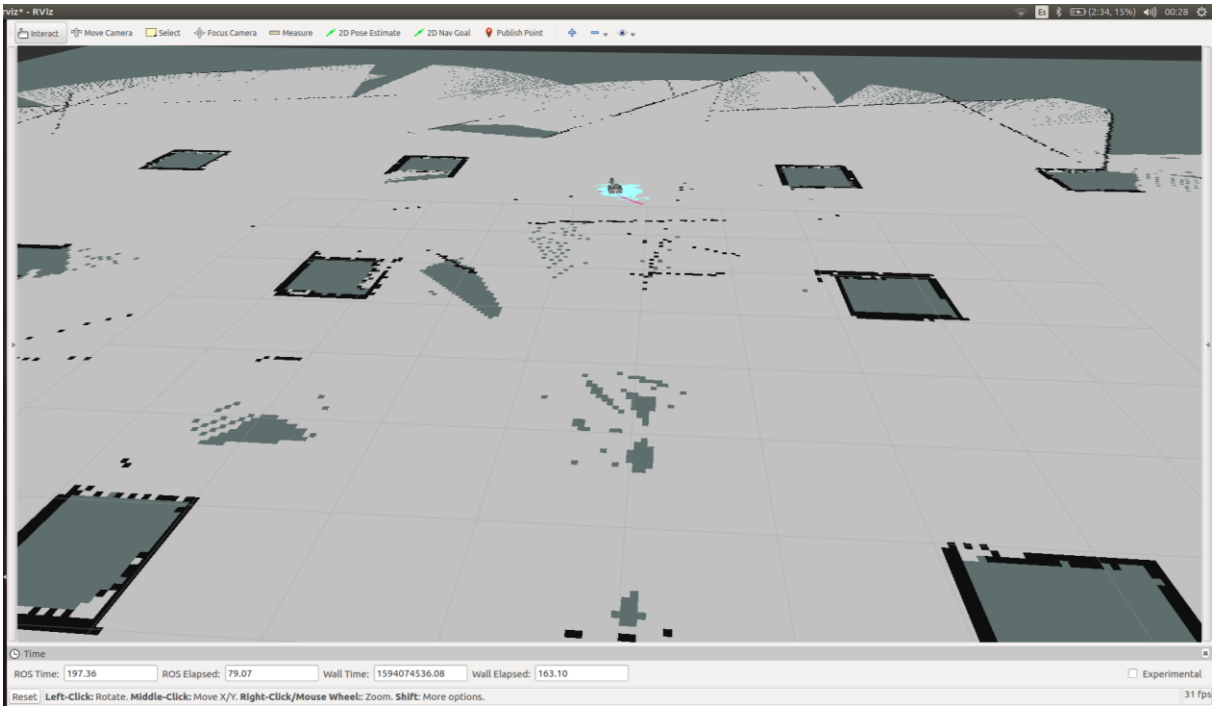

Figura 3.22. Desplazamiento del robot mediante "ActionClient". Llegada a la posición de destino.

3.3.1 Obtención de las coordenadas GPS del punto de destino

El modelo Rosbot de Husarion incluye un GPS el cual permitirá al robot ser localizado en todo momento en el lugar en el que se desplace. Además, permitiría corregir los errores de la odometría gracias a unos *"topics"* que publica el nodo de Ublox [20], fabricante del GPS que se integra. Dichos valores permiten conocer la distancia en metros desde un punto de referencia (fijo) a la posición actual del robot, lo cual otorgaría la posibilidad de modificar los valores de la odometría y actualizarlos para que el error no incremente. Esto no será implementado en este proyecto debido a que, al reducirse a simulación, no se tendrán valores reales de GPS.

En primer lugar, el robot podrá desplazarse de un lugar a otro mediante GPS. Sin embargo, deberá moverse mediante coordenadas en metros. Conociendo las medidas del lugar por el que este se moverá, se fija un punto de referencia, del cual se tendrán sus coordenadas GPS. Entonces, las coordenadas en metros del punto de destino del robot se calcularán desde dicho punto de referencia. El proceso para transformar esas coordenadas en metros a coordenadas GPS se explicará más adelante.

Una vez realizado el desplazamiento, el robot se supone que ha alcanzado su destino. No obstante, es poco probable que lo haya alcanzado con total precisión, por lo que se propone el uso de la odometría para alcanzar el punto exacto. De este modo, se debería distinguir dos maneras de trazado de trayectorias:

- Mediante GPS (distancia en metros).
- Mediante odometría (punto exacto en el mapa)

Por lo tanto, la primera trazada se realizaría mediante GPS, quedando en teoría el robot en una posición lo más cercana posible a la de destino. La segunda se realizaría corrigiendo la posible diferencia entre la posición de destino deseada y la actual mediante odometría. Puesto que dichos valores pueden ser corregidos por el GPS, permitiría un desplazamiento aún más exacto que usando sólo GPS.

La solución para la transformación de coordenadas en metros a GPS sería la siguiente: transformar las coordenadas del origen del mapa (punto de referencia), de GPS a UTM. Con las coordenadas UTM del origen, bastaría con sumarle las coordenadas en metros del punto de destino para obtener las coordenadas UTM del punto destino (teniendo en cuenta que se encuentren en la misma zona UTM), como se explica en [21]. De esta manera, realizando la transformación inversa obtendríamos las coordenadas GPS del punto de destino, el cual se le envía al robot.

Como en el "topic" "/move_base/goal" se publican las posiciones de los puntos de destino, se puede tomar desde ahí la información necesaria para las transformaciones mencionadas. A su vez, como los puntos de destino se envían en metros de distancia respecto a un origen de coordenadas y el robot necesita conocer las coordenadas GPS de dichos puntos, se ha creado un script en Python que lee la posición del punto desde el topic previamente mencionado y, dadas unas coordenadas GPS del punto de referencia, calcula las coordenadas GPS del punto de destino. Su nombre es "gps_goal_generator.py y puede observarse en el Anexo H.

Para ello se ha hecho uso del paquete "geonav_transform". El código se encarga de la transformación de la posición (x, y) en metros a coordenadas GPS, usando el comando "**xy2ll**" del paquete "geonav_conversions", detallado en [22], perteneciente al paquete "geonav_transform". Básicamente, lo que realiza es pasar las coordenadas GPS del punto de origen a coordenadas UTM. Suma a dichas coordenadas el valor de las posiciones x e y, y transforma esa suma de coordenadas UTM a GPS. La representación en el mapa de dichos puntos puede observarse en la Figura 3.23. Además de tomar los valores del "topic" sobre el que se envían los puntos de destino, el código también publica los valores de las coordenadas GPS de dichos puntos en un topic llamado "Goal_GPS_coordinates". En la Figura 3.24 puede verse este procedimiento de manera esquemática.

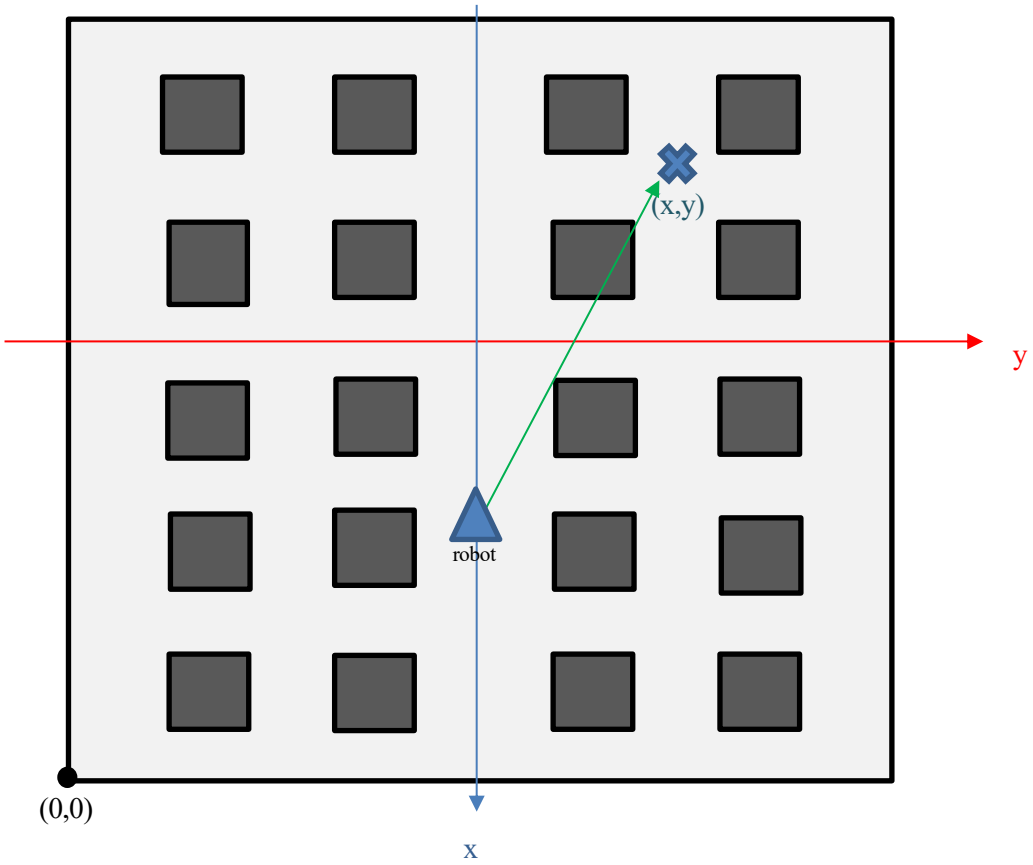

Figura 3.23. Esquema del mapa con los puntos de referencia y de destino.

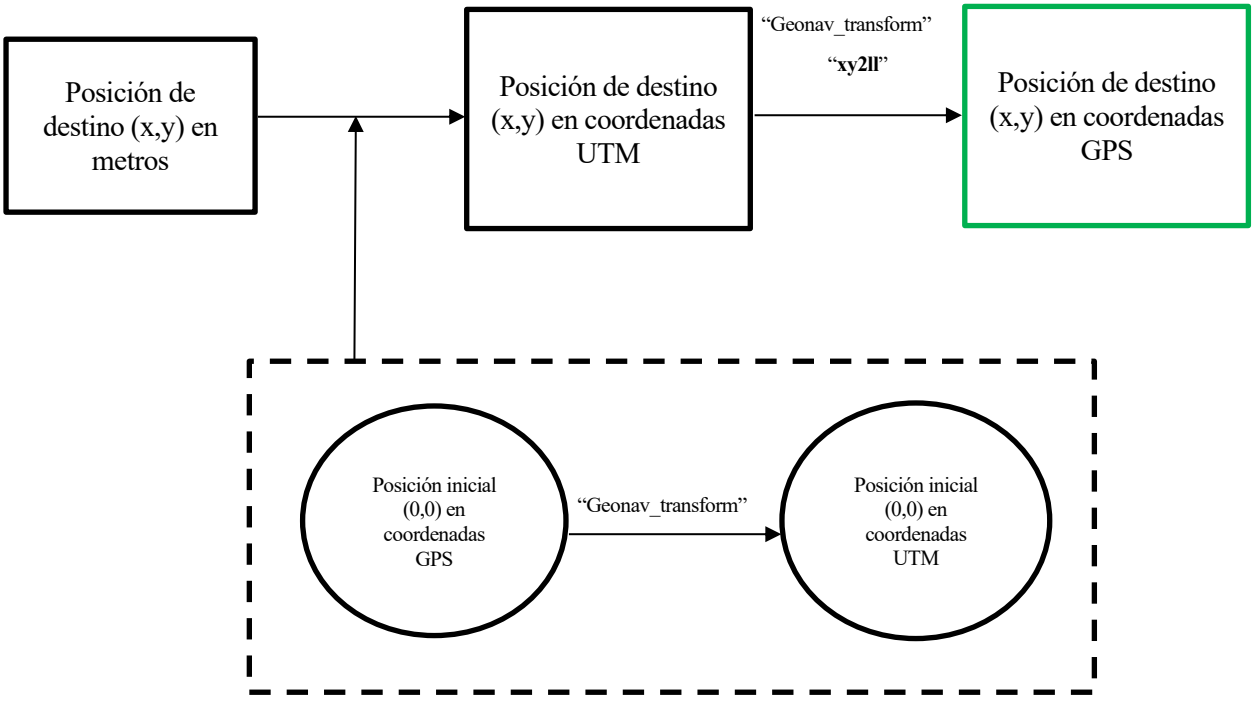

Figura 3.24. Obtención de las coordenadas del punto de destino (esquema).

A continuación, se muestra un ejemplo del uso del anterior código, estableciendo de manera manual la latitud y la longitud de la posición de origen así como del desplazamiento en x e y. Para ello se ha realizado una pequeña modificación del código, que lo simplifica sólo para el ejemplo, eliminando el apartado de suscripción y publicación de "topics". Véase la figura 3.25, donde se ejecuta el nodo y se aporta la información sobre la posición de origen, el desplazamiento total y la posición de destino.

```
usuario@Lenovo:~/ros_workspace/src/gps_goal/scripts$ rosrun gps_goal gps_goal_ma
nual.py
El origen de coordenadas es :  -6.00176 37.411173
la distancia a recorrer es (x,y)= ( 7.0 13.9 ) metres
La distancia total es h=  15.5630973781
Las coordenadas GPS del punto destino son:  37.4112365913 -6.00163448753
usuario@Lenovo:~/ros_workspace/src/gps_goal/scripts$
```

Figura 3.25. Obtención de las coordenadas GPS de un punto de destino. (Información del nodo)

El punto elegido como origen pertenece a uno de los laterales de la Escuela Técnica Superior de Ingeniería, mientras que el desplazamiento ha sido elegido aleatoriamente. En las Figuras 3.26 y 3.27 se pueden ver en detalle estos aspectos. Como se puede apreciar, la distancia obtenida en Google Maps no coincide con la estimada, con lo que se refuerza la idea de combinar odometría con GPS para obtener una localización más precisa.

Para mejorar este proceso bastaría con usar, como se ha mencionado previamente, los datos publicados en el "topic" "dat/shift" por parte del paquete de Ublox, el GPS físico que se le implementaría al robot. Este calcula la distancia exacta en metros desde el punto de referencia a la posición actual del robot. Por lo tanto, permitiría corregir el fallo en la estimación de las coordenadas realizada en el código previamente explicado.

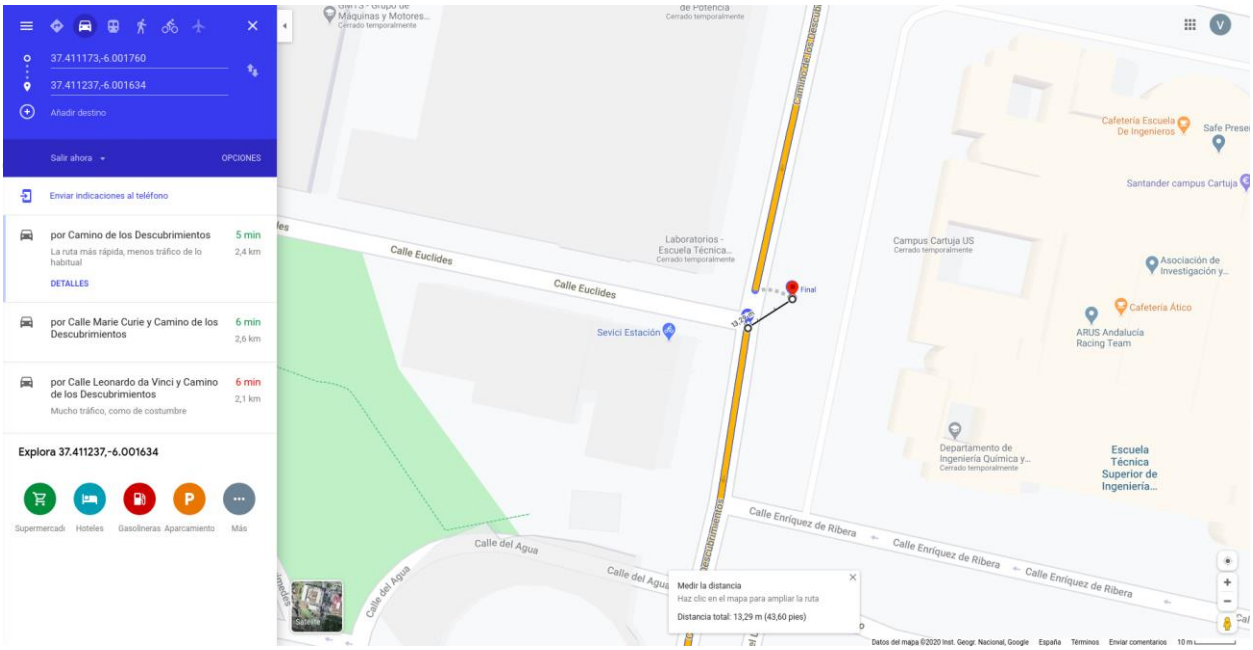

Figura 3.26. Distancia entre punto de origen y de destino. Obtenida de [32].

Figura 3.27. Detalle de la distancia entre punto de origen y de destino. Obtenida de [32].

4 NAVEGACIÓN SIMULTÁNEA DE TRES ROBOTS

A la hora de abordar la localización y navegación simultánea de tres robots, la tarea no dista mucho de la mera réplica de un robot a tres, teniendo en cuenta que en ROS se han de crear "topics" únicos para cada robot. Por lo tanto, la réplica, una vez realizada con éxito, valdría tanto para tres robots como para tantos quiera el usuario, siempre teniendo en cuenta la carga computacional que supone la simulación y navegación de más de un robot a la vez. En primer lugar, se abordará la simulación de tres robots simultáneamente, mostrándolos en el mapa en posiciones diferentes para evitar el choque inicial.

4.1 Simulación de tres robots de manera simultánea

Para llevar a cabo este proceso, se han tomado los códigos generado por un usuario, publicados en [23] y posteriormente modificados para versiones posteriores de ROS en [24]. En estos códigos, integrados en el paquete "multi_robot", se estructura la localización de la siguiente manera:

- "One_robot.launch": código que lanza un robot (recibe como argumentos la posición inicial y el nombre de dicho robot).
- "Robots.launch": código que agrupa el número deseado de robots, otorgándoles a cada uno un nombre y posición inicial diferentes. Además, determina la descripción del robot (el modelo a elegir) y es el el encargado de llamar al código "one_robot.launch" tantas veces como robots se vayan a simular.
- "Main.launch": código que lanza el entorno en el que los robots se van a mostar (archivo .world) y lanza el código "robots.launch", generando así un procedimiento claramente estructurado, cuyo esquema de funcionamiento puede observarse en la Figura 4.1.

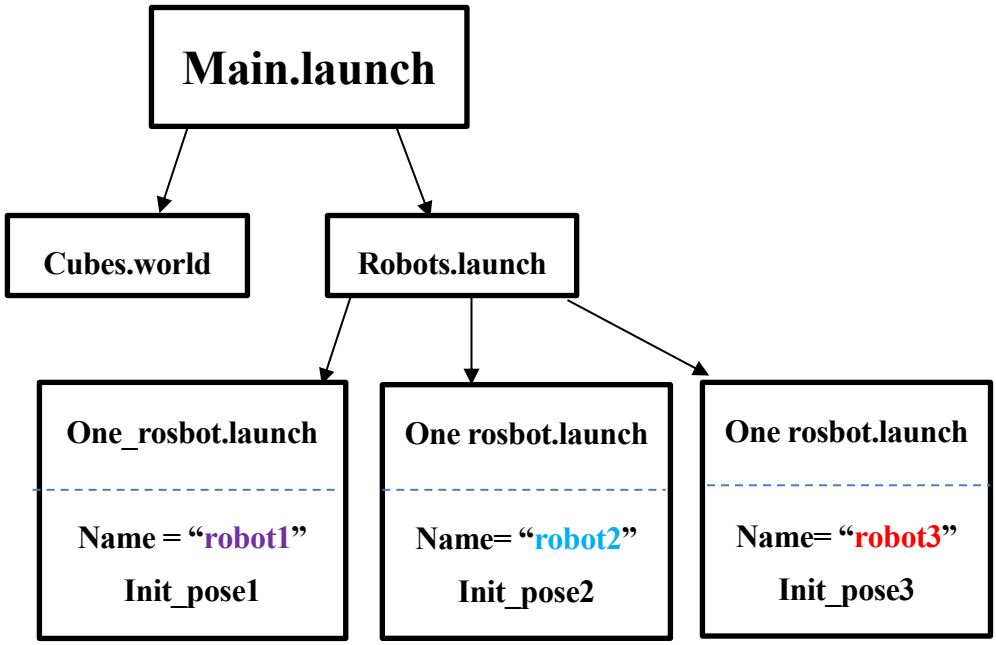

Figura 4.1. Estructura para la simulación de dos robots

Los códigos a ejecutar serían los siguientes:

- "One_robot.launch". Puede observarse en el Anexo I.

- “Robots.launch”. Puede observarse en el Anexo J.
- “Main.launch”. Puede observarse en el Anexo K.

Uno de los aspectos más importantes y que determinará el buen funcionamiento de la simulación simultánea de varios robots es que todos presenten “topics” únicos, determinados por los nombres que se les ha otorgado. Esto quiere decir que, si existe un solo “topic” “/odom”, otro “/cmd_vel”, no funcionará. Se necesita uno diferente para cada uno, teniendo los nombres “robotX/odom”, etc., siendo X el número dado al robot.

Ejecutando el archivo “main.launch” se consigue una simulación en Gazebo como muestra la Figura 4.2.

Sin embargo, de nada sirve simular los robots si no existe manera de conocer su localización. Para ello se usará una instancia del nodo AMCL (Adaptative Monte Carlo Localization) para cada robot, al igual que se ha usado para un solo robot en la Sección 2.3.

4.2 Localización de tres robots de manera simultánea

Para llevar a cabo esta tarea se ha creado un paquete llamado “multi_robots_nav”, el cual contiene el código “multi_robots_nav.launch”, encargado de lanzar el nodo de AMCL para cada robot (además de los nodos “move_base”, más adelante), los cuales tendrán la nomenclatura “amcl_robotX.launch", siendo X el número del robot.

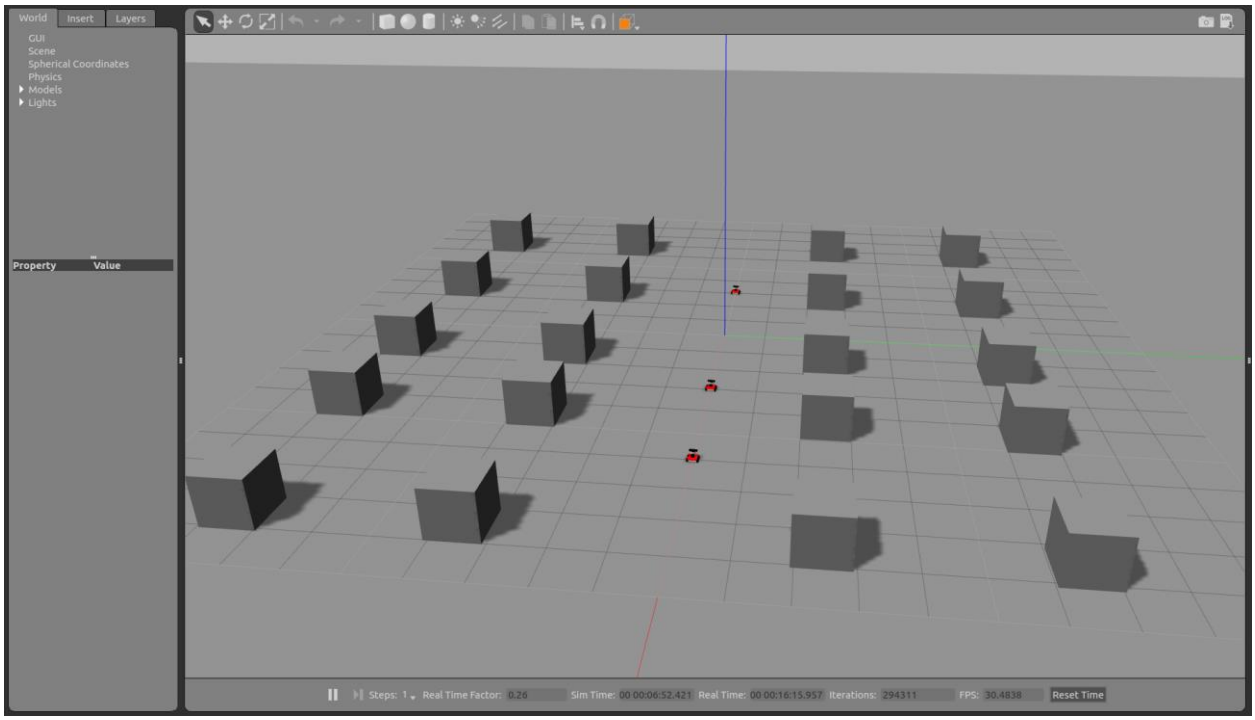

Figura 4.2. Simulación en Gazebo de tres robots simultáneamente.

Los códigos “multi_robots_nav.launch” y “amcl_robot1.launch” se encuentran en los Anexos L y M respectivamente. Con el primero se lanza el mapa donde se desplazan los robots y se lanzan a su vez los archivos .launch que ejecutan la localización AMCL para cada robot, mientras que con el segundo se definen los argumentos del nodo AMCL, entre ellos el mapa del entorno, el nombre del “topic” del láser, la posición inicial y los nombres de los “topics” de “odom” y “base_link”. En este caso, todos los “topics” vendrán precedidos por el nombre “robot1”. De esta manera, para cada robot se cambiará el número, y se crean “topics” independientes para cada uno de ellos.

La ejecución del código “multi_robots_nav.launch” (sumada a la ejecución de la simulación de los robots) permite localizar a cada robot en su posición inicial. Puede ser observado por los “topics” “/robotX/particlecloud”, que se representan como nubes de puntos en Rviz, como se puede ver en la Figura 4.3.

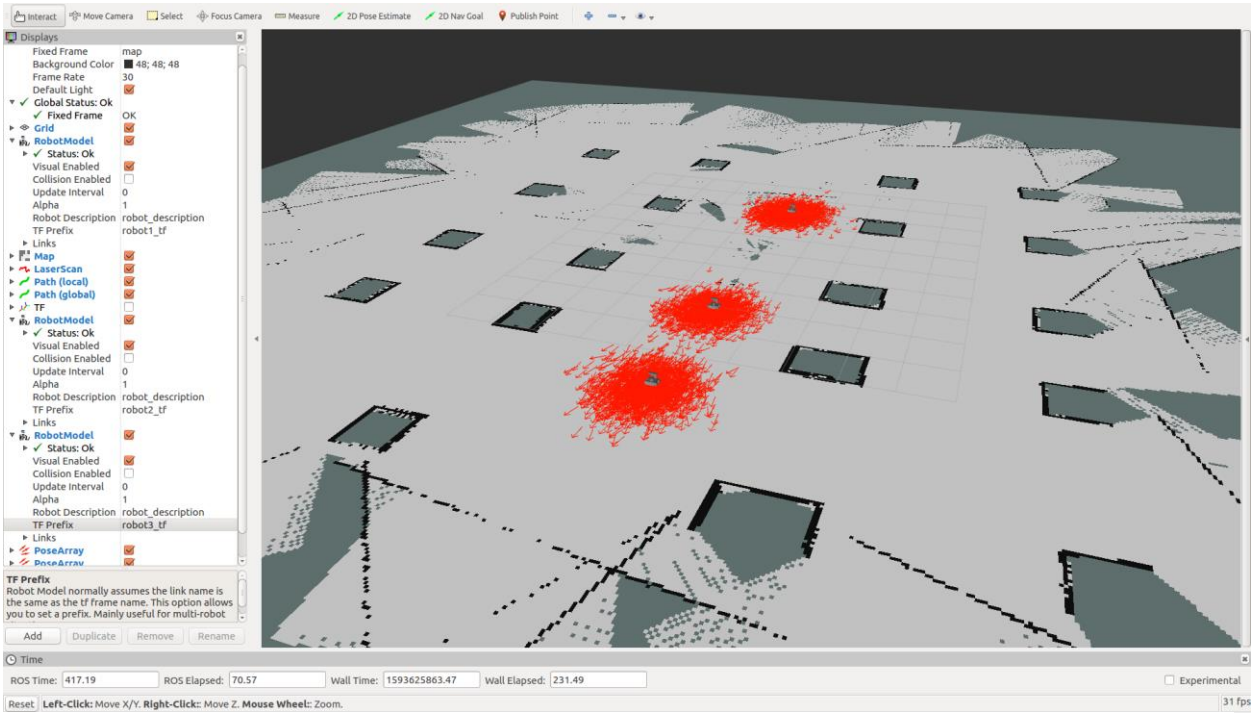

Figura 4.3. Localización en Rviz de tres robots de manera simultánea.

4.3 Navegación de tres robots de manera simultánea

Para llevar a cabo esta ardua tarea, se deberá ejecutar una instancia del nodo “move_base” para cada robot, realizando el “remapping” de los “topics” generados. Es decir, se cambiará el nombre de los “topics” a observar, que serán por defecto los genéricos (“/odom”, “/cmd_vel”, etc.), a los específicos para cada robot, de manera que se puedan controlar todos por separado.

Este proceso se realiza de igual manera que la localización explicada en la Sección 4.2. Para cada robot se crea un código llamado “move_baseX.launch”, siendo X el número del robot. La ejecución de estos se realiza en el código “multi_robots_nav.launch”, situado en el Anexo L.

Por otro lado, el código “move_base1.launch” realiza una tarea similar al “amcl_robot1.launch”. Su aspecto puede observarse en el Anexo N.

Una vez lanzados los nodos se puede enviar un punto de destino a los robots.

4.3.1 Punto de destino señalado en Rviz

El proceso que llevar a cabo es idéntico al explicado en la Sección 3.1. Rviz permite enviar un punto de destino a los robots mediante el comando “2D Nav Goal”, con la diferencia de que en las propiedades de éste es necesario cambiar el “topic” sobre el que se publica. Bastaría con clicar con el botón derecho sobre la herramienta “2D Nav Goal” y elegir “Tool Properties”. Mientras que en la Sección 3.1 el “topic” es “/move_base_simple/goal”, en este caso, el nombre vendría precedido del término “/robotX”, siendo X el número del robot. Por lo tanto, publicaría sobre los “topics” “/robot1/move_base_simple/goal”, “/robot2/move_base_simple/goal” y “/robot3/move_base_simple/goal”.

El uso de esta herramienta puede observarse en la Figura 4.4, mientras que en las Figuras 4.5 a 4.9 puede observarse el desplazamiento de uno de los robots, además de la localización isntantánea.

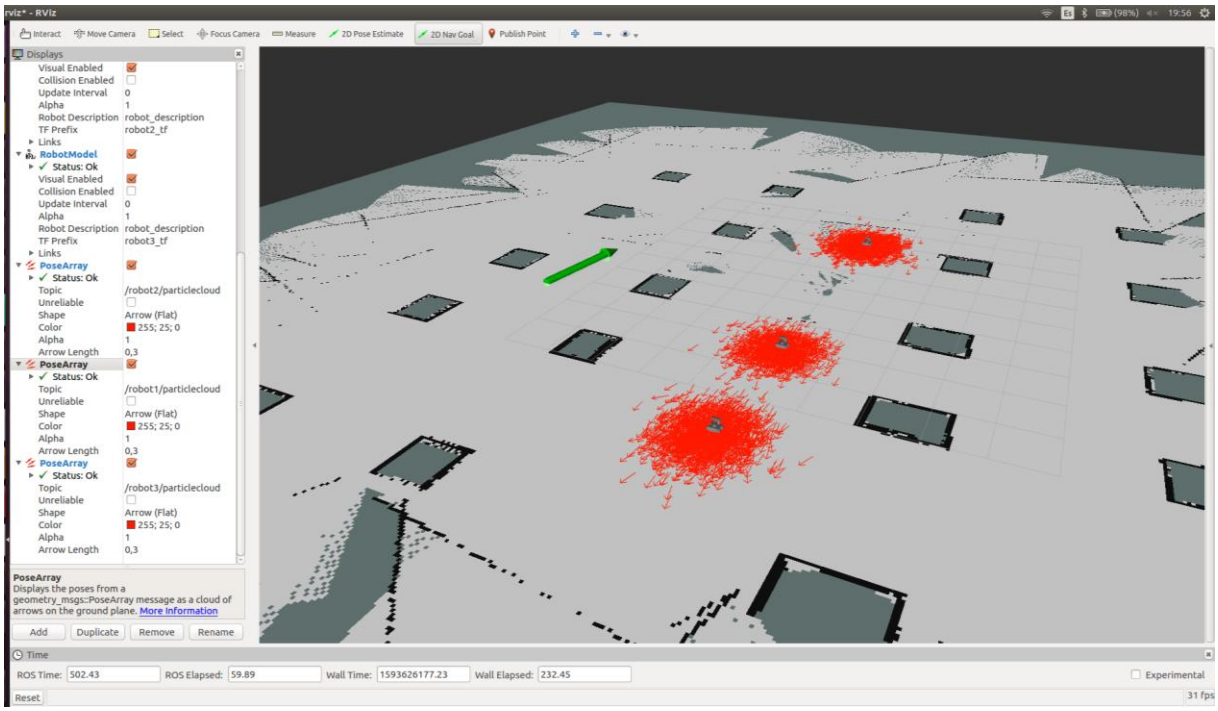

Figura 4.4. Comando "2D Send Goal" con varios robots.

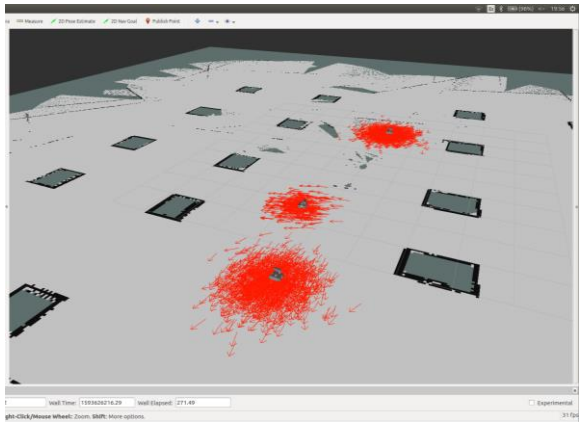

Figura 4.5. Viaje de un robot a su punto de destino en Rviz. Inicio del movimiento.

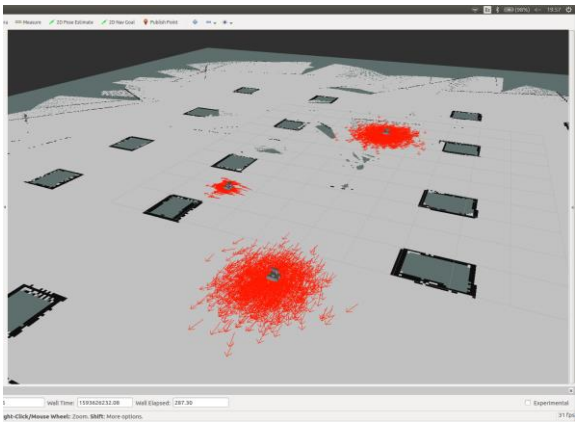

Figura 4.6. Viaje de un robot a su punto de destino en Rviz. Pequeño desplazamiento.

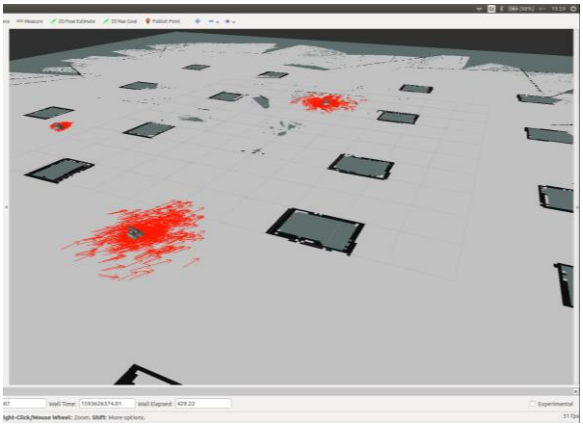

Figura 4.7. Viaje de un robot a su punto de destino en Rviz. Últimos instantes del desplazamiento.

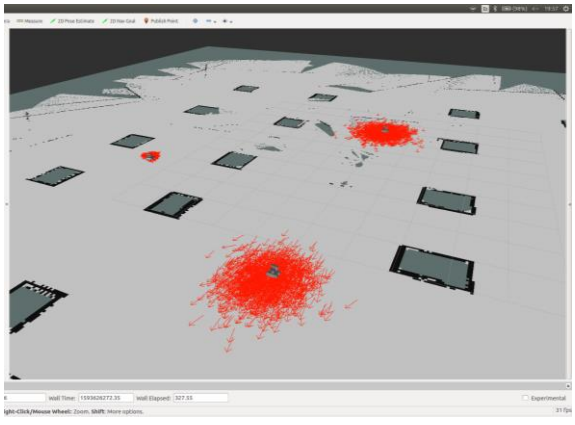

Figura 4.8. Viaje de un robot a su punto de destino en Rviz. Llegada a la posición final.

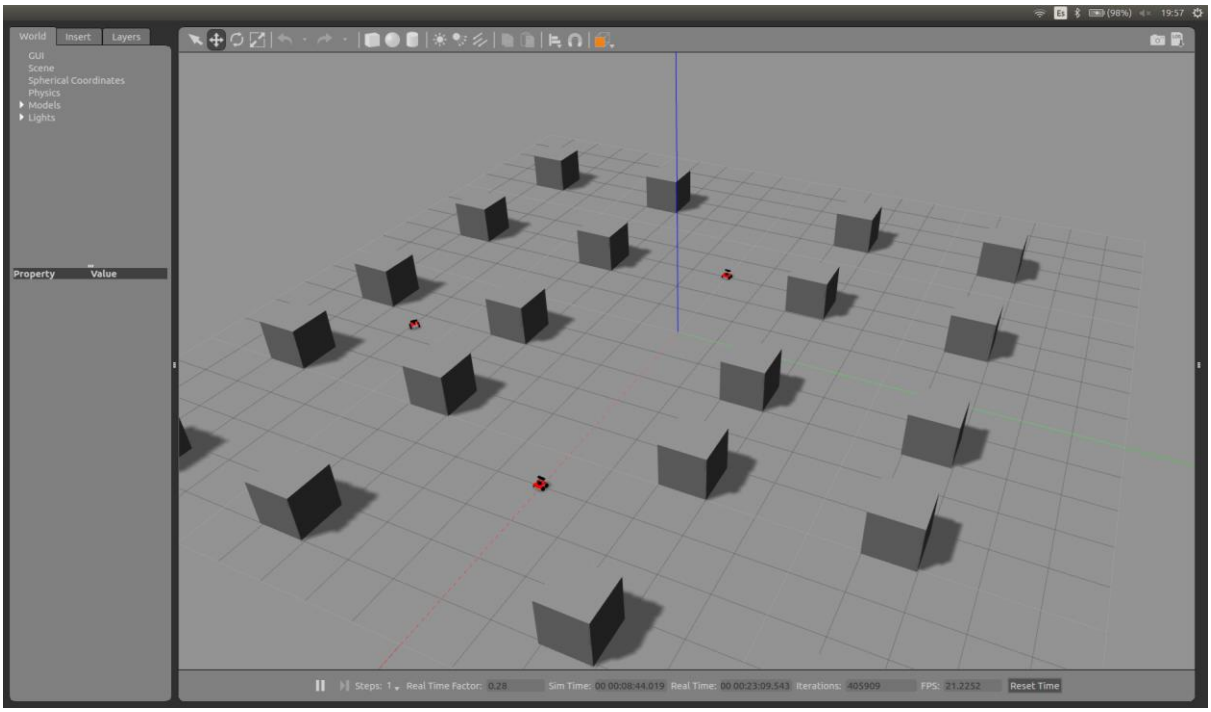

Figura 4.9. Viaje de un robot a su punto de destino. Posición final (en Gazebo).

4.3.2 Punto de destino enviado mediante "rostopic pub"

El proceso sería el mismo que en la Sección 3.2, de nuevo con la diferencia de que habría que publicar sobre los nuevos "topics" correspondientes a los robots. Los "topics" sobre los que se publica son los mencionados en la Subsección anterior. Se puede observar de nuevo en las Figuras 4.10 y 4.11.

```
^Cusuario@Lenovo:~$ rostopic pub /robot3/move_base/goal move_base_msgs/MoveBaseActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: map
goal_id:
  stamp:
    secs: 216
    nsecs: 842000000
  id: ''
goal:
  target_pose:
    header:
      seq: 0
      stamp:
        secs: 126
        nsecs: 842000000
      frame_id: map
    pose:
      position:
        x: 8.48
        y: -5.89
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.94
        w: -0.35"
publishing and latching message. Press ctrl-C to terminate
```

Figura 4.10. "rostopic pub" para "/robot1/move_base/goal"

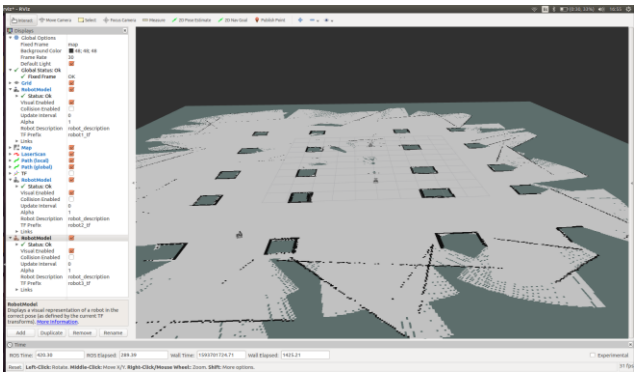

Figura 4.11. Navegación en Rviz tras el "rostopic pub" en "/robot1/move_base/goal"

4.3.3 Punto de destino enviado mediante "ActionClient"

El proceso que llevar a cabo en este caso es análogo a la Sección 3.3. Sin embargo, como se ha mencionado en las secciones previas, los "topics" han de ser únicos para cada robot. De manera que se realiza una réplica del nodo "simple_navigation_goals", publicando en el topic "/robotX/move_base/goal", siendo X el número del robot. Para ello se utiliza la técnica de "remap" en los archivos .launch. También se utiliza en la llamada de los nodos mediante "rosrun".

Para llevar a cabo la navegación simultáneamente a la localización, se añade la siguiente línea de código al archivo "main.launch" (perteneciente al paquete "multi_robot" (simulación de los robots)), de manera que se

simulan los robots, la localización y la navegación en un mismo archivo .launch:

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

Además, para enviar las posiciones de destino a los tres robots se han aunado tres archivos .launch en uno solo de manera que se ejecuta el envío de dichas posiciones de manera simultánea. El nombre es "three_robots_nav.launch" y su aspecto puede observarse en el Anexo O.

Si se prefiriese ejecutar nodos de manera individual, un ejemplo sería el siguiente, pudiendo observarse en la Figura 4.12.

```
rosrun simple_navigation_goals simple_navigation_goals _seq:=1 _secs:=216
_nsecs:=457000000 _frame_id:=base_link _pose_x:=-5.21542453766 _pose_y:=0.117576599121
_pose_z:=0.0 _orientation_x:=0.0 _orientation_y:=0.0 _orientation_z:=-0.639192421732
_orientation_w:=0.769046843827 /move_base:=/robot1/move_base
```

En negrita se puede observar la táctica previamente mencionada de "remap", que permite cambiar el nombre del topic que se publica. Este mismo código sería aplicable a los demás robots con tan sólo cambiar el número de éste.

En las Figuras 4.13 y 4.14 se puede observar la información sobre el nodo "simple_navigation_goals", encargado de publicar los puntos de destino para los robots y sobre el "topic" publicado por éste respectivamente.

Como se puede observar en la Figura 4.15, el robot3 procede a desplazarse a su punto de destino.

```
usuario@Lenovo:~$ rosrun simple_navigation_goals simple_navigation_goals _seq:=0
 _secs:=216 _nsecs:=457000000 _frame_id:=map _pose_x:=-5.21542453766 _pose_y:=0.
117576599121 _pose_z:=0.0 _orientation_x:=0.0 _orientation_y:=0.0 _orientation_z
:=-0.639192421732 _orientation_w:=0.769046843827 move_base:=/robot3/move_base
```

Figura 4.12. "rosrun" para "robot3".

```
usuario@Lenovo: ~
usuario@Lenovo:~$ rosnode info /simple_navigation_goals
--------------------------------------------------------------------------------
Node [/simple_navigation_goals]
Publications:
 * /robot3/move_base/cancel [actionlib_msgs/GoalID]
 * /robot3/move_base/goal [move_base_msgs/MoveBaseActionGoal]
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]
 * /robot3/move_base/feedback [move_base_msgs/MoveBaseActionFeedback]
 * /robot3/move_base/result [unknown type]
 * /robot3/move_base/status [actionlib_msgs/GoalStatusArray]

Services:
 * /simple_navigation_goals/get_loggers
 * /simple_navigation_goals/set_logger_level
```

Figura 4.13. Información del nodo "simple_navigation_goals" que publica "/robot3/move_base/goal".

```
usuario@Lenovo:~$ rostopic info /robot3/move_base/goal
Type: move_base_msgs/MoveBaseActionGoal

Publishers:
 * /move_base3 (http://Lenovo:44459/)
 * /simple_navigation_goals (http://Lenovo:42163/)

Subscribers:
 * /move_base3 (http://Lenovo:44459/)
```

Figura 4.14. "Rostopic info" del topic "/robot3/move_base/goal"

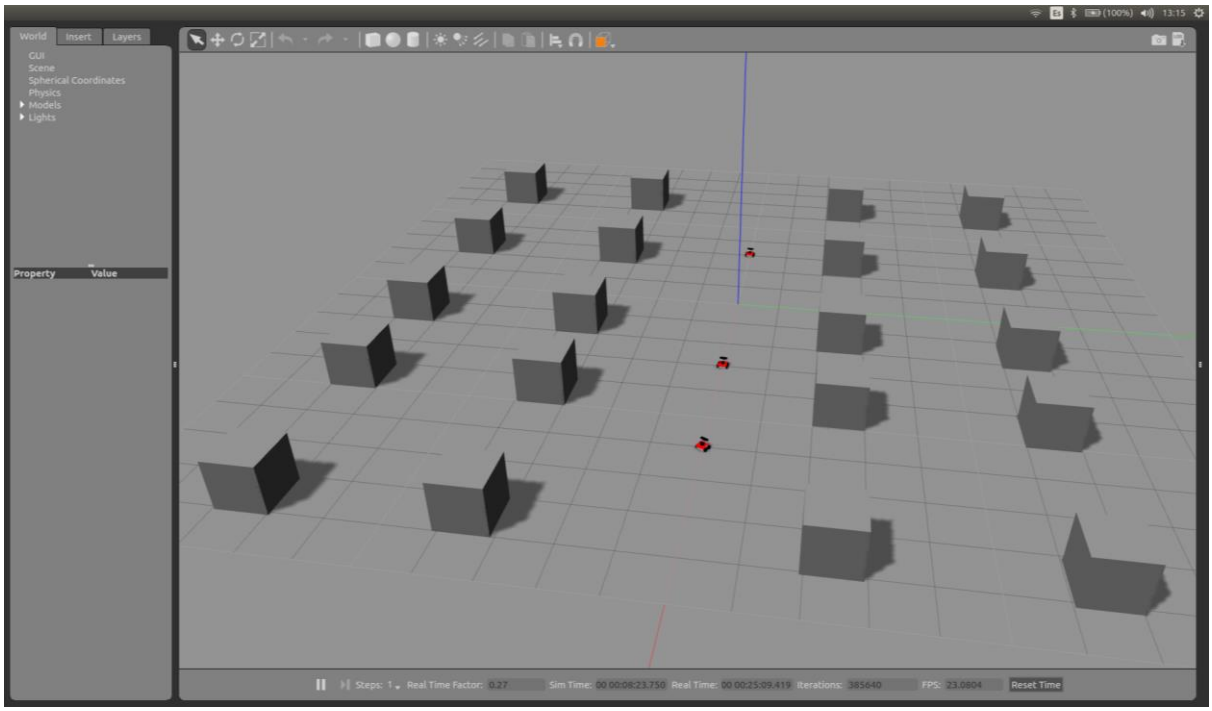

Figura 4.15. Movimiento del robot3 hacia el punto de destino indicado en "/robot3/move_base/goal".

En las Figuras 4.16 y 4.17, mientras tanto, se pueden observar diferentes trayectorias trazadas por los robots hacia sus puntos de destino.

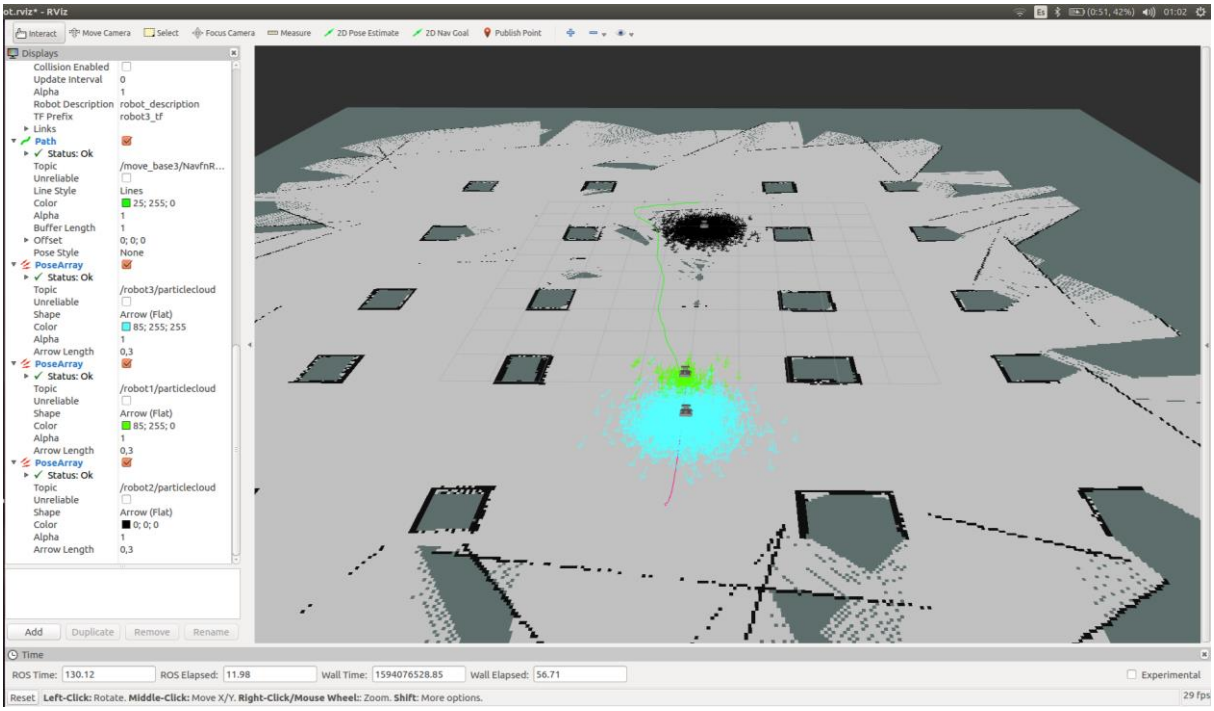

Figura 4.16. Trayectorias generadas por varios robots. Vista lateral.

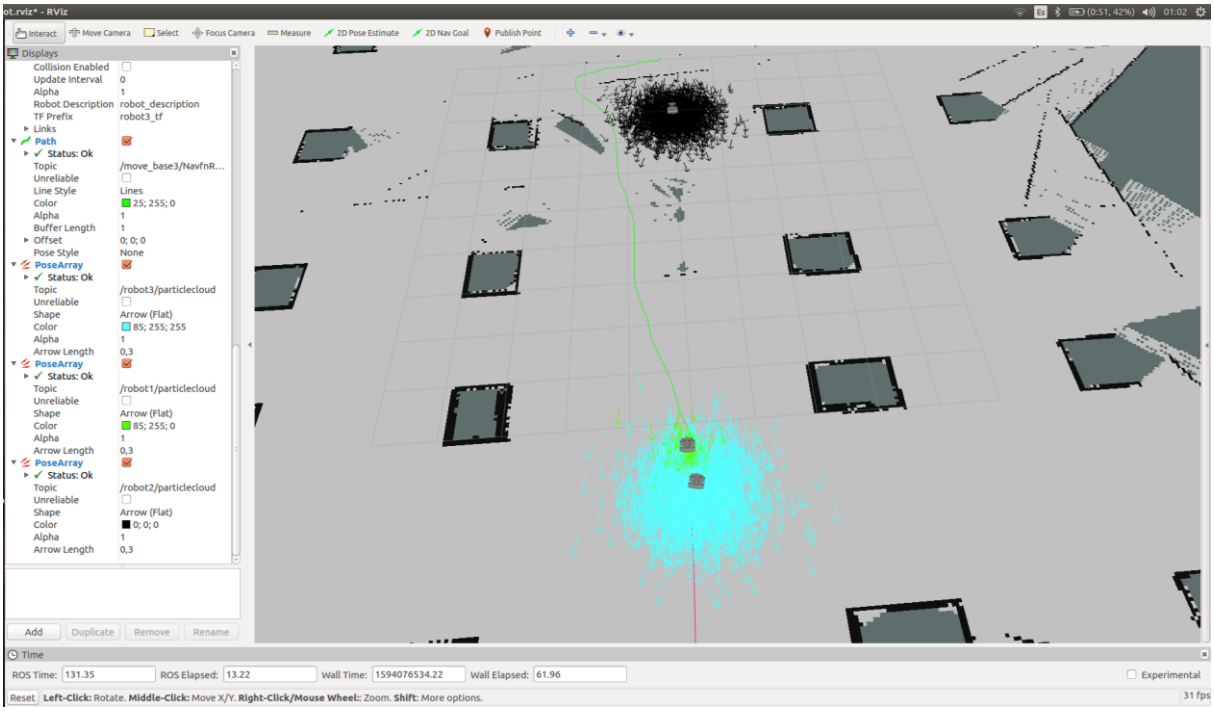

Figura 4.17. Trayectorias generadas por varios robots. Vista superior.

4.4 Posibles colisiones entre robots

El paquete “move_base”, como se ha explicado previamente, permite al robot desplazarse evitando obstáculos tanto estáticos como dinámicos. En las Figuras 4.18 a 4.25 se puede observar el trazado de trayectorias y desplazamiento de los robots, evitándose entre sí.

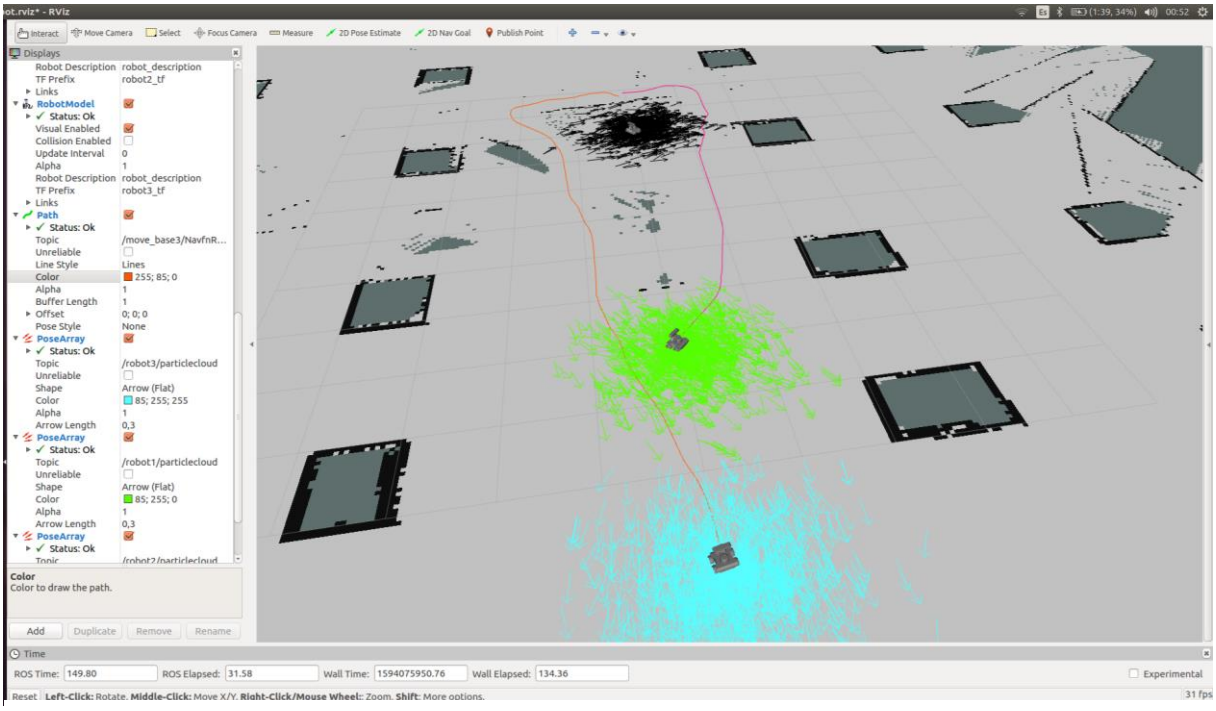

Figura 4.18. Trazado de trayectorias de varios robots. Inicio del movimiento.

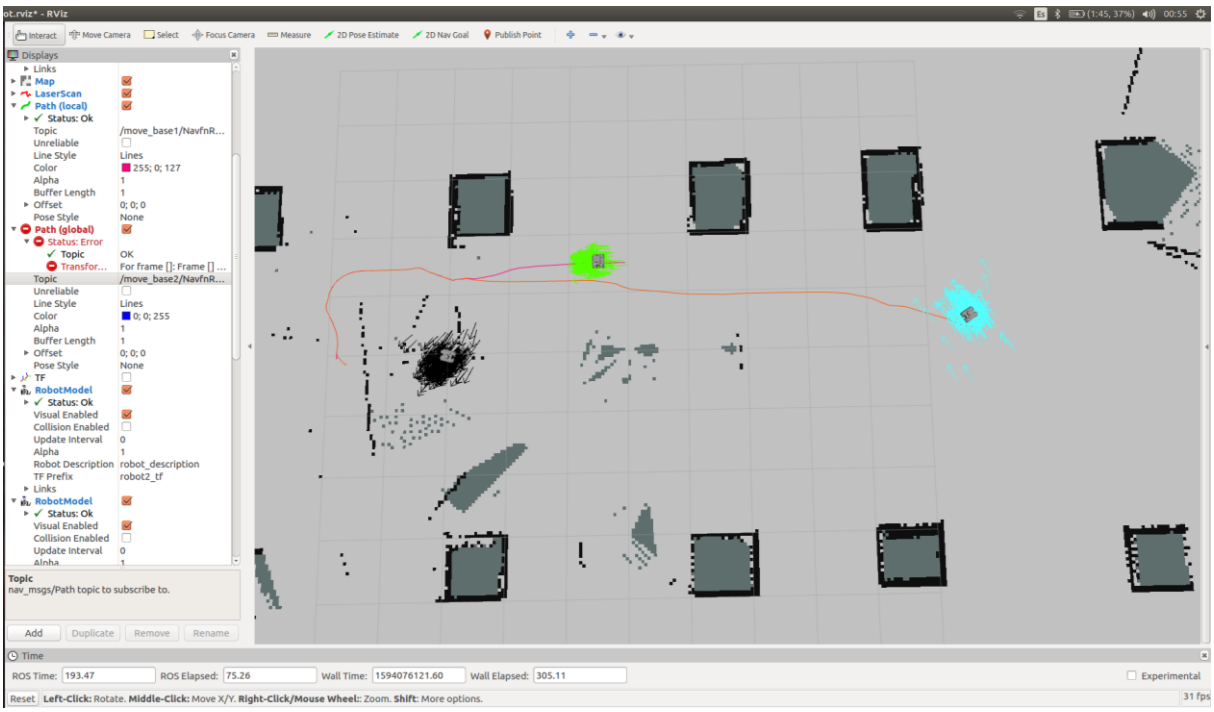

Figura 4.19. Trazado de trayectorias de varios robots. Pequeño desplazamiento.

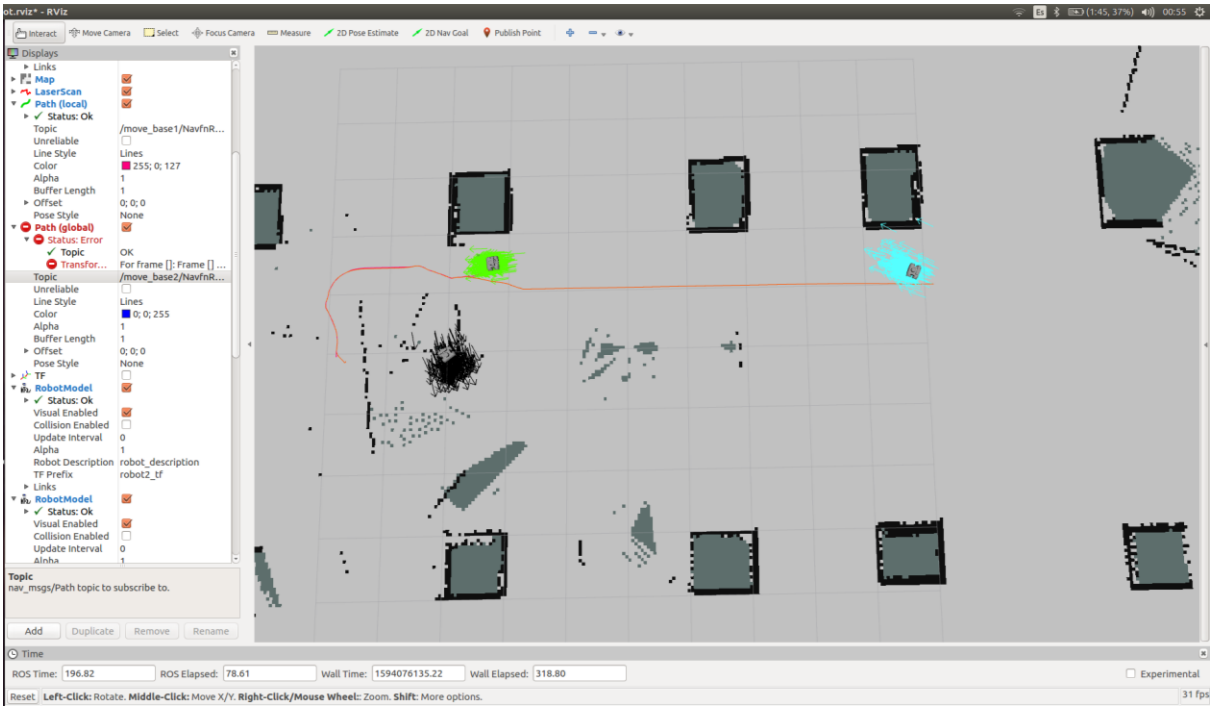

Figura 4.20. Trazado de trayectorias de varios robots. Continuación del desplazamiento.

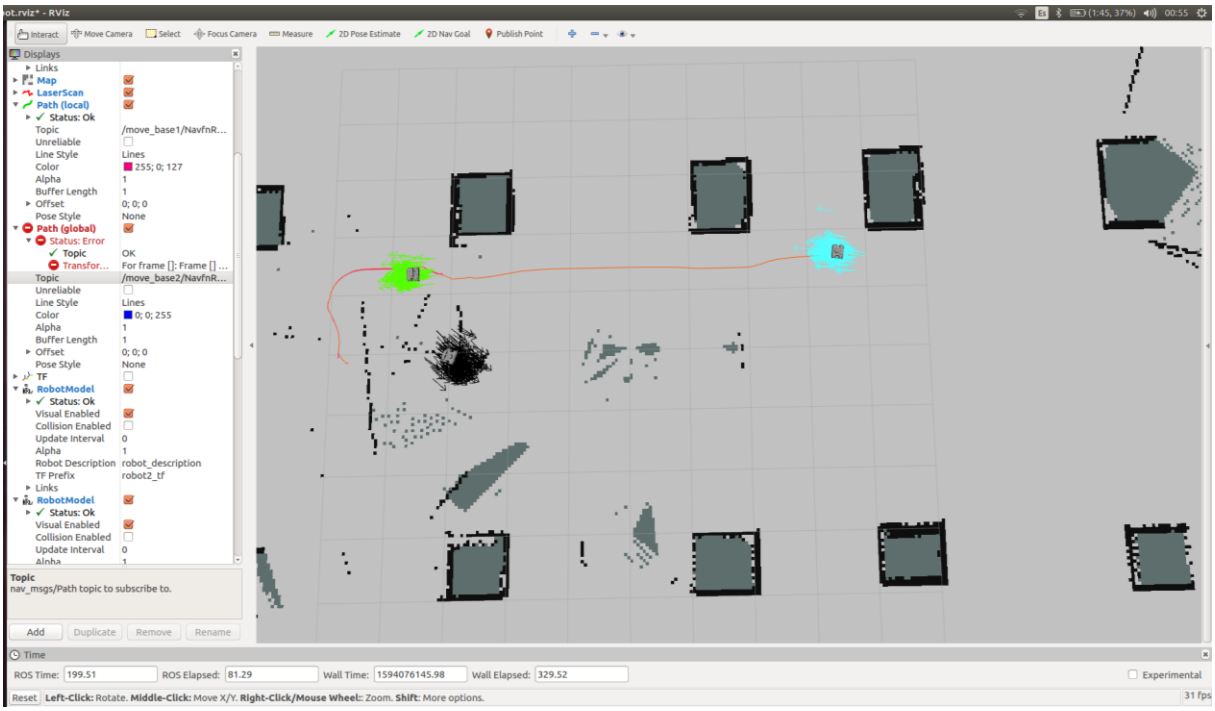

Figura 4.21. Trazado de trayectorias de varios robots. Movimiento dual de los robots.

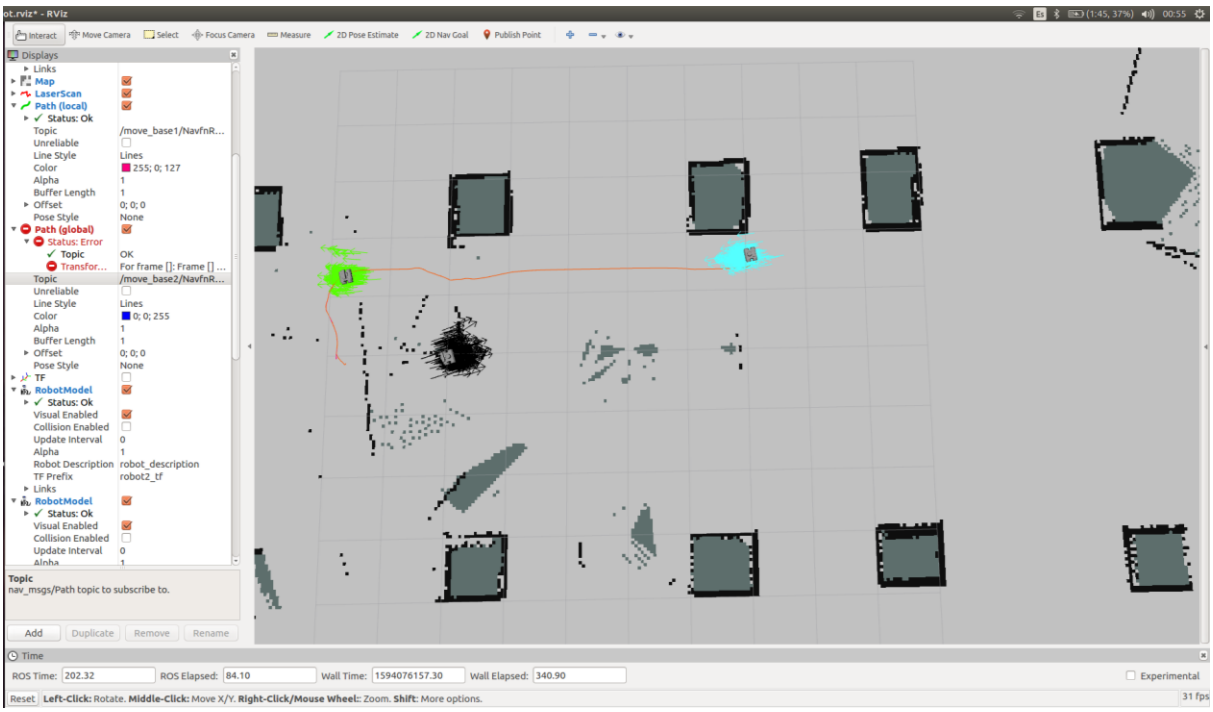

Figura 4.22. Trazado de trayectorias de varios robots. Últimos desplazamientos del “robot1”.

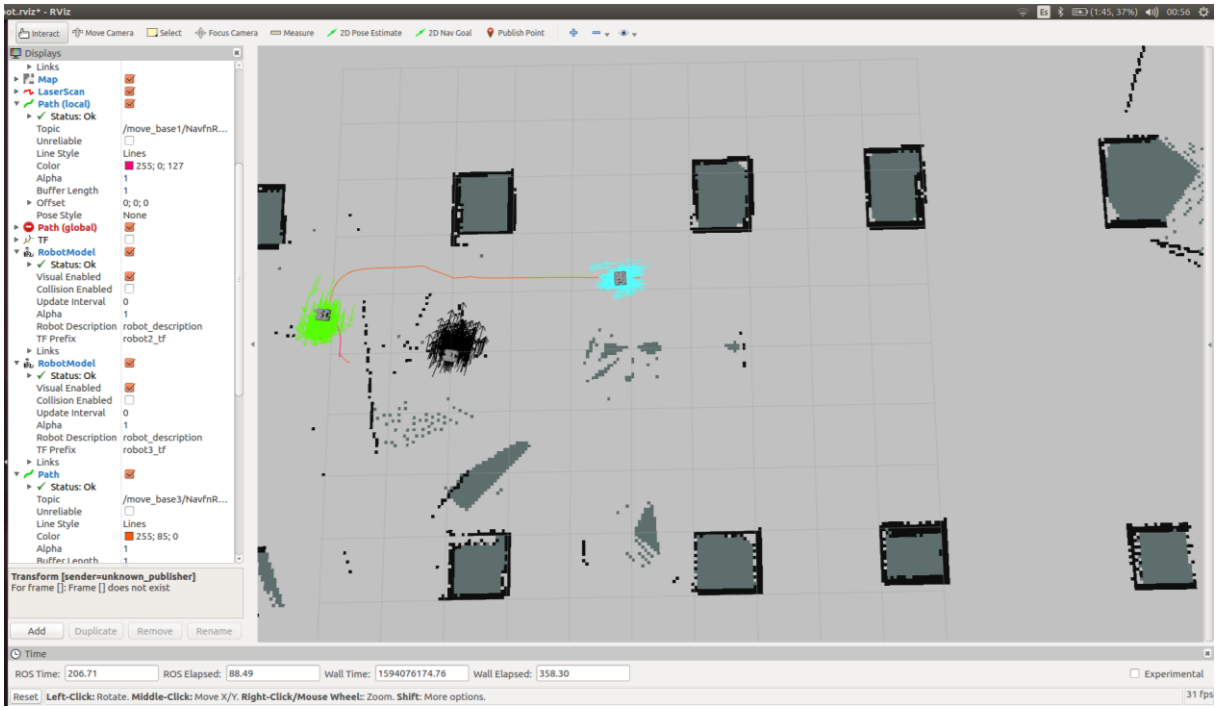

Figura 4.23. Trazado de trayectorias de varios robots. Acercamiento del “robot2”

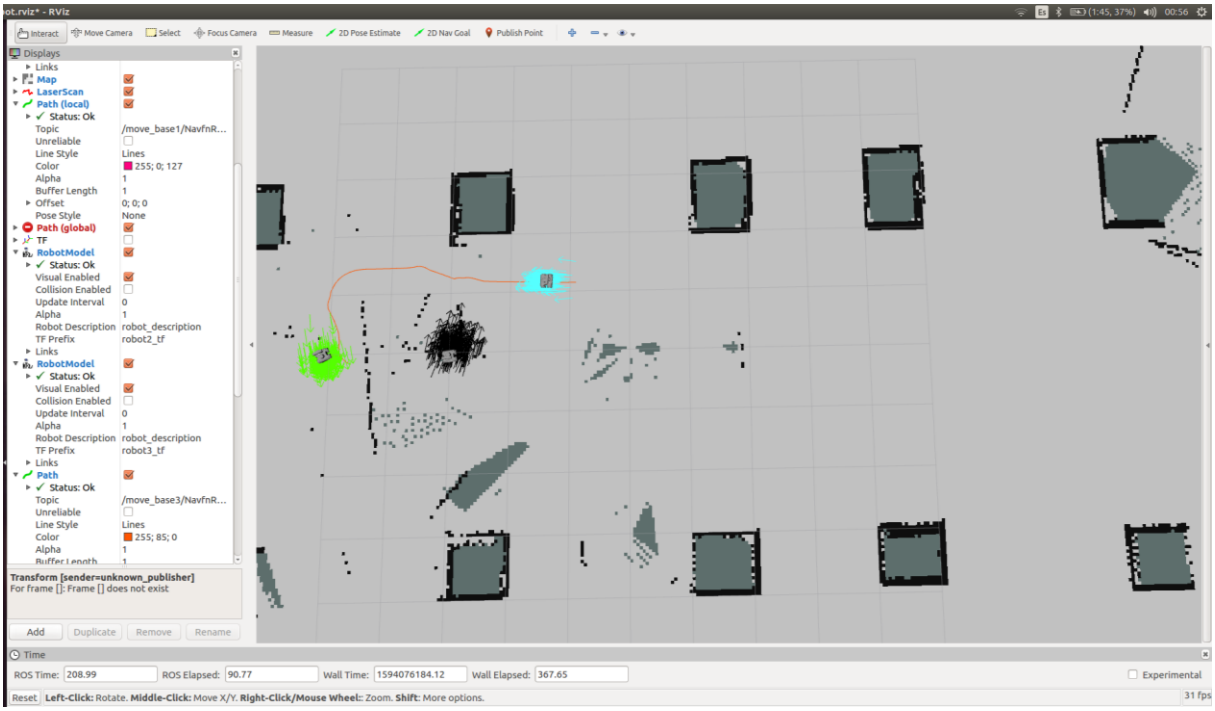

Figura 4.24. Trazado de trayectorias de varios robots. Llegada del “robot1” al punto de destino.

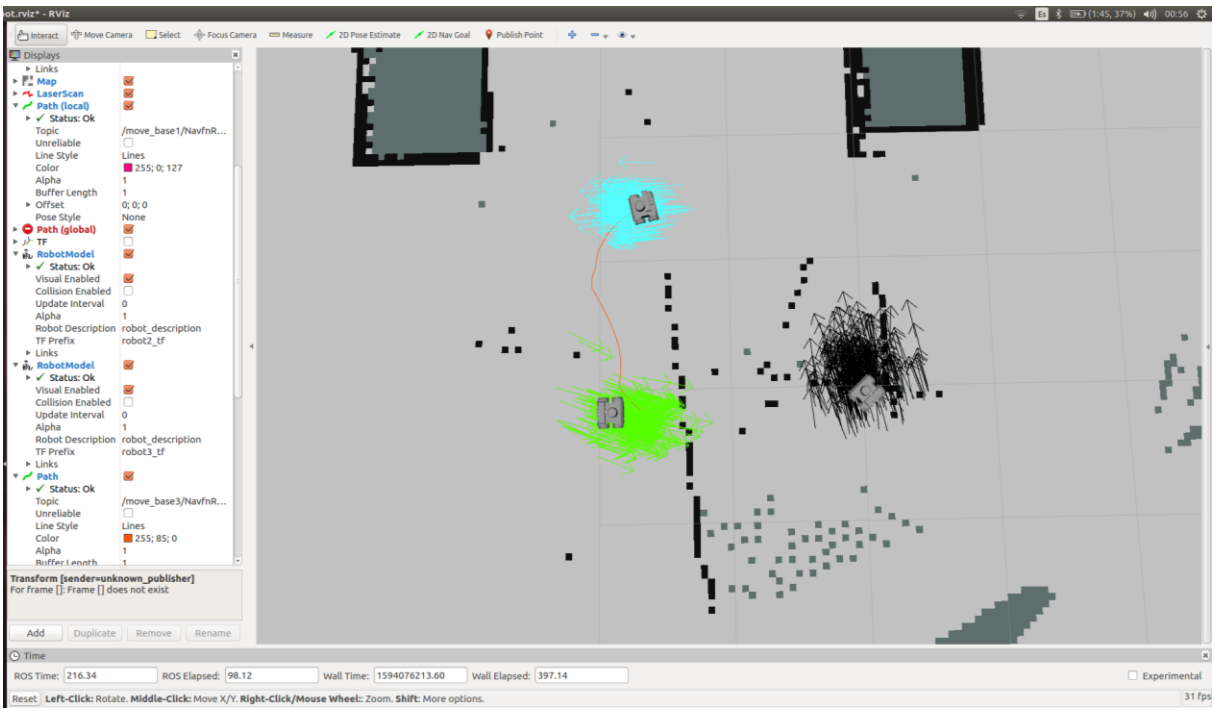

Figura 4.25. Trazado de trayectorias de varios robots. Posición final de ambos robots.

5 CONCLUSIÓN

Existen motivos para ser optimistas con el futuro de la navegación autónoma. Sin embargo, no se puede carecer de realismo cuando se tratan estos temas, debido a lo variable del entorno (el mundo se encuentra en constante cambio) y la alta carga computacional que supone el análisis de lo que rodea al robot, así como la comunicación con otros, etc. Son muchas las tareas que un robot debe llevar a cabo, y el secreto reside en un balance equilibrado entre todas. No obstante, el incesante avance de la tecnología permite soñar al respecto, y quién sabe, quizá sea posible presenciar un futuro donde los robots completamente autónomos no sean solo material de ciencia ficción.

En cuanto a ROS, no es fácil de comprender y manejar, con una curva de aprendizaje lenta. A pesar de ello, una vez los conceptos comienzan a afianzarse, se trata de un sistema muy potente, con posibilidades verdaderamente impactantes como lo puede ser la dualidad en lenguajes de programación para C++ y Python. De hecho, en este proyecto se han creado dos códigos con diferente funcionalidad, escritos en diferentes lenguajes, para así también probar dicha característica, que ofrece ventajas al programador y además amplía el público objetivo y con ello su comunidad, facilitando el trabajo en multitud de ocasiones al usuario.

En lo que a este proyecto respecta, el comportamiento del robot tanto para la localización, análisis del mapeado como para la navegación autónoma es adecuado. Sin embargo, algunas veces el robot se bloquea y comienza su giro de recuperación, el cual se mantiene en bucle. Además, cuando se ha realizado la navegación autónoma de varios robots, la ejecución del simulador Gazebo se ha visto lastrada en multitud de ocasiones. A pesar de ello, los robots alcanzan sus puntos de destino con éxito, y quizá no sea descabellado pensar que gran parte de los errores se debe a que el portátil con el que se ha llevado a cabo este proyecto no es lo suficientemente potente ni en procesador (Intel i3 7th Gen) ni en tarjeta gráfica (Intel Graphics HD). A pesar de dichos impedimentos, el trabajo se ha podido realizar de manera cómoda y los robots presentan el comportamiento esperado, a pesar de funcionar con lentitud en los apartados de mayor requerimiento en cuanto a carga computacional se refiere.

Sin embargo, como se mencionaba anteriormente, se ha de ser realista con los resultados obtenidos. En la realidad, el suelo no es plano, existen múltiples obstáculos que no se pueden simular con unos simples cubos, además de condiciones meteorológicas que pueden lastrar la navegación del robot, así como aspectos cruciales para el funcionamiento de este como lo puede ser una correcta sincronización del GPS y un estudio constante del estado de la batería. Por lo tanto, a pesar de lo positivo de lo conseguido, aplicar dichos algoritmos a un robot real conllevaría una gran cantidad de problemas que solucionar para poder conseguir el funcionamiento deseado.

5.1 Trabajos futuros

Como bien se ha indicado, este proyecto abarca tan sólo la simulación de modelo del robot. Como es bien sabido, trabajar con modelos no es más que una aproximación, en muchos casos bastante distante de la realidad debido a la enorme cantidad de suposiciones que se toman para probarla. Por lo tanto, como trabajos futuros quedaría aplicar dichos algoritmos de creación de mapas, localización y navegación en el robot real de Husarion. Para ello, además, se debería llevar un seguimiento del estado de la batería de los robots para llevarlos a la base y cargarlos, a ser posible de manera automática, implementar el GPS para permitir una navegación lo más precisa posible, así como contemplar el funcionamiento del robot en terrenos que disten de lo plano que es el que se usa en las simulaciones.

De esta manera, se compementaría y completaría el estudio iniciado en este proyecto, dando lugar a soluciones reales en entornos problemáticos, dotando así de mayor veracidad a un proyecto ya de por sí complejo.

REFERENCIAS

[1] Wikipedia, «R.U.R. (Robots Universales Rossum),» Marzo 2020. [En línea]. Available: https://es.wikipedia.org/wiki/R.U.R._(Robots_Universales_Rossum). [Último acceso: Junio 2020].

[2] Wikipedia, «Isaac Asimov,» 2020. [En línea]. Available: https://es.wikipedia.org/wiki/Isaac_Asimov. [Último acceso: Junio 2020].

[3] «Wikipedia (Robots autónomos),» 2020. [En línea]. Available: https://es.wikipedia.org/wiki/Robot_aut%C3%B3nomo. [Último acceso: Junio 2020].

[4] Boston Dynamics, «Boston Dynamics,» 2020. [En línea]. Available: https://www.bostondynamics.com/. [Último acceso: Julio 2020].

[5] Tesla, Inc., «Tesla,» 2020. [En línea]. Available: https://www.tesla.com/es_es. [Último acceso: Julio 2020].

[6] L. J. M. Paniagua, Localización robots móviles de recursos limitados basada en fusión sensorial por eventos, Valencia: Universitat Politècnica de València, 2014.

[7] J. M. A. Moreno, Localización geométrica de robots móviles autónomos, Leganés: Universidad Carlos III de Madrid, 1997.

[8] Keyence, «Keyence,» 2020. [En línea]. Available: https://www.keyence.com.mx/ss/products/sensor/sensorbasics/ultrasonic/info/. [Último acceso: Junio 2020].

[9] A. Chatterkee, Vision Based Autonomous Robot Navigation, Nueva York: Springer, 2013.

[10] K. Gao, J. Xin, H. Cheng, D. Liu y J. Li, «Multi-mobile Robot Autonomous Navigation System for Intelligent Logistics,» de *Multi-mobile Robot Autonomous Navigation System for Intelligent Logistics*, Xi'an, 2018.

[11] «Wikipedia (Robot Operating System (ROS)),» 2020. [En línea]. Available: https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico#Herramientas. [Último acceso: Junio 2020].

[12] «ROS.org (Tools),» 2019. [En línea]. Available: , http://wiki.ros.org/Tools. [Último acceso: Junio 2020].

[13] The Construct, *ROS Developers LIVE-Class #49: How to Map & Localize a Robot (ROS),* EEUU: YouTube, 2019.

[14] Ł. Mitka, «Husarion (SLAM Navigation),» 2020. [En línea]. Available: https://husarion.com/tutorials/ros-tutorials/6-slam-navigation/. [Último acceso: Junio 2020].

[15] EwingKang, «wiki.ROS.org (AMCL),» 27 Junio 2019. [En línea]. Available: http://wiki.ros.org/amcl. [Último acceso: Junio 2020].

[16] «move_base,» 2019. [En línea]. Available: http://wiki.ros.org/move_base. [Último acceso: Junio 2020].

[17] The Construct, *ROS Developers Live-Class #50: Autonomous Robot Navigation with ROS,* EEUU: YouTube, 2019.

[18] I. Saito, «wiki.ROS.org (actionlib),» 2018. [En línea]. Available: http://wiki.ros.org/actionlib. [Último acceso: Junio 2020].

[19] AnisKoubaa, «wiki.ROS.org (Sending Simple Goals),» 25 Agosto 2016. [En línea]. Available: http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals. [Último acceso: Junio 2020].

[20] «Github (Ublox),» [En línea]. Available: https://github.com/KumarRobotics/ublox. [Último acceso: Junio 2020].

[21] «wiki.ros.org/geonav_transform,» 2019. [En línea]. Available: http://wiki.ros.org/geonav_transform. [Último acceso: Junio 2020].

[22] B. Bingham, «docs.ros.org (Geonav_transform),» 2017. [En línea]. Available: http://docs.ros.org/kinetic/api/geonav_transform/html/geonav_transform.html. [Último acceso: Junio 2020].

[23] "Jakub", «wiki.ROS.org (Multiple robots simulation and navigation),» 15 Agosto 2012. [En línea]. Available: https://answers.ros.org/question/41433/multiple-robots-simulation-and-navigation/. [Último acceso: Junio 2020].

[24] The Construct , «YouTube ([ROS Q&A] 130 - How to launch multiple robots in Gazebo simulator?),» 18 Junio 2018. [En línea]. [Último acceso: Junio 2020].

[25] «wiki.ROS.org,» 2016. [En línea]. Available: http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals. [Último acceso: Junio 2020].

[26] «wiki.ROS.org/gmapping,» 2019. [En línea]. Available: http://wiki.ros.org/gmapping. [Último acceso: Junio 2020].

[27] J. Huang, *ROS tutorial #2.1: C++ walkthrough of publisher / subscriber lab,* YouTube, 2016.

[28] «Pc Componentes,» 2020. [En línea]. Available: https://www.pccomponentes.com/irobot-roomba-604-robot-aspirador?utm_campaign=afiliados&utm_source=effi-1395035814. [Último acceso: Junio 2020].

[29] D. Kucher, «Somag News,» Febrero 2020. [En línea]. Available: https://www.somagnews.com/film-like-story-first-real-robot-unimate-history/. [Último acceso: Junio 2020].

[30] «Hipertextual,» 2020. [En línea]. Available: https://hipertextual.com/2019/11/tesla-cybertruck-todo-que-sabemos -> foto (Cybertruck). [Último acceso: Junio 2020].

[31] The Construct, «YouTube (ROS Developers LIVE-Class #64: Multiple Robots Navigation in Gazebo),» 3 Septiembre 2019. [En línea]. Available: https://www.youtube.com/watch?v=es_rQmlgndQ. [Último acceso: Junio 2020].

[32] Google, «Google Maps,» [En línea]. Available: https://www.google.com/maps. [Último acceso: Julio 2020].

ANEXO

Anexo A. Código "rosbot.launch"

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

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

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

</launch>
```

Anexo B. Código "rosbot_gazebo.launch"

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

  <rosparam command="load" file="$(find
joint_state_controller)/joint_state_controller.yaml" />

  <node name="joint_state_controller_spawner" pkg="controller_manager"
type="spawner" output="screen" args="joint_state_controller" />

  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
rosbot_description)/urdf/rosbot.xacro'"/>

  <node name="rosbot_spawn" pkg="gazebo_ros" type="spawn_model"
output="screen" args="-urdf -param robot_description -model rosbot" />

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher"/>

</launch>
```

Anexo C. Código "rosbot_world.launch"

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

  <arg name="world" default="empty"/>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find rosbot_gazebo)/worlds/cubes.world"/>

    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
```

```
    <arg name="headless" value="$(arg headless)"/>
    <arg name="debug" value="$(arg debug)"/>
  </include>

</launch>
```

Anexo D. Código “my_mapping.launch”

```
<launch>
  <arg name="scan_topic"  default="scan" />
  <arg name="base_frame"  default="base_link"/>
  <arg name="odom_frame"  default="odom"/>

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
    <param name="base_frame" value="$(arg base_frame)"/>
    <param name="odom_frame" value="$(arg odom_frame)"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="200"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>
  <!--
    <param name="xmin" value="-50.0"/>
    <param name="ymin" value="-50.0"/>
    <param name="xmax" value="50.0"/>
    <param name="ymax" value="50.0"/>
  make the starting size small for the benefit of the Android client's
memory...
  -->
    <param name="xmin" value="-1.0"/>
    <param name="ymin" value="-1.0"/>
    <param name="xmax" value="1.0"/>
    <param name="ymax" value="1.0"/>
    <param name="delta" value="0.05"/>
    <param name="llsamplerange" value="0.01"/>
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>
    <remap from="scan" to="$(arg scan_topic)"/>
```

```
  </node>
</launch>
```

Anexo E. Código "my_localization_launcher"

```
<launch>
  <arg name="use_map_topic"   default="false"/>
  <arg name="scan_topic"      default="scan"/>
  <arg name="initial_pose_x"  default="0.0"/>
  <arg name="initial_pose_y"  default="0.0"/>
  <arg name="initial_pose_a"  default="0.0"/>
  <arg name="odom_frame_id"   default="odom"/>
  <arg name="base_frame_id"   default="base_link"/>
  <arg name="global_frame_id" default="map"/>
  <node pkg="amcl" type="amcl" name="amcl">
    <param name="use_map_topic"             value="$(arg use_map_topic)"/>
    <!-- Publish scans from best pose at a max of 10 Hz -->
    <param name="odom_model_type"           value="diff"/>
    <param name="odom_alpha5"               value="0.1"/>
    <param name="gui_publish_rate"          value="10.0"/>
    <param name="laser_max_beams"             value="60"/>
    <param name="laser_max_range"           value="12.0"/>
    <param name="min_particles"             value="500"/>
    <param name="max_particles"             value="2000"/>
    <param name="kld_err"                   value="0.05"/>
    <param name="kld_z"                     value="0.99"/>
    <param name="odom_alpha1"               value="0.2"/>
    <param name="odom_alpha2"               value="0.2"/>
    <!-- translation std dev, m -->
    <param name="odom_alpha3"               value="0.2"/>
    <param name="odom_alpha4"               value="0.2"/>
    <param name="laser_z_hit"               value="0.5"/>
    <param name="laser_z_short"             value="0.05"/>
    <param name="laser_z_max"               value="0.05"/>
    <param name="laser_z_rand"              value="0.5"/>
    <param name="laser_sigma_hit"           value="0.2"/>
    <param name="laser_lambda_short"        value="0.1"/>
    <param name="laser_model_type"          value="likelihood_field"/>
    <!-- <param name="laser_model_type" value="beam"/> -->
    <param name="laser_likelihood_max_dist" value="2.0"/>
    <param name="update_min_d"              value="0.25"/>
    <param name="update_min_a"              value="0.2"/>
    <param name="odom_frame_id"             value="$(arg odom_frame_id)"/>
    <param name="base_frame_id"             value="$(arg base_frame_id)"/>
    <param name="global_frame_id"           value="$(arg global_frame_id)"/>
    <param name="resample_interval"         value="1"/>
    <!-- Increase tolerance because the computer can get quite busy -->
    <param name="transform_tolerance"       value="1.0"/>
    <param name="recovery_alpha_slow"       value="0.0"/>
    <param name="recovery_alpha_fast"       value="0.0"/>
    <param name="initial_pose_x"            value="$(arg initial_pose_x)"/>
    <param name="initial_pose_y"            value="$(arg initial_pose_y)"/>
    <param name="initial_pose_a"            value="$(arg initial_pose_a)"/>
    <remap from="scan"                      to="$(arg scan_topic)"/>
  </node>
```

```
</launch>
```

Anexo F. Código "my_move_base.launch"

```
<!--ROS navigation stack with velocity smoother and safety (reactive)
controller-->
<launch>
  <include file="$(find
turtlebot_navigation)/launch/includes/velocity_smoother.launch.xml"/>
  <include file="$(find
turtlebot_navigation)/launch/includes/safety_controller.launch.xml"/>
  <arg name="odom_frame_id"   default="odom"/>
  <arg name="base_frame_id"   default="base_link"/>
  <arg name="global_frame_id" default="map"/>
  <arg name="odom_topic" default="odom" />
  <arg name="laser_topic" default="scan" />
  <arg name="custom_param_file" default="$(find
turtlebot_navigation)/param/dummy.yaml"/>
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find
turtlebot_navigation)/param/costmap_common_params.yaml" command="load"
ns="global_costmap" />
    <rosparam file="$(find
turtlebot_navigation)/param/costmap_common_params.yaml" command="load"
ns="local_costmap" />
    <rosparam file="$(find
turtlebot_navigation)/param/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/dwa_local_planner_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/move_base_params.yaml"
command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/global_planner_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/navfn_global_planner_params.yaml" command="load"
/>
    <!-- external params file that could be loaded into the move_base
namespace -->
    <rosparam file="$(arg custom_param_file)" command="load" />
    <!-- reset frame_id parameters using user input data -->
    <param name="global_costmap/global_frame" value="$(arg
global_frame_id)"/>
    <param name="global_costmap/robot_base_frame" value="$(arg
base_frame_id)"/>
    <param name="local_costmap/global_frame" value="$(arg odom_frame_id)"/>
    <param name="local_costmap/robot_base_frame" value="$(arg
base_frame_id)"/>
    <param name="DWAPlannerROS/global_frame_id" value="$(arg
odom_frame_id)"/>
    <remap from="cmd_vel" to="cmd_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <remap from="scan" to="$(arg laser_topic)"/>
  </node>
```

```
</launch>
```

Anexo G. Código "simple_navigation_goals.cpp"

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include "geometry_msgs/PoseStamped.h"

#include <vector>
#include <string>


typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

using std::vector;
using std::string;

//Se definen los valores de medio lado del cubo así como de la distancia del
eje del robot a sus laterales.

float half_cube=0.25;
float robot_width=0.114;
std::string string_cube;

//Definición de una class (como una estructura). Se usa al llamar a la
función initLandmarks.
class Landmark {
  public:
    Landmark(string name, double x, double y)
    : name(name), x(x), y(y) {}
    string name;
    double x;
    double y;
};

//Función CheckPosition

std::string CheckPosition(float x, float y, std::vector<Landmark> cubes){
  int a; int b; int error=0;
  int i;
  std::string string;

  if (x<0 && y<0){
    a=0; b=4;
  } else if (x<0 && y>0){
    a=4; b=8;
  } else if (x>0 && y<0){
    a=8; b=14;
  } else if (x>0 && y<0){
    a=14;
    b=20;
  }
```

```
  for(i=a; i<b; i++){

    const Landmark& cube_value = cubes[i];
    if (((x>(cube_value.x-half_cube-robot_width) &&
x<(cube_value.x+half_cube+robot_width)) && (y>(cube_value.y-half_cube-
robot_width) && y<(cube_value.y+half_cube+robot_width) ))){
      error=1;
      string=cube_value.name;}
    if (error!=0){
      break;}
  }

  if (error==0){
    string = "None";
  }
  return string;
};

//Función principal

int main(int argc, char** argv){
  ros::init(argc, argv, "simple_navigation_goals");

  //tell the action client that we want to spin a thread by default
  MoveBaseClient ac("move_base", true);

  //wait for the action server to come up
  while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server to come up");
  }
  ros::NodeHandle nh("~");
  //Declaration of variables which are received from user.
  std::string frame_id;
  double seq, secs, nsecs;
  float pose_x, pose_y, pose_z, orientation_x, orientation_y, orientation_z,
orientation_w;
  nh.getParam("seq", seq);
  ros::Time stamp(nh.getParam("secs", secs),     nh.getParam("nsecs",nsecs));
  nh.getParam("frame_id", frame_id);
  nh.getParam("pose_x", pose_x);
  nh.getParam("pose_y", pose_y);
  nh.getParam("pose_z", pose_z);
  nh.getParam("orientation_x", orientation_x);
  nh.getParam("orientation_y", orientation_y);
  nh.getParam("orientation_z", orientation_z);
  nh.getParam("orientation_w", orientation_w);

  move_base_msgs::MoveBaseGoal goal;

//Los valores de la posición a publicar son los recibidos como argumentos.

  //we'll send a goal to the robot to move 1 meter forward
  goal.target_pose.header.frame_id = frame_id;
  goal.target_pose.header.seq = seq;
  goal.target_pose.header.stamp = ros::Time::now();
```

```
  goal.target_pose.pose.position.x = pose_x;
  goal.target_pose.pose.position.y = pose_y;
  goal.target_pose.pose.position.z = pose_z;
  goal.target_pose.pose.orientation.x = orientation_x;
  goal.target_pose.pose.orientation.y = orientation_y;
  goal.target_pose.pose.orientation.z = orientation_z;
  goal.target_pose.pose.orientation.w = orientation_w;

  ROS_INFO("La posición a la que se quiere enviar el robot es: seq=%f,\n
secs=%f,\n nsecs=%f,\n frame_id=%s,\n pose_x=%f,\n pose_y=%f,\n pose_z=%f,\n
orientation_x=%f,\n orientation_y=%f,\n orientation_z=%f,\n
orientation_w=%f\n", seq, secs, nsecs, frame_id.c_str(), pose_x, pose_y,
pose_z, orientation_x, orientation_y, orientation_z, orientation_w);

//Definición de las posiciones de los cubos. (el vector cubes es del tipo
Landmark (name, x, y)).

  vector<Landmark> cubes;
  cubes.push_back(Landmark("Cube1",-6.474805,-7.465445));
  cubes.push_back(Landmark("Cube2",-6.517705,-3.470995));
  cubes.push_back(Landmark("Cube3",-2.502295,-7.485365));
  cubes.push_back(Landmark("Cube4",-2.509755,-3.473655));
  cubes.push_back(Landmark("Cube5",-6.485225,2.518645));
  cubes.push_back(Landmark("Cube6",-6.463125,6.494955));
  cubes.push_back(Landmark("Cube7",-2.563945,2.492495));
  cubes.push_back(Landmark("Cube8",2.492495,6.479435));
  cubes.push_back(Landmark("Cube9",1.489375,-7.537095));
  cubes.push_back(Landmark("Cube10",1.510725,-3.536545));
  cubes.push_back(Landmark("Cube11",4.508595,-7.527855));
  cubes.push_back(Landmark("Cube12",4.508305,-3.463135));
  cubes.push_back(Landmark("Cube13",8.555935,-7.474625));
  cubes.push_back(Landmark("Cube14",8.496135,-3.586645));
  cubes.push_back(Landmark("Cube15",1.518545,2.476055));
  cubes.push_back(Landmark("Cube16",1.617195,6.471065));
  cubes.push_back(Landmark("Cube17",4.493005,2.501285));
  cubes.push_back(Landmark("Cube18",4.538085,6.484765));
  cubes.push_back(Landmark("Cube19",8.549455,2.441395));
  cubes.push_back(Landmark("Cube20",8.592165,6.536445));

//La función CheckPosition devolverá una cadena de caracteres, que será
“none” en el caso de que el punto sea válido, o CubeX en el caso de que
corresponda con la posición de un cubo.

  string_cube=CheckPosition(pose_x, pose_y,cubes);
  if (string_cube!="None") {
  ROS_INFO("La posición a la que se quiere enviar el robot es errónea, está
ocupada por el %s.",string_cube.c_str());
  ros::shutdown();
  } else {
  ROS_INFO("La posición a la que se quiere enviar el robot es correcta (topic
published!)");
  ac.sendGoal(goal);
  }
```

```
  ac.waitForResult();

//Se envía un mensaje si el robot ha alcanzado su objetivo o si no ha podido
logarlo.

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO("Hooray, the base moved to the point");
  else
    ROS_INFO("The base failed to move to the point for some reason");

  return 0;
}
```

Anexo H. Código "gps_goal_generator.py"

```
#!/usr/bin/env python

import rospy
import sys
import geonav_transform.geonav_conversions as gc
reload(gc)
import alvinxy.alvinxy as axy

reload(axy)
from move_base_msgs.msg import MoveBaseGoal
from std_msgs.msg import Float32MultiArray

oLon_Degrees=37
oLon_Minutes=24
oLon_Seconds=41.7

oLat_Degrees=5
oLat_Minutes=59
oLat_Seconds=48.1

# Define local origin at lower right of UTM zone
olon=oLon_Degrees + oLon_Minutes/60 + oLon_Seconds/3600
olat=oLat_Degrees + oLat_Minutes/60 + oLat_Seconds/3600

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s",
data.target_pose.pose.position.x)
    x = data.target_pose.pose.position.x
    y = data.target_pose.pose.position.y
    lat_goal, lon_goal = gc.xy2ll(x,y,olat,olon)

    print "El origen de coordenadas es : ", olat, olon
    print "la distancia a recorrer es (x,y)= (", x, y, ") metres"
    print "Las coordenadas GPS del punto destino son: ", lat_goal, lon_goal

    Float32MultiArray.data[0]=lat_goal
    Float32MultiArray.data[1]=lon_goal
    pub.publish(Float32MultiArray)

    pub.publish()
def goal_generator():
```

```
    rospy.init_node('goal_generator', anonymous=True)
    rospy.Subscriber("move_base/goal", MoveBaseGoal, callback)
    pub = rospy.Publisher('Goal_GPS_coordinates', Float32MultiArray,
queue_size=10)
    rate = rospy.Rate(10) # 10hz
    #rospy.get_param('color')
    #print "El color elegido es", color

if __name__ == '__main__':
    try:
        goal_generator()
    except rospy.ROSInterruptException:
        pass
```

Anexo I. Código "one_robot.launch"

```
<launch>
    <arg name="robot_name"/>
    <arg name="init_pose"/>
    <rosparam command="load" file="$(find
joint_state_controller)/joint_state_controller.yaml" />
    <node name="joint_state_controller_spawner" pkg="controller_manager"
type="spawner" output="screen" args="joint_state_controller" />
    <node name="spawn_minibot_model" pkg="gazebo_ros" type="spawn_model"
     args="$(arg init_pose) -urdf -param /robot_description -model $(arg
robot_name)"
     respawn="false" output="screen" />
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen"/>
</launch>
```

Anexo J. Código "robots.launch"

```
<launch>
       <!-- No namespace here as we will share this description.
          Access with slash at the beginning -->
     <param name="robot_description" command="$(find xacro)/xacro.py $(find
   rosbot_description)/urdf/rosbot.xacro"/>
       <group ns="robot1">
         <param name="tf_prefix" value="robot1_tf" />
         <include file="$(find multi_robot)/launch/one_robot.launch" >
           <arg name="init_pose" value="-x 3.0 -y 0.0 -z 0.0" />
           <arg name="robot_name"  value="robot1"/>
         </include>
       </group>
       <group ns="robot2">
         <param name="tf_prefix" value="robot2_tf" />
         <include file="$(find multi_robot)/launch/one_robot.launch" >
           <arg name="init_pose" value="-x -3.0 -y 0.0 -z 0.0" />
           <arg name="robot_name"  value="robot2"/>
         </include>
       </group>
       <group ns="robot3">
         <param name="tf_prefix" value="robot3_tf" />
```

```
        <include file="$(find multi_robot)/launch/one_robot.launch" >
          <arg name="init_pose" value="-x 6.0 -y 0.0 -z 0.0" />
          <arg name="robot_name"  value="robot3"/>
        </include>
      </group>
    </launch>
```

Anexo K. Código "main.launch"

```
<launch>
  <param name="/use_sim_time" value="true" />
  <!-- start world -->
  <node name="gazebo" pkg="gazebo_ros" type="gazebo"
   args="$(find rosbot_gazebo)/worlds/cubes.world" respawn="false"
output="screen" />
  <!-- include our robots -->
  <include file="$(find multi_robot)/launch/robots.launch"/>
  <include file="$(find
multi_robots_nav)/launch/multi_robots_nav.launch"></include>
</launch>
```

Anexo L. Código "multi_robots_nav.launch"

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <param name="/use_sim_time" value="true" />
  <arg name="map_file" default="$(find
my_mapping_launcher)/config/cubes_map.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)"/>
  <include file="$(find multi_robots_nav)/launch/amcl_robot1.launch"></include>
  <include file="$(find multi_robots_nav)/launch/amcl_robot2.launch"></include>
  <include file="$(find multi_robots_nav)/launch/amcl_robot3.launch"></include>

 <include file="$(find multi_robots_nav)/launch/move_base1.launch"></include>
  <include file="$(find multi_robots_nav)/launch/move_base2.launch"></include>
  <include file="$(find multi_robots_nav)/launch/move_base3.launch"></include>

 <node pkg="rviz" type="rviz" name="rviz" output="screen" args="-d $(find
rosbot_description)/rviz/rosbot.rviz"/>
</launch>
```

Anexo M. Código "amcl_robot1.launch"

```
<launch>
<arg name="use_map_topic" default="false"/>
<arg name="scan_topic" default="/robot1/scan"/>
<arg name="initial_pose_x" default="3.0"/>
<arg name="initial_pose_y" default="0.0"/>
<arg name="initial_pose_a" default="0.0"/>
<arg name="odom_frame_id" default="/robot1_tf/odom"/>
<arg name="base_frame_id" default="/robot1_tf/base_link"/>
<arg name="global_frame_id" default="map"/>
<node pkg="amcl" type="amcl" name="robot1_amcl" output="screen">
```

```
  <param name="use_map_topic" value="$(arg use_map_topic)"/>
  <!-- Publish scans from best pose at a max of 10 Hz -->
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0.1"/>
  <param name="transform_tolerance" value="0.2" />
  <param name="gui_publish_rate" value="10.0"/>
  <param name="laser_max_beams" value="30"/>
  <param name="min_particles" value="500"/>
  <param name="max_particles" value="5000"/>
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="odom_alpha1" value="0.2"/>
  <param name="odom_alpha2" value="0.2"/>
  <!-- translation std dev, m -->
  <param name="odom_alpha3" value="0.8"/>
  <param name="odom_alpha4" value="0.2"/>
  <param name="laser_z_hit" value="0.5"/>
  <param name="laser_z_short" value="0.05"/>
  <param name="laser_z_max" value="0.05"/>
  <param name="laser_z_rand" value="0.5"/>
  <param name="laser_sigma_hit" value="0.2"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_model_type" value="likelihood_field"/>
  <!-- <param name="laser_model_type" value="beam"/> -->
  <param name="laser_likelihood_max_dist" value="2.0"/>
  <param name="update_min_d" value="0.2"/>
  <param name="update_min_a" value="0.5"/>
  <param name="odom_frame_id" value="$(arg odom_frame_id)"/>
  <param name="base_frame_id" value="$(arg base_frame_id)"/>
  <param name="global_frame_id" value="$(arg global_frame_id)"/>
  <param name="resample_interval" value="1"/>
  <param name="transform_tolerance" value="0.1"/>
  <param name="recovery_alpha_slow" value="0.0"/>
  <param name="recovery_alpha_fast" value="0.0"/>
  <param name="initial_pose_x" value="$(arg initial_pose_x)"/>
  <param name="initial_pose_y" value="$(arg initial_pose_y)"/>
  <param name="initial_pose_a" value="$(arg initial_pose_a)"/>
  <remap from="scan" to="$(arg scan_topic)" />
  <remap from="initialpose" to="/robot1/initialpose" />
  <remap from="amcl_pose" to="/robot1/amcl_pose" />
  <remap from="particlecloud" to="/robot1/particlecloud" />
</node>
</launch>
```

Anexo N. Código "move_base1.launch"

```
<!--ROS navigation stack with velocity smoother and safety (reactive)
controller-->
<launch>
  <include file="$(find
turtlebot_navigation)/launch/includes/velocity_smoother.launch.xml"/>
  <include file="$(find
turtlebot_navigation)/launch/includes/safety_controller.launch.xml"/>
```

```
  <arg name="odom_frame_id"   default="robot1_tf/odom"/>
  <arg name="base_frame_id"   default="robot1_tf/base_link"/>
  <arg name="global_frame_id" default="map"/>
  <arg name="odom_topic" default="/robot1/odom" />
  <arg name="laser_topic" default="/robot1/scan" />
  <arg name="custom_param_file" default="$(find
turtlebot_navigation)/param/dummy.yaml"/>

  <node pkg="move_base" type="move_base" respawn="false" name="move_base1"
output="screen">
    <rosparam file="$(find
turtlebot_navigation)/param/costmap_common_params.yaml" command="load"
ns="global_costmap" />
    <rosparam file="$(find
turtlebot_navigation)/param/costmap_common_params.yaml" command="load"
ns="local_costmap" />
    <rosparam file="$(find
turtlebot_navigation)/param/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/dwa_local_planner_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/move_base_params.yaml"
command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/global_planner_params.yaml" command="load" />
    <rosparam file="$(find
turtlebot_navigation)/param/navfn_global_planner_params.yaml" command="load" />
    <!-- external params file that could be loaded into the move_base namespace
-->
    <rosparam file="$(arg custom_param_file)" command="load" />
    <!-- reset frame_id parameters using user input data -->
    <param name="global_costmap/global_frame" value="$(arg global_frame_id)"/>
    <param name="global_costmap/robot_base_frame" value="$(arg
base_frame_id)"/>
    <param name="local_costmap/global_frame" value="$(arg odom_frame_id)"/>
    <param name="local_costmap/robot_base_frame" value="$(arg base_frame_id)"/>
    <param name="DWAPlannerROS/global_frame_id" value="$(arg odom_frame_id)"/>
    <remap from="cmd_vel" to="/robot1/cmd_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <remap from="scan" to="$(arg laser_topic)"/>
    <remap from="map" to="/map" />
    <remap from="/move_base_simple/goal" to="/robot1/move_base_simple/goal"/>
    <remap from="/move_base/TebLocalPlannerROS/global_plan"
to="/robot1/move_base/TebLocalPlannerROS/global_plan"/>
    <remap from="/move_base/TebLocalPlannerROS/local_plan"
to="/robot1/move_base/TebLocalPlannerROS/local_plan"/>
    <remap from="/move_base/TebLocalPlannerROS/teb_markers"
to="/robot1/move_base/TebLocalPlannerROS/teb_markers"/>
    <remap from="/move_base/TebLocalPlannerROS/teb_markers_array"
to="/robot1/move_base/TebLocalPlannerROS/teb_markers_array"/>
    <remap from="/move_base/TebLocalPlannerROS/teb_poses"
to="/robot1/move_base/TebLocalPlannerROS/teb_poses"/>
    <remap from="/move_base/global_costmap/costmap"
to="/robot1/move_base/global_costmap/costmap"/>
```

```
    <remap from="/move_base/global_costmap/costmap_updates"
to="/robot1/move_base/global_costmap/costmap_updates"/>
    <remap from="/move_base/local_costmap/costmap"
to="/robot1/move_base/local_costmap/costmap"/>
    <remap from="/move_base/local_costmap/costmap_updates"
to="/robot1/move_base/local_costmap/costmap_updates"/>
    <remap from="/move_base/local_costmap/footprint"
to="/robot1/move_base/local_costmap/footprint"/>
    <remap from="/move_base/GlobalPlanner/parameter_descriptions"
to="/robot1/move_base/GlobalPlanner/parameter_descriptions"/>
    <remap from="/move_base/GlobalPlanner/parameter_updates"
to="/robot1/move_base/GlobalPlanner/parameter_updates"/>
    <remap from="/move_base/GlobalPlanner/plan"
to="/robot1/move_base/GlobalPlanner/plan"/>
    <remap from="/move_base/GlobalPlanner/potential"
to="/robot1/move_base/GlobalPlanner/potential"/>
    <remap from="/move_base/TebLocalPlanner/obstacles"
to="/robot1/move_base/TebLocalPlanner/obstacles"/>
    <remap from="/move_base/TebLocalPlanner/parameter_descriptions"
to="/robot1/move_base/TebLocalPlanner/parameter_descriptions"/>
    <remap from="/move_base/TebLocalPlanner/parameter_updates"
to="/robot1/move_base/TebLocalPlanner/parameter_updates"/>
    <remap from="/move_base/cancel" to="/robot1/move_base/cancel"/>
    <remap from="/move_base/current_goal" to="/robot1/move_base/current_goal"/>
    <remap from="/move_base/feedback" to="/robot1/move_base/feedback"/>
    <remap from="/move_base/global_costmap/footprint"
to="/robot1/move_base/global_costmap/footprint"/>
    <remap
from="/move_base/global_costmap/inflation_layer/parameter_descriptions"
to="/robot1/move_base/global_costmap/inflation_layer/parameter_descriptions"/>
    <remap from="/move_base/global_costmap/inflation_layer/parameter_updates"
to="/robot1/move_base/global_costmap/inflation_layer/parameter_updates"/>
    <remap from="/move_base/global_costmap/obstacle_layer/clearing_endpoints"
to="/robot1/move_base/global_costmap/obstacle_layer/clearing_endpoints"/>
    <remap
from="/move_base/global_costmap/obstacle_layer/parameter_descriptions"
to="/robot1/move_base/global_costmap/obstacle_layer/parameter_descriptions"/>
    <remap from="/move_base/global_costmap/obstacle_layer/parameter_updates"
to="/robot1/move_base/global_costmap/obstacle_layer/parameter_updates"/>
    <remap from="/move_base/global_costmap/parameter_descriptions"
to="/robot1/move_base/global_costmap/parameter_descriptions"/>
    <remap from="/move_base/global_costmap/parameter_updates"
to="/robot1/move_base/global_costmap/parameter_updates"/>
    <remap from="/move_base/global_costmap/static_layer/parameter_descriptions"
to="/robot1/move_base/global_costmap/static_layer/parameter_descriptions"/>
    <remap from="/move_base/global_costmap/static_layer/parameter_updates"
to="/robot1/move_base/global_costmap/static_layer/parameter_updates"/>
    <remap from="/move_base/goal" to="/robot1/move_base/goal"/>
    <remap
from="/move_base/local_costmap/obstacle_layer/parameter_descriptions"
to="/robot1/move_base/local_costmap/obstacle_layer/parameter_descriptions"/>
    <remap from="/move_base/local_costmap/obstacle_layer/parameter_updates"
to="/robot1/move_base/local_costmap/obstacle_layer/parameter_updates"/>
    <remap from="/move_base/local_costmap/parameter_descriptions"
```

```
to="/robot1/move_base/local_costmap/parameter_descriptions"/>
    <remap from="/move_base/local_costmap/parameter_updates"
to="/robot1/move_base/local_costmap/parameter_updates"/>
    <remap from="/move_base/local_costmap/static_layer/parameter_descriptions"
to="/robot1/move_base/local_costmap/static_layer/parameter_descriptions"/>
    <remap from="/move_base/local_costmap/static_layer/parameter_updates"
to="/robot1/move_base/local_costmap/static_layer/parameter_updates"/>
    <remap from="/move_base/parameter_descriptions"
to="/robot1/move_base/parameter_descriptions"/>
    <remap from="/move_base/parameter_updates"
to="/robot1/move_base/parameter_updates"/>
    <remap from="/move_base/us" to="/robot1/move_base/result"/>
    <remap from="/move_base/status" to="/robot1/move_base/status"/>
    <remap from="/move_base_simple/goal" to="/robot1/move_base_simple/goal"/>
  </node>
</launch>
```

Anexo O. Código "three_robots_nav.launch"

```
<?xml version="1.0"?>
<launch>
  <include file="$(find
simple_navigation_goals)/launch/simple_navigation_goals.launch"></include>
  <include file="$(find
simple_navigation_goals)/launch/simple_navigation_goals_2.launch"></include>
  <include file="$(find
simple_navigation_goals)/launch/simple_navigation_goals_3.launch"></include>
</launch>
```